**CHƯƠNG 20 : LỚP DẪN XUẤT**

**20.1 Giới thiệu**

Từ Simula, C ++ mượn ý tưởng các lớp và cấu trúc phân cấp lớp. Sử dụng để mô hình hóa các khái niệm trong thế giới của lập trình viên và ứng dụng. C ++ cung cấp cấu trúc ngôn ngữ hỗ trợ trực tiếp các khái niệm thiết kế này. 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**

Trình quản lý có nguồn gốc từ Nhân viên và ngược lại, Nhân viên là một lớp cơ sở cho Trình quản lý. Trình quản lý lớp có các thành viên của lớp Employee (first\_name, phòng ban, v.v.) ngoài các thành viên của chính nó (nhóm, cấp, v.v.). Khởi tạo thường được biểu diễn bằng đồ thị bằng một con trỏ từ lớp dẫn xuất đến lớp cơ sở của nó cho biết rằng lớp dẫn xuất tham chiếu đến cơ sở của nó (thay vì ngược lại):

**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. Tuy nhiên, thuật ngữ này nhầm lẫn cho những người quan sát rằng dữ liệu trong một đối tượng lớp dẫn xuất là một tập siêu dữ liệu của một đối tượng thuộc lớp cơ sở của nó. Một lớp dẫn xuất thường lớn hơn (và không bao giờ nhỏ hơn) so với lớp cơ sở của nó theo nghĩa là nó chứa nhiều dữ liệu hơn và cung cấp nhiều chức năng hơn.

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

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

**Employee: Manager:**

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

Người quản lý (cũng) là một Nhân viên, vì vậy Người quản lý ∗ có thể được sử dụng như một Nhân viên ∗. Tuy nhiên,Nhân viên không nhất thiết phải là Người quản lý, vì vậy một Nhân viên ∗ không thể được sử dụng làm Người quản lý ∗. nếu một lớp Derived có một lớp cơ sở công khai Base, thì một Derived ∗ có thể được gán cho một biến kiểu Base ∗ mà không cần sử dụng chuyển đổi kiểu. Việc chuyển đổi ngược lại, từ Cơ sở ∗ sang Bắt nguồn ∗, phải rõ ràng. 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**

Cấu trúc dữ liệu đơn giản, Nhân viên và Người quản lý. Chúng ta cần cung cấp một kiểu thích hợp với một tập hợp các thao tác phù hợp và chúng ta cần làm như vậy mà không bị ràng buộc vào các chi tiết của một biểu diễn cụ thể. Ví dụ:

**Class NhanVien {**

**public:**

**void print () const;**

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

**// ...**

**private:**

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

**char middle\_initial;**

**// ...**

**};**

**Class Quanly: public NhanVien{**

**public:**

**void print () const;**

**// ...**

**};**

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. Người ta sẽ kiểm tra mọi tệp nguồn của chương trình hoàn chỉnh cho các lớp dẫn xuất, sau đó kiểm tra mọi chức năng của các lớp đó, sau đó tìm mọi lớp dẫn xuất từ ​​các lớp đó, v.v. Điều này thường không thực tế. Ở những nơi có thể chấp nhận được, các thành viên được bảo vệ - thay vì riêng tư - có thể được sử dụng (§20.5).

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

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

**{**

**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 (cơ sở trước thành viên và thành viên trước khi dẫn xuất) và phá hủy từ trên xuống (dẫn xuất trước thành viên và thành viên trước cơ sở);

• Mỗi lớp có thể khởi tạo các thành viên và cơ sở của nó (nhưng không trực tiếp là thành viên hoặc cơ sở của các cơ sở của nó)

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

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

• Độ phân giải của một lệnh gọi hàm ảo, một dynamic\_cast, hoặc một typeid () trong một phương thức khởi tạo hoặc de-structor phản ánh giai đoạn xây dựng và phá hủy (chứ không phải là kiểu của đối tượng chưa được hoàn thành); .

Trong khoa học máy tính, ‘‘ up ’’ và ‘‘ down ’’ có thể rất nhầm lẫn. Trong văn bản nguồn, các định nghĩa của các lớp cơ sở phải xuất hiện trước các định nghĩa của các lớp dẫn xuất của chúng. Điều này ngụ ý rằng đối với các phép kiểm tra nhỏ, các cơ sở xuất hiện phía trên các lớp dẫn xuất trên một màn hình. Hơn nữa, chúng ta có xu hướng vẽ cây với phần gốc ở trên cùng. Tuy nhiên, khi tôi nói về việc xây dựng các đối tượng từ dưới lên, ý tôi là bắt đầu với những gì cơ bản nhất (ví dụ: các lớp cơ sở) và xây dựng những gì phụ thuộc vào đó (ví dụ: các lớp dẫn xuất) sau đó. Chúng tôi xây dựng từ gốc (các lớp cơ sở) về phía lá (các lớp dẫn xuất).

20.3 Phân cấp lớp

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

**Nhân viên 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**

Do đó, như được giải thích chi tiết trong §21.3, C ++ có thể biểu diễn một đồ thị xoay chiều có hướng của các lớp.

20.3.1 Trường loại

Để sử dụng các lớp dẫn xuất không chỉ là một cách viết tắt thuận tiện trong các khai báo, chúng ta phải giải quyết fol-

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 (§3.4, Chương 23).

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

[3] Sử dụng dynamic\_cast (§22.2, §22.6).

[4] Sử dụng các hàm ảo (§3.2.3, §20.3.2).

Trừ khi bạn đã sử dụng cuối cùng (§20.3.4.2), giải pháp 1 dựa vào kiến ​​thức nhiều hơn về các loại

liên quan hơn là có sẵn cho trình biên dịch. Nói chung, không phải là một ý kiến ​​hay nếu cố gắng trở nên thông minh hơn hệ thống kiểu, nhưng (đặc biệt là khi kết hợp với việc sử dụng các khuôn mẫu), 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, tức là danh sách (con trỏ tới) các đối tượng thuộc một số kiểu khác nhau. Giải pháp [3] là một biến thể được hỗ trợ ngôn ngữ của giải pháp [2].

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 depar tment;

// ...};

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, đặc biệt là 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';

// ...

}

}

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

Nói cách khác, sử dụng trường kiểu là một kỹ thuật dễ xảy ra lỗi dẫn đến lỗi bảo trì. Các vấn đề tăng mức độ nghiêm trọng khi kích thước của chương trình tăng lên vì việc sử dụng trường atype gây ra vi phạm các lý tưởng của mô-đun và phân tích dữ liệu. 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.

Có vẻ như bất kỳ dữ liệu phổ biến nào có thể truy cập được từ mọi lớp dẫn xuất, chẳng hạn như trường kiểu, sẽ thúc giục mọi người thêm nhiều dữ liệu như vậy. Do đó, cơ sở chung trở thành kho lưu trữ tất cả các loại ‘‘ thông tin hữu ích ’’. Điều này, đến lượt nó, giúp việc triển khai các lớp cơ sở và lớp dẫn xuất được đan xen vào nhau theo những cách không mong muốn nhất. Trong một hệ thống phân cấp lớp lớn, dữ liệu có thể truy cập (không phải riêng tư) trong một lớp cơ sở chung sẽ trở thành '' biến toàn cục '' của hệ thống phân cấp. Để thiết kế sạch sẽ và bảo trì đơn giản hơn, chúng tôi muốn tách biệt các vấn đề riêng biệt và tránh phụ thuộc lẫn nhau.

20.3.2 Chức năng ảo

Các hàm ảo khắc phục các vấn đề với giải pháp trường kiểu bằng cách cho phép lập trình viên khai báo các hàm trong 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.