**CHƯƠNG 20 : LỚP DẪN XUẤT**

**20.1 Giới thiệu**

Từ Simula, C ++ mượn ý tưởng các lớp và cấu trúc phân cấp lớp. Một khái niệm (ý tưởng, khái niệm, v.v.) không tồn tại một cách cô lập. Vì chúng tôi sử dụng các lớp để thể hiện các khái niệm, nên vấn đề trở thành cách thể hiện mối quan hệ giữa các khái niệm. Chương này là một khám phá về ý nghĩa của ý tưởng đơn giản này, là cơ sở cho cái thường được gọi là lập trình hướng đối tượng. Các tính năng ngôn ngữ hỗ trợ xây dựng các lớp mới từ những lớp hiện có:

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

• Kế thừa giao diện: cho phép các lớp dẫn xuất khác nhau được sử dụng thay thế cho nhau thông qua giao diện được cung cấp bởi một lớp cơ sở chung Kế thừa giao diện thường được gọi là đa hình thời gian chạy (hoặc đa hình động).

**20.2 Lớp Dẫn Xuất**

**Employee**

**↑**

**Manager**

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

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

**Employee: Manager:**

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

Ví dụ:

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

**Nhân viên ∗ pe = & mm; // OK: mọi Người quản lý đều là Nhân viên**

**Người quản lý ∗ pm = & ee; // error: không phải mọi Nhân viên đều là Người quản lý**

**pm−> cấp = 2; // thảm họa: ee doesn’t have a lev el**

**pm = static\_cast <Manager ∗> (pe); // brute force:hoạt động vì pm**

**// tới Trình quản lý mm**

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

**}**

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

**20.2.1 Hàm thành viên**

Thành viên của lớp dẫn xuất có thể sử dụng công khai - và được bảo vệ (xem §20.5) - các thành viên của lớp cơ sở như thể chúng được khai báo trong chính lớp dẫn xuất. Ví dụ:

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

**{**

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

**// ...**

**}**

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

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

**{**

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

**// ...**

**}**

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

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**

**{**

**in(); //Opp!**

**// thông tin cụ thể của print Manager**

**}**

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, hàm tạo và hàm hủy là thiết yếu:

• Các đối tượng được xây dựng từ dưới lên và phá hủy từ trên xuống .

• Mỗi lớp có thể khởi tạo các thành viên và cơ sở của nó .

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

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

• Độ phân giải của một lệnh gọi hàm ảo, một dynamic\_cast, hoặc một typeid () trong một phương thức khởi tạo hoặc de-structor phản ánh giai đoạn xây dựng và phá .

**20.3 Phân cấp lớp**

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

**Nhân viên lớp {/ \* ... \* /};**

**quản lý lớp: public Employee {/ \* ... \* /};**

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

Một tập hợp các lớp liên quan như vậy theo truyền thống được gọi là hệ thống phân cấp lớp. Hệ thống phân cấp như vậy thường một cây, nhưng nó cũng có thể là một cấu trúc đồ thị tổng quát hơn. Ví dụ:

lớp Temporar y {/ \* ... \* /};

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

class Temp: public Temporar y, public Assistant {/ \* ... \* /};

tư vấn lớp: public Temporar y, public Manager {/ \* ... \* /};

hoặc bằng đồ thị:

**Tạm thời: Nhân viên**

**Phụ tá Người quản lý**

**NVBTG**

**Tư vấn Giám đốc**

**20.3.1 Trường loại**

Bài toán hạ thấp: Cho một con trỏ kiểu 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.

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

[3] Sử dụng dynamic\_cast .

[4] Sử dụng các hàm ảo .

Giải pháp 1 nó có thể được sử dụng để áp dụng- đề cập đến các vùng chứa đồng nhất (ví dụ: vectơ thư viện chuẩn 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, 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].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]. 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, family\_name;

char middle\_initial;

Ngày\_công\_dụng;

ngắn department;

// ...};

struct Manager: public Employee {

Người quản lý () {type = man; }

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

mức độ ngắn;

// ...

};

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

void print\_employee (const Employee ∗ e)

{

switch (e−> type) {

case Nhân viên :: empl:

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

// ...

nghỉ;

case Employee :: man:

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

// ...

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

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

// ...

nghỉ;

}

}

}

và sử dụng nó để in danh sách Nhân viên, như sau:

void print\_list (const list <Employee ∗> & elist)

{

cho (auto x: elist)

print\_employee (x);

}

Điều này hoạt động tốt trong một chương trình nhỏ do một người duy trì. Tuy nhiên, nó có một điểm yếu cơ bản là nó phụ thuộc vào việc người lập trình 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 hàm như print\_employee () thường được tổ chức để tận dụng tính phổ biến của các lớp liên quan:

void print\_employee (const Employee ∗ e)

{

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

// ...

if (e−> type == Employee :: man) {

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

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

// ...

}

}

Sử dụng trường kiểu là một kỹ thuật dễ xảy ra lỗi dẫn đến lỗi bảo trì.Mỗi hàm sử dụng trường kiểu phải biết về cách biểu diễn và các chi tiết khác về việc triển khai mọi lớp được dẫn xuất từ ​​lớp chứa trường kiểu.

**20.3.2 Chức năng ảo**

Các hàm ảo khắc phục các vấn đề với giải pháp trường kiểu bằng cách cho phép lập trình viên khai báo các hàm trong một lớp cơ sở có thể được định nghĩa lại trong mỗi lớp dẫn xuất. 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.

class Employee {

public:

Employee(const string& name, int dept);

virtual void print() const;

// ...

private:

string first\_name , family\_name;

short depar tment;

// ...

};

Từ khóa virtual chỉ ra rằng print () có thể hoạt động như một giao diện cho hàm print () được định nghĩa trong lớp này và các hàm print () được định nghĩa trong các lớp dẫn xuất từ ​​nó. Trong trường hợp các hàm print () như vậy được định nghĩa trong các lớp dẫn xuất, trình biên dịch đảm bảo rằng print () phù hợp cho đối tượng Employee đã cho 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 loại trả lại

(§20.3.6). Một hàm thành viên ảo đôi khi được gọi là một phương thức.

Một hàm ảo 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 Employee::print() const

{

cout << family\_name << '\t' << department << '\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 thích hợp nếu nó cần thiết. Ví dụ:

class Manager : public Employee {

public:

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

void print() const;

// ...

private:

list<Employee∗> group;

short level;

// ...

};

void Manager::print() const

{

Employee::print();

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

// ...

}

Một hàm từ một lớp dẫn xuất có cùng tên và cùng một tập hợp các kiểu đối số như một hàm ảo trong cơ sở được cho là ghi đè phiên bản lớp cơ sở của hàm ảo. Hơn nữa, nó 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 nào của một hàm ảo được gọi (như trong lệnh gọi Employee :: print ()), thì hàm ghi đè được chọn là phù 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.

Hàm toàn cục **print\_employee ()** (§20.3.1) bây giờ không cần thiết vì các hàm thành viên print () đã thay thế. Danh sách Nhân viên có thể được in như sau:

**void print\_list(const list<Employee∗>& s)**

**{**

**for (auto x : s)**

**x−>print();**

**}**

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

**int main()**

**{**

**Employee e {"Brown",1234};**

**Manager m {"Smith",1234,2};**

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

**}**

produced:

Smith 1234

level 2

Brown 1234

Lưu ý rằng điều này sẽ hoạt động ngay cả khi print\_list () được viết và biên dịch trước khi Trình quản lý lớp dẫn xuất cụ thể được hình thành! Đây là một khía cạnh quan trọng của các lớp học. Khi được sử dụng đúng cách, nó sẽ trở thành nền tảng của các thiết kế hướng đối tượng và cung cấp mức độ ổn định cho một chương trình phát triển.

Nhận được hành vi ‘‘ đúng ’’ từ các chức năng của Nhân viên một cách độc lập với chính xác loại Nhân viên thực sự được sử dụng được gọi là tính đa hình. Một kiểu có các chức năng ảo được gọi là poly-

kiểu đa hình hay (chính xác hơn) kiểu đa hình thời gian chạy. Để có được hành vi đa hình trong thời gian chạy-hoặc trong C ++, các hàm thành viên được gọi phải là ảo và các đối tượng phải được thao tác thông qua các điểm hoặc tham chiếu. Khi thao tác trực tiếp một đối tượng (thay vì thông qua con trỏ hoặc ref-erence), 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 lên một hàm ảo sẽ trở thành ảo. Chúng ta có thể, nhưng không nhất thiết phải lặp lại virtual trong một lớp dẫn xuất. Tôi không khuyên bạn nên lặp lại ảo. Nếu bạn muốn rõ ràng, hãy sử dụng ghi 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 Employee và sử dụng nó để gọi phiên bản phù hợp của hàm ảo print (). Trong triển khai không điển hình, không gian được sử dụng chỉ đủ để chứa một con trỏ (§3.2.3): kỹ thuật nhập thông thường là để trình biên dịch chuyển đổi tên của một hàm ảo thành một chỉ mục trong một bảng con trỏ thành các hàm. Bảng đó thường được gọi là bảng chức năng ảo hoặc đơn giản là vtbl. Mỗi lớp với các chức năng ảo có vtbl riêng xác định các chức năng ảo của nó. Điều này có thể làđược biểu diễn bằng đồ thị như thế này:

Employee: vtbl:

Employee::print()

...

first\_name

second\_name

...

Manager:

first\_name

second\_name

...

group

level

...

**Manager::print()**

...

Manager::print()

vtbl:

vtbl :

Các chức năng trong vtbl cho phép đối tượng được sử dụng chính xác ngay cả khi kích thước của đối tượng và bố cục dữ liệu của nó không được biết đối với người gọi. 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 bình 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 lệnh gọi hàm ordi-nary 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 một vtbl cho mỗi lớp như vậy. Bạn chỉ phải trả chi phí này cho các đối tượng của một lớp có chức năng ảo. Bạn chọn thanh toán này chi phí chỉ khi bạn cần các chức năng bổ sung mà các chức năng ảo cung cấp. Nếu bạn chọn sử dụng solu-tion trường loại thay thế, thì sẽ cần một lượng không gian tương đương cho trường loại.

Một hàm ảo được gọi từ một hàm tạo hoặc một hàm hủy phản ánh rằng đối tượng được cấu trúc 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 Manager :: print () đảm bảo rằng cơ chế ảo không được sử dụng:

void Manager::print() const

{

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

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

// ...

}

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

20.3.4 Kiểm soát ghi đè

Nếu bạn khai báo một hàm trong lớp dẫn xuất có cùng tên và kiểu với hàm ảo trong lớp cơ sở, thì hàm trong lớp dẫn xuất sẽ ghi đè hàm trong lớp cơ sở. Đó là một quy tắc đơn giản và hiệu quả. Tuy nhiên, đối với các cấu trúc phân cấp lớp lớn hơn, có thể khó để chắc chắn rằng bạn thực sự ghi đè chức năng mà bạn định ghi đè. 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; // overr ide f () trong lớp cơ sở

void g (int); // ghi đè g () trong lớp cơ sở

virtual int h (); // ghi đè h () trong lớp cơ sở

};

Điều này minh họa ba lỗi không rõ ràng khi chúng xuất hiện trong hệ thống phân cấp lớp thực, nơi các lớp B0 ... B5 mỗi lớp có nhiều thành viên và nằm rải rác trên nhiều tệp tiêu đề. Ở đây:

• B0 :: f () không phải là ảo, vì vậy bạn không thể ghi đè nó, chỉ ẩn nó (§20.3.5).

• D :: g () không có cùng kiểu 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 hàm nào được gọi là h () trong B0, nếu D :: h () ghi đè lên bất cứ điều 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 điều gì đó hoàn toàn khác đang xảy ra do các khai báo trong các lớp đó. Cá nhân tôi không (dư thừa) sử dụng ảo cho một chức năng nghĩa là ghi đè. Đối với các chương trình nhỏ hơn (đặc biệt là với trình biên dịch có cảnh báo phù hợp về các lỗi phổ biến) thì việc ghi đè được thực hiện đúng cách không khó. Tuy nhiên, đối với các phân cấp lớn hơn, các điều khiển cụ thể hơn sẽ hữu ích:

• ảo: Chức năng có thể bị ghi đè (§20.3.2).

• = 0: Hàm phải ảo và phải được ghi đè (§20.4).

• override: Hàm có nghĩa là ghi đè một hàm ảo trong một lớp cơ sở (§20.3.4.1).

• cuối cùng: Hàm không có nghĩa là bị ghi đè (§20.3.4.2).

Trong trường hợp không có bất kỳ điều khiển nào trong số này, một hàm thành viên không tĩnh sẽ là ảo nếu và chỉ khi nó vượt quá cưỡi một hàm ảo 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 dụng ghi đè cho bảy trong số chín hàm lớp cơ sở ảo có thể gây nhầm lẫn cho người bảo trì.

**//Ghi đè 20.3.4.1**

Chúng tôi có thể nói rõ về mong muốn ghi đè của chúng tôi:

struct D: B5 {

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

void g (int) ghi đè; // error: B0 :: f () nhận đối số kép

ghi đè ảo int h (); // lỗi: không có hàm h () để ghi đè lên id

};

Đưa ra định nghĩa này (và giả sử rằng các lớp cơ sở trung gian B1 ... B5 không cung cấp các hàm tương ứng), cả ba khai báo đều mắc lỗi.

Trong một hệ thống phân cấp lớp lớn hoặc phức tạp với nhiều chức năng ảo, tốt nhất là chỉ sử dụng ảo để giới thiệu một chức năng ảo mới và sử dụng ghi đè trên tất cả các chức năng được dùng làm trình ghi đè.

Sử dụng ghi đè hơi dài dòng nhưng làm rõ ý định của lập trình viên.

Thông số ghi đè xuất hiện 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 lên Ide)

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 tiền tố và ghi đè là hậu tố. Đây là một phần cái giá mà chúng tôi phải trả cho sự tương thích và ổn định trong nhiều thập kỷ.

Mã định nghĩa ghi đè không phải là một phần của kiểu hàm và không thể lặp lại trong định nghĩa ngoài lớp. Ví dụ:

class Derived: public Base {

void f () ghi đè; // OK nếu Base có f ảo ()

void g () ghi đè; // OK nếu Base có g () ảo

};

void Derived :: f () override // error: overr ide out of class

{

// ...

}

void g () // OK

{

// ...

}

Thật kỳ lạ, ghi đè không phải là một từ khóa; nó là những gì được gọi là một từ khóa theo ngữ cảnh. Nghĩa là, ghi đè có một ý nghĩa đặc biệt trong một vài 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 ham mê sự thông minh như vậy; nó làm phức tạp thêm việc bảo trì. Lý do duy nhất khiến việc ghi đè là một từ khóa theo ngữ cảnh, chứ không phải là một từ khóa thông thường, là tồn tại một lượng lớn mã đã sử dụng ghi đè làm mã định danh thông thường trong nhiều thập kỷ. Từ khóa theo ngữ cảnh khác là cuối cùng (§20.3.4.2).

**Ghi đè 20.3.4.1**

Chúng tôi có thể nói rõ về mong muốn ghi đè của chúng tôi:

struct D: B5 {

void f (int) const override; // error: B0 :: f () không phải là ảo

void g (int) override; // error: B0 :: f () nhận đối số kép

vitural int h (); // error: không có hàm h () để ghi đè lên id

};

Đưa ra định nghĩa này (và giả sử rằng các lớp cơ sở trung gian B1 ... B5 không cung cấp

hàm vant), cả ba khai báo đều cho lỗi.

Trong một hệ thống phân cấp lớp lớn hoặc phức tạp với nhiều chức năng ảo, tốt nhất là chỉ sử dụng ảo để giới thiệu một chức năng ảo mới và sử dụng ghi đè trên tất cả các chức năng được dùng làm trình ghi đè.

Sử dụng ghi đè hơi dài dòng nhưng làm rõ ý định của lập trình viên. Thông số ghi đè xuất hiện 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 lên Ide)

ghi đè void f (int) const noexcept; // lỗi cú pháp

void f (int) override const noexcept; // lỗi cú pháp

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

Một chỉ định ghi đè không phải là một phần của kiểu hàm và không thể được lặp lại trong một định nghĩa ngoài lớp. Ví dụ:

class Derived: public Base {

void f () override; // OK nếu Base có f ảo ()

void g () override; // OK nếu Base có ảo g ()

};

void Derived :: f () override // error: overr ide out of class

{

// ...

}

void g () // OK

{

// ...

}

Thật kỳ lạ, ghi đè không phải là một từ khóa; nó là những gì được gọi là một từ khóa theo ngữ cảnh. Nghĩa là, ghi đè có một ý nghĩa đặc biệt trong một vài ngữ cảnh nhưng có thể được sử dụng làm định danh ở những nơi khác. Ví dụ:

int override = 7;

struct Dx: Base {

override int;

override int f ()

{

Return override + :: override;

}

};

Đừng ham mê sự thông minh như vậy; nó làm phức tạp thêm việc bảo trì. Lý do duy nhất khiến việc ghi đè là một từ khóa theo ngữ cảnh, chứ không phải là một từ khóa thông thường, là tồn tại một lượng lớn mã đã sử dụng ghi đè làm mã định danh thông thường trong nhiều thập kỷ. Từ khóa theo ngữ cảnh khác là từ khóa cuối cùng (§20.3.4.2).

**20.3.4.2 cuối cùng**

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 dụng ảo cho các hàm mà chúng tôi muốn người viết các lớp dẫn xuất có thể định nghĩa hoặc xác định 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?

• Ghi đè một hàm có dễ bị lỗi không (tức 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 không ảo để đạt được sự đơn giản của thiết kế và đôi khi là một số hiệu suất (chủ yếu là từ nội tuyến). Thư viện tiêu chuẩn đã đầy ví dụ về điều này.

Hiếm khi hơn, chúng ta sử dụng hệ thống phân cấp lớp bắt đầu bằng các hàm ảo, nhưng sau khi xác định một tập hợp các lớp dẫn xuất, một trong những câu trả lời trở thành '' không ''. Ví dụ, chúng ta có thể tưởng tượng một cây cú pháp trừu tượng cho một ngôn ngữ mà tất cả các cấu trúc ngôn ngữ đã được xác định 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 hàm 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 {// interface class (lớp giao diện)

virtual Type type () = 0;

// ...

};

class If\_statement: public Node {

public:

Type type () override final ; // ngăn chặn việc ghi đè thêm

// ...

};

Trong một hệ thống phân cấp lớp thực tế, sẽ có một số lớp trung gian giữa liên mặt chung (ở đây, Node) và lớp dẫn xuất đại diện cho một cấu trúc ngôn ngữ cụ thể (ở đây, If\_state-ment). Tuy nhiên, điểm chính về ví dụ này là Node :: type () có nghĩa là được ghi đè (đó là lý do tại sao nó được khai báo là ảo) và ghi đè If\_statement :: type () thì không (đó là lý do tại sao nó được khai báo là chính). Sau khi sử dụng final cho một hàm thành viên, nó không thể bị ghi đè nữa và việc cố gắng làm như vậy sẽ xảy ra lỗi. Ví dụ:

Trong một hệ thống phân cấp lớp thực tế, sẽ có một số lớp trung gian giữa liên mặt chung (ở đây, Node) và lớp dẫn xuất đại diện cho một cấu trúc ngôn ngữ cụ thể (ở đây, If\_statement). Tuy nhiên, điểm chính về ví dụ này là Node :: type () có nghĩa là được ghi đè (đó là lý do tại sao nó được tuyên bố là ảo) và ghi đè If\_statement :: type () thì không (đó là lý do tại sao nó được khai báo là cuối cùng). Sau khi sử dụng final cho một hàm thành viên, nó không thể bị ghi đè nữa và việc cố gắng làm như vậy sẽ xảy ra lỗi. Ví dụ:

class Modified\_if\_statement: công khai If\_statement {

công cộng:

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

// ...

};

Chúng ta có thể biến mọi hàm thành viên ảo của một lớp trở thành cuối cùng; chỉ cần thêm cuối cùng sau tên lớp. Ví dụ:

class For\_statement final: public Node {

public :

Gõ kiểu () ghi đè;

// ...

};

class Modified\_for\_statement: public For\_statement {// error: For\_statement là cuối cùng

Gõ kiểu () ghi đè;

// ...

};

Đối với tốt và xấu, việc thêm cuối cùng vào lớp không chỉ ngăn chặn việc ghi đè, nó còn ngăn chặn việc dẫn xuất thêm từ một lớp. Có những người sử dụng cuối cùng để cố gắng đạt được hiệu suất - sau cùng, một người không chức năng ảo nhanh hơn chức năng ảo (có thể bằng 25% trên một triển khai hiện đại) và mang lại nhiều cơ hội hơn cho nội tuyến (§12.1.5). Tuy nhiên, không sử dụng cuối cùng một cách mù quáng như một chất tối ưu hóa; nó ảnh hưởng đến thiết kế phân cấp lớp (thường là tiêu cực) và các cải tiến hiệu suất hiếm khi đáng kể. Thực hiện một số phép đo nghiêm túc trước khi yêu cầu cải thiện hiệu quả. Sử dụng cuối cùng khi nó phản ánh rõ ràng thiết kế phân cấp lớp mà bạn cho là phù hợp. Nghĩa là, sử dụng final để phản ánh nhu cầu ngữ nghĩa. Ví dụ:

class Derived: public Base {

void f () final; // OK nếu Base có f ảo ()

void g () final; // OK nếu Base có ảo g ()

// ...

};

void Derived :: f () final // error: cuối cùng ra khỏi lớp

{

// ...

}

void g () final // OK

{

// ...

}

Giống như ghi đè (§20.3.4.1), cuối cùng là một từ khóa theo ngữ cảnh. Nghĩa là, cuối cùng có một 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 final = 7;

struct Dx: Base {

int final;

int f () final

{

return final + :: final;

}

};

Đừng ham mê sự thông minh như vậy; nó làm phức tạp thêm việc bảo trì. Lý do duy nhất khiến cuối cùng là một từ khóa liên văn bản, chứ không phải là một từ khóa thông thường, là tồn tại một lượng đáng kể mã đã sử dụng cuối cùng làm mã định danh thông thường trong nhiều thập kỷ. Từ khóa theo ngữ cảnh khác được ghi 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:

Struct Base {

void f (int);

};

struct Derived: Base {

void f (double);

};

void use(Derived d){

d.f (1); // gọi Derived :: f (double)

Base & br = d

br.f (1); // gọi Base :: f (int)

}

Điều này có thể khiến mọi người ngạc nhiên và đôi khi chúng tôi muốn quá tải để đảm bảo rằng hàm thành viên phù hợp nhất được sử dụng. Đối với không gian tên, dùng khai báo có thể được sử dụng để thêm một hàm vào một phạm vi. Ví dụ:

struct D2: Base

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

void f (double);

};

void use2 (D2 d)

{

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

Base & br = d

br.f (1); // call Base :: f (int)

}

Đây là một hệ quả đơn giản của việc một lớp cũng được coi là không gian tên (§16.2). Một số khai báo sử dụng có thể mang lại tên từ nhiều lớp cơ sở. Ví dụ:

struct B1 {

void f(int);

};

struct B2 {

void f(double);

};

struct D : B1, B2 {

using B1::f;

using B2::f;

void f(char);

};

void use(D d)

{

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

d.f('a'); // call D::f(char)

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

}

Chúng ta có thể đưa các hàm tạo vào một phạm vi lớp dẫn xuất; xem §20.3.5.1. Một tên được đưa vào phạm vi lớp dẫn xuất bởi một khai báo using có quyền truy cập của nó được xác định bởi vị trí của using-statementara-tion; xem §20.5.3. Chúng ta không thể sử dụng các chỉ thị using 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 vectơ std :: nhưng có kiểm tra phạm vi được đảm bảo. Tôi có thể thử điều này:

template<class 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(siz e\_type i) { if (this−>size()<i) throw rang e\_error{"Vector::check() failed"}; }

};

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}; // error: không có phương thức khởi tạo danh sách khởi tạo

Kiểm tra nhanh sẽ cho thấy rằng Vector không 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 cơ sở hoặc yêu cầu một lớp bất biến chặt chẽ hơn, thì sẽ là một thảm họa nếu kế thừa các hàm tạo. Tuy nhiên, Vector 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:

template<class T>

struct Vector : std::vector<T> {

using vector<T>::vector; // kế thừa các hàm tạo

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(siz e\_type i) { if (this−>size()<i) throw Bad\_index(i); }

};

Vector<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 hoàn toàn tương đương với việc sử dụng nó cho các chức năng 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 {

using B1 :: B1; // khai báo ngầm D1 (int)

string s ; // chuỗi có một hàm tạo mặc định

int x; // chúng tôi ‘‘ quên ’’ cung cấp cho việc khởi tạo x

};

Void test ()

{

Đ1 d {6}; // oops: d.x không được khởi tạo

Đ1 e; // error: D1 không có hàm tạo mặc định

}

Lý do mà D1 :: s được khởi tạo còn D1 :: x thì không là vì hàm tạo kế thừa tương đương với một hàm khởi 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; // chuỗi có một hàm tạo mặc định

int x; // chúng tôi ‘‘ quên ’’ cung cấp cho việc 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):

struct D1: B1 {

using B1 :: B1; // khai báo ngầm D1 (int)

int x {0}; // ghi chú: x được khởi tạo

};

Void test ()

{

Đ1 d {6}; // d.x 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 thành viên dữ liệu nào được thêm vào.

**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. Nghĩa là, nếu kiểu trả về ban đầu là B ∗, thì kiểu trả về của hàm ghi đè có thể là D ∗, với điều kiện B là cơ sở công khai của D. Tương tự, kiểu trả về B & có thể được nới lỏng thành D &. Điều này đôi khi được gọi là quy tắc trả về hiệp phương sai.

Việc nới lỏng này chỉ áp dụng cho các kiểu trả về là con trỏ hoặc tham chiếu, chứ không phải 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 kiểu đối số vì điều đó sẽ dẫn đến vi phạm kiểu. Ngoài các toán hạng để thao tác các biểu thức, lớp cơ sở Expr sẽ cung cấp các phương tiện để tạo các đối tượng biểu thức mới của các kiểu biểu thức khác nhau:

Class Expr {

public:

Expr (); // nhà xây dựng mặc định

Expr (const Expr &); // sao chép cấu trúc uctor

Virtual Expr ∗ new\_expr () = 0;

ảo Expr ∗ clone () = 0;

// ...

};

Ý tưởng là new\_expr () tạo một đối tượng mặc định của kiểu biểu thức và clone () làm cho đối tượng đó trở nên phổ biến. Cả hai sẽ trả về một đối tượng của một số lớp cụ thể có nguồn gốc từ Expr. Chúng không bao giờ chỉ có thể trả về một '' Expr đơn giản '' bởi 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 clone () 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 (); }

Cond ∗ clone () override {return new Cond (∗ this); }

// ...

};

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

void user(Expr∗ p)

{

Expr∗ p2 = p−>new\_expr();

// ...

}

Con trỏ được gán cho p2 được khai báo để trỏ đến một '' Expr thuần túy '', nhưng nó sẽ trỏ đến một đối tượng có kiểu dẫn xuất từ Expr, chẳng hạn như Cond. Kiểu trả về của Cond :: new\_expr () và Cond :: clone () là Cond ∗ chứ không phải Expr ∗. Điều này cho phép một Cond được sao chép mà không làm mất thông tin loại. Tương tự, một Phép cộng lớp dẫn xuất sẽ có một bản sao () trả về Phép cộng ∗. Ví dụ:

void user2(Cond∗ pc, Addition∗ pa)

{

Cond∗ p1 = pc−>clone();

Addition∗ p2 = pa−>clone();

// ...

}

Nếu chúng ta sử dụng clone () cho một Expr, chúng ta 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() retur ns an Expr\*

// ...

}

Vì các hàm như new\_expr () và clone () là ảo và chúng (gián tiếp) xây dựng các đối tượng, chúng thường được gọi là các hàm tạo ảo. Mỗi đơn giản chỉ sử dụng một hàm tạo để tạo một

Để tạo một đối tượng, một phương thức khởi tạo cần có kiểu chính xác của đối tượng mà nó sẽ tạo. Do đó, một phương thức khởi tạo không thể là ảo. Hơn nữa, một hàm tạo không phải là một hàm hoàn toàn bình thường. Nói cách khác, nó tương tác với các quy trình quản lý bộ nhớ theo những cách mà các chức năng thành viên thông thường không làm. 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. Ival\_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 Employee ở chỗ chúng hữu ích như chính chúng, làm giao diện cho các lớp dẫn xuất và là một phần của việc triển khai các lớp dẫn xuất. Đối với các lớp như vậy, kỹ thuật-niques 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, chẳng hạn như một lớp Shape, đại diện cho các khái niệm trừu tượng mà đối tượng không thể tồn tại. Hình dạng chỉ có ý nghĩa như là cơ sở của một số lớp bắt nguồn 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 {

public:

virtual void rotate (int) {throw runtime\_error {"Shape :: xoay"}; } // không phù hợp

virtual void draw () const {ném runtime\_error {"Shape :: draw"}; }

// ...

};

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

Shape s; // ngớ ngẩn: ‘‘ hình dạng không định hình ’’

Thật là ngớ ngẩn vì mọi thao tác trên s sẽ dẫn đến lỗi.

Một giải pháp thay thế tốt hơn là khai báo các hàm ảo của lớp Shape là các hàm ảo thuần túy.

Một hàm ảo được ‘‘ tạo thuần túy ’’ bởi ‘‘ bộ khởi tạo giả ’’ = 0:

lớp Shape {// lớp trừu tượng

public:

virsual void rotute (int) = 0; // hàm ảo thuần túy

voisual void draw () const = 0; // hàm ảo thuần túy

virsual bool is\_closed () const = 0; // hàm ảo thuần túy

// ...

Virsual ̃Shape (); // 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:

Shape ; // 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 đối với một lớp trừu tượng là có một trình hủy ảo (§3.2.4, §21.2.2). Bởi vì giao diện được cung cấp bởi một lớp trừu tượng không thể được sử dụng để tạo các đối tượng bằng cách sử dụng một hàm tạo, các lớp trừu tượng thường không có các hàm tạo. Một lớp trừu tượng chỉ có thể được sử dụng làm giao diện cho các lớp khác. Ví dụ:

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

class Circle : public Shape {

public:

void rotate(int) override { }

void draw() const override;

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

Circle(Point p, int r);

private:

Point center;

int radius;

};

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 đè idden (draw and rotate not overr idden)...

};

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

Đa giác vẫn còn trừu tượng vì chúng ta đã không ghi đè draw () và xoay (). Chỉ khi điều đó được thực hiện xong, chúng ta mới có một lớp mà từ đó chúng ta có thể tạo các đối tượng:

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:

virtural int open (int opt) = 0;

virtural int close (int opt) = 0;

virtural int read (char ∗ p, int n) = 0;

virtual int write (const char ∗ p, int n) = 0;

int ioctl (int ...) = 0; // điều khiển I / O thiết bị

virtual ̃Character\_device () {} // hàm hủy ảo

};

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

20.5 Kiểm soát truy cập

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

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

• Nếu nó được bảo vệ, tên của nó chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của lớp mà nó được khai báo và bởi các hàm thành viên và bạn bè của các lớp dẫn xuất từ lớp này (xem §19.4).

• Nếu nó là công khai, 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 triển khai lớp (bạn bè và thành viên của nó), các hàm triển khai một lớp dẫn xuất (bạn bè và thành viên của lớp dẫn xuất) và các hàm khác. Điều này có thể được trình bày bằng đồ thị:

người dùng chung

các hàm thành viên và bạn bè của lớp dẫn xuất

chức năng thành viên riêng và bạn bè

public:

protected:

private:

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 có các hàm thành viên riêng, kiểu, hằng số, v.v., cũng như các thành viên dữ liệu riêng. Ví dụ, một lớp danh sách không ủy thác hiệu quả thường yêu cầu cấu trúc dữ liệu để theo dõi các phần tử. Một danh sách là không cần thiết nếu nó không yêu cầu sửa đổi các phần tử của nó (ví dụ: bằng cách yêu cầu các loại phần tử phải có các trường liên kết). Thông tin và cấu trúc dữ liệu được sử dụng để tổ chức danh sách có thể được giữ kín:

template<class T>

class List {

public:

void insert(T);

T get();

// ...

private:

struct Link { T val; Link∗ next; };

struct Chunk {

enum { chunk\_siz e = 15 };

Link v[chunk\_siz e];

Chunk∗ next;

};

Chunk∗ allocated;

Link∗ free;

Link∗ get\_free();

Link∗ head;

};

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

template<class T>

void List<T>::insert(T val)

{

Link∗ lnk = get\_free();

lnk−>val = val;

lnk−>next = head;

head = lnk;

}

template<class T>

T List<T>::g et()

{

if (head == 0)

throw Underflow{}; // Dưới luồng là lớp ngoại lệ của tôi

Link∗ p= head;

head = p−>next;

p−>next = free;

free = p;

return p−>val;

}

Như thường lệ, định nghĩa của các hàm hỗ trợ (ở đây, riêng tư) phức tạp hơn một chút:

template<class T>

typename List<T>::Link∗ List<T>::get\_free()

{

if (free == 0) {

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

}

Link∗ p = free;

free = free−>next;

return p;

}

Phạm vi Danh sách <T> được nhập bằng cách nói Danh sách <T> :: trong định nghĩa hàm thành viên. Tuy nhiên, vì kiểu trả về của get\_free () được đề cập trước khi tên List <T> :: get\_free () được đề cập, nên tên đầy đủ List <T> :: Link phải được sử dụng thay vì tên viết tắt Link. Cách thay thế là sử dụng ký hiệu hậu tố cho các kiểu trả về (§12.1.4):

template<class T>

auto List<T>::get\_free() −> Link∗

{

// ...

}

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

template<typename T>

void would\_be\_meddler(List<T>∗ p)

{

List<T>::Link∗ q = 0; // error : List<T>::Link is private

// ...

q=p−>free; // error : List<T>::free is private

// ...

if (List<T>::Chunk::chunk\_siz e > 31) { // error : List<T>::Chunk::chunk\_size is private

// ...

}

}

Trong một lớp, các thành viên theo mặc định là riêng tư; trong một cấu trúc, các thành viên theo mặc định là công khai (§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:

template<class T>

struct Link2 {

T val;

Link2∗ next;

};

template<class T>

class List {

private:

Link2<T>∗ free;

// ...

};

Liên kết được tham số hóa ngầm định với tham số T. của Danh sách <T>. Đối với Link2, chúng ta phải làm rõ điều đó.

Nếu một loại 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 bản thân lớp lồng nhau không hữu ích và lớp bao quanh cần quyền truy cập vào biểu diễn của nó, thì việc khai báo lớp thành viên là bạn (§19.4.2) có thể là một ý tưởng hay:

template<class T> class List;

template<class T>

class Link3 {

friend class List<T>; // only List<T> can access Link<T>

T val;

Link3∗ next;

};

template<class T>

class List {

private:

Link3<T>∗ free;

// ...

};

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

class S {

public:

int m1;

public:

int m2;

};

Trình biên dịch có thể quyết định m2 đứng trước m1 trong bố cục của một đối tượng S. 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, không sử dụng nhiều công cụ chỉ định truy cập cho các thành viên dữ liệu mà không có lý do chính đáng.

**20.5.1 Thành viên được bảo vệ**

Khi thiết kế hệ thống phân cấp lớp, đôi khi chúng tôi cung cấp các hàm được thiết kế để được sử dụng bởi những người cố vấn của các lớp dẫn xuất nhưng không được sử dụng 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. Ví dụ:

Class buffer(bo dem lop ){

public:

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

// ...

protectic:

char & access (int i); // quyền truy cập bỏ chọn

// ...

};

class Circular\_buffer: public Buffer {

public:

void reallocate (char ∗ p, int s); // thay đổi vị trí và kích thước

// ...

};

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

// ...

}

void f (Bộ đệm & b)

{

b [3] = 'b'; // OK (đã chọn)

b.access (3) = 'c'; // error: 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:

Class buffer {

protected:

char a [128];

// ...

};

class Linked\_buffer: public Buffer {

// ...

};

class Circular\_buffer: public Buffer {

// ...

void f (Linked\_buffer ∗ p)

{

a [0] = 0; // OK: truy cập vào thành viên được bảo vệ của chính Circular\_buffer

p−> a [0] = 0; // error: quyền truy cập vào thành viên được bảo vệ thuộc loại khác

}

};

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

**20.5.1.1 Sử dụng các Thành viên 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"" Public public. "" Các thành viên và bạn bè triển khai các hoạt động trên lớp sẽ hoạt động trên các đối tượng của lớp thay mặt cho những người dùng này. 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 được tuyên bố là được bảo vệ dễ bị lạm dụng hơn nhiều so với các thành viên được tuyên bố là ở chế độ riêng tư. Trong par-ticular, việc khai báo các thành viên dữ liệu được bảo vệ thường là một lỗi thiết kế. Đặt một lượng đáng kể dữ liệu trong một lớp chung cho tất cả các lớp dẫn xuất sử dụng khiến dữ liệu đó có thể 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 kiếm mỗi khi sử dụng. Do đó, dữ liệu được bảo vệ trở thành một vấn đề bảo trì phần mềm.

May mắn thay, bạn không muốn sử dụng dữ liệu được bảo vệ; private 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. Tuy nhiên, không có phản đối nào trong số này là đáng kể đối với các chức năng của thành viên được bảo vệ; được bảo vệ là một cách tốt để chỉ định các hoạt động để sử dụng trong các lớp dẫn xuất. Ival\_slider trong §21.2.2 là một thử nghiệm của điều này. Nếu lớp triển khai là private trong ví dụ này, thì việc dẫn xuất thêm sẽ không khả thi. Mặt khác, việc đưa ra các căn cứ cung cấp thông tin chi tiết thực hiện một cách công khai lại dẫn đến những sai lầm và sử dụng sai mục đích.

**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 là riêng tư, được bảo vệ hoặc công khai. Ví dụ:

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

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

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

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

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

• Các cơ sở riêng hữu ích nhất khi xác định một lớp bằng cách giới hạn giao diện cho một cơ sở để có thể cung cấp các đảm bảo mạnh mẽ hơn. Ví dụ, B là một chi tiết triển khai của Z. Mẫu Vector con trỏ bổ sung kiểm tra kiểu cho cơ sở Vector <void ∗> của nó từ §25.3 là một ví dụ điển hình.

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

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

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

Mọi người mong đợi các lớp cơ sở là công khai (nghĩa là thể hiện mối quan hệ kiểu con), do đó, việc không có bộ chỉ định truy cập cho một cơ sở có thể sẽ gây ngạc nhiên cho một lớp chứ không phải cho một cấu trúc. Bộ định nghĩa truy cập cho một lớp cơ sở kiểm soát quyền truy cập vào các thành viên của lớp cơ sở và việc chuyển đổi con trỏ và tham chiếu từ kiểu lớp dẫn xuất sang kiểu lớp cơ sở. Hãy xem xét một lớp D bắt nguồn từ một lớp cơ sở B:

• Nếu B là cơ sở tư nhân, các thành viên công khai và được bảo vệ của nó chỉ có thể được sử dụng bởi các func-tions thành viên và bạn bè của D. Chỉ bạn bè và thành viên của D mới có thể chuyển đổi D ∗ thành B ∗.

• Nếu B là một cơ sở được bảo vệ, 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 D và bởi các chức năng thành viên và bạn bè của các lớp bắt nguồn từ D. Chỉ bạn bè và thành viên của D và bạn bè và thành viên của các lớp dẫn xuất từ D có thể chuyển aD ∗ thành B ∗.

• Nếu B là một cơ sở công cộng, các thành viên công cộng của nó có thể được sử dụng cho bất kỳ chức năng nào. Ngoài ra, các thành viên ủng hộ của nó có thể được sử dụng bởi các thành viên và bạn bè của D và các thành viên và bạn bè của các lớp có nguồn gốc từ D. Bất kỳ hàm nào cũng có thể chuyển đổi từ D ∗ sang B ∗. ). 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. Để có ví dụ, hãy xem Ival\_slider trong §21.2.2.

**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:

struct B {

int m;

static int sm;

// ...

};

class D1 : public virtual B { /\* ... \*/ } ;

class D2 : public virtual B { /\* ... \*/ } ;

class D12 : public D1, private D2 { /\* ... \*/ };

D12∗ pd = new D12;

B∗ 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:

class X1 : public B { /\* ... \*/ } ;

class X2 : public B { /\* ... \*/ } ;

class XX : public X1, public X2 { /\* ... \*/ };

XX∗ pxx = new XX;

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

int i2 = pxx−> sm; // OK: chỉ có một B :: sm trong XX (sm là thành viên tĩnh)

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

Không thể sử dụng khai báo sử dụng (§14.2.2, §20.3.5) để truy cập thông tin bổ sung. Nó chỉ đơn giản là một cơ chế để làm cho thông tin có thể truy cập được thuận tiện hơn khi sử dụng. Mặt khác, khi đã có quyền truy cập, nó có thể được cấp cho những người dùng khác. Ví dụ:

class B {

private:

int a;

protected:

int b;

public:

int c;

};

class D : public B {

public:

using B::a; // error : B::a is private

using B::b; // làm cho B :: b có sẵn công khai thông qua D

Khi một khai báo sử dụng 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, các tiện ích thường được cung cấp bởi một lớp. Ví dụ:

class BB : private B { // cấp quyền truy cập cho B :: b và B :: c, nhưng không cấp cho B :: a

public:

using B::b;

using B::c;

};

**20.6 Trỏ tới thành viên**

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 toán tử -> ∗ và. ∗ được cho là C ++ chuyên dụng nhất và ít được sử dụng nhất các toán tử. Sử dụng ->, chúng ta có thể truy cập một thành viên của lớp, m, bằng cách đặt tên cho nó: p−> m. Sử dụng -> ∗, chúng ta 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 tới thành viên không thể được gán cho void ∗ 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 nhiều 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 hiển thị trên màn hình cần được chuẩn bị để phản hồi. Ngoài ra, những 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:

class Std\_interface {

public:

virtual void start() = 0;

virtual void suspend() = 0;

virtual void resume() = 0;

virtual void quit() = 0;

virtual void full\_size() = 0;

virtual void small() = 0;

virtual ̃Std\_interface() {}

};

Ý nghĩa chính xác của mỗi thao tác được xác định bởi đối tượng mà nó được gọi. Thông thường, có một lớp phần mềm nằm giữa người hoặc chương trình đưa ra yêu cầu và đối tượng nhận yêu cầu. Điều đặc biệt là, các lớp phần mềm trung gian như vậy không cần phải biết bất kỳ điều gì về các hoạt động riêng lẻ như resume () 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 là truyền dữ liệu đại diện cho hoạt động được gọi từ nguồn của yêu cầu đến người nhận. Một cách đơn giản để thực hiện đó là gửi một chuỗi đại diện cho hoạt động được gọi. Ví dụ, để gọi Susan (), chúng ta có thể gửi chuỗi "Susan". 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 (). Howev er, trong khi một số nguyên có thể thuận tiện cho máy móc xử lý, nó có thể trở nên khá khó hiểu đối với con người. Chúng ta vẫn phải viết mã để xác định rằng 2 có nghĩa là đình chỉ () và gọi lệnh 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. Hãy xem xét Std\_interface. Nếu tôi muốn gọi hàm Susan () cho một số đối tượng mà không đề cập trực tiếp đến Susan (), tôi cần một con trỏ đến thành viên tham chiếu đến Std\_interface :: Susan (). Tôi cũng cần một con trỏ hoặc tham chiếu đến đối tượng mà tôi muốn tạm ngưng. Hãy xem xét một ví dụ nhỏ:

using Pstd\_mem = void (Std\_interface :: ∗) (); // kiểu con trỏ đến thành viên

void f (Std\_interface ∗ p)

{

Pstd\_mem s = & Std\_interface :: tạm ngưng; // con trỏ để tạm ngưng ()

p−> Susan (); // gọi trực tiếp

p -> ∗ s (); // gọi thông qua con trỏ đến thành viên

}

Một con trỏ đến thành viên có thể nhận được bằng cách áp dụng toán tử address-of, &, cho tên thành viên lớp đủ điều kiện, ví dụ: & Std\_interface :: pause. Một biến kiểu ‘‘ con trỏ tới thành viên của lớp X ’’ được khai báo bằng cách sử dụng bộ khai báo có dạng X :: ∗. Việc sử dụng một bí danh để bù đắp cho sự thiếu dễ đọc của cú pháp bộ khai báo C là typi-cal. Tuy nhiên, hãy lưu ý cách bộ khai báo X :: ∗ khớp chính xác với bộ khai báo ∗ truyền thống. Con trỏ tới thành viên m có thể được sử dụng kết hợp với một đối tượng. Các toán tử -> ∗ và. ∗ cho phép người lập trình thể hiện các kết hợp như vậy. Ví dụ, p -> ∗ m liên kết m với đối tượng được trỏ tới bởi p, và obj. ∗ m liên kết m với đối tượng obj. Kết quả có thể được sử dụng phù hợp với loại của m. Không thể lưu trữ kết quả của phép toán a -> ∗ hoặc a. ∗ để sử dụng sau này. Thông thường, nếu chúng ta biết mình muốn gọi thành viên nào, chúng ta sẽ gọi nó trực tiếp thay vì gây rối với các con trỏ đến 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 tới 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 đối với một đối tượng có kiểu phù hợp, nó mang lại một cái gì đó xác định một thành viên cụ thể của một đối tượng cụ thể. Lời gọi p -> ∗ s () có thể được biểu diễn bằng đồ thị như sau:

vtbl:

X::suspend

X::start

p

S:

Bởi vì con trỏ tới một thành viên ảo (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 đến 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 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 là ảo. Ví dụ, khi chúng ta gọi hàm Susan () thông qua một con trỏ đến hàm, chúng ta sẽ nhận được đúng hàm Susan () cho đối tượng mà con trỏ tới hàm được áp dụng. Đây là một khía cạnh thiết yếu của con trỏ đến các hàm. Khi viết 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:

map<string,Std\_interface∗> variable;

map<string,Pstd\_mem> operation;

void call\_member(string var, string oper)

{

(variable[var]−>∗operation[oper])(); // var.oper()

}

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

class Task {

// ...

static void schedule();

};

void (∗p)() = &Task::schedule; // OK

void (Task::∗ pm)() = &Task::schedule; // error : ordinar y pointer assigned

// to pointer to member

**20.6.2 Con trỏ đến các thành viên dữ liệu**

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 ∗ val;

int i;

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

int f1 (int);

void f2 ();

C (const char ∗ v) {val = v; }

};

using Pmfi = void (C :: ∗) (int); // con trỏ tới hàm thành viên của C lấy int

sử dụng Pm = const char ∗ C :: ∗; // con trỏ tới thành viên dữ liệu char \* của C

void f (C & z1, C & z2)

{

C ∗ p = & z2;

Pmfi pf = & C :: print;

pm pm = & C :: val;

z1.print (1);

(z1. ∗ pf) (2);

z1. ∗ pm = "nv1";

p -> ∗ pm = "nv2";

z2.print (3);

(p -> ∗ pf) (4);

pf = & C :: f1; // lỗi: loại trả về n không khớp

pf = & C :: f2; // error: kiểu đối số không khớp

pm = & C :: i; // error: gõ không khớp

pm = pf; // error: gõ 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 gán một cách an toàn một con trỏ cho một thành viên của lớp cơ sở cho một con trỏ tới một mem-ber của một lớp dẫn xuất, nhưng không phải ngược lại. Tính chất này thường được gọi là độ tương phản.Ví dụ:

class Text : public Std\_interface {

public:

void start();

void suspend();

// ...

virtual void print();

private:

vector s;

};

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

void (Text::∗pmt)() = &Std\_interface::start; // OK

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 đều tồn tại để bảo vệ

đảm bảo 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\_interface nào, và hầu hết các đối tượng như vậy có lẽ không thuộc loại Te xt. Do đó, chúng không có thành viên Text :: print 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.

Chương 21: CẤU TRÚC PHÂN CẤP LỚP

21.1 Giới thiệu

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

21.2 Thiết kế cấu trúc phân cấp lớp

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

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

**User Ival\_box:**

Value

**(via ‘‘system’’) application**

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

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

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

21.2.1 Kế thừa triển khai

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

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

**class Ival\_box {**

**protected:**

**int val;**

**int low, high;**

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

**public:**

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

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

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

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

**virtual void prompt() { }**

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

**virtual ̃Ival\_box() {};**

**};**

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

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

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

**{**

**pb−>prompt(); // aler t user**

**// ...**

**int i = pb−>get\_value();**

**if (pb−>was\_chang ed()) {**

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

**}**

**else {**

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

**}}**

**void some\_fct()**

**{**

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

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

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

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

**}**

Hầu hết mã ứng dụng được viết dưới dạng (con trỏ tới) Ival\_boxes thuần túy như cách tương tác (). Bằng cách đó, ứng dụng không cần biết về số lượng lớn các biến thể tiềm năng của khái niệm Ival\_box. Kiến thức của các lớp chuyên biệt như vậy bị cô lập trong tương đối ít các hàm tạo ra các đối tượng như vậy. Điều này cách ly người dùng khỏi những thay đổi trong việc triển khai các lớp dẫn xuất. Hầu hết các mã có thể bị lãng quên bởi thực tế là có nhiều loại Ival\_box khác nhau.

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

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

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

**private:**

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

**public:**

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

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

**void prompt() override;// ghi đè**

**};**

Các thành viên dữ liệu của Ival\_box đã được khai báo được bảo vệ để cho phép truy cập từ các lớp dẫn xuất. Do đó, Ival\_slider :: g et\_value () có thể gửi một giá trị vào Ival\_box :: val. Thành viên được bảo vệ có thể truy cập được từ các thành viên của chính một lớp và từ các thành viên của các lớp dẫn xuất, nhưng không phải đối với người dùng thông thường (xem §20.5).

Ngoài Ival\_slider, chúng tôi sẽ xác định các biến thể khác của khái niệm Ival\_box. Chúng có thể bao gồm Ival\_dial, cho phép bạn chọn một giá trị bằng cách xoay một núm; Flashing\_ival\_slider, nhấp nháy khi bạn yêu cầu nó nhắc (); và Popup\_ival\_slider, phản hồi với prompt () bằng cách xuất hiện ở một số nơi nổi bật, do đó khiến người dùng khó có thể bỏ qua. Chúng ta sẽ lấy đồ họa từ đâu? Hầu hết các hệ thống giao diện người dùng cung cấp một lớp xác định các thuộc tính cơ bản của một thực thể trên màn hình. Vì vậy, nếu chúng tôi sử dụng hệ thống từ ‘‘ Big Bucks Inc. ’’, chúng tôi sẽ phải làm cho mỗi lớp Ival\_slider, Ival\_dial, v.v., của chúng tôi trở thành một loại BBwid-get. Điều này đơn giản nhất sẽ đạt được bằng cách viết lại Ival\_box của chúng tôi để nó bắt nguồn từ BBwidget. Theo cách đó, tất cả các lớp của chúng ta kế thừa tất cả các thuộc tính của một BBwidget. Ví dụ: mọi Ival\_box đều có thể được đặt trên màn hình, tuân theo các quy tắc kiểu đồ họa, được thay đổi kích thước, được kéo xung quanh, v.v., phù hợp với tiêu chuẩn do hệ thống BBwidget thiết lập. Hệ thống phân cấp lớp của chúng ta sẽ trông như thế này:

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

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

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

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

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

hoặc bằng đồ thị:

BBwidget

Ival\_box

Ival\_slider Ival\_dial

Popup\_ival\_slider Flashing\_ival\_slider

21.2.1.1 Phê bình

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

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

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

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

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

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

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

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

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

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

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

21.2.2 Kế thừa giao diện

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

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

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

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

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

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

**class Ival\_box {**

**public:**

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

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

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

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

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

**virtual ̃Ival\_box() { }**

**};**

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

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

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

**public:**

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

**̃Ival\_slider() override;**

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

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

**// ...**

**protected:**

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

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

**private:**

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

**};**

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

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

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

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

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

**void f(Ival\_box∗ p)**

**{**

**// ...**

**delete p;**

**}**

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

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

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

**class Ival\_slider**

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

**class Ival\_dial**

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

**class Flashing\_ival\_slider**

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

**class Popup\_ival\_slider**

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

**hoặc bằng đồ thị:**

BBwidget Ival\_box Bbwidget

Ival\_slider Ival\_dial

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

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

21.2.3 Triển khai Thay thế

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

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

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

**class Ival\_slider**

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

**class Ival\_slider**

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

**// ...**

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

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

**class BB\_ival\_slider**

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

**class CW\_ival\_slider**

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

**// ...**

**hoặc bằng đồ thị:**

**Bbwidget Ival\_box Cwwidget**

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

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

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

**class Ival\_slider**

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

**class BB\_ival\_slider**

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

**class CW\_ival\_slider**

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

**// ...**

**hoặc bằng đồ thị:**

**Ival\_box**

**Bbwidget Ival\_slider CWwidget**

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

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

**class BB\_ival\_slider**

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

**class CW\_ival\_slider**

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

**hoặc bằng đồ thị:**

**BBwidget Ival\_box Cwwidget**

**Bbslider Ival\_slider Cwslider**

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

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

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

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

**class Ival\_slider**

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

**class Ival\_dial**

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

**class Flashing\_ival\_slider**

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

**class Popup\_ival\_slider**

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

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

**class BB\_ival\_slider**

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

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

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

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

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

**class CW\_ival\_slider**

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

**// ...**

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

Ival\_box

Ival\_slider Ival\_dial

Ipopup iflash

BBslider Bbslider CWsl CWsl BBb&w CWsl

Bbislider BBipop Cwipop Cwifl BBifl Cwislider

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

21.2.3.1 Phê bình

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

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

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

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

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

itance (§3.2.3, §20.1).

21.2.4 Bản địa hóa Tạo đối tượng

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

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

**class Ival\_maker {**

**public:**

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

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

**// ...**

**};**

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

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

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

**public:**

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

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

**// ...};**

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

**public:**

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

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

**// ...**

**};**

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

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

**{**

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

**}**

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

**{**

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

**}**

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

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

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

**// ...**

**}**

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

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

**void driver()**

**{**

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

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

**}**

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

21.3 Thừa kế nhiều

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

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

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

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

21.3.1 Nhiều giao diện

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

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

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

21.3.2 Nhiều lớp triển khai

Hãy xem xét một mô phỏng của các thiên thể quay quanh Trái đất, trong đó các vật thể quay xung quanh được thể hiện như đối tượng của lớp Vệ tinh. Một đối tượng Vệ tinh sẽ chứa quỹ đạo, kích thước, hình dạng, albedo, tham số mật độ, v.v. và cung cấp các hoạt động để tính toán quỹ đạo, sửa đổi thuộc tính, v.v. Ví dụ về vệ tinh sẽ là đá, mảnh vỡ từ các phương tiện vũ trụ cũ, vệ tinh liên lạc, và Trạm vũ trụ quốc tế. Các loại vệ tinh này sẽ là đối tượng của các lớp bắt nguồn từ Vệ tinh. Các lớp dẫn xuất như vậy sẽ thêm các thành viên và chức năng dữ liệu và sẽ ghi đè một số chức năng ảo của Vệ tinh thành

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

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

**public:**

**// ...**

**};**

hoặc bằng đồ thị:

Satellite Displayed

Comm\_sat

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

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

**{**

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

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

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

**}**

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

**void highlight(Displayed∗);**

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

**void g(Comm\_sat∗ p)**

**{**

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

**Pos x = center\_of\_gravity(p); // chuyển một con trỏ đến phần Satellite the Comm\_sat**

**}**

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

**class Satellite {**

**public:**

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

**// ...**

**};**

**class Displayed {**

**public:**

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

**// ...**

**};**

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

**public:**

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

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

**// ...**

**};**

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

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

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

**21.3.3 Độ phân giải mơ hồ**

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

**class Satellite { // vũ trụ**

**public:**

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

**// ...**

**};**

**class Displayed {**

**public:**

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

**// ...**

**};**

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

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

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

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

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

**}**

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

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

**public:**

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

**{**

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

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

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

**}**

**// ...**

**};**

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

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

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

**public:**

**void draw()**

**{**

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

**// ... nội dung riêng ..**

**}**

**// ...**

**};**

Hoặc đồ thị:

**Satellite Displayed**

**Comm\_sat**

**Telstar**

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

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

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

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

**class Satellite : public SimObj {**

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

**public:**

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

**// ...**

**};**

**class Displayed : public Widget {**

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

**public:**

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

**// ...**

**};**

Hoặc đồ thị :

**SimObj Widget**

**Satellite Displayed**

**Comm\_sat**

**Telstar**

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

**class Window {**

**public:**

**void draw(); //** hình ảnh hiển thị

**// ...**

**};**

**class Cowboy {**

**public:**

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

**// ...**

**};**

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

**// ...**

**};**

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

**struct WWindow : Window {**

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

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

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

**};**

**struct CCowboy : Cowboy{**

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

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

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

**};**

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

**public:**

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

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

**// ...**

**};**

**Hoặc đồ thị:**

Window Cowboy

Wwindow CCowboy

Cowboy\_window

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

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

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

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

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

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

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

**virtual ̃Storable() { }**

**};**

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

**class Transmitter : public Storable {**

**public:**

**void write() override;**

**// ...**

**};**

**class Receiver : public Storable {**

**public:**

**void write() override;**

**// ...**

**};**

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

**public:**

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

**void read() override;**

**void write() override;**

**// ...**

**};**

Hoặc đồ thị:

Storable

Receiver Transmitter

Radio

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

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

[2] Đặt dữ liệu trong một lớp cơ sở.

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

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

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

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

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

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

Receiver

Transmitter



Radio



Storable

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

21.3.5.1 Xây dựng cơ sở ảo

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

**struct V {**

**V(int i);**

**// ...**

**};**

**struct A {**

**A(); //** nhà xây dựng mặc định

**// ...**

**};**

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

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

**// ...**

**};**

**class C : virtual V {**

**public:**

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

**// ...**

**};**

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

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

**//** ngầm nhận cơ sở ảo A từ B

**public:**

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

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

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

**// ...**

**};**

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

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

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

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

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

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

**class Window {**

**public:**

**//** vật liệu cơ bản

**virtual void draw();**

**};**

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

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

//nội dung đồng hồ

**protected:**

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

**public:**

**void draw() override;**

**};**

Hoặc đồ thị:

**Window**

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

**Clock**

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

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

**{**

**Window::draw();**

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

**}**

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

**{**

**Window::draw();**

**own\_draw(); //** hiển thị menu

**}**

**void Clock::draw()**

**{**

**Window::draw();**

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

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

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

**}**

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

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

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

**Class BB\_ival\_slider**

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

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

**{**

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

**};**

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

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

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

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

**class BB\_ival\_slider**

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

**class Popup\_ival\_slider**

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

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

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

Hoặc đồ thị:

Ival\_slider BBslider

Popup\_ival\_slider BB\_ival\_slider

BB\_popup\_ival\_slider

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

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

**class BB\_ival\_slider**

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

**class Popup\_ival\_slider**

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

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

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

Hoặc đồ thị:

Ival\_slider Ival\_lider BBslider

Popup\_ival\_slider BB\_ival\_slider

BB\_popup\_ival\_slider

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

**void f(Ival\_slider∗ p);**

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

**{**

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

**}**

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

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

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

**// ...**

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

**// ...**

**}**

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

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

**21.3.6.1 Ghi đè các hàm cơ sở ảo**

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

**class Window {**

**// ...**

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

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

**};**

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

**// ...**

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

**};**

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

**// ...**

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

**};**

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

**// ...**

**};**

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

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

**// ...**

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

**};**

**Hoặc đồ thị:**

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

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

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

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

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

21.4 Lời khuyên

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

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

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

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

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

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

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

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

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

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