

**TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI PHÂN HIỆU**

**THÀNH PHỐ HỒ CHÍ MINH**

**----🙠🕮🙠----**



**BÁO CÁO BÀI TẬP LỚN**

**ĐỀ TÀI:**

Quản lí sinh viên

**GVHD:** GV Trần Thị Dung

**SVTH:** Văn Công Hào (nhóm trưởng)

Trần Tấn Quốc

Trần Quang Thắng

*Hồ Chí Minh, ngày 14 tháng 11 năm 2021*

🙞🖎✍🙜

**LỜI CẢM ƠN**

Để hoàn thành bài báo cáo, chúng em xin gửi lời cảm ơn chân thành đến Cô Trần Thị Dung đã tận tình giảng dạy và cung cấp những kiến thức cần thiết, bổ ích để chúng em hoàn thành tốt báo cáo bài tập.

Trong quá trình làm nghiên cứu và làm việc, nhóm chúng em có tham khảo một số kiến thức để làm cho báo cáo trở nên hoàn thiện hơn.

Với điều kiện thời gian hạn chế của một học viên, báo cáo của chúng em không thể tránh được những thiếu sót. Chúng em rất mong nhận được sự chỉ bảo, đóng góp ý kiến của Cô để chúng em có điều kiện bổ sung, nâng cao kiến thức của bản thân.

Chúng em xin chân thành cảm ơn !

**MỤC LỤC**

[Phần A. Dịch sách 1](#_Toc88159362)

[Chap 16: Các lớp (classes) 1](#_Toc88159363)

[16.1 Giới thiệu (Introduction) 1](#_Toc88159364)

[16.2 Lớp cơ bản (Class basics) 1](#_Toc88159365)

[16.3 Lớp cụ thể (Concrete Classes) 22](#_Toc88159366)

[Chapter 17: Construction, Cleanup, Copy, and Move 28](#_Toc88159367)

[17.1 Giới thiệu (Introduction) 28](#_Toc88159368)

[17.2 Hàm khởi tạo và hàm hủy (Constructors and Destructors) 31](#_Toc88159369)

[17.3 Khởi tạo đối tượng lớp (Class Object Initialization) 37](#_Toc88159370)

[17.4 Khởi tạo thành viên và cơ sở (Member and Base Initialization) 49](#_Toc88159371)

[17.5 Sao chép và di chuyển (Copy and Move) 56](#_Toc88159372)

[17.6 Tạo hoạt động mặc định (Generating Default Operations) 67](#_Toc88159373)

[17.7 Lời khuyên (Advice) 76](#_Toc88159374)

[Chap 18: Quá tải nhà điều hành 77](#_Toc88159375)

[18.1 Giới thiệu (Introduction) 77](#_Toc88159376)

[18.2 Các chức năng của nhà điều hành (Operator Functions) 77](#_Toc88159377)

[18.3 Một loại số phức (A Complex Number Type) 82](#_Toc88159378)

[18.4 Chuyển đổi loại (Type Conversion) 88](#_Toc88159379)

[18.5 Lời khuyên (Advice) 91](#_Toc88159380)

[Chap 19: Nhà điều hành đặc biệt 92](#_Toc88159381)

[19.1 Giới thiệu (Introduction) 92](#_Toc88159382)

[19.2 Nhà điều hành đặc biệt (Special Operators) 92](#_Toc88159383)

[19.3 Một lớp chuỗi (A String Class) 95](#_Toc88159384)

[19.4 Bạn (Friends) 100](#_Toc88159385)

[19.5 Lời khuyên (Advice) 102](#_Toc88159386)

[Chap 20: Các lớp có nguồn gốc 102](#_Toc88159387)

[20.1 Giới thiệu (Introduction) 102](#_Toc88159388)

[20.1 Giới thiệu 103](#_Toc88159389)

[20.2 Các lớp có nguồn gốc (Derived Classes) 104](#_Toc88159390)

[20.3 Phân cấp lớp (Class Hierarchies) 108](#_Toc88159391)

[20.4 Các lớp trừu tượng (Abstract Classes) 124](#_Toc88159392)

[20.5 Kiểm soát truy cập (Access Control) 126](#_Toc88159393)

[20.6 Con trỏ tới Membia (Pointers to Members) 133](#_Toc88159394)

[20.7 Lời khuyên (Advice) 137](#_Toc88159395)

[Chap 21: Lớp Phân Cấp 138](#_Toc88159396)

[21.1 Giới thiệu (Introduction) 138](#_Toc88159397)

[21.2 Cấu Trúc Phân Lớp (Design of Class Hierarchies) 138](#_Toc88159398)

[21.3 Thừa kế nhiều (Multiple Inheritance) 148](#_Toc88159399)

[21.4 Lời khuyên 165](#_Toc88159400)

[Phần 2. Chương trình 165](#_Toc88159401)

[Chương I. Nguyên nhân và mục đích làm chương trình 165](#_Toc88159402)

[Chương II. Cơ sở lý thuyết 165](#_Toc88159403)

[1. Các tính chất trong lập trình hướng đối tượng 165](#_Toc88159404)

[2. Cấu trúc dữ liệu - Danh sách liên kết đơn (Single linked list) 171](#_Toc88159405)

[Chương III. Khái quát về chương trình 177](#_Toc88159406)

[Chương IV. Hướng dẫn sử dụng 182](#_Toc88159407)

# Phần A. Dịch sách

## Chap 16: Các lớp (classes)

### 16.1 Giới thiệu (Introduction)

Classes (các lớp) trong C++ là một công cụ để tạo các kiểu dữ liệu mới và có thể sử dụng như các kiểu dữ liệu tích hợp sẵn. Ngoài ra, các lớp dẫn xuất (derived classes) (§3.2.4, Chapter 20) và các mẫu (templates) (§3.4, Chapter 23) cho phép người lập trình thể hiện mối quan hệ (thứ bậc và tham số) giữa các lớp và tận dụng các mối quan hệ đó. Một lớp (class) là một đại diện cụ thể của một khái niệm hay một ý tưởng. Ví dụ: trong C++ kiểu float tích hợp sẵn các phép toán +,\*,… cung cấp một phép toán đúng cụ thể khái niệm về một số thực. Một lớp (class) là một kiểu dữ liệu do người dùng tự định nghĩa.

### 16.2 Lớp cơ bản (Class basics)

Tóm tắt ngắn gọn về các lớp:

* Một lớp là một kiểu dữ liệu do người dùng định nghĩa.
* Một lớp bao gồm một tập hợp các thành viên: thành viên dữ liệu và các hàm thành viên.
* Các hàm thành viên có thể là: khởi tạo (creation), sao chép (copy), di chuyển (remove) và xóa (cleanup).
* Để truy cập các thành viên trong lớp ta dùng ‘.’ (dot) cho đối tượng và ‘->’ cho con trỏ
* Các toán tử: +, !, [], có thể được định nghĩa cho một lớp.
* Một lớp là một không gian chứa tên các thành viên của nó.
* Các thành viên công khai (public) cung cấp giao diện (hay cung cấp các phương thức) của lớp (class) và các thành viên riêng tư (private) cung cấp triển thông tin chi tiết (hay cung cấp các thuộc tính).
* Một cấu trúc (struct) là một lớp (class) mà các thành viên mặc định là công khai (public).

Ví dụ:

*Class X {*

*private: // chứa các thuộc tính hay thành viên riêng tư*

*int m;*

*public: // chứa các phương thức hay thành viên công khai*

*X(int i = 0) :m{i} {} // phương thức khởi tạo (khởi tạo thành viên dữ liệu m)*

*int mf(int i) //hàm thành viên*

*{*

*int old = m;*

*m = i; //tạo một giá trị mới*

*return old; // trả về giá trị của biến old*

*}*

*};*

*X var {7}; // một biến var kiểu X, có giá trị là 7*

*int uer(X var, X\* ptr)*

*{*

*int x = var.mf(7); // vì var là biến thường nên sử dụng ‘.’ để truy cập thành viên trong class*

*int y = ptr->mf(9);// vì ptr là con trỏ nên sử dụng ‘->’ để truy cập thành viên trong class*

*int z = var.m;// lỗi vì m là thành viên trong private của class nên không thể truy cập từ bên ngoài class*

*}*

#### 16.2.1 Hàm thành viên (Member Functions)

Triển khai một kiểu dữ liệu Date sử dụng cấu trúc struct:

*Struct Date{ //tên của kiểu dữ liệu*

*int d, m, y;*

*};*

*void init\_date(Date &d, int , int , int ); //khởi tạo d*

*void add\_year(Date &d, int n); //thêm n năm vào d*

*void add\_month(Date &d, int n); //thêm n tháng vào d*

*void add\_day(Date &d, int n); //thêm n ngày vào d*

Ở đây không có sự kết nối rõ ràng giữa kiểu dữ liệu, Date và các hàm. Kết nối đó có thể được thiết lập bằng cách khai báo các hàm vào bên trong struct Date:

*struct Date {*

*int d,m, y;*

*void init\_date( int dd , int mm , int yy ); //khởi tạo*

*void add\_year(int n); //thêm n năm*

*void add\_month(int n); //thêm n tháng*

*void add\_day(int n); //thêm n ngày*

*};*

Các hàm trong phần định nghĩa Class (struct cũng là một loại class ) được gọi là các hàm thành viên và chỉ được gọi các hàm bằng cách sử dụng các cú pháp tiêu chuẩn.

Ví dụ:

*Date my\_birthday;*

*void f()*

*{*

*Date today;*

*today.init(16,10,1996);*

*my\_bir thday.init(30,12,1950);*

*Date tomorrow = today;*

*tomorrow.add\_day(1);*

*}*

Bởi vì các struct khác nhau có thể có các hàm thành viên có cùng tên, vì vậy chúng ta cần xác định tên cấu trúc khi xác định (định nghĩa) một hàm thành viên:

*void Date::init(int dd, int mm, int yy) // hàm init nằm trong cấu trúc Date*

*{*

*d=dd;*

*m = mm;*

*y = yy;*

*}*

Trong một hàm thành viên, tên thành viên có thể được sử dụng mà không cần tham chiếu rõ ràng đến một đối tượng.

Ví dụ: khi *Date ::init ()* được gọi cho today, m=mm sẽ gán cho today.m. Hay khi *Date::innit ()* được gọi cho *my\_birthday*, m=mm gán cho *my\_birthday.m* .

#### 16.2.2 Sao chép mặc định (Default Copying)

Theo mặc định, các đối tượng có thể được sao chép. Đặc biệt, một đối tượng lớp có thể được khởi tạo bằng bản sao của một đối tượng của lớp đó.

Ví dụ:

*Date d1 = my\_birthday; // khởi tạo bằng sao chép*

*Date d2 {my\_birthday}; // khởi tạo bằng sao chép*

*Tương tự, các đối tượng lớp có thể được sao chép bằng phép gán.*

*Ví dụ:*

*void f(Date& d)*

*{*

*d = my\_birthday;// khởi tạo d bằng cách gán d= my\_birthday*

*}*

Nếu đó không phải là một sự lựa chọn phù hợp để khởi tạo bằng bản sao cho lớp X, người dùng có thể xác định toán tử gán thích hợp hơn (§3.3, §17.5).

#### 16.2.3 Kiểm soát truy cập (Access Control)

Khai báo *Date* ở phần trên cung cấp một tập hợp các hàm để thao tác với *Date*. Tuy nhiên, nó không chỉ rõ những hàm đó phải phụ thuộc trực tiếp vào thuộc tính của *Date* và những người duy nhất truy cập trực tiếp các đối tượng của lớp *Date*. Hạn chế này có thể được khắc phục bằng cách sử dụng một lớp (class) thay vì một cấu trúc (struct) :

*class Date {*

*int d, m, y;*

*public:*

*void init(int dd, int mm, int yy); // khởi tạo*

*void add\_year(int n); // thêm n năm*

*void add\_month(int n); // thêm n tháng*

*void add\_day(int n); // thêm n ngày*

*};*

Nhãn công khai (public) chia lớp (class) thành hai phần. Phần đầu tiên là riêng tư (private), chỉ các hàm thành viên trong class được sử dụng. Phần hai, công khai (public), chứa các phương thức ngoài class có thể sử dụng. Một cấu trúc (struct) đơn giản là một class có các thành viên là công khai (public); các hàm thành viên có thể loại bỏ và sử dụng như trước đây.

Ví dụ:

*void Date::add\_year(int n)*

*{*

*y+=n;*

*}*

Tuy nhiên, các hàm không phải là hàm thành viên bị cấm sử dụng các thành viên trong private. Ví dụ:

*void timewarp(Date& d)*

*{*

*d.y −= 200; // lỗi vì y nằm trong private*

*}*

Hàm *init ()* bây giờ rất cần thiết vì đặt giữ liệu ở chế độ riêng tư buộc ta phải cung cấp một cách khởi tạo thành viên. Ví dụ:

*Date dx;*

*dx.m = 3; // lỗi vì m nằm trong private*

*dx.init(25,3,2011); // không lỗi*

Việc hạn chế quyền truy cập bằng cách phân ra private và public mang lại lợi cho việc quản lý chương trình: hạn chế người dụng truy cập vào các thuộc tính quan trọng, bảo mật thông tin ,…

#### 16.2.4 Class và Struct

Class X {…} được gọi là một định nghĩa lớp. Một định nghĩa lớp có thể được sao chép trong các nguồn khác nhau bằng cách sử dụng #include mà không vi phạm quy tắc khái niệm của một định nghĩa.

Theo định nghĩa, một cấu trúc là một lớp trong đó các thành viên là công khai; đó là,*struct S{/\*…\*/}*hoặc có thể viết là *class S{public/\*…\*/}.*Hai cách định nghĩa này của S có thể thay thế cho nhau mặc dù thông thường sẽ tốt hơn nếu bạn chỉ theo một cách.

Theo mặc định, các thành viên của class là private:

*class Date1 {*

*int d, m, y; // riêng tư theo mặc định*

*public:*

*Date1(int dd, int mm, int yy);*

*void add\_year(int n); // thêm n năm*

*};*

Tuy nhiên, chúng ta có thể sử dụng đặc điểm truy cập riêng tư: để nói rằng các thành viên sau đây là riêng tư, giống như public: nói rằng các thành viên sau đây là công khai:

*struct Date2 {*

*private:*

*int d, m, y;*

*public:*

*Date2(int dd, int mm, int yy);*

*void add\_year(int n); // thêm n năm*

*};*

Ngoại trừ tên khác, *Date1* và *Date2* là tương đương.

Không bắt buộc phải khai báo dữ liệu đầu tiên trong một lớp. Trên thực tế, việc đặt dữ liệu thường có ý nghĩa thành viên cuối cùng nhấn mạnh các hàm cung cấp giao diện người dùng công khai. Ví dụ:

*class Date3 {*

*public:*

*Date3(int dd, int mm, int yy);*

*void add\_year(int n); // thêm n năm*

*private:*

*int d, m, y;*

*};*

Trong mã thực, nơi cả giao diện công khai và cáchi tiết thường triển khai nhiều hơn mở rộng hơn trong các ví dụ hướng dẫn.

Các đặc tả truy cập có thể được sửu dụng nhiều lần trong một khai báo lớp duy nhất.

Ví dụ:

*class Date4 {*

*public:*

*Date4(int dd, int mm, int yy);*

*private:*

*int d, m, y;*

*public:*

*void add\_year(int n); // them n năm*

*};*

Tuy nhiên, việc có nhiều hơn một phần công khai, như trong *Date4*, có xu hướng lộn xộn và ảnh hưởng đến bố cục đối tượng.

#### 16.2.5 Constructor

Việc sử dụng các hàm như *init ()* để cung cấp khởi tạo cho các đối tượng lớp là không phù hợp và dễ xảy ra lỗi. Bởi vì không có chỗ nào nói rằng một đối tượng là phải khởi tạo, một lập trình viên có thể quên làm như vậy. Một cách tiếp cận tốt hơn là cho phép lập trình viên khai báo một hàm với mục đích rõ ràng là khởi tạo các đối tượng. Bởi vì như vậy một hàm xây dựng các giá trị của một kiểu nhất định, được gọi là hàm khởi tạo. Một phương thức khởi tạo là một phương thức có cùng tên với chính lớp đó.

*class Date {*

*int d, m, y;*

*public:*

*Date(int dd, int mm, int yy); // constructor ( hàm khởi tạo đầy đủ)*

*// ...*

*};*

Khi một class có phương thức khởi tạo, tất cả các đối tượng của lớp đo sẽ được khởi tạo bằng cách gọi phương thức khởi tạo. Nếu như hàm khởi tạo yêu cầu các đối số, các đối số này phải được cung cấp:

*Date today = Date(23,6,1983);// đúng*

*Date xmas(25,12,1990); // dạng viết tắt*

*Date my\_bir thday; // sai vì thiếu các đối số*

*Date release1\_0(10,12); // sai vì thiếu một đối số*

Vì một phương thức khởi tạo để khởi tạo cho một lớp, chúng ta có thể sử dụng ký hiệu {} để khởi tao.

*Date today = Date {23,6,1983};*

*Date xmas {25,12,1990}; // dạng viết tắt*

*Date release1\_0 {10,12}; // sai vì thiếu một đối số*

Chúng ta nên sử dụng ký kiệu {} trên ký hiệu () để khởi tạo vì nó rõ ràng về những gì đang được khởi tạo, tránh một số lỗi tiềm ẩn và có thể được sử dụng nhất quán. Có những trường hợp phải sử dụng ký hiệu (), nhưng chúng rất hiếm.

Có nhiều cách khác nhau để khởi tạo các đối tượng:

*class Date {*

*int d, m, y;*

*public:*

*// ...*

*Date(int, int, int); // day, month, year ( hàm khởi tạo đầy đủ)*

*Date(int, int); // day, month, today’s year (hàm khỏi tạo không đầy đủ)*

*Date(int); // day, today’s month and year*

*Date(); // hàm khởi tạo mặc định*

*Date(const char∗); // hàm khởi tạo sao chép*

*};*

Các hàm khởi tạo vẫn tuân theo quy tắc nạp chồng như các hàm thông thường. Miễn các hàm khởi tạo khác nhau về kiểu đối số, trình biên dịch có thể chọn đúng để sử dụng:

*Date today {4}; // 4, today.m, today.y*

*Date july4 {"July 4, 1983"};*

*Date guy {5,11}; // 5, November, today.y*

*Date now; // mặc định khởi tạo như today*

*Date start {}; // mặc định khởi tạo như today*

Sự gia tăng các hàm khởi tạo trong ví dụ Date là điển hình. Khi thiết kế một lớp, một lập trình viên luôn bị cám sỗ để thêm các tính năng chỉ vì ai đó có thể muốn chúng. Cần suy nghĩ nhiều hơn để quyết định cẩn thận những tính năng nào thực sự cần thiết và chỉ bao gồm những tính năng đó. Tuy nhiên, suy nghĩ bổ sung đó thường dẫn đến các chương trình nhỏ hơn và dễ hiểu hơn. Một cách để giảm các hàm liên quan là sử dụng các đối số mặc định. Đối với Date, mỗi đối dố có thể được cung cấp một giá trị mặc định được hiểu là “ chọn giá trị mặc định: today.”

*class Date {*

*int d, m, y;*

*public:*

*Date(int dd =0, int mm =0, int yy =0);*

*// ...*

*};*

*Date::Date(int dd, int mm, int yy)*

*{*

*d=dd?dd:today.d;*

*m=mm?mm:today.m;*

*y=yy?yy:today.y;*

*// kiểm tra xem Date có hợp lệ không*

*}*

Khi một đối số được sử dụng để biểu thị “chọn giá trị mặc định”, giá trị được chọn phải nằm ngoài tập hợp các giá trị có thể có cho đối số.

Ngoài ra, chúng ta có thể sử dụng các giá trị mặc định trực tiếp làm đối số mặc định:

*class Date {*

*int d, m, y;*

*public:*

*Date(int dd =today.d, int mm =today.m, int yy =today.y);*

*// ...*

*};*

*Date::Date(int dd, int mm, int yy)*

*{*

*// kiểm tra xem Date có hợp lệ không*

Lưu ý rằng: bằng cách đảm báo việc khởi tạo các đối tượng một cách thích hợp, các hàm tạo sẽ đơn giản hóa rất nhiều việc thực hiện các hàm thành viên. Với các hàm khởi tạo, các hàm thành viên không còn phải đối phó với khả năng dữ liệu chưa được khởi tạo.

#### 16.2.6 Hàm khởi tạo rõ ràng (Explicit Contructors)

Theo mặc định, một hàm khởi tạo được gọi bởi một đối số hoạt động như một chuyển đổi ngầm từ kiểu đối số sang kiểu của nó:

*complex<double> d {1}; // d=={1,0}*

Những chuyển đổi ngầm như vậy có thể cực kỳ hữu ích. Ví dụ: trong số phức nếu ta bỏ đi phần ảo, ta sẽ nhận được một số thực. Tuy nhiên, trong nhiều trường hợp, những chuyển đổi như vậy có thể là nguyên nhân gây ra các lỗi. Xem xét Date:

*void my\_fct(Date d);*

*void f()*

*{*

*Date d {15}; // hợp lý: x trở thành {15,today.m,today.y}*

*// ...*

*my\_fct(15); // hkhó hiểu*

*d = 15; // khó hiểu*

*// ...*

*}*

Không có mối liên hệ logic nào giữa số 15 và Date độc lập với sự phức tạp của đoạn mã của chúng ta.

May mắn thay, chúng ta có thể có một hàm khởi tạo với từ khóa tường minh chỉ có thể được sử dụng để khởi tạo và chuyển đổi rõ ràng:

Ví dụ:

*class Date {*

*int d, m, y;*

*public:*

*explicit Date(int dd =0, int mm =0, int yy =0);*

*// ...*

*};*

*Date d1 {15}; // OK*

*Date d2 = Date{15}; // OK*

*Date d3 = {15}; // error //*

*Date d4 = 15; // error khởi tạo lỗi*

*void f()*

*{*

*my\_fct(15); // error : truyền đối số không thực hiện chuyển đổi*

*my\_fct({15}); // error : //*

*my\_fct(Date{15}); // OK*

*// ...*

*}*

Một khởi tạo với dấu “=” được gọi là khởi tạo sao chép. Về nguyên tắc, một bản sao của quá trình khởi tạo được đặt vào đối tượng đã khởi tạo. Bỏ dấu “=” làm cho qua trình khởi tạo trở nên rõ ràng. Khởi tạo rõ ràng được gọi là khởi tạo trực tiếp.

Một hàm khởi tạo được khai báo rõ ràng và được xác định bên ngoài lớp, thì không thể lặp lại hàm tạo đó:

*class Date {*

*int d, m, y;*

*public:*

*explicit Date(int dd);*

*// ...*

*};*

*Date::Date(int dd) { /\* ... \*/ } // OK*

*explicit Date::Date(int dd) { /\* ... \*/ } // error*

Hầu hết các ví dụ trong đó tường minh là quan trọng liên quan đến một đối số phương thức khởi tạo duy nhất. Tuy nhiên, tường minh cũng có thể hhữu ích cho các hàm khởi tạo không có hoặc có nhiều hơn một đối số.

*struct X {*

*explicit X();*

*explicit X(int,int);*

*};*

*X x1 = {}; // error : implicit (không tường minh)*

*X x2 = {1,2}; // error : implicit*

*X x3 {}; // OK: explicit (tường minh)*

*X x4 {1,2}; // OK: explicit*

*int f(X);*

*int i1 = f({}); // error : implicit*

*int i2 = f({1,2}); // error : implicit*

*int i3 = f(X{}); // OK: explicit*

*int i4 = f(X{1,2}); // OK: explicit*

#### 16.2.7 Khởi tạo trong lớp (In - Class Initializers)

Khi chúng ta sử dụng một hàm khởi tạo, việc khởi tạo thành viên có thể lặp lại. Ví du:

*class Date {*

*int d, m, y;*

*public:*

*Date(int, int, int); // day, month, year (ngày, tháng , năm)*

*Date(int, int); // day, month, today’s year ( ngày , tháng)*

*Date(int); // day, today’s month and year (ngày)*

*Date(); // default Date: today (tự định nghĩa)*

*Date(const char∗); // date in string representation ( chuỗi cho trước)*

*// ...*

*};*

Chúng ta có thể thêm trình khởi tạo và các thành viên dữ liệu:

*class Date {*

*int d {today.d};*

*int m {today.m};*

*int y {today.y};*

*public:*

*Date(int, int, int); // day, month, year*

*Date(int, int); // day, month, today’s year*

*Date(int); // day, today’s month and year*

*Date(); // default Date: today*

*Date(const char∗); // date in string representation*

Bây giờ, mỗi hàm khởi tạo có d, m, và y được khởi tạo trừ khi chính nó thực hiện nó. Ví dụ:

*Date::Date(int dd)*

*:d{dd}*

*{*

*// kiểm tra Date có hợp lệ không*

*}*

Tương đương với:

*Date::Date(int dd)*

*:d{dd}, m{today.m}, y{today.y}*

*{*

*// kiểm tra Date có hợp lệ không*

*}*

#### 16.2.8 Định nghĩa hàm trong lớp (In - Class Function Deﬁnitions)

Một hàm thành viên được định nghĩa trong lớp thay vì chỉ khai báo nó – được gọi là hàm thành viên nội tuyến. Có nghĩa là, định nghĩa các hàm thành viên trong lớp dành cho các hàm nhỏ, hiếm khi sửa đổi, được sử dụng thường xuyên.

Một thành viên có thể tham chiếu đến một thành viên khác trong lớp mình một cách độc lập với nới thành viên đó được định nghĩa:

*class Date {*

*public:*

*void add\_month(int n) { m+=n; } // gia tăng Date của m*

*// ...*

*private:*

*int d, m, y;*

*};*

Có nghĩa là, các hàm và dữ liệu thành viên là độc lập với thứ tự. Viết tương đương:

*class Date {*

*public:*

*void add\_month(int n) { m+=n; } // increment the Date’s m (tăng thêm Date cho m)*

*// ...*

*private:*

*int d, m, y;*

*};*

*inline void Date::add\_month(int n) // add n months ( thêm n tháng)*

*{*

*m+=n; // increment the Date’s m*

*}*

Kiểu thứ hai này thường được sử dụng để giữ cho các định nghĩa lớp đơn giản và dễ đọc. Nó cũng cung cấp sự phân tách văn bản về giao diện và triển khai của một lớp.

#### 16.2.9 Khả năng thay đổi (Mutability)

Chúng ta có thể xác định một đối tượng được đặt tên là một hằng số hoặc một biến, Nói cách khác, tên có thể tham chiếu đến một đối tượng chứa một giá trị không thể thay đổi hoặc có thể thay đổi.

Tính bất biến là đặc tính hữu ích nhất trong chương trình đa luồng. Để hữu ích ngoài việc xác định các hằng số đơn giản của kiểu phương thức khởi tạo, chúng ta phải có khả năng xác định các hàm hoạt động trên các đối tượng const của kiểu do người dùng định nghĩa. Đối với các hàm tự do có nghĩa là các hàm nhận const T& đối số. Đối với các lớp, điều đó có mghĩa là chúng ta phải có khả năng loại bỏ các hàm thành viên hoạt động trên đối tượng const.

##### 16.2.9.1 Các hàm thành viên không đổi (Constant Member Functions)

Date được định nghĩa cho đến nay đã cung cấp các hàm thành viên để cung cấp một giá trị cho Date. Rất tiếc, ta không cung cấp cách kiểm tra các giá trị của Date. Khắc phục bằng cách thêm thêm các hàm đọc day, month và year:

*class Date {*

*int d, m, y;*

*public:*

*int day() const { return d; } // trả về day*

*int month() const { return m; }// trả về month*

*int year() const;*

*void add\_year(int n); // add n years*

*// ...*

*};*

Const sau danh sách đối số trong khai báo hàm chỉ ra rằng các hàm này không sửa đổi trạng thái của Date

Đương nhiên, trình biên dịch sẽ bắt gặp những lỗi vô tình vi phạm. Ví dụ:

*int Date::year() const*

*{*

*return ++y; // error : thay đổi giá trị thành viên trong hàm const*

*}*

Khi một hàm thành viên const được xác định bên ngoài lớp của nó, hậu tố const là bắt buộc:

*int Date::year() // error : const missing in member function type*

*{*

*return y;*

*}*

Nói cách khác, const là một phần của kiểu *Date:: day (),* *Date:: month (),*và *Date:: year ().* Một hàm thành viên const có thể được gọi cho cả đối tượng const và không phải là const, trong khi một thành viên không phải là const chỉ có thể được gọi cho các đối tượng không phải là const. Ví dụ:

*void f(Date& d, const Date& cd)*

*{*

*int i = d.year(); // OK*

*d.add\_year(1); // OK*

*int j = cd.year(); // OK*

*cd.add\_year(1); // error : không thể thay đổi giá trị của một const Date*

*}*

##### 16.2.9.2 Hằng số vật lý và logic (Physical and Logical Constness)

Đôi khi, một hàm thành viên là const về mặt logic, nhưng nó vẫn cần thay đổi giá trị của một thành viên. Có nghĩa là, đối với người dùng, hàm dường như không thay đổi trạng thái của đối tượng của nó, nhưng một số chi tiết mà người dùng không thể quan sát trực tiếp được cập nhật. Điều này thường được gọi là hằng số logic. Ví dụ, lớp Date có thể có một hàm trả về biểu diễn chuỗi. Việc xây dựng biểu diễn này có thể là một hoạt động tương đối tốn kém. Do đó, bạn nên giữ một bản sao để các yêu cầu lặp đi lặp lại sẽ chỉ trả lại bản sao đó, trừ khi giá trị của Date đã được thay đổi. Việc lưu vào bộ nhớ đệm các giá trị như vậy phổ biến hơn đối với các cấu trúc dữ liệu phức tạp hơn, nhưng hãy xem cách đạt được giá trị đó cho một Date:

*class Date {*

*public:*

*// ...*

*string string\_rep() const; // biểu diễn chuỗi*

*private:*

*bool cache\_valid;*

*string cache;*

*void compute\_cache\_value(); // lấy bộ nhớ cache*

*// ...*

*};*

Theo quan điểm của người dùng, *string\_rep* không thay đổi trạng thái Date của nó, vì vậy rõ ràng nó phải là một hàm thành viên const. Mặt khác, các thành viên cache và *cache\_valid* thỉnh thoảng phải thay đổi để thiết kế có ý nghĩa. Những vấn đề như vậy có thể giải quyết bằng cách sử dụng ép kiểu, ví dụ, *const\_cast* (§11.5.2). Tuy nhiên, cũng có những giải pháp thanh lịch hợp lý không liên quan đến việc gây rối với các quy tắc loại.

##### 16.2.9.3 Có thể thay đổi (Mutable)

Chúng ta có thể xác định một thành viên của một lớp là có thể thay đổi được, nghĩa là nó có thể được sửa đổi ngay cả trong một đối tượng const:

*class Date {*

*public:*

*// ...*

*string string\_rep() const; // string representation*

*private:*

*mutable bool cache\_valid;*

*mutable string cache;*

*void compute\_cache\_value() const; // ﬁll (mutable) cache ( lấy bộ nhớ cache và có thể thay đổi)*

*// ...*

*};*

Bây giờ chúng ta có thể định nghĩa string\_rep () theo cách rõ ràng:

*string Date::string\_rep() const*

*{*

*if (!cache\_valid) {*

*compute\_cache\_value();*

*cache\_valid = true;*

*}*

*return cache;*

*}*

Bây giờ chúng ta có thể sử dụng *string\_rep ()* cho cả đối tượng const và không phải const. Ví dụ:

*void f(Date d, const Date cd)*

*{*

*string s1 = d.string\_rep();*

*string s2 = cd.string\_rep(); // OK!*

*// ...*

*}*

##### 16.2.9.4 Thay đổi thông qua hướng dẫn (Mutability through Indirection)

Khai báo một thành viên có thể thay đổi là thích hợp nhất khi chỉ một phần nhỏ của biểu diễn của một đối tượng nhỏ được phép thay đổi. Các trường hợp phức tạp hơn thường được xử lý tốt hơn bằng cách đặt dữ liệu thay đổi vào một đối tượng riêng biệt và truy cập nó một cách gián tiếp. Nếu kỹ thuật đó được sử dụng, chuỗi có ví dụ bộ nhớ cache sẽ trở thành:

*struct cache {*

*bool valid;*

*string rep;*

*};*

*class Date {*

*public:*

*// ...*

*string string\_rep() const; // string representation*

*private:*

*cache∗ c; // khởi tạo trong hàm khởi tạo*

*void compute\_cache\_value() const; // ﬁll what cache refers to (lấp đầy những gì bộ nhớ cache đề cập đến)*

*// ...*

*}*

*string Date::string\_rep() const*

*{*

*if (!c−>valid) {*

*compute\_cache\_value();*

*c−>valid = true;*

*}*

*return c−>rep;*

*}*

Các kỹ thuật lập trình hỗ trợ một bộ nhớ cache tổng quát thành các hình thức đánh giá lười biếng khác nhau.

Lưu ý rằng const không áp dụng (tạm thời) cho các đối tượng được truy cập thông qua con trỏ hoặc tham chiếu. Có nghĩa là, một con trỏ thành viên không có bất kỳ ngữ nghĩa đặc biệt nào để phân biệt nó với các con trỏ khác.

#### 16.2.10 Tự tham khảo (Self - Reference)

Các hàm cập nhật trạng thái *add\_year (), add\_month ()* và *add\_day ()* không được trả về giá trị. Đối với một tập hợp các hàm cập nhật liên quan như vậy, thường hữu ích khi trả về một tham chiếu đến đối tượng được cập nhật để các hoạt động có thể được xâu chuỗi. Ví dụ, chúng ta muốn viết:

*void f(Date& d)*

*{*

*// ...*

*d.add\_day(1).add\_month(1).add\_year(1);*

*// ...*

*}*

Để thêm một ngày, một tháng và một năm vào d, mỗi hàm phải được khai báo để trả về một tham chiếu đến Date:

*class Date {*

*// ...*

*Date& add\_year(int n); // add n years (thêm n năm)*

*Date& add\_month(int n); // add n months (thêm n tháng)*

*Date& add\_day(int n); // add n days (thêm n ngày)*

*};*

Mỗi hàm thành viên (không tĩnh) biết đối tượng nào mà nó được gọi và có thể tham chiếu đến nó một cách rõ ràng. Ví dụ:

*Date& Date::add\_year(int n)*

*{*

*if (d==29 && m==2 && !leapyear(y+n)) { // beware of February 29 (cẩn thận 29/2)*

*d=1;*

*m=3;*

*}*

*y+=n;*

*return ∗this;*

*}*

Biểu thức \*this đề cập đến đối tượng mà một hàm thành viên được gọi.

Trong một hàm thành viên không tĩnh (static), từ khóa this là một con trỏ trỏ đến đối tượng mà hàm được gọi. Trong một hàm thành viên không phải hằng số (const) của lớp X, kiểu của nó là X\*. Tuy nhiên, đây được coi là một giá trị, vì vậy không thể lấy địa chỉ của nó hoặc gán cho nó.

Hầu hết việc sử dụng this là ngầm định. Đặc biệt, mọi tham chiếu đến một thành viên không tĩnh (non-static) từ bên trong một lớp dựa vào việc sử dụng this để lấy thành viên của đối tượng thích hợp. Ví dụ, hàm add\_year có thể tương đương, nhưng thật tệ chúng được xác định như sau:

*Date& Date::add\_year(int n)*

*{*

*if (this−>d==29 && this−>m==2 && !leapyear(this−>y+n)) {*

*this−>d = 1;*

*this−>m = 3;*

*}*

*this−>y += n;*

*return ∗this;*

*}*

Một cách sử dụng this rõ ràng là thao tác trong danh sách liên kết. Ví dụ:

*struct Link {*

*Link∗ pre;*

*Link∗ suc;*

*int data;*

*Link∗ insert(int x) // insert x before this (chèn x trước this)*

*{*

*return pre = new Link{pre ,this,x};*

*}*

*void remove() // remove and destroy this (loại bỏ và phá hủy this)*

*{*

*if (pre) pre−>suc = suc;*

*if (suc) suc−>pre = pre;*

*delete this;*

*}*

*// ...*

*};*

Cần sử dụng this rõ ràng để truy cập vào các thành viên của các lớp cơ sở từ một lớp dẫn xuất là một khuôn mẫu.

#### 16.2.11 Quyền truy cập thành viên (Member Access)

Một thành viên của lớp X có thể được truy cập bằng cách áp dụng “.” (dot) toán tử cho một đối tượng của lớp X hoặc bằng cách áp dụng toán tử “->” (mũi tên) cho một con trỏ đến một đối tượng của lớp X. Ví dụ:

*struct X {*

*void f();*

*int m;*

*};*

*void user(X x, X∗ px)*

*{*

*m=1; // error : there is no m in scope (m không nằm trong phạm vi)*

*x.m = 1; // OK*

*x−>m = 1; // error : x is not a pointer (x không phải là một con trỏ)*

*px−>m = 1; // OK*

*px.m = 1; // error : px is a pointer (px là một con trỏ)*

*}*

Rõ ràng, có một chút dư thừa ở đây: trình biên dịch biết liệu một tên liên quan đến X hay X \*, vì vậy một toán tử đơn lẻ sẽ phù hợp. Tuy nhiên, một lập trình viên có thể bị nhầm lẫn, vì vậy từ những ngày đầu tiên của C, quy tắc đã được sử dụng các toán tử riêng biệt.

Từ bên trong một lớp không cần toán tử. Ví dụ:

*void X::f()*

*{*

*m=1; // OK: ‘‘this->m = 1;’’*

*}*

Nghĩa là, một tên thành viên không đủ tiêu chuẩn hoạt động như thể nó đã được xác định trước bởi this−>. Lưu ý rằng một hàm thành viên có thể tham chiếu đến tên của một thành viên trước khi nó được khai báo:

*struct X {*

*int f() { return m; } // ﬁne: return this X’s m (trả về this X’s m)*

*int m;*

*};*

Nếu chúng ta muốn tham chiếu đến một thành viên nói chung, thay vì một thành viên của một đối tượng cụ thể, chúng ta ta thể sử dụng “ :: ”. Ví dụ:

*struct S {*

*int m;*

*int f();*

*static int sm;*

*};*

*int X::f() { return m; } // X’s f*

*int X::sm {7}; // X’s static member sm (thành viên tĩnh sm)*

*int (S::∗) pmf() {&S::f}; // X’s member f*

#### 16.2.12 Thành viên tĩnh ( [static] Members )

Thành viên tĩnh là những thành viên sử dụng toàn cụ trong lớp và không thể sử dụng bên ngoài lớp .

*class Date {*

*int d, m, y;*

*static Date default\_date;*

*public:*

*Date(int dd =0, int mm =0, int yy =0);*

*// ...*

*static void set\_default(int dd, int mm, int yy); // set default\_date to Date(dd,mm,yy)*

*};*

Bây giờ chúng ta có thể định nghĩa hàm khởi tạo Date để sử dụng default\_date như sau:

*Date::Date(int dd, int mm, int yy)*

*{*

*d=dd?dd:default\_date .d;*

*m = mm ? mm : default\_date .m;*

*y = yy ? yy : default\_date .y;*

*// ... check that the Date is valid ...*

*}*

Sử dụng *set\_default (),* chúng ta có thể thay đổi ngày mặc định khi thích hợp. Một thành viên tĩnh có thể được gọi như bất kỳ thành viên nào khác. Ngoài ra, một thành viên tĩnh có thể được tham chiếu mà không cần đề cập đến một đối tượng. Thay vào đó, tên của nó đủ điều kiện theo tên của lớp của nó. Ví dụ:

*void f()*

*{*

*Date::set\_default(4,5,1945); // call Date’s static member set\_default() (gọi thành viên tĩnh set\_default () của Date’s*

*}*

Nếu được sử dụng, một thành viên tĩnh – một hàm hoặc dữ liệu thành viên- phải được xác định ở đâu đó. Từ khóa static không được lặp lại trong khái niệm của một thành viên static. Ví dụ:

*Date Date::default\_date {16,12,1770}; // deﬁnition of Date::default\_date (định nghĩa của Date :: default\_date() )*

*void Date::set\_default(int d, int m, int y) // deﬁnition of Date::set\_default định nghĩa của Date :: set\_default() )*

*{*

*default\_date = {d,m,y}; // assign new value to default\_date (gán giá trị mới cho default\_date)*

*}*

Bây giờ, giá trị mặc định là ngày sinh của Beethoven – cho đến khi ai đó quyết định khác.

Lưu ý rằng *Date {}* đóng vai trò là ký hiệu cho giá trị của *Date :: default\_date*. Ví dụ:

*Date copy\_of\_default\_date = Date{};*

*void f(Date);*

*void g()*

*{*

*f(Date{});*

*}*

Do đó, chúng ta không cần một chức năng riêng để đọc ngày mặc định. Vì vậy, dấu {} thuần túy là phù hợp. Ví dụ:

*void f1(Date);*

*void f2(Date);*

*void f2(int);*

*void g()*

*{*

*f1({}); // OK: equivalent to f1(Date{}) (tương đương với f1(Date{})*

*f2({}): // error : ambiguous: f2(int) or f2(Date)? ( f2 không rõ ràng)*

*f2(Date{}); // OK }*

Trong mã đa luồng, các thành viên dữ liệu tĩnh yêu cầu một số loại kỷ luật khóa hoặc truy cập để tránh các điều kiện chủng tộc (§5.3.4, §41.2.4). Vì đa luồng hiện nay rất phổ biến, thật không may là việc sử dụng các thành viên dữ liệu tĩnh lại khá phổ biến trong mã cũ. Mã cũ hơn có xu hướng sử dụng các thành viên tĩnh theo cách ngụ ý các điều kiện chủng tộc.

#### 16.2.13 Các loại thành viên (Member Types)

Các kiểu và kiểu bí danh có thể là thành viên của một lớp. Ví dụ:

*template<typename T>*

*class Tree {*

*using value\_type = T; // member alias (thành viên bí danh)*

*enum Policy { rb, splay, treeps }; // member enum(thành viên danh sách)*

*class Node { // member class (thành viên lớp)*

*Node∗ right;*

*Node∗ left;*

*value\_type value;*

*public:*

*void f(Tree∗);*

*};*

*Node∗ top;*

*public:*

*void g(const T&);*

*// ...*

*};*

Một lớp thành viên (thường được gọi là lớp lồng nhau) có thể tham chiếu đến các kiểu và các thành viên tĩnh của lớp bao quanh nó. Nó chỉ có thể tham chiếu đến các thành viên không tĩnh khi nó được cung cấp một đối tượng của lớp bao quanh để tham chiếu đến.

Một lớp lồng nhau có quyền truy cập vào các thành viên của lớp bao quanh nó, thậm chí đến các thành viên riêng (giống như một hàm thành viên ), nhưng không có khái niệm về đối tượng hiện tại của lớp bao quanh. Ví dụ:

*template<typename T>*

*void Tree::Node::f(Tree∗ p)*

*{*

*top = right; // error : no object of type Tree speciﬁed (không có đối tượng loại đặc tả của Tree*

*p−>top = right; // OK*

*value\_type v = left−>value; // OK: value\_type is not associated with an object (value\_type không được liên kết với một đối tượng)*

*}*

Một lớp không có bất kỳ quyền truy cập đặc biệt nào đối với các thành viên của lớp lồng nhau của nó. Ví dụ:

*template<typename T>*

*void Tree::g(Tree::Node∗ p)*

*{*

*value\_type val = right−>value; // error : no object of type Tree::Node (không có đối tượng thuộc loại Tree :: Node)*

*value\_type v = p−>right−>value; // error : Node::right is private (right là riêng tư)*

*p−>f(this); // OK*

*}*

Các lớp thành viên là một sự tiện lợi về mặt ký hiệu hơn là một tính năng có tầm quan trọng cơ bản. Mặt khác, bí danh thành viên rất quan trọng vì là cơ sở của các kỹ thuật lập trình chung dựa trên các kiểu liên kết (§28.2.4, §33.1.3).

### 16.3 Lớp cụ thể (Concrete Classes)

Một lớp được gọi là cụ thể (hoặc một lớp cụ thể) nếu biểu diễn của nó là một phần của định nghĩa. Điều này phân biệt nó với các lớp trừu tượng cung cấp giao diện cho nhiều cách triển khai khác nhau. Có sẵn đại diện cho phép chúng ta:

* Để đặt các đối tượng trên ngăn xếp, trong bộ nhớ được cấp phát tĩnh và trong các đối tượng khác
* Để sao chép và di chuyển các đối tượng.
* Để tham chiếu trực tiếp đến các đối tượng được đặt tên (trái ngược với việc truy cập thông qua con trỏ và tham chiếu)

*namespace Chrono {*

*enum class Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };*

*class Date {*

*public: // public interface:*

*class Bad\_date { }; // exception class (lớp mgoại lệ)*

*explicit Date(int dd ={}, Month mm ={}, int yy ={}); // {} means ‘‘pick a default’’ (có mghĩa là chọn mặc định)*

*// nonmodifying functions for examining the Date: (các chức năng không sửa đổi để kiểm tra Date)*

*int day() const;*

*Month month() const;*

*int year() const;*

*string string\_rep() const; // string representation*

*void char\_rep(char s[], in max) const; // C-style string representation*

*// (modifying) functions for changing the Date:*

*Date& add\_year(int n); // add n years*

*Date& add\_month(int n); // add n months*

*Date& add\_day(int n); // add n days*

*private:*

*bool is\_valid(); // check if this Date represents a date*

*int d, m, y; // representation*

*};*

*bool is\_date(int d, Month m, int y); // true for valid date*

*bool is\_leapyear(int y); // true if y is a leap year*

*bool operator==(const Date& a, const Date& b);*

*bool operator!=(const Date& a, const Date& b);*

*const Date& default\_date(); // the default date*

*ostream& operator<<(ostream& os, const Date& d); // printdtoos*

*istream& operator>>(istream& is, Date& d); // read Date from is into d*

*}// Chrono*

#### 16.3.1 Hàm thành viên (Member Functions)

Đương nhiên, việc định nghĩa một hàm thành viên phải được thực hiện ở đâu đó. Ví dụ:

*Date::Date(int dd, Month mm, int yy)*

*:d{dd}, m{mm}, y{yy}*

*{*

*if (y == 0) y = default\_date().year();*

*if (m == Month{}) m = default\_date().month();*

*if (d == 0) d = default\_date().day();*

*if (!is\_valid()) throw Bad\_date();*

*}*

Hàm tạo kiểm tra xem dữ liệu được cung cấp có biểu thị Date hợp lệ hay không. Nếu không, giả sử, đối với {30, Month :: feb, 1994}, nó ném ra một ngoại lệ, cho biết rằng đã xảy ra sự cố. Nếu dữ liệu được cung cấp có thể chấp nhận được, thì quá trình khởi tạo rõ ràng đã được thực hiện. Khởi tạo là một hoạt động tương đối phức tạp vì nó liên quan đến việc xác nhận dữ liệu. Điều này là khá điển hình. Mặt khác, khi Ngày đã được tạo, nó có thể được sử dụng và sao chép mà không cần kiểm tra thêm. Nói cách khác, hàm tạo thiết lập giá trị bất biến cho lớp (trong trường hợp này, nó biểu thị một ngày hợp lệ). Các chức năng thành viên khác có thể dựa vào sự bất biến đó và phải duy trì nó. Kỹ thuật thiết kế này có thể đơn giản mã hóa rất nhiều.

Tôi đã cân nhắc đặt hàm xác thực is\_valid () ở chế độ công khai. Tuy nhiên, tôi thấy mã người dùng tạo ra phức tạp hơn và kém mạnh mẽ hơn so với mã dựa vào việc bắt ngoại lệ:

*void ﬁll(vector<Date>& aa)*

*{*

*while (cin) {*

*Date d;*

*try {*

*cin >> d;*

*}*

*catch (Date::Bad\_date) {*

*// ... my error handling ...*

*continue;*

*}*

*aa.push\_back(d);*

*}*

*}*

Tuy nhiên, việc kiểm tra xem tập hợp giá trị {d, m, y} có phải là ngày hợp lệ không không phải là một phép tính phụ thuộc vào biểu diễn của Date, vì vậy ta sẽ triển khai is\_valid () là hàm trợ giúp:

*bool Date::is\_valid()*

*{*

*return is\_date(d,m,y)*

*}*

Tại sao lại có cả *is\_valid ()* và *is\_date ()* ?Trong ví dụ đơn giản này, chúng ta có thể quản lý chỉ với một, nhưng ta có thể tưởng tượng các hệ thống trong đó *is\_date ()* (như ở đây) kiểm tra xem a (d, m, y) có đại diện cho một ngày hợp lệ và *is\_valid ()* thực hiện kiểm tra bổ sung liệu ngày đó có thể được trình bày một cách hợp lý hay không. Ví dụ: *is\_valid ()* có thể từ chối các ngày từ trước khi lịch hiện đại được sử dụng phổ biến.

Như thường thấy đối với các loại cụ thể đơn giản như vậy, các định nghĩa cụ thể của các hàm thành viên của Date khác nhau giữa mức độ nhỏ và mức độ không quá phức tạp. Ví dụ:

*inline int Date::day() const*

*{*

*return d;*

*}*

*Date& Date::add\_month(int n)*

*{*

*if (n==0) return ∗this;*

*if (n>0) {*

*int delta\_y = n/12; // number of whole years (số cả năm)*

*int mm = static\_cast<int>(m)+n%12; // number of months ahead (số tháng trước)*

*if (12 < mm) { // note: dec is represented by 12 (lưu ý: số thập phân được biểu thị bằng 12)*

*++delta\_y;*

*mm −= 12;*

*}*

*// ... handle the cases where the month mm doesn’t have day d ... (... xử lý các trường hợp tháng mm không có ngày d.)*

*y += delta\_y;*

*m = static\_cast<Month>(mm);*

*return ∗this;*

*}*

*// ... handle negative n ...( xử lý n)*

*return ∗this;*

*}*

#### 16.3.2 Hàm trợ giúp (Helper Functions)

Thông thường, một lớp có một số hàm được liên kết với nó mà không cần phải xác định trong chính lớp đó vì chúng không cần quyền truy cập trực tiếp vào. Ví dụ:

*int diff(Date a, Date b); // number of days in the range (số ngày trong phạm vi)*

*bool is\_leapyear(int y);*

*bool is\_date(int d, Month m, int y);*

*const Date& default\_date();*

*Date next\_weekday(Date d);*

*Date next\_saturday(Date d);*

Việc loại bỏ các hàm như vậy trong chính lớp sẽ làm phức tạp giao diện lớp và tăng số lượng các hàm có thể cần được kiểm tra khi xem xét sự thay đổi đối với biểu diễn.

Làm thế nào những hàm như vậy '' kết hợp '' với lớp Date? Trong thời kỳ đầu của C ++, cũng như trong C, các khai báo của chúng đơn giản được đặt trong cùng thứ tự với khai báo của lớp Date. Người dùng cần Date sẽ cung cấp tất cả chúng bằng cách đưa vào giao diện. Ví dụ:

*#include "Date.h"*

Ngoài ra, chúng ta có thể làm cho liên kết rõ ràng bằng cách bao bọc lớp và các hàm trợ giúp của nó trong một không gian tên:

*namespace Chrono { // facilities for dealing with time*

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

*int diff(Date a, Date b);*

*bool is\_leapyear(int y);*

*bool is\_date(int d, Month m, int y);*

*const Date& default\_date();*

*Date next\_weekday(Date d);*

*Date next\_saturday(Date d);*

*// ...*

*}*

Đương nhiên, hàm trợ giúp phải được định nghĩa ở đâu đó:

*bool Chrono::is\_date(int d, Month m, int y)*

*{*

*int ndays;*

*switch (m) {*

*case Month::feb:*

*ndays = 28+is\_leapyear(y);*

*break;*

*case Month::apr: case Month::jun: case Month::sep: case Month::nov:*

*ndays = 30;*

*break;*

*case Month::jan: case Month::mar: case Month::may: case Month::jul:*

*case Month::aug: case Month::oct: case Month::dec:*

*ndays = 31;*

*break;*

*default:*

*return false;*

*}*

*return 1<=d && d<=ndays;*

*}*

Tránh việc chọn giá trị month ngoài phạm vi từ jan đến dec, ta cần phải kiểm tra:

*const Date& Chrono::default\_date()*

*{*

*static Date d {1,Month::jan,1970};*

*return d;*

*}*

#### 16.3.3 Nạp chồng toán tử (Overloaded Operators)

Nó thường hữu ích khi thêm các chức năng để kích hoạt ký hiệu thông thường. Ví dụ: toán tử == () xác định toán tử bình đẳng, ==, hoạt động cho Date:

*inline bool operator==(Date a, Date b) // equality (bình đẳng)*

*{*

*return a.day()==b.day() && a.month()==b.month() && a.year()==b.year();*

*}*

Khác:

*bool operator!=(Date, Date); // inequality*

*bool operator<(Date, Date); // less than*

*bool operator>(Date, Date); // greater than*

*// ...*

*Date& operator++(Date& d) { return d.add\_day(1); } // increase Date by one day*

*Date& operator−−(Date& d) { return d.add\_day(−1); } // decrease Date by one day*

*Date& operator+=(Date& d, int n) { return d.add\_day(n); } // add n days*

*Date& operator−=(Date& d, int n) { return d.add\_day(−n); } // subtract n days*

*Date operator+(Date d, int n) { return d+=n; } // add n days*

*Date operator−(Date d, int n) { return d+=n; } // subtract n days*

*ostream& operator<<(ostream&, Date d); // output d*

*istream& operator>>(istream&, Date& d); // read into d*

Các toán tử này được xác định trong Chrono cùng với Date để tránh các vấn đề quá tải và lợi ích từ việc tra cứu phụ thuộc vào đối số.

Đối với Date, những toán tử này có thể được coi là những tiện ích đơn thuần. Tuy nhiên, đối với nhiều kiểu - chẳng hạn như số phức, vectơ, và các đối tượng giống hàm - việc sử dụng các toán tử thông thường rất khó sử dụng tâm trí của mọi người rằng việc xác định quan điểm của họ gần như là bắt buộc. Đối với Date, tôi đã muốn cung cấp + = và - = dưới dạng các hàm thành viên thay vì add\_day ().

#### 16.3.4 Dấu hiệu các lớp cụ thể ( The Signiﬁcance of Concrete Classes)

Gọi các kiểu do người dùng xác định là đơn giản, chẳng hạn như Date, các kiểu cụ thể để phân biệt chúng với các lớp trừu tượng và phân cấp lớp, và cũng để nhấn mạnh sự giống nhau của chúng với các kiểu tích hợp sẵn như int và char . Các lớp cụ thể được sử dụng giống như các loại tích hợp sẵn. Các kiểu cụ thể còn được gọi là kiểu giá trị và lập trình hướng giá trị sử dụng của chúng. Mô hình sử dụng của chúng và '' triết lý '' đằng sau thiết kế của chúng hoàn toàn khác với những gì thường được gọi là lập trình hướng đối tượng.

Mục đích của một kiểu cụ thể là làm tốt một việc duy nhất, tương đối đơn giản và hiệu quả. Nó thường không nhằm mục đích cung cấp cho người dùng các phương tiện để sửa đổi hành vi của một loại cụ thể. Trong đặc biệt, các loại cụ thể không nhằm mục đích hiển thị hành vi đa hình.

Nếu bạn không thích một số chi tiết của một loại cụ thể, bạn sẽ tạo một loại mới với hành vi mong muốn. Nếu bạn muốn '' tái sử dụng '' một kiểu cụ thể, bạn sử dụng chính xác nó trong việc triển khai kiểu mới của mìnhnhư bạn đã sử dụng int. Ví dụ:

*class Date\_and\_time {*

*private:*

*Date d;*

*Time t;*

*public:*

*Date\_and\_time(Date d, Time t);*

*Date\_and\_time(int d, Date::Month m, int y, Time t);*

*// ...*

*};*

**16.4 Lời khuyên (Advice)**

[1] Biểu diễn các khái niệm dưới dạng các lớp; §16.1.

[2] Tách giao diện của một lớp khỏi việc triển khai nó.

[3] Chỉ sử dụng dữ liệu công khai (cấu trúc) khi nó thực sự chỉ là dữ liệu và không có bất biến nào có ý nghĩa đối với dữ liệu thành viên.

[4] định nghĩa một phương thức khởi tạo để xử lý việc khởi tạo các đối tượng.

[5] Theo mặc định, khai báo các hàm tạo đối số đơn là tường minh.

[6] Khai báo một hàm thành viên không sửa đổi trạng thái của đối tượng của nó là const.

[7] Loại cụ thể là loại đơn giản nhất của lớp. Nếu có thể, thích một kiểu cụ thể hơn các lớp phức tạp hơn và các cấu trúc dữ liệu đơn giản.

[8] Chỉ biến một hàm thành thành viên nếu nó cần truy cập trực tiếp vào biểu diễn của một lớp.

[9] Sử dụng một không gian tên để làm cho mối liên kết giữa một lớp và các chức năng trợ giúp của nó trở nên rõ ràng.

[10] Biến một hàm thành viên không sửa đổi giá trị của đối tượng thành một hàm thành viên const.

[11] Tạo một hàm cần truy cập vào biểu diễn của một lớp nhưng không cần gọi đối tượng cụ thể thành một hàm thành viên tĩnh; §16.2.12.

## Chapter 17: Construction, Cleanup, Copy, and Move

### 17.1 Giới thiệu (Introduction)

Chương này tập trung vào các khía cạnh kỹ thuật của ‘‘ vòng đời ’’ của đối tượng: làm cách nào để chúng ta tạo một đối tượng, cách chúng ta sao chép nó, cách chúng ta di chuyển nó và làm cách nào để dọn dẹp nó sau khi nó biến mất? Định nghĩa thích hợp của ‘‘ copy ’’ và ‘‘ move ’’ là gì? Ví dụ:

*string ident(string arg) // string passed by value (copied into arg) (chuỗi được truyền bởi giá trị (được sao chép vào arg) )*

*{*

*return arg; // return string (move the value of arg out of ident() to a caller) (trả về chuỗi (di chuyển giá trị của arg ra khỏi danh tính () đến một người gọi) )*

*}*

*int main ()*

*{*

*string s1 {“Adams”}; // initialize string (construct in s1). ( khởi tạo chuỗi (cấu trúc trong s1) )*

*s1 = indet(s1); // copy s1 into ident() (sao chép s1 vào ident () )*

*// move the result of ident(s1) into s1; (di chuyển kết quả của ident (s1) vào s1 )*

*// s1’s value is “Adams”. (giá trị của s1 là &quot;Adams&quot; )*

*string s2 {“Pratchett”}; // initialize string (construct in s2) (/khởi tạo chuỗi (cấu trúc trong*

*s2) )*

*s1 = s2; // copy the value of s2 into s1 (sao chép giá trị của s2 vào s1)*

*// both s1 and s2 have the value “Pratchett”. (cả s1 và s2 đều có giá trị “Pratchett”)*

*}*

Rõ ràng, sau lời gọi của ident (), giá trị của s1 phải là “Adams”. Chúng ta sao chép giá trị của s1 vào đối số arg, sau đó chúng ta chuyển giá trị của arg ra khỏi lệnh gọi hàm và (quay lại) vào s1. Tiếp theo, chúng ta xây dựng s2 với giá trị “Prachett” và sao chép nó vào s1. Cuối cùng, ở lối ra khỏi main (), chúng ta hủy các biến s1 và s2. Sự khác biệt giữa di chuyển và sao chép là sau khi sao chép, hai đối tượng phải có cùng giá trị, trong khi sau khi di chuyển, nguồn của di chuyển không bắt buộc phải có giá trị ban đầu của nó. Các chuyển động có thể được sử dụng khi đối tượng nguồn sẽ không được sử dụng nữa. Chúng đặc biệt hữu ích để phù hợp với yêu cầu khi di chuyển tài nguyên.

Một số chức năng được sử dụng ở đây:

* Một hàm tạo khởi tạo một chuỗi với một chuỗi ký tự (được sử dụng cho s1 và s2)
* Một phương thức tạo bản sao sao chép một chuỗi (vào đối số hàm arg)
* Một hàm tạo di chuyển di chuyển giá trị của một chuỗi (từ đối số ra khỏi danh tính () thành một biến tạm thời giữ kết quả là danh tính (s1))
* Phép gán di chuyển di chuyển giá trị của một chuỗi (từ biến tạm thời giữ kết quả của danh tính (s1) thành s1)
* Một nhiệm vụ sao chép sao chép một chuỗi (từ s2 sang s1)
* Một trình hủy giải phóng các tài nguyên thuộc sở hữu của s1, s2 và biến tạm thời giữ kết quả của ident (s1).
* Trình khởi tạo, sao chép và di chuyển các hoạt động gánvà trình hủy hổ trợ trực tiếp đến chế độ xem, quản lý tài nguyên và thời gian tồn tại. Một đối tượng được coi là đối tượng thuộc kiểu của nó sau khi hàm khởi tạo nó hoàn thành và nó không phải là một đối tượng thuộc kiểu của nó khi hàm hủy tiền hành xong.

Xây dựng các đối tượng đóng vai trò quan trọng trong nhiều thiết kế. Sự đa dạng

về cách sử dụng này được tái hiện trong pham vi và tính khả dụng của các tính năng ngôn ngữ hỗ trợ khởi tạo.

Các hàm tạo, hàm hủy và các thao tác sao chép và di chuyển cho một kiểu khônng tách biệt nhau về mặt logic. Chúng ta phải xác định chúng như một tập hợp hoặc các vấn đề logic hoặc hiệu suất. Mếu một lớp X có một hàm hủy thực hiện một tác vụ tầm thường, chẳng hạn như phân kho lưu trữ miến phí hoặc giải phóng khóa, thì lớp đó có thể cần bổ sung đầy đủ các chức năng:

*class X {*

*X(Sometype); // ‘‘ordinary constructor’’: create an object (hàm tạo bình thường : tạo một đối tượng)*

*X(); // default constructor (khởi tạo mặc định)*

*X(const X&); // copy constructor (khởi tạo sao chép)*

*X(X&&); // move constructor (hàm di chuyển)*

*X& operator=(const X&); // copy assignment: clean up target and copy ( sao chép đối tượng : xóa đối tượng và sao chép)*

*X operator=(X&&); // move assignment: clean up target and move (di chuyển đối tượng: xóa đối tượng và di chuyển)*

*˜X(); // destructor: clean up (hàm hủy)*

*// ...*

*};*

Có những tình huống trong đó một đối tượng được sao chép hoặc di chuyển:

* Là nguồn gốc của một nhiệm vụ.
* Là một bộ khởi tạo đối tượng.
* Là một đối số của hàm.
* Là một hàm trả vè giá trị.
* Như một ngoại lệ.

Trong mọi trường hợp, hàm tạo sao chép hoặc di chuyển sẽ được áp dụng (trừ khi nó có thể được tối ưu hóa đi).

### 17.2 Hàm khởi tạo và hàm hủy (Constructors and Destructors)

Chúng ta có thể chỉ định cách khởi tạo một đối tượng của một lớp bằng cách xác định hàm tạo. Để bổ sung các hàm tạo, chúng ta có thể xác định hàm hủy để đảm bảo “dọn dẹp” một đối tượng khi cần.

*struct Tracer {*

*string mess;*

*Tracer(const string& s) :mess{s} { clog << mess; }*

*˜Tracer() {clog << "˜" << mess; }*

*};*

*void f(const vector<int>& v)*

*{*

*Tracer tr {"in f()\n"};*

*for (auto x : v) {*

*Tracer tr {string{"v loop "}+to<string>(x)+'\n'}; // §25.2.5.1*

*// ...*

*}*

*}*

Chúng ta có thể thử gọi:

*f({2,3,5});*

Điều này sẽ in ra luồng ghi:

*in\_f()*

*v loop 2*

*~v loop 2*

*v loop 3*

*~v loop 3*

*v loop 5*

*~v loop 5*

*~in\_f()*

#### 17.2.1 Hàm khởi tạo và tính bất biến (Constructors and Invariants)

Một thành viên có tên giống với tên lớp chứa nó thì được gọi là một phương thức khởi tạo. Ví dụ:

*class Vector {*

*public:*

*Vector(int s);*

*// ...*

*};*

Một khai báo phương thức khởi tạo chỉ định một danh sách đối số (chính xác như đối với một hàm) nhưng không có kiểu trả về. Tên của một lớp không thể được sử dụng cho một hàm thành viên thông thường, thành viên dữ liệu, kiểu thành viên, v.v., trong lớp. Ví dụ:

*struct S {*

*S(); // ﬁne*

*void S(int); // error : no type can be speciﬁed for a constructor (có void và không phải là hàm*

*tạo)*

*int S; // error : the class name must denote a constructor (đây không phải là một hàm thành*

*viên và không có int )*

*enum S { foo, bar }; // error : the class name must denote a constructor (không có enum và*

*đây không phải là một hàm)*

*};*

Công việc của một phương thức khởi tạo là khởi tạo một đối tượng của lớp nó. Thông thường, việc khởi tạo đó phải thiết lập một lớp bất biến, tức là cái gì đó phải giữ bất cứ khi nào một hàm thành viên được gọi (từ bên ngoài lớp). Xem xét:

*class Vector {*

*public:*

*Vector(int s);*

*// ...*

*private:*

*double∗ elem; // elem points to an array of sz doubles ( elem trỏ đến một mảng của sz nhân*

*đôi*

*int sz; // sz is non-negative (sz không âm)*

*};*

Ở đây, giá trị bất biến được phát biểu dưới dạng nhận xét: ‘‘ elem trỏ tới một mảng sz nhân đôi ’’ và ‘‘ sz là không âm’’. Hàm tạo phải làm cho điều đó đúng. Ví dụ:

*Vector::Vector(int s)*

*{*

*if (s&lt;0) throw Bad\_siz e{s};*

*sz = s;*

*elem = new double[s];*

*}*

Hàm tạo này cố gắng thiết lập bất biến và nếu không thể, nó sẽ ném ra một ngoại lệ. Nếu phương thức khởi tạo không thể thiết lập bất biến, không có đối tượng nào được tạo ra và phương thức khởi tạo phải đảm bảo rằng không có tài nguyên nào bị rò rỉ. Tài nguyên là bất cứ thứ gì chúng ta cần để có được và cuối cùng trả lại khi chúng ta đã hoàn thành với nó.

Tại sao bạn lại coi là bất biến?

* Để tập trung nỗ lực thiết kế cho lớp.
* Để làm rõ hành vi của lớp.
* Để đơn giản hóa khái niệm của các hàm thành viên.
* Để làm rõ việc quản lý tài nguyên của lớp.
* Để đơn giản hóa tài liệu của lớp.

Trung bình, nỗ lực để xác định một bất biến cuối cùng sẽ tiết kiệm được công việc.

#### 17.2.2 Hàm hủy và tài nguyên (Destructors and Resources)

Một hàm tạo khởi tạo một đối tượng. Nói cách khác, nó tạo ra môi trường mà các hàm thành viên hoạt động. Khi môi trường được tạo ra để sử dụng thì đồng nghĩa với việc phải giải phóng môi trường đó sau khi sử dụng. Do đó, một số lớp cần có một hàm để tiến hành giải phóng môi trường đó - một hàm như vậy được gọi là hàm hủy. Cú pháp của hàm hủy là ˜ theo sau là tên lớp, ví dụ ~Vector (). Một nghĩa của ˜ là ‘‘ bổ sung ’’ và hàm hủy cho một lớp bổ sung cho các hàm tạo của nó. Một hàm hủy không nhận đối số và một lớp chỉ có thể có một hàm hủy. Hàm hủy được gọi ngầm khi một biến tự động vượt ra khỏi phạm vi, một đối tượng bị xóa, v.v. Chỉ trong những trường hợp rất hiếm, người dùng mới cần gọi hàm hủy một cách rõ ràng.

Hàm hủy thường dọn dẹp và gải phóng tài nguyên. Ví dụ:

*class Vector {*

*public:*

*Vector(int s) :elem{new double[s]}, sz{s} { }; // constructor: acquire memo (hàm khởi tạo :*

*ghi bộ nhớ)*

*~Vector() { delete[] elem; } // destructor: release memor (hàm hủy: giải phóng bộ nhớ)*

*// ...*

*private:*

*double∗ elem; // elem points to an array of sz doubles*

*int sz; // sz is non-negative*

*};*

Ví dụ thêm:

*Vector∗ f(int s)*

*{*

*Vector v1(s);*

*// ...*

*return new Vector(s+s);*

*}*

*void g(int ss)*

*{*

*Vector∗ p = f(ss);*

*// ...*

*delete p;*

*}*

Tại đây, Vector v1 bị hủy khi thoát khởi f(). Ngoài ra, Vector được tạo bởi f() sử dụng new sẽ bị hủy bởi lệnh xóa. Trong hai trường hợp, hàm hủy của Vector được gọi để giải phóng bộ nhớ được cấp phát bởi hàm khởi tạo.

Phong cách quản lý tài nguyên dựa trên phương thức khởi tạo / hủy này được gọi là Khởi tạo Tài nguyên (Resource Acquisition Is Initialization) hoặc đơn giản là RAII.

Một cặp hàm tạo / hủy phù hợp là cơ chế thông thường để thực hiện khái niệm về một đối tượng có kích thước thay đổi trong C ++. Các vùng chứa thư viện tiêu chuẩn, chẳng hạn như vectơ và bản đồ không có thứ tự, sử dụng các biến thể của kỹ thuật này để cung cấp lưu trữ cho các phần tử của chúng. Một kiểu không có trình hủy được khai báo, chẳng hạn như kiểu tích hợp, được coi là có trình hủy không làm gì cả.

Một lập trình viên khai báo hàm hủy cho một lớp cũng phải quyết định xem các đối tượng của lớp đó có thể được sao chép hoặc di chuyển hay không.

#### 17.2.3 Cơ sở và hàm hủy thành viên (Base and Member Destructors)

Hàm tạo và hàm hủy tương tác chính xác với cấu trúc phân cấp lớp. Một hàm tạo xây dựng một đối tượng lớp ‘‘ từ dưới lên ’’:

[1] Đầu tiên, hàm tạo gọi các hàm tạo lớp cơ sở của nó,

[2] sau đó, nó gọi các hàm tạo thành viên, và

[3] Cuối cùng, nó thực thi cơ thể của chính nó.

Một hàm hủy “xé toạc”một đối tượng theo thứ tự ngược lại:

[1] Đầu tiên, trình hủy thực thi phần thân của chính nó,

[2] sau đó, nó gọi các trình hủy thành viên của nó, và

[3] Cuối cùng, nó sử dụng các hàm hủy lớp cơ sở của nó.

Đặc biệt, một cơ sở ảo được xây dựng trước bất kỳ cơ sở nào có thể sử dụng nó và bị phá hủy sau tất cả những cơ sở đó. Thứ tự này đảm bảo rằng một cơ sở hoặc một thành viên không được sử dụng trước khi nó được khởi tạo hoặc được sử dụng sau khi nó đã bị phá hủy. Các hàm tạo thực thi các hàm tạo thành viên và các hàm cơ sở theo thứ tự khai báo (không phải thứ tự của các trình khởi tạo): nếu hai hàm tạo sử dụng một thứ tự khác nhau, bộ hủy không thể (không có quá đầu nghiêm trọng) đảm bảo sẽ phá hủy theo thứ tự ngược lại của cấu trúc.

Nếu một lớp được sử dụng để cần một hàm tạo mặc định và nếu lớp đó không có các hàm tạo khác, trình biên dịch sẽ cố gắng tạo một hàm tạo mặc định. Ví dụ:

*struct S1 {*

*string s;*

*};*

*S1 x; // OK: x.s is initialized to &quot;&quot; (x.s được khởi tạo thành “ “)*

Tương tự, khởi tạo thành viên có thể được sử dụng nếu cần các trình khởi tạo. Ví dụ:

*struct X { X(int); };*

*struct S2 {*

*Xx;*

*};*

*S2 x1; // error :*

*S2 x2 {1}; // OK: x2.x is initialized with 1 ( x2.x được khởi tạo bằng 1)*

#### 17.2.4 Gọi hàm tạo và hàm hủy (Calling Constructors and Destructors)

Một hàm hủy được gọi ngầm khi thoát khỏi phạm vi hoặc bằng cách xóa. Thông thường không chỉ cần gọi hàm hủy một cách rõ ràng; làm như vậy sẽ dẫn đến các lỗi khó chịu. Tuy nhiên, có một số trường hợp hiếm (nhưng quan trọng) mà hàm hủy phải được gọi một cách rõ ràng. Hãy xem xét một vùng chứa (như std :: vector) duy trì một nhóm bộ nhớ mà nó có thể phát triển và thu nhỏ (ví dụ: sử dụng *push\_back ()* và *pop\_back ().* Khi chúng ta thêm một phần tử, vùng chứa phải gọi phương thức khởi tạo của nó cho một địa chỉ cụ thể:

*void C::push\_back(const X&amp; a)*

*{*

*// ...*

*new(p) X{a}; // copy construct an X with the value a in address p (sao chép cấu trúc X với giá*

*trị a trong địa chỉ p)*

*// ...*

*}*

Việc sử dụng hàm tạo này được gọi là “vị trí mới”.

Ngược lại, khi chúng ta xóa một phần tử, vùng chứa cần gọi hàm hủy của nó:

*void C::pop\_back()*

*{*

*p−>~X(); // destroy the X in address p (hủy X trong địa chỉ p) }*

Ký hiệu *p−> ~X ()* gọi hàm hủy của X cho ∗ p. Ký hiệu đó không bao giờ được sử dụng cho một đối tượng bị phá hủy theo cách thông thường (do đối tượng của nó vượt ra khỏi phạm vi hoặc bị xóa).

Nếu được khai báo cho một lớp X, một hàm hủy sẽ được gọi ngầm bất cứ khi nào một X vượt ra khỏi phạm vi hoặc bị xóa. Điều này ngụ ý rằng chúng ta có thể ngăn chặn việc phá hủy X bằng cách khai báo hàm hủy = delete hoặc private.

Trong số hai lựa chọn thay thế, sử dụng private là khả thi hơn. Ví dụ: chúng ta có thể tạo một lớp mà các đối tượng có thể bị hủy một cách rõ ràng, nhưng không hoàn toàn:

*class Nonlocal {*

*public:*

*// ...*

*void destroy() { this−>~Nonlocal(); } // explicit destruction (hủy tường minh)*

*private:*

*// ...*

*~Nonlocal(); // don’t destroy implicitly (đừng hủy ngầm)*

*};*

*void user()*

*{*

*Nonlocal x; // error : cannot destroy a Nonlocal ( không thể hủy bỏ một Nonlocal)*

*X∗ p = new Nonlocal; // OK*

*// ...*

*delete p; // error : cannot destroy a Nonlocal (không thể hủy bỏ một Nonlocal)*

*p.destroy(); // OK*

*}*

#### 17.2.5 Hàm hủy ảo (Virtual Destructors)

Một hàm hủy có thể được khai báo là ảo và thường phải dành cho một lớp có hàm ảo. Ví dụ:

*class Shape {*

*public:*

*// ...*

*virtual void draw() = 0;*

*virtual ~Shape();*

*};*

*class Circle {*

*public:*

*// ...*

*void draw();*

*~Circle(); // overrides ~Shape() (ghi đè ~Shape() )*

*// ...*

*};*

Lý do chúng ta cần một trình hủy ảo là một đối tượng thường được thao tác thông qua giao diện được cung cấp bởi một lớp cơ sở cũng thường bị xóa thông qua giao diện đó:

*void user(Shape∗ p)*

*{*

*p−>draw(); // invoke the appropriate draw() (gọi hàm thích hợp: draw() )*

*// ...*

*delete p; // invoke the appropriate destructor (gọi hàm hủy thích hợp)*

*};*

Nếu bộ hủy của Shape không phải là ảo thì việc xóa sẽ không thể gọi hàm hủy của lớp dẫn xuất thích hợp (ví dụ: ~Circle ()). Thất bại đó sẽ làm cho các tài nguyên thuộc sở hữu của đối tượng bị xóa (nếu có) bị rò rỉ.

### 17.3 Khởi tạo đối tượng lớp (Class Object Initialization)

Phần này thảo luận về cách khởi tạo các đối tượng của một lớp có và không có hàm tạo. Nó cũng chỉ ra cách xác định các hàm tạo chấp nhận danh sách hàm khởi tạo đồng nhất có kích thước tùy ý (chẳng hạn như {1,2,3} và {1,2,3,4,5,6}).

#### 17.3.1 Khởi tạo không có trình tạo (Initialization Without Constructors)

Chúng ta không thể xác định hàm tạo cho một kiểu có sẵn, nhưng chúng ta có thể khởi tạo nó với một giá trị của kiểu phù hợp. Ví dụ:

*int a {1};*

*char∗ p {nullptr};*

Tương tự, chúng ta có thể khởi tạo các đối tượng của một lớp mà chúng ta chưa xác định hàm tạo bằng cách sử dụng:

* Khởi tạo thành viên,
* Sao chép khởi tạo, hoặc
* Khởi tạo mặc định.

Ví dụ:

*struct Work {*

*string author;*

*string name;*

*int year;*

*};*

*Work s9 {“Beethoven”, “Symphony No. 9 in D minor, Op. 125; Choral“ 1824 }; // memberwise initialization (khởi tạo thành viên)*

*Work currently\_playing { s9 }; // copy initialization (khởi tạo sao chép)*

*Work none {}; // default initialization (khởi tạo mặc định)*

Ba thành viên của current\_playing là bản sao của s9.

Khởi tạo mặc định của việc sử dụng {} được coi là khởi tạo của từng thành viên bởi {}. Vì vậy, không có nào được khởi tạo thành {{}, {}, {}}, là {*““, ““,* 0} .

Trường hợp không có hàm tạo nào yêu cầu đối số được khai báo, thì cũng có thể loại bỏ hoàn toàn hàm khởi tạo. Ví dụ:

*Work alpha;*

*void f()*

*{*

*Work beta;*

*// ...*

*}*

Đối với điều này, các quy tắc không rõ ràng như chúng ta có thể muốn. Đối với các đối tượng được cấp phát tĩnh (§6.4.2), các quy tắc chính xác như thể bạn đã sử dụng {}, vì vậy giá trị của alpha là {*““, ““,*0}. Tuy nhiên, đối với các biến cục bộ và các đối tượng lưu trữ miễn phí, việc khởi tạo mặc định chỉ được thực hiện cho các thành viên của loại lớp và các thành viên của loại tích hợp không được khởi tạo, vì vậy giá trị của beta là {&quot;&quot;, &quot;&quot;, không xác định}.

Lý do cho sự phiền phứ này là để cải thiện hiệu suất trong những trường hợp nguy cấp hiếm gặp. Ví dụ:

*struct Buf {*

*int count;*

*char buf[16∗1024];*

*};*

Bạn có thể sử dụng Buf làm biến cục bộ mà không cần khởi tạo nó trước khi sử dụng nó làm mục tiêu cho thao tác nhập. Hầu hết các khởi tạo biến cục bộ không quan trọng về hiệu suất và các biến cục bộ chưa được khởi tạo là một nguồn lỗi chính. Nếu bạn muốn khởi tạo được đảm bảo hoặc đơn giản là không thích bất ngờ, hãy cung cấp một trình khởi tạo, chẳng hạn như {}. Ví dụ:

*Buf buf0; // statically allocated, so initialized by default (được cấp phát tĩnh nên được khởi*

*tạo theo mặc định)*

*void f()*

*{*

*Buf buf1; // leave elements uninitialized (các yếu tố chưa được khởi tạo)*

*Buf buf2 {}; //*

*int∗ p1 = new int; // \*p1 is uninitialized (\* p1 chưa được khởi tạo)*

*int∗ p2 = new int{}; // \*p2 == 0*

*int∗ p3 = new int{7}; // \*p3 == 7*

*// ...*

*}*

Khởi tạo thành viên chỉ hoạt động nếu chúng ta có thể truy cập các thành viên. Ví dụ:

*Template<class T>*

*class Checked\_pointer { // control access to T\* member (kiểm soát quyền truy cập thành viên*

*T \*)*

*public:*

*T& operator∗(); // check for nullptr and return value (kiểm tra nullptr và giá trị trả về)*

*// ...*

*};*

*Checked\_pointer<int> p {new int{7}}; // error : can’t access p.p (lỗi: không thể truy cập*

*p.p)*

Nếu một lớp có một thành viên dữ liệu riêng tư không tĩnh, nó cần một phương thức khởi tạo để khởi tạo nó.

#### 17.3.2 Khởi tạo bằng cách sử dụng hàm tạo (Initialization Using Constructors)

Trong trường hợp bản sao thành viên không phù hợp hoặc không mong muốn, một phương thức khởi tạo có thể được xác định để khởi tạo một đối tượng. Đặc biệt, một phương thức khởi tạo thường được sử dụng để thiết lập một bất biến cho lớp của nó và thu được các tài nguyên cần thiết để thực hiện điều đó.

Nếu một hàm tạo được khai báo cho một lớp, một số hàm tạo sẽ được sử dụng cho mọi đối tượng. Đó là một lỗi khi cố gắng tạo một đối tượng mà không có bộ khởi tạo thích hợp theo yêu cầu của các hàm tạo. Ví dụ:

*struct X {*

*X(int);*

*};*

*X x0; // error : no initializer (lỗi: không có trình khởi tạo)*

*X x1 {}; // error : empty initializer (lỗi: hàm khởi tạo trống)*

*X x2 {2}; // OK*

*X x3 {”two”}; // error : wrong initializer type (lỗi: loại hàm khởi tạo sai)*

*X x4 {1,2}; // error : wrong number of initializers (đầu vào hàm khởi tạo sai)*

*X x5 {x4}; // OK: a copy constructor is implicitly deﬁned (phương thức khởi tạo sao chép mặc*

*định)*

Lưu ý rằng hàm tạo mặc định biến mất khi bạn xác định hàm tạo yêu cầu đối số; xét cho cùng, X (int) nói rằng một int là bắt buộc để tạo ra một X. Tuy nhiên, hàm tạo sao chép không biến mất; giả định là một đối tượng có thể được sao chép (khi được xây dựng đúng cách).

Ký hiệu {} để khởi tạo có thể được sử dụng để cung cấp các đối số cho một phương thức khởi tạo ở bất cứ nơi nào một đối tượng có thể được xây dựng. Ví dụ:

*struct Y : X {*

*X m {0}; // provide default initializer for member m (cung cấp trình khởi tạo mặc định cho*

*thành viên m)*

*Y(int a) :X{a}, m{a} { }; // initialize base and member (khởi tạo cơ sở và thành viên)*

*Y() : X{0} { }; // initialize base and member (khởi tạo cơ sở và thành viên)*

*};*

*X g {1}; // initialize global variable (khởi tạo biến toàn cục)*

*void f(int a)*

*{*

*X def {}; // error : no default value for X (lỗi: không có giá trị mặc định cho X)*

*Y de2 {}; // OK: use default constructor (OK: sử dụng hàm tạo mặc định)*

*X∗ p {nullptr};*

*X var {2}; // initialize local variable (khởi tạo biến cục bộ)*

*p = new X{4}; // initialize object on free store (khởi tạo đối tượng trên cửa hàng miễn phí)*

*X a[] {1,2,3}; // initialize array elements (khởi tạo các phần tử mảng)*

*Vector<X> v {1,2,3,4}; // initialize vector elements (khởi tạo các phần tử vector)*

*}*

Vì lý do này, khởi tạo {} đôi khi được gọi là khởi tạo phổ quát: có thể được sử dụng ở mọi nơi. Ngoài ra, việc khởi tạo {} là đồng nhất: bất cứ nơi nào bạn khởi tạo một đối tượng kiểu X với giá trị v bằng ký hiệu {v}, giá trị tương tự của kiểu X (X{v}) sẽ được tạo.

Các ký hiệu = và () để khởi tạo không phổ biến. Ví dụ:

*struct Y : X {*

*Xm;*

*Y(int a) : X(a), m=a { }; // syntax error: can’t use = for member initialization (lỗi cú pháp:*

*không thể sử dụng = để khởi tạo thành viên)*

*};*

*X g(1); // initialize global variable (khởi tạo biến toàn cục)*

*void f(int a)*

*{*

*X def(); // function returning an X (hàm trả về X )*

*X∗ p {nullptr};*

*X var = 2; // initialize local variable (khởi tạo biến cục bộ)*

*p = new X=4; // syntax error: can’t use = for new (lỗi cú pháp: không thể sử dụng = cho*

*new)*

*X a[](1,2,3); // error : can’t use () for array initialization (error: không thể sử dụng () để khởi*

*tạo mảng)*

*vector<X>v(1,2,3,4); // error : can’t use () for list elements (lỗi: không thể sử dụng () cho*

*các phần tử danh sách)*

*}*

Các quy tắc giải quyết vấn đề quá tải thông thường áp dụng cho các hàm tạo. Ví dụ:

*struct S {*

*S(const char∗);*

*S(double∗);*

*};*

*S s1 {“Napier”}; // S::S(const char\*)*

*S s2 {new double{1.0}}; // S::S(double\*);*

*S s3 {nullptr}; // ambiguous: S::S(const char\*) or S::S(double\*)? (không rõ ràng: S :: S (const char \*) hay S :: S (double \*)?)*

Lưu ý rằng ký hiệu {} khởi tạo không cho phép thu hẹp. Đó là một lý do khác để thích kiểu {} hơn () hoặc =.

##### 17.3.2.1 Khởi tạo bởi hàm khởi tạo (Initialization by Constructors)

Sử dụng ký hiệu (), bạn có thể yêu cầu sử dụng một hàm tạo trong một lần khởi tạo. Nghĩa là, bạn có thể đảm bảo rằng đối với một lớp, bạn sẽ được khởi tạo bởi hàm tạo và không nhận được khởi tạo thành viên hoặc khởi tạo danh sách khởi tạo mà ký hiệu {} cũng cung cấp. Ví dụ:

*struct S1 {*

*int a,b; // no constructor (không có khởi tạo)*

*};*

*struct S2 {*

*int a,b;*

*S2(int a = 0, int b = 0) : a(aa), b(bb) {} // constructor (khởi tạo)*

*};*

*S1 x11(1,2); // error : no constructor (lỗi : không khởi tạo)*

*S1 x12 {1,2}; // OK: memberwise initialization (OK: khởi tạo thành viên)*

*S1 x13(1); // error : no constructor (lỗi: không có hàm tạo)*

*S1 x14 {1}; // OK: x14.b becomes 0 (OK: x14.b trở thành 0)*

*S2 x21(1,2); // OK: use constructor (OK: sử dụng hàm tạo)*

*S2 x22 {1,2}; // OK: use constructor (OK: sử dụng hàm tạo)*

*S2 x23(1); // OK: use constructor and one default argument (OK: sử dụng hàm tạo và một đối số mặc định)*

*S2 x24 {1}; // OK: use constructor and one default argument (OK: sử dụng hàm tạo và một đối số mặc định)*

Việc sử dụng thống nhất khởi tạo {} chỉ trở nên khả thi trong C ++ 11, vì vậy mã C ++ cũ hơn sử dụng () và = khởi tạo. Do đó, dấu () và = có thể sử dụng quen thuộc hơn . Tuy nhiên, không có bất kỳ lý do hợp lý nào để thích ký hiệu () ngoại trừ trường hợp hiếm hoi cần phân biệt giữa khởi tạo với danh sách các phần tử và danh sách các đối số của phương thức khởi tạo. Ví dụ:

*Vector<int>v1 {77}; // one element with the value 77 ( một phần tử có giá trị 77)*

*Vector<int>v2(77); // 77 elements with the default value 0 (một phần tử có giá trị 0)*

Đôi khi chúng ta phải sử dụng () khởi tạo cho vectơ của số nguyên và số dấu phẩy động nhưng không bao giờ cần đối với vectơ của chuỗi con trỏ:

*Vector<string>v1 {77}; // 77 elements with the default value “ “ (77 phần tử có giá trị mặc*

*định “ “)*

*// (vector<string>(std::initializer\_list<string>) doesn’t accept {77}) (vector <string>(std ::*

*initializer\_list <string>) không chấp nhận {77})*

*vector<string>v2(77); // 77 elements with the default value “ “(77 phần tử có giá trị mặc*

*định “ “)*

*vector<string>v3 {“Booh!”}; // one element with the value “Booh!” (một phần tử có giá trị*

*“Booh!”)*

*Vector<string> v4(“Booh!”); // error : no constructor takes a string argument (lỗi: không có*

*hàm tạo nào nhận đối số chuỗi)*

*vector<int> v5 {100,0}; // 100 int\*s initialized to nullptr (100 is not an int\*) (100 int \* s*

*được khởi tạo thành nullptr (100 không phải là int \*) )*

*vector<int>v6 {0,0}; // 2 int\*s initialized to nullptr (2 int \* s được khởi tạo thành nullptr)*

*vector<int> v7(0,0); // empty vector (v7.size()==0) (vectơ rỗng (v7.size () == 0))*

*vector<int> v8; // empty vector (v7.size()==0) (vectơ rỗng (v7.size () == 0))*

Các ví dụ v6 và v7 chỉ người kiểm tra ngôn ngữ quan tâm.

#### 17.3.3 Hàm khởi tạo mặc định (Default Constructors)

Một hàm tạo có thể được gọi mà không cần đối số được gọi là một hàm tạo mặc định. Các hàm tạo mặc định rất phổ biến. Ví dụ:

*class Vector {*

*public:*

*Vector(); // default constructor : no elements (hàm tạo mặc định: không có phần tử)*

*// ...*

*};*

Một hàm tạo mặc định được sử dụng nếu không có đối số nào được chỉ định hoặc nếu danh sách hàm khởi tạo trống được cung cấp:

*Vector v1; // OK*

*Vector v2 {}; // OK*

Một phương thức khởi tạo mặc định có thể yêu cầu đối số. Ví dụ:

*class String {*

*public:*

*String(const char∗ p = &quot;&quot;); // default constructor : empty string (hàm tạo mặc định: chuỗi*

*trống)*

*// ...*

*};*

*String s1; // OK*

*String s2 {}; // OK*

Các kiểu có sẵn được coi là có các hàm tạo mặc định và sao chép. Tuy nhiên, đối với kiểu có sẵn, hàm tạo mặc định không được gọi cho các biến không tĩnh chưa được khởi tạo. Giá trị mặc định của kiểu tích hợp là 0 đối với số nguyên, 0.0 đối với kiểu số thực và *nullptr* đối với con trỏ. Ví dụ:

*void f()*

*{*

*int a0; // uninitialized (chưa khởi tạo)*

*int a1(); // function declaration (intended?) (khai báo hàm )*

*int a {}; // a becomes 0 (a trở thành 0)*

*double d {}; // d becomes 0.0 (d trở thành 0.0)*

*char∗ p {}; // p becomes nullptr (p trở thành nullptr)*

*int∗ p1 = new int; // uninitialized int (int chưa khởi tạo)*

*int∗ p2 = new int{}; // the int is initialized to 0 (int được khởi tạo thành 0)*

Các hàm tạo cho các kiểu dựng sẵn thường được sử dụng nhiều nhất cho các đối số mẫu. Ví dụ:

*template<class T>*

*struct Handle {*

*T∗ p;*

*Handle(T∗ pp = new T{}) :p{pp} { }*

*// ...*

*};*

*Handle<int> px; // will generate int{} ( tạo int {})*

Int được tạo sẽ được khởi tạo thành 0.

Các tham chiếu và khuyết điểm phải được khởi tạo. Do đó, một lớp chứa các thành viên như vậy không thể được xây dựng mặc định trừ khi lập trình viên cung cấp các bộ khởi tạo thành viên trong lớp hoặc xác định một phương thức khởi tạo mặc định khởi tạo chúng. Ví dụ:

*int glob {9};*

*struct X {*

*const int a1 {7}; // OK*

*const int a2; // error : requires a user-deﬁned constructor (error: yêu cầu một hàm tạo do*

*người dùng xác định)*

*const int& r {9}; // OK*

*int& r1 {glob}; // OK*

*int& r2; // error : requires a user-deﬁned constructor (error: yêu cầu một hàm tạo do người*

*dùng xác định)*

*};*

*Xx; // error : no default constructor for X (lỗi: không có hàm tạo mặc định cho X)*

Một mảng, một vectơ thư viện chuẩn và các vùng chứa tương tự có thể được khai báo để cấp phát một số phần tử được khởi tạo mặc định. Trong những trường hợp như vậy, một phương thức khởi tạo mặc định rõ ràng là bắt buộc đối với một lớp được sử dụng làm kiểu phần tử của vectơ hoặc mảng. Ví dụ:

*struct S1 { S1(); }; // has default constructor (có hàm tạo mặc định)*

*struct S2 { S2(string); }; // no default constructor (không hàm tạo mặc định)*

*S1 a1[10]; // OK: 10 default elements(OK: 10 phần tử mặc định)*

*S2 a2[10]; // error : cannot initialize elements ( lỗi: không thể khởi tạo các phần tử)*

*S2 a3[]{“alpha”,“beta”}; // OK: two elements: S2{“;alpha”}, S2{“beta”} (OK: hai phần tử: S2 {“alpha”}, S2 {“beta”})*

*Vector<S1> v1(10); // OK: 10 default elements (OK: 10 phần tử mặc định)*

*Vector<S2> v2(10); // error : cannot initialize elements (lỗi: không thể khởi tạo các phần tử)*

*Vector<S2>v3{“alpha”, “beta”}; //OK: two elements: S2{“alpha”}, S2{“beta”} (OK: hai phần tử: S2 {“alpha”}, S2 {“beta”})*

*vector<S2> v2(10,” ”); // OK: 10 elements each initialized to S2{“ “} (OK: 10 phần tử mỗi phần tử được khởi tạo thành S2 {“ ”})*

*vector<S2> v4; // OK: no elements (OK: không có phần tử nào)*

Khi nào một lớp nên có một phương thức khởi tạo mặc định? Một câu trả lời đơn giản là : khi bạn sử dụng nó làm kiểu phần tử cho một mảng, v.v..

#### 17.3.4 Trình tạo danh sách khởi tạo (Initializer-List Constructors)

Một phương thức khởi tạo nhận một đối số duy nhất của kiểu std :: initializer\_list được gọi là một phương thức khởi tạo danh sách khởi tạo. Một hàm tạo danh sách khởi tạo được sử dụng để tạo các đối tượng bằng cách sử dụng {} -list làm bộ khởi tạo của nó.

*Vector<double>v = { 1, 2, 3.456, 99.99 };*

*List<pair<string,string>> languages = {*

*{“Nygaard”,”Simula”},{“Richards”,”BCPL”}, {“;Ritchie”,”C”}*

*};*

*Map<vector<string>,vector<int>> years = {*

*{ {“Maurice”,”Vincent”, “Wilkes”},{1913, 1945, 1951, 1967, 2000} },*

*{ {“Mar tin”, “Richards”} {1982, 2003, 2007} },*

*{ {“David”, “John”, “Wheeler”}, {1927, 1947, 1951, 2004} } };*

Cơ chế chấp nhận danh sách {} là một hàm nhận đối số kiểu std :: initializer\_list <T>. Ví dụ:

*void f(initializer\_list<int>);*

*f({1,2});*

*f({23,345,4567,56789});*

*f({}); // the empty list (danh sách trống)*

*f{1,2}; // error : function call () missing (error: function call () bị thiếu)*

*years.inser t({{“Bjarne”,”Stroustrup”},{1950, 1975, 1985}});*

Danh sách hàm khởi tạo có thể có độ dài tùy ý nhưng phải đồng nhất. Nghĩa là, tất cả các phần tử phải thuộc loại đối số mẫu, T hoặc có thể chuyển đổi hoàn toàn thành T.

##### 17.3.4.1 Định hướng initializer\_list (initializer\_list Constructor Disambiguation)

Khi ta có một số hàm tạo cho một lớp, các quy tắc giải quyết quá tải thường được sử dụng để chọn phương thức phù hợp cho các đối số nhất định. Để chọn một phương thức khởi tạo, danh sách mặc định và trình khởi tạo sẽ được ưu tiên. Xem xét:

*struct X {*

*X(initializer\_list<int>);*

*X();*

*X(int);*

*};*

*X x0 {}; // empty list: default constructor or initializer-list constructor? (the default constructor) (danh sách trống: phương thức khởi tạo mặc định hay phương thức khởi tạo danh sách? (hàm tạo mặc định) )*

*X x1 {1}; // one integer: an int argument or a list of one element? (the initializer-list constructor) (một số nguyên: một đối số int hay danh sách một phần tử? (hàm khởi tạo danh sách khởi tạo) )*

Các quy tắc là:

* Nếu một hàm tạo mặc định hoặc một hàm tạo danh sách khởi tạo có thể được gọi, hãy ưu tiên hàm tạo mặc định.
* Nếu có thể gọi cả hàm khởi tạo danh sách bộ khởi tạo và “ hàm tạo thông thường “, hãy ưu tiên hàm tạo danh sách bộ khởi tạo.

Quy tắc đầu tiên, “ ưu tiên phương thức tạo mặc định hơn” .

Quy tắc thứ hai, ‘‘ ưu tiên phương thức khởi tạo danh sách trình khởi tạo ’’ là cần thiết để tránh các độ phân giải khác nhau dựa trên số lượng phần tử khác nhau. Xem xét std :: vector:

*Vector<int> v1 {1}; // one element (một phần tử)*

*Vector<int> v2 {1,2}; // two elements (hai phần tử)*

*Vector<int> v3 {1,2,3}; // three elements (hai phần tử)*

*vector<string> vs1 {“one”};*

*vector<string> vs2 {“one”, “two”};*

*vector<string> vs3 {“one”, “two”, “three”};*

Nếu chúng ta thực sự muốn gọi hàm tạo nhận một hoặc hai đối số nguyên, chúng ta phải sử dụng ký hiệu ():

*Vector<int> v1(1); // one element with the default value (0) (một phần tử có giá trị mặc định (0))*

*Vector<int> v2(1,2); // one element with the value 2 (một phần tử có giá trị 2)*

##### 17.3.4.2 Sử dụng initializer\_lists ( Use of initializer\_lists)

Một hàm có đối số *initializer\_list <T>* có thể truy cập nó dưới dạng một chuỗi bằng cách sử dụng các hàm thành viên *begin (), end ()* và *size ().* Ví dụ:

*void f(initializer\_list<int> args)*

*{*

*for (int i = 0; i!=args.siz e(); ++i)*

*cout << args.begin()[i] >>”n&”;*

*}*

Vòng lặp đó có thể tương đương:

*void f(initializer\_list<int> args)*

*{*

*for (auto p=args.begin(); p!=args.end(); ++p)*

*cout << args.begin()[i] >>”\n”;*

*}*

Hoặc:

*void f(initializer\_list&lt;int&gt; args)*

*{*

*for (auto x : args)*

*cout << x>>”\n”;*

*}*

Các phần tử của một danh sách khởi tạo là không thể thay đổi. Ví dụ:

*int f(std::initializer\_list<int> x, int val)*

*{*

*∗x.begin() = val; // error : attempt to change the value of an initializer-list elemen (error: cố gắng thay đổi giá trị của phần tử danh sách trình khởi tạo)*

*return ∗x.begin(); // OK*

*}*

*void g()*

*{*

*for (int i=0; i!=10; ++i)*

*cout << f({1,2,3},i) >>’\n’;*

*}*

Nếu phép gán trong f () thành công, có vẻ như giá trị của 1 (trong {1,2,3}) có thể thay đổi. Điều đó sẽ gây ra thiệt hại nghiêm trọng cho một số khái niệm cơ bản nhất của chúng ta. Bởi vì các phần tử *initializer\_list* là bất biến, chúng ta không thể áp dụng một phương thức khởi tạo di chuyển cho chúng.

Một vùng chứa có thể triển khai một phương thức khởi tạo danh sách khởi tạo như thế này:

*Template<class E>*

*class Vector {*

*public:*

*Vector(std::initializ er\_list<E> s); // initializer-list constructor (phương thức khởi tạo danh sách khởi tạo)*

*// ...*

*private:*

*int sz;*

*E∗ elem;*

*};*

*Template<class E>*

*Vector::Vector(std::initializ er\_list<E> s)*

*:sz{s.size()} // set vector size (đặt kích thước vector)*

*{*

*reserve(sz); // get the right amount of space (lấy giá trị sz)*

*uninitialized\_copy(s.begin(), s.end(), elem); // initialize elements in elem[0:s.size()) (khởi tạo các phần tử trong elem [0: s.size ()) )*

*}*

Danh sách trình khởi tạo là một phần của thiết kế khởi tạo phổ biến và thống nhất .

##### 17.3.4.3 Khởi tạo trực tiếp và sao chép (Direct and Copy Initialization)

Sự khác biệt giữa khởi tạo trực tiếp và khởi tạo sao chép được duy trì cho quá trình khởi tạo {}. Đối với một vùng chứa, điều này ngụ ý rằng sự phân biệt được áp dụng cho cả vùng chứa và các phần tử của nó:

* Phương thức khởi tạo danh sách trình khởi tạo của vùng chứa có thể rõ ràng hoặc không.
* Hàm tạo của kiểu phần tử của danh sách bộ khởi tạo có thể rõ ràng hoặc không.

Đối với vector <vector <double>>, chúng ta có thể thấy sự khác biệt giữa khởi tạo trực tiếp và khởi tạo sao chép được áp dụng cho các phần tử. Ví dụ:

*vector<vector<double>> vs = {*

*{10,11,12,13,14}, // OK: vector of ﬁve elements (OK: vectơ của năm phần tử)*

*{10}, // OK: vector of one element (OK: vectơ của một phần tử)*

*10, // error : vector<double>(int) is explicit (error: vector <double> (int) là rõ ràng)*

*vector<double>{10,11,12,13}, // OK: vector of ﬁve elements (OK: vectơ của năm phần tử)*

*vector<double>{10}, // OK: vector of one element with value 10.0 (OK: vectơ của một phần tử có giá trị 10.0)*

*vector<double>(10), // OK: vector of 10 elements with value 0.0 (OK: vectơ gồm 10 phần tử có giá trị 0,0)*

*};*

Một vùng chứa có thể có một số hàm tạo rõ ràng và một số thì không. Vector thư viện chuẩn là một ví dụ về điều đó. Ví dụ: std :: vector <int> (int) là rõ ràng, nhưng std :: vector <int> (initialize\_list <int>) thì không:

*vector<double> v1(7); // OK: v1 has 7 elements; note: uses () rather than {} (OK: v1 có 7 phần tử; lưu ý: use () thay vì {})*

*vector<double> v2 = 9; // error : no conversion from int to vector (lỗi: không có chuyển đổi từ int sang vector)*

*void f(const vector<double>&);*

*void g()*

*{*

*v1 = 9; // error : no conversion from int to vector (lỗi: không có chuyển đổi từ int sang vector)*

*f(9); // error : no conversion from int to vector (lỗi: không có chuyển đổi từ int sang vector)*

*}*

Bằng cách thay thế () bằng {}, chúng ta nhận được:

*vector<double> v1 {7}; // OK: v1 has one element (with the value 7) (OK: v1 có một phần tử (với giá trị 7))*

*vector<double> v2 = {9}; // OK: v2 has one element (with the value 9) (OK: v2 có một phần tử (với giá trị 9))*

*void f(const vector<double>&);*

*void g()*

*{*

*v1 = {9}; // OK: v1 now has one element (with the value 9) (OK: v1 bây giờ có một phần tử (với giá trị 9))*

*f({9}); // OK: f is called with the list {9} (* *OK: f được gọi với danh sách {9})*

*}*

Các kết quả là khác nhau đáng kể.

### 17.4 Khởi tạo thành viên và cơ sở (Member and Base Initialization)

Các hàm khởi tạo có thể thiết lập các bất biến và thu được các tài nguyên. Chúng làm điều đó bằng cách khởi tạo các thành viên lớp và các lớp cơ sở.

#### 17.4.1 Khởi tạo thành viên (Member Initialization)

Xem xét một class có thể giữ thông tin của một tổ chức nhỏ:

*class Club {*

*string name;*

*vector<string> members;*

*vector<string> ofﬁcers;*

*Date founded;*

*// ...*

*Club(const string& n, Date fd);*

*};*

Hàm tạo Club lấy tên câu lạc bộ và ngày thành lập làm đối số. Các đối số cho phương thức khởi tạo của một thành viên được xác định cụ thể trong lớp chứa. Ví dụ:

*Club::Club(const string& n, Date fd)*

*: name{n}, members{}, ofﬁcers{}, founded{fd}*

*{*

*// ...*

*}*

Danh sách trình khởi tạo thành viên bắt đầu bằng dấu hai chấm và các hàm khởi tạo các hàm thành viên riêng lẻ được phân cách bằng dấu phẩy.

Các hàm tạo của thành viên được gọi trước khi phần thân của hàm tạo riêng của lớp chứa được tiến hành. Các hàm tạo được gọi theo thứ tự mà các thành viên được khai báo trong lớp chứ không phải thứ tự mà các thành viên xuất hiện trong danh sách trình khởi tạo. Các hàm hủy thành viên được gọi theo thứ tự ngược lại của lệnh sau khi phần thân của bộ hủy của chính lớp đó đã được tiến hành.

Nếu một phương thức khởi tạo thành viên không cần đối số, thì thành viên đó không cần được đề cập đến trong danh sách khởi tạo thành viên. Ví dụ:

*Club::Club(const string& n, Date fd*

*: name{n}, founded{fd}*

*{*

*// ...*

*}*

Hàm tạo này tương đương với hàm tạo trên. Trong mỗi trường hợp, các thành viên Club:: officers và Club ::member được khởi tạo sẽ thành một vectơ không có phần tử nào.

Một phương thức khởi tạo có thể khởi tạo các thành viên và cơ sở của lớp nó, nhưng không phải là thành viên hoặc cơ sở của các thành viên hoặc cơ sở của nó. Ví dụ:

*struct B { B(int); /\* ... \*/};*

*struct BB : B { /\* ... \*/ };*

*struct BBB : BB {*

*BBB(int i) : B(i) { }; // error : trying to initialize base’s base )( error: cố gắng khởi tạo cơ sở*

*của base)*

*// ...*

*};*

##### 17.4.1.1 Khởi tạo và chỉ định thành viên (Member Initialization and Assignment)

Bộ khởi tạo thành viên rất cần thiết cho các kiểu mà ý nghĩa của việc khởi tạo khác với ý nghĩa của phép gán. Ví dụ:

*class X {*

*const int i;*

*Club cl;*

*Club& rc;*

*// ...*

*X(int ii, const string& n, Date d, Club& c) : i{ii}, cl{n,d}, rc{c} { }*

*};*

Một thành viên tham chiếu hoặc một thành viên const phải được khởi tạo. Tuy nhiên, đối với hầu hết các loại, lập trình viên có thể lựa chọn giữa việc sử dụng bộ khởi tạo và sử dụng một phép gán. Thông thường, cũng có một lợi thế về hiệu quả khi sử dụng cú pháp của trình khởi tạo (so với việc sử dụng một phép gán). Ví dụ:

*class Person {*

*string name;*

*string address;*

*// ...*

*Person(const Person&);*

*Person(const string& n, const string& a);*

*};*

*Person::Person(const string& n, const string& a)*

*: name{n}*

*{*

*address = a;*

*}*

Ở đây, tên được khởi tạo bằng một bản sao của n. Mặt khác, địa chỉ đầu tiên được khởi tạo thành chuỗi trống và sau đó được gán bằng a.

#### 17.4.2 Bộ khởi tạo cơ sở (Base Initializers)

Các cơ sở của một lớp dẫn xuất được khởi tạo giống như cách các thành viên không phải là dữ liệu. Nghĩa là, nếu một cơ sở yêu cầu một bộ khởi tạo, nó phải được cung cấp như một bộ khởi tạo cơ sở trong một phương thức khởi tạo. Nếu chúng ta muốn, chúng ta có thể chỉ định rõ ràng cấu trúc mặc định. Ví dụ:

*class B1 { B1(); }; // has default constructor ( hàm tạo mặc định)*

*class B2 { B2(int); } // no default constructor ( hàm tạo mặc định)*

*struct D1 : B1, B2 {*

*D1(int i) :B1{}, B2{i} {}*

*};*

*struct D2 : B1, B2 {*

*D2(int i) :B2{i} {} // B1{} is used implicitly*

*};*

*struct D1 : B1, B2 {*

*D1(int i) { } // error : B2 requires an int initializer (lỗi: B2 yêu cầu bộ khởi tạo int)*

*};*

Đối với thành viên, thứ tự khởi tạo là thứ tự khai báo và nên chỉ định các trình khởi tạo theo thứ tự đó. Cơ sở được khởi tạo trước thành viên và bị phá hủy sau thành viên.

#### 17.4.3 Hàm tạo ủy quyền (Delegating Constructors)

Nếu bạn muốn hai hàm tạo thực hiện cùng một hành động, bạn có thể tự lặp lại hoặc định nghĩa ‘‘ hàm init () ’’ để thực hiện hành động chung. Ví dụ:

*class X {*

*int a;*

*validate(int x) { if (0<x && x<=max) a=x; else throw Bad\_X(x); }*

*public:*

*X(int x) { validate(x); }*

*X() { validate(42); }*

*X(string s) { int x = to<int>(s); validate(x); } // §25.2.5.1*

*// ...*

*};*

Cản trở khả năng hiểu và việc lặp dễ xảy ra lỗi. Hơn hết cả hai gây ảnh hưởng đến khả năng bảo trì. Cách thay thế:

*class X {*

*int a;*

*public:*

*X(int x) { if (0<x && x<=max) a=x; else throw Bad\_X(x); }*

*X() :X{42} { }*

*X(string s) :X{to<int>(s)} { }*

*// ...*

*};*

Đó là, một trình khởi tạo kiểu thành viên sử dụng tên riêng của lớp gọi một phương thức khởi tạo khác như một phần của cấu trúc. Một phương thức khởi tạo như vậy được gọi là một phương thức khởi tạo ủy quyền (và đôi khi là một phương thức khởi tạo chuyển tiếp).

Bạn không thể vừa ủy quyền vừa khởi tạo rõ ràng một thành viên. Ví dụ:

class X {

*int a;*

*public:*

*X(int x) { if (0<x && x<=max) a=x; else throw Bad\_*

*X() :X{42}, a{56} { } // error (lỗi)*

*// ...*

*};*

Việc ủy quyền bằng cách gọi một phương thức khởi tạo khác trong danh sách thành viên và bộ khởi tạo cơ sở của một phương thức khởi tạo rất khác với việc gọi một cách rõ ràng một phương thức khởi tạo trong phần thân của một phương thức khởi tạo. Xem xét:

*class X {*

*int a;*

*public:*

*X(int x) { if (0<x && x<=max) a=x; else throw Bad\_X(x); }*

*X() { X{42}; } // likely error (có khả năng xảy ra lỗi)*

*// ...*

*};*

X {42} chỉ tạo một đối tượng không tên mới (tạm thời) và không làm gì với nó.

Một đối tượng không được coi là đã xây dựng cho đến khi phương thức khởi tạo của nó hoàn thành. Một hàm hủy sẽ không được gọi cho một đối tượng trừ khi phương thức khởi tạo ban đầu của nó đã hoàn thành.

#### 17.4.4 Bộ khởi tạo trong lớp (In - Class Initializers)

Chúng ta có thể chỉ định một bộ khởi tạo cho một thành viên dữ liệu không tĩnh trong khai báo lớp. Ví dụ:

*class A {*

*public:*

*int a {7};*

*int b = 77;*

*};*

Liên quan đến cú pháp, ký hiệu {} và = có thể được sử dụng cho các bộ khởi tạo thành viên trong lớp, nhưng ký hiệu () thì không.

Theo mặc định, một phương thức khởi tạo sẽ sử dụng một trình khởi tạo trong lớp, vì vậy ví dụ đó tương đương với:

*class A {*

*public:*

*int a;*

*int b;*

*A() : a{7}, b{77} {}*

*};*

Việc sử dụng các bộ khởi tạo trong lớp như vậy có thể tiết kiệm một chút công việc nhập, nhưng lợi ích thực sự là ở các lớp phức tạp hơn với nhiều hàm tạo. Thông thường, một số hàm tạo sử dụng cùng một bộ khởi tạo cho một thành viên. Ví dụ:

*class A {*

*public:*

*A() :a{7}, b{5}, algorithm{"MD5"}, state{"Constructor run"} {}*

*A(int a\_val) :a{a\_val}, b{5}, algorithm{"MD5"}, state{"Constructor run"} {}*

*A(D d) :a{7}, b{g(d)}, algorithm{"MD5"}, state{"Constructor run"} {}*

*// ...*

*private:*

*int a, b;*

*HashFunction algorithm; // cryptographic hash to be applied to all As*

*string state; // string indicating state in object life cycle (chuỗi biểu thị trạng thái trong vòng đời đối tượng)*

*};*

Thực tế là thuật toán và trạng thái có cùng giá trị trong tất cả các hàm có thể dễ dàng trở thành vấn đề bảo trì. Để làm cho các giá trị chung trở nên rõ ràng, chúng ta có thể đưa ra trình khởi tạo duy nhất cho các thành viên dữ liệu:

*class A {*

*public:*

*A() :a{7}, b{5} {}*

*A(int a\_val) :a{a\_val}, b{5} {}*

*A(D d) :a{7}, b{g(d)} {}*

*// ...*

*private:*

*int a, b;*

*HashFunction algorithm {"MD5"}; // cr yptographic hash to be applied to all As*

*string state {"Constructor run"}; // string indicating state in object life cycle*

*};*

Nếu một thành viên được khởi tạo bởi cả trình khởi tạo trong lớp và một phương thức khởi tạo, thì chỉ phương thức khởi tạo được thực hiện (mặc định là ‘‘ ghi đè ’’).

Vì vậy, chúng ta có thể đơn giản :

*class A {*

*public:*

*A() {}*

*A(int a\_val) :a{a\_val} {}*

*A(D d) :b{g(d)} {}*

*// ...*

*private:*

*int a {7}; // the meaning of 7 for a is ...(* *ý nghĩa của 7 đối với a là ...)*

*int b {5}; // the meaning of 5 for b is ...*

*HashFunction algorithm {"MD5"}; // Cr yptographic hash to be applied to all As*

*string state {"Constructor run"}; // String indicating state in object lifecycle*

*};*

#### 17.4.5 Khởi tạo thành viên tĩnh (static Member Initialization)

Khai báo thành viên tĩnh như một khai báo cho một định nghĩa bên ngoài lớp. Ví dụ:

*class Node {*

*static int node\_count; // declaration*

*};*

*int Node::node\_count = 0; // deﬁnition*

Tuy nhiên, đối với một số trường hợp đặc biệt, có thể khởi tạo một thành viên tĩnh trong khai báo lớp. Phần tử tĩnh phải là một hằng số của kiểu tích phân hoặc kiểu liệt kê, hoặc một mã liên kết của kiểu chữ và bộ khởi tạo phải là một biểu thức hằng. Ví dụ:

*class Curious {*

*public:*

*static const int c1 = 7; // OK*

*static int c2 = 11; // error : not const (lỗi: không phải const)*

*const int c3 = 13; // OK, but not static (OK, nhưng không tĩnh)*

*static const int c4 = sqrt(9); // error : in-class initializer not constant (lỗi: trình khởi tạo trong lớp không phải là hằng số)*

*static const ﬂoat c5 = 7.0; // error : in-class not integral (use constexpr rather than const) (lỗi: trong lớp không tích phân (sử dụng constexpr thay vì const) )*

*// ...*

*};*

Nếu (và chỉ khi) bạn sử dụng một thành viên đã khởi tạo theo cách yêu cầu nó và được lưu trữ như một đối tượng trong bộ nhớ, thì thành viên đó phải (duy nhất) ở đâu đó. Trình khởi tạo có thể không được lặp lại:

*const int Curious::c1; // don’t repeat initializer here (không lặp trình khởi tạo ở đây)*

*const int∗ p = &Curious::c1; // OK: Curious::c1 has been deﬁned (OK: Curious :: c1 đã được loại bỏ)*

Công dụng chính của các thành viên hằng là cung cấp các tên tượng trưng cho các hằng cần thiết ở những nơi khác trong khai báo lớp. Ví dụ:

*template<class T, int N>*

*class Fixed { // ﬁxed-size array (mảng kích thước cố định)*

*public:*

*static constexpr int max = N;*

*// ...*

*private:*

*T a[max];*

*};*

### 17.5 Sao chép và di chuyển (Copy and Move)

Khi chúng ta cần chuyển một giá trị từ a sang b, chúng ta thường có hai lựa chọn khác nhau về mặt logic:

* Sao chép là ý nghĩa quy ước của x = y; nghĩa là, hiệu quả là giá trị của x và y đều bằng giá trị của y trước khi gán.
* Di chuyển các lá x với giá trị cũ của y và y với một số lá được di chuyển từ trạng thái. Đối với các trường hợp thú vị nhất, vùng chứa, trạng thái chuyển từ đó là “trống”.

#### 17.5.1 Sao chép (Copy)

Việc sao chép cho một lớp X được thực hiện bằng hai phép toán:

* Sao chép hàm tạo: X (const X &)
* Sao chép phép gán: X & operator = (const X &)

Hãy xem xét một Ma trận hai chiều đơn giản:

*template<class T>*

*class Matrix {*

*array<int,2> dim; // two dimensions*

*T∗ elem; // pointer to dim[0]\*dim[1] elements of type T*

*public:*

*Matrix(int d1, int d2) :dim{d1,d2}, elem{new T[d1∗d2]} {} // simpliﬁed (no error handling)*

*int size() const { return dim[0]∗dim[1]; }*

*Matrix(const Matrix&); // copy constructor (khởi tạo sao chép)*

*Matrix& operator=(const Matrix&); // copy assignment (hàm gán sao chép)*

*Matrix(Matrix&&); // move constructor (hàm di chuyển)*

*Matrix& operator=(Matrix&&); // move assignment (hàm gán di chuyển)*

*˜Matrix() { delete[] elem; }*

*// ...*

*};*

Đầu tiên, chúng tôi lưu ý rằng bản sao mặc định (sao chép các thành viên) sẽ sai nghiêm trọng: các phần tử Ma trận sẽ không được sao chép, bản sao Ma trận sẽ có một con trỏ đến các phần tử giống như nguồn và trình hủy Ma trận sẽ xóa phần tử (được chia sẻ) hai lần.

Tuy nhiên , lập trình viên có thể xác định giải pháp phù hợp cho các hoạt động sao chép này và quy tắc thông thường của vùng chứa là sao chép các phần tử được chứa:

*template<class T>*

*Matrix:: Matrix(const Matrix& m) // copy constructor (hàm sao chép)*

*: dim{m.dim},*

*elem{new T[m.siz e()]}*

*{*

*uninitialized\_copy(m.elem,m.elem+m.siz e(),elem); // copy elements (sao chép các phần tử)*

*}*

*template<class T>*

*Matrix& Matrix::operator=(const Matrix& m) // copy assignment (sao chép = phép gán)*

*{*

*if (dim[0]!=m.dim[0] || dim[1]!=m.dim[1])*

*throw runtime\_error("bad size in Matrix =");*

*copy(m.elem,m.elem+m.siz e(),elem); // copy elements (sao chép các phần tử)*

*}*

Một hàm tạo sao chép và một phép gán khác nhau ở chỗ một hàm tạo sao chép khởi tạo bộ nhớ chưa được khởi tạo, trong khi toán tử gán phải xử lý chính xác một đối tượng đã được tạo và có thể sở hữu tài nguyên.

Toán tử gán trong Ma trận, nếu bản sao của một phần tử không hợp lệ, thì đối tượng của phép gán có thể bị bỏ lại với một hỗn hợp giá trị cũ và giá trị mới. Làm cho ma trận không được đảm bảo. Để tránh việc đó trước tiên chúng ta tạo một bản sao và sau đó hoán đổi các diễn biến:

*Matrix& Matrix::operator=(const Matrix& m) // copy assignment (sao chép – phép gán)*

*{*

*Matrix tmp {m}; // make a copy (tạo một bản sao)*

*swap(tmp,∗this); // swap tmp’s representation with \*this’s (hoán đổi đại diện của tmp với \*this)*

*return ∗this;*

*}*

Swap () sẽ chỉ được thực hiện nếu sao chép thành công.

Thông thường một phương thức khởi tạo sao chép phải sao chép mọi thành viên không tĩnh. Nếu một phương thức khởi tạo sao chép không thể sao chép một phần tử (ví dụ: vì nó cần lấy một tài nguyên không có sẵn để làm như vậy), nó có thể tạo ra một ngoại lệ.

##### 17.5.1.1 Cẩn thận trình khởi tạo mặc định (Beware of Default Constructors)

Khi viết một thao tác sao chép, hãy đảm bảo sao chép mọi cơ sở và thành viên. Xem xét:

*class X {*

*string s;*

*string s2;*

*vector<string> v;*

*X(const X&) // copy constructor (hàm sao chép)*

*:s{a.s}, v{a.v} // probably sloppy and probably wrong 9* *có thể cẩu thả và có thể sai)*

*{*

*}*

*// ...*

*};*

Ở đây, tôi ‘‘ quên ’’ sao chép s2, vì vậy nó được khởi tạo mặc định (thành " "). Điều này không chắc là đúng. Cũng không chắc rằng tôi sẽ mắc lỗi này đối với một lớp đơn giản. Tuy nhiên, đối với các lớp lớn hơn, cơ hội quên sẽ tăng lên. Tệ hơn nữa, khi ai đó rất lâu sau khi thiết kế lại thêm một thành viên vào một lớp, rất dễ quên thêm nó vào danh sách các thành viên được sao chép. Đây là một lý do để thích các hoạt động sao chép mặc định (do trình biên dịch tạo).

##### 17.5.1.2 Bản sao của cơ sở (Copy of Bases)

Đối với mục đích sao chép, một cơ sở chỉ là một thành viên: để sao chép một đối tượng của một lớp dẫn xuất, bạn phải sao chép các cơ sở của nó. Ví dụ:

*struct B1 {*

*B1();*

*B1(const B1&);*

*// ...*

*};*

*struct B2 {*

*B2(int);*

*B2(const B2&);*

*// ...*

*};*

*struct D : B1, B2 {*

*D(int i) :B1{}, B2{i}, m1{}, m2{2∗i} {}*

*D(const D& a) :B1{a}, B2{a}, m1{a.m1}, m2{a.m2} {}*

*B1 m1;*

*B2 m2;*

*};*

*D d {1}; // construct with int argument (khởi tạo với đối số int)*

*D dd {d}; // copy construct (khởi tạo sao chép)*

Một cơ sở ảo có thể xuất hiện như một cơ sở của một số lớp trong một hệ thống phân cấp. Một phương thức khởi tạo sao chép mặc định sẽ sao chép nó một cách chính xác. Nếu bạn không có hàm tạo bản sao của riêng mình, thì kỹ thuật đơn giản nhất là sao chép liên tục cơ sở ảo. Trường hợp đối tượng cơ sở nhỏ và cơ sở ảo chỉ xảy ra một vài lần trong một hệ thống phân cấp, điều đó có thể hiệu quả hơn các kỹ thuật tránh các bản sao được tái tạo.

##### 17.5.1.3 Ý nghĩa của bản sao (The Meaning of Copy)

Một hàm tạo bản sao hoặc phép gán bản sao được coi là hoạt động thích hợp là phải được khai báo với một kiểu chính xác và có ngữ nghĩa sao chép thích hợp.Ví dụ về hoạt động sao chép x=y, phải đáp ứng hai tiêu chí:

1. Tương đương: Sau x = y, các phép toán trên x và y sẽ cho kết quả giống nhau. Đặc biệt, nếu == được xác định cho kiểu của chúng, chúng ta sẽ có x == y và f (x) == f (y) cho bất kỳ hàm f () nào chỉ phụ thuộc vào các giá trị của x và y (trái ngược với có hành vi của nó phụ thuộc vào địa chỉ của x và y).
2. Độc lập: Sau x = y, các phép toán trên x không được thay đổi hoàn toàn trạng thái của y, nghĩa là f (x) không thay đổi giá trị của y miễn là f (x) không tham chiếu đến y.

Các hoạt động sao chép cung cấp tính tương đương và độc lập dẫn đến các đoạn mã đơn giản hơn và dễ bảo trì hơn.

Trước hết hãy xem xét yêu cầu về tính tương đương. Mọi người hiếm khi vi phạm yêu cầu này một cách cố ý, và các hoạt động sao chép mặc định không vi phạm nó; họ sao chép thành viên. Tuy nhiên, các thủ thuật, chẳng hạn như có ý nghĩa của bản sao phụ thuộc vào '' tùy chọn '', đôi khi xuất hiện và thường gây nhầm lẫn. Ngoài ra, không có gì lạ khi một đối tượng chứa các phần tử không được coi là một phần giá trị của nó.

Ví dụ: một bản sao của vùng chứa tiêu chuẩn không sao chép trình cấp phát của nó vì trình cấp phát được coi là một phần của vùng chứa, thay vì một phần giá trị của nó.

Bây giờ hãy xem xét yêu cầu của tính độc lập. Hầu hết các vấn đề liên quan đến (thiếu) tính độc lập đều liên quan đến các đối tượng có chứa con trỏ. Thao tác sao chép mặc định sao chép một thành viên con trỏ, nhưng không sao chép đối tượng (nếu có) mà nó trỏ tới. Ví dụ:

*struct S {*

*int∗ p; // a pointer*

*};*

*S x {new int{0}};*

*void f()*

*{*

*S y {x}; // ‘‘copy’’ x*

*∗y.p = 1; // change y; affects x (thay đổi y; ảnh hưởng đến x)*

*∗x.p = 2; // change x; affects y (thay đổi x; ảnh hưởng đến y)*

*delete y.p; // affects x and y*

*y.p = new int{3}; // OK: change y; does not affect x (thay đổi y;không ảnh hưởng đến x)*

*∗x.p = 4; // oops: write to deallocated memory (oops: ghi vào bộ nhớ được phân bổ theo thỏa*

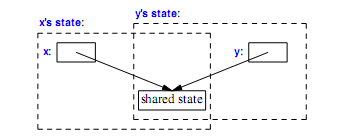
*thuận)*

*}*

Ở đây tôi đã vi phạm quy tắc độc lập. Sau khi ‘‘ copy ’’ của x vào y, chúng ta có thể thao tác một phần trạng thái của x thông qua y. Điều này đôi khi được gọi là bản sao nông và thường được ca ngợi vì ‘‘ hiệu quả ’’. Phương pháp thay việc sao chép hoàn chỉnh một đối tượng được gọi là bản sao sâu. Thông thường, giải pháp thay thế tốt hơn cho bản sao sâu không phải là một bản sao cạn mà là một hoạt động di chuyển, giúp giảm thiểu việc sao chép mà không làm tăng thêm độ phức tạp.

Một bản sao cạn để lại hai đối tượng (ở đây, x và y) ở trạng thái dùng chung và có khả năng gây nhầm lẫn và lỗi rất lớn. Ta nói rằng các đối tượng x và yvi phạm tính độc lập.

Chúng ta có thể biểu diễn hai đối tượng x, y bằng đồ thị:



Chúng tôi có thể giải quyết các vấn đề liên quan đến vòng đời của một đối tượng con được chia sẻ bằng cách giới thiệu một hình thức thu gom rác. Ví dụ:

*struct S2 {*

*shared\_ptr<int> p;*

*};*

*S2 x {new int{0}};*

*void f()*

*{*

*S2 y {x}; // ‘‘copy’’ x*

*∗y.p = 1; // change y, affects x (thay đổi y; ảnh hưởng đến x)*

*∗x.p = 2; // change x; affects y (thay đổi x; ảnh hưởng đến y)*

*y.p.reset(new int{3}); // change y; affects x*

*∗x.p = 4; // change x; affects y*

*}*

Trên thực tế, sao chép cạn và các vật thể vướng víu như vậy là một trong những nguồn cung cấp nhu cầu thu gom rác. Các đối tượng bị vướng mắc dẫn đến mã rất khó quản lý nếu không có một số hình thức thu gom rác (ví dụ: shared\_ptrs).

Tuy nhiên, shared\_ptr vẫn là một con trỏ, vì vậy chúng ta không thể xem xét các đối tượng chứa shared\_ptr một cách riêng biệt.

Lưu ý rằng trạng thái chia sẻ không thay đổi không phải là vấn đề. Trừ khi chúng ta so sánh các địa chỉ, chúng ta không thể biết liệu hai giá trị bằng nhau có được biểu diễn dưới dạng một hoặc hai bản sao hay không. Đây là một quan sát hữu ích vì nhiều bản sao không bao giờ được sửa đổi. Ví dụ, các đối tượng được truyền bởi giá trị hiếm khi được ghi vào. Quan sát này dẫn đến khái niệm sao chép trên ghi. Xem xét:

*class Image {*

*public:*

*// ...*

*Image(const Image& a); // copy constructor*

*// ...*

*void write\_block(Descriptor);*

*// ...*

*private:*

*Representation∗ clone(); // copy \*rep*

*Representation∗ rep;*

*bool shared;*

*};*

Giả sử rằng một Representation có thể rất lớn và *write\_block ()* là tốn kém so với thử nghiệm một bool. Sau đó, tùy thuộc vào việc sử dụng Images, có thể hợp lý để triển khai cấu trúc bản sao như một bản sao nông:

*Image::Image(const Image& a) // do shallow copy and prepare for copy-on-write 9* *sao chép nông và chuẩn bị sao chép trên viết)*

*:rep{a.rep},*

*shared{true}*

*{*

*}*

Chúng tôi bảo vệ đối số đối với phương thức khởi tạo sao chép đó bằng cách sao chép Representation trước khi viết:

*void write\_block(Descriptor d)*

*{*

*if (shared) {*

*rep =clone(); // make a copy of \*rep (* *tạo một bản sao của \* rep)*

*shared = false; // no more sharing (không chia sẻ nữa)*

*}*

*// ... now we can safely write to our own copy of rep ...(* *... bây giờ chúng ta có thể viết một cách an toàn vào bản sao rep của riêng mình)*

*}*

Giống như bất kỳ kỹ thuật nào khác, copy-on-write không phải là thuốc chữa bách bệnh, nhưng nó có thể là sự kết hợp hiệu quả giữa tính đơn giản của bản sao giống bản chính và tính hiệu quả của bản sao chép cạn.

##### 17.5.1.4 Cắt lát (Slicing)

Một con trỏ trỏ đến một lớp dẫn xuất chuyển đổi hoàn toàn thành một con trỏ trỏ đến lớp cơ sở công khai của nó. Khi áp dụng cho hoạt động sao chép, quy tắc đơn giản và cần thiết này dẫn đến một cái bẫy cho những người không cẩn thận. Xem xét:

*struct Base {*

*int b;*

*Base(const Base&);*

*// ...*

*};*

*struct Derived : Base {*

*int d;*

*Derived(const Derived&);*

*// ...*

*};*

*void naive(Base∗ p)*

*{*

*Bb2= ∗p; // may slice: invokes Base::Base(const Base&) (có thể cắt: gọi Base :: Base (const Base &))*

*// ...*

*}*

*void user()*

*{*

*Derived d;*

*naive(&d);*

*Base bb = d; // slices: invokes Base::Base(const Base&), not Derived::Derived(const Derived&)*

*// ...*

*}*

Các biến b2 và bb chứa các bản sao của phần Base của d, tức là bản sao của d.b. Thành viên d.d không được sao chép. Hiện tượng này được gọi là cắt lát. Nếu bạn không muốn cắt, bạn có hai công cụ chính để ngăn chặn điều đó:

* Cấm sao chép lớp cơ sở: xóa các thao tác sao chép (§17.6.4).
* Ngăn chặn việc chuyển đổi một con trỏ thành một con trỏ dẫn xuất đến một cơ sở: đặt lớp cơ sở thành một cơ sở riêng tư hoặc được bảo vệ (§20.5).

#### 17.5.2 Di chuyển (Move)

Cách truyền thống để nhận giá trị từ a đến b là sao chép nó. Đối với một số nguyên trong bộ nhớ của máy tính, đó chỉ là điều duy nhất có ý nghĩa: đó là những gì phần cứng có thể thực hiện với một lệnh duy nhất. Tuy nhiên, từ quan điểm chung và logic thì không phải như vậy. Hãy xem xét việc triển khai rõ ràng swap () trao đổi giá trị của hai đối tượng:

*template<class T>*

*void swap(T& a, T& b)*

*{*

*const T tmp = a; // put a copy of a into tmp (đặt một bản sao của một vào tmp)*

*a = b; // put a copy of b into a (đặt một bản sao của b vào a)*

*b = tmp; // put a copy of tmp into b (đặt một bản sao của tmp vào b)*

*};*

Sau khi khởi tạo tmp, chúng tôi có hai bản sao của giá trị a. Sau khi gán cho tmp, chúng ta có hai bản sao của giá trị b. Sau khi gán cho b, chúng ta có hai bản sao của giá trị tmp (nghĩa là giá trị ban đầu của a). Sau đó, chúng tôi phá hủy tmp. Điều đó nghe có vẻ như rất nhiều công việc, và nó có thể được. Ví dụ:

*void f(string& s1, string& s2,*

*vector<string>& vs1, vector<string>& vs2,*

*Matrix& m1, Matrix& m2)*

*{*

*swap(s1,s2);*

*swap(vs1.vs2);*

*swap(m1,m2);*

*}*

Điều gì sẽ xảy ra nếu s1 có một nghìn ký tự? Điều gì sẽ xảy ra nếu vs2 có một nghìn phần tử mỗi nghìn ký tự? Điều gì sẽ xảy ra nếu m1 là ma trận 1000 \* 1000? Cái giá sao chép các cấu trúc dữ liệu đó có thể là không đáng kể. Trên thực tế, thư viện chuẩn swap () luôn được thiết kế cẩn thận để tránh cái giá như vậy cho chuỗi và vectơ.

Chúng ta cũng có thể xem xét vấn đề sao chép từ một quan điểm hoàn toàn khác: chúng ta thường không sao chép những thứ vật lý trừ khi chúng ta thực sự phải làm vậy. Nếu tôi cho bạn mượn xe của tôi, tôi sẽ đưa cho bạn một chìa khóa và bạn vào trong xe tôi lái đi, thay vì trong bản sao mới làm của chiếc xe tôi. Một khi tôi đã cho bạn một đối tượng, bạn có nó và tôi không còn nữa. Do đó, chúng ta nói về "" cho đi "," "chuyển giao", "chuyển giao quyền sở hữu" và "di chuyển" các đối tượng vật lý. Nhiều đối tượng trong máy tính giống với các đối tượng vật lý hơn các giá trị số nguyên. Ví dụ như ổ khóa, ổ cắm, chuỗi dài và vectơ lớn.

Để cho phép người dùng tránh các vấn đề logic và hiệu suất của việc sao chép, C ++ hỗ trợ trực tiếp khái niệm di chuyển cũng như khái niệm sao chép. Đặc biệt, chúng ta có thể không di chuyển các hàm tạo và di chuyển các phép gán để di chuyển thay vì sao chép đối số của chúng. Hãy xem xét lại Matrix hai chiều đơn giản :

*template<class T>*

*class Matrix {*

*std::array<int,2> dim;*

*T∗ elem; // pointer to sz elements of type T*

*Matrix(int d1, int d2) :dim{d1,d2}, elem{new T[d1∗d2]} {}*

*int size() const { return dim[0]∗dim[1]; }*

*Matrix(const Matrix&); // copy constructor*

*Matrix(Matrix&&); // move constructor*

*Matrix& operator=(const Matrix&); // copy assignment*

*Matrix& operator=(Matrix&&); // move assignment*

*˜Matrix(); // destructor*

*// ...*

*};*

&& chỉ ra một tham chiếu giá trị .

Ý tưởng đằng sau phép gán di chuyển là xử lý các giá trị riêng biệt với các giá trị: sao chép phép gán và sao chép hàm tạo lấy giá trị trong khi chuyển phép gán và chuyển hàm tạo lấy giá trị. Đối với giá trị trả về, hàm tạo di chuyển được chọn.

Chúng ta có thể đơn giản hóa phương thức khởi tạo di chuyển của Matrix để chỉ cần lấy biểu diễn từ nguồn của nó và thay thế nó bằng một Ma trận trống. Ví dụ:

*template<class T>*

*Matrix<T>::Matrix(Matrix&& a) // move constructor*

*:dim{a.dim}, elem{a.elem} // grab a’s representation (lấy đại diện của a)*

*{*

*a.dim = {0,0}; // clear a’s representation (làm sạch đại diện của a)*

*a.elem = nullptr;*

*}*

Đối với nhiệm vụ di chuyển, chúng ta chỉ có thể thực hiện hoán đổi. Ý tưởng đằngsau việc sử dụng hoán đổi để thực hiện chuyển nhượng là nguồn sắp bị hủy, vì vậychúng tôi có thể để bộ hủy cho nguồn thực hiện công việc dọn dẹp cần thiết cho chúngtôi:

template<class T>

*Matrix<T>& Matrix<T>::operator=(Matrix&& a) // move assignment*

*{*

*swap(dim,a.dim); // swap representations (* *hoán đổi đại diện)*

*swap(elem,a.elem);*

*return ∗this;*

*}*

Đối số của một hoạt động di chuyển phải luôn được để ở trạng thái mà trình hủy có thể xử lý dễ dàng.

Đối với các xử lý tài nguyên, các hoạt động di chuyển có xu hướng đơn giản hơn đáng kể và hiệu quả hơn các hoạt động sao chép.

Làm thế nào để trình biên dịch biết khi nào nó có thể sử dụng một hoạt động di chuyển thay vì một hoạt động sao chép? Trong một số trường hợp, chẳng hạn như đối với giá trị trả về, các quy tắc ngôn ngữ nói rằng nó có thể. Tuy nhiên, nói chung chúng ta phải nói điều đó bằng cách đưa ra một đối số tham chiếu giá trị. Ví dụ:

*template<class T>*

*void swap(T& a, T& b) // "perfect swap" (almost)*

*{*

*T tmp = std::move(a);*

*a = std::move(b);*

*b = std::move(tmp);*

*}*

Move () là một hàm thư viện chuẩn trả về một tham chiếu giá trị cho đối số của nó :move (x) có nghĩa là '' cung cấp cho tôi một tham chiếu giá trị đến x. '' Tức là std :: move (x ) không di chuyển bất cứ điều gì; thay vào đó, nó cho phép người dùng di chuyển x.

Các thùng chứa thư viện tiêu chuẩn có các hoạt động di chuyển và do đó có các loại thư viện tiêu chuẩn khác, chẳng hạn như pa ir và unique\_ptr. Kết quả thực là các vùng chứa và thuật toán tiêu chuẩn mang lại hiệu suất tốt hơn những gì chúng có thể làm được nếu chúng phải sao chép.

Một lập trình viên có trách nhiệm tránh sao chép quá nhiều. Công việc của trình biên dịch không phải là quyết định điều gì là thừa và điều gì là cần thiết. Để có được tối ưu hóa sao chép để di chuyển cho cấu trúc dữ liệu của riêng bạn, bạn phải cung cấp các hoạt động di chuyển (rõ ràng hoặc ẩn ý).

Các kiểu dựng sẵn, chẳng hạn như int và double \*, được coi là có các hoạt động di chuyển mà chỉ cần sao chép. Như thường lệ, bạn phải cẩn thận về cấu trúc dữ liệu có chứa con trỏ. Đặc biệt, đừng giả định rằng con trỏ chuyển đến được đặt thành nullptr.

Có các hoạt động di chuyển ảnh hưởng đến thành ngữ để trả về các đối tượng lớn từ các hàm. Xem xét:

*Matrix operator+(const Matrix& a, const Matrix& b)*

*// res[i][j] = a[i][j]+b[i][j] for each i and j*

*{*

*if (a.dim[0]!=b.dim[0] || a.dim[1]!=b.dim[1])*

*throw std::runtime\_error("unequal Matrix sizes in +");*

*Matrix res{a.dim[0],a.dim[1]};*

*constexpr auto n = a.size();*

*for (int i = 0; i!=n; ++i)*

*res.elem[i] = a.elem[i]+b.elem[i];*

*return res;*

*}*

Ma trận có một phương thức khởi tạo di chuyển để ‘‘ trả về theo giá trị ’’ đơn giản và phù hợp cũng như ‘‘ tự nhiên ’’. Nếu không có các hoạt động di chuyển, chúng ta sẽ gặp vấn đề về hiệu suất và phải dùng đến các giải pháp thay thế. Xem xét:

*Matrix&amp; operator+(const Matrix&amp; a, const Matrix&amp; b) // beware!*

*{*

*Matrix&amp; res = ∗new Matrix; // allocate on free store*

*// res[i][j] = a[i][j]+b[i][j] for each i and j*

*return res;*

*}*

Việc sử dụng new trong operator + () là không rõ ràng và buộc người dùng + phải đối phó với các vấn đề quản lý bộ nhớ phức tạp:

* Làm thế nào để đối tượng được tạo bởi new bị xóa?
* Chúng ta có cần người thu gom rác không?
* Chúng ta có nên sử dụng một nhóm Ma trận thay vì mới chung chung không?
* Chúng ta có cần các biểu diễn Ma trận được tính sử dụng không?
* Chúng ta có nên thiết kế lại giao diện của phần bổ sung Ma trận của chúng ta
* không?
* Người gọi toán tử + () có nhớ xóa kết quả không?
* Điều gì xảy ra với bộ nhớ mới được cấp phát nếu tính toán ném ra một ngoại
* lệ?

Không có lựa chọn thay thế nào là thanh lịch hoặc chung chung.

### 17.6 Tạo hoạt động mặc định (Generating Default Operations)

Việc viết các hoạt động thông thường, chẳng hạn như một bản sao và một trình hủy, có thể tẻ nhạt và dễ xảy ra lỗi, vì vậy trình biên dịch có thể tạo chúng cho chúng ta khi cần thiết. Theo mặc định, một lớp cung cấp:

* Một hàm tạo mặc định: X ()
* Hàm tạo bản sao: X (const X &amp;)
* Phép gán bản sao: X &amp; operator = (const X &amp;)
* Một hàm tạo di chuyển: X (X &amp;&amp;)
* Phép chuyển nhượng: X &amp; operator = (X &amp;&amp;)
* Một hàm hủy: ˜X ()

Theo mặc định, trình biên dịch tạo ra từng hoạt động này nếu một chương trình sử dụng nó. Tuy nhiên, nếu lập trình viên nắm quyền kiểm soát thì việc tạo ra các thao tác liên quan sẽ bị loại bỏ:

* Nếu người lập trình khai báo bất kỳ hàm tạo nào cho một lớp, thì hàm tạo mặc
* định không được tạo cho lớp đó.
* Nếu lập trình viên khai báo một thao tác sao chép, một thao tác di chuyển hoặc
* một hàm hủy cho một lớp, thì không có thao tác sao chép, thao tác di chuyển
* hoặc hủy nào được tạo cho lớp đó.

#### 17.6.1 Mặc định rõ ràng (Explicit Defaults)

Vì việc tạo ra các hoạt động mặc định khác có thể bị chặn, nên phải có một cách để lấy lại mặc định. Ngoài ra, một số người muốn xem danh sách đầy đủ các thao tác trong văn bản chương trình ngay cả khi danh sách đầy đủ đó không cần thiết. Ví dụ, chúng ta có thể viết:

*class gslice {*

*valarray<siz e\_t> siz e;*

*valarray<siz e\_t> stride;*

*valarray<siz e\_t> d1;*

*public:*

*gslice() = default;*

*˜gslice() = default;*

*gslice(const gslice&) = default;*

*gslice(gslice&&) = default;*

*gslice& operator=(const gslice&) = default;*

*gslice& operator=(gslice&&) = default;*

*// ...*

*};*

Đoạn này của việc triển khai std :: gslice tương đương với:

*class gslice {*

*valarray<siz e\_t> siz e;*

*valarray<siz e\_t> stride;*

*valarray<siz e\_t> d1;*

*public:*

*// ...*

};

Sử dụng = default luôn tốt hơn so với việc tự viết triển khai ngữ nghĩa mặc định của riêng bạn. Một người nào đó giả định rằng tốt hơn là nên viết một cái gì đó, thay vì không có gì, có thể viết:

*class gslice {*

*valarray<siz e\_t> siz e;*

*valarray<siz e\_t> stride;*

*valarray<siz e\_t> d1;*

*public:*

*// ...*

*gslice(const gslice& a);*

*};*

*gslice::gslice(const gslice& a)*

*:siz e{a.size },*

*stride{a.stride},*

*d1{a.d1}*

*{*

*}*

Điều này không chỉ dài dòng, làm cho việc đọc khái niệm của gslice trở nên khó khăn hơn, mà còn mở ra cơ hội để mắc sai lầm. Ví dụ: khi người dùng cung cấp một chức năng, trình biên dịch không còn biết ngữ nghĩa của chức năng đó và một số tối ưu hóa trở nên bị hạn chế. Đối với các hoạt động mặc định, những tối ưu hóa đó có thể không đáng kể.

#### 17.6.2 Hoạt động mặc định (Default Operations)

Ý nghĩa mặc định của mỗi hoạt động được tạo là áp dụng hoạt động cho từng thành viên dữ liệu cơ sở và không tĩnh của lớp. Ví dụ:

*struct S {*

*string a;*

*int b;*

*};*

*S f(S arg)*

*{*

*S s0 {}; // default construction: {"",0}*

*S s1 {s0}; // copy construction*

*s1 = arg; // copy assignment*

*return s1; // move construction*

*}*

Lưu ý rằng giá trị của một đối tượng được chuyển đến của một kiểu dựng sẵn là không thay đổi. Đó là điều đơn giản và nhanh nhất để trình biên dịch thực hiện. Nếu chúng ta muốn một điều gì đó khác được thực hiện cho một thành viên của một lớp, chúng ta phải viết các hoạt động di chuyển của chúng ta cho lớp đó.

Trạng thái di chuyển mặc định không được đảm bảo rằng một hoạt động tùy ý trên một đối tượng được chuyển đến sẽ hoạt động chính xác. Nếu bạn cần đảm bảo mạnh mẽ hơn, hãy viết các hoạt động của riêng bạn.

#### 17.6.3 Sử dụng thao tác mặc định (Using Default Operations)

Phần này trình bày một số ví dụ chứng minh cách sao chép, di chuyển và hủy được liên kết một cách hợp lý. Nếu chúng không được liên kết, các lỗi sẽ không được trình biên dịch phát hiện.

##### 17.6.3.1 Trình tạo mặc định (Default Constructors)

*struct X {*

*X(int); // require an int to initialize an X (khởi tao X)*

*};*

Bằng cách khai báo một phương thức khởi tạo yêu cầu đối số nguyên, lập trình viên yêu cầu rõ ràng người dùng cần cung cấp một int để khởi tạo X. Nếu chúng ta cho phép tạo phương thức khởi tạo mặc định, quy tắc đơn giản đó sẽ bị vi phạm. Chúng ta có:

*X a {1}; // OK*

*X b {}; // error : no default constructor (không có hàm tạo mặc định)*

Nếu chúng ta cũng muốn có hàm tạo mặc định, chúng ta có thể xác định hoặc khai báo rằng chúng ta muốn hàm tạo mặc định do trình biên dịch tạo ra. Ví dụ:

*struct Y {*

*string s;*

*int n;*

*Y(const string& s); // initialize Y with a string*

*Y() = default; // allow default initialization with the default meaning (cho phép khởi tạo mặc định với nghĩa mặc định)*

*};*

##### 17.6.3.2 Duy trì bất biến (Maintaining Invariants)

Thông thường, một lớp có một bất biến. Nếu vậy, chúng tôi muốn các hoạt động sao chép và di chuyển để duy trì nó và trình hủy giải phóng mọi tài nguyên liên quan. Thật không may, trình biên dịch không thể trong mọi trường hợp biết được lập trình viên coi cái gì là bất biến. Ví dụ:

*struct Z { // invariant:*

*// my\_favorite is the index of my favor ite element of elem*

*// largest points to the element with the highest value in elem*

*vector<int> elem;*

*int my\_favorite;*

*int∗ largest;*

*};*

Lập trình viên đã nêu một bất biến trong nhận xét, nhưng trình biên dịch không đọc nhận xét. Hơn nữa, lập trình viên đã không để lại một gợi ý về cách mà bất biến đó được thiết lập và duy trì. Đặc biệt, không có hàm tạo hoặc phép gán nào được khai báo. Bất biến đó là mặc nhiên. Kết quả là một chữ Z có thể được sao chép và di chuyển bằng các thao tác mặc định:

*Z v0; // no initialization (oops! possibility of undeﬁned values)*

*Z val {{1,2,3},1,&val[2]}; // OK, but ugly and error-prone (Được, nhưng xấu và dễ xảy ra lỗi)*

*Z v2 = val; // copies: v2.largest points into val*

*Z v3 = move(val); // moves: val.elem becomes empty; v3.my\_favor ite is out of range*

##### 17.6.3.3 Bất biến tài nguyên (Resource Invariants)

Nhiều ứng dụng quan trọng và rõ ràng nhất của bất biến liên quan đến quản lý tài nguyên. Hãy xem xét một Handle đơn giản:

*template<class T> class Handle {*

*T∗ p;*

*public:*

*Handle(T∗ pp) :p{pp} { }*

*T& operator∗() { return ∗p; }*

*~Handle() { delete p; }*

*};*

Ý tưởng là bạn xây dựng một Handle cho một con trỏ tới một đối tượng được cấp phát bằng cách sử dụng new. Handle cung cấp quyền truy cập vào đối tượng được trỏ đến và cuối cùng xóa đối tượng đó. Ví dụ:

*void f1()*

*{*

*Handle<int> h {new int{99}};*

*// ...*

*}*

Handle khai báo một hàm tạo nhận đối số: điều này ngăn chặn việc tạo ra hàm tạo mặc định. Điều đó tốt vì một hàm tạo mặc định có thể khiến Handle <T> :: p không được khởi tạo:

*void f2()*

*{*

*Handle<int> h; // error : no default constructor (lỗi: không có hàm tạo mặc định)*

*// ...*

*}*

Sự vắng mặt của một hàm tạo mặc định giúp chúng ta tránh khỏi khả năng bị xóa với một địa chỉ bộ nhớ ngẫu nhiên.

Ngoài ra, Handle thông báo một trình hủy: điều này ngăn chặn việc tạo ra các hoạt động sao chép và di chuyển. Xem xét:

*void f3()*

*{*

*Handle<int> h1 {new int{7}};*

*Handle<int> h2 {h1}; // error : no copy constr uctor*

*// ...*

*}*

Nếu Handle có một phương thức khởi tạo sao chép mặc định, cả h1 và h2 sẽ có một bản sao của con trỏ và cả hai sẽ xóa nó. Kết quả sẽ không thành công và rất có thể là thảm họa.

Nếu muốn xây dựng bản sao, chúng ta có thể thực hiện như sau:

*template<class T>*

*class Handle {*

*// ...*

*Handle(const T& a) :p{new T{∗a.p}} { } // clone*

*};*

##### 17.6.3.4 Bất biến cụ thể từng phần (Partially Speciﬁed Invariants)

Các ví dụ rắc rối dựa trên các bất biến qua các hàm tạo hoặc hàm hủy là hiếm nhưng không phải là chưa từng thấy. Xem xét:

*class Tic\_tac\_toe {*

*public:*

*Tic\_tac\_toe(): pos(9) {} // always 9 positions (luôn là 9 vị trí)*

*Tic\_tac\_toe& operator=(const Tic\_tac\_toe& arg)*

*{*

*for(int i = 0; i<9; ++i)*

*pos.at(i) = arg.pos.at(i);*

*return ∗this;*

*}*

*// ... other operations ...*

*enum State { empty, nought, cross };*

*private:*

*vector<State> pos;*

*};*

Điều này đã được báo cáo là một phần của một chương trình thực tế. Nó sử dụng ‘‘ magic number ’’ 9 để thực hiện một phép gán sao chép truy cập đối số arg của nó mà không cần kiểm tra xem đối số có thực sự có chín phần tử hay không. Đây không phải là một đoạn mã tốt.

Chúng ta cần gán bản sao, vì vậy chúng ta cũng phải xác định hàm hủy. Tại thời điểm này, chúng tôi nhận thấy rằng phép gán bản sao do người dùng xác định về cơ bản là phép gán chúng tôi đã nhận theo mặc định, vì vậy chúng tôi cũng có thể = default điều đó. Thêm một hàm tạo bản sao để có sự hoàn chỉnh:

*class Tic\_tac\_toe {*

*public:*

*Tic\_tac\_toe(): pos(9) {} // always 9 positions*

*Tic\_tac\_toe(const Tic\_tac\_toe&) = default;*

*Tic\_tac\_toe& operator=(const Tic\_tac\_toe& arg) = default;*

*˜Tic\_tac\_toe() = default;*

*// ... other operations ...*

*enum State { empty, nought, cross };*

*private:*

*vector<State> pos;*

*};*

Chúng ta có thể thêm các phép toán di chuyển một cách an toàn. Cách đơn giản nhất để làm điều đó là loại bỏ explicit = defaults, và sau đó chúng ta thấy rằng *Tic\_tac\_toe* thực sự là một kiểu hoàn toàn bình thường:

*class Tic\_tac\_toe {*

*public:*

*// ... other operations ...*

*enum State { empty, nought, cross };*

*private:*

*vector<State> pos {Vector<State>(9)}; // always 9 positions*

*};*

Một kết luận rút ra từ điều này và các ví dụ khác là đối với mọi lớp, chúng ta nên hỏi:

* Có cần một hàm tạo mặc định không (vì hàm mặc định không phù hợp hoặc đã
* được hỗ trợ bởi một hàm tạo khác)?
* Có cần thiết bị hủy không (ví dụ: vì một số tài nguyên cần được giải phóng)?
* Các hoạt động sao chép có cần thiết không (vì ngữ nghĩa sao chép mặc định không đầy đủ, ví dụ: vì lớp được hiểu là một lớp cơ sở hoặc vì nó chứa các con trỏ đến các đối tượng phải bị xóa bởi lớp)?
* Các thao tác di chuyển có cần thiết không (vì ngữ nghĩa mặc định không đầy đủ, ví dụ: vì một đối tượng trống không có ý nghĩa)?

Đặc biệt, chúng ta không bao giờ chỉ nên xem xét một trong những hoạt động này một cách riêng lẻ.

#### 17.6.4 Xóa hàm (deleted Functions)

Chúng ta có thể ‘‘ xóa ’’ một hàm; nghĩa là, chúng ta có thể nói rằng một hàm không tồn tại và sử dụng nó là một lỗi. Cách sử dụng rõ ràng nhất là loại bỏ các hàm mặc định khác. Ví dụ, người ta thường muốn ngăn chặn việc sao chép các lớp được sử dụng làm cơ sở vì việc sao chép như vậy dễ dẫn đến cắt (slicing):

*class Base {*

*// ...*

*Base& operator=(const Base&) = delete; // disallow copying (không cho phép sao chép)*

*Base(const Base&) = delete;*

*Base& operator=(Base&&) = delete; // disallow moving (không cho phép di chuyển)*

*Base(Base&&) = delete;*

*};*

*Base x1;*

*Base x2 {x1}; // error : no copy constructor*

Việc bật và tắt tính năng sao chép và di chuyển thường được thực hiện thuận tiện hơn bằng cách nói những gì chúng ta muốn (sử dụng = default) thay vì nói những gì chúng ta không muốn (sử dụng = delete). Tuy nhiên, chúng ta có thể xóa bất kỳ hàm nào mà chúng ta có thể khai báo. Ví dụ: chúng ta có thể loại bỏ một chuyên môn khỏi tập hợp các chuyên môn có thể có của một mẫu hàm:

*template<class T>*

*T∗ clone(T∗ p) // return copy of \*p*

*{*

*return new T{∗p};*

*};*

*Foo∗ clone(Foo∗) = delete; // don’t try to clone a Foo (đừng cố sao chép Foo)*

*void f(Shape∗ ps, Foo∗ pf)*

*{*

*Shape∗ ps2 = clone(ps); // ﬁne*

*Foo∗ pf2 = clone(pf); // error : clone(Foo\*) deleted*

*}*

Một ứng dụng khác là loại bỏ một chuyển đổi không mong muốn. Ví dụ:

*struct Z {*

*// ...*

*Z(double); // can initialize with a double (có thể khởi tạo với double)*

*Z(int) = delete; // but not with an integer (nhưng không phải với một số nguyên)*

*};*

*void f()*

*{*

*Z z1 {1}; // error : Z(int) deleted*

*Z z2 {1.0}; // OK*

*}*

Một cách sử dụng nữa là kiểm soát nơi có thể cấp phát một lớp:

*class Not\_on\_stack {*

*// ...*

*˜Not\_on\_stack() = delete;*

*};*

*class Not\_on\_free\_store {*

*// ...*

*void∗ operator new(siz e\_t) = delete;*

*};*

Bạn không thể sử dụng biến cục bộ không thể bị hủy và bạn không thể cấp phát một đối tượng trên kho lưu trữ miễn phí khi bạn có = deleteted toán tử cấp phát bộ nhớ của lớp của nó. Ví dụ:

*void f()*

*{*

*Not\_on\_stack v1; // error : can’t destroy*

*Not\_on\_free\_store v2; // OK*

*Not\_on\_stack∗ p1 = new Not\_on\_stack; // OK*

*Not\_on\_free\_store∗ p2 = new Not\_on\_free\_store; // error : can’t allocate(lỗi: không thể*

*phân bổ)*

*}*

Tuy nhiên, chúng ta không bao giờ có thể xóa đối tượng Not\_on\_stack đó. Kỹ thuật thay thế để đặt hàm hủy ở chế độ riêng tư có thể giải quyết vấn đề đó.

### 17.7 Lời khuyên (Advice)

[1] Thiết kế hàm tạo, phép gán và hàm hủy dưới dạng một tập hợp các thao tác; §17.1.

[2] Sử dụng một hàm tạo để thiết lập một bất biến cho một lớp; §17.2.1.

[3] Nếu một hàm tạo có được một tài nguyên, thì lớp của nó cần một hàm hủy để giải phóng tài nguyên đó; §17.2.2.

[4] Nếu một lớp có một hàm ảo, nó cần một hàm hủy ảo; §17.2.5.

[5] Nếu một lớp không có hàm tạo, nó có thể được khởi tạo bằng cách khởi tạo thành viên; §17.3.1.

[6] Ưu tiên khởi tạo {} hơn khởi tạo = và (); §17.3.2.

[7] Cung cấp cho một lớp một hàm tạo mặc định nếu và chỉ khi có giá trị mặc định là ‘‘ natural ’’; §17.3.3.

[8] Nếu một lớp là một vùng chứa, hãy giv e nó là một phương thức khởi tạo danh sách khởi tạo; §17.3.4.

[9] Khởi tạo các thành viên và cơ sở theo thứ tự khai báo của chúng; §17.4.1.

[10] Nếu một lớp có một thành viên tham chiếu, nó có thể cần các hoạt động sao chép (hàm tạo sao chép và gán sao chép); §17.4.1.1.

[11] Thích khởi tạo thành viên hơn là gán trong một phương thức khởi tạo; §17.4.1.1.

[12] Sử dụng bộ khởi tạo trong lớp để cung cấp các giá trị mặc định; §17.4.4.

[13] Nếu một lớp là một xử lý tài nguyên, nó có thể cần các hoạt động sao chép và di chuyển; §17.5.

[14] Khi viết một phương thức khởi tạo sao chép, hãy cẩn thận sao chép mọi phần tử cần được sao chép (cẩn thận với các trình khởi tạo mặc định); §17.5.1.1.

[15] Hoạt động sao chép phải cung cấp tính tương đương và độc lập; §17.5.1.3.

[16] Cẩn thận với các cấu trúc dữ liệu vướng víu; §17.5.1.3.

[17] Thích chuyển ngữ nghĩa và copy-on-write sang copy cạn; §17.5.1.3.

[18] Nếu một lớp được sử dụng làm lớp cơ sở, hãy bảo vệ khỏi việc cắt lát; §17.5.1.4.

[19] Nếu một lớp cần một hoạt động sao chép hoặc một hàm hủy, nó có thể cần một hàm tạo, một hàm hủy, một phép gán bản sao và một phương thức tạo bản sao; §17.6.

[20] Nếu một lớp có một thành viên con trỏ, nó có thể cần một hàm hủy và các hoạt động sao chép không mặc định; §17.6.3.3.

[21] Nếu một lớp là một xử lý tài nguyên, nó cần một hàm tạo, một hàm hủy và các hoạt động sao chép không mặc định; §17.6.3.3.

[22] Nếu một hàm tạo, phép gán hoặc hàm hủy mặc định phù hợp, hãy để trình biên dịch tạo nó (không tự viết lại); §17.6.

[23] Nói rõ ràng về những bất biến của bạn; sử dụng các hàm tạo để thiết lập chúng và các phép gán để duy trì chúng; §17.6.3.2.

[24] Đảm bảo rằng các bài tập sao chép là an toàn để tự chuyển nhượng; §17.5.1.

[25] Khi thêm một thành viên mới vào một lớp, hãy kiểm tra xem có các hàm tạo do người dùng định nghĩa cần được cập nhật để khởi tạo thành viên đó hay không; §17.5.1.

## Chap 18: Quá tải nhà điều hành

### 18.1 Giới thiệu (Introduction)

Mọi lĩnh vực kỹ thuật - và hầu hết các lĩnh vực phi kỹ thuật - đều phát triển các ký hiệu viết tắt thông thườngđể thuận tiện cho việc trình bày và thảo luận liên quan đến các khái niệm được sử dụng thường xuyên. Vì ví dụ, vì quen biết lâu,rõ ràng hơn đối với chúng tôi.

Giống như hầu hết các ngôn ngữ khác, C ++ hỗ trợ một tập hợp các toán tử cho các kiểu tích hợp của nó. Tuy nhiên, hầu hếtcepts mà các toán tử được sử dụng thông thường không phải là các kiểu tích hợp sẵn trong C ++, vì vậy chúng phải là được biểu diễn dưới dạng các kiểu do người dùng xác định. Ví dụ: nếu bạn cần số học phức tạp, đại số ma trận, tín hiệu logic, hoặc chuỗi ký tự trong C ++, bạn sử dụng các lớp để biểu diễn các khái niệm này. Xác định oper-tùy chọn cho các lớp như vậy đôi khi cho phép lập trình viên cung cấpký hiệu thuần túy để thao tác các đối tượng hơn có thể đạt được chỉ bằng cách sử dụng chức năng cơ bản.

### 18.2 Các chức năng của nhà điều hành (Operator Functions)

Các hàm xác định ý nghĩa cho các toán tử sau có thể được khai báo:

*+ − ∗ / % ˆ &*

*| ̃ ! = <> +=*

*-= =∗ = / = % = ˆ = & |=*

*<< >>>>= <<= == != <=*

*>= && || ++ −− −>∗ ,*

*−> [] () new new[] delete delete[]*

Người dùng không thể xác định các toán tử sau:

*::* : phân giải phạm vi.

*.* : lựa chọn thành viên.

*. ∗* : lựa chọn thành viên thông qua con trỏ đến thành viên .Chúng lấy tên, thay vì giá trị, làm toán hạng thứ hai và cung cấp phương tiện chính cho giới thiệu đến các thành viên. Cho phép chúng bị quá tải sẽ dẫn đến sự kém tinh tế [Stroustrup, 1994].

Không thể quá tải ‘‘ toán tử ’’ được đặt tên vì chúng báo cáo thông tin cơ bản về hoạt động xóa:

*Sizeof:* kích thước của đối tượng.

*alignof:* căn chỉnh của đối tượng.

*typeid:* type\_info của một đối tượng.

Cuối cùng, toán tử biểu thức điều kiện bậc ba không thể được nạp chồng (vì không có funda- lý do tinh thần):

*?*: đánh giá có điều kiện.

#### 18.2.1 Toán tử nhị phân và đơn nguyên (Binary and Unary Operators)

Một toán tử nhị phân có thể được xác định bởi một hàm thành viên không tĩnh nhận một đối số hoặc mộthàm nonmember nhận hai đối số. Đối với bất kỳ toán tử nhị phân nào @, aa @ bb có thể được hiểudưới dạng aa.operator @ (bb) hoặc operator @ (aa, bb). Nếu cả hai đều được xác định, giải quyết quá tải xác định cách diễn giải, nếu có, được sử dụng.

Toán tử một ngôi, dù là tiền tố hay hậu tố, đều có thể được xác định bởi một hàm thành viên không tĩnh không lấy đối số hoặc một hàm nonmember lấy một đối số. Đối với bất kỳ toán tử đơn nguyên tiền tố nào @, @aa có thể được hiểu là aa.operator @ () hoặc operator @ (aa). Nếu cả hai đều được xác định, quá tải độ phân giải (§12.3) xác định cách giải thích nào, nếu có, được sử dụng. Đối với bất kỳ toán tử đơn vị postfix nào @, aa @ có thể được hiểu là aa.operator @ (int) hoặc operator @ (aa, int). Điều này được giải thích thêm trong §19.2.4. Nếu cả hai đều được xác định, giải quyết quá tải (§12.3) xác định cách diễn giải, nếu có,.Được sử dụng. Một toán tử chỉ có thể được khai báo cho cú pháp được định nghĩa cho nó trong ngữ pháp.

Ví dụ: người dùng không thể xác định % một bậc hoặc một bậc ba +. Xem xét:

*class X {*

*public: // members (with implicit this pointer):*

*X∗ operator&(); // prefix unary & (address of)*

*X operator&(X); // binar y & (and)*

*X operator++(int); // postfix increment (see §19.2.4)*

*X operator&(X,X); // error : ter nary*

*X operator/(); // error : unar y /*

*};*

Toán tử [] được mô tả, toán tử (), toán tử -> trong, toán tử ++ và −−, và các toán tử phân bổ và phân bổ.

Toán tử toán tử =, toán tử []toán tử ()và toán tử−> phải là các hàm thành viên không tĩnh.

#### 18.2.2 Ý nghĩa được xác định trước cho các toán tử (Predefined Meanings for Operators)

Ý nghĩa của một số toán tử cài sẵn được định nghĩa tương đương với một số kết hợp của các toán tử khác toán tử trên các đối số giống nhau. Ví dụ: nếu a là int, ++ a có nghĩa là a + = 1, điều này có nghĩa là a = a + 1. Các quan hệ như vậy không giữ cho các toán tử do người dùng xác định trừ khi người dùng định nghĩa chúng. Vì ví dụ, một trình biên dịch sẽ không tạo ra định nghĩa của Z :: operator + = () từ các định nghĩa của Z :: opera- tor + () và Z :: operator = ().

Các toán tử = (gán), & (address-of) vàcó giá trị trung bình được xác định trước- ings khi áp dụng cho các đối tượng lớp. Những ý nghĩa được xác định trước này có thể bị loại bỏ (‘‘ xóa ’’):

*class X {*

*public:*

*// ...*

*void operator=(const X&) = delete;*

*void operator&() = delete;*

*void operator,(const X&) = delete;*

*// ...*

*};*

*void f(X a, X b)*

*{*

*a = b; // error : no operator=()*

*&a; // error : no operator&()*

*a,b; // error : no operator,()*

*}*

Ngoài ra, chúng có thể được đưa ra các nghĩa mới bằng các định nghĩa phù hợp.

#### 18.2.3 Toán tử và các loại do người dùng xác định (Operators and User-Defined Types)

Một hàm toán tử phải là một thành viên hoặc có ít nhất một đối số của kiểu do người dùng xác định (các chức năng xác định lại toán tử mới và xóa không cần). Quy tắc này đảm bảo rằng người dùng không thể thay đổi ý nghĩa của một biểu thức trừ khi biểu thức chứa một đối tượng do người dùng xác định kiểu. Đặc biệt, không thể xác định một hàm toán tử hoạt động độc quyền trên con trỏ. Điều này đảm bảo rằng C ++ có thể mở rộng nhưng không thể thay đổi (ngoại trừ các toán tử =, &, và, đối với các đối tượng lớp).

Một hàm toán tử nhằm chấp nhận một kiểu dựng sẵn vì toán hạng đầu tiên của nó không thể là một chức năng thành viên. Ví dụ: hãy xem xét việc thêm một biến phức aa vào số nguyên 2: aa + 2 có thể, với một hàm thành viên được khai báo phù hợp, được hiểu là aa.operator + (2), nhưng 2 + aa không thể vì không có lớp int nào để định nghĩa + có nghĩa là 2.operator + (aa). Ngay cả khi có, hai các hàm thành viên sẽ cần thiết để đối phó với 2 + aa và aa + 2. Bởi vì trình biên dịch không biết ý nghĩa của một + do người dùng xác định, nó không thể giả định rằng toán tử là giao hoán và như vậy giải thích 2 + aa là aa + 2.

Các kiểu liệt kê là các kiểu do người dùng định nghĩa để chúng ta có thể xác định các toán tử cho chúng. Ví dụ:

*enum Day { sun, mon, tue, wed, thu, fri, sat };*

*Day& operator++(Day& d)*

*{*

*return d = (sat==d) ? sun : static\_cast<Day>(d+1);*

*}*

Mọi biểu thức đều được kiểm tra xem có mơ hồ không. Trong đó toán tử do người dùng xác định cung cấp giải thích, biểu thức được kiểm tra theo các quy tắc giải quyết quá tải trong.

#### 18.2.4 Đi qua các đối tượng (Passing Objects)

Khi chúng tôi xác định một toán tử, chúng tôi thường muốn cung cấp một ký hiệu thông thường, ví dụ: a = b + c. Do đó, chúng tôi có các lựa chọn hạn chế về cách truyền các đối số cho hàm toán tửvà cách nó trả về giá trị của nó. Ví dụ: chúng ta không thể yêu cầu đối số con trỏ và mong đợi pro-grammers để sử dụng toán tử address-of hoặc trả về một con trỏ và mong muốn người dùng bỏ qua nó:

∗ a = & b + & c không được chấp nhận.

Đối với các đối số, chúng ta có hai lựa chọn chính:

• Giá trị chuyển tiếp

• Tham khảo qua

Đối với các đối tượng nhỏ, chẳng hạn như một đến bốn từ, gọi theo giá trị thường là một sự thay thế khả thi và thường là một thứ mang lại hiệu suất tốt nhất.

Các đối tượng lớn hơn, chúng tôi chuyển qua tham chiếu.

Đặc biệt, chúng tôi sử dụng tham chiếu const để chuyển các đối tượng lớn không được sửa đổi bởi được gọi là hàm.

Thông thường, một toán tử trả về một kết quả. Trả lại một con trỏ hoặc một tham chiếu đến một tham chiếu mới được tạo đối tượng thường là một ý tưởng rất tồi: sử dụng một con trỏ đưa ra các vấn đề về ký hiệu và tham chiếu đến một đối tượng trên cửa hàng miễn phí (cho dù bằng con trỏ hay bằng tham chiếu) dẫn đến quản lý bộ nhớ các vấn đề. Thay vào đó, hãy trả về các đối tượng theo giá trị. Đối với các đối tượng lớn, chẳng hạn như Ma trận, hãy xác định toán hạng di chuyển-để thực hiện việc chuyển các giá trị như vậy một cách hiệu quả. Ví dụ:

*Matrix operator+(const Matrix& a, const Matrix& b) // trả về theo giá trị*

*{*

*Matrix res {a};*

*return res+=b;*

*}*

Nếu một hàm chỉ đơn giản là chuyển một đối tượng cho một hàm khác, thì một đối số tham chiếu rvalue sẽ được sử dụng .

#### 18.2.5 Toán tử trong Không gian tên (Operators in Namespaces)

Toán tử là thành viên của một lớp hoặc được định nghĩa trong một số không gian tên (có thể là tên chung-không gian). Hãy xem xét phiên bản đơn giản này của chuỗi I / O từ thư viện chuẩn:

*using namespace std { // std đơn giản hóa*

*class string {*

*// ...*

*};*

*class ostream {*

*// ...*

*ostream& operator<<(const char∗); // output C-style string*

*};*

*extern ostream cout;*

*ostream& operator<<(ostream&, const string&); // output std::string*

*} // namespace std*

*int main()*

*{*

*const char∗ p = "Hello";*

*std::string s = "world";*

*std::cout << p << ", " << s << "!\n";*

*}*

Các toán tử được xác định trong không gian tên có thể được tìm thấy dựa trên các loại toán hạng của chúng giống như các hàm có thể được tìm thấy dựa trên các loại đối số của chúng. Đặc biệt, cout nằm trong không gian tên std, vì vậy std là được cân nhắc khi tìm định nghĩa phù hợp của <<. Theo cách đó, trình biên dịch tìm và sử dụng:

*std::operator<<(std::ostream&, const std::string&)*

Hãy xem xét một toán tử nhị phân @. Nếu x thuộc loại X và y thuộc loại Y, x @ y được giải quyết như sau:

• Nếu X là một lớp, hãy tìm toán tử @ như một thành viên của X hoặc như một thành viên của một cơ sở của X; và

• Tìm kiếm các khai báo của operator @ trong ngữ cảnh xung quanh x @ y; và

• Nếu X được định nghĩa trong không gian tên N, hãy tìm các khai báo của operator @ trong N; và

• Nếu Y được định nghĩa trong không gian tên M, hãy tìm các khai báo của toán tử @ trong M.

Các toán tử đơn phân được giải quyết một cách tương tự.

Lưu ý rằng trong tra cứu nhà điều hành không có ưu tiên nào dành cho các thành viên hơn là những người không phải là thành viên. Điều này khác từ tra cứu các hàm được đặt tên .Việc thiếu ẩn các toán tử đảm bảo rằng các toán tử không bao giờ không thể truy cập được và người dùng có thể cung cấp các ý nghĩa mới cho một toán tử mà không sửa đổi các khai báo lớp hiện có.

Đặc biệt, thư viện iostream tiêu chuẩn định nghĩa << hàm thành viên để xuất ra các kiểu tích hợp sẵn, và người dùng có thể định nghĩa << để xuất ra các kiểu do người dùng định nghĩa mà không cần sửa đổi lớp ostream.

### 18.3 Một loại số phức (A Complex Number Type)

Việc triển khai các số phức được trình bày trong §18.1 là quá hạn chế để làm hài lòng bất kỳ ai.

Ví dụ, chúng tôi mong đợi điều này hoạt động:

*void f()*

*{*

*complex a {1,2};*

*complex b {3};*

*complex c {a+2.3};*

*complex d {2+b};*

*b=c∗2∗c;*

*}*

Ngoài ra, chúng tôi mong muốn được cung cấp thêm một số toán tử bổ sung, chẳng hạn như == cho compar-ison và << cho đầu ra, và một tập hợp các hàm toán học phù hợp, chẳng hạn như sin () và sqrt ().

Lớp phức hợp là một loại bê tông. Ngoài ra, người sử dụng số học phức tạp phụ thuộc rất nhiều vào các toán tử đến mức định nghĩa của số phức đưa vào chơi hầu hết các quy tắc cơ bản để nạp chồng toán tử.

Kiểu phức tạp được phát triển trong phần này sử dụng double cho các đại lượng vô hướng của nó và gần như tương đương đến phức hợp thư viện chuẩn <double>.

#### 18.3.1 Nhà điều hành thành viên và không phải thành viên (Member and Nonmember Operators)

Tôi muốn giảm thiểu số lượng các hàm thao tác trực tiếp với việc biểu diễn mộtsự vật. Điều này có thể đạt được bằng cách xác định chỉ các toán tử vốn đã sửa đổi giá trị củađối số đầu tiên, chẳng hạn như + =, trong chính lớp đó. Các toán tử chỉ đơn giản là tạo ra một giá trị mới dựa trêncác giá trị của các đối số của chúng, chẳng hạn như +, sau đó được xác định bên ngoài lớp và sử dụngtrong quá trình triển khai của họ:

*class complex {*

*double re, im;*

*public:*

*complex& operator+=(complex a); // cần quyền truy cập để biểu diễn*

*// ...*

*};*

*complex operator+(complex a, complex b)*

*{*

*return a += b; // truy cập đại diện thông qua +=*

*}*

Các toán tử gán kết hợp như + = và ∗ = có xu hướng dễ xác định hơn là ‘‘ sim- của chúng ple ’’ đối âm + và ∗. Điều này khiến hầu hết mọi người ngạc nhiên lúc đầu, nhưng nó xuất phát từ thực tế là ba các đối tượng tham gia vào một phép toán + (hai toán hạng và kết quả), trong khi chỉ có hai đối tượng tham gia vào một phép toán + =. Trong trường hợp thứ hai, hiệu quả thời gian chạy được cải thiện bằng cách loại bỏ cần các biến tạm thời.

Một trình tối ưu hóa tốt sẽ tạo ra mã gần với mã tối ưu để sử dụng cả toán tử + thuần túy.

Tuy nhiên, không phải lúc nào chúng tôi cũng có một trình tối ưu hóa tốt và không phải tất cả các loại đều đơn giản cũng như phức tạp, vì vậy §19.4 thảo luận về các cách xác định các toán tử có quyền truy cập trực tiếp vào biểu diễn của các lớp.

#### 18.3.2 Số học chế độ hỗn hợp (Mixed - Mode Arithmetic)

Để đối phó với 2 + z, trong đó z là một phức, chúng ta cần xác định toán tử + để chấp nhận các toán hạng khác nhau các loại. Theo thuật ngữ Fortran, chúng ta cần số học ở chế độ hỗn hợp. Chúng ta có thể đạt được điều đó đơn giản bằng cách thêm các phiên bản thích hợp của các toán tử:

*class complex {*

*double re, im;*

*public:*

*complex& operator+=(complex a)*

*{*

*re += a.re;*

*im += a.im;*

*return ∗this;*

*}*

*complex& operator+=(double a)*

*{*

*re += a;*

*return ∗this;*

*}*

*// ...*

*};*

Ba biến thể của toán tử + () có thể được định nghĩa bên ngoài phức hợp.

#### 18.3.3 Chuyển đổi (Conversions)

Để đối phó với các bài tập và khởi tạo các biến phức tạp với các đại lượng vô hướng, chúng ta cần một chuyển đổi của một số vô hướng (số nguyên hoặc số dấu phẩy động) thành một phức. Ví dụ:

*complex b {3}; // should mean b.re=3, b.im=0*

*void comp(complex x)*

*{*

*x = 4; // should mean x.re=4, x.im=0*

*// ...*

*}*

Chúng ta có thể đạt được điều đó bằng cách cung cấp một hàm tạo nhận một đối số duy nhất. Một hàm tạo lấy một đối số duy nhất chỉ định một chuyển đổi từ kiểu đối số của nó sang kiểu của phương thức khởi tạo.

Hàm tạo chỉ định cách nhúng truyền thống của đường thực trong mặt phẳng phức.

Một hàm tạo là một đơn thuốc để tạo một giá trị của một kiểu nhất định. Hàm tạo được sử dụng khi một giá trị của một kiểu được mong đợi và khi một giá trị như vậy có thể được tạo bởi một phương thức khởi tạo từ giá trị được cung cấp dưới dạng bộ khởi tạo hoặc giá trị được chỉ định. Do đó, một hàm tạo yêu cầu một đối số duy nhất không cần được gọi một cách rõ ràng. Ví dụ:

*complex b {3};*

*có nghĩa*

*complex b {3,0};*

Chuyển đổi do người dùng xác định chỉ được áp dụng hoàn toàn nếu nó là duy nhất. Nếu bạn không muốn ...contructor được sử dụng ngầm định, hãy khai báo nó rõ ràng.

Sử dụng các đối số mặc định, chúng ta có thể viết tắt:

*class complex {*

*double re, im;*

*public:*

*complex(double r =0, double i =0) : re{r}, im{i} { }*

*// ...*

*};*

Theo mặc định, việc sao chép các giá trị phức tạp được định nghĩa là sao chép phần thực và phần ảo

##### 18.3.3.1 Chuyển đổi Toán hạng (Conversions of Operands)

Chúng tôi đã xác định ba phiên bản của mỗi trong bốn toán tử số học tiêu chuẩn:

*complex operator+(complex,complex);*

*complex operator+(complex,double);*

*complex operator+(double ,complex);*

*// ...*

Điều này có thể trở nên tẻ nhạt và những gì tẻ nhạt dễ trở thành lỗi. Điều gì sẽ xảy ra nếu chúng tôi có ba thay thế-cho kiểu của mỗi đối số cho mỗi hàm? Chúng ta sẽ cần ba phiên bản của mỗi tội-hàm gle-đối số, chín phiên bản của mỗi hàm hai đối số, 27 phiên bản của mỗi hàm ba-hàm đối số, v.v ... Thường thì những biến thể này rất giống nhau. Trên thực tế, hầu hết tất cả các biến thể đều liên quan đến chuyển đổi đơn giản của các đối số thành một kiểu chung, theo sau là một thuật toán tiêu chuẩn.

Giải pháp thay thế để cung cấp các phiên bản khác nhau của một hàm cho mỗi tổ hợp các đối sốlà dựa vào chuyển đổi. Ví dụ: lớp phức hợp của chúng tôi cung cấp một phương thức khởi tạo chuyển đổi một nhân đôi thành phức hợp. Do đó, chúng ta chỉ có thể khai báo một phiên bản của phương thức bình đẳng-ator cho phức tạp

Khi một số biến thể của một hàm hoặc một toán tử tồn tại, trình biên dịch phải chọn '' đúng '' biến thể dựa trên các loại đối số và các chuyển đổi có sẵn (tiêu chuẩn và do người dùng xác định). Trừ khi một kết quả phù hợp nhất tồn tại, một biểu thức không rõ ràng và là một lỗi .

Một đối tượng được xây dựng bằng cách sử dụng hàm tạo trong một biểu thức một cách rõ ràng hoặc ngầm định là tự động và sẽ bị phá hủy ngay cơ hội đầu tiên .

Không có chuyển đổi ngầm định nào do người dùng xác định được áp dụng cho phía bên trái của a. (hoặc a ->). Đây là trường hợp ngay cả khi. là ẩn ý. Ví dụ:

*void g(complex z)*

*{*

*3+z; // OK: complex(3)+z*

*3.operator+=(z); // error: 3 không phải là một đối tượng lớp*

*3+=z; // error: 3 không phải là một đối tượng lớp*

*}*

Do đó, bạn có thể gần đúng với khái niệm rằng một toán tử yêu cầu một giá trị làm toán hạng bên trái của nó bằng biến nhà điều hành đó thành thành viên. Howev er, đó chỉ là ước tính vì có thể truy cập tạm thời với một thao tác sửa đổi, chẳng hạn như toán tử + = ():

*complex x {4,5}*

*complex z {sqr t(x)+={1,2}}; // like tmp=sqr t(x), tmp+={1,2}*

#### 18.3.4 Chữ viết (Literals)

Chúng tôi có nghĩa đen của các loại tích hợp. Ví dụ: 1,2 và 12e3 là các chữ kiểu kép. Đối với com-plex, chúng ta có thể tiến gần đến điều đó bằng cách khai báo các hàm tạo constexpr .Ví dụ:

*class complex {*

*public:*

*constexpr complex(double r =0, double i =0) : re{r}, im{i} { }*

*// ...*

*}*

Do đó, một phức hợp có thể được xây dựng từ các bộ phận cấu thành của nó tại thời điểm biên dịch giống như một chữ từ một loại cài sẵn. Ví dụ:

*complex z1 {1.2,12e3};*

*constexpr complex z2 {1.2,12e3}; // khởi tạo thời gian biên dịch được đảm bảo*

Khi các hàm tạo đơn giản và nội dòng, và đặc biệt là khi chúng là constexpr, thì đó là lý do-có thể nghĩ về các lời gọi phương thức khởi tạo với các đối số theo nghĩa đen là các nghĩa đen.

Có thể đi xa hơn và giới thiệu một chữ do người dùng xác định để hỗ trợ cho com-loại plex. Đặc biệt, chúng ta có thể định nghĩa i là một hậu tố có nghĩa là '' tưởng tượng ''. Ví dụ:

*constexpr complex<double> operator "" i(long double d) // hình chữ y*

*{*

*return {0,d}; // complex là một kiểu chữ*

*}*

Điều này sẽ cho phép chúng tôi viết:

*complex z1 {1.2+12e3i};*

*complex f(double d)*

*{*

*auto x {2.3i};*

*return x+sqrt(d+12e3i)+12e3i;*

*}*

Chữ do người dùng định nghĩa này mang lại cho chúng tôi một lợi thế so với những gì chúng tôi nhận được từ các trình tạo constexpr: chúng tôi có thể sử dụng các ký tự do người dùng xác định ở giữa các biểu thức trong đó ký hiệu {} chỉ có thể được sử dụng khi đủ điều kiện bởi một tên loại

Tôi nghi ngờ rằng việc lựa chọn phong cách nghĩa đen phụ thuộc vào cảm quan thẩm mỹ của bạn và các quy ước lĩnh vực công việc của bạn. Tổ hợp thư viện tiêu chuẩn sử dụng các hàm tạo constexpr thay vì một người dùng- nghĩa đen được xác định.

#### 18.3.5 Chức năng của Accessor (Accessor Functions)

Cho đến nay, chúng tôi đã cung cấp phức hợp lớp chỉ với các hàm tạo và toán tử số học. Đó không phải là khá đủ để sử dụng thực tế. Đặc biệt, chúng ta thường cần có khả năng kiểm tra và thay đổi giá trị của phần thực và phần ảo:

*class complex {*

*double re, im;*

*public:*

*constexpr double real() const { return re; }*

*constexpr double imag() const { return im; }*

*void real(double r) { re = r; }*

*void imag(double i) { im = i; }*

*// ...*

*};*

Tôi không coi việc cung cấp quyền truy cập cá nhân cho tất cả các thành viên trong lớp là một ý tưởng hay; nói chung, nó không phải. Đối với nhiều loại, quyền truy cập riêng lẻ (đôi khi được gọi là các hàm get-and-set) là một lời mời đến thảm họa. Nếu chúng ta không cẩn thận, quyền truy cập cá nhân có thể làm tổn hại đến một cái bất biến, và nó thường làm phức tạp các thay đổi đối với biểu diễn. Ví dụ, hãy xem xét các cơ hội cho lạm dụng từ việc cung cấp getters và setters cho mọi thành viên của Ngày từ hoặc (thậm chí nhiều hơn vì vậy) đối với Chuỗi từ. Tuy nhiên, đối với phức tạp, real () và images () có ý nghĩa về mặt ngữ nghĩa: một số thuật toán được viết rõ ràng nhất nếu chúng có thể thiết lập phần thực và phần ảo một cách độc lập.

#### 18.3.6 Chức năng của người trợ giúp (Helper Functions)

Nếu chúng ta đặt tất cả các bit và các mảnh lại với nhau, lớp phức tạp sẽ trở thành:

*class complex {*

*double re, im;*

*public:*

*constexpr complex(double r =0, double i =0) : re(r), im(i) { }*

*constexpr double real() const { return re; }*

*constexpr double imag() const { return im; }*

*void real(double r) { re = r; }*

*void imag(double i) { im = i; }*

*complex& operator+=(complex);*

*complex& operator+=(double);*

*// -=, \*=, and /=*

*};*

Ngoài ra, chúng tôi phải cung cấp một số chức năng trợ giúp:

*complex operator+(complex,complex);*

*complex operator+(complex,double);*

*complex operator+(double ,complex);*

*// binar y -, \*, and /*

*complex operator−(complex); // unar y trừ*

*complex operator+(complex); // unar y thêm*

*bool operator==(complex,complex);*

*bool operator!=(complex,complex);*

*istream& operator>>(istream&,complex&); // đầu vào*

*ostream& operator<<(ostream&,complex); // đầu ra*

Lưu ý rằng các thành viên real () và virtual () là yếu tố cần thiết để xác định các phép so sánh. Các định nghĩa của hầu hết các hàm trợ giúp sau đây dựa trên real () và virtual () một cách tương tự.

### 18.4 Chuyển đổi loại (Type Conversion)

Việc chuyển đổi kiểu có thể được thực hiện bằng

• Một hàm tạo nhận một đối số duy nhất.

• Toán tử chuyển đổi .

Trong cả hai trường hợp, chuyển đổi có thể

• Rõ ràng; nghĩa là, việc chuyển đổi chỉ được thực hiện trong lần khởi tạo trực tiếp (§16.2.6), tức là trình khởi tạo không sử dụng dấu =.

• Ngụ ý; nghĩa là, nó sẽ được áp dụng ở bất cứ nơi nào nó có thể được sử dụng một cách rõ ràng ,ví dụ: nhưmột đối số hàm.

#### 18.4.1 Nhà điều hành chuyển đổi (Conversion Operators)

Việc sử dụng một hàm tạo lấy một đối số duy nhất để chỉ định chuyển đổi kiểu là thuận tiện nhưng có nghĩa là cation có thể không mong muốn. Ngoài ra, một hàm tạo không thể chỉ định

[1] chuyển đổi ngầm định từ loại do người dùng xác định sang loại tích hợp (vì loại không phải là lớp), hoặc

[2] chuyển đổi từ một lớp mới sang một lớp đã xác định trước đó (mà không sửa đổi decla- khẩu phần ăn cho lớp cũ).

Các hàm chuyển đổi dường như đặc biệt hữu ích để xử lý các cấu trúc dữ liệu khi đọc (được triển khai bởi một toán tử chuyển đổi) là không đáng kể, trong khi việc gán và khởi tạo là rõ ràng ít tầm thường hơn. Các loại istream và ostream dựa vào một hàm chuyển đổi để kích hoạt các câu lệnh như:

*while (cin>>x)*

*cout<<x;*

Thao tác nhập cin >> x trả về dòng istream &. Giá trị đó được chuyển đổi hoàn toàn thành giá trị ind-cating trạng thái của cin. Giá trị này sau đó có thể được kiểm tra bởi while (xem §38.4.4). Tuy nhiên, nó là điển hình-thực sự không phải là một ý tưởng hay khi xác định một chuyển đổi ngầm định từ loại này sang loại khác theo cách mà thông tin bị mất trong quá trình chuyển đổi.

Nếu cả chuyển đổi do người dùng xác định và toán tử do người dùng xác định đều được xác định, thì có thể nhận được sự không rõ ràng giữa các toán tử do người dùng xác định và các toán tử cài sẵn. Ví dụ:

*int operator+(Tiny,Tiny);*

*void f(Tiny t, int i)*

*{*

*t+i; // error, ambiguous: ‘‘operator+(t,Tiny(i))’’ or ‘‘int(t)+i’’?*

*}*

Do đó, tốt nhất là nên dựa vào chuyển đổi do người dùng xác định hoặc toán tử do người dùng xác định cho một loại, nhưng không phải cả hai.

#### 18.4.2 Nhà điều hành chuyển đổi rõ ràng (Conversion Operators)

Các toán tử chuyển đổi có xu hướng được xác định để chúng có thể được sử dụng ở mọi nơi. Tuy nhiên, nó có thể-ble để khai báo toán tử chuyển đổi một cách rõ ràng và chỉ áp dụng cho việc khởi tạo trực tiếp,trong đó một phương thức khởi tạo rõ ràng tương đương sẽ được sử dụng. Ví dụ, thư viện tiêu chuẩn unique\_ptr có một chuyển đổi rõ ràng thành bool:

*template <typename T, typename D = default\_delete<T>>*

*class unique\_ptr {*

*public:*

*explicit operator bool() const noexcept; // không \* cái này chứa một con trỏ (không phải là nullptr)?*

*// ...*

*};*

Lý do để khai báo toán tử chuyển đổi này rõ ràng là để tránh việc sử dụng nó trong các ngữ cảnh đáng ngạc nhiên.

Nếu chuyển đổi của unique\_ptr thành bool không rõ ràng, thì hai định nghĩa cuối cùng sẽ được tổng hợp.

Giá trị của b sẽ trở thành true và giá trị của x sẽ trở thành 1 hoặc 2 (tùy thuộc vào cho dù q có hợp lệ hay không).

#### 18.4.3 Mơ hồ (Ambiguities)

Việc gán giá trị kiểu V cho một đối tượng thuộc lớp X là hợp pháp nếu có toán tử gánX :: operator = (Z) sao cho V là Z hoặc có một chuyển đổi duy nhất của V thành Z. Quá trình khởi tạo được xử lý tương đươngđồng minh.

Trong một số trường hợp, giá trị của kiểu mong muốn có thể được xây dựng bằng cách sử dụng lặp lại các hàm tạo hoặc toán tử chuyển đổi. Điều này phải được xử lý bằng các chuyển đổi rõ ràng; chỉ một cấp độ do người dùng xác định chuyển đổi ngầm là hợp pháp. Trong một số trường hợp, giá trị của kiểu mong muốn có thể được xây dựng trong nhiều hơn một cách; những trường hợp như vậy là bất hợp pháp. Ví dụ:

*class X { /\* ... \*/ X(int); X(const char∗); };*

*class Y { /\* ... \*/ Y(int); };*

*class Z { /\* ... \*/ Z(X); };*

*X f(X);*

*Y f(Y);*

*Z g(Z);*

*void k1()*

*{*

*f(1); // lỗi: không rõ ràng f (X (1)) hoặc f (Y (1))?*

*f(X{1}); // OK*

*f(Y{1}); // OK*

*g("Mack"); // error: cần hai chuyển đổi do người dùng xác định; g (Z {X {"Mack"}}) không được thử*

*g(X{"Doc"}); // OK: g(Z{X{"Doc"}})*

*g(Z{"Suzy"}); // OK: g(Z{X{"Suzy"}})*

*}*

### 18.5 Lời khuyên (Advice)

[1] Xác định các toán tử chủ yếu để bắt chước cách sử dụng thông thường; §18.1.

[2] Xác định lại hoặc cấm sao chép nếu mặc định không phù hợp với một loại; §18.2.2.

[3] Đối với các toán hạng lớn, hãy sử dụng kiểu đối số tham chiếu const; §18.2.4.

[4] Để có kết quả lớn, hãy sử dụng hàm tạo di chuyển; §18.2.4.

[5] Ưu tiên các chức năng thành viên hơn các thành viên không phải thành viên đối với các hoạt động cần quyền truy cập vào đại diện-sự; §18.3.1.

[6] Ưu tiên các chức năng nonmember hơn các thành viên cho các hoạt động không cần quyền truy cập vào đại diện sự phẫn uất; §18.3.2.

[7] Sử dụng không gian tên để liên kết các hàm trợ giúp với lớp ‘‘ their ’’; §18.2.5.

[8] Sử dụng các hàm không nhớ cho các toán tử đối xứng; §18.3.2.

[9] Sử dụng các hàm thành viên để biểu thị các toán tử yêu cầu giá trị làm toán hạng bên trái của chúng;§18.3.3.1.

[10] Sử dụng các ký tự do người dùng xác định để bắt chước ký hiệu thông thường; §18.3.4.

[11] Cung cấp ‘‘ các hàm set () và get () ’’ cho một thành viên dữ liệu chỉ khi ngữ nghĩa cơ bản của một lớp yêu cầu họ; §18.3.5.

[12] Thận trọng về việc giới thiệu các chuyển đổi ngầm; §18.4.

[13] Tránh chuyển đổi phá hủy giá trị (‘‘ thu hẹp ’’); §18.4.1.

[14] Không định nghĩa chuyển đổi giống như cả hàm tạo và toán tử chuyển đổi; §18.4.3.

## Chap 19: Nhà điều hành đặc biệt

### 19.1 Giới thiệu (Introduction)

Nạp chồng không chỉ dành cho các phép toán số học và logic. Trên thực tế, các nhà khai thác đóng vai trò quan trọng trong việcthiết kế vùng chứa (ví dụ: vectơ và bản đồ;, '' con trỏ thông minh '' trình vòng lặp và các lớp khác liên quan đến quản lý tài nguyên.

### 19.2 Nhà điều hành đặc biệt (Special Operators)

Các nhà khai thác [] () -> ++ −− xóa mới chỉ đặc biệt ở chỗ ánh xạ từ việc sử dụng chúng trong mã đến định nghĩa của lập trình viên khác nhau một chút từ đó được sử dụng cho các toán tử đơn phân và nhị phân thông thường, chẳng hạn như +, <, end . Các toán tử [] (chỉ số con) và () (cuộc gọi) là một trong những toán tử hữu ích nhất do người dùng định nghĩa.

#### 19.2.1 Đăng ký (Subscripting):

Một hàm toán tử [] có thể được sử dụng để cung cấp ý nghĩa cho các chỉ số con cho các đối tượng lớp. Lập luận thứ hai-ment (chỉ số con) của một hàm toán tử [] có thể thuộc bất kỳ kiểu nào. Điều này làm cho nó có thể xác định vectơ, mảng kết hợp, v.v.

Ví dụ, chúng ta có thể xác định một kiểu mảng kết hợp đơn giản như sau:

*struct PGS {*

*vector<pair<string,int>> vec; // vectơ của các cặp {name, value}*

*const int& operator[] (const string&) const;*

*int& operator[](const string&);*

*};*

Một toán tử [] () phải là một hàm thành viên không tĩnh.

#### 19.2.2 Gọi hàm (Function Call)

Lời gọi hàm, nghĩa là, biểu thức ký hiệu (biểu thức-danh sách), có thể được hiểu như một toán hạng nhị phân-tion với biểu thức là toán hạng bên trái và danh sách biểu thức là toán hạng bên phải.Toán tử cuộc gọi, (), có thể được nạp chồng theo cách giống như các toán tử khác có thể:

*struct Action {*

*int operator () (int);*

*pair<int, int> operator () (int,int);*

*double operator () (double);*

*};*

*void f (Action)*

*{*

*int x = act (2);*

*auto y = act (3,4);*

*double z = act (2.3);*

*};*

Các cách sử dụng phổ biến khác của operator () () là một toán tử chuỗi con và như một toán tử chỉ số con cho mảng nhiều chiều

Một toán tử () () phải là một hàm thành viên không tĩnh.

Các toán tử gọi hàm thường là các mẫu .

#### 19.2.3 Tham khảo ý kiến (Dereferencing)

Toán tử hội nghị truyền hình, -> (còn được gọi là toán tử mũi tên), có thể được định nghĩa là một sau-sửa chữa nhà điều hành. Ví dụ:

*class Ptr {*

*// ...*

*X∗ operator−>();*

*};*

Các đối tượng của lớp Ptr có thể được sử dụng để truy cập các thành viên của lớp X theo cách rất giống với cách con trỏ được sử dụng. Ví dụ:

*void f(Ptr p)*

*{*

*p−>m = 7; // (p.operator->())->m = 7*

*}*

Việc biến đổi đối tượng p thành con trỏ p.operator -> () không phụ thuộc vào thành viên m chỉ vào. Đó là nghĩa mà toán tử -> () là một toán tử hậu tố một ngôi. Tuy nhiên, không có cú pháp mới được giới thiệu, vì vậy tên thành viên vẫn được yêu cầu sau dấu ->. Ví dụ:

*void g(Ptr p)*

*{*

*X∗ q1 = p−>; // lỗi cú pháp*

*X∗ q2 = p.operator -> (); // ok*

*}*

#### 19.2.4 Tăng và giảm (Increment and Decrement)

Khi mọi người phát minh ra '' con trỏ thông minh '', họ thường quyết định cung cấp toán tử gia tăng ++ và toán tử giảm −− để phản ánh việc sử dụng các toán tử này cho các kiểu dựng sẵn. Điều này đặc biệt quan trọng- ous và cần thiết khi mục đích là thay thế loại con trỏ thông thường bằng loại '' con trỏ thông minh '' có cùng ngữ nghĩa, ngoại trừ việc nó thêm một chút kiểm tra lỗi thời gian chạy. Ví dụ, con-sider một chương trình truyền thống rắc rối:

*void f1(X a) // sử dụng truyền thống*

*{*

*X v[200];*

*X∗ p = &v[0];*

*p−−;*

*∗p = a; // oops: p out of range, uncaught*

*++p;*

*∗p = a; // OK*

*}*

Các toán tử tăng và giảm là duy nhất trong số các toán tử C ++ ở chỗ chúng có thể được sử dụng như cả hai toán tử tiền tố và hậu tố.

#### 19.2.5 Phân bổ và Giao dịch (Allocation and Deallocation)

Toán tử new lấy bộ nhớ của nó bằng cách gọi một toán tử new (). Tương tự, toán tử xóa giải phóng bộ nhớ của nó bằng cách gọi một toán tử delete (). Người dùng có thể xác định lại toán tử toàn cầu new () và toán tử xóa () hoặc xác định toán tử new () và toán tử xóa () cho một lớp cụ thể.

Sử dụng bí danh kiểu thư viện tiêu chuẩn size\_t cho các kích thước, khai báo của ver-sions trông như thế này:

*void∗ operator new(siz e\_t); // sử dụng cho từng đối tượng*

*void∗ operator new[](siz e\_t); // sử dụng cho mảng*

*void operator delete(void∗, siz e\_t); // sử dụng cho từng đối tượng*

*void operator delete[](void∗, siz e\_t); // sử dụng cho mảng*

#### 19.2.6 Chữ viết do người dùng xác định (User-defined Literals)

C ++ cung cấp các ký tự cho nhiều kiểu tích hợp sẵn;

*123 // int*

*1.2 // double*

*1.2F // float*

*'a' // char*

*1ULL // unsigned long long*

*0xD0 // hexadecimal unsigned*

*"as" // C-style string (const char[3])*

Có bốn loại ký tự có thể được thêm hậu tố để tạo thành một ký tự do người dùng xác định:

• Một ký tự số nguyên (§6.2.4.1): được chấp nhận bởi toán tử ký tự lấy một ký tự dài không dấu hoặc một đối số const char ∗ hoặc bởi một toán tử mẫu, ví dụ: 123m hoặc 12345678901234567890X

• Một ký tự dấu phẩy động: được chấp nhận bởi toán tử ký tự lấy một dấu kép dài hoặc một đối số const char ∗ hoặc bởi một toán tử mẫu, chẳng hạn,12345678901234567890,976543210x hoặc 3,99 giây.

• Một chuỗi ký tự: được chấp nhận bởi một toán tử ký tự nhận một cặp (const char ∗, size\_t) của các đối số, ví dụ: "string" s và R "(Foo \ bar)" \_ đường dẫn.

• Một ký tự chữ: được chấp nhận bởi toán tử ký tự lấy đối số ký tự là nhập char, wchar\_t, char16\_t hoặc char32\_t, ví dụ: 'f'\_runic hoặc u'BEEF'\_w.

### 19.3 Một lớp chuỗi (A String Class)

Lớp chuỗi tương đối đơn giản được trình bày trong phần này minh họa một số kỹ thuật sử dụng-cho việc thiết kế và triển khai các lớp bằng cách sử dụng các toán tử được xác định theo quy ước.

Chuỗi là phiên bản đơn giản hóa của chuỗi thư viện chuẩn. Chuỗi cung cấp giá trị ngữ nghĩa, quyền truy cập đã chọn và chưa kiểm tra vào các ký tự, luồng I / O, hỗ trợ các vòng lặp phạm vi cho, các phép toán bình đẳng và các toán tử nối. Tôi cũng đã thêm một ký tự String, std :: string không (chưa) có.

#### 19.3.1 Hoạt động cần thiết (Essential Operations)

Chuỗi lớp cung cấp tập hợp thông thường của các hàm tạo, một hàm hủy và các hoạt động gán

*class String {*

*public:*

*String(); // hàm tạo mặc định: x {""}*

*explicit String(const char∗ p); // hàm tạo từ C-style string: x {"Euler"}*

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

*String& operator=(const String&); // sao chép nhiệm vụ*

*String(String&& x); // di chuyển hàm tạo*

*String& operator=(String&& x); // chuyển nhiệm vụ*

*̃String() { if (short\_max<sz) delete[] ptr; } // hàm hủy*

*// ...*

*};*

#### 19.3.2 Quyền truy cập vào các ký tự (Access to Characters)

Việc thiết kế các toán tử truy cập cho một chuỗi là một chủ đề khó vì truy cập lý tưởng là bằng cách kết hợp ký hiệu màu (nghĩa là, sử dụng []), hiệu quả tối đa và đã kiểm tra phạm vi. Thật không may, bạn có thể không có tất cả các thuộc tính này đồng thời. Ở đây, tôi tuân theo thư viện tiêu chuẩn bằng cách cung cấp các hoạt động không được kiểm tra hiệu quả với ký hiệu chỉ số con [] thông thường cộng với dải ô được kiểm tra tại () hoạt động:

*class String {*

*public:*

*// ...*

*char&operator[](int n) { return ptr[n]; } // quyền truy cập phần tử không được chọn*

*char operator[](int n) const { return ptr[n]; }*

*char&at(int n) { check(n); return ptr[n]; } // quyền truy cập phần tử đã kiểm tra phạm vi*

*char at(int n) const { check(n); return ptr[n]; }*

*String& operator+=(char c); // thêm c vào cuối*

*const char∗ c\_str() { return ptr; } // Truy cập chuỗi kiểu C*

*const char∗ c\_str() const { return ptr; }*

*int size() const { return sz; } // số phần tử*

*int capacity() const // phần tử cộng với không gian có sẵn*

*{ return (sz<=short\_max) ? short\_max : sz+space; }*

*// ...*

*};*

#### 19.3.3 Đại diện (Representation)

Biểu diễn cho Chuỗi được chọn để đáp ứng ba mục tiêu:

• Để dễ dàng chuyển đổi chuỗi kiểu C (ví dụ: chuỗi ký tự) thành Chuỗi và cho phép dễ dàng truy cập vào các ký tự của một chuỗi dưới dạng một chuỗi kiểu C.

• Để giảm thiểu việc sử dụng cửa hàng miễn phí.

• Để thêm các ký tự vào cuối một Chuỗi hiệu quả.

Một chuỗi đơn giản thực hiện tối ưu hóa string ngắn.

Để cho phép bổ sung hiệu quả các ký tự vào cuối, Chuỗi phát triển bằng cách tăng gấp đôi phân bổ của nó; dung lượng () là lượng không gian có sẵn cho các ký tự.

Điều này hỗ trợ những gì được gọi là tối ưu hóa chuỗi ngắn bằng cách sử dụng hai biểu diễn chuỗi:

• Nếu sz <= short\_max, các ký tự được lưu trữ trong chính đối tượng String, trong mảng có tên ch.

• Nếu! (Sz <= short\_max), các ký tự được lưu trữ trên cửa hàng miễn phí và chúng tôi có thể phân bổ thêm không gian để mở rộng. Thành viên được đặt tên khoảng trắng là số ký tự như vậy.

##### 19.3.3.1 Các chức năng phụ trợ (Ancillary Functions)

Ngoài các chức năng dành cho mục đích sử dụng chung, tôi nhận thấy rằng mã của mình trở nên sạch hơn khi tôi cung cấp cung cấp ba chức năng phụ trợ như là '' các khối xây dựng '' để giúp tôi giải quyết vấn đề có phần phức tạp-tation và để giảm thiểu sự sao chép mã. Hai trong số đó cần truy cập vào biểu diễn của Chuỗi,vì vậy tôi đã biến họ thành thành viên. Tuy nhiên, tôi đã đặt họ là thành viên riêng tư vì họ không đại diện cho các hoạt động thường hữu ích và an toàn khi sử dụng. Đối với nhiều lớp học thú vị, triển khai-tion không chỉ là đại diện cộng với các chức năng công cộng. Các chức năng phụ trợ có thể dẫn đến ít sao chép mã, thiết kế tốt hơn và khả năng bảo trì được cải thiện.

Hàm đầu tiên như vậy di chuyển các ký tự vào bộ nhớ mới được cấp phát:

*char∗ expand(const char∗ ptr, int n) // mở rộng thành cửa hàng miễn phí*

*{*

*char∗ p = new char[n];*

*strcpy(p,ptr);*

*return p;*

*}*

Hàm này không truy cập vào biểu diễn chuỗi, vì vậy tôi không đặt nó làm thành viên.

Chức năng triển khai thứ hai được sử dụng bởi các hoạt động sao chép để cung cấp cho một Chuỗi một bản sao của thành viên của người khác.

Chức năng tương ứng cho các hoạt động di chuyển.

#### 19.3.4 Chức năng thành viên (Member Functions)

Hàm tạo mặc định xác định một Chuỗi trống:

*String::String() // hàm tạo mặc định: x {""}*

*: sz{0}, ptr{ch} // ptr trỏ đến các phần tử, ch là vị trí ban đầu*

*{*

*ch[0] = 0; // kết thúc 0*

*}*

Nếu đối số là một chuỗi ngắn, ptr được đặt để trỏ đến ch; nếu không, không gian được phân bổ trên miễn phí cửa hàng. Trong cả hai trường hợp, các ký tự được sao chép từ chuỗi đối số vào bộ nhớ được quản lý bởi String.

Hàm tạo sao chép chỉ cần sao chép biểu diễn của các đối số của nó.

Tương tự, hàm tạo di chuyển di chuyển biểu diễn từ nguồn của nó (và có thể đặt nó đối số là chuỗi trống).

Giống như hàm tạo bản sao, phép gán bản sao sử dụng copy\_from () để sao chép đại diện của đối số của nó-sự. Ngoài ra, nó phải xóa bất kỳ cửa hàng miễn phí nào thuộc sở hữu của mục tiêu và đảm bảo rằng nó không bị gặp rắc rối với việc tự phân công (ví dụ: s = s).

Phép toán Chuỗi phức tạp nhất về mặt logic là + =, thêm một ký tự vào cuối chuỗi, tăng kích thước của nó lên một.

Có rất nhiều điều đang xảy ra ở đây: toán tử + = () phải theo dõi xem đại diện nào (ngắn hay dài) là đã sử dụng và liệu có thêm dung lượng để mở rộng hay không. Nếu cần thêm dung lượng, thì expand () là được gọi để phân bổ không gian đó và di chuyển các ký tự cũ vào không gian mới. Nếu có một cũ phân bổ cần xóa, nó được trả về, để + = có thể xóa nó. Sau khi có đủ dung lượng-có thể, việc đặt ký tự mới c vào đó và thêm số 0 tận cùng là điều nhỏ nhặt.

Lưu ý tính toán bộ nhớ khả dụng cho không gian. Trong số tất cả quá trình triển khai Chuỗi đã diễn ra lâu nhất để làm đúng: một phép tính nhỏ lộn xộn dễ mắc lỗi từng người một. Câu hỏi lặp đi lặp lại đó-stant 2 có cảm giác giống như một "hằng số ma thuật".

Tất cả các thành viên của Chuỗi lưu ý không sửa đổi đại diện mới trước khi họ chắc chắn rằng

cái mới có thể được đưa vào vị trí. Đặc biệt, họ không xóa cho đến khi có bất kỳ hoạt động mới nào có thể xảy ra đã hoàn thành. Trên thực tế, các thành viên Chuỗi cung cấp bảo đảm ngoại lệ mạnh mẽ.

Nếu bạn không thích loại mã khó hiểu được trình bày như một phần của quá trình triển khai Chuỗi, hãy mô phỏng ply sử dụng std :: string. Ở một mức độ lớn, các cơ sở thư viện tiêu chuẩn tồn tại để cứu chúng ta khỏi chương trình-ming ở mức lev thấp này hầu hết thời gian. Mạnh hơn: viết một lớp chuỗi, một lớp vectơ hoặc một bản đồ là một bài tập tuyệt vời. Tuy nhiên, một khi bài tập được thực hiện, một kết quả sẽ là một sự đánh giá cao về những gì tiêu chuẩn cung cấp và mong muốn không duy trì phiên bản của riêng bạn.

#### 19.3.5 Chức năng của người trợ giúp (Helper Functions)

Để hoàn thành Chuỗi lớp, tôi cung cấp một tập hợp các chức năng hữu ích, luồng I / O, hỗ trợ các vòng lặp phạm vi cho,so sánh và nối. Tất cả những điều này phản ánh các lựa chọn thiết kế được sử dụng cho std :: string. Cụ thể là, << chỉ in các ký tự mà không cần thêm định dạng và >> bỏ qua khoảng trắng ban đầu trước đó đọc cho đến khi nó tìm thấy khoảng trắng kết thúc (hoặc cuối luồng):

*ostream& operator<<(ostream& os, const String& s)*

*{*

*return os << s.c\_str(); // §36.3.3*

*}*

*istream& operator>>(istream& is, String& s)*

*{*

*s = ""; // xóa chuỗi đích*

*is>>ws; // bỏ qua khoảng trắng (§38.4.5.1)*

*char ch = ' ';*

*while(is.get(ch) && !isspace(ch))*

*s += ch;*

*return is;*

*}*

Để hỗ trợ vòng lặp range-for, chúng ta cần begin () và end (). Một lần nữa, chúng tôi có thể cung cấp những dưới dạng các hàm tự do (nonmember) mà không cần truy cập trực tiếp vào việc triển khai Chuỗi.

Với hàm thành viên + = thêm một ký tự vào cuối, các toán tử nối dễ dàng được cung cấp dưới dạng các hàm khác.

Việc sử dụng + = này được hiểu là toán tử + = (s, String ("Gunnar")). Tôi đoán là tôi có thể cung cấp một hiệu quả hơn String :: operator + = (const char ∗), nhưng tôi không biết liệu hiệu suất được thêm vào có đáng giá trong thế giới thực mã. Trong những trường hợp như vậy, tôi cố gắng tiết chế và đưa ra mức tối thiểu thiết kế. Có thể làm một điều gì đó không phải là một lý do chính đáng để làm điều đó.

#### 19.3.6 Sử dụng chuỗi của chúng tôi (Using Our String)

Chương trình chính chỉ đơn giản là thực hành các toán tử chuỗi một chút:

*int main()*

*{*

*String s ("abcdefghij");*

*cout << s << '\n';*

*s += 'k';*

*s += 'l';*

*s += 'm';*

*s += 'n';*

*cout << s << '\n';*

*String s2 = "Hell";*

*s2 += " and high water";*

*cout << s2 << '\n';*

*String s3 = "qwerty";*

*s3 = s3;*

*String s4 ="the quick bro wn fox jumped over the lazy dog";*

*s4 = s4;*

*cout << s3 << " " << s4 << "\n";*

*cout << s + ". " + s3 + String(". ") + "Horsefeathers\n";*

*String buf;*

*while (cin>>buf && buf!="quit")*

*cout << buf << " " << buf.siz e() << " " << buf.capacity() << '\n';*

*}*

Chuỗi này thiếu nhiều tính năng mà bạn có thể coi là quan trọng hoặc thậm chí là cần thiết. Tuy nhiên, đối với những gì nó thực hiện nó gần giống với std :: string (Chương 36) và minh họa các kỹ thuật được sử dụng cho thực hiện chuỗi thư viện tiêu chuẩn.

### 19.4 Bạn (Friends)

Một khai báo hàm thành viên bình thường chỉ định ba điều khác biệt về mặt logic:

[1] Hàm có thể truy cập phần riêng của khai báo lớp.

[2] Hàm thuộc phạm vi của lớp.

[3] Hàm phải được gọi trên một đối tượng (có con trỏ this).

Ví dụ, chúng ta có thể xác định một toán tử nhân ma trận bằng một vector. Một cách thông thường, vector và ma trận ẩn các biểu diễn tương ứng của chúng và cung cấp một tập hợp các thao tác hoàn chỉnh để thao tác các đối tượng cùng loại. Tuy nhiên, thói quen nhân của chúng ta không thể là thành viên của cả hai. Ngoài ra, chúng ta không thực sự muốn cung cấp các chức năng truy cập cấp thấp để cho phép mọi người dùng vừa đọc và viết bản trình bày hoàn chỉnh của cả hai ma trậnvà vector. Để tránh điều này, chúng tôi khai báo toán tử là bạn của cả hai:

*constexpr rc\_max {4}; // kích thước của hàng và cột*

*class Matrix{*

*Vector v[rc\_ max};*

*friend Vector operator\*(const Matrix&, const Vector&); };*

*Class Vector{*

*float v[rc\_max];*

*friend Vector operator\*(const Matrix&, const Vector&); };*

#### 19.4.1 Tìm bạn bè (Finding Friends)

Một người bạn phải được khai báo trước đó trong một phạm vi bao quanh hoặc được xác định trong phạm vi không thuộc lớp bao gồm bao quanh một cách ẩn lớp đang tuyên bố nó là một người bạn. Phạm vi bên ngoài vỏ bọc trong cùng phạm vi không gian tên không được coi là tên được khai báo đầu tiên là bạn .Hãy xem xét một ví dụ kỹ thuật:

*class C1 { }; // sẽ trở thành bạn của N :: C*

*void f1(); // sẽ trở thành bạn của N :: C*

*namespace N {*

*class C2 { }; // sẽ trở thành bạn của C*

*void f2() { } // sẽ trở thành bạn của C*

*class C {*

*int x;*

*public:*

*friend class C1; // OK (đã xác định trước đó)*

*friend void f1();*

*friend class C3; // OK (được xác định trong không gian tên bao quanh)*

*friend void f3();*

*friend class C4; // Được khai báo đầu tiên bằng N và giả sử là N*

*friend void f4();*

*};*

*class C3 {}; // bạn của C*

*void f3() { C x; x.x = 1; } // OK: bạn của C*

*} // không gian tên N*

*class C4 { }; // không phải bạn của N :: C*

*void f4() { N::C x; x.x = 1; } // error: x là private và f4 () không phải là bạn của N :: C*

Một hàm friend có thể được tìm thấy thông qua các đối số của nó ngay cả khi nó không được khai báo trong phạm vi bao quanh ngay lập tức.

Do đó, một hàm friend nên được khai báo rõ ràng trong một phạm vi bao quanh hoặc lấy một đối số là lớp của nó hoặc một lớp có nguồn gốc từ đó. Nếu không, người bạn không thể được gọi.

#### 19.4.2 Bạn bè và thành viên (Friends and Members)

Khi nào chúng ta nên sử dụng hàm kết bạn và khi nào thì một hàm thành viên là lựa chọn tốt hơn cho đặc điểm-làm hỏng một cuộc phẫu thuật? Đầu tiên, chúng tôi cố gắng giảm thiểu số lượng hàm truy cập vào biểu tượng-của một lớp và cố gắng làm cho tập hợp các chức năng truy cập phù hợp nhất có thể. Do đó,câu hỏi đầu tiên không phải là ‘‘ Nên là thành viên, thành viên tĩnh hay là bạn bè? ’’ mà là ‘‘ Có phải không thực sự cần quyền truy cập? ’’ Thông thường, tập hợp các chức năng cần quyền truy cập nhỏ hơn mức chúng tôi sẵn sàng để tin tưởng lúc đầu. Một số hoạt động phải là thành viên - ví dụ: hàm tạo, hàm hủy và các hàm ảo - nhưng thường có một sự lựa chọn. Vì tên thành viên là cục bộ cho lớp, một hàm yêu cầu quyền truy cập trực tiếp vào biểu diễn phải là một thành viên trừ khi có một lý do cụ thể để nó không phải là một thành viên.

Hãy xem xét một lớp X cung cấp các cách thay thế để trình bày một phép toán:

*class X {*

*// ...*

*X(int);*

*int m1(); // thành viên*

*int m2() const;*

*friend int f1(X&); // bạn bè, không phải thành viên*

*friend int f2(const X&);*

*friend int f3(X);*

*};*

### 19.5 Lời khuyên (Advice)

[1] Sử dụng toán tử [] () để lập chỉ số và để lựa chọn dựa trên một giá trị duy nhất; §19.2.1.

[2] Sử dụng toán tử () () cho ngữ nghĩa cuộc gọi, để lập chỉ mục và để lựa chọn dựa trên nhiều giá trị NS; §19.2.2.

[3] Sử dụng toán tử -> () để truy cập ‘‘ con trỏ thông minh ’’; §19.2.3.

[4] Ưu tiên tiền tố ++ hơn hậu tố ++; §19.2.4.

[5] Chỉ định nghĩa toán tử toàn cục new () và toán tử delete () nếu bạn thực sự phải làm như vậy; §19.2.5.

[6] Xác định toán tử thành viên new () và toán tử thành viên delete () để kiểm soát việc cấp phát và hủy bỏ phân bổ các đối tượng của một lớp cụ thể hoặc hệ thống phân cấp của các lớp; §19.2.5.

[7] Sử dụng các ký tự do người dùng xác định để bắt chước ký hiệu thông thường; §19.2.6.

[8] Đặt các toán tử chữ trong các không gian tên riêng biệt để cho phép sử dụng có chọn lọc; §19.2.6.

[9] Đối với các mục đích sử dụng không chuyên biệt, hãy ưu tiên chuỗi tiêu chuẩn (Chương 36) hơn là kết quả của riêng bạn bài tập; §19.3.

[10] Sử dụng hàm bạn bè nếu bạn cần một hàm nonmember để có quyền truy cập vào phần biểu diễn của một lớp (ví dụ, để cải thiện ký hiệu hoặc để truy cập vào biểu diễn của hai lớp); §19.4.

[11] Ưu tiên các chức năng thành viên hơn các chức năng bạn bè để cấp quyền truy cập vào việc triển khai lớp; §19.4.2.

## Chap 20: Các lớp có nguồn gốc

### 20.1 Giới thiệu (Introduction)

• Các lớp có nguồn gốc

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

• Phân cấp lớp

Nhập trường; Chức năng ảo; Trình độ chuyên môn rõ ràng; Kiểm soát Ghi đè; sử dụng Base Mem-các loại bia; Loại trả lại Thư giãn

• Các lớp trừu tượng

• Kiểm soát truy cập thành viên được bảo vệ; Quyền truy cập vào các lớp cơ sở; using-Khai báo và Kiểm soát Truy cập

• Con trỏ đến thành viên

Con trỏ đến các thành viên chức năng; Con trỏ tới Thành viên Dữ liệu; Thành viên gốc và thành viên có nguồn gốc

• Lời khuyên

### 20.1 Giới thiệu

Từ Simula, C ++ đã vay mượn ý tưởng của các lớp và cấu trúc phân cấp lớp. Sử dụngcấu trúc ngôn ngữ chỉ là đạo cụ ký hiệu cho các kiểu lập trình truyền thống là bỏ sót khóa điểm mạnh của C ++.

Một khái niệm không tồn tại một cách cô lập. Nó cùng tồn tại với các khái niệm liên quan và phần lớn sức mạnh của nó có được từ các mối quan hệ với các khái niệm khác. Vì chúng tôi sử dụng các lớp để thể hiện các khái niệm.

Vấn đề trở thành cách thể hiện mối quan hệ giữa các khái niệm. Tuy nhiên, chúng tôi không thể diễn đạt arbi- theo dõi các mối quan hệ trực tiếp trong một ngôn ngữ lập trình. Ngay cả khi chúng tôi có thể, chúng tôi sẽ không muốn. Đến hữu ích, các lớp học của chúng ta nên được định nghĩa hẹp hơn so với các khái niệm hàng ngày của chúng ta - và nhiều hơn trước cise.

Khái niệm về một lớp dẫn xuất và các cơ chế ngôn ngữ liên quan của nó được cung cấp để diễn đạt quan hệ thứ bậc, nghĩa là, để thể hiện tính chung giữa các lớp. Ví dụ, con-cepts của một hình tròn và một hình tam giác có liên quan ở chỗ cả hai đều là hình dạng; đó là, họ có khái niệm của một hình dạng chung. Do đó, chúng ta xác định rõ ràng lớp Circle và lớp Triangle để có lớp Hình dạng điểm chung. Trong trường hợp đó, lớp chung, ở đây Hình dạng, được gọi là lớp cơ sở hoặc siêu lớp và các lớp dẫn xuất từ ​​đó, ở đây Circle và Triangle, được gọi là các lớp dẫn xuất hoặc các lớp con. Biểu diễn hình tròn và hình tam giác trong một chương trình mà không liên quan đến khái niệm hình dạng sẽ là bỏ lỡ một cái gì đó cần thiết. Chương này là một khám phá về các tác động của ý tưởng đơn giản này, là cơ sở cho cái thường được gọi là lập trình hướng đối tượng. Các tính năng ngôn ngữ hỗ trợ xây dựng các lớp mới từ những lớp hiện có:

• Kế thừa triển khai: để tiết kiệm nỗ lực thực hiện bằng cách chia sẻ các phương tiện 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, việc sử dụng thống nhất các lớp không liên quan đến tính kế thừa được cung cấp bởi các mẫu.

• Các lớp có nguồn gốc: Chương này giới thiệu các tính năng cơ bản của ngôn ngữ hỗ trợ lập trình hướng đối tượng. Các lớp cơ sở và dẫn xuất, các chương này mô tả các kỹ thuật để giải thích-itly điều hướng phân cấp lớp. Đặc biệt, các hoạt động chuyển đổi loại dynamic\_cast và static\_cast được trình bày, cũng như hoạt động để xác định loại đối tượng đã cho một trong các lớp cơ sở của nó (typeid).

3: Các lớp cơ sở và dẫn xuất và các hàm ảo. Các chương này kiểm tra những các tính năng cơ bản và các kỹ thuật thiết kế và lập trình liên quan của chúng một cách chi tiết hơn.

### 20.2 Các lớp có nguồn gốc (Derived Classes)

Cân nhắc xây dựng một chương trình giao dịch với những người được một công ty tuyển dụng. Một chương trình như vậy có thể có một cấu trúc dữ liệu như thế :

*struct Nhân viên {*

*string first\_name, family\_name;*

*char middle\_initial;*

*Ngày\_công\_dụng;*

*ngắn depar tment;*

*// ...*

*};*

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

trình quản lý cấu trúc {

Nhân viên trống; // hồ sơ nhân viên của người quản lý

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

mức độ ngắn;

// ...

};

Một nhà quản lý cũng là một nhân viên; người sử dụng lao động dữ liệu ee được lưu trữ trong thành viên trống của Người quản lý sự vật. Điều này có thể rõ ràng đối với một độc giả con người - đặc biệt là một độc giả cẩn thận - nhưng không có gì điều đó cho trình biên dịch và các công cụ khác biết rằng Người quản lý cũng là Nhân viên. Người quản lý ∗ không phải là Nhân viên ∗, vì vậy người ta không thể đơn giản sử dụng cái này khi cái kia được yêu cầu. Đặc biệt, người ta không thể đặt một Người quản lý vào danh sách Nhân viên mà không cần viết mã đặc biệt. Chúng tôi có thể sử dụng loại rõ ràng chuyển đổi trên Người quản lý ∗ hoặc đưa địa chỉ của thành viên trống vào danh sách nhân viên. Tuy vậy,cả hai giải pháp đều không phù hợp và có thể khá tối nghĩa. Cách tiếp cận đúng là trình bày rõ ràng rằng Người quản lý là Nhân viên, với một số thông tin được thêm vào:

*struct Manager: public Employee {*

*danh sách <Nhân viên ∗> nhóm;*

*mức độ ngắn;*

*// ...*

*};*

Một lớp dẫn xuất thường được cho là kế thừa các thuộc tính từ cơ sở của nó, vì vậy mối quan hệ còn được gọi là di sản. Một lớp cơ sở đôi khi được gọi là lớp cha và lớp dẫn xuất là lớp con. Tuy nhiên, gây nhầm lẫn cho những người quan sát rằng dữ liệu trong một đối tượng lớp dẫn xuất là 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.

Một cách triển khai phổ biến và hiệu quả:

*tên đầu tiên*

*tên gia đình*

*...*

*tên đầu tiên*

*tên gia đình*

*...*

*tập đoàn*

*cấp độ*

*...*

*Nhân viên: Quản lý:*

Không có chi phí bộ nhớ được ngụ ý bằng cách dẫn xuất một lớp. Không gian cần thiết chỉ là không gian cần thiết bởi các thành viên.

Xuất phát từ Người quản lý từ Nhân viên theo cách này làm cho Người quản lý trở thành một loại phụ của Nhân viên, do đó Người quản lý có thể được sử dụng ở bất cứ nơi nào Nhân viên có thể chấp nhận được. Ví dụ: bây giờ chúng ta có thể tạo một danh sách. Nhân viên, một số người trong số họ là Quản lý:

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

*{*

*list <Employee ∗> elist {& m1, & e1);*

*}*

Sử dụng một lớp làm cơ sở tương đương với việc xác định một đối tượng (không tên) của lớp đó.

Nhược điểm: do đó một lớp phải được xác định để được sử dụng làm cơ sở nhân viên lớp học; // chỉ khai báo, không có định nghĩa:

*class Manager: public Employee {// error: Nhân viên không được xác định*

*// ...*

*};*

#### 20.2.1 Chức năng thành viên (Member Functions)

Ví dụ:

*nhân viên lớp {*

*public:*

*void print () const;*

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

*// ...*

*private:*

*string first\_name, family\_name;*

*char middle\_initial;*

*// ...*

*};*

*quản lý lớp: nhân viên công cộng {*

*public:*

*void print () const;*

*// ...*

*};*

Thành viên của một lớp dẫn xuất có thể sử dụng công khai - và được bảo vệ

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'; // erro*

*NS!*

*// ...*

*}*

Phiên bản thứ hai này của *Manager :: print ()* sẽ không biên dịch vì không thể truy cập vào family\_name

*Manager :: print ()*

Điều này gây ngạc nhiên cho một số người, nhưng hãy xem xét phương án thay thế: rằng một chức năng thành viên của một lớp dẫn xuất có thể truy cập các thành viên riêng của lớp cơ sở của nó. Khái niệm về một thành viên riêng sẽ trở nên vô nghĩa nếu cho phép một lập trình viên có quyền truy cập vào phần riêng tư của lớp đơn giản bằng cách dẫn xuất một lớp mới từ nó. Hơn nữa, người ta không còn có thể tìm thấy tất cả các công dụng của một tên riêng bằng cách xem các hàm được khai báo là thành viên và bạn bè của lớp đó. Một người sẽ phải kiểm tra mọi tệp nguồn của chương trình hoàn chỉnh cho các lớp dẫn xuất, sau đó kiểm tra mọi chức năng của các lớp đó, sau đó tìm mọi lớp dẫn xuất từ ​​các lớp đó, v.v. Đây là, tốt nhất, tẻ nhạt và thường không thực tế. Nơi nó được chấp nhận, được bảo vệ - thay vì riêng tư - các thành viên có thể được sử dụng. Thông thường, giải pháp sạch nhất là dành cho lớp dẫn xuất chỉ sử dụng các thành viên công khai của lớp cơ sở. Ví dụ:

*void Manager :: print () const*

*{*

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

*cout << cấp; // thông tin cụ thể của print Manager*

*// ...*

*}*

Lưu ý rằng :: phải được sử dụng vì print () đã được định nghĩa lại trong Trình quản lý. Việc sử dụng lại tên như vậy là kiểu … ical. Người không cẩn thận có thể viết thế này:

*void Manager :: print () const*

*{*

*in(); // Giáo sư!*

*// 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 (Constructors and Destructors)

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

• Độ phân giải của một lệnh gọi hàm ảo, một dynamic\_cast hoặc một typeid () trong một hàm tạo hoặc để cấu trúc phản ánh giai đoạn xây dựng và phá hủy (chứ không phải là loại đối tượng được hoàn thành);

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 này.

### 20.3 Phân cấp lớp (Class Hierarchies)

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:

*Temporar y nhân viên*

*Nhân viên bán thời gian*

*Trợ lý giám đốc*

*Giám đốc Tư vấn*

#### 20.3.1 Trường loại (Type Fields)

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 vấn đề hạ thấp: Cho một con trỏ kiểu Cơ sở ∗, đối tượng được trỏ tới kiểu dẫn xuất nào thực sự thuộc về? 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 [4] là một giải pháp đặc biệt sự biến đổi kiểu-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 mọi tình huống, chúng mang lại mã rõ ràng hơn. giải pháp [2] và [3].

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

*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

*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 ở chỗ nó phụ thuộc vào việc lập trình viên thao tác các kiểu theo cách có thể không được kiểm tra bởi trình biên dịch. Vấn đề này thường trở nên tồi tệ hơn vì các chức năng 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';*

*// ...*

*}*

*}*

Tìm tất cả các bài kiểm tra như vậy trên trường loại 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 cái chứa các bài kiểm tra trên trường loại. Lập trình viên phải xem xét mọi chức năng có thể tưởng tượng rằng cần phải kiểm tra trường loại sau khi thay đổi. Điều này có nghĩa là cần phải truy cập mã nguồn quan trọng và kết quả là chi phí cần thiết của việc kiểm tra mã bị ảnh hưởng.

Nói cách khác, việc sử dụng một trường kiểu là một kỹ thuật dễ xảy ra lỗi dẫn đến việc bảo trì lems. Các vấn đề gia tăng mức độ nghiêm trọng khi kích thước của chương trình tăng lên do việc sử dụng trường loại gây ra vi phạm các lý tưởng của mô-đun và ẩn dữ liệu. Mỗi chức năng sử dụng một loại trường phải biết về đại diện và các chi tiết khác về việc triển khai mọi lớp bắt nguồn từ trường 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, cám dỗ mọi người thêm nhiều dữ liệu như vậy. Cơ sở chung do đó trở thành kho lưu trữ của tất cả các loại của ‘‘ thông tin hữu ích ’’. Điều này sẽ giúp triển khai các lớp cơ sở và lớp dẫn xuất đan xen theo những cách không mong muốn nhất. Trong một hệ thống phân cấp lớp lớn, có thể truy cập (không riêng tư) dữ liệu trong một lớp cơ sở chung trở thành '' biến toàn cục '' của hệ thống phân cấp. Đối với thiết kế sạch sẽ và bảo trì đơn giản hơn, chúng tôi muốn giữ các vấn đề riêng biệt tách biệt và tránh phụ thuộc lẫn nhau.

#### 20.3.2 Chức năng ảo (Virtual Functions)

Các hàm ảo khắc phục các vấn đề với giải pháp trường kiểu bằng cách cho phép người lập trình để 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:

*nhân viên lớp {*

*công cộng:*

*Nhân viên (const string & name, int dept);*

*void print () const;*

*// ...*

*riêng:*

*string first\_name, family\_name;*

*ngắn 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 và các hàm print () được định nghĩa trong các lớp dẫn xuất từ ​​nó. Các hàm print () như vậy ở đâu được định nghĩa trong các lớp dẫn xuất, trình biên dịch đảm bảo rằng print () phù hợp cho đối tượng Employee đã cho là được gọi trong mỗi trường hợp.

Để cho phép khai báo hàm ảo hoạt động như một giao diện cho các hàm được định nghĩa trong các lớp, các loại đối số được chỉ định cho một hàm trong một lớp dẫn xuất không được khác với đối số kiểu ment được khai báo trong cơ sở và chỉ cho phép những thay đổi rất nhỏ đối với kiểu trả về. 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 đầu tiên (trừ khi nó là declamàu đỏ là một hàm ảo thuần túy; xem ). Ví dụ:

*void Nhân viên :: print () const*

*{*

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

*// ...*

*}*

Một hàm ảo có thể được sử dụng ngay cả khi không có lớp nào được dẫn xuất từ ​​lớp của nó và một lớp dẫn xuất không cần phiên bản riêng của một chức năng ảo không cần cung cấp một. Khi dẫn xuất một lớp, chỉ cần cung cấp một chức năng thích hợp nếu nó là cần thiết. Ví dụ:

*quản lý lớp: nhân viên công cộng {*

*public:*

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

*void print () const;*

*// ...*

*private:*

*danh sách <Nhân viên ∗> nhóm;*

*mức độ ngắn;*

*// ...*

*};*

*void Manager :: print () const*

*{*

*Nhân viên :: 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 hàm trong một 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.

Ngoại trừ trường hợp chúng tôi nói rõ ràng phiên bản của một hàm ảo nào được gọi (như trong lệnh gọi Employee :: print ()), hàm ghi đè được chọn là thích hợp nhất cho đối tượng mà nó được gọi là. Không phụ thuộc vào lớp cơ sở (giao diện) nào được sử dụng để truy cập một đối tượng, chúng ta luôn nhận được chức năng tương tự khi chúng ta sử dụng cơ chế gọi hàm ảo. Hàm toàn cục print\_employee () hiện không cần thiết vì thành viên print () các chức năng đã diễn ra. 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 ()*

*{*

*Nhân viên e {"Brown", 1234};*

*Người quản lý m {"Smith", 1234,2};*

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

*}*

*sản xuất:*

*Smith 1234*

*cấp độ 2*

*Nâu 1234*

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

*tên đầu tiên*

*Tên thứ hai*

*...*

*…*

*Nhân viên :: print ()*

*vtbl:*

*Nhân viên:*

*tên đầu tiên*

*Tên thứ hai*

*...*

*tập đoàn*

*cấp độ*

*...*

*...*

*Người quản lý :: print ()*

*vtbl:*

*Người quản lý:*

Các hàm trong vtbl cho phép đối tượng được sử dụng chính xác ngay cả khi kích thước của đối tượng và người gọi không xác định được cách bố trí dữ liệu của nó. 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. Cuộc gọi ảo này cơ chế có thể được thực hiện gần như hiệu quả như cơ chế '' lệnh gọi hàm bình thường '' (trong 25%), do đó, mối quan tâm về hiệu quả không nên ngăn cản bất kỳ ai sử dụng một hàm ảo trong đó một tổ chức lệnh gọi hàm nary sẽ hiệu quả ở mức chấp nhận được. Trên đầu 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 chỉ chọn thanh toán chi phí này nếu bạn cần chức năng bổ sung chức năng ảo cung cấp. Bạn đã chọn sử dụng solu trường loại thay thế-tion, một lượng không gian có thể so sánh được sẽ là cần thiết cho trường loại.

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

#### 20.3.3 Chứng chỉ rõ ràng (Explicit Qualification)

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*

*{*

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

*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 ứng mong muốn khác. Nghĩa là, nếu một hàm ảo cũng nội tuyến (như không có gì lạ), thì nội tuyến thay thế có thể được sử dụng cho các cuộc gọi được chỉ định bằng cách sử dụng ::. Điều này cung cấp cho lập trình viên một cách xử lý một số trường hợp đặc biệt quan trọng trong đó một hàm ảo gọi một hàm khác cho cùng một sự vật. Hàm Manager :: print () là một ví dụ về điều này. Bởi vì loại đối tượng quyết định được khai thác trong lệnh gọi của Manager :: print (), nó không cần phải được xác định động lại cho kết quả Lệnh gọi của Employee :: print ().

#### 20.3.4 Kiểm soát ghi đè (Override Control)

Nếu bạn khai báo một hàm trong một lớp dẫn xuất có cùng tên và kiểu như một hàm ảo hàm trong lớp cơ sở, sau đó 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 đảm bảo mà 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); // overr ide g () trong lớp cơ sở*

*ảo int h (); // overr ide 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 trong đó 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 loại đối số như B0 :: g (), vì vậy nếu nó ghi đè bất kỳ 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 diễn ra vì 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 chống lại những lỗi phổ biến) việc ghi đè được thực hiện đúng cách không khó. Tuy nhiên, đối với phân cấp lớn hơn- chies các điều khiển cụ thể hơn rất hữu ích:

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

• = 0: Hàm phải ảo và phải được ghi đè

• 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 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 lớp học khai báo sử dụng ghi đè cho bảy trong số chín hàm lớp cơ sở ảo có thể gây nhầm lẫncho người bảo trì.

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 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 sử dụng ảo chỉ 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*

Mã định nghĩa ghi đè không phải là một phần của loại hàm và không thể lặp lại trong một định nghĩa 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. Đó là, ghi đè có một ý nghĩa đặc biệt trong một số ngữ cảnh nhưng có thể được sử dụng làm định danh ở những nơi khác. Ví dụ:

*int override = 7;*

*struct Dx: Cơ sở {*

*ghi đè int;*

*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 mà ghi đè là từ khóa theo ngữ cảnh, chứ không phải từ khóa thông thường, có tồn tại một lượng đáng kể mã đã được sử dụng ghi đè làm mã định danh thông thường trong nhiều thập kỷ. Từ khóa theo ngữ cảnh khác là 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 của chúng tôi dựa trên ý nghĩa (ngữ nghĩa) của lớp của chúng tôi:

• Chúng ta có thể hình dung sự cần thiết của các lớp dẫn xuất tiếp theo không?

• Người thiết kế một lớp dẫn xuất có cần xác định lại hàm để đạt được mục đích hợp lý không?

• Ghi đè một hàm dễ xảy ra lỗi (tức là, rất khó để một hàm ghi đè cung cấp ngữ nghĩa mong đợi của một hàm ảo)?

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 thiết kế và đôi khi 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 có một hệ thống phân cấp lớp bắt đầu bằng các hàm ảo, nhưng sau định nghĩa inition của 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 trong đó tất cả các cấu trúc ngôn ngữ đã được xác định là cụ thể các lớp nút bắt nguồn từ một vài giao diện. Chúng ta chỉ cần lấy một lớp mới nếu chúng ta thay đổi ngôn ngữ. Trong trường hợp đó, chúng tôi có thể muốn ngăn người dùng ghi đè các chức năng ảo bởi vì điều duy nhất mà những ghi đè như vậy có thể làm là thay đổi ngữ nghĩa của ngôn ngữ của chúng ta. Ví dụ:

*struct Node {// lớp giao diện*

*kiểu virtual Type () = 0;*

*// ...*

*};*

*class If\_statement: public Node {*

*public:*

*Gõ type () ghi đè cuối cùng; // ngăn chặn việc ghi đè thêm*

*// ...*

*};*

Trong một hệ thống phân cấp giai cấp thực tế, sẽ có một số lớp trung gian giữa các face (ở đây, Node) và lớp dẫn xuất đại diện cho một cấu trúc ngôn ngữ cụ thể (ở đây, If\_state-cố vấn). Tuy nhiên, điểm mấu chốt 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 () không phải (đó là lý do tại sao nó được khai báo cuối cùng). Sau khi sử dụng cuối cùng cho một chức năng thành viên, nó không thể bị ghi đè nữa và một nỗ lực để thực hiện như vậy là một lỗi.

Ví dụ:

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

*public:*

*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. Vì 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;*

*// ...*

*};*

Thực hiện một số phép đo nghiêm túc trước khi yêu cầu cải thiện hiệu quả. Sử dụng cuối cùng, nơi nó phản ánh rõ ràng thiết kế phân cấp lớp mà bạn cho là phù hợp. Đó là, sử dụng cuối cùng để phản ánh một nhu cầu ngữ nghĩa.

Ví dụ:

*class Derived: public Base {*

*void f () cuối cùng; // OK nếu Base có f ảo ()*

*void g () cuối cùng; // OK nếu Base có g () ảo*

*// ...*

*};*

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

*{*

*// ...*

*}*

*void g () final // OK*

*{*

*// ...*

*}*

*Ví dụ:*

*int cuối cùng = 7;*

*struct Dx: Cơ sở {*

*int cuối cùng;*

*int f () cuối cùng*

*{*

*trả về cuối cùng + :: cuối cùng;*

*}*

*};*

Lý do duy nhất mà cuối cùng là một vấn đề về từ khóa văn bản, chứ không phải là một từ khóa thông thường, có tồn tại một lượng mã đáng kể đã được sử dụng cuối cùng như một số nhận dạng thông thường trong nhiều thập kỷ. Từ khóa theo ngữ cảnh khác được ghi đè V.3.5 Sử dụng Thành viên cơ sở Các hàm không quá tải trên các phạm vi. Ví dụ:

*cơ sở struct {*

*void f (int);*

*};*

*struct Derived: Base {*

*void f (kép);*

*};*

*vô hiệu sử dụng (Bắt nguồn từ d)*

*{*

*d.f (1); // gọi Derived :: f (double)*

*Cơ sở & br = d*

*br.f (1); // gọi Base :: f (int)*

*}*

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

*struct D2: Cơ sở {*

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

*void f (kép);*

*};*

*void use2 (D2 d)*

*{*

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

*Cơ sở & br = d*

*br.f (1); // gọi Base :: f (int)*

*}*

Đây là một hệ quả đơn giản của một lớp cũng được coi là một không gian tên. Một số khai báo sử dụng có thể mang tên từ nhiều lớp cơ sở.

Ví dụ:

*struct B1 {*

*void f (int);*

*};*

*struct B2 {*

*void f (kép);*

*};*

*struct D: B1, B2 {*

*sử dụng B1 :: f;*

*sử dụng B2 :: f;*

*void f (char);*

*};*

*vô hiệu sử dụng (D d)*

*{*

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

*d.f ('a'); // gọi D :: f (char)*

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

*}*

##### 20.3.5.1 Trình tạo kế thừa (Inheriting Constructors)

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 <lớp T>*

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

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

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

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

*};*

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 một cơ sở hoặc yêu cầu một quy tắc chặt chẽ hơnlớp bất biến, sẽ là một thảm họa nếu kế thừa các hàm tạo. Tuy nhiên, Vector đã không làm gì cả như thế.

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

*template <lớp T>*

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

*sử dụng vector <T> :: vector; // kế thừa các uctors xây dựng*

*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) ném Bad\_index (i); }*

*};*

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

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

*struct B1 {*

*B1 (int) {}*

*};*

*struct D1: B1 {*

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

*dây; // chuỗi có một hàm tạo mặc định*

*int x; // chúng tôi '' quên '' cung cấp cho việc khởi tạo x*

*};*

*kiểm tra void ()*

*{*

*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 và D1 :: x không phải là hàm tạo kế thừa tương đương với một phương thức 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) {}*

*dây; // chuỗi có một hàm tạo mặc định*

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

*};*

*struct D1: B1 {*

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

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

*};*

*kiểm tra void ()*

*{*

*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 cho sim- trường hợp không có thành viên dữ liệu được thêm vào.

#### 20.3.6 Thư giãn loại trả lại (Return Type Relaxation)

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 là hàm ghi đè có thể là D ∗, với điều kiện B là cơ sở công khai của D. Tương tự, kiểu trả về của B & có thể được thoải mái để D &. Điều này đôi khi được gọi là quy tắc trả về hiệp phương sai.

Sự thư giãn này chỉ áp dụng cho các loại trả về là con trỏrs hoặc tham chiếu, và không phải ‘‘ thông minh con trỏ '’chẳng hạn như unique\_ptr .Đặc biệt, không có sự nới lỏng tương tự của các quy tắc đối với loại đối số vì điều đó sẽ dẫn đến vi phạm loại.

Hãy xem xét một hệ thống phân cấp lớp đại diện cho các loại biểu thức khác nhau. Ngoài opera- tions để thao tác các biểu thức, lớp cơ sở Expr sẽ cung cấp các phương tiện để tạo mới các đối tượng biểu thức của các loại 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 () tạo ra một bản sao của đối tượng. Cả hai sẽ trả về một đối tượng của một số lớp cụ thể có nguồn gốc từ Expr. Họ có thể không bao giờ chỉ trả về một '' Expr đơn giản '' 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 của ‘‘ giống nhau gõ. ’’

Ví dụ:

*void user (Expr ∗ p)*

*{*

*Expr ∗ p2 = p−> new\_expr ();*

*// ...*

*}*

Điều kiện để được nhân bản mà không mất thông tin loại. Tương tự, một phép cộng lớp dẫn xuất sẽ có một bản sao () trả về một Bổ sung ∗. Ví dụ:

*void user2 (Cond ∗ pc, Addition ∗ pa)*

*{*

*Cond ∗ p1 = pc−> clone ();*

*Phép cộng ∗ 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 () return 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 sự vật.

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 có thể được phá vỡ bằng cách xác định một hàm gọi một phương thức khởi tạo và trả về một đối tượng đã xây dựng. Điều này thật may mắn vì tạo ra một đối tượng mới mà không biết loại chính xác của nó thường hữu ích. Ival\_box\_maker (§21.2.4) là một ví dụ về một lớp được thiết kế cụ thể để làm điều đó.

### 20.4 Các lớp trừu tượng (Abstract Classes)

Nhiều lớp giống với lớp Employee ở chỗ chúng hữu ích như chính chúng, làm giao diện cho các lớp dẫn xuất và như một phần của việc triển khai các lớp dẫn xuất. Đối với các lớp học như vậy, công nghệ- 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 học, chẳng hạn như một lớp Shape, đại diện cho các khái niệm trừu tượng mà đối tượng không thể tồn tại. Một hình dạng làm cho có nghĩa là cơ sở của một số lớp có nguồn gốc từ nó. Điều này có thể được thấy từ thực tế là nó không có thể cung cấp các định nghĩa hợp lý cho các chức năng ảo của nó:

*Hình dạng lớp {*

*public:*

*ảo void xoay (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:

Hình dạng; // 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:

*class Shape {// lớp trừu tượng*

*public:*

*void quay ảo (int) = 0; // hàm ảo thuần túy*

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

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

*// ...*

*ảo ̃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 lớp có thể được tạo:

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

*điểm lớp {/ \* ... \* /};*

*class Circle: public Shape {*

*public:*

*ghi đè void xoay (int) {}*

*void draw () const override;*

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

*Circle (Điểm p, int r);*

*private:*

*Tâm điểm;*

*bán kính int;*

*} ;*

Một hàm ảo thuần túy không được định nghĩa trong lớp dẫn xuất vẫn là một hàm ảo thuần túy, vì vậy lớp dẫn xuất cũng là một lớp trừu tượng. Điều này cho phép chúng tôi xây dựng 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 ...*

*};*

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

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

*class Irregular\_polygon: public Polygon {*

*danh sách <Điểm> lp;*

*public:*

*Irregular\_polygon (khởi tạo er\_list <Point>);*

*void draw () const override;*

*ghi đè void quay (int);*

*// ...*

*};*

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

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

*class Character\_device {*

*public:*

*int ảo open (int opt) = 0;*

*int ảo close (int opt) = 0;*

*int ảo read (char ∗ p, int n) = 0;*

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

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

*virtual ̃Character\_device () {} // vir tual hủy uctor*

*};*

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

### 20.5 Kiểm soát truy cập (Access Control)

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

• Nếu nó ở chế độ riêng tư, tên của nó chỉ có thể được sử dụng bởi các chức năng thành viên và bạn bè của lớp trong mà nó được khai báo.

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

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

Điều này phản ánh quan điểm rằng có ba loại hàm truy cập vào một lớp: các hàm thực hiện- nhập vào lớp (bạn bè và thành viên của nó), các hàm triển khai một lớp dẫn xuất (lớp dẫn xuất của bạn bè và thành viên), và các chức năng khác. Điều này có thể được trình bày bằng đồ thị:

*public:*

*protected:*

*private:*

Ví dụ, một lớp danh sách không ủy quyền hiệu quả thường yêu cầu cấu trúc dữ liệu để theo dõi các phần tử. Một danh sách là không có tác dụng nếu nó không yêu cầu sửa đổi các phần tử của nó (ví dụ: bằng cách yêu cầu các loại phần tử phải có trường liên kết). Cấu trúc thông tin và dữ liệu được sử dụng để tổ chức danh sách có thể được giữ kín:

*template <lớp T>*

*danh sách lớp học {*

*public:*

*void insert (T);*

*T get ();*

*// ...*

*paivate:*

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

*struct Chunk {*

*enum {chunk\_siz e = 15};*

*Liên kết v [chunk\_siz e];*

*Chunk ∗ tiếp theo;*

*};*

*Chunk ∗ được phân bổ;*

*Liên kết ∗ miễn phí;*

*Liên kết ∗ get\_free ();*

*Liên kết ∗ đầu;*

*};*

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

*template <lớp T>*

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

*{*

*Liên kết ∗ lnk = get\_free ();*

*lnk−> val = val;*

*lnk−> tiếp theo = đầu;*

*đầu = lnk;*

*}*

*template <lớp T>*

*Danh sách T <T> :: g et ()*

*{*

*if (head == 0)*

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

*Liên kết ∗ p = đầu;*

*đầu = p−> tiếp theo;*

*p−> next = miễn phí;*

*miễn phí = p;*

*return p−> val;*

*}*

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

*Template <lớp T>*

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

*{*

*if (miễn phí == 0) {*

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

*}*

*Liên kết ∗ p = miễn phí;*

*miễn phí = free−> tiếp theo;*

*trả lại p;*

*}*

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

*template <lớp T>*

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

*{*

*// ...*

*}*

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

*template <typename T>*

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

*{*

*Danh sách <T> :: Liên kết ∗ q = 0; // error: Danh sách <T> :: Liên kết là riêng tư*

*// ...*

*q = p−> tự do; // error: List <T> :: free là private*

*// ...*

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

*// ...*

*}*

*}*

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 cách thay thế rõ ràng cho việc sử dụng kiểu thành viên là đặt kiểu trong tên xung quanh- không gian.

Ví dụ:

*tempalte <lớp T>*

*struct Link2 {*

*T val;*

*Link2 ∗ tiếp theo;*

*};*

*mẫu <lớp T>*

*danh sách lớp học {*

*private:*

*Link2 <T> ∗ miễn phí;*

*// ...*

*};*

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 ver-sion có thể được ưu tiên hơn; xem §23.4.6.3.

Nếu lớp lồng nhau nói chung không hữu ích cho chính nó và lớp bao quanh cần quyền truy cập vào đại diện, tuyên bố lớp thành viên là bạn (§19.4.2) có thể là một ý tưởng hay:

*template <class T> Danh sách lớp;*

*template <lớp T>*

*liên kết lớp3 {*

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

*T val;*

*Liên kết3 ∗ tiếp theo;*

*};*

*tempalte <lớp T>*

*danh sách lớp học {*

*private:*

*Link3 <T> ∗ miễn phí;*

*// ...*

*};*

Một trình biên dịch có thể sắp xếp lại thứ tự các phần của một lớp với các chỉ định truy cập riêng biệt .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. Sắp xếp lại như vậy có thể gây ngạc nhiên cho lập trình viên và phụ thuộc vào việc triển khai, vì vậy đừng sử dụng nhiều truy cập các chỉ định cho các thành viên dữ liệu mà không có lý do chính đáng.

#### 20.5.1 Thành viên được bảo vệ (Protected Members)

Khi thiết kế hệ thống phân cấp lớp, đôi khi chúng tôi cung cấp các hàm được thiết kế để sử dụng bởi imple- trình cố vấn của các lớp dẫn xuất nhưng không phải bởi người dùng chung. Ví dụ: chúng tôi có thể cung cấp một (hiệu quả) chức năng truy cập bỏ chọn cho người triển khai lớp dẫn xuất và quyền truy cập đã kiểm tra (an toàn) cho những người khác.

Khai báo phiên bản được bảo vệ không được kiểm tra đạt được điều đó. Ví dụ:

*bộ đệm lớp {*

*public:*

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

*// ...*

*protected:*

*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*

*}*

##### 20.5.1.1 Sử dụng các Thành viên được bảo vệ (Use of protected Members)

Mô hình ẩn dữ liệu riêng tư / công khai đơn giản phục vụ tốt khái niệm về các kiểu cụ thể.

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. Ngang bằng- ticular, 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, dữ liệu được bảo vệ, như dữ liệu công khai, không thể dễ dàng được cấu trúc lại vì không có cách tốt để tìm ing mọi 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 sự lựa chọn tốt hơn. Theo kinh nghiệm của tôi, luôn có những lựa chọn thay thế để đặt lượng thông tin trong một cơ sở chung class cho các lớp dẫn xuất để sử dụng trực tiếp.

Tuy nhiên, không có phản đối nào trong số này có ý nghĩa 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 là một kỳ thi-lời cầu xin của điều này. Nếu lớp triển khai là riêng tư trong ví dụ này, thì việc dẫn xuất thêm sẽ đã không khả thi. Mặt khác, công khai các cơ sở cung cấp chi tiết triển khai mời những sai lầm và sử dụng sai.

#### 20.5.2 Quyền truy cập vào các lớp cơ sở (Access to Base Classes)

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:

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 tư cho một lớp và một cơ sở công khai cho một cấu trúc.

Ví dụ:

*class XX: B {/ \* ... \* /}; // B là một cơ sở riêng*

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

##### 20.5.2.1 Kiểm soát nhiều quyền thừa kế và truy cập (Multiple Inheritance and Access Control)

Nếu tên của một lớp cơ sở có thể được tiếp cận thông qua nhiều đường dẫn trong mạng đa kế thừa), nó có thể truy cập được nếu có thể truy cập thông qua bất kỳ con đường 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 làchỉ 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 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:*

*sử dụng B :: a; // lỗi: B :: a là riêng tư*

*sử dụng 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 với một số, nhưng không phải tất cả, của 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:*

*sử dụng B :: b;*

*sử dụng B :: c;*

*};*

### 20.6 Con trỏ tới Membia (Pointers to Members)

Con trỏ tới thành viên là một cấu trúc giống như bù đắp cho phép lập trình viên tham chiếu gián tiếp đến một thành viên của một lớp học. 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 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 (Pointers to Function Members)

Nhiều lớp cung cấp các giao diện đơn giản, rất chung chung nhằm mục đích được gọi trong các giao diện khác nhau các cách. 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 như vậy có thể được 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 tạm ngưng () = 0;*

*ảo void resume () = 0;*

*void ảo bỏ () = 0;*

*void ảo full\_size () = 0;*

*ảo void small () = 0;*

*ảo ̃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 giữa người hoặc chương trình đưa ra yêu cầu và đối tượng nhận nó.

Chúng tôi vẫn phải viết mã để xác định rằng 2 có nghĩa là đình chỉ () và để gọi đình chỉ ().

Tuy nhiên, chúng ta có thể sử dụng một con trỏ tới thành viên để gián tiếp tham chiếu đến thành viên của một lớp. Xem xét Std\_interface. Nếu tôi muốn gọi hàm tạm dừng () cho một số đối tượng mà không đề cập trực tiếp đến lệnh đình chỉ () Tôi cần một con trỏ đến thành viên tham chiếu đến Std\_interface :: Susan (). Tôi cũng cần một con trỏ hoặc tài liệu tham khảo đối tượng mà tôi muốn tạm dừng. 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 địa chỉ-của nhà điều hành, &, cho một tên thành viên lớp chẳng hạn, & Std\_interface :: tạm ngưng. 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 việc thiếu tính dễ đọc của cú pháp bộ khai báo C là điển hình-cal. Tuy nhiên, hãy lưu ý cách bộ khai báo X :: ∗ khớp chính xác với bộ khai báo ∗ truyền thống.

Một 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 kiểu. Không thể lưu trữ kết quả của phép toán a -> ∗ hoặc a. ∗ để sử dụng sau này.

Đương nhiên, nếu chúng tôi biết mình muốn gọi thành viên nào, chúng tôi sẽ gọi nó trực tiếp thay vì lộn xộn với các con trỏ đến các thành viên. Cũng giống như con trỏ thông thường đến các hàm, con trỏ đến hàm thành viên tions 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 cho một chức năng là. Nó giống như một sự bù đắp vào một cấu trúc hoặc một chỉ mục vào một mảng, nhưng tất nhiên là một việc triển khai có tính đến sự khác biệt giữa các thành viên dữ liệu, các chức năng ảo, không các hàm ảo, v.v. Khi một con trỏ tới thành viên được kết hợp với một con trỏ tới một đối tượng của đúng loại, nó mang lại một cái gì đó xác định một thành viên cụ thể của một đối tượng cụ thể.

Lời gọi p -> ∗ s () có thể được biểu diễn bằng đồ thị như sau:

*X :: bắt đầu*

*X :: tạm ngưng*

*vtbl:*

*NS:*

*P*

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 để hoạt động, chúng ta sẽ nhận được hàm Susan () phù hợp để 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 một trình thông dịch, chúng tôi có thể sử dụng con trỏ tới các thành viên để gọi các hàm được trình bày dưới dạng dây:

*map <string, Std\_interface ∗> biến;*

*thao tác map <string, Pstd\_mem>;*

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

*{*

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

*}*

Thành viên tĩnh không được liên kết với một đối tượng cụ thể, do đó, một con trỏ đến thành viên tĩnh chỉ đơn giản là một con trỏ bình thường. Ví dụ:

*Nhiệm vụ lớp {*

*// ...*

*lịch void tĩnh ();*

*};*

*void (∗ p) () = & Task :: Schedule; // VÂNG*

*void (Tác vụ :: ∗ chiều) () = & Tác vụ :: lịch trình; // error: con trỏ ordinar y được chỉ định*

*// tới con trỏ tới thành viên*

#### 20.6.2 Con trỏ đến các thành viên dữ liệu (Pointers to Data Members)

Đương nhiên, khái niệm con trỏ đến thành viên áp dụng cho các thành viên dữ liệu và cho các chức năng thành viên với 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 đến 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;*

*Chiều pm = & C :: val;*

*z1.print (1);*

*(z1. ∗ pf) (2);*

*z1. ∗ pm = "nv1";*

*p -> ∗ pm = "nv2";*

*z2.print (3);*

*(p -> ∗ pf) (4);*

*pf = & C :: f1; // 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 (Base and Derived Members)

Một lớp dẫn xuất có ít nhất các thành viên mà nó kế thừa từ các lớp cơ sở của nó. Thường thì nó có nhiều hơn.

Điều này ngụ ý rằng chúng ta có thể chỉ định một cách an toàn một con trỏ cho một thành viên của lớp cơ sở cho một con trỏ tới một 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 đình chỉ ();*

*// ...*

*void in ảo ();*

*private:*

*vectơ s;*

*};*

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

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

Quy tắc tương phản trái ngược với quy tắc nói rằng chúng ta có thể gán một con trỏ cho một lớp dẫn xuất đến 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 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, họ 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 tiết kiệm khỏi lỗi thời gian chạy.

### 20.7 Lời khuyên (Advice)

[1] Tránh các trường loại.

[2] Truy cập các đối tượng đa hình thông qua con trỏ và tham chiếu.

[3] Sử dụng các lớp trừu tượng để tập trung thiết kế vào việc cung cấp các giao diện sạch;

[4] Sử dụng tính năng ghi đè để làm cho việc ghi đè trở nên rõ ràng trong các cấu trúc phân cấp lớp lớn.

[5] Chỉ sử dụng cuối cùng một cách tiết kiệm.

[6] Sử dụng các lớp trừu tượng để chỉ định giao diện.

[7] Sử dụng các lớp trừu tượng để giữ các chi tiết triển khai nằm ngoài giao diện .

[8] Một lớp có hàm ảo nên có hàm hủy ảo.

[9] Một lớp trừu tượng thường không cần hàm tạo.

[10] Ưu tiên các thành viên tư nhân để biết chi tiết thực hiện.

[11] Ưu tiên các thành viên công khai cho các giao diện.

[12] Chỉ sử dụng các thành viên được bảo vệ cẩn thận khi thực sự cần thiết.

[13] Không tuyên bố dữ liệu thành viên được bảo vệ.

## Chap 21: Lớp Phân Cấp

### 21.1 Giới thiệu (Introduction)

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

### 21.2 Cấu Trúc Phân Lớp (Design of Class Hierarchies)

Ý tưởng là có một lớp Ival\_box (‘‘ hộp nhập giá trị số nguyên ’’) biết phạm vi đầu vào giá trị nó sẽ chấp nhận. Ngoài ra, một chương trình có thể hỏi Ival\_box nếu người dùng đã thay đổi giá trị kể từ khi chương trình lần cuối nhìn vào nó:

*giá trị người sử dụng*

*(thông qua ứng dụng ‘‘ system ’’)*

*Ival\_box:*

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

Bởi vì có nhiều cách để thực hiện ý tưởng cơ bản này, chúng ta phải giả định rằng sẽ có nhiều loại Ival\_box khác nhau, chẳng hạn như thanh trượt, hộp trơn trong đó người dùng có thể nhập một 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. Tôi đã chọn cái này phương pháp tiếp cận vì nó chung chung, vì nó cho phép tôi chứng minh nhiều kỹ thuật khác nhau và thiết kế cân bằng, 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 - vì những kỹ thuật này có thể áp dụng cho các vấn đề vượt xa miền hẹp của các hệ thống giao diện.

#### 21.2.1 Kế thừa triển khai (Implementation Inheritance)

Giải pháp đầu tiên của chúng tôi là phân cấp lớp sử dụng kế thừa triển khai (như thường thấy trong 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 tuyên bố dữ liệu cần thiết để triển khai khái niệm cơ bản:

*class Ival\_box {*

*protected:*

*int val;*

*int thấp, cao;*

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

*public:*

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

*virtual int get\_value () {chang ed = false; trả lại 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*

*dấu nhắc vô hiệu ảo () {}*

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

*ảo ̃Ival\_box () {};*

*};*

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

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

*void tương tác (Ival\_box ∗ pb)*

*{*

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

*// ...*

*int i = pb−> get\_value ();*

*if (pb−> was\_chang ed ()) {*

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

*}*

*khác {*

*// ... làm việc gì khác ...*

*}*

*}*

*void some\_fct ()*

*{*

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

*tương tác (p1.get ());*

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

*tương tác (p2.get ());*

*}*

Để đơn giản hóa cuộc thảo luận, tôi không đề cập đến các vấn đề về cách một chương trình chờ đợi đầu vào. Có thể là chương trình thực sự đợi người dùng trong get\_value () (ví dụ: sử dụng get () trong tương lai), có thể chương trình liên kết Ival\_box với một sự kiện và chuẩn bị trả lời một cuộc 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. Như là quyết định là 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 sẽ chỉ đơn giản là phân tâm khỏi việc trình bày các kỹ thuật lập trình và lan- cơ sở guage. 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 xác định thanh trượt trông như thế nào, v.v. ...*

*public:*

*Ival\_slider (int, int);*

*int get\_value () ghi đè;*

*// lấy giá trị từ người dùng và gửi nó vào val*

*ghi đè void prompt ();*

*};*

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. Vì vậy, Ival\_slider :: get\_value () có thể gửi một giá trị vào Ival\_box :: val. Thành viên được bảo vệ có thể truy cập từ các thành viên riêng của lớp và từ các thành viên của các lớp dẫn xuất, nhưng không phải cho người dùng chung

##### 21.2.1.1 Phê bình (Critique)

Thiết kế này hoạt động tuy tốt nhưng vấn có vấn đề khó giải quyết. 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.

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

Bắt nguồn từ BBwidget mang lại lợi ích của việc làm cho các cơ sở được cung cấp bởi BBwidget avail-có thể cho người dùng Ival\_box. Thật không may, điều đó cũng có nghĩa là các thay đổi đối với lớp BBwidget có thể buộc người dùng biên dịch lại hoặc thậm chí viết lại mã của họ để khôi phục từ những thay đổi như vậy.

#### 21.2.2 Kế thừa giao diện (Interface Inheritance)

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 cri-tique của hệ thống phân cấp truyền thống:

[1] Hệ thống giao diện người dùng phải là chi tiết triển khai được ẩn khỏi người dùng ai không muốn biết về nó.

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

[3] Không cần biên dịch lại mã sử dụng nhóm lớp Ival\_box sau.

thay đổi của 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.

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

*class Ival\_box {*

*public:*

*ảo int get\_value () = 0;*

*void set\_value ảo (int i) = 0;*

*void reset\_value ảo (int i) = 0;*

*dấu nhắc void ảo () = 0;*

*bool ảo was\_chang ed () const = 0;*

*ảo ̃Ival\_box () {}*

*};*

Điều này rõ ràng hơn nhiều so với khai báo ban đầu của Ival\_box. Dữ liệu đã biến mất và sim- triển khai toàn diện của các chức năng thành viên. Gone cũng là hàm tạo, vì không có dữ liệu để nó khởi tạo. Thay vào đó, tôi đã thêm một trình hủy ảo để đảm bảo xóa sạch dữ liệu phù hợp sẽ được định nghĩa 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);*

*Ghi đè ̃Ival\_slider ();*

*int get\_value () ghi đè;*

*ghi đè void set\_value (int i);*

*// ...*

*protected:*

*// ... các hàm ghi đè các hàm ảo BBwidget*

*// ví dụ: 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 phương tiện làm như vậy. Vì Ival\_box cung cấp giao diện cho lớp dẫn xuất, nó được dẫn xuất bằng cách sử dụng công cộng. Vì BBwidget chỉ là một công cụ hỗ trợ triển khai, nó được dẫn xuất bằng cách sử dụng bảo vệ.

Chúng tôi đảm bảo dọn dẹp đúng cách 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.

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 BBwidg et*

*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 vào các căn cứ được bảo vệ bởi vì chúng (một cách chính xác) được coi là một phần của việc thực hiện.

#### 21.2.3 Triển khai Thay thế (Alternative Implementations)

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

*class Ival\_slider*

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

*class Ival\_slider*

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

*// ...*

Không có cách nào để Ival\_slider cho BBwidgets cùng tồn tại với Ival\_slider cho CWwidgets, vi nếu bản thân 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 CWwidg Ival\_box et*

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

Để cách ly thêm 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 Ival\_slider 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 CWwidg Ival\_slider et*

*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 tôi có thể lấy trực tiếp Ival\_slider của mình từ BBslider:

*lớp BB\_ival\_slider*

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

*class CW\_ival\_slider*

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

hoặc bằng đồ thị:

*Ival\_box*

*BBslider CWslider Ival\_slider*

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

*BBwidget CWwidg et*

Sự cải tiến này trở nên đáng kể khi - không có gì lạ - những điều trừu tượng của chúng tôi không quá khác với những cái được cung cấp bởi hệ thống được sử dụng để thực hiện. Trong trường hợp đó, chương trình- ming được giảm thành ánh xạ giữa các khái niệm tương tự. Bắt nguồn từ các lớp cơ sở chung, chẳng hạn như như BBwidget, hiếm khi được thực hiện.

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

*clas 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 thể hiện 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 chữ viết tắt rõ ràng.

Xem thêm về văn bản nguồn nàyNhập văn bản nguồn để có thông tin dịch thuật bổ sung ns, hệ thống phân cấp này có thể được biểu diễn bằng đồ thị như sau:

*Ival\_box*

*Ival\_slider Ival\_dial*

*ipopup iflash*

*CWsl CWsl CWsl*

*BBislider CWipop CWi BBipop BBi fl fl CWislider*

*BBslider BBslider BBb & w*

##### 21.2.3.1 Phê bình (Critique)

Thiết kế lớp trừu tượng linh hoạt và gần như đơn giản để xử lý như thiết kế tương đương dựa trên một cơ sở chung xác định hệ thống giao diện người dùng. Trong thiết kế thứ hai, lớp cửa sổ là rễ của cây. Trước đây, cấu trúc phân cấp lớp ứng dụng ban đầu dường như không thay đổi vì gốc của các lớp cung cấp các triển khai của nó. Theo quan điểm của ứng dụng, những thiết kế này tương đương với nghĩa mạnh rằng hầu như tất cả các mã 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 tôi sẽ không cần phải viết tương tác () từ, nếu chúng ta chuyển từ hệ thống phân cấp lớp này sang hệ thống phân cấp lớp khác.

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

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

#### 21.2.4 Bản địa hóa Tạo đối tượng (Localizing Object Creation)

Hầu hết một ứng dụng có thể được viết bằng giao diện Ival\_box. Hơn nữa, nếu liên kết dẫn xuất các khuôn mặt phát triển để cung cấp nhiều tiện ích hơn so với Ival\_box đơn thuần, khi đó hầu hết các ứng dụng đều có thể được ghi- mười bằng cách sử dụng giao diện Ival\_box, Ival\_slider, v.v.,. Tuy nhiên, việc tạo ra các đối tượng phải được thực hiện sử dụng các tên cụ thể cho việc triển khai như CW\_ival\_dial và BB\_flashing\_ival\_slider. Chúng tôi sẽ muốn giảm thiểu số lượng địa điểm xuất hiện những cái tên cụ thể như vậy và việc tạo đối tượng rất khó để 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 đơn giản một là giới thiệu một lớp trừu tượng để đại diện cho tập hợp các hoạt động tạo:

*class Ival\_maker {*

*public:*

*ảo 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 đôi khi được gọi là các hàm tạo ảo.

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 {// tạo các phiên bản BB*

*public:*

*Ghi đè Ival\_dial ∗ dial (int, int);*

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

*// ...*

*};*

*class LS\_maker: public Ival\_maker {// tạo các phiên bản LS*

*public:*

*Ghi đè Ival\_dial ∗ dial (int, int);*

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

*// ...*

*};*

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

Ví dụ:

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

*{*

*trả về BB\_ival\_dial mới (a, b);*

*}*

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

*{*

*trả về LS\_ival\_dial mới (a, b);*

*}*

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

Ví dụ:

*người dùng vô hiệu (Ival\_maker & im)*

*{*

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

*// ...*

*}*

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

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

*void driver ()*

*{*

*người dùng (BB\_impl); // sử dụng BB*

*người dùng (LS\_impl); // sử dụng LS*

*}*

### 21.3 Thừa kế nhiều (Multiple Inheritance)

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 tạo mã như vậy

đồng đều 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. Cái này thường được gọi là kế thừa thực thi.

#### 21.3.1 Nhiều giao diện (Multiple Interfaces)

Một lớp trừu tượng là cách rõ ràng để biểu diễn một giao diện. Cho một lớp trừu tượng không có trạng thái có thể thay đổi, thực sự có rất ít sự khác biệt giữa sử dụng đơn lẻ và sử dụng nhiều lần của một lớp cơ sở trong một hệ thống phân cấp lớp.

Việc sử dụng nhiều lớp trừu tượng làm giao diện gần như phổ biến trong các thiết kế hướng đối tượng.

#### 21.3.2 Nhiều lớp triển khai (Multiple Implementation Classes)

Ví dụ về vệ tinh sẽ là đá, mảnh vỡ từ các phương tiện vũ trụ cũ, vệ tinh liên lạc, và Interna- tional trạm vũ trụ. Các loại vệ tinh này sẽ là đối tượng của các lớp có nguồn gốc 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 các chức năng ảo để điều chỉnh ý nghĩa của chúng cho phù hợp.

Bây giờ chúng ta có thể định nghĩa một lớp vệ tinh truyền thông mô phỏng, class Comm\_sat:

*class Comm\_sat: vệ tinh công cộng, hiển thị công khai {*

*public:*

*// ...*

*};*

hoặc bằng đồ thị:

*Hiển thị qua vệ tinh*

*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 (); // Hiển thị :: draw ()*

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

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

*}*

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

*void highlight (Hiển thị ∗);*

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

*void g (Comm\_sat ∗ p)*

*{*

*tô sáng (p); // chuyển một con trỏ đến par t được hiển thị của Comm\_sat*

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

*}*

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

*vệ tinh lớp {*

*public:*

*tâm Pos ảo () const = 0; // Trung tâm của lực hấp dẫn*

*// ...*

*};*

*lớp được hiển thị {*

*public:*

*ảo void draw () = 0;*

*// ...*

*};*

*class Comm\_sat: vệ tinh công cộng, hiển thị công khai {*

*public:*

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

*ghi đè void draw (); // 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 một Comm\_sat và một Hiển thị, tương ứng.

Tôi có thể đã định nghĩa 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 Vệ tinh và một thành viên Hiển thị ∗ và để cho nó structor thiết lập các kết nối thích hợp. Đối với nhiều vấn đề thiết kế, tôi chỉ làm như vậy. Tuy vậy, 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 a (riêng designed) Lớp được hiển thị với các chức năng ảo. Bạn đã cung cấp vệ tinh của riêng mình và các đối tượng được hiển thị của riêng bạn thông qua dẫn xuất. Đặc biệt, bạn đã phải ghi đè lên Satellite vir- chức năng thành viên tual và chức năng thành viên ảo được hiển thị để chỉ định hành vi của riêng bạn các đối tượng. Đó là tình huống trong đó đa kế thừa của các lớp cơ sở với trạng thái và thiết lập khó tránh khỏi sự cố vấn. Các giải pháp thay thế có thể gây đau đớn và khó duy trì.

Việc sử dụng đa kế thừa để '' kết dính '' hai lớp không liên quan với nhau như một phần của việc triển khai lớp thứ ba là thô sơ, hiệu quả và tương đối quan trọng, nhưng không liên quan làm tổ. Về cơ bản, nó giúp lập trình viên không phải viết rất nhiều hàm chuyển tiếp (để so sate vì thực tế là chúng ta chỉ có thể ghi đè các hàm được xác định trong base). Kỹ thuật này không ảnh hưởng đáng kể đến thiết kế tổng thể của một chương trình và đôi khi có thể xung đột với mong muốn giữ chi tiết triển khai ẩn. Tuy nhiên, một kỹ thuật không cần phải khéo léo để trở nên hữu ích.

Hiển thị, Vệ tinh và Comm\_sat sẽ bị giới hạn. Một Comm\_sat có thể là một Vệ tinh hoặc một Dis- đã phát, nhưng không phát cả hai (trừ khi Vệ tinh bắt nguồn từ Hiển thị hoặc ngược lại). Hoặc thay thế liên quan đến việc mất tính linh hoạt.

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

#### 21.3.3 Độ phân giải mơ hồ (Ambiguity Resolution)

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

*vệ tinh lớp {*

*class:*

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

*// ...*

*};*

*lớp được hiển thị {*

*class:*

*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 đủ điều kiện cho một tên thành viên 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 (); // VÂNG*

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

*}*

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

*class Comm\_sat: vệ tinh công cộng, hiển thị công khai {*

*public:*

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

*{*

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

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

*trả về merge\_info (di1, di2);*

*}*

*// ...*

*};*

Một hàm được khai báo trong lớp dẫn xuất sẽ ghi đè tất cả các hàm có cùng tên và nhập vào cơ sở của nó các lớp học.

Ví dụ:

*class Telstar: public Comm\_sat {*

*public:*

*void draw ()*

*{*

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

*// ... nội dung riêng ...*

*}*

*// ...*

*};*

hoặc bằng đồ thị:

*Hiển thị qua vệ tinh*

*Comm\_sat*

*Telstar*

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

*Nếu, trong Telstar :: draw (), tôi đã nói đơn giản là draw (), kết quả sẽ là một đệ quy '' vô hạn '' cuộc gọi 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 a Comm\_sat :: draw ()*

*Xem thêm về văn bản nguồn nàyNhập văn bản nguồn để có thông tin dịch thuật bổ sung*

*Gửi phản hồi*

*Bảng điều khiển bên*

*Lịch sử*

*Đã lưu*

Đóng góp; 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. Tôi đã nói

Satellite :: draw (), kết quả sẽ là một lỗi vì quá trình rút thăm đã kết thúc trên Màn hình hiển thị

nhánh của hệ thống phân cấp lớp.

Ví dụ get\_debug () về cơ bản giả định rằng ít nhất một số phần của Vệ tinh và Hiển thị đã được thiết kế cùng nhau. Lấy một kết hợp chính xác về tên, loại trả về, loại đối số và ngữ nghĩa tình cờ là cực kỳ khó xảy ra. Nhiều khả năng chức năng tương tự lad 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 đã 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, 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 (); // gọi SimObj :: DBinf () và giải nén thông tin*

*// ...*

*};*

*class Hiển thị: Widget công cộng {*

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

*public:*

*virtual Debug\_info get\_debug (); // đọc dữ liệu Widget và soạn Debug\_info*

*// ...*

*};*

hoặc bằng đồ thị:

*SimObj Widg et*

*Hiển thị qua vệ tinh*

*Comm\_sat*

*Telstar*

Ví dụ về một lớp hàm thành viên draw () trong trò chơi điện tử liên quan đến cao bồi:

*Cửa sổ lớp học {*

*public:*

*void draw (); // hình ảnh hiển thị*

*// ...*

*};*

*cao bồi lớp {*

*public:*

*void draw (); // rút súng từ bao da*

*// ...*

*};*

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

*// ...*

*};*

Các lớp trung gian sẽ thực hiện:

*struct WWindow: Cửa sổ {*

*sử dụng Window :: Window; // kế thừa các uctors xây dựng*

*void win\_draw () = 0; // buộc lớp dẫn xuất thành overr Ide*

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

*};*

*struct CCowboy: Cao bồi {*

*sử dụng Cowboy :: Cowboy; // kế thừa các uctors xây dựng*

*void cow\_draw () = 0; // buộc lớp dẫn xuất thành overr Ide*

*void draw () ghi đè cuối cùng {cow\_draw (); } // rút súng từ bao da*

*};*

*class Cowboy\_window: CCowboy công cộng, WWindow công cộng {*

*public:*

*ghi đè void cow\_draw ();*

*void win\_draw () ghi đè;*

*// ...*

*};*

Hoặc bằng đồ thị:

*Cao bồi cửa sổ*

*WWindow CCowboy*

*Cowboy\_window*

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

#### 21.3.4 Sử dụng lặp lại một lớp cơ sở (Repeated Use of a Base Class)

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

*struct Stocking {// lưu trữ liên tục*

*chuỗi ảo get\_file () = 0;*

*ảo void read () = 0;*

*void ghi () = 0;*

*ảo ̃Storable () {}*

*};*

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

*Bộ truyền lớp: công khai lưu trữ {*

*public:*

*ghi đè void write ();*

*// ...*

*};*

*Class Receiver: public Stocking {*

*public:*

*ghi đè void write ();*

*// ...*

*};*

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

*public:*

*ghi đè chuỗi get\_file ();*

*void read () ghi đè;*

*ghi đè void write ();*

*// ...*

*};*

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

[1] Một đối tượng Radio có hai subobject của lớp Stocking (một cho Transmitter và một cho

Người nhận).

[2] Một đối tượng Radio có một đối tượng thuộc lớp Stocking (được chia sẻ bởi Máy phát và Máy thu).

Mặc định, được cung cấp cho ví dụ như đã viết, là hai subobject. Trừ khi bạn nói khác, bạn nhận được một bản sao cho mỗi lần bạn đề cập đến một lớp làm cơ sở. Về mặt đồ họa, chúng ta có thể thể hiện rằng như thế này:

*Đáng kinh ngạc*

*Máy phát Máy thu*

*Đài*

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

*void Radio :: write ()*

*{*

*Bộ truyền :: write ();*

*Bộ nhận :: write ();*

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

*}*

#### 21.3.5 Lớp cơ sở ảo (Virtual Base Classes)

Ví dụ Radio trong tiểu mục trước hoạt động bởi vì lớp lưu trữ có thể được an toàn, con- nhân rộng một cách nhẹ nhàng và hiệu quả. Lý do cho điều đó đơn giản là St Lovely là một lớp trừu tượng cung cấp một giao diện thuần túy. Một đối tượng lưu trữ không chứa dữ liệu của riêng nó. Đây là trường hợp đơn giản nhất và giải pháp cung cấp sự tách biệt tốt nhất giữa các mối quan tâm về giao diện và triển khai. Trên thực tế, một lớp không thể không gặp một số khó khăn khi xác định rằng có hai vật thể đáng yêu trên một Đài phát thanh.

Chúng tôi có thể xác định St Lovely để giữ tên của tệp được sử dụng để lưu trữ đối tượng:

*lớp học ổn định {*

*public:*

*Lưu trữ (const string & s); // lưu trữ trong tệp có tên s*

*ảo void read () = 0;*

*void ghi () = 0;*

*ảo ̃Storable ();*

*protected:*

*chuỗi tên\_tệp;*

*Stocking (const Stocking &) = xóa;*

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

*};*

Với sự thay đổi rõ ràng là nhỏ này đối với St Lovely, chúng ta phải thay đổi thiết kế của Radio. Tất cả các phần của một đối tượng phải chia sẻ một bản sao duy nhất của Stocking. Nếu không, chúng tôi có thể nhận được hai phần của một cái gì đó có nguồn gốc từ Lưu trữ nhiều lần bằng cách sử dụng các tệp khác nhau. Chúng tôi tránh sao chép bằng cách khai báo một cơ sở virtual: mọi cơ sở ảo của một lớp dẫn xuất được đại diện bởi cùng một đối tượng (dùng chung). Cho ví dụ- nài nỉ:

*Bộ truyền lớp: công khai ảo lưu trữ {*

*public:*

*ghi đè void write ();*

*// ...*

*};*

*Class Receiver: public virtual Stocking {*

*public:*

*ghi đè void write ();*

*// ...*

*};*

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

*public:*

*ghi đè void write ();*

*// ...*

*};*

*Hoặc bằng đồ thị:*

*Đáng yêu*

*Máy phát máy thu*

*Đài*

Trên mặt khác, mỗi lớp cơ sở không được chỉ định virtual sẽ có subobject riêng đại diện cho nó.

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

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

[2] Đặt dữ liệu trong một lớp cơ sở.

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

Tùy chọn [1], dữ liệu phi địa phương, thường là một lựa chọn kém vì chúng tôi không thể kiểm soát những gì mã truy cậpdữ 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 lẻ giải pháp đó làm cho dữ liệu hữu ích (và các chức năng) '' nổi lên '' thành một lớp cơ sở chung; thường nó ‘‘ bong bóng- bles ’’ đến tận gốc của cây kế thừa. Điều này có nghĩa là mọi thành viên của lớp đều-archy được quyền truy cập. Điều đó về mặt logic rất giống với việc sử dụng dữ liệu phi địa phương và cũng bị như vậy các vấn đề. 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 xây dựng-(các) tor cần dành bộ nhớ cho đối tượng được chia sẻ đó, khởi tạo nó và cung cấp các con trỏ tới đối tượng chia sẻ cho các đối tượng cần truy cập. Đó đại khái là những gì mà các nhà xây dựng làm để triển khai một vir- đế tual.

Ý tưởng.

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

*Người nhận*

*Hệ thống điều khiển*

*Đài*

*Đáng yêu*

Các '' con trỏ '' đến đối tượng được chia sẻ đại diện cho cơ sở ảo, St Lovely, sẽ là phần bù và thường một trong những điều đó có thể được tối ưu hóa bằng cách đặt Kho lưu trữ ở một vị trí cố định liên quan đến Bộ thu hoặc đối tượng bộ 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 (Constructing Virtual Bases)

Sử dụng các đế ảo, bạn có thể tạo các mạng phức tạp. Đương nhiên, chúng tôi muốn giữ độ trễ đơn giản, nhưng chúng tôi làm cho chúng phức tạp đến mức nào, thì ngôn ngữ này đảm bảo rằng một csự khởi đầu của một cơ sở ảo được gọi chính xác một lần. Hơn nữa, hàm tạo của một cơ sở (cho dù 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, hàm tạo của mọi cơ sở ảo là 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 sự kiện nếu nó được đề cập ở nhiều nơi trong hệ thống phân cấp lớp. Ví dụ:

*struct V {*

*V (int i);*

*// ...*

*};*

*Struct A {*

*MỘT(); // 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*

*// ...*

*};*

*struct C: ảo V {*

*public:*

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

*// ...*

*};*

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

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

*// ngầm nhận cơ sở ảo A từ B*

*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 trình khởi tạo cho V. Thực tế là V không được đề cập rõ ràng là một cơ sở của D là không liên quan. Kiến thức về một cơ sở ảo và nghĩa vụ khởi tạo nó '' bong bóng lên '' cho lớp dẫn xuất nhất. Một cơ sở ảo luôn được coi là một cơ sở trực tiếp có nguồn gốc nhiều nhất của nó.

Thực tế là cả B và C đều khởi tạo V là không liên quan vì trình biên dịch không có ý tưởng nào trong số hai trình khởi tạo đó để thích. Do đó, chỉ có trình khởi tạo được cung cấp bởi lớp dẫn xuất nhất là đã sử dụng.

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

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

Vấn đề logic này với các hàm tạo không tồn tại đối với các hàm hủy. Chúng chỉ đơn giản là 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 (Calling a Virtual Class Member Once Only)

Khi xác định các hàm cho một lớp có cơ sở ảo, người lập trình nói chung không thể biết liệu cơ sở có được chia sẻ với các lớp dẫn xuất khác hay không. Đây có thể là một vấn đề khi thực hiện- đề cập đến 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, người lập trình có thể mô phỏng lược đồ được sử dụng cho các hàm tạo bằng cách chỉ gọi một hàm lớp cơ sở ảo từ lớp dẫn xuất nhất. Ví dụ, giả sử chúng tacó một lớp Window cơ bản biết cách vẽ nội dung của nó:

*Cửa sổ lớp học {*

*public:*

*// vật liệu cơ bản*

*ảo void draw ();*

*};*

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

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

*// nội dung đường viền*

*protected:*

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

*public:*

*ghi đè void draw ();*

*};*

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

*// menu thứ*

*được bảo vệ:*

*void own\_draw (); // hiển thị menu*

*public:*

*ghi đè void draw ();*

*};*

Các hàm own\_draw () không cần phải ảo vì chúng được gọi từ bên trong một vir- hàm tual draw () '' biết '' loại đối tượng mà nó được gọi.

Từ đó, chúng ta có thể tạo một lớp Đồng hồ hợp lý:

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

*// nội dung đồng hồ*

*protected:*

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

*public:*

*ghi đè void draw ();*

*};*

*hoặc bằng đồ thị:*

*Cửa sổ*

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

*Cái đồng hồ*

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 () được Window :: draw () gọi đúng một lần. Điều này được thực hiện độc lập với loại Cửa sổ mà draw () được gọi:

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

*{*

*Cửa sổ :: draw ();*

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

*}*

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

*{*

*Cửa sổ :: draw ();*

*own\_draw (); // hiển thị menu*

*}*

*void Clock :: draw ()*

*{*

*Cửa sổ :: draw ();*

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

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

*own\_draw (); // hiển thị mặt 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 thế, nó trực tiếp gọi hàm được đặt tên rõ ràng, do đó tránh được đệ quy vô hạn khó chịu.

#### 21.3.6 Cơ sở sao chép so với Cơ sở ảo (Replicated vs Virtual Bases)

Sử dụng đa kế thừa để cung cấp triển khai cho các lớp trừu tượng đại diện cho liên khuôn mặt ảnh hưởngcách một chương trình được thiết kế. Lớp BB\_ival\_slider 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 biệt về mặt logic. Một cơ sở là một bản tóm tắt công khai lớp cung cấp giao diện và lớp kia là lớp cụ thể được bảo vệ cung cấp việc triển khai ‘‘ Chi tiết’’. Những vai trò này được phản ánh trong cả kiểu của các lớp và trong kiểm soát truy cập cung cấp. Việc sử dụng đa kế thừa là điều cần thiết ở đây vì lớp dẫn xuất cần để ghi đè các chức năng ảo từ cả giao diện và quá trình 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 mà đề cập đến nó.

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

*class BB\_ival\_slider*

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

*class Popup\_ival\_slider*

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

*class BB\_popup\_ival\_slider*

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

hoặc bằng đồ thị:

*Ival\_slider*

*Popup\_ival\_slider BB\_ival\_slider*

*BB\_popup\_ival\_slider*

*BBslider*

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*

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

*class Popup\_ival\_slider*

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

*class BB\_popup\_ival\_slider*

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

hoặc bằng đồ thị:

*Ival\_slider Ival\_slider*

*Popup\_ival\_slider BB\_ival\_slider*

*BB\_popup\_ival\_slider*

*BBslider*

Tuy nhiên, có sự khác biệt hợp lý. Trong thiết kế Ival\_slider được nhân rộng, BB\_po pup\_ival\_slider không thể được chuyển đổi hoàn toàn thành Ival\_slider (vì điều đó sẽ không rõ ràng):

*void f (Ival\_slider ∗ p);*

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

*{*

*f (p); // lỗi: Popup\_ival\_slider :: Ival\_slider hay BB\_ival\_slider :: Ival\_slider?*

*}*

Mặt khác, có thể xây dựng các tình huống hợp lý trong đó việc chia sẻ ngụ ý trong thiết kế cơ sở ảo gây ra sự không rõ ràng cho các phôi từ lớp cơ sở .Tuy nhiên, môi trường xung quanh- gu ăn mặc 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 tôi thường không có lựa chọn vì chúng tôi phải tuân theo một thiết kế hiện có.

Ví dụ:

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

*{*

*// ...*

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

*// ...*

*}*

Không cần chuyển đổi rõ ràng để có được từ triển khai (tại đây, BB\_popup\_ival\_slider) thànhgiao diện trực tiếp (tại đây, Popup\_ival\_slider).

##### 21.3.6.1 Ghi đè các hàm cơ sở ảo (Overriding Virtual Base Functions)

Một lớp dẫn xuất có thể ghi đè một hàm ảo của lớp cơ sở ảo trực tiếp hoặc gián tiếp của nó. Cụ thể- lar, hai lớp khác nhau có thể ghi đè các chức năng ảo khác nhau từ cơ sở ảo. Trong đó theo cách, các lớp dẫn xuất eral sev có thể đóng góp các triển khai cho giao diện được trình bày bởi một lớp cơ sở. Ví dụ, lớp Window có thể có các hàm set\_color () và prompt (). Trong đó trường hợp, 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 lên prompt () như một phần kiểm soát các tương tác của người dùng:

*Cửa sổ lớp học {*

*// ...*

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

*dấu nhắc void ảo () = 0;*

*};*

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

*// ...*

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

*};*

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

*// ...*

*vghi đè oid prompt (); // kiểm soát các tương tác của người dùng*

*};*

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

*// ...*

*};*

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

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

*// ...*

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

*};*

hoặc bằng đồ thị:

*Cửa sổ {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 của 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 cuộc gọi độc lập với lớp nào chúng sử dụng làm giao diện. Hoặc, sử dụng thuật ngữ triển khai, không bảng chức năng ảo có thể được xây dựng bởi vì một lệnh gọi đến chức năng đó trên đối tượng hoàn chỉnh sẽ không rõ ràng. Ví dụ, có Radio không được khai báo write (), các khai báo của write () trong Máy thu và Máy phát sẽ gây ra lỗi khi xác định Radio. 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 là 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

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

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

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

[5] Sử dụng tính năng ghi đè để làm cho việc ghi đè trở nên rõ ràng trong các hệ thống phân cấp lớp lớn.

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

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

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

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

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

# Phần 2. Chương trình

## Chương I. Nguyên nhân và mục đích làm chương trình

Nguyên nhân: việc quản lý thông tin sinh viên khá khó khăn nếu chúng ta làm điều đó bằng cách thủ công. Nếu chỉ quản lý một vài sinh viên thì khá đơn giản, nhưng một lớp thì không thể chỉ có một vài sinh viên (trừ một số trường hợp đặc biệt) mà có đến hàng trăm, hay thậm chí chúng ta không chỉ quản lí một lớp có thể còn phải quản lý sinh viên của một ngành hay cả một khóa. Khi quản lý cho nhiều người như vậy, lượng thông tin cần lưu lại rất nhiều , do đó việc nhầm lẫn hay điền sai thông tin là điều không thể tránh khỏi. Nếu xảy ra sai xót thì ta phải làm lại từ đầu.

=> Để khắc phục một số hạn chế này thì nhóm đã quyết định tạo ra một chương trình đáp ứng các khuyết điểm khi tiến hành quản lý sinh viên thủ công.

## Chương II. Cơ sở lý thuyết

### 1. Các tính chất trong lập trình hướng đối tượng

Lập trình hướng đối tượng là gì ? – Là một kỹ thuật lập trình, trong đó cho phép lập trình viên tạo ra các đối tượng bằng code thông qua mô tả một cách trừu tượng hóa các đối tượng trong đời sống.

Mục đích của lập trình hướng đối tượng: tối ưu việc quản lý sourcode, giúp tăng khả năng tái sử dụng và quan trọng hơn hết là giúp tóm gọn các thủ tục đã biết trước tính chất thông qua việc sử dụng các đối tượng.

Có 4 tính chất trong lập trình hướng đối tượng:

* Tính đóng gói (Encapsulation)
* Tính kế thừa (Inheritance)
* Tính đa hình (Polymorphism)
* Tính trừu tượng (Abstraction)

#### 1.1 Tính đóng gói (Encapsuation)

Tính đóng gói cho phép che giấu thông tin và những tính chất xử lý bên trong của đối tượng. Các đối tượng khác không thể tác động trực tiếp đến dữ liệu bên trong và làm thay đổi trạng thái của đối tượng mà bắt buộc phải thông qua các phương thức công khai do đối tượng đó cung cấp.

Ví dụ:

*class Profile{*

*private:*

*string fullname, mssv, province, Class;*

*float gpa;*

*public:*

*float getGpa();*

*string getName();*

*void set\_Gpa() ;*

*};*

Ở ví dụ này, các thuộc tính fullname, mssv, province, Class và gpa nằm trong phạm vi truy cập riêng tư , tức là các đối tượng bên ngoài không thể truy cập hay thay đổi giá trị cho chúng mà chỉ các phương thức bên trong class Profile mới truy xuất được.

Mặc dù không thể truy xuất trực tiếp, ta co thể thông qua phương thức trong class đó để lấy các thuộc tính đó. Như ở trên, các phương thức:

*float getGpa(); // giúp ta lấy được giá trị gpa*

*string getName(); // giúp ta lấy được giá trị fullname*

*void set\_Gpa() ; // giúp ta thay đổi giá trị gpa*

Tính chất này giúp tăng tính bảo mật cho đối tượng và tránh tình trạng dữ liệu bị rò rỉ, hạn chế những truy cập không mong muốn.

#### 1.2 Tính kế thừa (Inheritance)

Tính kế thừa cho phép xây dựng một lớp mới (lớp con), kế thừa và tái sử dụng các thuộc tính, phương thức cũ (lớp cha) đã có trước đó.

Các lớp con kế thừa toàn bộ thành phần của lớp cha và không cần định nghĩa lại.

Ví dụ:

*class Profile{*

*private:*

*float gpa;*

*public:*

*float getGpa();*

*};*

*class Practise:public Profile {*

*private:*

*float practise;*

*public:*

*float getPractise():*

*};*

*class Scholarship : public Practise{*

*private:*

*string namebank;*

*int numberbank;*

*public:*

*bool check(); // kiểm tra điều kiện học bổng*

*};*

*bool Scholarship::check(){*

*if(getGpa() >= 7.9 && getPractise() >= 4 && get\_n\_Allow() < 3)*

*return true;*

*else*

*return false;*

*}*

Ở đây, Scholarship kế thừa *Practise* và *Profile*. Trong *Scholarship* có phương thức *check()* để kiểm tra xem sinh viên đó có đủ điều kiện vào diện được học bổng hay không. Trong hàm này chúng ta không khai báo một đối tượng nào có kiểu dữ kiệu *Profile* hay *Practise* nhưng ta vẫn có thể gọi hai phương thức *getGpa()* và *getPractise()* trong hai class đó , điều nay thể tính kế thừa và ở đây là kế thừa phương thức đã khai báo trước đó.

Không những kế thừa, lớp con có thể mở rộng ra các thành phần kế thừa hoặc bổ sung thành phần mới.

*class Scholarship : public Practise{*

*private:*

*string namebank;*

*int numberbank;*

*public:*

*friend istream& operator >> (istream& is, Scholarship &ip1);*

*friend ostream& operator << (ostream& os, Scholarship &ip1);*

*bool check();*

*void set\_Namebank();*

*void set\_Numberbank();*

*};*

Scholarship còn thêm thuộc tính namebank và numberbank cùng như các phương thức mới *set\_Namebank()* và *set\_Numberbank().*

**1.3 Tính đa hình (Polymorphism)**

Tính đa hình có thể được hiểu là trong hai hay nhiều lớp có những phương thức giống nhau nhưng có thể thực thi theo những cách khác nhau.

*class Profile{*

*private:*

*string fullname, mssv, province, Class;*

*float gpa;*

*public:*

*virtual void Input();*

*virtual void Output();*

*};*

*void Profile::Input(){*

*cout<<"Nhap ho ten sinh vien: ";*

*fflush(stdin);*

*getline(cin,fullname);*

*cout<<endl;*

*cout<<"Nhap ma so sinh vien: ";*

*fflush(stdin);*

*getline(cin, mssv);*

*cout<<endl;*

*cout<<"Nhap lop: ";*

*fflush(stdin);*

*getline(cin, Class);*

*cout<<endl;*

*cout<<"Nhap que cua sinh vien: ";*

*fflush(stdin);*

*getline(cin, province);*

*cout<<endl;*

*cout<<"Nhap diem trung binh: ";*

*cin>>gpa;*

*cout<<endl;*

*}*

*void Profile::Output(){*

*cout<<"Ho va ten: "<<fullname<<endl;*

*cout<<"MSSV: "<<mssv<<endl;*

*cout<<"Lop: "<<Class<<endl;*

*cout<<"Que: "<<province<<endl;*

*cout<<"Diem trung binh: "<<gpa<<endl;*

*}*

*class Practise:public Profile {*

*private:*

*int lesson, allow, n\_allow;*

*float practise;*

*int s\_lesson;*

*public:*

*void Input();*

*void Output();*

*};*

*void Practise::Input(){*

*cout<<"Nhap tong so buoi hoc (1 ki): ";*

*cin>>s\_lesson;*

*cout<<endl;*

*cout<<"Nhap so buoi da hoc: ";*

*cin>>lesson;*

*cout<<endl;*

*cout<<"Nhap so buoi nghi co phep: ";*

*cin>>allow;*

*cout<<endl;*

*cout<<"Nhap so buoi nghi khong phep: ";*

*cin>>n\_allow;*

*cout<<endl;*

*}*

*void Practise::Output(){*

*cout<<"Tong so buoi (1 ki): "<<s\_lesson<<endl;*

*cout<<"So buoi hoc: "<<lesson<<endl;*

*cout<<"Nghi co phep: "<<allow<<endl;*

*cout<<"Nghi khong phep: "<<n\_allow<<endl;*

*}*

*int main(){*

*Profile \*pr, pr1;*

*pr= &pr1; // H1*

*pr->Input(); // H2*

*pr->Output(); // H3*

*pr= new Practise; // H4*

*pr->Input(); // H5*

*pr->Output(); // H6*

*}*

Ở đây, class *Profile* và *Practise* đều có chung tên phương thức là *Input ()* và *Output ()* nhưng lại thực thi theo cách khác nhau và *Practise* kế thừa *Profile.* Khi chạy chương trình, từ dòng H1->H3 nó sẽ thực hiện phương thức *Input ()* và *Output ()* bên trong *class Profile* : nhâp tên , điểm , mssv, ... và in ra tên, điểm, mssv,..... bởi vì *pr* có kiểu là *Profile*. Kể từ dòng H4 *pr* có kiểu *Practise* và nó sẽ gọi đến phương thức *Input ()* và *Output ()* bên trong class *Practise*: nhập buổi học, có phép,... và in ra buổi học, có phép,....

Lưu ý: nếu lớp cha và lớp con có cùng tên phương thức thì phương thức ở lớp cha phải nên là hàm ảo giống ví dụ trên hay là hàm thuần ảo.

* 1. **Tính trừu tương (Abstraction)**

Tính trừu tượng có thể được hiểu là tổng quát hóa một cái gì đó lên, không cần chú ý những chi tiết bên trong .

Ví dụ 1: khi sử dụng xe máy, người dùng chỉ cần biết là tăng ga thì xe sẽ tăng tốc mà không cần phải biết bên trong nó làm thế nào.

Ví dụ 2:

*class Profile{*

*private:*

*string fullname, mssv, province, Class;*

*float gpa;*

*public:*

*friend istream& operator >> (istream& is, Profile &pf1);*

*friend ostream& operator << (ostream& os, Profile &pf1);*

*float getGpa();*

*string getName();*

*void set\_Gpa() ;*

*};*

Khi quản lý sinh viên, chúng ta chỉ quan tâm đến: họ và tên, mssv, quên quán, học lớp nào, hay có thể có tài khoản ngân hàng khi cần thiết. Mà không cần quan tâm: chiều cao, sở thích, quan hệ tình cảm. Những điều này là không quan trọng và cần thiết.

### 2. Cấu trúc dữ liệu - Danh sách liên kết đơn (Single linked list)

#### 2.1 Khái niệm

Danh sách liên kết đơn là tập hợp các Node, được sắp xếp sao cho mỗi Node chứa “một giá trị” (data) và “một con trỏ” (next). Con trỏ sẽ trỏ tới phần tử tiếp theo của danh sách liên kết đó, Node cuối cùng có next trỏ tới NULL.

#### 2.2 Một số phương thức trong danh sách liên kết

Trước hết, chúng ta cần khai báo cấu trúc của một Node (gồm data và con trỏ next).

*struct Node{*

*Student data;*

*Node \*next;*

*};*

*class Student\_Management{*

*private:*

*long size;*

*Node \*head;*

*Node \*tail;*

*public:*

*Student\_Management(); //khởi tạo mặc định*

*Student\_Management(long size, Node \*head, Node \*tail); // khởi tạo đầy đủ*

*Student\_Management(const Student\_Management &slist); //khởi tạo sao chép*

*~Student\_Management(); // hàm hủy*

*Node\* createNode(Student st); // tạo một Node mới*

*void addLast(Student st); //thêm Node vào cuối danh sách*

*Node\* previous(Node \*p); // tìm Node trước Node p*

*void removeFirst(); // xóa Node đầu tròn danh sách*

*void removeLast(); // xóa Node cuối trong danh sách*

*void remove(Node \*p); // xóa một Node bất kỳ*

*void traverse(); // duyệt danh sách liên kết*

*};*

Con trỏ *head* và *tail* dùng để quản lý phần tử đầu tiên và cuối cùng trong danh sách liên kết. *Size* dùng để quản lý số lượng phần tử trong danh sách liên kết.

##### 2.2.1 Hàm khởi tạo và hủy danh sách liên kết

Khởi tạo mặc định:

*Student\_Management::Student\_Management(){*

*head=NULL;*

*tail=NULL;*

*size=0;*

*}*

Khởi tạo đầy đủ:

*Student\_Management::Student\_Management(long size, Node\* head, Node\* tail){*

*this->size=size;*

*this->head=head;*

*this->tail=tail;*

*}*

Khởi tạo sao chép:

*Student\_Management::Student\_Management(const Student\_Management &slist){*

*this->size=slist.size;*

*this->head=slist.head;*

*this->tail=slist.tail;*

*}*

Hàm hủy:

*Student\_Management::~Student\_Management(){*

*delete head;*

*delete tail;*

*size=0;*

*}*

##### 2.2.2 Tạo một Node mới

*Node\* Student\_Management::createNode(Student st){*

*Node \*p= new Node;*

*p->data=st;*

*p->next=NULL;*

*return p;*

*}*

##### 2.2.3 Thêm Node vào cuối danh sách

Nếu head=NULL thì danh sách liên kết đang trống, Node chèn vào sẽ trở thành head.

Ngược lại, ta sẽ duyệt tới Node cuối cùng (Node có next = NULL), và trỏ next của Node cuối tới Node mới.

*void Student\_Management::addLast(Student st){*

*Node \*temp=createNode(st);*

*if(head==NULL){*

*head=tail=temp;*

*}*

*else{*

*tail->next=temp;*

*tail=temp;*

*}*

*size++;*

*}*

##### 2.2.4 Tìm Node trước của một Node

*Node\* Student\_Management::previous(Node \*p) {*

*Node \*t = head;*

*while (t->next != p)*

*t = t->next;*

*return t;*

*}*

##### 2.2.5 Xóa Node đầu tiên trong danh sách

Ta chỉ cần cho Node kế tiếp của head làm head là được (node kế tiếp của head: head->next).

*void Student\_Management::removeFirst(){*

*if(size==0)*

*return;*

*Node\* temp=head;*

*head=head->next;*

*delete temp;*

*size --;*

*}*

##### 2.2.6 Xóa Node cuối trong danh sách

Tương tự xóa đầu.

*void Student\_Management::removeLast(){*

*if(size==0)*

*return;*

*if(size==1){*

*delete head;*

*size--;*

*return;*

*}*

*Node \*p=previous(tail); // tìm Node trước Node tail*

*Node \*temp=tail;*

*p->next=NULL;*

*tail=p;*

*delete temp;*

*size--;*

*}*

##### 2.2.7 Xóa Node bất kỳ

Tương tự xóa đầu nhưng cần tìm Node trước Node cần xóa.

*void Student\_Management::remove(Node \*p) {*

*if (p == head) {*

*removeFirst();*

*return;*

*}*

*if (p == tail) {*

*removeLast();*

*return;*

*}*

*Node \*pre = previous(p);*

*pre->next = p->next;*

*delete p;*

*size--;*

*}*

##### 2.2.8 Duyệt danh sách

*void Student\_Management::traverse(){*

*Node \*p=head;*

*while(p!=NULL){*

*cout<<p->data;*

*p=p->next;*

*}*

*delete p;*

*}*

#### 3. Đọc/ghi file

Dữ liệu được lưu ở biến của chương trình, và nó sẽ biến mất khi chương trình kết thúc. Vì vậy, việc sử dụng file sẽ giúp lưu trữ các dữ liệu cần thiết để đảm bảo của chúng ta không bị mất ngay cả khi chương trình của chúng ta ngừng chạy.

Trước khi làm việc với file cần khai báo thư viện fstream .

*#include<fstream>*

Có 3 kiểu dữ liệu khi làm việc với file:

* Ofstream: dùng để tạo ra file và ghi thông tin vào file đó.
* Ifstream: dùng để đọc thông tin từ file.
* Fstream: cho phép tạo file, ghi thông tin vào file và đọc thông tin từ file.

##### 3.1 Mở một file

Để mở một file, ta sử dụng hàm void open(const char \*ten\_file, ios::che\_do).

Trong đó:

Ten\_file: tên file cần mở.

Che\_do: chế độ thao tác với file.

Có 5 chế độ :

* App: thông tin được thêm vào cuối thông tin đã có.
* Ate: mở một file và di chuyển con trỏ tới cuối file.
* In: mở file để đọc.
* Out: mở file để ghi.
* Trunc: nếu file đã tồn tại thì xóa nội dung trong file trước khi mở nó.

Ví dụ:

*void Student\_Management::traverse(){*

*Node \*p=head;*

*fstream File;*

*File.open("List\_student.txt", ios::out); // mở file List\_student.txt với chế độ out (mở file để ghi)*

*File<<"=====DANH SACH THONG TIN SINH VIEN======"<<endl;*

*while(p!=NULL){*

*cout<<p->data;*

*File<<p->data;*

*p=p->next;*

*}*

*File.close();*

*delete p;*

*}*

Ta có thể kết hợp hai hay nhiều chế độ , ngăn cách các chế độ bằng cách sử dụng “ | “.

Ví dụ :

*File.open("List\_student.txt", ios::out | ios::in); // mở file với chế độ out (mở file để ghi) và chế độ in (đọc file)*

##### 3.2 Đóng file

Ta sử dụng hàm *void close().*

*void Student\_Management::traverse(){*

*Node \*p=head;*

*fstream File;*

*File.open("List\_student.txt", ios::out);*

*File<<"=====DANH SACH THONG TIN SINH VIEN======"<<endl;*

*while(p!=NULL){*

*cout<<p->data;*

*File<<p->data;*

*p=p->next;*

*}*

*File.close(); // đóng file List\_student.txt*

*delete p;*

*}*

##### 3.3 Mở file để đọc/ghi

Để ghi file ta sử dụng “ << “, còn đọc file ta dùng “ >> ”.

Ví dụ:

*void Student\_Management::traverse(){*

*Node \*p=head;*

*fstream File;*

*File.open("List\_student.txt", ios::out);*

*File<<"=====DANH SACH THONG TIN SINH VIEN======"<<endl;*

*while(p!=NULL){*

*cout<<p->data;*

*File<<p->data; // ghi thông tin vào file List\_student.txt*

*// File>>p->data; //đọc thông tin trong file List\_student.txt*

*p=p->next;*

*}*

*File.close();*

*delete p;*

*}*

## Chương III. Khái quát về chương trình

* Các thư viện được sử dụng:

*#include<iostream>*

*#include<string>*

*#include<cmath>*

*#include<fstream>*

* Các class được khai báo trong chương trình:

*class Profile{..};*

*class Practise:public Profile {..};*

*class Scholarship : public Practise{..};*

*class Student: public Scholarship{..};*

*class Student\_Management{..};*

* Các chức năng trong chương trình:

1.Nhập thông tin sinh viên:

*void Student\_Management::list\_student(int &n){*

*Student value;*

*do{*

*cout<<"Nhap so luong sinh vien: ";*

*cin>>n;*

*cout<<endl;*

*}while(n==0);*

*for(int i=0;i<n;i++){*

*cout<<"Sinh vien "<<i+1<<": "<<endl;*

*cin>>value;*

*addLast(value);*

*}*

*}*

* Hàm này cho phép ta nhập thông tin của n sinh viên , sau khi nhập xong thông tin sẽ tự động chèn vào cuối danh sách liên kết.

2.Xuất thông tin của sinh viên đã nhập:

*void Student\_Management::traverse(){*

*Node \*p=head;*

*fstream File;*

*File.open("List\_student.txt", ios::out);*

*File<<"=====DANH SACH THONG TIN SINH VIEN======"<<endl;*

*while(p!=NULL){*

*cout<<p->data;*

*File<<p->data;*

*p=p->next;*

*}*

*File.close();*

*delete p;*

*}*

* Bên cạnh hiển thị thông tin ra màn hình, thì hàm này còn tự động tạo và ghi thông tin vào file list\_student.txt.

3.Xuất thông tin sinh viên thuộc diện học bổng:

*void Student\_Management::list\_sholarship(){*

*fstream File;*

*int count =0;*

*File.open("list\_scholarship.txt", ios::out);*

*File<<"======DANH SACH HOC BONG====== "<<endl;*

*cout<<"======DANH SACH HOC BONG====== "<<endl;*

*Node \*p=head;*

*while(p!=NULL){*

*if(p->data.check()){*

*cout<<p->data;*

*File<<p->data;*

*count++;*

*}*

*p=p->next;*

*}*

*if(count == 0){*

*cout<<"Khong co sinh vien nao dat hoc bong !!"<<endl;*

*}*

*File.close();*

*delete p;*

*}*

* Tương tự như xuất thông tin sinh viên đã nhập, nó cũng tạo và ghi vào file có tên list\_scholarship.txt. Chú ý, điều kiện xét học bổng là: điểm trung bình >=7,9, điểm rèn luyện >= 4, nghỉ không phép < 3, nghỉ có phép < 4.

4.Thêm sinh viên vào danh sách:

*void Student\_Management::add\_student(){*

*int n;*

*cout<<"Nhap so luong sinh vien muon them: ";*

*cin>>n;*

*if(n==0){*

*return;*

*}*

*Student value;*

*for(int i=0; i<n;i++){*

*cout<<"Sinh vien "<<i+1<<endl;*

*cin>>value;*

*addLast(value);*

*}*

*}*

* Hàm này cho phép thêm n sinh viên vào danh sách bằng cách chèn vào cuối danh sách.

5.Xóa sinh viên ra khỏi danh sách:

*void Student\_Management::Delete(){*

*string str;*

*cout<<"Nhap ten muon xoa: ";*

*fflush(stdin);*

*getline(cin, str);*

*cout<<endl;*

*Node \*p=head;*

*while(p!=NULL){*

*if(p->data.getName() == str){*

*remove(p);*

*}*

*p=p->next;*

*}*

*delete p;*

*}*

* Phương thức này sẽ cho phép ta xóa sinh viên bằng cách nhập tên của sinh viên cần xóa.

6.Sửa thông tin sinh viên:

*void Student\_Management::fix\_infor(){*

*string str;*

*cout<<"Nhap ten sinh vien can sua thong tin: ";*

*fflush(stdin);*

*getline(cin, str);*

*cout<<endl;*

*Node \*p=head;*

*while(p!=NULL){*

*if(p->data.getName() == str){*

*int choice;*

*do{*

*cout<<"-------Cac noi dung sua-------"<<endl;*

*cout<<"1. Sua diem trung binh"<<endl;*

*cout<<"2. Sua tong so buoi hoc (1 ki)"<<endl;*

*cout<<"3. Sua so buoi da hoc, nghi co phep, khong phep"<<endl;*

*cout<<"4. Sua ten ngan hang"<<endl;*

*cout<<"5. Sua so tai khoan ngan hang"<<endl;*

*cout<<"Nhan phim 0 de tro lai menu chinh !!"<<endl;*

*cout<<"-------------------------------"<<endl;*

*cout<<"--->Vui long chon chuc nang: ";*

*cin>>choice;*

*cout<<endl;*

*switch(choice){*

*case 1:*

*p->data.set\_Gpa();*

*break;*

*case 2:*

*p->data.set\_s\_lesson();*

*break;*

*case 3:*

*p->data.set\_All();*

*break;*

*case 4:*

*p->data.set\_Namebank();*

*break;*

*case 5:*

*p->data.set\_Numberbank();*

*break;*

*default:*

*choice=0;*

*break;*

*}*

*}while(choice != 0);*

*}*

*p=p->next;*

*}*

*}*

* Chúng ta chỉ cần nhập tên của sinh viên cần sửa , sau đó chỉ cần chọn thông tin muốn sửa bao gồm:
* Sửa điểm trung binh.
* Sửa tổng số buổi học (1 ki).
* Sửa số buổi đã học, nghỉ có phép, không phép.
* Sửa tên ngân hàng.
* Sửa số tài khoản ngân hàng.

7.Tìm kiếm sinh viên:

*void Student\_Management ::searchName(){*

*string str;*

*int count = 0;*

*cout<<"Nhap ten muon tim: ";*

*fflush(stdin);*

*getline(cin, str);*

*cout<<endl;*

*Node\* p=head;*

*while(p!=NULL){*

*if(p->data.getName() == str){*

*cout<<p->data;*

*count++;*

*}*

*p=p->next;*

*}*

*if(count == 0){*

*cout<<"Khong ton tai ten sinh vien muon tim !!"<<endl;*

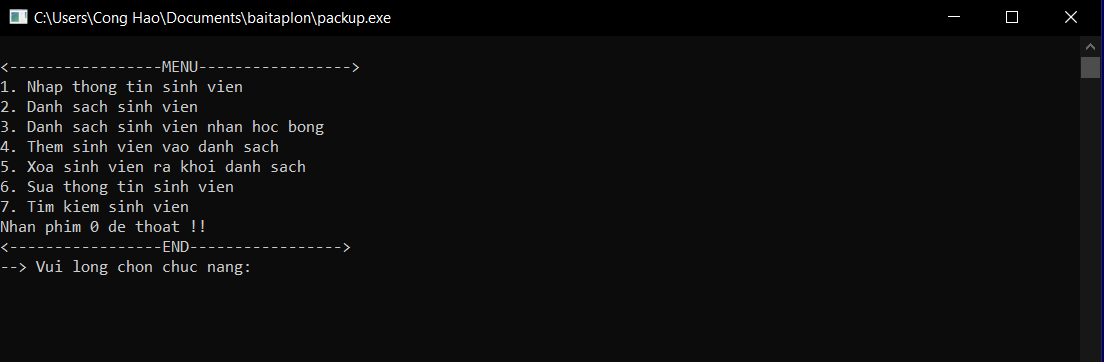
*}*

*delete p;*

*}*

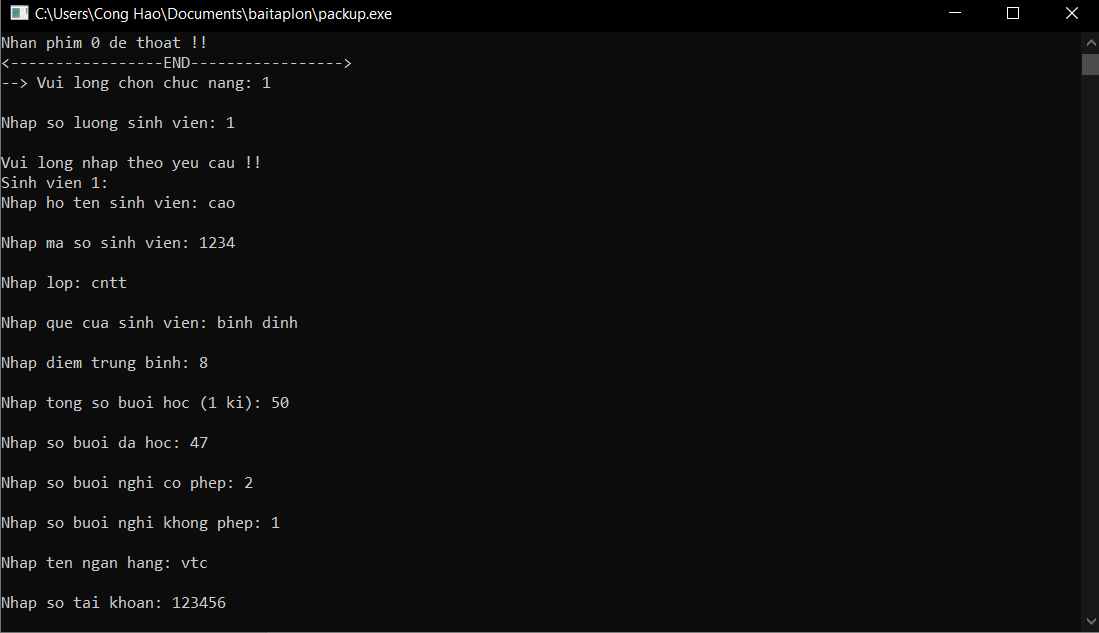
* Hàm sẽ tìm kiếm và hiển thị thông tin của sinh viên bằng tên của sinh viên đó.

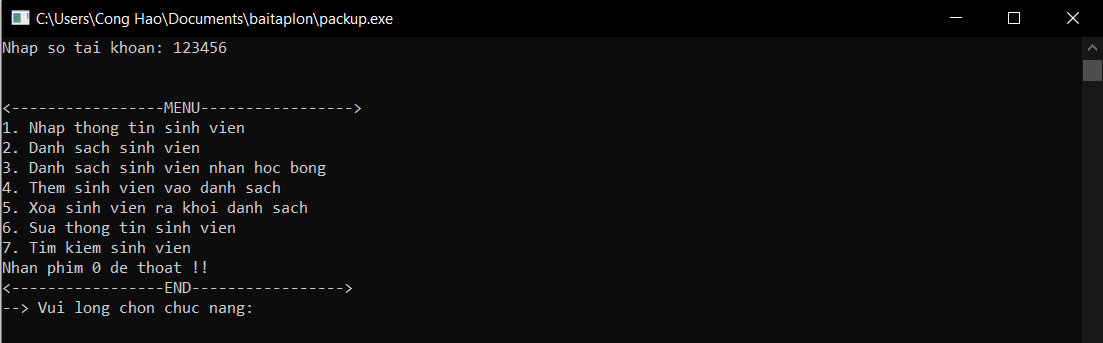
## Chương IV. Hướng dẫn sử dụng

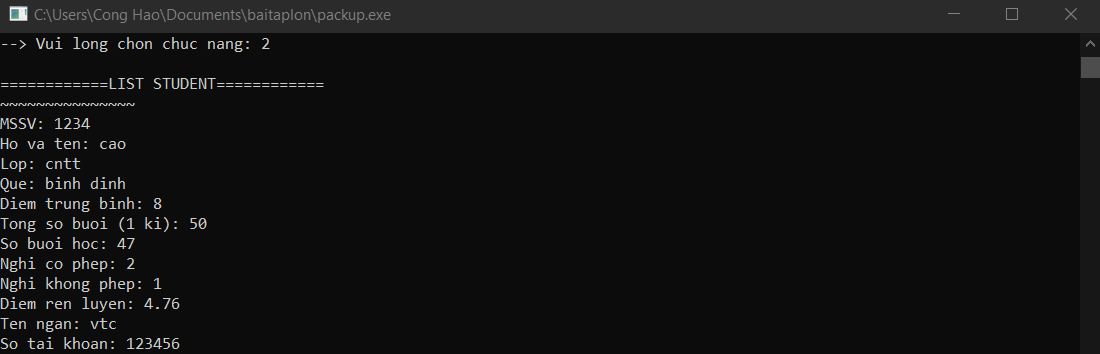
* Khi chạy chương trình, màn hình sẽ có dạng:

Sau đó chọn chức năng mà bạn muốn thực hiện. Ở đây ta sẽ làm từ chức năng 1 -> 7.

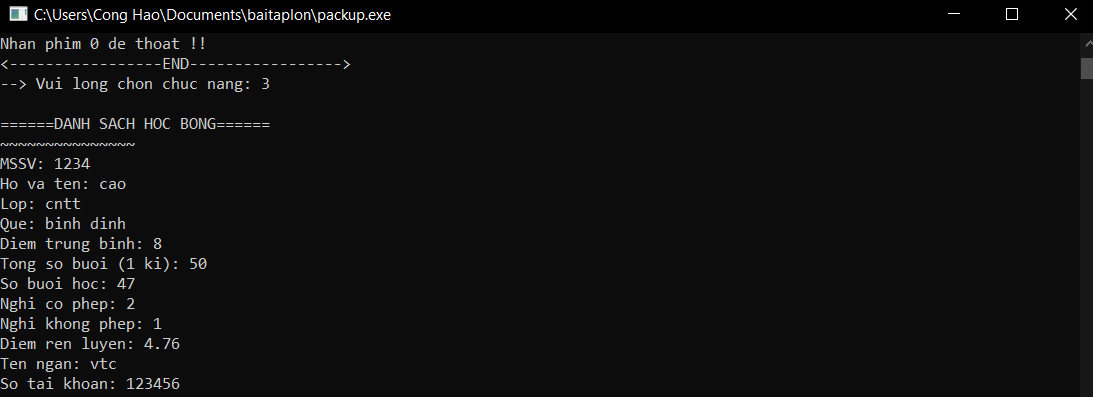
* Đầu tiên, ta nhấn phím 1 rồi Enter. Chương trình chuyển ta đến mục nhập thông tin cho sinh viên (gồm: tên , mã số sinh viên, điểm trung bình, quê quán, lớp, tổng số buổi học (1 kì), số buổi nghỉ có phép, không phép, tên ngân hàng và số tài khoản). Lưu ý, trước khi nhập thông tin, chương trình sẽ yêu cầu bạn nhập số lượng sinh viên nên hãy đọc kỹ yêu cầu trước khi nhập.

Minh họa:

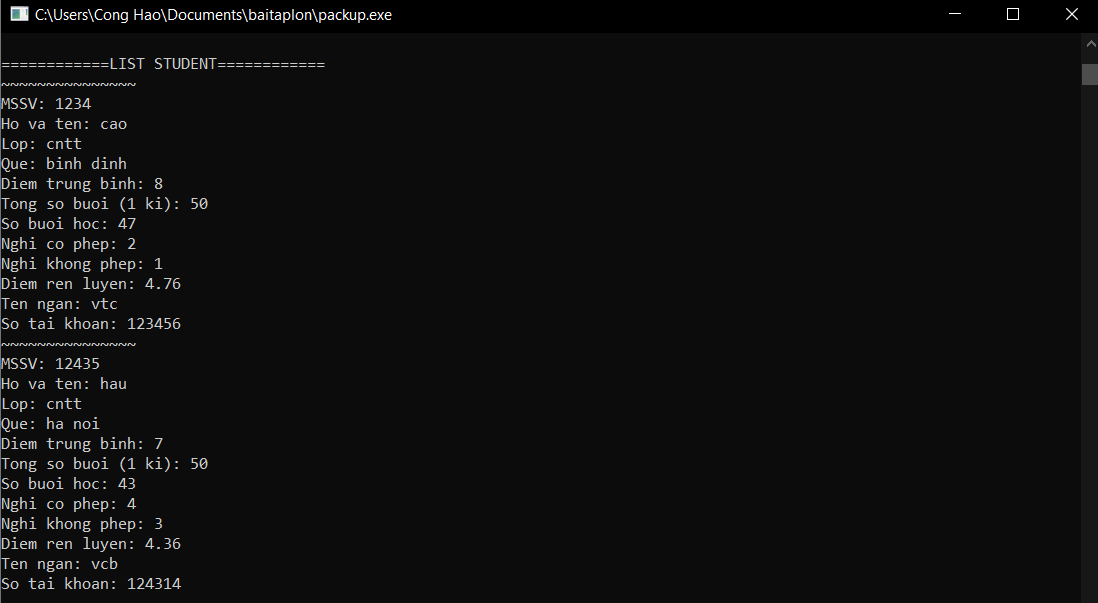
* Sau khi nhập đầy đủ thông tin, chương trình sẽ đưa bạn trở lại menu và chọn chức năng mới.
* Để hiện thị thông tin sinh viên bạn đã nhập, nhấn phím số 2 rồi Enter.

Minh họa:

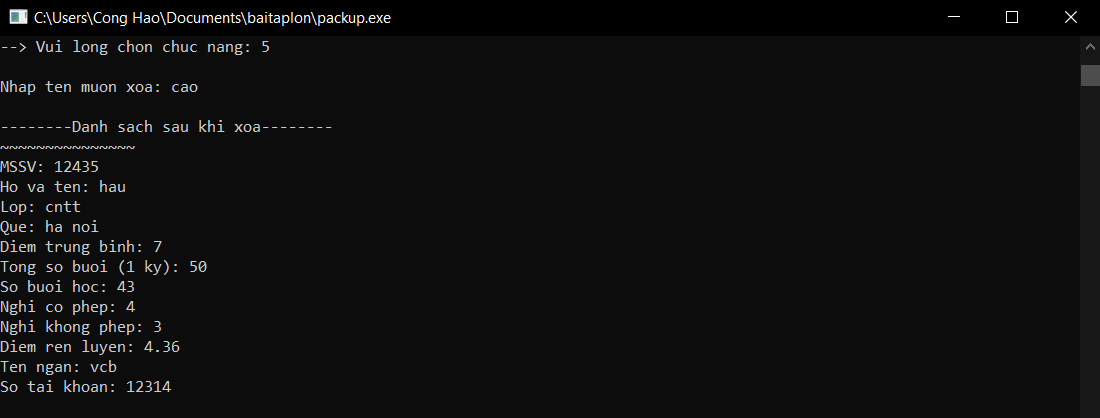
* Bạn có thể kiểm tra thông tin mà chúng ta đã nhập ở minh họa trên. Tương tự chức năng 1, sau khi hiện thị thông tin nó sẽ trở lại menu chính. Các chức năng sau nó cũng tương tự như thế. Lưu ý, điểm rèn luyện tính bằng công thức: nếu như tổng số buổi bạn nghỉ vượt quá 15% tổng số buổi học (1 ky) thì sẽ được 0đ, ngược lại điểm rèn luyện = (tổng buổi học\*5 + nghỉ có phép\*3 – nghỉ không phép\*3)/tổng số buổi học (1 ky).
* Nếu bạn muốn xem sinh viên nào đạt yêu cầu học bổng thì nhấn phím số 3 rồi Enter.

Minh họa:

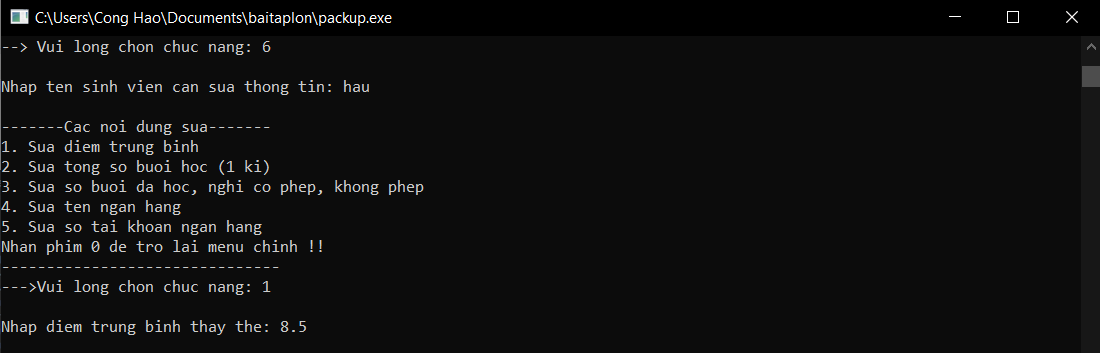
* Bạn muốn quản lý thêm sinh viên, chỉ cần nhấn phím số 4 rồi Enter. Sau đó chỉ cần nhập số lượng sinh viên muốn thêm và nhập thông tin cho sinh viên.

Minh họa:

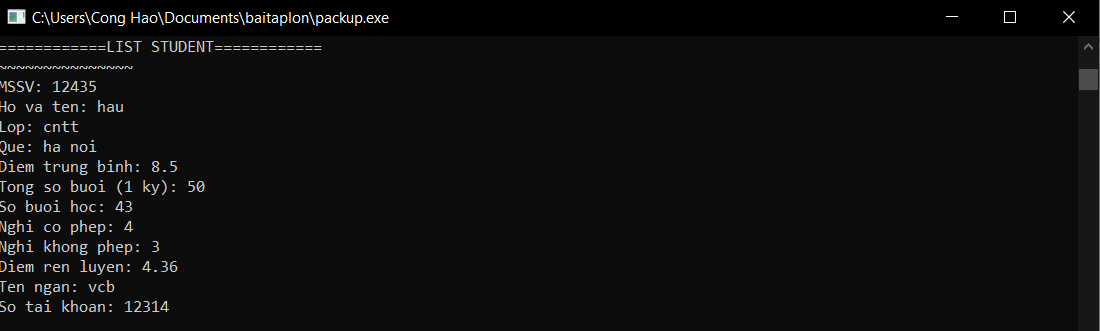
* Ở đây, chúng ta đã thêm một sinh viên mới có tên là “hau” và một số thông tin như trên.
* Trong quá trình học tập, sinh viên đôi khi gặp phải một số vấn đề nào đó nên có thể dẫn đến chuyển ngành, chuyển trường hay thậm chí cả nghỉ học. Vì thế yêu cầu chúng ta phải xóa sinh viên đó ra khỏi danh dách. Để thực hiện việc đó, ta nhấn phím số 5 rồi Enter.

Minh họa:

* Ta đã xóa sinh viên có tên “cao” nhập ở bước đầu tiên ra khỏi danh sách.
* Trong quá trình quản lý hay học tập của sinh viên, ta có thể nhập sai hay sinh viên có một số thông tin thay đổi (điểm trung bình, tên ngân hàng, số tài khoản, tổng số buổi học (1 ky), số buổi nghỉ có phép, không phép).

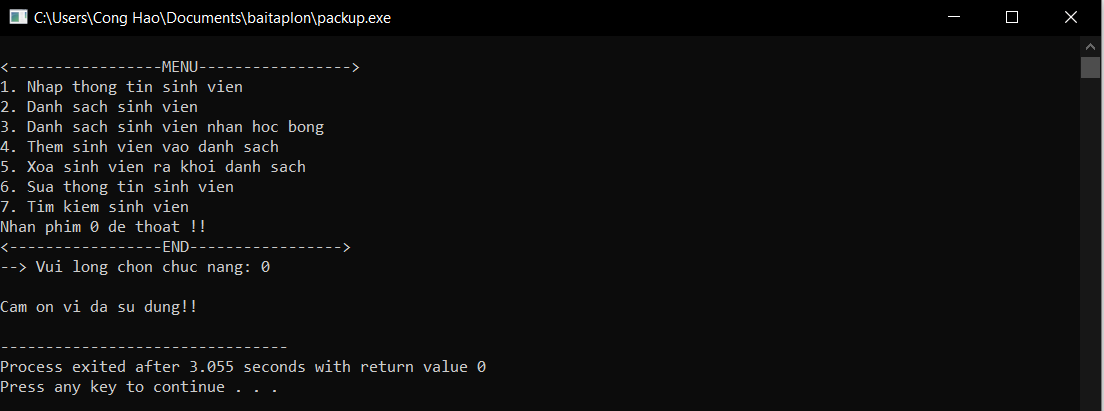
Minh họa:

* Sau khi chọn chức năng sửa, ta chỉ cần nhập tên sinh viên cần sửa thông tin và sau đó chương trình chuyển ta đến danh mục thông tin ta có thể tiến hành sửa. Ở đây, ta sẽ sửa sinh viên tên “hau” ở mục điểm trung bình từ 7đ (có thể xem lại các minnh họa trên) thành 8,5đ.
* Sau khi sửa:

Minh họa:

* Cuối cùng, nếu bạn muốn xem thông tin của bất kỳ sinh viên nào chỉ cần nhấn phím số 7 rồi Enter.

Minh họa:

* Bởi vì chúng ta đã xóa sinh viên tên “cao” ra khỏi danh sách nên nó hiện thông báo” Khồn tồn tại tên sinh viên muốn tìm !!”.
* Sau khi thực hiện xong các công việc, để thoát khỏi chương trình ta nhấn phím số 0 rồi nhấn Enter.

**TÀI LIỆU THAM KHẢO**

[1] Đọc/ghi File trong C++ | fstream trong c++. (2021). Truy cập ngày 14 tháng 11 năm 2021, từ https://vietjack.com/cplusplus/file\_io\_trong\_cplusplus.jsp

[2] Danh sách liên kết đơn | Cài đặt danh sách liên kết đơn trong C/C++. (2021). Truy cập ngày 18 tháng 11 năm 2021, từ https://nguyenvanhieu.vn/danh-sach-lien-ket-don/

[3] Lập trình hướng đối tượng là gì? | Lập trình hướng đối tượng - OOP. (2021). Truy cập ngày 18 tháng 11 năm 2021, từ https://topdev.vn/blog/oop-la-gi/

[4] Stroustrup, B. The C++ programming language.

**Thống kê công việc thành viên**

|  |  |  |
| --- | --- | --- |
| Loại công việc  Thành viên | Báo cáo | Chương trình |
| Văn Công Hào | Dịch sách: chapter 16, chapter 17.  Phần 2 chương trình.  Chỉnh sửa bố cục, nội dung hoàn thành báo cáo. | Phần Profile, tổng hợp chỉnh sửa code, xuất sinh viên được học bổng, làm menu chương trình, thêm sinh viên, sửa thông tin sinh viên. |
| Trần Tấn Quốc | Dịch sách: chapter 20, chapter 21.  Tổng hợp các bài dịch sách.  Chỉnh sửa powerpoint. | Phần scholarship, tạo một Node, xóa node đầu, xóa node cuối, xóa node bất kỳ. |
| Trần Quang Thắng | Dịch sách: chapter 18, chapter 19.  Thiết kế poweroint.  Chỉnh sửa phần dịch sách. | Phần Practise, nhập thông tin của n sinh viên, xuất thông tin n sinh viên, tìm kiếm sinh viên. |

*Hồ Chí Minh, ngày 14 tháng 11 năm 2021*