 [**Translated from English to Vietnamese - www.onlinedoctranslator.com**](https://www.onlinedoctranslator.com/en/?utm_source=onlinedoctranslator&utm_medium=pdf&utm_campaign=attribution)

**20**



***Không nhân lên các đối tượng mà không cần thiết.***

***- William Occam***

* **Giới thiệu**
* **Các lớp có nguồn gốc**

**Chức năng Thành viên; Người xây dựng và Người phá hủy**

**• Phân cấp lớp**

**Nhập trường; Chức năng ảo; Trình độ chuyên môn rõ ràng; Kiểm soát Ghi đè;sử dụng Thành viên cơ sở; Loại trả lại Thư giãn**

* **Các lớp trừu tượng**
* **Kiểm soát truy cập**

**được bảo vệ Các thành viên; Quyền truy cập vào các lớp cơ sở;sử dụng-Khai báo và Kiểm soát truy cập**



* **Con trỏ đến thành viên Con trỏ đến hàm**
* **Lời khuyên**

**20.1 Giới thiệu**

**Từ Simula, C ++ đã vay mượn ý tưởng của các lớp và cấu trúc phân cấp lớp. Ngoài ra, nó vay mượn ý tưởng thiết kế rằng các lớp nên được sử dụng để mô hình hóa các khái niệm trong thế giới của lập trình viên và ứng dụng. C +**

* **cung cấp các cấu trúc ngôn ngữ hỗ trợ trực tiếp các khái niệm thiết kế này. 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. Việc sử dụng các cấu trúc ngôn ngữ chỉ như các đạo cụ ký hiệu cho các kiểu lập trình truyền thống là bỏ sót những điểm mạnh chính của C ++.**

**Một khái niệm (ý tưởng, khái niệm, v.v.) không tồn tại một cách cô lập. Nó cùng tồn tại với các khái niệm liên quan và tạo ra phần lớn sức mạnh của nó từ các mối quan hệ với các khái niệm khác. 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 đại diện cho các khái niệm,**



**578** **Các lớp có nguồn gốc** **Chương 20**

**vấn đề trở thành cách thể hiện mối quan hệ giữa các khái niệm. Tuy nhiên, chúng ta không thể biểu thị các mối quan hệ tùy ý trực tiếp trong một ngôn ngữ lập trình. Ngay cả khi chúng tôi có thể, chúng tôi sẽ không muốn. Để trở nên hữu ích, các lớp của chúng ta nên được định nghĩa hẹp hơn so với các khái niệm hàng ngày của chúng ta - và chính xác hơn.**

**Khái niệm về một lớp dẫn xuất và các cơ chế ngôn ngữ liên quan của nó được cung cấp để 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 chúng có khái niệm chung về một hình dạng. Do đó, chúng tôi xác định rõ ràng lớpKhoanh tròn và lớp học Tam giác có lớp học Hình dạngđiểm chung. Trong trường hợp đó, lớp học bình thường, ở đâyHình dạng, được gọi là *cơ sở* lớp học hoặc *lớp siêu cấp* và các lớp bắt nguồn từ đó, tại đây Khoanh tròn và Tam giác, Được gọi là *nguồn gốc* các lớp học hoặc*các lớp con.* Đại diện cho một hình dạng sẽ là bỏ lỡ som**

* **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 thực hiện bằng cách chia sẻ các phương tiện được cung cấp bởimột lớp cơ sở**
  + ***Kế thừa giao diện*: cho phép các lớp dẫn xuất khác nhau được sử dụng thay thế cho nhau thông qua giaodiện 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 tính kế thừa được cung cấp bởi các mẫu (§3.4, Chương**

1. **thường được gọi là *tính đa hình thời gian biên dịch* (hoặc *đa hình tĩnh*). Cuộc thảo luận về phân cấp lớp được tổ chức thành ba chương:**
   * ***Các lớp có nguồn gốc* (Chương 20): Chương này giới thiệu các tính năng cơ bản của ngôn ngữ hỗtrợ lập trình hướng đối tượng. Các lớp cơ sở và dẫn xuất, các hàm ảo và kiểm soát truy cập được bao phủ.**
   * ***Cấu trúc phân cấp lớp* (Chương 21): Chương này tập trung vào việc sử dụng các lớp cơ sở và lớp dẫn xuấtđể tổ chức mã một cách hiệu quả xung quanh khái niệm phân cấp lớp. Phần lớn chương này được dành**

**cho thảo luận**

**tance (các lớp có m**

* + ***Nhận dạng loại thời gian chạy* (Chương 22): Chương này mô tả các kỹ thuật để điều hướng rõ ràng**



**các cấu trúc phân cấp lớp. Đặc biệt, các thao tác chuyển đổi loại hìnhdynamic\_castvà static\_cast được trình bày, cũng như hoạt động xác định kiểu của một đối tượng cho một trong các lớp cơ sở của nó (người đánh chữ).**

**Có thể tìm thấy phần giới thiệu ngắn gọn về ý tưởng cơ bản về tổ chức phân cấp của các kiểu trong Chương 3: các lớp cơ sở và dẫn xuất (§3.2.2) và các hàm ảo (§3.2.3). Các chương này sẽ xem xét các tính năng cơ bản này và các kỹ thuật thiết kế và lập trình liên quan của chúng một cách chi tiết hơn.**

**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 cấu trúc dữ liệu như sau:**



**Mục 20.2** **Các lớp có nguồn gốc** **579**

**struct Nhân viên {**

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

**char middle\_initial;**

**Ngày\_công\_dụng;**

**sở đoản;**

* ***...***

**};**

**Tiếp theo, chúng tôi có thể cố gắng xác định một người quản lý:**

**trình quản lý cấu trúc {**

**Nhân viên trống;**

**danh sách <Nhân viên∗> mức**

**độ ngắn của gro;**

* ***...***

**};**

**Một nhà quản lý cũng là một nhân viên; NSNhân viên dữ liệu được lưu trữ trong trống rỗng thành viên của một Người quản lý sự vật. Điều này có thể hiển nhiên đối với người đọc con người - đặc biệt là người đọc cẩn thận - nhưng không có gì nói với trình biên dịch và các công cụ khác rằngNgười quản lý cũng là một Nhân viên. MỘT Người quản lý∗ không phải là mộtNhân viên∗, vì vậy người ta không thể đơn giản sử dụng cái này khi cái kia được yêu cầu. Đặc biệt, người ta không thể đặt một Người quản lý lên danh sách Nhân viêns mà không cần viết mã đặc biệt. Chúng tôi có thể sử dụng chuyển đổi loại rõ ràng trên mộtNgười quản lý∗ hoặc đặt địa chỉ của trống rỗng thành viên trong danh sách Nhân viênNS. Tuy nhiên, cả hai giải pháp đều không phù hợp và có thể khá mù mờ. Cách tiếp cận đúng là tuyên bố rõ ràng rằngNgười quản lý *Là* một Nhân viên, với một vài thông tin được thêm vào:**

**struct Manager: công nhân viên {**

**danh sách <Nhân viên∗> nhóm;**

**mức độ ngắn;**

* ***...***

**};**



**Các Người quản lý Là *nguồn gốc* từ lớp học**

**Người quản lý có thành viên riêng của các**

**mem (tập đoàn, cấp độ, Vân vân.).**

**Khởi tạo thường được biểu diễn bằng đồ thị bởi một con trỏ từ lớp dẫn xuất đến lớp cơ sở của nó cho biết rằng lớp dẫn xuất tham chiếu đến cơ sở của nó (thay vì ngược lại):**

**Nhân viên**



**Người quản lý**

**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*di* *sản*. Một lớp cơ sở đôi khi được gọi là*lớp siêu cấp* và một lớp dẫn xuất a *lớp con*. Tuy nhiên, thuật ngữ này gâynhầm lẫn cho những người quan sát rằng dữ liệu trong một đối tượng lớp dẫn xuất là một 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.**



**580** **Các lớp có nguồn gốc** **Chương 20**

**Cách triển khai phổ biến và hiệu quả của khái niệm lớp dẫn xuất có một đối tượng của lớp dẫn xuất được biểu diễn như một đối tượng của lớp cơ sở, với thông tin thuộc về lớp dẫn xuất cụ thể được thêm vào cuối. Ví dụ:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Nhân viên:** | | **Người quản lý:** | |
|  |  |  |  |
|  | **tên đầu tiên** |  | **tên đầu tiên** |
|  | **tên gia đình** |  | **tên gia đình** |
|  | **. . .** |  | **. . .** |
|  |  |  |  |
|  |  |  | **tập đoàn** |
|  |  |  |  |

**Không có chi phí bộ nhớ được ngụ ý bằng cách dẫn xuất một lớp. Không gian yêu cầu chỉ là không gian yêu cầu của các thành viên.**

**Phát sinh Người quản lý từ Nhân viên theo cách này làm cho Người quản lý một loại phụ của Nhân viên, vì vậy mà một**

**Người quản lý có thể được sử dụng ở bất cứ đâu Nhân viên là chấp nhận được. Ví dụ: bây giờ chúng ta có thể tạo một danh sáchNhân viêns, một số người trong số họ là Người quản lýNS:**

**void f (Người quản lý m1, Nhân viên e1) {**

**danh sách <Nhân viên∗> danh sách {& m1, & e1); //**

***...***

**}**

**MỘT Người quản lý là (cũng) là một Nhân viên, vì vậy a Người quản lý∗ có thể được sử dụng như một Nhân viên∗. Tương tự, một Người quản lý& có thể được sử dụng như một Nhân viên&. Tuy nhiên, mộtNhân viên không nhất thiết phải là một Người quản lý, vì vậy mộtNhân viên∗ không thể được sử dụng như một Người quản lý∗. Nói chung, nếu một lớpNguồn gốc có một lớp cơ sở công khai (§20.5) Cơ sở, sau đó a Nguồn gốc∗ có thể được gán cho một biến loại Cơ sở∗ mà không cần sử dụng chuyển đổi kiểu rõ ràng. Đối lập**



**void g (Người quản lý mm, Nhân viên ee) {**

**Nhân viên∗ pe = & mm;**

**Người quản lý∗ pm = & ee;**

* ***OK: mọi Người quản lý đều là Nhân viên*// *lỗi: không phải mọi Nhân viên đều là Người quản lý***

**pm−> cấp = 2;** **// *thảm họa: ee không có cấp độ***

**pm = static\_cast <Người quản lý∗> (pe);** **// *brute force: hoạt động vì điểm pe*// *cho***

***người quản lý mm***

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

**}**

**Nói cách khác, một đối tượng của lớp dẫn xuất có thể được coi như một đối tượng của lớp cơ sở của nó khi được thao tác thông qua con trỏ và tham chiếu. Điều ngược lại là không đúng. Việc sử dụngstatic\_cast vàdynamic\_cast được thảo luận trong §22.2.**



**Mục 20.2** **Các lớp có nguồn gốc** **581**

**Sử dụng một lớp làm cơ sở tương đương với việc xác định một đối tượng (không tên) của lớp đó. Do đó, một lớp phải được định nghĩa để được sử dụng làm cơ sở (§8.2.2):**

**nhân viên lớp học;** **// *chỉ khai báo, không có định nghĩa***

**quản lý lớp: public Employee {// *lỗi: Nhân viên không được xác định***

* ***...***

**};**

**20.2.1 Chức năng thành viên**

**Cấu trúc dữ liệu đơn giản, đặc**

**biệt hữu ích. Chúng tôi nee**

**để làm như vậy mà không bị ràng buộc vào các chi tiết của một đại diện cụ thể. Ví dụ:**

**nhân viên lớp {**

**công cộng:**

**void print () const;**

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

***...***

**riêng:**

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

**char middle\_initial;**

* ***...***

**};**

**quản lý lớp: public Employee**

**{public:**

**void print () const;**

* ***...***

**};**



**Một thành viên của một lớp mệnh đề**

**dẫn xuất như thể chúng được khai báo**

**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:**

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

**cout << "name is" << family\_name << '\ n'; //** **// *lỗi!***

***...***

**}**

**Phiên bản thứ hai này của Người quản lý :: print () sẽ không biên dịch bởi vì tên gia đình không thể truy cập vào**

**Người quản lý :: print ().**



**582** **Các lớp có nguồn gốc** **Chương 20**

**Điều này gây ngạc nhiên cho một số người, nhưng hãy xem xét phương án thay thế: rằng một hàm thành viên của một lớp dẫn xuất có thể truy cập vào các thành viên riêng của lớp cơ sở của nó. Khái niệm 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. Hơn nữa, người ta không còn có thể tìm thấy tất cả các cách sử dụng của một tên riêng bằng cách nhìn vào các hàm được khai báo là thành viên và bạn bè của lớp đó. Người ta sẽ phải kiểm tra mọi tệp nguồn của chương trình hoàn chỉnh cho các lớp dẫn xuất, sau đó kiểm tra mọi chức năng của các lớp đó, sau đó tìm mọi lớp dẫn xuất từ các lớp đó, v.v. Điều này tốt nhất là tẻ nhạt và thường không thực tế. Ở những nơi có thể chấp nhận được, các thành viên được bảo vệ - thay vì riêng tư - có thể được sử dụng (§20.5).**

**Thông thường, giải pháp rõ ràng nhất là cho lớp dẫn xuất chỉ sử dụng các thành viên công khai của lớp cơ sở của nó. Ví dụ:**

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

**Nhân viên :: print ();** **// *in thông tin nhân viên*// *thông tin***

**cout << cấp;** ***cụ thể về trình quản lý in***

* ***...***

**}**

**Lưu ý rằng :: phải được sử dụng bởi vì in() đã được xác định lại trong Người quản lý. Việc tái sử dụng tên như vậy là điển hình. Người không cẩn thận có thể viết thế này:**

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

**in();** **// *Giáo sư!***

* ***thông tin cụ thể về trình quản lý in***

**}**

**Kết quả là một chuỗi các cuộc gọi đệ quy kết thúc bằng một số dạng lỗi chương trình.**

**20.2.2 Cấu tạo và cấu trúc phá hủy**



**Như thường lệ, các hàm tạo và d**

* **Các đối tượng được xây dựng từ dưới lên (cơ sở trước thành viên và thành viên trước khi dẫn xuất) và phá hủy từ trên xuống (dẫn xuất trước thành viên và thành viên trước cơ sở); §17.2.3.**
* **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 các cơ sở**

**của nó); §17.4.1.**

* **Thông thường, các trình hủy trong một hệ thống phân cấp cần phải ảo; §17.2.5.**
* **Các hàm tạo sao chép của các lớp trong một hệ thống phân cấp nên được sử dụng cẩn thận (nếu có) để tránh cắt; §17.5.1.4.**
* **Độ phân giải của một lệnh gọi hàm ảo, một dynamic\_cast, hoặc một typeid () trong một trình xây dựng**

**hoặc trình hủy phản ánh giai đoạn xây dựng và phá hủy (chứ không phải là loại của đối tượng chưa hoàn thành); §22.4.**

**Trong khoa học máy tính, '' lên '' và '' xuống '' có thể rất lẫn lộn. Trong văn bản nguồn, các định nghĩa của các lớp cơ sở phải xuất hiện trước các định nghĩa của các lớp dẫn xuất của chúng. Điều này ngụ ý rằng đối với các ví dụ nhỏ, các cơ sở xuất hiện phía trên các lớp dẫn xuất trên một màn hình. Hơn nữa, chúng ta có xu hướng vẽ cây với phần gốc ở trên cùng. Tuy nhiên, khi tôi nói về việc xây dựng các đối tượng từ dưới lên, ý tôi là**



**Mục 20.2.2** **Người xây dựng và Người phá hủy** **583**

**bắt đầu với những gì cơ bản nhất (ví dụ, các lớp cơ sở) và xây dựng những gì phụ thuộc vào đó (ví dụ, các lớp dẫn xuất) sau đó. Chúng tôi xây dựng từ gốc (các lớp cơ sở) về phía lá (các lớp dẫn xuất).**

**20.3 Phân cấp lớp**

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

**nhân viên của lớp {/ \* *...* \* /};**

**quản lý lớp: nhân viên công khai {/ \* *...* \* /};**

**Giám đốc lớp học: Public Manager {/ \**...* \* /};**

**Một tập hợp các lớp liên quan như**

**một cây, nhưng nó cũng có thể là**

**lớp học Tạm thời {/ \* *...* \* /};**

**trợ lý lớp học: nhân viên công cộng {/ \* *...* \* /};**

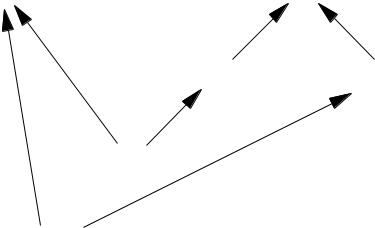
**class Temp: công cộng Tạm thời, công khai Trợ lý {/ \* *...* \* /}; Nhà tư vấn**

**lớp học: công chúng Tạm thời, người quản lý công khai {/ \**...* \* /};**

**hoặc bằng đồ thị:**

**Tạm thời**

**Nhân viên**



**Phụ tá**

**Người quản lý**



**Nhân viên bán thời gian**

**Tư vấn**



**Giám đốc**

**Vì vậy, như được giải thích trong deta**

**20.3.1 Trường loại**

**Để sử dụng các lớp dẫn xuất không chỉ là một cách viết tắt thuận tiện trong khai báo, chúng ta phải giải quyết vấn đề sau: Cơ sở∗, đối tượng được trỏ đến thực sự thuộc về kiểu dẫn xuất nào? Có bốn giải pháp cơ bản:**

1. **Đảm bảo rằng chỉ các đối tượng của một kiểu duy nhất được trỏ tới (§3.4, Chương 23).**
2. **Đặt một trường kiểu trong lớp cơ sở để các chức năng kiểm tra.**
3. **Sử dụng dynamic\_cast (§22.2, §22.6).**
4. **Sử dụng các hàm ảo (§3.2.3, §20.3.2).**

**Trừ khi bạn đã sử dụng final (§20.3.4.2), giải pháp 1 dựa trên nhiều kiến thức hơn về các kiểu liên quan hơn là khả dụng cho trình biên dịch. Nói chung, không phải là một ý kiến hay nếu cố gắng trở nên thông minh hơn hệ thống kiểu, nhưng (đặc biệt là kết hợp với việc sử dụng các mẫu), nó có thể được sử dụng để triển khai các vùng chứa đồng nhất (ví dụ: thư viện chuẩnvectơ và bản đồ) với hiệu suất vượt trội. Các giải pháp [2], [3], và [4] có thể được sử dụng để xây dựng danh sách không đồng nhất, tức là danh sách (con trỏ tới) các đối tượng thuộc một số kiểu khác nhau. Giải pháp [3] là một biến thể được hỗ trợ ngôn ngữ của giải pháp [2].**



**584** **Các lớp có nguồn gốc** **Chương 20**

**Giải pháp [4] là một biến thể đặc biệt an toàn của giải pháp [2]. Sự kết hợp của các giải pháp [1] và [4] đặc biệt thú vị và mạnh mẽ; trong hầu hết các tình huống, chúng mang lại mã rõ ràng hơn so với các giải pháp [2] và [3].**

**Trước tiên, chúng ta hãy kiểm tra giải pháp trường loại đơn giản để xem tại sao nó thường được tránh tốt nhất. Ví dụ về người quản lý / nhân viên có thể được định nghĩa lại như sau:**

**struct Nhân viên {**

**enum Empl\_type {man, empl};**

**Kiểu trống;**

**Nhân viên (): nhập {empl} {}**

**string first\_name, fa**

**char middle\_initial;**

**Ngày\_công\_dụng;**

**sở đoản;**

* ***...***

**};**

**struct Manager: công nhân viên {**

**Người quản lý () {type = man; }**

**danh sách <Nhân viên∗> nhóm; //*người quản lý***

**mức độ ngắn;**

* ***...***

**};**

**Với điều này, bây giờ chúng ta có thể viết một hàm in ra thông tin về mỗi Nhân viên:**

**void print\_employee (const Employee∗ e) {**



**switch (e−> type) {**

**case Nhân viên :: emp**

**cout << e−> family\_name << '\ t' << e−> khoa << '\ n'; //*...***

**nghỉ;**

**case Employee :: man:**

**{cout << e−> family\_name << '\ t' << e−> khoa << '\ n'; //*...***

**quản lý const∗ p = static\_cast <const Manager∗> (e);**

**cout << "level" << p−> level << '\ n';**

* ***...***

**nghỉ;**

**}**

**}**

**}**

**và sử dụng nó để in danh sách Nhân viêns, như thế này:**



**Mục 20.3.1** **Nhập trường** **585**

**void print\_list (const list <Nhân viên∗> & danh sách) {**

**cho (auto x: elist)**

**print\_employee (x);**

**}**

**Điều này hoạt động tốt, đặc biệt là trong một chương trình nhỏ do một người duy trì.** **Tuy nhiên, nó có một**

**điểm yếu cơ bản ở chỗ nó phụ thuộc vào việc lập trình viên thao tác các kiểu theo cách mà trình biên dịch không thể kiểm tra được. Vấn đề này thường trở nên tồi tệ hơn vì các chức năng nhưprint\_employee () thường được tổ chức để tận dụng tính chung của các lớp liên quan:**

**void print\_employee (khuyết điểm {**

**cout << e−> family\_name << '\ t' << e−> khoa << '\ n'; //*...***

**if (e−> type == Employee :: man) {**

**quản lý const∗ p = static\_cast <const Manager∗> (e);**

**cout << "level" << p−> level << '\ n';**

* ***...***

**}**

**}**

**Việc tìm kiếm tất cả các bài kiểm tra như vậy trên trường kiểu được chôn trong một hàm lớn xử lý nhiều lớp dẫn xuất có thể khó khăn. Ngay cả khi chúng đã được tìm thấy, việc hiểu những gì đang xảy ra vẫn có thể khó khăn. Hơn nữa, bất kỳ sự bổ sung nào của một loại mớiNhân viên liên quan đến sự thay đổi đối với tất cả các chức năng chính trong hệ thống - những chức năng chứa các bài kiểm tra trên trường loại. Lập trình viên phải xem xét mọi chức năng có thể cần kiểm tra trên trường kiểu sau khi thay đổi. Điều này ngụ ý nhu cầu truy cập mã nguồn quan trọng và dẫn đến chi phí cần thiết cho việc kiểm tra mã bị ảnh hưởng. Việc sử dụng chuyển đổi kiểu rõ ràng là một gợi ý mạnh mẽ rằng có thể cải thiện được.**

**Nói cách khác, sử dụng trường kiểu là một kỹ thuật dễ xảy ra lỗi dẫn đến các vấn đề về bảo trì. Các vấn đề gia tăng**



**trường loại gây ra vi phạm**

**trường phải biết về cách biểu diễn và các chi tiết khác về việc thực hiện 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 cho việc triển khai các lớp cơ sở và lớp dẫn xuất được đan xen theo những cách không mong muốn nhất. Trong một hệ thống phân cấp lớp lớn, có thể truy cập được (không riêng) dữ liệu trong một lớp cơ sở chung 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.**

**20.3.2 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 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. Ví dụ:**



**586** **Các lớp có nguồn gốc** **Chương 20**

**nhân viên lớp {**

**công cộng:**

**Nhân viên (const string & name, int**

**dept); void print () const;**

* ***...***

**riêng:**

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

* ***...***

**};**

**Từ khóa ảo lớp chỉ thị và**

**in() chức năng**

**được định nghĩa trong các lớp dẫn xuất, trình biên dịch đảm bảo rằng quyền in() cho những gì đã cho Nhân viên đối tượng được gọi trong mỗi trường hợp.**

**Để cho phép một khai báo hàm ảo hoạt động như một giao diện cho các hàm được xác định trong các lớp dẫn xuất, các kiểu đối số được chỉ định cho một hàm trong lớp dẫn xuất không được khác với các kiểu đối số được khai báo trong cơ sở và chỉ cho phép những thay đổi rất nhỏ đối với kết quả trả về loại (§20.3.6). Một hàm thành viên ảo đôi khi được gọi là*phương pháp*. Một chức năng ảo*cần phải* được định**

**nghĩa cho lớp mà nó được khai báo lần đầu (trừ khi nó được khai báo là một hàm ảo thuần túy; xem §20.4). Ví dụ:**

**void Nhân viên :: print () const {**

**cout << family\_name << '\ t' << khoa << '\ n'; //*...***

**}**

**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. Khi dẫn xuất một lớp, chỉ cần cung cấp một hàm**



**Quản lý lớp: public Em**

**public:**

**Trình quản lý (const string & name, int dept, int lvl);**

**void print () const;**

* ***...***

**riêng:**

**danh sách <Nhân viên∗> nhóm; mức độ ngắn;**

* ***...***

**};**

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

**Nhân viên :: print ();**

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

***...***

**}**



**Mục 20.3.2** **Chức năng ảo** **587**

**Một hàm từ một lớp dẫn xuất có cùng tên và cùng một tập hợp các kiểu đối số như một hàm ảo trong một cơ sở được cho là *ghi đè* phiên bản lớp cơ sở của hàm ảo. Hơn nữa, có thể ghi đè một hàm ảo từ một cơ sở có kiểu trả về dẫn xuất hơn (§20.3.6).**

**Ngoại trừ trường hợp chúng tôi nói rõ ràng phiên bản của một hàm ảo nào được gọi (như trong lệnh gọi**

**Nhân viên :: print ()), hàm ghi đè được chọn là thích hợp nhất cho đối tượng mà nó được gọi. Độc lập với lớp cơ sở (giao diện) nào được sử dụng để truy cập một đối tượng, chúng ta luôn nhận được cùng một hàm khi chúng ta sử dụng cơ chế gọi hàm ảo.**

**Chức năng toàn cục print\_employee () (§20.3.1) hiện không cần thiết vì in() các chức năng thành viên đã thay thế. Một danh sách vềNhân viêns có thể được in như thế này:**

**void print\_list (const list <E {**

**for (auto x: s)**

**x−> print ();**

**}**

**Mỗi Nhân viên sẽ được viết ra theo loại của nó. Ví dụ:**

**int main ()**

**{**

**Nhân viên e {"Brown", 1234}; Người**

**quản lý m {"Smith", 1234,2};**

**print\_list ({& e, & m});**

**}**

**sản xuất:**

**Smith 1234**

**cấp độ 2**

**Nâu 1234**



**Lưu ý rằng điều này sẽ hoạt động lớp**

**ev Người quản lý thậm chí đã đồng**

**trở thành nền tảng của các thiết kế hướng đối tượng và cung cấp mức độ ổn định cho một chương trình đang phát triển.**

**Nhận được hành vi '' đúng '' từ Nhân viêncác chức năng của nó độc lập với chính xác loại Nhân viên thực sự được sử dụng được gọi là *đa hình*. Một loại có các chức năng ảo được gọi là*loại đa hình* hoặc (chính xác hơn) a *kiểu đa hình thời gian chạy*. Để có được hành vi đa hình thời gian chạy trong C ++, các hàm thành viên được gọi phải làảo và các đối tượng phải được thao tác thông qua con trỏ hoặc tham chiếu. Khi thao tác trực tiếp một đối tượng (thay vì thông qua con trỏ hoặc tham chiếu), kiểu chính xác của nó được trình biên dịch biết nên không cần đến tính đa hình thời gian chạy.**

**Theo mặc định, một hàm ghi đè một hàm ảo chính nó sẽ trở thành ảo. Chúng tôi có thể, nhưng không**

**phải, lặp lại ảo trong một lớp dẫn xuất. Tôi không khuyên bạn nên lặp lạiảo. Nếu bạn muốn rõ ràng, hãy sử dụngghi đè (§20.3.4.1).**

**Rõ ràng, để triển khai tính đa hình, trình biên dịch phải lưu trữ một số loại thông tin kiểu trong mỗi đối tượng của lớp Nhân viên và sử dụng nó để gọi đúng phiên bản của hàm ảo in(). Trong một triển khai điển hình, không gian được lấy chỉ đủ để chứa một con trỏ (§3.2.3): kỹ thuật thực thi thông thường là để trình biên dịch chuyển đổi tên của một hàm ảo thành một chỉ mục thành**

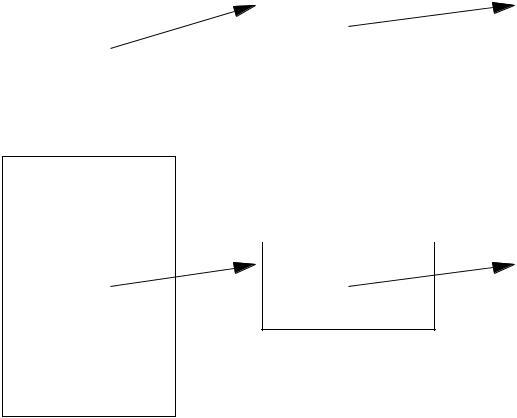


**588** **Các lớp có nguồn gốc** **Chương 20**

**một bảng con trỏ đến các hàm. Bảng đó thường được gọi là*bảng chức năng ảo* hoặc đơn giản làvtbl. Mỗi lớp với các chức năng ảo đều cóvtbl xác định các chức năng ảo của nó. Điều này có thể được biểu diễn bằng đồ thị như sau:**

**Nhân viên:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **tên đầu tiên** | **vtbl:** | | | |  |
| **Tên thứ hai** |  |
| **. . .** |  |  |  | **Nhân viên :: print ()** |  |
|  |  | **. . .** |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |  |



**Người quản lý:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **tên đầu tiên** |  |  |  |  |  |
| **Tên thứ hai** | **vtbl:** | | | |  |
| **. . .** |  |  |  | **Người quản lý :: print ()** |  |
|  |  |  |  |  |
|  |  |  |  |  |  |

**. . .**

**tập đoàn**

**cấp độ**

**. . .**

**Các chức năng trong vtbl cho phép đối tượng được sử dụng chính xác ngay cả khi kích thước của đối tượng và cách bố trí dữ liệu của nó đối với người gọi là không xác định. Việc thực hiện một người gọi chỉ cần biết vị trí của vtbl trong một Nhân viên và chỉ mục được sử dụng cho mỗi chức năng ảo. Cơ chế gọi ảo này có thể được thực hiện gần như hiệu quả như cơ chế '' lệnh gọi hàm thông thường '' (trong phạm vi 25%), do đó, mối quan tâm về hiệu quả sẽ không ngăn cản bất kỳ ai sử dụng một hàm ảo nơi một lệnh gọi hàm thông thường sẽ hiệu quả ở mức chấp nhận được. Chi phí không gian của nó là một con trỏ trong mỗi đối tượng của một lớp với các hàm ảo cộng với mộtvtbl cho mỗi lớp như vậy. Bạn chỉ trả chi phí này cho các đối tượng của một lớp có av**



**thêm chức năng ảo f**

**tion, một lượng không gian có thể so sánh được sẽ là cần thiết cho trường loại.**

**Một hàm ảo được gọi từ một hàm tạo hoặc một hàm hủy phản ánh rằng đối tượng được xây dựng một phần hoặc bị phá hủy một phần (§22.4). Do đó, thường là một ý tưởng tồi nếu gọi một hàm ảo từ một hàm tạo hoặc một hàm hủy.**

**20.3.3 Chứng chỉ rõ ràng**

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

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

**Nhân viên :: print (); //*không phải là một cuộc***

***gọi ảo*cout << "\ tlevel" << level << '\ n'; //*...***

**}**



**Mục 20.3.3** **Chứng chỉ rõ ràng** **589**

**Nếu không thì, Người quản lý :: print () sẽ phải chịu một đệ quy vô hạn. Việc sử dụng một tên đủ điều kiện có một hiệu quả mong muốn khác. Đó là, nếu một hàm ảo cũngnội tuyến (như không có gì lạ), sau đó thay thế nội tuyến có thể được sử dụng cho các cuộc 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. CácNgười quản lý :: print () chức năng là một ví dụ về điều này. Bởi vì loại đối tượng được xác định trong lệnh gọi củaNgười quản lý :: print (), nó không cần phải được xác định động lại cho kết quả**

**cuộc gọi của Nhân viên :: print ().**

**20.3.4 Kiểm soát ghi đè**

**Nếu bạn khai báo một hàm trong**

**hàm trong một lớp cơ sở, th**

**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. Xem xét:**

**struct B0 {**

**void f (int) const;**

**void ảo g (double);**

**};**

|  |  |  |
| --- | --- | --- |
| **struct B1: B0 {/ \* *...*** | **\* /};** |  |
| **struct B2: B1 {/ \**...*** | **\* /};** |  |
| **struct B3: B2 {/ \**...*** | **\* /};** |  |
| **struct B4: B3 {/ \**...*** | **\* /};** |  |
| **struct B5: B4 {/ \**...*** | **\* /};** |  |
| **struct D: B5 {** |  |  |
| **void f (int) const;** | | **// *ghi đè f () trong lớp cơ sở*//** |
| **void g (int);** |  | ***ghi đè g () trong lớp cơ sở*//** |
| **ảo int h ();** |  | ***ghi đè h () trong lớp cơ sở*** |



**};**

**Điều này minh họa ba lỗi trong đó các lớp B0...B5 có nhiều thành viên và nằm rải rác trên nhiều tệp tiêu đề. Ở đây:**

* **B0 :: f () không phải ảo, vì vậy bạn không thể ghi đè nó, chỉ ẩn nó (§20.3.5).**
* **D :: g () không có cùng loại đối số như B0 :: g (), vì vậy nếu nó ghi đè bất cứ thứ gì thì nó không**

**phải là hàm ảo B0 :: g (). Rất có thể,D :: g () chỉ ẩn B0 :: g ().**

* **Không có chức năng nào được gọi NS() trong B0, nếu như D :: h () ghi đè bất cứ thứ gì, nó không phải là một hàm từ B0. Rất có thể, nó đang giới thiệu một chức năng ảo hoàn toàn mới.**

**Tôi đã không cho bạn thấy những gì trong B1...B5, vì vậy có thể một cái gì đó hoàn toàn khác đang xảy ra do các khai báo trong các lớp đó. Cá nhân tôi không (dư thừa) sử dụngảo cho một chức năng có nghĩa là ghi đè. Đối với các chương trình nhỏ hơn (đặc biệt là với một trình biên dịch có cảnh báo phù hợp chống lại các lỗi phổ biến) thì việc ghi đè được thực hiện đúng cách không khó. Tuy nhiên, đối với các hệ thống phân cấp lớn hơn, các điều khiển cụ thể hơn rất hữu ích:**

* **ảo: Hàm có thể bị ghi đè (§20.3.2).**
* **=0: Chức năng phải là ảo và phải được ghi đè (§20.4).**



**590** **Các lớp có nguồn gốc** **Chương 20**

* **ghi đè: Hàm được dùng để ghi đè một hàm ảo trong một lớp cơ sở (§20.3.4.1).**
* **final: Hàm không có nghĩa là bị ghi đè (§20.3.4.2).**

**Trong trường hợp không có bất kỳ kiểm soát nào trong số này,tĩnh hàm thành viên là ảo nếu và chỉ khi nó ghi đè ảo chức năng trong một lớp cơ sở (§20.3.2).**

**Một trình biên dịch có thể cảnh báo chống lại việc sử dụng không nhất quán các điều khiển ghi đè rõ ràng. Ví dụ, một khai báo lớp sử dụngghi đè đối với bảy trong số chín hàm lớp cơ sở ảo có thể gây nhầm lẫn cho người bảo trì.**

**20.3.4.1 ghi đè**

**Chúng tôi có thể rõ ràng về ou**

**struct D: B5 {**

**void f (int) const ghi đè;** **// *lỗi: B0 :: f () không phải là ảo*// *error:***

**void g (int) ghi đè;** ***B0 :: f () nhận đối số kép*// *lỗi: không có***

**ghi đè ảo int h ();** ***hàm h () để ghi đè***

**};**

**Đưa ra định nghĩa này (và giả sử rằng các lớp cơ sở trung gian B1...B5 không cung cấp các chức năng liên quan), cả ba tờ khai đều sai sót.**

**Trong một hệ thống phân cấp lớp lớn hoặc phức tạp với nhiều chức năng ảo, tốt nhất là sử dụng ảo**

**chỉ để giới thiệu một chức năng ảo mới và sử dụng ghi đè trên tất cả các chức năng được sử dụng làm bộ ghi đè.**

**Sử dụngghi đè hơi dài dòng nhưng làm rõ ý định của lập trình viên.**

**Các ghi đè specifier đứng cuối cùng trong một khai báo, sau tất cả các phần khác. Ví dụ:**

**void f (int) const noexcept override; //*OK (nếu có f () thích hợp để ghi đè)*ghi đè void f (int) const noexcept; //*lỗi cú pháp*void f (int) ghi đè const noexcept; //*lỗi cú pháp***

**Và vâng, thật phi lý khi ảo là một tiền tố và ghi đè là một hậu tố. Đây là một phần của cái giá mà chúng tôi phải trả cho khả năng tương thích và ổn định trong nhiều thập kỷ.**



**Một ghi đè specifier là**

**định nghĩa lớp. Ví dụ**

**class Derived: public Base { void f () ghi đè;**

**void g () ghi đè;**

* ***OK nếu Base có f ảo ()*// *OK nếu Base có một g () ảo***

**};**

**void Derived :: f () ghi đè {** **// *error: override out of class***

* ***...***

**}**

**void g ()** **// *VÂNG***

**{**

* ***...***

**}**

**Thật kỳ lạ, ghi đè không phải là một từ khóa; nó là cái được gọi là*từ khóa theo ngữ cảnh*. Đó là,ghi đè có**



**Mục 20.3.4.1** **ghi đè** **591**

**một ý nghĩa đặc biệt trong một số ngữ cảnh nhưng có thể được sử dụng làm định danh ở những nơi khác. Ví dụ:**

**int override = 7;**

**struct Dx: Cơ sở {**

**int ghi đè;**

**ghi đè int f ()**

**{**

**trả về ghi đè + :: ghi đè;**

**}**

**};**

**Đừng say mê như vậy**

**từ khóa theo ngữ cảnh, chứ không phải từ khóa thông thường, là có một lượng mã đáng kể đã được sử dụng ghi đè như một định danh thông thường trong nhiều thập kỷ. Từ khóa theo ngữ cảnh khác làcuối cùng (§20.3.4.2).**

**20.3.4.2 final**

**Khi chúng ta khai báo một hàm thành viên, chúng ta có một sự lựa chọn giữa ảo và không ảo (mặc định). Chúng tôi sử dụngảo đối với các hàm, chúng tôi muốn người viết các lớp dẫn xuất có thể định nghĩa hoặc định nghĩa lại. Chúng tôi lựa chọn dựa trên ý nghĩa (ngữ nghĩa) của lớp chúng tôi:**

* **Chúng ta có thể hình dung sự cần thiết của các lớp dẫn xuất tiếp theo không?**
* **Người thiết kế lớp dẫn xuất có cần xác định lại hàm để đạt được mục đích hợp lý không?**
* **Việc ghi đè một hàm có dễ bị lỗi không (nghĩa là một hàm ghi đè có khó cung cấp ngữ nghĩa**

**mong đợi của một hàm ảo) không?**

**Nếu câu trả lời là '' không '' cho cả ba câu hỏi, chúng ta có thể để hàm khôngảo để đạt được sự đơn giản của thiết kế và đôi khi một số hiệu suất (chủ yếu là từ nội tuyến). Thư viện tiêu chuẩn có đầy đủ các ví dụ về điều này.**



**Hiếm hơn rất nhiều, chúng ta có sự khởi**

**đầu của một tập hợp các cl có nguồn gốc**

**một cây cú pháp trừu tượng cho một ngôn ngữ trong đó tất cả các cấu trúc ngôn ngữ đã được định nghĩa là các lớp nút cụ thể bắt nguồn từ một vài giao diện. Chúng ta chỉ cần lấy một lớp mới nếu chúng ta thay đổi ngôn ngữ. Trong trường hợp đó, chúng tôi có thể muốn ngăn người dùng ghi đè các chức năng ảo bởi vì điều duy nhất mà những ghi đè như vậy có thể làm là thay đổi ngữ nghĩa của ngôn ngữ của chúng tôi. Đó là, chúng tôi có thể muốn đóng thiết kế của mình để sửa đổi từ người dùng. Ví dụ:**

**struct Node {// *lớp giao diện***

**kiểu virtual Type () = 0; //**

***...***

**};**

**class If\_statement: public Node**

**{public:**

**Gõ type () ghi đè cuối** **// *ngăn chặn việc ghi đè hơn nữa***

**cùng; //*...***

**};**



**void Derived :: f () final {**

**// *...***

**}**

**// *lỗi: cuối cùng ra khỏi lớp***

**};**

**// *...***

**void g () cuối cùng;**

**void f () cuối cùng;**

**class Derived: public Base {**

**// *OK nếu Base có f ảo ()*// *OK* *nếu Base có một g () ảo***

***// ...***

***};***

***Đối với tốt và xấu, thêm final vào lớp không chỉ ngăn ghi đè, nó còn ngăn chặn việc dẫn xuất thêm từ một lớp. Có những người sử dụngfinal để cố gắng đạt được hiệu suất - xét cho cùng,ảo chức năng nhanh hơn một ảo một (có thể bằng 25% đối với cách triển khai hiện đại) và mang lại cơ hội lớn hơn cho nội tuyến (§12.1.5). Tuy nhiên, đừng sử dụng một cách mù quángfinal như một trợ giúp tối ưu hóa; nó ảnh hưởng đến lớp học***

***hiếm khi đáng kể. Làm một sốcuối cùng nơi nó phản ánh rõ ràng thiết kế phân cấp lớp mà bạn cho là phù hợp. Đó là, sử dụngfinal để phản ánh một nhu cầu ngữ nghĩa.***

***MỘT final specifier không phải là một phần của kiểu hàm và không thể được lặp lại trong một định nghĩa ngoài lớp. Ví dụ:***

***// error: For\_statement là cuối cùng***

***class For\_statement final: public Node {public:***

***Gõ kiểu () ghi đè;***

***// ...***

***};***

***class Modified\_for\_statement: public For\_statement { Gõ kiểu () ghi đè;***

***// ...***

***};***

***Chúng tôi có thể tạo ra mọi ảo chức năng thành viên của một lớp final; chỉ cần thêmfinal sau tên lớp. Ví dụ:***

**592** **Các lớp có nguồn gốc** **Chương 20**

**Trong một hệ thống phân cấp lớp thực tế, sẽ có một số lớp trung gian giữa giao diện chung (ở đây, Nút) và lớp dẫn xuất đại diện cho một cấu trúc ngôn ngữ cụ thể (ở đây, If\_statement). Tuy nhiên, điểm mấu chốt về ví dụ này làNode :: type () có nghĩa là bị ghi đè (đó là lý do tại sao nó được khai báo ảo) và bộ ghi đè của nó If\_statement :: type () không (đó là lý do tại sao nó được khai báocuối cùng). Sau khi sử dụngfinal đối với một hàm thành viên, nó không thể bị ghi đè nữa và việc cố gắng làm như vậy là một lỗi. Ví dụ:**

**class Modified\_if\_statement: public If\_statement**

**{public:**

**Gõ kiểu () ghi đè;** **// *error: if\_statement :: type () là cuối cùng***

******

**Mục 20.3.4.2** **cuối cùng** **593**

**void g () cuối cùng** **// *VÂNG***

**{**

* ***...***

**}**

**Như ghi đè (§20.3.4.1), final là một từ khóa theo ngữ cảnh. Đó là,final có ý nghĩa đặc biệt trong một số ngữ cảnh nhưng có thể được sử dụng như một định danh thông thường ở những nơi khác. Ví dụ:**

**int cuối cùng = 7;**

**struct Dx: Cơ sở {**

**int cuối cùng;**

**int f () cuối cùng**

**{**

**trả về cuối cùng + :: cuối cùng;**

**}**

**};**

**Đừng ham thích sự thông minh như vậy; nó làm phức tạp thêm việc bảo trì. Lý do duy nhất màfinal là một từ khóa theo ngữ cảnh, chứ không phải là một từ khóa thông thường, có tồn tại một lượng mã đáng kể đã được sử dụng final như một định danh thông thường trong nhiều thập kỷ. Từ khóa theo ngữ cảnh khác làghi đè(§20.3.4.1).**

**20.3.5 sử dụng Thành viên cơ sở**

**Các hàm không quá tải trên các phạm vi (§12.3.3). Ví dụ:**

**cơ sở struct {**

**void f (int);**

**};**



**struct Derived: Base {**

**void f (kép);**

**};**

**vô hiệu sử dụng (Bắt nguồn từ d)**

**{**

**df (1);** **// *gọi Derived :: f (double)***

**Cơ sở & 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 hàm thành viên phù hợp nhất được sử dụng. Đối với không gian tên,sử dụng-các khai báo có thể được sử dụng để thêm một chức năng vào một phạm vi. Ví dụ:**

**struct D2: Cơ sở {**

**sử dụng Base :: f;** **// *đưa tất cả fs từ Base vào D2***

**void f (kép);**

**};**



**594** **Các lớp có nguồn gốc** **Chương 20**

**void use2 (D2 d)**

**{**

**df (1);** **// *gọi D2 :: f (int), tức là Base :: f (int)***

**Cơ sở & 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 (§16.2).**

**Vài sử dụng-các khai báo có thể mang tên từ nhiều lớp cơ sở. Ví dụ:**

**struct B1 {**

**void f (int);**

**};**

**struct B2 {**

**void f (kép);**

**};**

**struct D: B1, B2 {**

**sử dụng B1 :: f;**

**sử dụng B2 :: f;**

**void f (char);**

**};**

**vô hiệu sử dụng (D d)**

**{**

**df (1);** **// *gọi D :: f (int), nghĩa là B1 :: f (int)***

**df ('a');** **// *gọi D :: f (char)***

**df (1,0);** **// *gọi D :: f (double), nghĩa là B2 :: f (double)***

**}**



**Chúng ta có thể mang lại phạm vi lớp cho các hàm tạo i bằng một sử dụng-dec Tuyên bố có quyền truy cập của nó được xác định bởi vị trí của sử dụng-tuyên ngôn; xem §20.5.3. Chúng tôi không thể sử dụngsử dụng-directives để đưa tất cả các thành viên của một lớp cơ sở vào một lớp dẫn xuất.**

**20.3.5.1 Trình tạo kế thừa**

**Giả sử tôi muốn một vectơ giống như std :: vector, nhưng với kiểm tra phạm vi đảm bảo. Tôi có thể thử điều này:**

**mẫu <lớp T>**

**struct Vector: std :: vector <T> {**

**T & operator [] (size\_type i) {check (i); return this−> elem (i); }**

**const T & operator [] (size\_type i) const {check (i); return this−> elem (i); }**

**void check (size\_type i) {if (this−> size () <i) throw range\_error {"Vector :: check () fail"}; }**

**};**



**Mục 20.3.5.1** **Trình tạo kế thừa** **595**

**Thật không may, chúng tôi sẽ sớm phát hiện ra rằng định nghĩa này không đầy đủ. Ví dụ:**

**Vectơ <int> v {1, 2, 3, 5, 8}; //*lỗi: không có phương thức khởi tạo danh sách khởi tạo***

**Kiểm tra nhanh sẽ cho thấy rằng Véc tơ không thể kế thừa bất kỳ hàm tạo nào từ std :: vector.**

**Đó không phải là một quy tắc bất hợp lý: nếu một lớp thêm các thành viên dữ liệu vào một cơ sở hoặc yêu cầu một lớp bất biến chặt chẽ hơn, sẽ là một thảm họa để kế thừa các hàm tạo. Tuy vậy,Véc tơ đã không làm bất cứ điều gì như vậy.**

**Chúng tôi giải quyết vấn đề bằng cách đơn giản nói rằng các hàm tạo phải được kế thừa:**

**mẫu <lớp T>**

**struct Vector: std :: vector <T> {**

**sử dụng vectơ <T> :: ve**

**T & operator = [] (size\_type i) {check (i); return this−> elem (i); }**

**const T & operator = (size\_type i) const {check (i); return this−> elem (i); }**

**void check (size\_type i) {if (this−> size () <i) ném Bad\_index (i); }**

**};**

**Vectơ <int> v {1, 2, 3, 5, 8};** **// *OK: sử dụng phương thức khởi tạo danh sách khởi tạo từ std :: vector***

**Việc sử dụng này của sử dụng hoàn toàn tương đương với việc sử dụng nó cho các hàm thông thường (§14.4.5, §20.3.5).**

**Nếu bạn chọn như vậy, bạn có thể tự bắn vào chân mình bằng cách kế thừa các hàm tạo trong một lớp dẫn xuất, trong đó bạn xác định các biến thành viên mới cần khởi tạo rõ ràng:**

**struct B1 {**

**B1 (int) {}**

**};**

**struct D1: B1 {**

**sử dụng B1 :: B1;** **// *khai báo ngầm D1 (int)*//**



**dây;**

***Strin***

**int x;**

**// *chúng tôi***

**};**

**kiểm tra void ()**

**{**

**Đ1 d {6};**

**// *oops: dx không được khởi tạo*// *lỗi: D1***

**Đ1 e;**

***không có hàm tạo mặc định***

**}**

**Lý do mà D1 :: s được khởi tạo và D1 :: x không phải là hàm tạo kế thừa tương đương với một hàm tạo chỉ đơn giản là khởi tạo cơ sở. Trong trường hợp này, chúng tôi có thể viết:**

**struct D1: B1 {**

**D1 (int i): B1 (i) {}**

**string s;**

**int x;**

**// *chuỗi có một hàm tạo mặc định*// *chúng tôi***

***'' quên '' cung cấp để khởi tạo x***

**};**

**Một cách để loại bỏ viên đạn khỏi chân của bạn là thêm trình khởi tạo thành viên trong lớp (§17.4.4):**



**596** **Các lớp có nguồn gốc** **Chương 20**

**struct D1: B1 {**

**sử dụng B1 :: B1;** **// *khai báo ngầm D1 (int)*//**

**int x {0};** ***lưu ý: x được khởi tạo***

**};**

**kiểm tra void ()**

**{**

**Đ1 d {6};** **// *dx bằng 0***

**}**

**Thông thường, tốt nhất là nên tránh khôn khéo và hạn chế việc sử dụng các hàm tạo kế thừa trong các trường hợp đơn giản mà không có bộ nhớ dữ liệu**

**20.3.6 Thư giãn loại trả lại**

**Có một quy tắc nới lỏng rằng kiểu của một hàm ghi đè phải giống với kiểu của hàm ảo mà nó ghi đè. Đó là, nếu loại trả lại ban đầu làNS∗, thì kiểu trả về của hàm ghi đè có thể là NS∗, cung cấp NS là một cơ sở công cộng của NS. Tương tự, kiểu trả về củaNS&có thể được thoải mái NS&. Điều này đôi khi được gọi là*lợi nhuận đồng biến* luật lệ.**

**Việc nới lỏng này chỉ áp dụng cho các loại trả về là con trỏ hoặc tham chiếu, và không áp dụng cho '' con trỏ thông minh '' chẳng hạn như unique\_ptr (§5.2.1). Đặc biệt, không có sự nới lỏng tương tự các quy tắc cho các loại đối số vì điều đó sẽ dẫn đến vi phạm loại.**

**Hãy xem xét một hệ thống phân cấp lớp đại diện cho các loại biểu thức khác nhau. Ngoài các phép toán thao tác với các biểu thức, lớp cơ sởExpr sẽ cung cấp các phương tiện để tạo các đối tượng biểu thức mới của các kiểu biểu thức khác nhau:**

**lớp Expr {**

**công cộng:**

**Expr ();** **// *nhà xây dựng mặc định***

**Expr (const Expr &);** **// *sao chép hàm tạo***



**Expr ảo∗ new\_ex virtual**

**Expr∗ dòng vô tính() // *...***

**};**

* **tưởng là new\_expr () tạo một đối tượng mặc định của kiểu biểu thức và dòng vô tính() tạo một bản sao của đối tượng. Cả hai sẽ trả về một đối tượng của một số lớp cụ thể có nguồn gốc từExpr. Họ không bao giờ có thể trả lại một '' đồng bằngExpr'' tại vì Expr đã được khai báo một cách có chủ ý và thích hợp là một lớp trừu tượng.**

**Một lớp dẫn xuất có thể ghi đè new\_expr () và / hoặc dòng vô tính() để trả về một đối tượng thuộc kiểu riêng của nó:**

**class Cond: public Expr**

**{public:**

**Cond ();**

**Cond (const Cond &);**

**Cond∗ new\_expr () override {return new Cond (); } Điều**

**kiện∗ clone () override {return new Cond (∗cái này); } //*...***

**};**



**Mục 20.3.6** **Loại trả lại Thư giãn** **597**

**Điều này có nghĩa là đã cho một đối tượng của lớp Expr, người dùng có thể tạo một đối tượng mới '' chỉ cùng loại. '' Ví dụ:**

**người dùng vô hiệu (Expr∗ P) {**

**Expr∗ p2 = p−> new\_expr (); //**

***...***

**}**

**Con trỏ được chỉ định cho p2 được tuyên bố là trỏ đến một đồng bằng '' Expr, '' nhưng nó sẽ trỏ đến một đối tượng thuộc loại bắt nguồn từ Expr, nhu la Cond. Loại trả lại củaCond Cond được nhân bản mà không cần la dòng vô tính() trả**

**lại một Thêm vào∗. Ví dụ:**

**void user2 (Cond∗ pc, Bổ sung∗ pa) {**

**Cond∗ p1 = pc−> clone (); Thêm**

**vào∗ p2 = pa−> clone (); //*...***

**}**

**Nếu chúng ta sử dụng dòng vô tính() cho một Expr chúng tôi chỉ biết rằng kết quả là một Expr∗:**

**void user3 (Cond∗ pc, Expr∗ pe) {**

**Cond∗ p1 = pc−> clone ();**

**Cond∗ p2 = pe−> clone (); //** **// *error: Expr :: clone () trả về một Expr \****

***...***

**}**

**Bởi vì các chức năng như new\_expr () và dòng vô tính() là ảo và chúng (gián tiếp) xây dựng các đối tượng, chúng thường được gọi là *các nhà xây dựng ảo*. Mỗi đơn giản chỉ sử dụng một phương thức khởi tạo để tạo một đối tượng phù hợp.**



**Để tạo một đối tượng, một hàm tạo không thể được ảo. Hơn nữa, một hàm tạo không phải là một hàm hoàn toàn bình thường. Đặc biệt, nó tương tác với các quy trình quản lý bộ nhớ theo cách mà các hàm thành viên thông thường không làm được. Vì vậy, bạn không thể lấy một con trỏ đến một phương thức khởi tạo và chuyển con trỏ đó đến một hàm tạo đối tượng.**

**Cả hai hạn chế này đều có thể được phá vỡ bằng cách xác định một hàm gọi một phương thức khởi tạo và trả về một đối tượng được xây dựng. Điều này thật may mắn vì việc tạo một đối tượng mới mà không biết loại chính xác của nó thường rất hữu ích. CácIval\_box\_maker (§21.2.4) là một ví dụ về một lớp được thiết kế đặc biệt để làm điều đó.**

**20.4 Các lớp trừu tượng**

**Nhiều lớp giống với lớp Nhân viên ở chỗ chúng hữu ích như chính chúng, làm giao diện cho các lớp dẫn xuất và như một phần của việc triển khai các lớp dẫn xuất. Đối với các lớp như vậy, các kỹ thuật được mô tả trong §20.3.2 là đủ. Tuy nhiên, không phải tất cả các lớp đều tuân theo khuôn mẫu đó. Một số lớp học, chẳng hạn như một lớp họcHình dạng, biểu diễn các khái niệm trừu tượng mà đối tượng không thể tồn tại. MỘTHình dạng làm cho**



**598** **Các lớp có nguồn gốc** **Chương 20**

**chỉ có nghĩa là cơ sở của một số lớp có nguồn gốc từ nó. Điều này có thể được thấy từ thực tế là không thể cung cấp các định nghĩa hợp lý cho các chức năng ảo của nó:**

**Hình dạng lớp {**

**công cộng:**

**virtual void xoay (int) {throw runtime\_error {"Shape :: xoay"}; } virtual** **// *không lịch sự***

**void draw () const {throw runtime\_error {"Shape :: draw"}; } //*...***

**};**

**Cố gắng tạo ra một hình dạng không xác định này là ngớ ngẩn nhưng hợp pháp:**

**Hình dạng; //*ngớ ngẩn: '' shapeles***

**Thật là ngớ ngẩn bởi vì mọi hoạt động**

**Một giải pháp thay thế tốt hơn là khai báo các hàm ảo của lớp Hình dạng được *các chức năng ảo thuần túy*.**

**Một chức năng ảo được '' tạo ra thuần túy '' bởi '' trình khởi tạo giả '' = 0:**

**Hình dạng lớp {** **// *lớp trừu tượng***

**công cộng:**

**void quay ảo (int) = 0; ảo void draw () const = 0; bool ảo is\_closed () const = 0; //*...***

**ảo ˜Shape ();**

**};**

**// *chức năng ảo thuần túy*//**

***chức năng ảo thuần túy*//**

***chức năng ảo thuần túy***

* ***ảo***

**Một lớp có một hoặc nhiều hàm ảo thuần túy là một *lớp trừu tượng*và không có đối tượng nào của lớp trừu tượng đó có thể được tạo:**

**Hình dạng; //*error: biến của lớp trừu tượng Hình dạng***

**Một lớp trừu tượng được sử dụng như một giao diện cho các đối tượng được truy cập thông qua con trỏ và tham chiếu (để duy trì hành vi đa hình). Do đó, điều quan trọng là đối với một lớp trừu tượng phải có một trình hủy ảo (§3.2.4,**



**được sử dụng để tạo các đối tượng bằng cách sử dụng**

**Một lớp trừu tượng chỉ có thể được sử dụng làm giao diện cho các lớp khác. Ví dụ:**

**điểm lớp {/ \* *...* \* /};**

**class Circle: public Shape**

**{public:**

**void xoay (int) override {}**

**void draw () const override;**

**bool is\_closed () const override {return true; }**

**Circle (Điểm p, int r);**

**riêng:**

**Tâm điểm;**

**bán kính int;**

**};**



**Mục 20.4** **Các lớp trừu tượng** **599**

**Một hàm thuần ảo không được định nghĩa trong lớp dẫn xuất vẫn là một hàm thuần ảo, vì vậy lớp dẫn xuất cũng là một lớp trừu tượng. Điều này cho phép chúng tôi xây dựng các triển khai theo từng giai đoạn:**

**class Polygon: public Shape** **// *lớp trừu tượng***

**{public:**

**bool is\_closed () const override {return**

**true; } //*... vẽ và xoay không bị ghi đè ...***

**};**

**Đa giác b {p1, p2, p3, p4};** **// *error: khai báo đối tượng của lớp trừu tượng Polygon***

**Đa giác vẫn còn trừu tượng vì chúng tôi đã không ghi đè vẽ() và quay(). Chỉ khi điều đó được thực hiện, chúng tôi mới có một lớp học mà từ đó**

**class Irregular\_polygon: public Polygon {**

**danh sách <Điểm> lp;**

**công cộng:**

**Irregular\_polygon (Initializer\_list <Điểm>);**

**void draw () const override;**

**ghi đè void quay (int); //*...***

**};**

**Đa giác không đều {p1, p2, p3, p4};** **// *giả sử rằng p1 .. p4 là các Điểm được xác định ở đâu đó***

**Một lớp trừu tượng cung cấp một giao diện mà không để lộ chi tiết triển khai. Ví dụ: một hệ điều hành có thể ẩn thông tin chi tiết về trình điều khiển thiết bị của nó đằng sau một lớp trừu tượng:**

**class Character\_device**

**{public:**

**int ảo open (int opt) = 0; int**

**ảo đóng (int o**

**đọc int ảo (char∗ghi**

**int ảo (khuyết điểm**

**ảo int ioctl (int ...) = 0;**

**//*kiểm soát I / O thiết bị***



**ảo C̃haracter\_device() {}**

**// *máy hủy ảo***

**};**

**Sau đó, chúng tôi có thể chỉ định trình điều khiển là các lớp bắt nguồn từ Character\_device và thao tác với nhiều trình điều khiển khác nhau thông qua giao diện đó.**

**Phong cách thiết kế được hỗ trợ bởi các lớp trừu tượng được gọi là *kế thừa giao diện* trái ngược với**

***kế thừa thực hiện* được hỗ trợ bởi các lớp cơ sở với trạng thái và / hoặc các chức năng thành viên được xác định.Có thể kết hợp hai cách tiếp cận. Có nghĩa là, chúng ta có thể định nghĩa và sử dụng các lớp cơ sở với cả trạng thái và chức năng ảo thuần túy. Tuy nhiên, những cách tiếp cận hỗn hợp như vậy có thể gây nhầm lẫn và cần phải cẩn thận hơn.**

**Với sự ra đời của các lớp trừu tượng, chúng ta có các phương tiện cơ bản để viết một chương trình hoàn chỉnh theo kiểu mô-đun sử dụng các lớp làm khối xây dựng.**



**600** **Các lớp có nguồn gốc** **Chương 20**

**20.5 Kiểm soát truy cập**

**Một thành viên của một lớp học có thể là riêng, được bảo vệ, hoặc công cộng:**

* **Nếu nó là riêng, 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 là được bảo vệ, tên của nó chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của lớp**

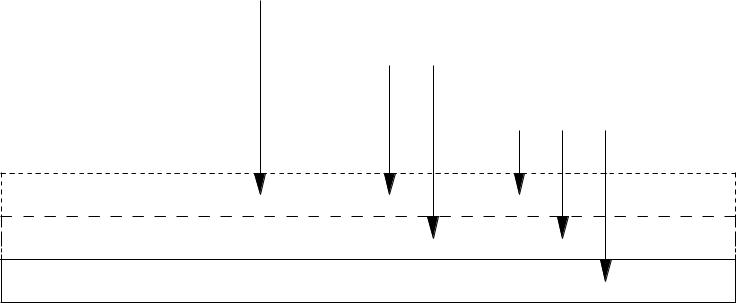
**mà nó được khai báo và bởi các hàm thành viên và bạn bè của các lớp dẫn xuất từ lớp này (xem §19.4).**

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

**Điều này phản ánh quan điểm rằng có ba loại hàm truy cập vào một lớp: các hàm thực thi lớp (bạn bè của nó và**

**bạn bè và thành viên), và o**

**người dùng chung**



**các hàm thành viên và bạn bè của lớp dẫn xuất**

**chức năng thành viên riêng và bạn bè**

**công cộng:**

**được bảo vệ:**

**riêng:**

**Kiểm soát truy cập được áp dụng thống nhất cho các tên. Những gì một tên đề cập đến không ảnh hưởng đến việc kiểm soát việc sử dụng nó. Điều này có nghĩa là chúng ta có thể córiêng hàm thành viên, kiểu, hằng số, v.v., cũng nhưriêng dữ liệu thành viên. Ví dụ, một lớp danh sách không ủy quyền hiệu quả thường yêu cầu cấu trúc dữ liệu để theo dõi các phần tử. Một danh sách là*không tin tưởng* nếu nó không yêu cầu sửa đổi các phần tử của nó (ví dụ: bằng cách yêu cầu phần tử**



**tổ chức danh sách có thể được giữ**

**mẫu <lớp T>**

**danh sách lớp học {**

**công cộng:**

**void insert (T);**

**T get ();**

* ***...***

**riêng:**

**struct Liên kết {T val; Liên kết∗ Kế tiếp; };**

**struct Chunk {**

**enum {chunk\_size = 15}; Liên**

**kết v [chunk\_size];**

**Chunk∗ Kế tiếp;**

**};**



**Mục 20.5** **Kiểm soát truy cập** **601**

**Chunk∗ phân bổ;**

**Liên kết∗ miễn phí;**

**Liên kết∗ dùng miễn phí();**

**Liên kết∗ cái đầu;**

**};**

**Các định nghĩa của các chức năng công cộng khá đơn giản:**

**mẫu <lớp T>**

**Danh sách void <T> :: insert (T val) {**

**Liên kết∗ lnk = get\_free**

**(lnk−> val = val;**

**lnk−> tiếp theo = đầu;**

**đầu = lnk;**

**}**

**mẫu <lớp T>**

**Danh sách T <T> :: get ()**

**{**

**if (head == 0)**

**ném Underflow {}; //*Underflow là lớp ngoại lệ của tôi***

**Liên kết∗ p = cái đầu;**

**đầu = p−> tiếp theo;**

**p−> next = miễn phí;**

**miễn phí = p;**

**return p−> val;**

**}**



**Như thường lệ, định nghĩa**

**mẫu <lớp T>**

**typename Danh sách <T> :: Liên kết∗ Danh sách <T> :: get\_free () {**

**if (miễn phí == 0) {**

* ***... phân bổ một đoạn mới và đặt các Liên kết của nó vào danh sách miễn phí ...***

**}**

**Liên kết∗ p = miễn phí; miễn**

**phí = free−> tiếp theo;**

**trả lại p;**

**}**

**Các Danh sách <T> phạm vi được nhập bằng cách nói Danh sách <T> :: trong một định nghĩa hàm thành viên. Tuy**

**nhiên, vì kiểu trả về củadùng miễn phí() được đề cập trước tên Danh sách <T> :: get\_free () được đề cập, tên đầy đủ Danh sách <T> :: Liên kết phải được sử dụng thay vì viết tắt Liên kết. Cách thay thế là sử dụng ký hiệu hậu tố cho các kiểu trả về (§12.1.4):**



**602** **Các lớp có nguồn gốc** **Chương 20**

**mẫu <lớp T>**

**Danh sách tự động <T> :: get\_free () -> Liên kết∗{**

* ***...***

**}**

**Các chức năng nonmember (ngoại trừ bạn bè) không có quyền truy cập như vậy:**

**mẫu <typename T>**

**void would\_be\_meddler (Danh sách <T>∗ P) {**

**Danh sách <T> :: Liên kết∗ q =**

**0; //*...***

**q = p−> tự do;**

* ***...***

**if (Danh sách <T> :: Chunk :: chunk\_size> 31) {** **// *lỗi: Danh sách <T> :: Chunk :: chunk\_size là riêng tư***

* ***...***

**}**

**}**

**Trong một lớp, các thành viên theo mặc định riêng; trong mộtcấu trúc, các thành viên theo mặc định công cộng (§16.2.4).**

**Cách thay thế rõ ràng cho việc sử dụng kiểu thành viên là đặt kiểu đó trong không gian tên xung quanh. Ví dụ:**

**mẫu <lớp T>**

**struct Link2 {**

**T val;**

**Link2∗ Kế tiếp;**

**};**

**mẫu <lớp T>**



**danh sách lớp học {**

**riêng:**

**Link2 <T>∗ miễn phí;**

* ***...***

**};**

**Liên kết được tham số hóa ngầm định với Danh sách <T>tham số của NS. VìLink2, chúng ta phải làm cho điều đó rõ ràng. Nếu một kiểu thành viên không phụ thuộc vào tất cả các tham số của lớp mẫu, thì phiên bản không phải là thành viên có**

**thể thích hợp hơn; xem §23.4.6.3.**

**Nếu lớp lồng nhau nói chung không hữu ích cho chính nó và lớp bao quanh cần quyền truy cập vào biểu diễn của nó, hãy khai báo lớp thành viên a bạn bè (§19.4.2) có thể là một ý tưởng hay:**

**template <class T> Danh sách lớp;**

**mẫu <lớp T>**

**liên kết lớp3 {**

**Danh sách bạn bè của lớp <T>;** **// *chỉ Danh sách <T> mới có thể truy cập Liên kết <T>***

**T val;**

**Liên kết3∗ Kế tiếp;**

**};**



**Mục 20.5** **Kiểm soát truy cập** **603**

**mẫu <lớp T>**

**danh sách lớp học {**

**riêng:**

**Liên kết3 <T>∗ miễn phí;**

* ***...***

**};**

**Một trình biên dịch có thể sắp xếp lại các phần của một lớp với các chỉ định truy cập riêng biệt (§8.2.6). Ví dụ:**

**lớp S {**

**công cộng:**

**int m1;**

**công cộng:**

**int m2;**

**};**

**Trình biên dịch có thể quyết định m2 đi trước m1 trong bố cục của một NS sự vật. Việc sắp xếp lại thứ tự như vậy có thể gây ngạc nhiên cho lập trình viên và phụ thuộc vào việc triển khai, vì vậy đừng sử dụng các bộ chỉ định nhiều truy cập cho các thành viên dữ liệu mà không có lý do chính đáng.**

**20.5.1 được bảo vệ Các thành viên**

**Khi thiết kế hệ thống phân cấp lớp, đôi khi chúng tôi cung cấp các hàm được thiết kế để được sử dụng bởi người triển khai các lớp dẫn xuất mà không phải bởi người dùng chung. Ví dụ: chúng tôi có thể cung cấp chức năng truy cập không được kiểm tra (hiệu quả) cho những người triển khai lớp dẫn xuất và quyền truy cập đã kiểm tra (an toàn) cho những người khác. Khai báo phiên bản không được kiểm trađược bảo vệ đạt được điều đó. Ví dụ:**

**bộ đệm lớp {**

**công cộng:**

**char & operator [] (int i); //** **// *đã kiểm tra quyền truy cập***



***...***

**được bảo vệ:**

**char & access (int i);**

* ***...***

**};**

**class Circular\_buffer: public Buffer**

**{public:**

**void phân bổ lại (char∗ p, int** **// *thay đổi vị trí và kích thước***

**s); //*...***

**};**

**void Circular\_buffer :: reallocate (char∗ p, int s) // *thay đổi vị trí và kích thước*{**

**// *...***

**for (int i = 0; i! = old\_sz; ++ i)**

**p [i] = access (i);** **// *không kiểm tra dư thừa***

**// *...***

**}**



**604** **Các lớp có nguồn gốc** **Chương 20**

**void f (Bộ đệm & b)**

**{**

**b [3] = 'b';**

**// *OK (đã kiểm tra)***

**b.access (3) = 'c';**

**// *lỗi: Buffer :: access () được bảo vệ***

**}**

**Ví dụ khác, hãy xem Window\_with\_border trong §21.3.5.2.**

**Một lớp dẫn xuất chỉ có thể truy cập các thành viên được bảo vệ của lớp cơ sở đối với các đối tượng thuộc kiểu riêng của nó:**

**bộ đệm lớp {**

**được bảo vệ:**

**char a [128];**

* ***...***

**};**

**class Linked\_buffer: public Buffer {**

* ***...***

**};**

**class Circular\_buffer: public Buffer {**

* ***...***

**void f (Linked\_buffer∗ P) {**

**a [0] = 0;**

**p−> a [0] = 0;**

**// *OK: truy cập vào thành viên được bảo vệ của chính Circular\_buffer*// *lỗi:***

***quyền truy cập vào thành viên được bảo vệ thuộc loại khác nhau***

**}**

**};**

**Điều này ngăn ngừa các lỗi nhỏ có thể xảy ra khi một lớp dẫn xuất làm hỏng dữ liệu thuộc các lớp dẫn xuất khác.**



**20.5.1.1 Sử dụng được bảo vệ**

**Mô hình ẩn dữ liệu riêng tư / công khai đơn giản phục vụ tốt khái niệm về các kiểu cụ thể (§16.3). Tuy nhiên, khi các lớp dẫn xuất được sử dụng, có hai loại người dùng của một lớp: các lớp dẫn xuất và '' công chúng ''. . Mô hình private / public cho phép lập trình viên phân biệt rõ ràng giữa những người triển khai và công chúng, nhưng nó không cung cấp một cách cung cấp cụ thể cho các lớp dẫn xuất.**

**Các thành viên đã tuyên bố được bảo vệ cởi mở hơn với lạm dụng so với các thành viên đã tuyên bố riêng. Ngang bằng-**

**ticular, khai báo các thành viên dữ liệu được bảo vệ thường là một lỗi thiết kế. Việc đặt một lượng lớn dữ liệu trong một lớp chung để tất cả các lớp dẫn xuất sử dụng sẽ khiến dữ liệu đó dễ bị hỏng. Tệ hơn nữa, dữ liệu được bảo vệ, như dữ liệu công khai, không thể dễ dàng được cấu trúc lại vì không có cách nào tốt để tìm mọi cách sử dụng. Do đó, dữ liệu được bảo vệ trở thành một vấn đề bảo trì phần mềm.**

**May mắn thay, bạn không phải sử dụng dữ liệu được bảo vệ; riêng là mặc định trong các lớp và thường là lựa chọn tốt hơn. Theo kinh nghiệm của tôi, luôn có những lựa chọn thay thế cho việc đặt một lượng thông tin đáng kể trong một lớp cơ sở chung để các lớp dẫn xuất sử dụng trực tiếp.**



**Mục 20.5.1.1** **Sử dụng được bảo vệ Các thành viên** **605**

**Tuy nhiên, không có phản đối nào trong số này là đáng kể đối với thành viên được bảo vệ *chức* *năng*;được bảo vệlà một cách tốt để chỉ định các hoạt động để sử dụng trong các lớp dẫn xuất. CácIval\_slider trong §21.2.2 là một ví dụ về điều này. Đã có lớp triển khairiêng trong ví dụ này, việc dẫn xuất thêm sẽ không khả thi. Mặt khác, làm cơ sở cung cấp chi tiết thực hiệncông cộngmời những sai lầm và sử dụng sai.**

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

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

**lớp X: công khai B {/ \* *...* \* /};**

**lớp Y: được bảo vệ B {/ \**.*lớp**

**Z: riêng tư B {/ \* *...* \***

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

* **công cộng sự dẫn xuất làm cho lớp dẫn xuất trở thành một kiểu con của cơ sở của nó. Ví dụ,NS Là một loại NS. Đây là hình thức dẫn xuất phổ biến nhất.**
* **riêng base hữu ích nhất khi xác định một lớp bằng cách giới hạn giao diện cho một base**

**để có thể cung cấp các đảm bảo mạnh mẽ hơn. Ví dụ,NS là một chi tiết triển khai của Z. CácVéc tơ mẫu con trỏ bổ sung kiểm tra kiểu vào Véc tơ <void∗> cơ sở từ §25.3 là một ví dụ điển hình.**

* **được bảo vệ các base rất hữu ích trong phân cấp lớp trong đó dẫn xuất thêm là tiêu chuẩn. Nhưriêng nguồn gốc, được bảo vệ dẫn xuất được sử dụng để biểu diễn các chi tiết thực hiện. CácIval\_slider từ §21.2.2 là một ví dụ điển hình.**

**Có thể bỏ qua thông số truy cập cho một lớp cơ sở. Trong trường hợp đó, cơ sở mặc định là cơ sở riêng cho mộtlớp và một cơ sở công cộng cho một cấu trúc. Ví dụ:**

**lớp XX: B {/ \* *...* \* /}; cấu** **// *B là cơ sở tư nhân*// *B***

**trúc YY: B {/ \**...* \* /};** ***là cơ sở công cộng***



**Mọi người mong đợi các lớp cơ sở là**

**một chỉ định truy cập cho một cơ sở**

**Mã xác định quyền truy cập cho**

**chuyển đổi con trỏ và tham chiếu từ kiểu lớp dẫn xuất sang kiểu lớp cơ sở. Xem xét một lớp họcNS bắt nguồn từ một lớp cơ sở NS:**

* **Nếu như NS là một riêng cơ sở, các thành viên công khai và được bảo vệ của nó chỉ có thể được sử dụng bởi các chức năng thành viên và bạn bè của NS. Chỉ bạn bè và thành viên củaNS có thể chuyển đổi một NS∗ đến một NS∗.**
* **Nếu như NS là một được bảo vệ cơ sở, các thành viên công khai và được bảo vệ của nó chỉ có thể được sử dụng bởi các chức năng thành viên và bạn bè của NS và bởi các hàm thành viên và bạn bè của các lớp bắt nguồn từ NS. Chỉ bạn**

**bè và thành viên củaNS và bạn bè và thành viên của các lớp học bắt nguồn từ NS có thể chuyển đổi một**

**NS∗ đến một NS∗.**

* **Nếu như NS là một công cộng cơ sở, các thành viên công khai của nó có thể được sử dụng bởi bất kỳ chức năng nào. Ngoài ra, các thành viên được bảo vệ của nó có thể được sử dụng bởi các thành viên và bạn bè củaNS và các thành viên và bạn bè của các lớp học bắt nguồn từ NS. Bất kỳ hàm nào cũng có thể chuyển đổi mộtNS∗ đến một NS∗.**

**Điều này về cơ bản trình bày lại các quy tắc cho quyền truy cập của thành viên (§20.5). Khi thiết kế một lớp, chúng tôi chọn quyền truy cập cho các cơ sở giống như cách chúng tôi làm cho các thành viên. Ví dụ, hãy xemIval\_slider trong §21.2.2.**



**606** **Các lớp có nguồn gốc** **Chương 20**

**20.5.2.1 Kiểm soát nhiều quyền thừa kế và truy cập**

**Nếu tên của một lớp cơ sở có thể được truy cập thông qua nhiều đường dẫn trong mạng đa kế thừa (§21.3), thì nó có thể truy cập được nếu có thể truy cập thông qua bất kỳ đường dẫn nào. Ví dụ:**

**cấu trúc B {**

**int m;**

**tĩnh int sm;**

* ***...***

**};**

**lớp D1: công cộng ảo B lớp**

**D2: công cộng ảo B lớp D12:**

**công cộng D1, riêng tư**

**D12∗ pd = mới D12;**

**NS∗ pb = pd;** **// *OK: có thể truy cập thông qua D1***

**int i1 = pd−> m;** **// *OK: có thể truy cập thông qua D1***

**Nếu một thực thể duy nhất có thể truy cập được thông qua một số đường dẫn, chúng tôi vẫn có thể tham chiếu đến nó mà không bị mơ hồ. Ví dụ:**

**lớp X1: công khai B {/ \* *...* \* /}; lớp X2: công**

**khai B {/ \**...* \* /}; lớp XX: công khai X1, công**

**khai X2 {/ \**...* \* /};**

**XX∗ pxx = XX mới;**

**int i1 = pxx−> m; int** **// *lỗi, không rõ ràng: XX :: X1 :: B :: m hoặc XX :: X2 :: B :: m?*// *OK: chỉ***

**i2 = pxx−> sm;** ***có một B :: sm trong XX (sm là thành viên tĩnh)***

**20.5.3 sử dụng-Khai báo và Kiểm soát truy cập**



**MỘT sử dụng-dec Tuyên bố (§14.2.**

**chỉ đơn giản là một cơ chế cho ma**

**khi có quyền truy cập, nó có thể được cấp cho những người dùng khác. Ví dụ:**

**lớp B {**

**riêng:**

**int a;**

**được bảo vệ:**

**int b;**

**công cộng:**

**int c;**

**};**

**class D: public B**

**{public:**

**sử dụng B :: a;** **// *lỗi: B :: a là riêng tư*// *làm cho B :: b có sẵn***

**sử dụng B :: b;** ***công khai thông qua D***

**};**



**Mục 20.5.3** **sử dụng-Khai báo và Kiểm soát truy cập** **607**

**Khi một sử dụng- khai báo được kết hợp với dẫn xuất riêng tư hoặc được bảo vệ, nó có thể được sử dụng để chỉ định giao diện cho một số, nhưng không phải tất cả, của các cơ sở thường được cung cấp bởi một lớp. Ví dụ:**

**lớp BB: private B**

**{public:**

* ***cấp quyền truy cập cho B :: b và B :: c, nhưng không cấp cho B :: a***

**sử dụng B :: b;**

**sử dụng B :: c;**

**};**

**Xem thêm §20.3.5.**

**20.6 Con trỏ tới tôi**

**Con trỏ đến thành viên là một cấu trúc giống như bù đắp cho phép lập trình viên tham chiếu gián tiếp đến thành viên của một lớp. Các nhà khai thác->∗ và .∗ được cho là các toán tử C ++ chuyên dụng nhất và ít được sử dụng nhất. Sử dụng->, chúng tôi có thể truy cập một thành viên của một lớp học, NS, bằng cách đặt tên cho nó: p−> m. Sử dụng->∗, chúng tôi có thể truy cập một thành viên (về mặt khái niệm) có tên được lưu trữ trong một con trỏ tới thành viên, ptom: p−>∗ptom. Điều này cho phép chúng tôi truy cập các thành viên với tên của họ được chuyển làm đối số. Trong cả hai trường hợp,P phải là một con trỏ đến một đối tượng của một lớp thích hợp.**

**Một con trỏ đến thành viên không thể được chỉ định cho một vô hiệu∗ hoặc bất kỳ con trỏ thông thường nào khác. Một con trỏ null (ví dụ:nullptr) có thể được gán cho một con trỏ tới thành viên và sau đó đại diện cho '' không có thành viên ''.**

**20.6.1 Con trỏ đến các thành viên hàm**

**Nhiều lớp cung cấp các giao diện đơn giản, rất chung chung nhằm mục đích được gọi theo một số cách khác nhau. Ví dụ: nhiều giao diện người dùng '' hướng đối tượng '' xác định một tập hợp các yêu cầu mà mọi đối tượng được trình bày trên màn hình phải được chuẩn bị để đáp ứng. Ngoài ra, các yêu cầu đó có thể được trình bày trực tiếp hoặc gián tiếp từ các chương trình. Hãy xem xét một biến thể đơn giản của ý tưởng này:**



**lớp Std\_interface {**

**công cộng:**

**ảo void start () = virtual**

**void pause () = 0; ảo void**

**resume () = 0; void ảo bỏ ()**

* **0; void ảo full\_size () = 0; ảo void small () = 0;**

**ảo ˜Std\_interface () {}**

**};**

* **nghĩa chính xác của mỗi thao tác được xác định bởi đối tượng mà nó được gọi. Thông thường, có một lớp phần mềm giữa người hoặc chương trình đưa ra yêu cầu và đối tượng nhận nó. Lý tưởng nhất là các lớp phần mềm trung gian như vậy không cần phải biết bất cứ điều gì về các hoạt động riêng lẻ nhưbản tóm tắt() và full\_size (). Nếu có, các lớp trung gian sẽ phải được cập nhật mỗi khi một hoạt động thay đổi. Do đó, các lớp trung gian như vậy chỉ đơn giản truyền dữ liệu đại diện cho hoạt động được gọi từ nguồn của yêu cầu đến người nhận của nó.**



**608** **Các lớp có nguồn gốc** **Chương 20**

**Một cách đơn giản để làm điều đó là gửi một dây đại diện cho hoạt động được gọi. Ví dụ, để gọi đình chỉ() chúng tôi có thể gửi chuỗi "đình chỉ". Tuy nhiên, ai đó phải tạo chuỗi đó và ai đó phải giải mã nó để xác định nó tương ứng với hoạt động nào - nếu có. Thông thường, điều đó có vẻ gián tiếp và tẻ nhạt. Thay vào đó, chúng tôi có thể chỉ cần gửi một số nguyên đại diện cho hoạt động. Ví dụ,2 có thể được sử dụng để có nghĩa là đình chỉ(). Tuy nhiên, trong khi một số nguyên có thể thuận tiện cho máy móc xử lý, nó có thể khá khó hiểu đối với con người. Chúng tôi vẫn phải viết mã để xác định rằng2 có nghĩa đình chỉ() và để gọi đình chỉ().**

**Tuy nhiên, chúng ta có thể sử dụng một con trỏ tới thành viên để gián tiếp tham chiếu đến thành viên của một lớp. Xem xét**

**Std\_interface. Nếu tôi muốn gọiđình chỉ() cho một số đối tượng mà không đề cập đến đình chỉ() trực tiếp, tôi cần một con trỏ đến thành viên đề cập đến Std\_interface :: pause (). Tôi cũng cần một con trỏ hoặc tham chiếu đến đối tượng mà tôi muốn tạm dừng**

**sử dụng Pstd\_mem = void (Std\_interface ::∗) ();**

**// *kiểu con trỏ đến thành viên***

**void f (Std\_interface∗ P) {**

**Pstd\_mem s = & Std\_interface :: tạm ngưng;**

**p−> Susan ();**

**p−>∗NS();**

**// *con trỏ để tạm ngưng ()*//**

***gọi trực tiếp***

**// *gọi thông qua con trỏ đến thành viên***

**}**

**MỘT *con trỏ đến thành viên* có thể đạt được bằng cách áp dụng toán tử address-of, &, với tên thành viên lớp đủ điều kiện, chẳng hạn, &Std\_interface :: tạm ngưng. Một biến kiểu '' con trỏ đến thành viên của lớpNS'' được khai báo bằng cách sử dụng một bộ khai báo của biểu mẫu NS::∗.**

**Việc sử dụng bí danh để bù đắp cho việc thiếu khả năng đọc của cú pháp bộ khai báo C là điển hình.**

**Tuy nhiên, hãy lưu ý cáchNS::∗ bộ khai báo khớp với truyền thống ∗ người khai báo chính xác.**

**Một con trỏ đến thành viên NS có thể được sử dụng kết hợp với một đối tượng. Các nhà khai thác->∗ và .∗**

**cho phép người lập trình thể hiện các kết hợp như vậy. Ví dụ,p−>∗NS ràng buộc NS đến đối tượng được trỏ tới P, và phản đối.∗NS ràng buộc NS đối tượng phản đối. Kết quả có thể được sử dụng phù hợp vớiNS của loại. Không thể st**

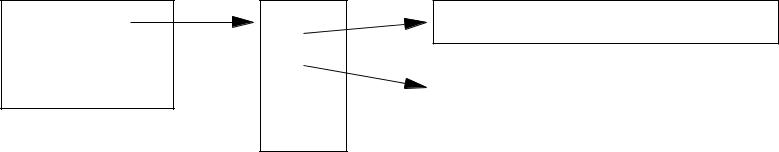


**Đương nhiên, nếu chúng ta biết w**

**lộn xộn với các con trỏ đến các thành viên. Cũng giống như các con trỏ tới các hàm thông thường, các con trỏ tới các hàm thành viên được sử dụng khi chúng ta cần tham chiếu đến một hàm mà không cần biết tên của nó. Tuy nhiên, một con trỏ tới thành viên không phải là một con trỏ tới một phần bộ nhớ như cách một con trỏ tới một biến hoặc một con trỏ đến một hàm. Nó giống như một sự bù đắp vào một cấu trúc hoặc một chỉ mục vào một mảng, nhưng tất nhiên việc triển khai có tính đến sự khác biệt giữa các thành viên dữ liệu, các hàm ảo, các hàm không ảo, v.v. Khi một con trỏ tới thành viên được kết hợp với một con trỏ tới một đối tượng của đúng loại, nó mang lại một cái gì đó xác định một thành viên cụ thể của một đối tượng cụ thể.**

**Các p−>∗NS() cuộc gọi có thể được biểu diễn bằng đồ thị như sau:**

**vtbl:**



|  |  |
| --- | --- |
| **P** | **X :: bắt đầu** |



**NS:**

**X :: tạm ngưng**



**Mục 20.6.1** **Con trỏ đến các thành viên hàm** **609**

**Bởi vì một con trỏ đến một thành viên ảo (NS trong ví dụ này) là một loại bù đắp, nó không phụ thuộc vào vị trí của đối tượng trong bộ nhớ. Do đó, một con trỏ tới một thành viên ảo có thể được chuyển giữa các không gian địa chỉ khác nhau miễn là sử dụng cùng một bố cục đối tượng trong cả hai. Giống như con trỏ tới các hàm thông thường,**

**con trỏ tới các hàm thành viên không phải là ảo không thể được trao đổi giữa các không gian địa chỉ. Lưu ý rằng hàm được gọi thông qua con trỏ tới hàm có thể ảo. Ví dụ,**

**khi chúng tôi gọi đình chỉ() thông qua một con trỏ để hoạt động, chúng tôi có quyền đình chỉ() cho đối tượng mà con trỏ tới chức năng được áp dụng. Đây là một khía cạnh thiết yếu của con trỏ đến các hàm.**

**Khi viết một trình thông dịch, chúng ta có thể sử dụng con trỏ tới các thành viên để gọi các hàm được trình bày dưới dạng chuỗi:**

**bản đồ <string, Std\_interface**

**map <string, Pstd\_mem> o**

**void call\_member (string var, string oper) {**

**(biến [var] ->∗hoạt động [oper]) (); //*var.oper ()***

**}**

**MỘT tĩnh thành viên không được liên kết với một đối tượng cụ thể, vì vậy một con trỏ đến một tĩnh thành viên chỉ đơn giản là một con trỏ bình thường. Ví dụ:**

**Nhiệm vụ lớp {**

**// *...***

**lịch void tĩnh ();**

**};**

**vô hiệu (∗p) () = & Task :: lịch trình; void (Tác**

**// *VÂNG***

**vụ ::∗ pm) () = & Task :: lịch trình;**

**// *lỗi: con trỏ thông thường được chỉ***

***định*// *trỏ tới thành viên***

**Con trỏ đến các thành viên dữ liệu được mô tả trong §20.6.2.**



**20.6.2 Con trỏ tới Đạt**

**Đương nhiên, khái niệm con trỏ tới thành viên áp dụng cho các thành viên dữ liệu và cho các hàm thành viên với các đối số và kiểu trả về. Ví dụ:**

**struct C {**

**const char∗ valy;**

**int i;**

**void print (int x) {cout << val << x << '\ n'; } int**

**f1 (int);**

**void f2 ();**

**C (const char∗ v) {val = v; }**

**};**

**sử dụng Pmfi = void (C ::∗)(NS); sử**

**// *con trỏ đến hàm thành viên của C lấy một int*// *con***

**dụng Pm = const char∗ NS::∗;**

***trỏ tới thành viên dữ liệu char \* của C***



**610** **Các lớp có nguồn gốc** **Chương 20**

**void f (C & z1, C & z2) {**

**NS∗ p = & z2;**

**Pmfi pf = & C :: print;**

**Chiều pm = & C :: val;**

**z1.print (1);**

**(z1.∗pf) (2);**

**z1.∗pm = "nv1"; p**

**−>∗pm = "nv2";**

**z2.print (3);**

**(p−>∗pf) (4);**

**pf = & C :: f1;**

**pf = & C :: f2;**

**pm = & C :: i;**

**pm = pf;**

**// *lỗi: kiểu trả về không khớp*//**

***lỗi: loại đối số không khớp*// *lỗi:***

***nhập không khớp*// *lỗi: nhập***

***không khớp***

**}**

**Loại con trỏ tới hàm được kiểm tra giống như bất kỳ loại nào khác.**

**20.6.3 Các thành viên cơ sở và có nguồn gốc**

**Một lớp dẫn xuất có ít nhất các thành viên mà nó kế thừa từ các lớp cơ sở của nó. Thường thì nó có nhiều hơn. Điều này ngụ ý rằng chúng ta có thể chỉ định một cách an toàn một con trỏ cho một thành viên của lớp cơ sở cho một con trỏ tới một thành viên của lớp dẫn xuất, nhưng không phải ngược lại. Thuộc tính này thường được gọi là*sự trái ngược*. Ví dụ:**

**class Text: public Std\_interface**

**{public:**

**void start ();**



**void đình chỉ ();**

**// *...***

**void in ảo ();**

**riêng:**

**vectơ s;**

**};**

**void (Std\_interface ::∗ pmi) () = & Text :: print; void**

**// *lỗi***

**(Văn bản ::∗pmt) () = & Std\_interface :: start;**

**// *VÂNG***

**Quy tắc tương phản này dường như ngược lại với quy tắc nói rằng chúng ta có thể gán một con trỏ cho một lớp dẫn xuất cho một con trỏ đến lớp cơ sở của nó. Trên thực tế, cả hai quy tắc tồn tại để bảo đảm cơ bản rằng một con trỏ có thể không bao giờ trỏ đến một đối tượng ít nhất không có các thuộc tính mà con trỏ hứa hẹn. Trong trường hợp này,Std\_interface ::∗ có thể được áp dụng cho bất kỳ Std\_interfacevà hầu hết các đối tượng như vậy có lẽ không thuộc loại Chữ. Do đó, họ không có thành viênVăn bản :: in mà chúng tôi đã cố gắng khởi tạo pmi. Bằng cách từ chối khởi tạo, trình biên dịch giúp chúng ta tránh khỏi lỗi thời gian chạy.**



**Mục 20.7** **Lời khuyên** **611**

**20.7 Lời khuyên**

1. **Tránh các trường loại; §20.3.1.**
2. **Truy cập các đối tượng đa hình thông qua con trỏ và tham chiếu; §20.3.2.**
3. **Sử dụng các lớp trừu tượng để tập trung thiết kế vào việc cung cấp các giao diện sạch; §20.4.**
4. **Sử dụng ghi đè để làm cho việc ghi đè rõ ràng trong các cấu trúc phân cấp lớp lớn; §20.3.4.1.**
5. **Sử dụng final chỉ một cách tiết kiệm; §20.3.4.2.**
6. **Sử dụng các lớp trừu tượng để chỉ định giao diện; §20.4.**
7. **Sử dụng các lớp trừu tượng để giữ các chi tiết triển khai nằm ngoài giao diện; §20.4.**
8. **Một lớp có hàm ảo nên có hàm hủy ảo; §20.4.**
9. **Một typi lớp trừu tượng**
10. **Thích riêng membe**
11. **Thích công cộng thành viên cho các giao diện; §20.5.**
12. **Sử dụng được bảo vệ thành viên chỉ cẩn thận khi thực sự cần thiết; §20.5.1.1.**
13. **Không khai báo thành viên dữ liệu được bảo vệ; §20.5.1.1.**



***Trang này cố ý để trống***



**21**



***Trừu tượng là sự thiếu hiểu biết có chọn lọc.***

***- Andrew Koenig***

* **Giới thiệu**
* **Thiết kế cấu trúc phân cấp lớp**

**Kế thừa thực hiện; Kế thừa giao diện; Triển khai Thay thế; Bản địa hóa tạo đối tượng**

**• Nhiều người thừa kế**

**Nhiều giao diện; Nhiều lớp thực hiện; Độ phân giải mơ hồ; Sử dụng lặp lại một lớp cơ sở; Các lớp cơ sở ảo; Cơ sở sao chép so với Cơ sở ảo**

**• Lời khuyên**



**21.1 Giới thiệu**

**Trọng tâm chính của chương này là các kỹ thuật thiết kế, hơn là các tính năng ngôn ngữ. Các ví dụ được lấy từ thiết kế giao diện người dùng, nhưng tôi tránh chủ đề về lập trình hướng sự kiện như 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 rõ về GUI, hãy xem một trong nhiều thư viện C ++ GUI.**

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

**Hãy xem xét một bài toán thiết kế đơn giản: Cung cấp một cách để một chương trình ('' một ứng dụng '') nhận một giá trị nguyên từ người dùng. Điều này có thể được thực hiện theo nhiều cách khác nhau. Để cách ly chương trình của chúng ta 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 ta về thao tác nhập đơn giản này.**



**614** **Cấu trúc phân cấp lớp** **Chương 21**

* **tưởng là có một lớp học Ival\_box (''hộp nhập giá trị số nguyên '') biết phạm vi giá trị đầu vào mà nó sẽ chấp nhận. Một chương trình có thể yêu cầu mộtIval\_box cho giá trị của nó và yêu cầu nó nhắc người dùng nếu cần. Ngoài ra, một chương trình có thể yêu cầu mộtIval\_box nếu người dùng thay đổi giá trị kể từ lần cuối chương trình xem xét nó:**

**người sử dụng**

**(thông qua '' hệ thống '')**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Ival\_box:** | | |  |  |  |  |
|  |  |  |  |  | **ứng dụng** |  |
|  |  | **giá trị** |  |
|  |  |  |  |
| **đặt giá trị()** | |  |  | **get\_value ()** | |  |
|  |  |



**Bởi vì có nhiều cách để thực hiện ý tưởng cơ bản này, chúng ta phải giả định rằng sẽ có nhiều loại khác nhau Ivalquay 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 các hệ thống giao diện người dùng '' thực '' và - quan trọng nhất - bởi vì những kỹ thuật có thể áp dụng cho các vấn đề vượt xa phạm vi hẹp của hệ thống giao diện.**

**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 cấu trúc phân cấp lớp sử dụng kế thừa triển khai (như thường thấy trong các chương trình cũ hơn).**

**Lớp Ival\_box xác định giao diện cơ bản cho tất cả Ival\_boxes và chỉ định một triển khai mặc định mà các loại cụ thể hơn**



**dữ liệu cần thiết để xây dựng**

**lớp Ival\_box {**

**được bảo vệ:**

**int val;**

**int thấp, cao;**

**bool đã thay đổi {false};** **// *được thay đổi bởi người dùng bằng set\_value ()***

**công cộng:**

**Ival\_box (int ll, int hh): val {ll}, thấp {ll}, cao {hh} {}**

**virtual int get\_value () {đã thay đổi = false; trả lại val; }** **// *cho ứng dụng***

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

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

**dấu nhắc void ảo () {}**

**virtual bool was\_changed () const {return đã thay đổi; }**

**ảo ˜Ival\_box () {};**

**};**



**Mục 21.2.1** **Kế thừa triển khai** **615**

**Việc triển khai mặc định của các chức năng là khá cẩu thả và được cung cấp ở đây chủ yếu để minh họa 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 tương tác (Ival\_box∗ pb) {**

**pb−> prompt (); //*cảnh báo người dùng*//**

***...***

**int i = pb−> get\_value (); if**

**(pb−> was\_changed ()) {**

* ***... giá trị mới; làm việc gì đó ...***

**}**

**khác {**

**// *... làm việc gì khác ...***

**}**

**}**

**void some\_fct ()**

**{**

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

**tương tác (p1.get ());**

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

**tương tác (p2.get ());**

**}**

**Hầu hết mã ứng dụng được viết dưới dạng (con trỏ đến) đơn giản Ival\_boxes the way tương tác() Là. Bằng cách đó, ứng dụng không cần phải biết về số lượng lớn các biến thể tiềm ẩn củaIval\_box ý tưởng. Kiến thức của các lớp chuyên biệt như vậy bị cô lập trong tương đối ít chức năng 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 loạiIval\_boxes.**



**tôi sử dụng unique\_ptr (§5.2.1,**

**để đơn giản hóa thảo luận**

**chương trình thực sự đợi người dùng trong get\_value () (ví dụ, sử dụng một hiểu được() trên một Tương lai; §5.3.5.1), có thể chương trình liên kếtIval\_box với một sự kiện và chuẩn bị trả lời một cuộc gọi lại hoặc có thể chương trình tạo ra một chuỗi cho Ival\_box và sau đó 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 khác nhau của Ival\_boxes được định nghĩa là các lớp có nguồn gốc từ Ival\_box. Ví dụ:**

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

**{private:**

* ***... nội dung đồ họa để xác định thanh trượt trông như thế nào, v.v. ...***

**công cộng:**

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



**616** **Cấu trúc phân cấp lớp** **Chương 21**

**int get\_value () ghi đè; //*nhận giá trị từ người dùng và gửi nó vào val*ghi đè void prompt ();**

**};**

**Các thành viên dữ liệu của Ival\_box đã được tuyên bố được bảo vệ để cho phép truy cập từ các lớp dẫn xuất. Vì vậy, Ival\_slider :: get\_value () có thể gửi một giá trị vào Ival\_box :: val. MỘTđược bảo vệ thành viên có thể truy cập từ một**

**các thành viên riêng của lớp và từ các thành viên của các lớp dẫn xuất, nhưng không tới người dùng chung (xem §20.5).**

**Ngoài Ival\_slider, chúng tôi sẽ xác định các biến thể khác của Ival\_box ý tưởng. Chúng có thể bao gồmIval\_dial, cho phép bạn chọn một giá trị bằng cách xoay một núm; Flashing\_ival\_slider, sẽ nhấp nháy khi bạn yêu cầu lời nhắc(); vàPopup\_ival\_slider, đáp ứng với lời nhắc() bằng cách xuất hiện ở một số nơi nổi bật, do đó khiến người dùng khó có thể bỏ qua.**

**Từ đâu, chúng ta sẽ xác**

**định đúng đối tượng cơ bản**

**Bucks Inc., '' chúng tôi sẽ phải làm cho mỗi Ival\_slider, Ival\_dial, v.v., các lớp một loại BBwidget. Điều này đơn giản nhất sẽ đạt được bằng cách viết lạiIval\_box để nó bắt nguồn từ BBwidget. Theo cách đó, tất cả các lớp của chúng ta kế thừa tất cả các thuộc tính củaBBwidget. Ví dụ, mọiIval\_box có thể được đặt trên màn hình, tuân theo các quy tắc về kiểu đồ họa, được thay đổi kích thước, được kéo xung quanh, v.v., theo tiêu chuẩn được thiết lập bởi BBwidget hệ thống. Hệ thống phân cấp lớp của chúng ta sẽ trông như thế này:**

**class Ival\_box: public BBwidget {/ \* *...* \* /};** **// *được viết lại 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**



**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 trang bị thêm BBwidget là cơ sở của Ival\_box. Điều này không hoàn toàn đúng (ngay cả khi phong cách này là**

**mon trong hệ thống thế giới thực). Việc sử dụngBBwidget không phải là một phần trong khái niệm cơ bản của chúng tôi về một Ival\_box; nó là một chi tiết thực hiện. Phát sinhIval\_box từ BBwidget đã nâng chi tiết triển khai 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 khaiIval\_boxes cho các hệ thống từ '' Imperial**



**Mục 21.2.1.1** **Phê bình** **617**

**Chuối, '' '' Phần mềm được giải phóng, '' và '' Tiếng kêu của trình biên dịch ''? Chúng tôi sẽ phải duy trì bốn phiên bản khác nhau của chương trình của mình:**

|  |  |  |
| --- | --- | --- |
| **class Ival\_box: public BBwidget {/ \* *...*** | **\* /};** | **// *Phiên bản BB*** |
| **class Ival\_box: public CWwidget {/ \**...*** | **\* /};** | **// *Phiên bản CW*** |
| **lớp Ival\_box: IBwidget công khai {/ \**...*** | **\* /};** | **// *Phiên bản IB*** |
| **class Ival\_box: public LSwindow {/ \**...*** | **\* /};** | **// *Phiên bản LS*** |

**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**

**ảnh hưởng đến hệ thống phân cấp lớp của chúng ta de**

**Một vấn đề khác là mọi lớp dẫn xuất đều chia sẻ dữ liệu cơ bản được khai báo trong Ival\_box. Dữ liệu đó**

**tất nhiên là một chi tiết triển khai cũng len lỏi vào Ival\_box giao diện. Từ quan điểm thực tế, nó cũng là dữ liệu sai trong nhiều trường hợp. Ví dụ, mộtIval\_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 hiệnget\_value (). Nói chung, việc giữ hai bộ dữ liệu có liên quan, nhưng khác nhau, sẽ gặp rắc rối. Không sớm thì muộn ai đó sẽ khiến chúng không đồng bộ. Ngoài ra, kinh nghiệm cho thấy rằng các lập trình viên mới vào nghề 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 tốt hơn nên được giữ kín 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 của việc làm cho các cơ sở được cung cấp bởi BBwidget có sẵn cho người dùng của Ival\_box. Thật không may, nó cũng có nghĩa là thay đổi đối với lớpBBwidget có thể buộc người dùng phải biên dịch lại hoặc thậm chí viết lại mã của họ để khôi phục từ những thay đổi như vậy. Đặc biệt, cách hoạt động của hầu hết các triển khai C ++ ngụ ý rằng sự thay đổi kích thước của lớp cơ sở yêu cầu biên dịch lại tất cả các lớp dẫn xuất.**



**Cuối cùng, chương trình m hệ thống**

**giao diện người dùng của chúng tôi coex**

**hoặc vì chương trình của chúng tôi cần giao tiếp với người dùng trên các hệ thống khác nhau. Có hệ thống giao diện người dùng của chúng tôi '' có dây '' làm cơ sở duy nhất và duy nhất của chúng tôiIval\_box giao diện 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 phần phê bình của hệ thống phân cấp truyền thống:**

1. **Hệ thống giao diện người dùng phải là một chi tiết triển khai được ẩn khỏi những người dùng không muốn biết về nó.**
2. **Sự Ival\_box lớp không được chứa dữ liệu.**
3. **Không biên dịch lại mã bằng cách sử dụng Ival\_box họ các lớp nên được yêu cầu 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.**



**618** **Cấu trúc phân cấp lớp** **Chương 21**

**Một số cách tiếp cận khác có thể được thực hiện 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 như một giao diện thuần túy:**

**lớp Ival\_box {**

**công cộng:**

**ảo int get\_value () = 0; void**

**set\_value ảo (int i) = 0; void**

**reset\_value ảo (int i) = 0; dấu**

**nhắc void ảo () = 0;**

**bool ảo was\_changed () const = 0; ảo**

**˜Ival\_box () {}**

**};**

**Điều này rõ ràng hơn nhiều so với tuyên bố ban đầu của Ival\_box. Dữ liệu đã biến mất và các triển khai đơn giản của các hàm 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 trông như thế này:**

**lớp Ival\_slider: Ival\_box công khai, BBwidget được bảo vệ**

**{public:**

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

**Ghi đè ˜Ival\_slider ();**

**int get\_value () ghi đè; ghi đè**

**void set\_value (int i); //*...***

**được bảo vệ:**

* ***... các chức năng ghi đè các chức năng ảo BBwidget*// *ví dụ: BBwidget :: draw (), BBwidget :: mouse1hit () ...***



**riêng:**

* ***... dữ liệu cần thiết cho s***

**};**

**Lớp dẫn xuất Ival\_slider kế thừa từ một lớp trừu tượng (Ival\_box) yêu cầu nó thực hiện các chức năng ảo thuần túy của lớp cơ sở. Nó cũng kế thừa từBBwidget cung cấp cho nó các phương tiện để làm như vậy. Từ Ival\_box cung cấp giao diện cho lớp dẫn xuất, nó được dẫn xuất bằng cách sử dụngcông cộng. TừBBwidget chỉ là một trợ giúp thực hiện, nó được tạo ra bằng cách sử dụng được bảo vệ (§20.5.2). Điều này ngụ ý rằng một lập trình viên sử dụngIval\_slider không thể sử dụng trực tiếp các cơ sở được xác định bởi BBwidget. Giao diện được cung cấp bởiIval\_slider là người được thừa kế từ Ival\_box, cộng với cái gì Ival\_slider tuyên bố rõ ràng. Tôi đã sử dụngđược bảo vệ dẫn xuất thay vì hạn chế hơn (và thường an toàn hơn) riêngnguồn gốc để thực hiện BBwidget có sẵn cho các lớp bắt nguồn từ Ival\_slider. Tôi đã sử dụng rõ ràngghi đèbởi vì '' phân cấp tiện ích con '' này chính xác là loại phân cấp lớn, 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 rằngIval\_slider phải ghi đè các chức năng từ cả hai Ival\_box và BBwidget. Do đó, nó phải được xuất phát trực tiếp hoặc gián tiếp từ cả hai. Như thể hiện trong §21.2.1.1, dẫn xuấtIval\_slider gián tiếp từBBwidget bằng việc tạo ra BBwidget một cơ sở của Ival\_box là có thể, nhưng làm như vậy có mặt không mong muốn**



**Mục 21.2.2** **Kế thừa giao diện** **619**

**các hiệu ứng. Tương tự, tạo '' lớp triển khai ''BBwidget một thành viên của Ival\_box không phải là một giải pháp vì một lớp không thể ghi đè các chức năng ảo của các thành viên của nó. Biểu diễn cửa sổ bằng một BBwidget∗ thành viên 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 thừa kế '' chỉ ra một cái gì đó phức tạp và đáng sợ. Tuy nhiên, việc sử dụng một lớp cơ sở cho các chi tiết triển khai và một lớp khác cho giao diện (lớp trừu tượng) là phổ biến đối với tất cả các ngôn ngữ hỗ trợ giao diện được kiểm tra kế thừa và thời gian biên dịch. Đặc biệt, việc sử dụng lớp trừu tượngIval\_box gần giống như việc sử dụng một giao diện trong Java hoặc C #.**

**Thật thú vị, tuyên bố này của Ival\_slider cho phép mã ứng dụng được viết chính xác như trước đây. Tất cả những gì chúng tôi đã làm là**

**Nhiều lớp học yêu cầu lớp học như vậy 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 có yêu cầu một số. Chúng tôi đảm bảo dọn dẹp đúng cách bằng cách xác định một trình hủy ảoIval\_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) {**

* ***...***

**xóa p;**

**}**

**Các xóa bỏ toán tử hủy rõ ràng đối tượng được trỏ tới bởi P. Chúng tôi không có cách nào để biết chính xác lớp mà đối tượng trỏ đếnP thuộc về, nhưng cảm ơn Ival\_boxtrình hủy ảo của lớp, việc dọn dẹp thích hợp như (tùy chọn) được định nghĩa bởi trình hủy của lớp đó sẽ được thực hiện.**

**Các Ival\_box phân cấp bây giờ có thể được định nghĩa như sau:**

**lớp Ival\_box {/ \* *...* \* /};**

**lớp Ival\_slider**



* **Ival\_box công khai, BBwidget được bảo vệ {/ \* *...* \* /}; lớp Ival\_dial**
* **Ival\_box công khai,**

**Flashing\_ival\_slider lớp chuyên nghiệp**

* **Ival\_slider công khai {/ \* *...* \* /}; lớp Popup\_ival\_slider**
* **Ival\_slider công khai {/ \* *...* \* /};**

**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.**



**620** **Cấu trúc phân cấp lớp** **Chương 21**

**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:**

**lớp Ival\_box {/ \* *...* \* /}; //*chung*lớp**

**Ival\_slider**

* + **Ival\_box công khai, BBwidget được bảo vệ {/ \* *...* \* /}; //*cho BB* lớp Ival\_slider**
  + **Ival\_box công khai, CWwidget được bảo vệ {/ \* *...* \* /}; //*cho CW***
* ***...***

**Không có cách nào để có ngay cả khi hai người dùng interfa khác nhau Ival\_slider các lớp có tên riêng:**

**lớp Ival\_box {/ \* *...* \* /};**

**lớp BB\_ival\_slider**

* + **Ival\_box công khai, BBwidget được bảo vệ {/ \* *...* \* /}; lớp CW\_ival\_slider**
  + **Ival\_box công khai, CWwidget được bảo vệ {/ \* *...* \* /};**
* ***...***

**hoặc bằng đồ thị:**

**BBwidget** **Ival\_box** **CWwidget**



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

**Để cách ly hơn nữa theo định hướng ứng dụng của chúng tôi Ival\_box các lớp từ chi tiết triển khai, chúng ta có thể lấy ra một bản tóm tắt Ival\_slider lớp học từ Ival\_box và sau đó lấy ra hệ thống cụ thể Ival\_sliders từ đó:**



**lớp Ival\_box {/ \* *...* \* /};**

**lớp Ival\_slider**

* + **public Ival\_box {/ \* *...* \* /}; lớp BB\_ival\_slider**
  + **Ival\_slider công khai, BBwidget được bảo vệ {/ \* *...* \* /}; lớp CW\_ival\_slider**
  + **Ival\_slider công khai, CWwidget được bảo vệ {/ \* *...* \* /};**
* ***...***

**hoặc bằng đồ thị:**

**Ival\_box**



**BBwidget** **Ival\_slider** **CWwidget**



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



**Mục 21.2.3** **Triển khai thay thế** **621**

**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 một lớp thanh trượt, chúng tôi có thể lấyIval\_slider trực tiếp**

**từ BBslider:**

**lớp BB\_ival\_slider**

* **Ival\_slider công khai, BBslider được bảo vệ {/ \* *...* \* /}; lớp CW\_ival\_slider**
* **Ival\_slider công khai, CWslider được bảo vệ {/ \* *...* \* /};**

**hoặc bằng đồ thị:**

**BBw**



**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, lập trình được rút gọn thành ánh xạ giữa các khái niệm tương tự. Bắt nguồn từ các lớp cơ sở chung, chẳng hạn nhưBBwidget, sau đó hiếm khi được thực hiện.**

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

**lớp Ival\_box {/ \* *...* \* /};**

**lớp Ival\_slider**

* **public Ival\_box {/ \* *...* \* /}; lớp Ival\_dial**
* **public Ival\_box {/ \* *...* \* /}; lớp Flashing\_ival\_slider**
* **public Ival\_slider {class Popup\_ival\_slider**
* **Ival\_slider công khai {/ \* *...* \* /};**



**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:**

**lớp BB\_ival\_slider**

* **Ival\_slider công khai, BBslider được bảo vệ {/ \* *...* \* /}; lớp BB\_flashing\_ival\_slider**
* **public Flashing\_ival\_slider, BBwidget\_with\_bells\_and\_whistles được bảo vệ {/ \* *...* \* /}; lớp BB\_popup\_ival\_slider**
* **public Popup\_ival\_slider, BBslider được bảo vệ {/ \* *...* \* /};**

**lớp CW\_ival\_slider**

**: Ival\_slider công khai, CWslider được bảo vệ {/ \* *...* \* /};**

* ***...***

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

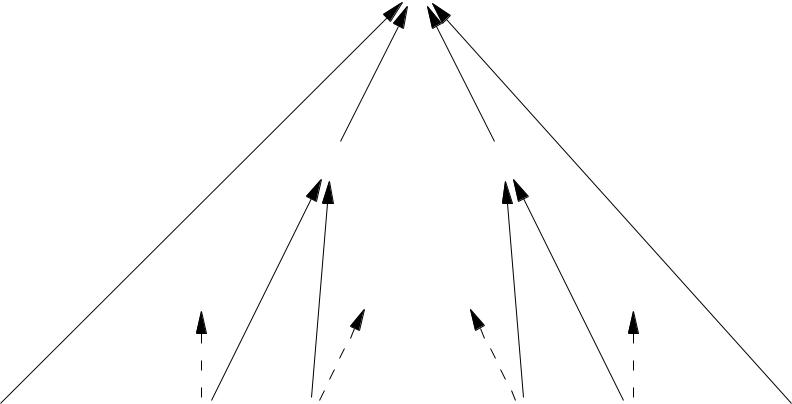


**622** **Cấu trúc phân cấp lớp** **Chương 21**

**Ival\_box**



**Ival\_slider** **Ival\_dial**



**ipopup** **iflash**

**BBslider**



**BBislider** **BBipop** **CWipop** **CWifl** **BBifl** **CWislider**

**Bản gốc Ival\_box phân cấp lớp 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 ở chỗ 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 xemIval\_box gia đình của các lớp mà không cần bận tâm đến các chi tiết triển khai liên quan đến cửa sổ hầu hết thời gian. Ví dụ, chúng tôi sẽ không cần phải viết lạitương tác() từ §21.2.1**



**Trong cả hai trường hợp, số lần hiển thị**

**giao diện của hệ thống giao diện người dùng thay đổi. Tuy nhiên, trong thiết kế lớp trừu tượng, hầu như tất cả mã người dùng được bảo vệ chống lại những thay đổi đối với hệ thống phân cấp thực thi và không yêu cầu biên dịch lại sau khi thay đổi như vậy. Điều này đặc biệt quan trọng khi nhà cung cấp phân cấp triển khai đưa ra bản phát hành mới '' gần như tương thích ''. 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 củaIval\_box hệ thống phân cấp ứng dụng lớp trừu tượng không thể vô tình sử dụng các phương tiện từ việc triển khai bởi vì chỉ các phương tiện được chỉ định rõ ràng trong Ival\_box hệ thống phân cấp có thể truy cập được; không có gì được kế thừa 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 kế thừa thực thi (§3.2.3, §20.1).**



**Mục 21.2.4** **Bản địa hóa tạo đối tượng** **623**

**21.2.4 Bản địa hóa Tạo đối tượng**

**Hầu hết một ứng dụng có thể được viết bằng cách sử dụng Ival\_box giao diện. Hơn nữa, nếu các giao diện có nguồn gốc phát triển để cung cấp nhiều tiện ích hơn so vớiIval\_box, thì hầu hết một ứng dụng có thể được viết bằng cách sử dụng Ival\_box, Ival\_slider, v.v., giao diện. Tuy nhiên, việc tạo các đối tượng phải được thực hiện bằng cách sử dụng các tên dành riêng cho việc triển khai nhưCW\_ival\_dial và BB\_flashing\_ival\_slider. Chúng tôi muốn giảm thiểu số lượng địa điểm 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:**

**lớp Ival\_maker {**

**công cộng:**

**Ival\_dial ảo∗ quay số (int, int) = 0; Popup\_ival\_slider** **// *thực hiện quay số***

**ảo∗ popup\_slider (int, int) = 0; //*...*** **// *tạo thanh trượt bật lên***

**};**

**Đối với mỗi giao diện từ Ival\_box họ lớp mà người dùng nên biết, lớpIval\_maker cung cấp một hàm tạo một đối tượng. Một lớp như vậy đôi khi được gọi là*xưởng sản xuất*và các chức năng của nó (hơi gây hiểu lầm) đôi khi được gọi là *các nhà xây dựng 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:**

**lớp BB\_maker: public Ival\_maker // *tạo phiên bản BB* {public:**

**Ival\_dial∗ ghi đè quay số (int, int); Popup\_ival\_slider∗**

**ghi đè popup\_slider (int, int); //*...***

**};**

**lớp LS\_maker: public Ival\_maker** **// *tạo phiên bản LS***



**{public:**

**Ival\_dial∗ quay số (int, int Popup\_ival\_slider∗ ghi đè**

**popup\_slider (int, int); //*...***

**};**

**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) {**

**trả về BB\_ival\_dial mới (a, b);**

**}**

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

**trả về LS\_ival\_dial mới (a, b);**

**}**

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



**624** **Cấu trúc phân cấp lớp** **Chương 21**

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

**unique\_ptr <Hộp\_mẫu\_vào> pb {im.dial** **// *tạo quay số thích hợp***

**(0,99)}; //*...***

**}**

**BB\_mặt BB\_impl;** **// *cho người dùng BB***

**LS\_sản\_phẩm LS\_impl;** **// *cho người dùng LS***

**void driver ()**

**{**

**người dùng (BB\_impl);**

**người dùng (LS\_impl);**

**}**

**Việc chuyển các đối số cho '' các hàm tạo ảo '' 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 được chia sẻ*: dẫn đến việc sao chép mã ít hơn bằng cách sử dụng các lớp và làm cho mã đóđồng nhất hơn. Điều này thường được gọi là*đa hình thời gian chạy* hoặc *kế thừa giao diện*.**
* ***Triển khai được chia sẻ*: dẫn đến ít mã hơn và mã triển khai thống nhất hơn. Điều này**

**thường được gọi là*kế thừa thực hiện*.**

**Một lớp có thể kết hợp các khía cạnh của hai phong cách này.**

* **đây, chúng tôi khám phá cách sử dụng chung hơn của nhiều lớp cơ sở và kiểm tra các vấn đề kỹ thuật khác liên quan đến kết hợp và ac**



**21.3.1 Nhiều Interf**

**Một lớp trừu tượng (ví dụ: Ival\_box; §21.2.2) là cách hiển nhiên 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 việc sử dụng một lớp và 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 mạng đa kế thừa 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 biểu thị là đối tượng của lớp Vệ tinh. MỘTVệ tinh đối tượng sẽ chứa các thông số quỹ đạo, kích thước, hình dạng, albedo, mật độ, v.v. và cung cấp các hoạt động để tính toán quỹ đạo, sửa đổi thuộc tính, v.v. Ví dụ về**



**Mục 21.3.2** **Nhiều lớp triển khai** **625**

**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 từVệ tinh. Các lớp dẫn xuất như vậy sẽ thêm các thành viên và chức năng dữ liệu và sẽ ghi đè một sốVệ tinhcủa các chức năng ảo để điều chỉnh ý nghĩa của chúng một cách thích 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 hiển thị 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 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 xác định một mệnh đề**

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

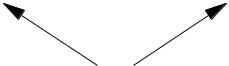
**{public:**

* ***...***

**};**

**hoặc bằng đồ thị:**

**Vệ tinh** **Hiển thị**



**Comm\_sat**

**Ngoài bất kỳ hoạt động nào được xác định cụ thể cho một Comm\_sat, liên hiệp các hoạt động trên Vệ tinh và Hiển thị có thể ứng tuyển. Ví dụ:**

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

**{**

**s.draw ();** **// *Hiển thị :: draw ()***

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



**s.transmit ();**

**}**

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

**void highlight (Đã hiển thị∗);**

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

**void g (Comm\_sat∗ P) {**

**tô sáng (p); // *chuyển một con trỏ đến phần Hiển thị của Comm\_sat*// Pos x = center\_of\_gravity (p); *chuyển một con trỏ đến phần Vệ tinh của 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 xem một phần khác của một Comm\_sat hơn là làm các chức năng mong đợi một Hiển thị. Các chức năng ảo hoạt động như bình thường. Ví dụ:**



**626** **Cấu trúc phân cấp lớp** **Chương 21**

**vệ tinh lớp {**

**công cộng:**

**tâm Pos ảo () const = 0; //*...***

* ***Trung tâm của lực hấp dẫn***

**};**

**lớp được hiển thị {**

**công cộng:**

**ảo void draw () = 0; //*...***

**};**

**class Comm\_sat: public**

**public:**

**Ghi đè trung tâm Pos () const; // *ghi đè Satellite :: center ()*//**

**ghi đè void draw ();** ***ghi đè Displayed :: draw ()***

**// *...***

**};**

**Điều này đảm bảo rằng Comm\_sat :: center () và Hiển thị :: draw () sẽ được gọi cho một Comm\_sat đối xử như là Một Comm\_sat và một Hiển thị, tương ứng. Tại sao tôi không giữ lạiVệ tinh và Hiển thị các bộ phận của một**

**Comm\_sat hoàn toàn tách biệt? Tôi có thể đã định nghĩaComm\_sat để có một Vệ tinh thành viên và một Hiển thị thành viên. Ngoài ra, tôi có thể đã xác địnhComm\_sat để có một Vệ tinh∗ thành viên và một Hiển thị∗ thành viên và để phương thức khởi 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 truyền cảm hứng cho ví dụ này được xây dựng dựa trên ý tưởng về một Vệ tinh lớp với các chức năng ảo và một (được thiết kế riêng) Hiển thị lớp 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 đã phải ghi đèVệ tinh chức năng thành viên ảo và Hiển thị chức năng thành viên ảo để chỉ định hành vi của các đối tượng của riêng bạn. Đó là situati**



**khó tránh khỏi sự cố vấn.**

**Việc sử dụng đa kế thừa để '' gắn kết '' hai lớp không liên quan với nhau như một phần của việc triển khai lớp thứ ba là thô thiển, hiệu quả và tương đối quan trọng, nhưng không thú vị lắm. Về cơ bản, nó giúp lập trình viên không phải viết 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 nhất thiết 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 có 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), các lựa chọn của lập trình viên để triển khai các lớp**

**Hiển thị, Vệ tinh, và Comm\_sat sẽ bị giới hạn. MỘTComm\_sat có thể là một Vệ tinh hoặc một Dis-**

**chơi, nhưng không phải 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.**



**Mục 21.3.2** **Nhiều lớp triển khai** **627**

**Tại sao mọi người lại muốn có một lớp học Comm\_sat? Trái với phỏng đoán của một số người,Vệ tinh ví dụ 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 hệ thống thông tin liên lạc liên quan đến vệ tinh, trạm mặt đất, v.v. Thực tế,Vệ tinh được bắt nguồn 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 giao thông liên lạc, xác định phản ứng thích hợp cho trạm mặt đất đang bị mưa bão chặn, xem xét sự cân bằng giữa kết nối vệ tinh và kết nối liên kết với Trái đất, v.v.**

**21.3.3 Độ phân giải mơ hồ**

**Hai lớp cơ sở có thể có**

**vệ tinh lớp {**

**công cộng:**

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

***...***

**};**

**lớp được hiển thị {**

**công cộng:**

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

***...***

**};**

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

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

**{**

**Debug\_info di = cs.get\_debug (); di** **// *lỗi: mơ hồ***

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



**cs.Displayed :: ge**

**}**

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

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

**{public:**

**Debug\_info get\_debug () // *ghi đè Comm\_sat :: get\_debug () và Displayed :: get\_debug ()*{**

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

**Debug\_info di2 = Displayed :: get\_debug (); trả**

**về merge\_info (di1, di2);**

**}**

* ***...***

**};**

**Một hàm được khai báo trong một lớp dẫn xuất sẽ ghi đè *tất cả các* các hàm cùng tên và nhập vào 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 choảo là cho một**



**628** **Cấu trúc phân cấp lớp** **Chương 21**

**lệnh gọi để có tác dụng tương tự độc lập với giao diện nào đã 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. Tên đủ điều kiện, chẳng hạn nhưTelstar :: hòa, có thể tham khảo một vẽ 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 ();** **// *finds Hiển thị :: vẽ***

* ***... đồ riêng ...***

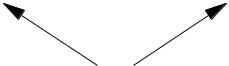
**}**

* ***...***

**};**

**hoặc bằng đồ thị:**

**Vệ tinh** **Hiển thị**



**Comm\_sat**



**Telstar**

**Nếu như Comm\_sat :: draw không giải quyết vẽ tuyên bố trong Comm\_sat, trình biên dịch tìm kiếm một cách đệ quy trong các lớp cơ sở của nó; đó là, nó tìm kiếmVệ tinh :: vẽ và Hiển thị :: vẽ, và nếu cần thiết sẽ tìm 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 thì,Comm\_sat :: draw không được tìm thấy hoặc không rõ ràng.**

**Nếu, trong Telstar :: draw (), Tôi đã nói đơn giản vẽ(), kết quả sẽ là một đệ quy '' vô hạn ''**



**cuộc gọi của Telstar :: draw ().**

**Tôi có thể đã nói Displa**

**Một 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 nóiComm\_sat :: Displayed :: draw (), nhưng điều đó sẽ là thừa. Tôi đã nóiVệ tinh :: draw (), kết quả sẽ là một lỗi vì vẽ kết thúc trên Hiển thịnhánh của hệ thống phân cấp lớp.**

**Các get\_debug () ví dụ về cơ bản giả định rằng ít nhất một số phần của Vệ tinh và Hiển thị 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 cung cấp theo những cách khác nhau để có thể hợp nhất nó thành một thứ có thể được sử dụng cùng nhau. Ban đầu chúng ta có thể đã được giới thiệu với hai lớpSimObj 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 kếVệ tinh và Hiển thị như các lớp giao diện của chúng tôi, cung cấp '' lớp ánh xạ '' cho các lớp cấp cao hơn của chúng tôi sử dụng:**



**Mục 21.3.3** **Độ phân giải mơ hồ** **629**

**lớp Satellite: public SimObj {**

* ***ánh xạ các cơ sở của SimObj tới một thứ dễ sử dụng hơn để mô phỏng vệ tinh***

**công cộng:**

**virtual Debug\_info get\_debug (); // // *gọi SimObj :: DBinf () và trích xuất thông tin* *...***

**};**

**lớp Hiển thị: Widget công cộng {**

* ***ánh xạ các cơ sở widget sang thứ gì đó dễ sử dụng hơn để hiển thị các kết quả mô phỏng vệ tinh***

**công cộng:**

**virtual Debug\_info g**

* ***...***

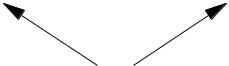
**};**

**hoặc bằng đồ thị:**

**SimObj** **Widget**



**Vệ tinh** **Hiển thị**



**Comm\_sat**



**Telstar**

**Thật thú vị, đây chính xác là kỹ thuật chúng tôi sẽ sử dụng để phân biệt trong trường hợp không chắc chắn khi hai lớp cơ sở cung cấp các hoạt động có cùng tên chính xác, nhưng với ngữ nghĩa khác nhau: thêm một lớp giao diện. Hãy xem xét ví dụ cổ điển (nhưng chủ yếu là giả thuyết / lý thuyết) về một lớpvẽ() chức năng thành viên trong trò chơi điện tử liên quan đến cao bồi:**



**Cửa sổ lớp học {**

**công cộng:**

**void draw ();** **// *hình ảnh hiển thị***

* ***...***

**};**

**cao bồi lớp {**

**công cộng:**

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

* ***...***

**};**

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

* ***...***

**};**

**Làm thế nào để chúng tôi ghi đè Cowboy :: draw () và Cửa sổ :: 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 lên hai**



**630** **Cấu trúc phân cấp lớp** **Chương 21**

**các chức năng riêng biệt. Không có giải pháp ngôn ngữ trực tiếp nào cho vấn đề (kỳ lạ) này, nhưng việc thêm các lớp trung gian sẽ làm được:**

**struct WWindow: Cửa sổ {**

**sử dụng Window :: Window;** **// *kế thừa các hàm tạo*// *buộc lớp***

**void win\_draw () = 0;** ***dẫn xuất ghi đè*// *hình ảnh hiển***

**void draw () ghi đè final {win\_draw (); }** ***thị***

**};**

**struct CCowboy: Cao bồi {**

**sử dụng Cowboy :: Cowboy;** **// *kế thừa các hàm tạo***

**khoảng trống ảo cow\_dra**

**ghi đè void draw ()**

**};**

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

**{public:**

**ghi đè void cow\_draw ();**

**void win\_draw () ghi đè; //**

***...***

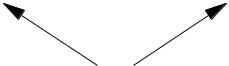
**};**

**Hoặc bằng đồ thị:**

**Cửa sổ** **Cao bồi**



**WWindow** **CCowboy**



**Cowboy\_window**



**Đã có nhà thiết kế của**

**Windovấn đề sẽ bay hơi**

**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 các 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:**

**cấu trúc lưu trữ {** **// *lưu trữ liên tục***

**chuỗi ảo get\_file () = 0; ảo**

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

**0;**

**ảo ˜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:**



**Mục 21.3.4** **Sử dụng lặp lại một lớp cơ sở** **631**

**class Transmitter: public Stocking**

**{public:**

**ghi đè void write ();**

* ***...***

**};**

**Class Receiver: public Stocking**

**{public:**

**ghi đè void write ();**

* ***...***

**};**

**class Radio: public Trans**

**public:**

**ghi đè chuỗi get\_file ();**

**void read () ghi đè;**

**ghi đè void write ();**

* ***...***

**};**

**Do đó, chúng ta có thể hình dung hai trường hợp:**

**[1] A Đài đối tượng có hai subobjects của lớp Đáng yêu (một cho Hệ thống điều khiển và một cho**

**Người nhận).**

1. **A Đài đối tượng có một đối tượng của lớp Đáng yêu (chia sẻ bởi Hệ thống điều khiển và Người nhận). Mặc định, được cung cấp cho ví dụ như đã viết, là hai subobject. Trừ khi bạn nêu cách khác, bạn sẽ nhận được một bản sao cho mỗi lần bạn đề cập đến một lớp làm cơ sở. Về mặt đồ họa, chúng ta có thể biểu diễn điều đó như sau:**

**Đáng yêu** **Đáng yêu**



**Đài**

**Một hàm ảo của một lớp cơ sở được sao chép có thể được ghi đè bởi một hàm (đơn) trong một lớp dẫn xuất. Thông thường, một hàm ghi đè gọi các phiên bản lớp cơ sở của nó và sau đó thực hiện công việc cụ thể cho lớp dẫn xuất:**

**void Radio :: write ()**

**{**

**Bộ truyền :: write ();**

**Bộ nhận :: write ();**

* ***... viết thông tin cụ thể về radio ...***

**}**

**Truyền từ một lớp cơ sở được sao chép sang một lớp dẫn xuất được thảo luận trong §22.2. Đối với một kỹ thuật để ghi đè mỗiviết() các hàm với các hàm riêng biệt từ các lớp dẫn xuất, xem §21.3.3.**



**632** **Cấu trúc phân cấp lớp** **Chương 21**

**21.3.5 Lớp cơ sở ảo**

**Các Đài ví dụ trong tiểu mục trước hoạt động vì lớp Đáng yêu có thể được sao chép một cách an toàn, thuận tiện và hiệu quả. Lý do cho điều đó chỉ đơn giản làĐáng yêu là một lớp trừu tượng cung cấp một giao diện thuần túy. MỘT Đáng yêu đối tượng không chứa dữ liệu của riêng nó. Đây là trường hợp đơn giản nhất và là trường hợp cung cấp sự tách biệt tốt nhất giữa các mối quan tâm về giao diện và triển khai. Trên thực tế, một lớp học không thể không gặp một số khó khăn khi xác định rằng có haiĐáng yêu subobjects trên một Đài. Chuyện gì xảy ra nếuĐáng yêu đã**

**giữ dữ liệu và điều quan trọng là nó không được sao chép? Ví dụ, chúng tôi có thể xác địnhĐáng yêu giữ tên của tệp được sử dụng để lưu trữ đối tượng:**

**lớp học ổn định {**

**công cộng:**

**Lưu trữ (chuỗi const**

**ảo void read () = 0; void**

**ghi () = 0; ảo ˜Storable**

**();**

**được bảo vệ:**

**chuỗi tên\_tệp;**

**Stocking (const Stocking &) = xóa;**

**Stocking & operator = (const Stocking &) = delete;**

**};**

**Với thay đổi rõ ràng là nhỏ này đối với Đáng yêu, chúng ta phải thay đổi thiết kế của Đài. Tất cả các bộ phận của một đối tượng phải dùng chung một bản sao củaĐáng yêu. Nếu không, chúng ta có thể nhận được hai phần của thứ gì đó bắt nguồn từĐáng yêu nhiều lần bằng cách sử dụng các tệp khác nhau. Chúng tôi tránh sao chép bằng cách khai báo một cơ sởảo: mỗi ảo cơ sở của một lớp dẫn xuất được đại diện bởi cùng một đối tượng (dùng chung). Ví dụ:**

**class Transmitter: public virtual Stocking**

**{public:**



**ghi đè void write ()**

* ***...***

**};**

**class Receiver: public virtual Stocking**

**{public:**

**ghi đè void write ();**

* ***...***

**};**

**Class Radio: Máy phát công cộng, Máy thu công cộng**

**{public:**

**ghi đè void write ();**

* ***...***

**};**

**Hoặc bằng đồ thị:**



**Mục 21.3.5** **Các lớp cơ sở ảo** **633**

**Đáng yêu**



**Người nhận** **Hệ thống điều khiển**



**Đài**

**So sánh sơ đồ này với bản vẽ của Đáng yêu đối tượng 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 cụ thể được chỉ định làảo sẽ được đại diện bởi một đối tượng duy nhất của lớp đó. Mặt khác, mỗi lớp cơ sở**

**Tại sao ai đó sẽ w**

**các cách để hai lớp trong hệ thống phân cấp lớp chia sẻ dữ liệu:**

1. **Làm cho dữ liệu 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) '' bong bóng '' thành một lớp cơ sở chung; thường nó '' bong bóng '' đến tận gốc của cây thừa kế. Điều này có nghĩa là mọi thành viên của hệ thống phân cấp lớp 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) phương thức khởi tạo 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.**



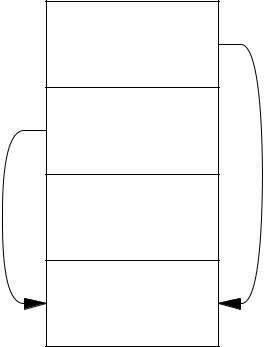
**Nếu bạn không cần sharin**

**thường đơn giản hơn cho nó. Ho**

**cally có một sự lựa chọn giữa các ý**

**tưởng.**

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



**Người nhận**

**Hệ thống điều khiển**

**Đài**

**Đáng yêu**



**634** **Cấu trúc phân cấp lớp** **Chương 21**

**Các '' con trỏ '' đến đối tượng được chia sẻ đại diện cho cơ sở ảo, Đáng yêu, sẽ là phần bù và thường một trong số đó có thể được tối ưu hóa bằng cách đặt Đáng yêu ở một vị trí cố định liên quan đếnNgười nhận hoặc là Hệ thống điều khiển subobject. 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 là ini**

**được gọi (ngầm hiểu hoặc rõ ràng**

**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);**

* ***...***

**};**

**cấu trúc A {**

**MỘT();** **// *nhà xây dựng mặc định***

* ***...***

**};**

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

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

**};**



**lớp C: virtual V**

**{public:**

**C (int i): V {i} {/ \* *...***

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

**// *phải khởi tạo cơ sở V***

**};**

**lớp D: công cộng ảo B, công cộng ảo C {**

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

**công cộng:**

**NS() { /\* *...* \* /} D (int i): C {i} {/ \* *...***

**// *lỗi: không có hàm tạo mặc định cho C hoặc V*//**

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

***lỗi: không có hàm tạo mặc định cho V***

**\* /} // *...***

**// *VÂNG***

**};**

**Lưu ý rằng NS có thể và phải cung cấp trình khởi tạo cho V. Thực tế làV không được đề cập rõ ràng như một cơ sở của NS không liên quan. 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 có nguồn gốc nhiều nhất của nó**



**Mục 21.3.5.1** **Xây dựng cơ sở ảo** **635**

**lớp. Thực tế là cả haiNS và NS khởi tạo V là không liên quan vì trình biên dịch không có ý tưởng nào trong hai trình 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 lấy được một lớp khác,DD, từ NS, sau đó DD phải làm công việc để khởi tạo các cơ sở ảo. Trừ khi chúng ta có thể kế thừa một cách đơn giảnNSCác hàm tạo của (§20.3.5.1), điều đó có thể gây phiền toái. Điều đó phải 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 chỉ 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 xác định 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 cơ bản Cửa sổ lớp biết cách vẽ nội dung của nó:**

**Cửa sổ lớp học {**

**công cộng:**

* ***vật liệu cơ bản***

**ảo 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 {**



* ***công cụ biên giới***

**được bảo vệ:**

**void own\_draw ();**

**công cộng:**

**ghi đè void draw ();**

**};**

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

* ***thực đơn***

**được bảo vệ:**

**void own\_draw ();** **// *hiển thị menu***

**công cộng:**

**ghi đè void draw ();**

**};**

**Các own\_draw () các chức năng không cần phải ảo bởi vì chúng có nghĩa là được gọi từ bên trong một vẽ() hàm '' biết '' loại đối tượng mà nó được gọi.**

**Từ điều này, chúng tôi có thể tạo ra một Cái đồng hồ lớp:**



**636** **Cấu trúc phân cấp lớp** **Chương 21**

**class Clock: public Window\_with\_border, public Window\_with\_menu {**

* ***công cụ đồng hồ***

**được bảo vệ:**

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

**công cộng:**

**ghi đè void draw ();**

**};**

**hoặc bằng đồ thị:**

**Cửa sổ**



**W**



**Cái đồng hồ**

**Các vẽ() các chức năng bây giờ có thể được xác định bằng cách sử dụng own\_draw () để một người gọi bất kỳvẽ() được Cửa sổ :: draw () được gọi chính xác một lần. Điều này được thực hiện độc lập với loạiCửa sổmà vẽ() Được gọi:**

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

**Cửa sổ :: draw ();**

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

**}**

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

**Cửa sổ :: draw ();**



**own\_draw (); //*hiển thị menu***

**}**

**void Clock :: draw ()**

**{**

**Cửa sổ :: draw ();**

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

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

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

**}**

**Lưu ý rằng một cuộc gọi đủ điều kiện, chẳng hạn như Cửa sổ :: 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. LớpBB\_ival\_slider (§21.2.3) là một ví dụ:**



|  |  |  |
| --- | --- | --- |
| **Mục 21.3.6** | **Cơ sở sao chép so với Cơ sở ảo** | **637** |
| **lớp BB\_ival\_slider** |  |  |
| **: Ival\_slider công khai,** | **// *giao diện*** |  |
| **BBslider được bảo vệ** | **// *thực hiện*** |  |

**{**

* ***triển khai các chức năng do Ival\_slider và BBslider yêu cầu, sử dụng các tiện ích từ BBslider***

**};**

**Trong ví dụ này, hai lớp cơ sở đóng các vai trò khác biệt 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 chi tiết triển khai ''. '' Các vai trò này được phản ánh trong cả kiểu của các lớp và trong điều khiển truy cập (§20.5) được cung cấp. Việc sử dụng đa**

**để ghi đè các chức năng ảo**

**Ví dụ, hãy xem xét một Ival\_box các lớp trừu tượng để phản ánh vai trò của chúng như là các 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 đã được thực hiện trong 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).**
* **Tạo 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 như một cơ sở ảo cung cấp cho chúng tôi:**

**lớp BB\_ival\_slider**

* **Ival\_slider ảo công khai, BBslider được bảo vệ {/ \* *...* \* /}; lớp Popup\_ival\_slider**
* **Ival\_slider ảo công khai {/ \* *...* \* /};**

**lớp BB\_popup\_ival\_slider**

**: Popup\_ival\_slider ảo công khai, BB\_ival\_slider được bảo vệ {/ \* *...* \* /};**

**hoặc bằng đồ thị:**



**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 bản sao**

**Ival\_slider các đối tượng:**

**lớp BB\_ival\_slider**

* **Ival\_slider công khai, BBslider được bảo vệ {/ \* *...* \* /}; lớp Popup\_ival\_slider**
* **Ival\_slider công khai {/ \* *...* \* /};**

**lớp BB\_popup\_ival\_slider**

**: public Popup\_ival\_slider, BB\_ival\_slider được bảo vệ {/ \* *...* \* /};**



**638**

**Cấu trúc phân cấp lớp**

**Chương 21**

**hoặc bằng đồ thị:**

**Ival\_slider**



**Ival\_slider**



**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 khác. Có sự khác biệt logic**

**không thể chuyển đổi ngầm**

**void f (Ival\_slider∗ P);**

**void g (BB\_popup\_ival\_slider∗ P) {**

**f (p); //*error: Popup\_ival\_slider :: Ival\_slider hay 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 mơ hồ cho các phôi *từ* lớp cơ sở (§22.2). Tuy nhiên, những sự mơ hồ như vậy rất dễ bị xử lý.**

**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, thông thường nhất, chúng ta không có lựa chọn vì chúng ta phải tuân theo một thiết kế hiện có. Khi có sự lựa chọn, chúng ta có thể tính đến rằng (đáng ngạc nhiên là) giải pháp cơ sở sao chép có xu hướng dẫn đến các đối tượng nhỏ hơn một chút (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 nhà máy '' (§21.2.4). Ví dụ:**



**Popup\_ival\_slider∗ bật lên {**

* ***...***

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

***...***

**}**

**Không cần chuyển đổi rõ ràng để nhận được từ triển khai (tại đây, BB\_popup\_ival\_slider) với các giao diện trực tiếp của nó (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 các triển khai cho giao diện được trình bày bởi một lớp cơ sở ảo. Ví dụ, Cửa sổ lớp có thể có các chức năng set\_color () và lời nhắc(). Trong trường hợp đó,Window\_with\_border có thể ghi đè set\_color () như một phần của việc kiểm soát bảng màu vàWindow\_with\_menu có thể ghi đè lời nhắc() như một phần của việc kiểm soát các tương tác của người dùng:**



|  |  |  |
| --- | --- | --- |
| **Mục 21.3.6.1** | **Ghi đè các hàm cơ sở ảo** | **639** |
| **Cửa sổ lớp học {** |  |  |
| **// *...*** |  |  |
| **void set\_color (Color) = 0; dấu** | **// *đặt màu nền*** |  |
| **nhắc void ảo () = 0;** |  |  |

**};**

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

* ***...***

**ghi đè void set\_color (Màu);** **// *kiểm soát màu nền***

**};**

**lớp Window\_with\_menu**

* ***...***

**ghi đè void prompt ();** **// *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,Cửa sổ của tôi có thể ghi đè lời nhắc() để cải thiện những gì Thắng lợi-**

**dow\_with\_menu cung cấp:**

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

* ***...***

**ghi đè void prompt (); //*không để các tương tác của người dùng làm cơ sở***

**};**

**hoặc bằng đồ thị:**



**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 đơn lẻ 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, đãĐài trong §21.3.5 không được khai báo viết(), tuyên bố củaviết() trong Người nhận và Hệ thống điều khiển sẽ gây ra lỗi khi xác định Đài. Như vớiĐài, 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*.**



**640** **Cấu trúc phân cấp lớp** **Chương 21**

**21.4 Lời khuyên**

1. **Sử dụng unique\_ptr hoặc shared\_ptr để tránh quên xóa bỏ các đối tượng được tạo ra bằng cách sử dụng Mới; §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 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 nhiều thừa kế**
10. **Sử dụng cơ sở ảo để**

**chy; §21.3.5.**

