

[This Photo](http://www.freebordersandclipart.com/BordersPages/CornerInlayBorders.htm) by Unknown Author is licensed under [CC BY-SA](https://creativecommons.org/licenses/by-sa/3.0/)

[This Photo](http://www.freebordersandclipart.com/BordersPages/CornerInlayBorders.htm) by Unknown Author is licensed under [CC BY-SA](https://creativecommons.org/licenses/by-sa/3.0/)

**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐH GIAO THÔNG VẬN TẢI PH- TP HCM**

**Năm học: 2020-2021**



**BÁO CÁO BÀI TẬP LỚN**

**CHỦ ĐỀ:** Quản lí sinh viên

**Nhóm: 20**

**Nguyễn Ngọc Lễ(NT)**

**Trương Quang Duy**

**Nguyễn Thành Liêm**

🙢🕮🙠

PHỤ LỤC

[I) DỊCH SÁCH 3](#_Toc88233665)

[CHƯƠNG 16 3](#_Toc88233666)

[Các loại toán tử 8](#_Toc88233667)

[CHƯƠNG 17 9](#_Toc88233668)

[Xây dựng, dọn dẹp, sao chép và di chuyển 9](#_Toc88233669)

[CHƯƠNG 18 17](#_Toc88233670)

[Operato Overloading 17](#_Toc88233671)

[Giới thiệu 17](#_Toc88233672)

[CHƯƠNG 19 : SEPICAL OPERATORS 30](#_Toc88233673)

[CHƯƠNG 20: CÁC LỚP CÓ NGUỒN GỐC. 39](#_Toc88233674)

[CHƯƠNG 21: Lớp Phân Cấp 70](#_Toc88233675)

[II)CHƯƠNG TRÌNH ỨNG DỤNG 92](#_Toc88233676)

# **DỊCH SÁCH**

## **CHƯƠNG 16**

**CLASS**

**16.1 Giới thiệu**

Các lớp C ++ là một công cụ để tạo các kiểu mới có thể được sử dụng thuận tiện như các kiểu tích hợp sẵn. Ngoài ra, các lớp dẫn xuất (§3.2.4, Chương 20) và mẫu (§3.4, Chương 23) cho phép người lập trình thể hiện các mối quan hệ (hierachical và parametric) giữa các lớp và tận dụng các mối quan hệ đó.

Một kiểu là một đại diện cụ thể của một khái niệm (một ý tưởng, một khái niệm, v.v.). Ví dụ, kiểu float tích hợp trong C ++ với các phép toán +, -, ∗, v.v., cung cấp một giá trị gần đúng cụ thể về khái niệm toán học của một số thực. Một lớp là một kiểu do người dùng định nghĩa. Chúng tôi thiết kế một kiểu mới để cung cấp định nghĩa về một khái niệm không có đối tác trực tiếp giữa các kiểu được tích hợp sẵn. Ví dụ: chúng tôi có thể cung cấp loại Trunk\_line trong chương trình xử lý điện thoại, loại Explosion cho trò chơi điện tử hoặc danh sách loại <Paragraph> cho chương trình xử lý văn bản. Một chương trình cung cấp các kiểu phù hợp chặt chẽ với các khái niệm của ứng dụng có xu hướng dễ hiểu hơn, dễ lý luận hơn và dễ sửa đổi hơn một chương trình không có.

Một tập hợp các kiểu do người dùng xác định được lựa chọn tốt cũng làm cho chương trình trở nên ngắn gọn hơn. Ngoài ra, nó làm cho nhiều loại phân tích mã khả thi. Đặc biệt, nó cho phép trình biên dịch phát hiện việc sử dụng bất hợp pháp các đối tượng mà nếu không thì chỉ được tìm thấy thông qua kiểm tra toàn diện. Ý tưởng cơ bản trong việc xác định một kiểu mới là tách các chi tiết ngẫu nhiên của việc triển khai (ví dụ: cách bố trí dữ liệu được sử dụng để lưu trữ một đối tượng của kiểu) khỏi các thuộc tính cần thiết cho việc sử dụng nó một cách chính xác (ví dụ: danh sách đầy đủ của các chức năng có thể truy cập dữ liệu). Sự tách biệt như vậy được thể hiện tốt nhất bằng cách phân luồng tất cả các mục đích sử dụng cấu trúc dữ liệu và các quy trình vệ sinh nội bộ của nó thông qua một giao diện cụ thể.

**16.2 Cơ bản về Class**

Dưới đây là một bản tóm tắt rất ngắn gọn về các lớp:

• Một lớp là một kiểu do người dùng định nghĩa.

• Một lớp bao gồm một tập hợp các thành viên. Các loại thành viên phổ biến nhất là thành viên dữ liệu và chức năng thành viên.

• Các hàm thành viên có thể xác định ý nghĩa của khởi tạo (tạo), sao chép, di chuyển và dọn dẹp (phá hủy).

• Các thành viên được truy cập bằng cách sử dụng. (dấu chấm) cho các đối tượng và -> (mũi tên) cho con trỏ.

• Các toán tử, chẳng hạn như +,!, Và [], có thể được định nghĩa cho một lớp.

• Một lớp là một không gian tên chứa các thành viên của nó.

• Các thành viên công khai cung cấp giao diện của lớp và các thành viên riêng cung cấp chi tiết.

• Một cấu trúc là một lớp mà các thành viên là công khai theo mặc định.

For example:

class X {

private: // khai báo kiểu riêng tư

int m;

public: // giao diện người dùng là công khai

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

int mf(int i) // một chức năng thành viên

{

int old = m;

m = i; // đặt một giá trị mới

return old; // trả về giá trị cũ

}

};

X var {7}; // một biến kiểu X, được khởi tạo thành 7

int user(X var, X∗ ptr)

{

int x = var.mf(7); // truy cập bằng cách sử dụng. (dấu chấm)

int y = ptr−>mf(9); // truy cập bằng cách sử dụng ->(mũi tên)

int z = var.m; // lỗi: không thể truy cập thành viên riêng tư

}

**16.2.1 Chức năng thành viên**

Xem xét việc triển khai khái niệm ngày bằng cách sử dụng cấu trúc (§2.3.1, §8.2) để xác định biểu diễn của Ngày và một tập hợp các hàm để thao tác với các biến kiểu này:

struct Date {

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

Không có kết nối rõ ràng nào giữa kiểu dữ liệu, Ngày tháng và các hàm này. Một kết nối như vậy có thể được thiết lập bằng cách khai báo các chức năng là thành viên:

struct Date {

int d, m, y;

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

};

Các hàm được khai báo trong định nghĩa lớp (struct là một loại lớp; §16.2.4) được gọi là các hàm thành viên và chỉ có thể được gọi cho một biến cụ thể của kiểu thích hợp bằng cách sử dụng cú pháp chuẩn để truy cập thành viên cấu trúc (§8.2). Ví dụ:

Date my\_birthday;

void f()

{

Date today;

today.init(16,10,1996);

my\_birthday.init(30,12,1950);

Date tomorrow = today;

tomorrow.add\_day(1);

// ...

}

Bởi vì các cấu trúc khác nhau có thể có các hàm thành viên có cùng tên, chúng ta phải chỉ định tên cấu trúc khi xác định một hàm thành viên:

void Date::init(int dd, int mm, int yy)

{

d = dd;

m = mm;

y = yy;

}

**16.2.2 Sao chép mặc định**

Theo mặc định, các đối tượng có thể được sao chép. Đặc biệt, một đối tượng lớp có thể được khởi tạo bằng một bản sao của một đối tượng thuộc lớp của nó. Ví dụ:

Date d1 = my\_birthday;

Date d2 {my\_birthday};

**16.2.3 Kiểm soát truy cập**

Khai báo Ngày trong mục nhỏ trước khi cung cấp một tập hợp các hàm để thao tác với Ngày. Tuy nhiên, nó không chỉ rõ rằng những hàm đó phải là những hàm duy nhất phụ thuộc trực tiếp vào biểu diễn của Date và những hàm duy nhất để truy cập trực tiếp vào các đối tượng của lớp Date. Hạn chế này có thể được thể hiện bằng cách sử dụng một lớp thay vì một cấu trúc:

class Date {

int d, m, y;

public:

void init(int dd, int mm, int yy);

void add\_year(int n);

void add\_month(int n);

void add\_day(int n);

};

**16.2.4 Lớp và Cấu trúc**

Xây dựng:

class X { ... };

được gọi là định nghĩa lớp; nó định nghĩa một kiểu gọi là X. Vì lý do lịch sử, một định nghĩa lớp thường được coi là một khai báo lớp. Ngoài ra, giống như các khai báo không phải là định nghĩa, định nghĩa lớp có thể được sao chép trong các tệp nguồn khác nhau bằng cách sử dụng #include mà không vi phạm quy tắc một định nghĩa.

Theo định nghĩa, một cấu trúc là một lớp trong đó các thành viên được mặc định là công khai; đó là:

struct S { /\* ... \*/ };

chỉ đơn giản là viết tắt của

class S { public: /\* ... \*/ };

**16.2.6 Trình tạo rõ ràng**

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

complex<double> d {1};

**16.2.7 Bộ khởi tạo trong lớp**

Khi chúng ta sử dụng một số hàm tạo, việc khởi tạo thành viên có thể trở nên lặp đi lặp lại. Ví dụ:

class Date {

int d, m, y;

public:

Date(int, int, int);

Date(int, int);

Date(int);

Date();

Date(const char∗);

// ...

};

Chúng ta có thể giải quyết vấn đề đó bằng cách giới thiệu các đối số mặc định để giảm số lượng hàm tạo. Ngoài ra, chúng ta có thể thêm trình khởi tạo vào các thành viên dữ liệu:

class Date {

int d {today.d};

int m {today.m};

int y {today.y};

public:

Date(int, int, int);

Date(int, int);

Date(int);

Date();

Date(const char∗);

// …

Bây giờ, mỗi hàm 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 xem Ngày có hợp lệ không

}

**16.2.8 Định nghĩa hàm trong lớp**

Một hàm thành viên của một lớp là một hàm mà có định nghĩa hoặc prototype của nó bên trong định nghĩa lớp giống như bất kỳ biến nào khác. Nó hoạt động trên bất kỳ đối tượng nào của lớp mà nó là một thành viên, và có sự truy cập tới tất cả thành viên của một lớp cho đối tượng đó.

**16.2.9 Mutability**

Mutable Object: khi khởi tạo 1 đối tượng, tức ta có 1 tham chiếu tới 1 thể hiện của 1 lớp, thì trạng thái của đối tượng có thể thay đổi được sau khi việc khởi tạo đối tượng thành công. (Trạng thái đối tượng ở đây có thể là các trường thông tin mà đối tượng đó nắm giữ. Ví dụ: tên, tuổi của 1 đối tượng sinh viên chẳng hạn).

**16.2.10 Const member functions**

Const member functions là function không thể làm thay đổi trạng thái hiện tại của object, nói cách khác nó không thể thay đổi các member variable khác của class!

**16.2.9.2 Hằng số vật lý và lôgic**

Đôi khi, một hàm thành viên về mặt logic là const, 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 Ngày 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, trừ khi giá trị của Ngày 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 Ngày:

class Date {

public:

string string\_rep() const;

private:

bool cache\_valid;

string cache;

void compute\_cache\_value();

};

**16.2.9.3 mutable**

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

**16.2.9.4 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.

**16.2.10 Self-Reference**

Các hàm cập nhật trạng thái add\_year (), add\_month () và add\_day () (§16.2.3) được định nghĩa không 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.

**16.2.11 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.

**16.2.12 [static] Members**

**Static member** là những thành phần được khởi tạo và cấp phát vào 1 vùng nhớ cố định. Thông thường 1 thuộc tính chỉ có thể được sử dụng khi có 1 object đại diện được khởi tạo, tuy nhiên với **static member**, ta có thể sử dụng mà không cần phải khai báo 1 object nào cả.

**16.2.13 Member Types**

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 có), nhưng không có khái niệm về đối tượng hiện tại của lớp bao quanh.

**16.3 Concrete Classes**

**16.3.1 Member Functions**

**16.3.2 Helper Functions**

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

**16.3.3 Overloaded Operators**

Nạp chồng toán tử (Operator Overloading) được dùng để định nghĩa toán tử cho có sẵn trong c++ phục vụ cho dữ liệu riêng do bạn tạo ra.  
Giả sử có lớp PhanSo và có các phương thức tính toán như Cong, Tru, Nhan, Chia.  
Nếu gặp một biểu thức phức tạp, số lượng phép tính nhiều thì việc sử dụng các phương thức trên khá khó khăn và có thể gây rối cho người lập trình. Vì thế ta sẽ nạp chồng lại các toán tử để có thể tạo một cái nhìn trực quan vào code, giảm thiểu các lỗi sai không đáng có.

### Các loại toán tử

* C++ chỉ cho phép người dùng overloading lại các toán tử có sẵn trong c++
* Một toán tử có thể được định nghĩa cho nhiều kiểu dữ liệu khác nhau.

**16.3.4 The Significance of Concrete Classes**

**16.4 Advice**

## **CHƯƠNG 17**

# **Xây dựng, dọn dẹp, sao chép và di chuyển**

**17.2 Constructors and Destructors**

Chúng ta có thể chỉ định cách khởi tạo một đối tượng của một lớp bằng cách định nghĩa một phương thức khởi tạo (§16.2.5, §17.3). Để bổ sung cho các hàm tạo, chúng ta có thể xác định một hàm hủy để đảm bảo '' dọn dẹp '' tại điểm phá hủy một đối tượng (ví dụ: khi nó vượt ra khỏi phạm vi). Một số kỹ thuật hiệu quả nhất để quản lý tài nguyên trong C ++ dựa trên các cặp hàm tạo / hủy. Các kỹ thuật khác cũng vậy dựa trên một cặp hành động, chẳng hạn như thực hiện / hoàn tác, bắt đầu / dừng, trước / sau, v.v.

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 tôi có thể thử gọi:

f ({2,3,5});

Điều này sẽ in ra luồng ghi nhật ký:

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 Constructors and Invariants**

**Constructor** là một loại hàm thành viên đặc biệt của class, được gọi tự động khi một đối tượng của class đó được khởi tạo. Các constructors thường được sử dụng để khởi tạo các biến thành viên của class theo các giá trị mặc định phù hợp hoặc do người dùng cung cấp, hoặc để thực hiện bất kỳ các bước thiết lập cần thiết nào cho class (ví dụ: Mở file hoặc cơ sở dữ liệu).

Không giống như các hàm thành viên thông thường, các hàm constructors có các quy tắc riêng về việc đặt tên:

* Các hàm constructors phải có cùng tên với tên class (phải giống cả về việc ký tự viết hoa hay viết thường)
* Các hàm constructor không có kiểu trả về (kể cả là kiểu void)

**17.2.2 Destructors and Resources**

Một hàm tạo khởi tạo một đối tượng. Nói cách khác, nó tạo ra môi trường mà các chức năng thành viên hoạt động. Đôi khi, việc tạo ra môi trường đó liên quan đến việc lấy một tài nguyên - chẳng hạn như tệp, khóa hoặc một số bộ nhớ phải được giải phóng sau khi sử dụng (§5.2, §13.3). Do đó, một số lớp cần một hàm được đảm bảo sẽ được gọi khi một đối tượng bị hủy theo cách tương tự như cách một phương thức khởi tạo được đảm bảo sẽ được gọi khi một đối tượng được tạo. Không thể tránh khỏi, một hàm như vậy được gọi là hàm hủy. Tên của hàm hủy là ̃ theo sau là tên lớp, ví dụ ~ Vector (). Một ý nghĩa của là ‘‘ bổ sung ’’ (§11.1.2) 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 định khi một biến tự động vượt ra khỏi phạm vi, một đối tượng trên cửa hàng miễn phí bị xóa, v.v. Chỉ trong những trường hợp rất hiếm, người dùng mới cần gọi hàm hủy một cách rõ ràng.

Hàm hủy có thể rất hữu ích để giải phóng resource trước khi thoát khỏi chương trình, ví dụ: đóng file, giải phóng bộ nhớ…

**17.2.3 Base and Member Destructors**

Hàm tạo và hàm hủy tương tác chính xác với cấu trúc phân cấp lớp (§3.2.4, Chương 20). 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 tự thực thi cơ thể của mình. 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 hàm hủy thành viên của nó, và

[3] cuối cùng, nó gọi các hàm hủy lớp cơ sở của nó.

Đặc biệt, một căn cứ ảo được xây dựng trước bất kỳ căn cứ nào có thể sử dụng nó và bị giải phóng sau tất cả những căn cứ đó (§21.3.5.1). 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 giải phóng.

Lập trình viên có thể đánh bại quy tắc đơn giản và thiết yếu này, nhưng chỉ thông qua việc cố ý phá vỡ liên quan đến việc chuyển con trỏ đến các biến chưa được khởi tạo làm đối số. Làm như vậy vi phạm các quy tắc ngôn ngữ và kết quả thường là thảm hại.

Các hàm tạo thực thi các hàm khởi tạo thành viên và 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 chi phí 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, thì 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 ""

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 {

X x;

};

S2 x1; // error :

S2 x2 {1}; // OK: x2.x is initialized with 1

**17.2.4 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 rõ ràng một hàm hủy một cách rõ ràng; làm như vậy sẽ dẫn đến các lỗi khó chịu. Tuy nhiên, có

các trường hợp hiếm (nhưng quan trọng) trong đó trình 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 ()).

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

{

// ...

new(p) X{a};

// ...

}

Việc sử dụng hàm tạo này được gọi là ‘‘placement new’’ (§11.2.4).

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

void C::pop\_back()

{

// ...

p−> ̃X(); // hủy X trong địa chỉ p

}

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

Để có ví dụ đầy đủ hơn về quản lý rõ ràng các đối tượng trong vùng bộ nhớ, hãy xem §13.6.1.

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 = xóa (§17.6.4) hoặc private.

Trong số hai lựa chọn thay thế, sử dụng private là linh hoạt 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(); } // phá hủy rõ ràng

private:

// ...

̃Nonlocal(); // đừng phá hủy một cách ngầm

};

void user()

{

Nonlocal x; //lỗi: không thể hủy bỏ một Nonlocal

X∗ p = new Nonlocal; // OK

// ...

delete p; //lỗi: không thể hủy bỏ một Nonlocal

p.destroy(); // OK

}

**17.2.5 virtual Destructors**

Trong một lớp thì Destructor có thể được đánh dấu làm hàm ảo còn Constructor thì không được đánh dấu là hàm ảo.

virtual Product(); // illegalvirtual ~Product(); // legal

**17.3 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 để chấp nhận danh sách bộ 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 Initialization Without Constructors**

Chúng ta không thể định nghĩa một phương thức khởi tạo cho một kiểu dựng sẵn, nhưng chúng ta có thể khởi tạo nó với một giá trị của kiểu phù hợp.

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 (không có bộ khởi tạo hoặc có danh sách bộ khởi tạo trống).

**17.3.2 Initialization Using Constructors**

Trong trường hợp bản sao thành viên là không đủ hoặc không mong muốn, một phương thức khởi tạo có thể được định nghĩa 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 (§17.2.1).

Nếu một phương thức khởi tạo được khai báo cho một lớp, một số phương thức khởi 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.

**17.3.2.1 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 (§17.3.4) mà ký hiệu {} cũng cung cấp.

**17.3.3 Default Constructors**

#include <iostream>

using namespace std;

class NoConstructorsAtAll {

public:

int i;

float f;

void Display(void) { cout << "i=" << i << ",f=" << f << endl; }

};

int main(void) {

NoConstructorsAtAll o1;

NoConstructorsAtAll \*o2;

o2 = new NoConstructorsAtAll;

o1.Display();

o2 -> Display();

return 0;

}

Lớp *NoConstructorsAtAll* trong ví dụ này là một lớp mà không có constructor nào. Một lớp như thế này thể hiện ý của developer kiểu như: “tôi không cần sử dụng hàm khởi tạo của lớp này”. Ý muốn đó sẽ được thừa nhận một phần – lớp này sẽ được trình biên dịch tự động trang bị cho một constructor mặc định, không có tham số và cũng không làm gì cả.

Hãy xem xem kĩ ví dụ. Lớp NoConstructorsAtAll không có hàm tạo nên kết quả là các trường dữ liệu của nó sẽ không được khởi tạo bằng bất cứ cách nào. Các giá trị được xuất ra bởi hàm display hoàn toàn là các giá trị ngẫu nhiên. Kết quả hiển thị ra màn hình sẽ không lặp lại khi bạn chạy chương trình.

Một trong những kết quả khi chúng tôi chạy chương trình là như sau:

I=-8589, f=-1.07374e+08

I=-842150451, f=-4.31602e+08

**17.3.4 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. Phương thức khởi tạo danh sách khởi tạo được sử dụng để xây dựng các đối tượng bằng cách sử dụng {} -list làm giá trị trình khởi tạo của nó. Các vùng chứa thư viện tiêu chuẩn (ví dụ: vectơ và bản đồ) có các hàm tạo, phép gán cho danh sách khởi tạo, v.v. (§31.3.2, §31.4.3).

**17.3.4.1 initializer\_list Constructor Disambiguation**

Khi bạn có các hàm tạo sev eral cho một lớp, các quy tắc giải quyết quá tải thông thường (§12.3) được sử dụng để chọn đúng cho một tập hợp các đối số nhất định. Để chọn một phương thức khởi tạo, danh sách mặc định và trình khởi tạo sẽ được ưu tiên.

struct X {

X(initializer\_list<int>);

X();

X(int);

};

X x0 {}; // danh sách trống: phương thức khởi tạo mặc định hay phương thức khởi tạo danh sách? (hàm tạo mặc định)

X x1 {1}; // một số nguyên: một đối số int hay danh sách một phần tử? (phương thức khởi tạo danh sách khởi tạo)

Các quy tắc là:

• Nếu một hàm tạo mặc định hoặc một hàm khởi tạo danh sách 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.

**17.3.4.2 Use of initializer\_lists**

**17.3.4.3 Direct and Copy Initialization**

Sự khác biệt giữa khởi tạo trực tiếp và khởi tạo sao chép (§16.2.6) 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 bộ 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

**17.4 Member and Base Initialization**.

Các nhà xây dựng có thể thiết lập các bất biến và thu được các tài nguyên. Nói chung, chúng làm điều đó bằng cách khởi tạo các thành viên lớp và các lớp cơ sở.

**17.4.1 Member Initialization**

**17.4.1.1 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.

Thành viên tham chiếu hoặc thành viên const phải được khởi tạo (§7.5, §7.7, §17.3.3). 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. Trong trường hợp đó, tôi thường thích sử dụng cú pháp của trình khởi tạo thành viên để làm rõ rằng quá trình khởi tạo đang được thực hiện. Thông thường, cũng có một lợi thế hiệu quả khi sử dụng cú pháp trình khởi tạo (so với việc sử dụng một phép gán).

**17.4.2 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, thì 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.

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

**17.4.3 Delegating Constructors**

Nếu bạn muốn hai hàm tạo thực hiện cùng một hành động, bạn có thể tự lặp lại hoặc xác định ‘‘ hàm init () ’’ để thực hiện hành động chung. Cả hai '' giải pháp '' đều phổ biến (vì các phiên bản C ++ cũ hơn không cung cấp bất kỳ điều gì tốt hơn).

**17.4.4 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.

class A {

public:

int a {7};

int b = 77;

};

Vì những lý do kỹ thuật khá khó hiểu liên quan đến phân tích cú pháp và tra cứu tên, ký hiệu {} và = khởi tạo có thể được sử dụng cho bộ khởi tạo thành viên trong lớp, nhưng ký hiệu () thì không.

**17.4.5 static Member Initialization**

Một thành viên lớp tĩnh được cấp phát tĩnh chứ không phải là một phần của mỗi đối tượng của lớp. Nói chung, khai báo thành viên tĩnh hoạt động như một khai báo cho một định nghĩa bên ngoài lớp.

**17.5 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 trạng thái đã chuyển từ. Đối với các trường hợp thú vị nhất, vùng chứa, trạng thái chuyển từ đó là '' trống ''.

Sự phân biệt logic đơn giản này bị nhầm lẫn bởi truyền thống và thực tế là chúng ta sử dụng cùng một ký hiệu-

tion cho cả di chuyển và sao chép.

Thông thường, một nước đi không thể ném, trong khi một bản sao có thể (vì nó có thể cần lấy tài nguyên), và một nước đi thường hiệu quả hơn một bản sao. Khi bạn viết một thao tác di chuyển, bạn nên để đối tượng nguồn ở trạng thái hợp lệ nhưng không xác định vì cuối cùng nó sẽ bị hủy và trình hủy không thể hủy đối tượng còn lại ở trạng thái không hợp lệ. Ngoài ra, các thuật toán thư viện tiêu chuẩn dựa vào việc có thể gán cho (sử dụng di chuyển hoặc sao chép) một đối tượng được chuyển đến. Vì vậy, hãy thiết kế các bước di chuyển của bạn để không ném và để các đối tượng nguồn của chúng ở trạng thái cho phép phá hủy và chuyển nhượng.

Để giúp chúng ta thoát khỏi công việc lặp đi lặp lại tẻ nhạt, hãy sao chép và di chuyển các định nghĩa mặc định (§17.6.2).

**17.5.1 Copy**

Sao chép cho một lớp X được xác định bằng hai phép toán:

• Sao chép hàm tạo: X (const X &)

• Sao chép phép gán: X & operator = (const X &)

Bạn có thể xác định hai phép toán này bằng các loại đối số mạo hiểm hơn, chẳng hạn như X dễ bay hơi &, nhưng không; bạn sẽ chỉ gây nhầm lẫn cho chính mình và những người khác. Một phương thức khởi tạo sao chép được cho là tạo một bản sao của một đối tượng mà không cần sửa đổi nó. Tương tự, bạn có thể sử dụng const X & làm kiểu trả về của phép gán bản sao. Ý kiến của tôi là làm như vậy gây ra nhiều nhầm lẫn hơn mức đáng có, vì vậy cuộc thảo luận của tôi về bản sao giả định rằng hai phép toán có các kiểu thông thường**.**

**17.5.1.1 Beware of Default Constructors**

Khi viết một thao tác sao chép, hãy đảm bảo sao chép mọi cơ sở và thành viên.

**17.5.1.2 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.

**17.5.1.3 The Meaning of Copy**

**17.5.1.4 Slicing**

Một con trỏ đến một lớp dẫn xuất chuyển đổi hoàn toàn thành một con 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 (§3.2.4, §20.2) dẫn đến một cái bẫy cho những người không cẩn thận.

**17.5.2 Move**

Bạn có thể sử dụng di chuyển khi bạn cần "chuyển" nội dung của một đối tượng ở nơi khác mà không cần sao chép (ví dụ: nội dung không bị trùng lặp, đó là lý do tại sao nó có thể được sử dụng trên một số đối tượng không thể sao chép, như unique\_ptr). Cũng có thể một đối tượng lấy nội dung của một đối tượng tạm thời mà không cần sao chép (và tiết kiệm rất nhiều thời gian), với std :: move.

**17.6 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 tôi khi cần thiết. Theo mặc định, một lớp cung cấp:

• Một hàm tạo mặc định: X ()

• Hàm tạo bản sao: X (const X &)

• Phép gán bản sao: X & operator = (const X &)

• Một hàm tạo di chuyển: X (X &&)

• Phép chuyển nhượng: X & operator = (X &&)

• Một hàm hủy: ~X ())

**17.6.1 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 thích xem danh sách đầy đủ các thao tác trong văn bản chương trình ngay cả khi danh sách đầy đủ đó không cần thiết**.**

**17.6.2 Default Operations**

Ý nghĩa mặc định của mỗi hoạt động được tạo, như được triển khai khi trình biên dịch tạo ra nó, là áp dụng hoạt động cho từng thành viên dữ liệu cơ sở và không tĩnh của lớp. Đó là, chúng tôi nhận được bản sao của thành viên, bản dựng mặc định của thành viên, v.v.

**17.6.3 Using Default Operations**

**17.6.3.1 Default Constructors**

Hàm khởi tạo mặc định mặc định (tức là được tạo) mặc định xây dựng từng thành viên. '' Khởi tạo mặc định '' của một thành viên tích hợp khiến thành viên đó chưa được khởi tạo.

**17.6.3.2 Maintaining Invariants**

[1] Thiết lập một bất biến trong một phương thức khởi tạo (bao gồm cả khả năng thu nhận tài nguyên).

[2] Duy trì tính bất biến với các thao tác sao chép và di chuyển (với các tên và kiểu thông thường).

[3] Thực hiện mọi thao tác dọn dẹp cần thiết trong trình hủy (bao gồm cả giải phóng tài nguyên).

**17.6.3.3 Resource Invariants**

Một bất biến là "khái niệm" nhiều hơn một biến. Nói chung, đó là một tài sản của trạng thái chương trình luôn luôn đúng. Một hàm hoặc phương thức đảm bảo rằng bất biến giữ được cho là duy trì bất biến.

## **CHƯƠNG 18**

## **Operato Overloading**

# **Giới thiệu**

• Chức năng của toán tử

Toán tử nhị phân và đơn nguyên; Ý nghĩa được xác định trước cho các nhà khai thác; Các nhà khai thác và các loại do người dùng xác định; Đối tượng Chuyền; Toán tử trong Không gian tên

• Một loại số phức

Các nhà điều hành thành viên và không phải thành viên; Số học Chế độ Hỗn hợp; Chuyển đổi; Chữ viết;

Chức năng của Accessor; Chức năng của người trợ giúp

• Chuyển đổi loại

Nhà điều hành chuyển đổi; Nhà điều hành chuyển đổi rõ ràng; Mơ hồ

**18.1 Giới thiệu**

Mọi lĩnh vực kỹ thuật - và hầu hết các lĩnh vực phi kỹ thuật - đều phát triển các ký hiệu viết tắt thông thường để 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,

x + y ∗ z

rõ ràng hơn đối với chúng tôi

nhân y với z và cộng kết quả với x

Thật khó để đánh giá quá cao tầm quan trọng của ký hiệu súc tích đối với các phép toán thông thường.

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 hết các cept con mà các toán tử được sử dụng thông thường không phải là các kiểu tích hợp sẵn trong C ++, vì vậy chúng phải

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. Việc xác định công suất hoạt động cho các lớp như vậy đôi khi cho phép lập trình viên cung cấp một ký hiệu thông thường và thuận tiện hơn để thao tác các đối tượng hơn là có thể đạt được khi chỉ sử dụng chức năng cơ bản.

**18.2 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 (§6.3.4, §16.2.12)

**.**  lựa chọn thành viên (§8.2)

**. ∗** lựa chọn thành viên thông qua con trỏ đến thành viên (§20.6)

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 xóa op của chúng:

**Sizeof**  kích thước của đối tượng (§6.2.8)

**alignof**  căn chỉnh của đối tượng (§6.2.9)

**typeid** type\_info của một đối tượng (§22.5)

Cuối cùng, toán tử biểu thức điều kiện bậc ba không thể được nạp chồng (không vì lý do tinh thần đặc biệt nào):

**?:** đánh giá có điều kiện (§9.4.1)

Ngoài ra, các ký tự do người dùng định nghĩa (§19.2.6) được xác định bằng cách sử dụng ký hiệu **operator"".** Đây là một loại toán tử con cú pháp vì không có toán tử nào được gọi là "". Tương tự, **operatorT ()** định nghĩa một chuyển đổi sang kiểu **T** (§18.4).

Tên của một hàm **operator** là toán tử từ khóa được theo sau bởi chính toán tử đó, cho

ví dụ, **operator<<.** Một hàm toán tử được khai báo và có thể được gọi như bất kỳ hàm nào khác. Một việc sử dụng toán tử chỉ là cách viết tắt cho một lệnh gọi rõ ràng của hàm toán tử. Ví dụ:

void f(complex a, complex b)

{

complex c = a + b; // rút ngắn

complex d = a.operator+(b); // gọi rõ ràng

}

Với định nghĩa trước đây về **complex**, hai bộ khởi tạo đồng nghĩa

**18.2.1 Toán tử nhị phân và đơn nguyên**

Một toán tử nhị phân có thể được xác định bởi một hàm thành viên **non-static** (không tĩnh) nhận một đối số hoặc một

hàm nonmember nhận hai đối số. Đối với bất kỳ toán tử nhị phân nào **@**, **aa @ bb** có thể được hiểu

dưới dạng **aa.operator @ (bb)** hoặc **operator @ (aa, bb).** Nếu cả hai đều được xác định, giải quyết quá tải (§12.3)

xác định cách diễn giải, nếu có, được sử dụng. Ví dụ

class X {

public:

void operator+(int);

X(int);

};

void operator+(X,X);

void operator+(X,double);

void f(X a) {

a+1; // a.operator+(1)

1+a; // ::operator+(X(1),a)

a+1.0; // ::operator+(a,1.0)

}

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 (§iso.A).

Ví dụ: người dùng không thể xác định**%** một bậc hoặc một bậc ba **+.** Xem xét:

class X {

public: // các thành viên (với con trỏ này ngầm định):

X∗ operator&(); // tiền tố một ngôi & (địa chỉ của)

X operator&(X); // nhị phân &()

X operator++(int); // tăng tiền tố (see §19.2.4)

X operator&(X,X); // error : bậc ba

X operator/(); // error : một ngôi /

};

// các hàm không phải thành viên

X operator−(X); // tiền tố một ngôi trừ

X operator−(X,X); // trừ nhị phân

X operator−−(X&,int); // giảm hậu tố

X operator−(); // error : không có toán hạng

X operator−(X,X,X); // error : bậc ba

X operator%(X); // error : một ngôi %

Toán tử [] được mô tả trong §19.2.1, toán tử **()** trong §19.2.2, toán tử **->** trong §19.2.3, toán tử ++ và **−−** trong §19.2.4, và các toán tử phân bổ và phân bổ trong §11.2.4 và §19.2.5.

Toán tử **operator =** (§18.2.2), **operator []** (§19.2.1), **operator ()** (§19.2.2) và **operator−>** (§19.2.3) phải là các hàm thành viên không tĩnh.

Ý nghĩa mặc định của **&&, ||,** và, (dấu phẩy) liên quan đến trình tự: toán hạng đầu tiên được đánh giá trước thứ hai (và cho **&&** và **||** toán hạng thứ hai không phải lúc nào cũng được đánh giá). Quy tắc đặc biệt này không áp dụng cho các phiên bản do người dùng xác định của **&&, ||** và, (dấu phẩy); thay vào đó những toán tử này được xử lý chính xác như các toán tử nhị phân khác.

**18.2.2 Ý nghĩa được xác định trước cho các toán tử**

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 một định nghĩa của **Z :: operator + = ()** từ các định nghĩa của **Z :: opera tor + ()** và **Z :: operator = ().**

Các toán tử **=** (gán), & (address-of) và, (trình tự; §10.3.2) có giá trị trung bình được xác định trước khi áp dụng cho các đối tượng lớp. Những ý nghĩa được xác định trước này có thể bị loại bỏ (‘‘ xóa ’’; §17.6.4):

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 Các toán tử và các loại do người dùng xác định**

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 **new** và **delete** 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 (§6.2.1) 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**. Ví dụ này được xử lý thông thường bằng cách sử dụng một hoặc nhiều hàm khác (§18.3.2, §19.4).

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(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 §12.3.

**18.2.5 Các toán tử trong Không gian tên**

Một toán tử hoặc là một thành viên của một lớp hoặc được định nghĩa trong một số không gian tên (có thể là không gian tên chung). Hãy xem xét phiên bản đơn giản này của chuỗi I / O từ thư viện tiêu chuẩn:

namespace std { // simplified std

class string {

// ...

};

class ostream {

// ...

ostream& operator<<(const char\*); //

};

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";

}

Đương nhiên, điều này viết ra **Xin chào, thế giới !**. Nhưng tại sao? Lưu ý rằng tôi không làm mọi thứ từ **std** có thể truy cập bằng cách viết:

using namespace std;

Thay vào đó, tôi đã sử dụng tiền tố **std ::** cho chuỗi và **cout.** Nói cách khác, tôi đã thực hiện hành vi tốt nhất của mình và không gây ô nhiễm không gian tên chung hoặc theo cách khác, tạo ra các phụ thuộc không cần thiết.

Toán tử đầu ra cho chuỗi kiểu C là một thành viên của **std :: ostream**, vì vậy theo định nghĩa

std::cout << p

có ngĩa là

std::cout.operator<<(p)

Tuy nhiên, **std :: ostream** không có hàm thành viên để xuất ra một chuỗi **std :: string**, vì vậy

std::cout << s

có nghĩa là

operator<<(std::cout,s)

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 (§14.2.4). 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&)

**18.3 Một loại số phức**

Việc triển khai các số phức được trình bày trong §18.1 là quá hạn chế để làm hài lòng bất kỳ ai. Cho 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 đợi sẽ được cung cấp thêm một số toán tử, chẳng hạn như **==** để so sánh và **<<** cho đầu ra, và một tập hợp các hàm toán học phù hợp, chẳng hạn như **sin ()** và **sqrt ().**

Lớp **complex** là một loại bê tông, vì vậy thiết kế của nó tuân theo các hướng dẫn từ §16.3. Ngoài ra những người sử dụng số học phức tạp phụ thuộc rất nhiều vào các toán tử đến mức định nghĩa về **số phức** đưa vào hầu hết các quy tắc cơ bản để nạp chồng toán tử.

Kiểu **complex** được phát triển trong phần này sử dụng **double** cho các đại lượng vô hướng của nó và gần như tương đương tới thư viện chuẩn **complex <double>** (§40.4).

**18.3.1 Các toán tử thành viên và không phải thành viên**

Tôi muốn giảm thiểu số lượng hàm thao tác trực tiếp với việc biểu diễn một sự vật. Điều này có thể đạt được bằng cách xác định chỉ các toán tử vốn đã sửa đổi giá trị của đối số đầu tiên, chẳng hạn như **+ =**, trong chính lớp đó. Các toán tử chỉ đơn giản là tạo ra một giá trị mới dựa trên các giá trị của các đối số của chúng, chẳng hạn như **+**, sau đó được xác định bên ngoài lớp và sử dụng trong quá trình triển khai của họ:

class complex {

double re, im;

public:

complex& operator+=(complex a); // cần quyền truy cập

// ...

};

complex operator+(complex a, complex b)

{

return a += b; // trả về thông qua +=

}

Các đối số cho **toán tử + ()** này được truyền theo giá trị, vì vậy **a + b** không sửa đổi các toán hạng của nó. Với những khai báo này, chúng ta có thể viết:

void f(complex x, complex y, complex z)

{

complex r1 {x+y+z}; // r1 = operator+(operator+(x,y),z)

complex r2 {x}; // r2 = x

r2 += y; // r2.operator+=(y)

r2 += z; // r2.operator+=(z)

}

Ngoại trừ sự khác biệt về hiệu quả có thể xảy ra, các tính toán của **r1** và **r2** là tương đương.

Các toán tử gán kết hợp như **+ =** và **∗ =** có xu hướng dễ xác định hơn so với các toán tử '' sim ple '' + và ∗ của chúng. Điều này khiến hầu hết mọi người ngạc nhiên lúc đầu, nhưng nó xuất phát từ thực tế là ba 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. Ví dụ:

inline complex& complex::operator+=(complex a)

{

re += a.re;

im += a.im;

return ∗this;

}

Điều này không yêu cầu một biến tạm thời để giữ kết quả của phép cộng và đơn giản đối với trình biên dịch sang nội tuyến một cách hoàn hảo.

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 toán tử có quyền truy cập trực tiếp vào biểu diễn của các lớp.

**18.3.3 Chuyển đổi**

Để đối phó với các bài tập và khởi tạo các biến complex 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 complex.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. Đối với kỳ thi xin:

class complex {

double re, im;

public:

complex(double r) :re{r}, im{0} { } // xây dựng một số phức từ a double

// ...

};

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 là complex b {3,0};

Chuyển đổi do người dùng xác định chỉ được áp dụng hoàn toàn nếu nó là duy nhất (§12.3). Nếu bạn không muốn một cấu trúc con được sử dụng ngầm, hãy khai báo nó một cách rõ ràng (§16.2.6).

Đương nhiên, chúng ta vẫn cần hàm tạo có giá trị gấp đôi và một hàm tạo mặc định khởi tạo một **complex** thành **{0,0},** cũng rất hữu ích:

class complex {

double re, im;

public:

complex() : re{0}, im{0} { }

complex(double r) : re{r}, im{0} { }

complex(double r, double i) : re{r}, im{i} { }

// ...

};

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} { }

// ...

};

**18.3.3.1 Chuyển đổi toán hạng**

Chúng tôi đã xác định ba phiên bản của mỗi trong số bốn toán tử số học tiêu chuẩn:

complex operator+(complex,complex);

complex operator+(complex,double);

complex operator+(double ,complex);

// ...

Điều gì sẽ xảy ra nếu chúng ta có ba lần thay thế cho loại của mỗi đối số cho mỗi hàm? Chúng ta sẽ cần ba phiên bản của mỗi hàm sin gle-đối số, chín phiên bản của mỗi hàm hai đối số, 27 phiên bản của mỗi hàm ba 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 số là dựa vào chuyển đổi. Ví dụ: lớp **complex** của chúng tôi cung cấp một phương thức khởi tạo chuyển đổi một **double** thành **complex**. Do đó, chúng tôi chỉ có thể khai báo một phiên bản của trình tổ chức bình đẳng cho **complex**:

bool operator==(complex,complex);

void f(complex x, complex y)

{

x==y; // có nghĩa là operator==(x,y)

x==3; // có nghĩa là operator==(x,complex(3))

3==y; // có nghĩa là operator==(complex(3),y)

}

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 (xem §10.3.4).

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 is not a class object

3+=z; // error : 3 is not a class object

}

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. Tuy nhiên, đó chỉ là ước tính vì có thể truy cập tạm thời với một thao tác sửa đổi, chẳng hạn như **operator + = ():**

complex x {4,5}

complex z {sqr t(x)+={1,2}}; // như tmp=sqrt(x), tmp+={1,2}

Nếu chúng tôi không muốn chuyển đổi ngầm định, chúng tôi có thể sử dụng chuyển đổi rõ ràng để ngăn chặn chúng (§16.2.6, §18.4.2).

**18.3.4 Chữ viết**

Chúng tôi có chữ viết 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 (§10.4). Ví dụ:

class complex

{

public:

constexpr complex(double r =0, double i =0) : re{r}, im{i} { }

// ...

}

Do đó, một **complex** 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 hoàn toàn có lý do có thể nghĩ đến các lệnh gọi hàm tạo với các đối số theo nghĩa đen là các chữ.

**18.3.5 Chức năng người truy cập**

Cho đến nay, chúng tôi đã cung cấp **lớp complex** 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 **Date** từ §16.3 hoặc (thậm chí nhiều hơn vì vậy) đối với **Chuỗi** từ §19.3. Tuy nhiên, đối với **complex, 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

Ví dụ: cho trước **real ()** và **image ()**, chúng ta có thể đơn giản hóa các phép toán đơn giản, phổ biến và hữu ích, chẳng hạn như  **==,** dưới dạng hàm không phải là thành viên (mà không ảnh hưởng đến hiệu suất):

inline bool operator==(complex a, complex b)

{

return a.real()==b.real() && a.imag()==b.imag();

}

**18.3.6 Chức năng của người trợ giúp**

Nếu chúng ta đặt tất cả các bit và các mảnh lại với nhau, lớp **complex** 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);

// binary -, \*, and /

complex operator−(complex); // trừ một ngôi

complex operator+(complex); // cộng một ngôi

bool operator==(complex,complex);

bool operator!=(complex,complex);

istream& operator>>(istream&,complex&); // input

ostream& operator<<(ostream&,complex); // output

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.

Chúng tôi có thể cung cấp các chức năng để cho phép người dùng suy nghĩ về tọa độ cực:

complex polar(double rho, double theta);

complex conj(complex);

double abs(complex);

double arg(complex);

double norm(complex);

double real(complex); // để thuận tiện cho việc ghi chú

double imag(complex); // để thuận tiện cho việc ghi chú

Cuối cùng, chúng ta phải cung cấp một tập hợp các hàm toán học tiêu chuẩn thích hợp:

complex acos(complex);

complex asin(complex);

complex atan(complex);

// ...

Theo quan điểm của người dùng, kiểu **complex** được trình bày ở đây gần giống với kiểu **complex<double>** được tìm thấy trong **<complex>** trong thư viện chuẩn (§5.6.2, §40.4).

**18.4 Chuyển đổi kiểu**

Việc chuyển đổi kiểu có thể được thực hiện bằng

• Một hàm tạo nhận một đối số duy nhất (§16.2.5)

• Toán tử chuyển đổi (§18.4.1)

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 (§18.4.3), ví dụ: như một đối số hàm.

**18.4.1 Các toán tử chuyển đổi**

Sử dụng một hàm tạo lấy một đối số duy nhất để chỉ định chuyển đổi kiểu là thuận tiện nhưng có các cation hàm ý 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] một chuyển đổi từ một lớp mới sang một lớp đã xác định trước đó (mà không sửa đổi khẩu phần decla cho lớp cũ).

Những vấn đề này có thể được xử lý bằng cách xác định toán tử chuyển đổi cho kiểu nguồn. Một thành viên

hàm **X :: operator T (),** trong đó **T** là tên kiểu, xác định một chuyển đổi từ **X** sang **T**. Ví dụ, chúng ta có thể xác định một số nguyên không âm 6 bit, Tiny, có thể trộn tự do với các số nguyên trong các toán hạng số học. **Tiny** ném **Bad\_range** nếu các hoạt động của nó bị tràn hoặc bị tràn:

class Tiny {

char v;

void assign(int i) { if (i&˜077) throw Bad\_rang e(); v=i; }

public:

class Bad\_range { };

Tiny(int i) { assign(i); }

Tiny& operator=(int i) { assign(i); return ∗this; }

operator int() const { return v; } // chuyển đổi sang hàm int

};

Phạm vi được kiểm tra bất cứ khi nào một **Tiny** được khởi tạo bởi một **int** và bất cứ khi nào một **int** được gán cho một.

Không cần kiểm tra phạm vi khi chúng tôi sao chép **Tiny**, vì vậy hàm tạo và gán bản sao mặc định là đúng rồi.

Để kích hoạt các hoạt động số nguyên thông thường trên các biến **Tiny,** chúng tôi xác định chuyển đổi ngầm định từ **Tiny** thành **int, Tiny :: operator int ().** Lưu ý rằng kiểu đang được chuyển đổi thành là một phần của tên của công cụ hoạt động và không thể được lặp lại dưới dạng giá trị trả về của hàm chuyển đổi:

Tiny::operator int() const { return v; } // right

int Tiny::operator int() const { return v; } // error

Về mặt này, toán tử chuyển đổi cũng giống như một phương thức khởi tạo.

Bất cứ khi nào một Tiny xuất hiện ở nơi cần sử dụng int, int thích hợp sẽ được sử dụng. Ví dụ:

int main()

{

Tiny c1 = 2;

Tiny c2 = 62;

Tiny c3 = c2−c1; // c3 = 60

Tiny c4 = c3; // không kiểm tra phạm vi (không cần thiết)

int i = c1+c2; // i = 64

c1 = c1+c2; // phạm vi error: c1 không thể là 64

i = c3−64; // i = -4

c2 = c3−64; // phạm vi error: c2 không thể là -4

c3 = c4; // không kiểm tra phạm vi (không cần thiết)

}

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 rang 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 biểu thị trạng thái **cin.** Giá trị này sau đó có thể được kiểm tra bằng **while** (xem §38.4.4). Tuy nhiên, thông thường không phải là một ý kiến 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.

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 nhất định, nhưng không phải cả hai.

**18.4.2 Các toán tử chuyển đổi rõ ràng**

Các toán tử chuyển đổi có xu hướng được xác định để chúng có thể được sử dụng ở mọi nơi. Tuy nhiên, có thể khai báo toán tử chuyển đổi một cách rõ ràng và chỉ áp dụng cho việc khởi tạo trực tiếp (§16.2.6), trong đó một phương thức khởi tạo rõ ràng tương đương sẽ được sử dụng. Ví dụ, thư viện tiêu chuẩn unique\_ptr (§5.2.1, §34.3.1) 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; // \* điều này có giữ một con trỏ (đó 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.

Xem xét:

void use(unique\_ptr p, unique\_ptr q)

{

if (!p) // OK: we want this use

throw Invalid\_uninque\_ptr{};

bool b = p; // error ; suspicious use

int x = p+q; // error ; we definitly don’t want this

}

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 việc **q** có hợp lệ hay không).

**18.5 Lời khuyên**

*[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 cơ quan đại diện; §18.3.1.*

*[6] Ưu tiên các chức năng nonmember hơn các thành viên cho các hoạt động không cần quyền truy cập vào bản gửi lại đại diện; §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 nonmember cho các toán tử đối xứng; §18.3.2.*

*[9] Sử dụng các hàm thành viên để thể hiện 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 ‘‘ 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.*

## **CHƯƠNG 19 : SEPICAL OPERATORS**

Giới thiệu

• Toán tử đặc biệt

Ghi danh; Gọi chức năng; Hội thảo; Tăng và Giảm; Phân bổ và phân bổ De; Chữ viết do người dùng xác định

• Một lớp chuỗi

Hoạt động cần thiết; Quyền truy cập vào các ký tự; Đại diện; Chức năng Thành viên; Người giúp đỡ

Chức năng; Sử dụng chuỗi của chúng tôi

• Bạn

Bạn bè và thành viên

**19.1 Giới thiệu**

Overloading không chỉ dành cho các phép toán số học và logic. Trên thực tế, các nhà khai thác đóng vai trò quan trọng trong việc thiết kế vùng chứa (ví dụ: vectơ và bản đồ; §4.4), '' con trỏ thông minh '' (ví dụ: unique\_ptr và shared\_ptr; §5.2.1), trình vòng lặp (§4.5) và các lớp khác liên quan đến quản lý tài nguyên.

**19.2 Các toán tử đặc biệt**

Các toán tử

[ ] () -> ++ −− new delete

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 +, <, và ˜ (§18.2.3). Các toán tử [ ] (subscript) và () (call) là một trong những toán tử hữu ích nhất do người dùng định nghĩa.

**19.2.1 Đăng ký**

Một hàm **toán tử []** có thể được sử dụng để cung cấp ý nghĩa cho các chỉ số con cho các đối tượng lớp. Chức năng lập luận thứ hai (chỉ số con) của một **hàm toán tử []** có thể thuộc bất kỳ loại 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 Assoc {

vector <pair <string, int >> vec; // vectơ của các cặp {name, value}

const int & operator [] (const string &) const;

int & operator [] (const string &);

};

Chúng ta có thể sử dụng **Assoc** như thế này:

int main () // đếm số lần xuất hiện của mỗi từ trên đầu vào

{

Assoc values ;

string buf;

while (cin >> buf) ++ values [buf];

for (auto x: values.vec)

cout << '{' << x.first << ',' << x.second << "} \ n";

}

Bản đồ thư viện tiêu chuẩn và bản đồ không có thứ tự là những phát triển tiếp theo của ý tưởng về một mảng (§4.4.3, §31.4.3) với các triển khai ít ngây thơ hơn.

**Toán tử [] ()** phải là một hàm thành viên không tĩnh.

**19.2.2 Gọi hàm**

Lời gọi hàm, nghĩa là, biểu thức ký hiệu (biểu thức-danh sách), có thể được hiểu như một toán hạng nhị phân 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. Ví dụ:

struct Action {

int operator () (int);

pair <int, int> operator () (int, int);

double operator () (double);

// ...

};

void f (Action act)

{

int x = act (2);

auto y = act (3,4);

double z = act (2.3);

// ...

};

Danh sách đối số cho toán tử () () được đánh giá và kiểm tra theo các quy tắc truyền đối số thông thường. Rõ ràng nhất và cũng là quan trọng nhất, việc sử dụng toán tử () là cung cấp cú pháp gọi hàm cho các đối tượng theo một cách nào đó hoạt động giống như các hàm. Một đối tượng hoạt động giống như một hàm thường được gọi là một đối tượng giống hàm hoặc đơn giản là một đối tượng hàm (§3.4.3). Chức năng như vậy các đối tượng cho phép chúng ta viết mã lấy các phép toán tầm thường làm tham số. Trong nhiều trường hợp, nó là thiết yếu là các đối tượng chức năng có thể chứa dữ liệu cần thiết để thực hiện hoạt động của chúng. Ví dụ, chúng tôi có thể định nghĩa một lớp bằng **toán tử () ()** để thêm giá trị được lưu trữ vào đối số của nó:

class Add {

complex val;

public:

Add (complex c): val {c} {} // lưu một giá trị

Add (double r, double i): val {{r, i}} {}

void operator () (complex & c) const {c + = val; } // thêm giá trị vào đối số

};

**19.2.4 Tăng và Giảm**

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 hữu ích 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 ngoài phạm vi , ngoại lệ

++ 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. Do đó, chúng ta phải xác định gia số tiền tố và hậu tố và giảm cho **Ptr <T>**. Ví dụ:

template <typename T>

class Ptr {

T ∗ ptr;

T ∗ mảng;

int sz;

public:

template <int N>

Ptr (T ∗ p, T (& a) [N]); // liên kết với mảng a, sz == N, giá trị ban đầu p

Ptr (T ∗ p, T ∗ a, int s); // liên kết với mảng a có kích thước s, giá trị ban đầu p

Ptr (T ∗ p); // liên kết với một đối tượng, sz == 0, giá trị ban đầu p

Ptr & operator ++ (); // tiền tố

Ptr operator ++ (int); // hậu tố

Ptr & operator −− (); // tiền tố

Ptr operator −− (int); // hậu tố

T & operator \*(); // tiền tố

};

Đối số **int** được sử dụng để chỉ ra rằng hàm sẽ được gọi cho ứng dụng hậu tố của **++.**

**Int** này không bao giờ được sử dụng; đối số chỉ đơn giản là một giả được sử dụng để phân biệt giữa tiền tố và ứng dụng sửa lỗi sau. Cách để nhớ phiên bản nào của **toán tử ++** là tiền tố là lưu ý rằng phiên bản không có đối số giả là tiền tố, chính xác như tất cả các toán tử số học và logi cal một bậc khác. Đối số giả chỉ được sử dụng cho hậu tố '' lẻ '' **++** và **−−.**

**19.2.5 Allocation and Deallocation**

**Toán tử new** (§11.2.3) lấy bộ nhớ của nó bằng cách gọi **một toán tử new ().** Tương tự, **toán tử delete** 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** (§6.2.8) cho các kích thước, khai báo của sions ver toàn cục trông giống như sau:

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

// để biết thêm phiên bản, hãy xem §11.2.4

Nghĩa là, khi **new** cần bộ nhớ trên kho lưu trữ miễn phí cho một đối tượng kiểu X, nó sẽ gọi **operator new (sizeof (X)).** Tương tự, khi mới cần bộ nhớ trên kho lưu trữ miễn phí cho một mảng **N** đối tượng của kiểu **X**, nó gọi **operator new [] (N ∗ sizeof (X)).** Một biểu thức mới có thể yêu cầu nhiều bộ nhớ hơn là được chỉ ra bởi **N ∗ sizeof (X),** nhưng nó sẽ luôn làm như vậy về một số ký tự (tức là một số của byte). Thay thế **toán tử toàn cục new ()** và **toán tử delete ()** không được khuyến khích. Rốt cuộc, người khác có thể dựa vào một số khía cạnh của hành vi mặc định hoặc thậm chí có thể đã cung cấp các phiên bản khác của các chức năng này.

**19.3 Một lớp chuỗi**

Lớp chuỗi tương đối đơn giản được trình bày trong phần này minh họa một số kỹ thuật được sử dụng để thiết kế và triển khai các lớp bằng cách sử dụng các toán tử được định nghĩa theo quy ước. Cái này

**String** là phiên bản đơn giản hóa của chuỗi thư viện chuẩn (§4.2, Chương 36). 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 chuỗi ký tự, **std :: string**

Để cho phép khả năng tương tác đơn giản với các chuỗi kiểu C (bao gồm các ký tự chuỗi (§7.3.2)), tôi thực hiện lại các chuỗi đã gửi dưới dạng mảng ký tự có kết thúc bằng không. Đối với chủ nghĩa hiện thực, tôi thực hiện tối ưu hóa chuỗi ngắn. Nghĩa là, một chuỗi chỉ có một vài ký tự sẽ lưu trữ các ký tự đó trong đối tượng lớp chính nó, thay vì trên cửa hàng miễn phí. Điều này tối ưu hóa việc sử dụng chuỗi cho các chuỗi nhỏ. Kinh nghiệm

cho thấy rằng đối với một số lượng lớn các ứng dụng, hầu hết các chuỗi đều ngắn. Việc tối ưu hóa này đặc biệt quan trọng trong các hệ thống đa luồng, nơi mà việc chia sẻ thông qua các con trỏ (hoặc tham chiếu) là không cần thiết và việc phân bổ và phân bổ kho lưu trữ tự do tương đối tốn kém.

Để cho phép các **String** '' phát triển '' một cách hiệu quả bằng cách thêm các ký tự vào cuối, tôi triển khai một lược đồ

để giữ thêm không gian cho sự phát triển như vậy tương tự như không gian được sử dụng cho vectơ (§13.6.1). Điều này làm cho Chuỗi mục tiêu phù hợp cho nhiều dạng đầu vào khác nhau.

Viết một lớp chuỗi tốt hơn và / hoặc một lớp cung cấp nhiều tiện ích hơn là một bài tập tốt. Điều đó xong, chúng ta có thể vứt bỏ bài tập của mình và sử dụng **std :: string** (Chương 36)

**19.3.1 Các toán tử cần thiết**

Chuỗi lớp cung cấp tập hợp thông thường của các hàm tạo, một hàm hủy và các hoạt động gán (§17.1):

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 &); // hàm tạo sao chép

string & operator = (const string &); // hàm gán

string (string && x); // di chuyển hàm tạo

string & operator = (string && x); // di chuyển hàm gán

˜String () {if (short\_max <sz) delete [] ptr; } // hàm hủy

// ...

};

Chuỗi này có ngữ nghĩa giá trị. Tức là sau một phép gán **s1 = s2**, hai chuỗi **s1** và **s2** là

hoàn toàn khác biệt và những thay đổi tiếp theo đối với cái này không ảnh hưởng đến cái khác. Sự thay thế sẽ là để cung cấp ngữ nghĩa con trỏ chuỗi. Điều đó có nghĩa là để các thay đổi đối với **s2** sau **s1 = s2** cũng ảnh hưởng đến giá trị của **s1**. Nếu nó có ý nghĩa, tôi thích ngữ nghĩa giá trị hơn; các ví dụ là phức tạp, vectơ, Ma trận và string. Tuy nhiên, để ngữ nghĩa giá trị phù hợp với túi tiền, chúng ta cần chuyển các Chuỗi bằng cách tham chiếu khi chúng tôi không cần bản sao và triển khai ngữ nghĩa chuyển động (§3.3.2, §17.5.2) để tối ưu hóa lợi nhuận.

Biểu diễn chuỗi nhỏ không đáng kể được trình bày trong §19.3.3. Lưu ý rằng nó yêu cầu

phiên bản do người dùng xác định của hoạt động sao chép và di chuyển.

**19.3.2 Quyền truy cập vào các ký tự**

Việc thiết kế các toán tử truy cập cho một chuỗi là một chủ đề khó vì truy cập lý tưởng là bằng ký hiệu bậc đồng nhất (nghĩa là sử dụng **[ ]**), hiệu quả tối đa và được kiểm tra phạm vi. Thật không may, bạn không thể có tất cả các thuộc tính này cùng một lúc. 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 khả dụng

{return (sz <= short\_max)? short\_max: sz + space; }

// ...

};

**19.3.3.1 Các hàm phụ trợ**

Ngoài các hàm dành cho mục đích sử dụng chung, tôi nhận thấy rằng mã của tôi trở nên sạch hơn khi tôi cung cấp ba hàm phụ trợ làm '' khối xây dựng '' để giúp tôi kiểm tra lại cách sửa chữa hơi phức tạp và giảm thiểu việc 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 thú vị, triển khai không chỉ là biểu diễn cộng với các hàm công khai. 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.

**19.3.4 Hàm thành viên**

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 (§19.3.3)

{

ch [0] = 0; // kết thúc 0

}

Với **copy\_from ()** và **move\_from ()**, các hàm tạo, di chuyển và phép gán khá đơn giản để

thực hiện. Hàm tạo nhận đối số chuỗi kiểu C phải xác định số và lưu trữ chúng một cách thích hợp:

String :: String (const char ∗ p)

: sz {strlen (p)},

ptr {(sz <= short\_max)? ch: new char [sz + 1]},

space{0}

{

strcpy (ptr, p); // sao chép các ký tự vào ptr từ p

}

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:

String :: String (const String & x) // hàm tạo sao chép

{

copy\_from (x); // sao chép biểu diễn từ x

}

Tôi không bận tâm đến việc cố gắng tối ưu hóa trường hợp kích thước của nguồn bằng với kích thước của mục tiêu

(như đã làm đối với **vectơ**; §13.6.3). Tôi không biết liệu điều đó có đáng giá hay không.

Tương tự, hàm tạo di chuyển di chuyển biểu diễn từ nguồn của nó (và có thể đặt nó

đối số là chuỗi trống):

String :: String (String && x) // di chuyển hàm tạo

{

move\_from (x);

}

Giống như hàm tạo bản sao, phép gán bản sao sử dụng copy\_from () để sao chép đại diện đối số của nó. 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):

String & String :: operator = (const String & x)

{

if (this == & x) return ∗ this;

char ∗ p = (rút gọn t\_max <sz)? ptr: 0;

copy\_from (x);

delete[] p;

return ∗ this;

}

**19.3.5 Hàm người trợ giúp**

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**. Trong thông tin cụ thể, **<<** chỉ in các ký tự mà không cần thêm định dạng và **>>** bỏ qua khoảng trắng ban đầu trước khi đọc cho đến khi 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;

}

Tôi cung cấp **==** và **!=** Để so sánh

bool operator==(const String& a, const String& b)

{

if (a.size()!=b.siz e())

return false;

for (int i = 0; i!=a.size(); ++i)

if (a[i]!=b[i])

return false;

return true;

}

bool operator!=(const String& a, const String& b) {

return !(a==b);

}

Với hàm thành viên **+=** thêm một ký tự vào cuối, các toán tử nối dễ dàng được cung cấp dưới dạng các hàm khác:

String& operator +=(String& a, const String & b) /

{

for (auto x: b)

a += x;

return a;

}

String operator+(const String& a, const String& b)

{

String res {a};

res += b;

return res;

}

**19.4 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).

Bằng cách khai báo một hàm thành viên **stati**c (§16.2.12), chúng ta chỉ có thể cung cấp cho nó hai thuộc tính đầu tiên. Bằng cách khai báo một hàm nonmember là bạn, chúng ta chỉ có thể cung cấp cho nó thuộc tính đầu tiên. Đó là, một chức năng

Bạn đã khai báo được cấp quyền truy cập vào việc triển khai một lớp giống như một hàm thành viên nhưng không phụ thuộc vào lớp đó.

Ví dụ, chúng ta có thể xác định một toán tử nhân Ma trận với một Vectơ. Đương nhiên, Véc tơ và **Matrix** ẩn các biểu diễn tương ứng của chúng và cung cấp một tập hợp các thao tác hoàn chỉnh để thao tác với các đối tượng cùng loại. Tuy nhiên, thói quen nhân của chúng ta không thể là thành viên của cả hai.

Ngoài ra, chúng tôi không thực sự muốn cung cấp các chức năng truy cập cấp thấp để cho phép mọi người dùng đều có thể đọc

và viết biểu diễn đầy đủ của cả **Ma trận** và **Vectơ**. Để tránh điều này, chúng tôi tuyên bố người hoạt động hoặc là bạn của cả hai:

constexpr rc\_max {4}; // kích thước hàng và cột

class Matrix;

class Vector {

float v[rc\_max];

// ...

friend Vector operator∗(const Matrix&, const Vector&);

};

class Matrix {

Vector v[rc\_max];

// ...

friend Vector operator∗(const Matrix&, const Vector&);

};

Bây giờ toán tử **∗()** có thể tiếp cận việc triển khai cả **Vectơ** và **Ma trận**. Điều đó sẽ cho phép

kỹ thuật thực hiện phức tạp, nhưng thực hiện đơn giản sẽ là:

Vector operator∗(const Matrix& m, const Vector& v)

{

Vector r;

for (int i = 0; i!=rc\_max; i++) { // r[i] = m[i] \* v;

r.v[i] = 0;

for (int j = 0; j!=rc\_max; j++)

r.v[i] += m.v[i].v[j] ∗ v.v[j];

}

return r;

}

Một khai báo **bạn** có thể được đặt trong phần riêng tư hoặc phần công khai của một khai báo lớp; nó không quan trọng ở đâu. Giống như một hàm thành viên, một hàm bạn được khai báo rõ ràng trong phân tách của lớp mà nó là bạn bè. Do đó, nó là một phần của giao diện đó giống như một hàm thành viên.

**19.4.2 Bạn bè và thành viên**

Khi nào chúng ta nên sử dụng **hàm bạn** và khi nào thì **một hàm thành viên** là lựa chọn tốt hơn để xác định một thao tác? Đầu tiên, chúng tôi cố gắng giảm thiểu số lượng các hàm truy cập đại diện của một lớp và cố gắng làm cho tập hợp các hàm 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 sang để tin tưởng lúc đầu. Một số hoạt động phải là thành viên - ví dụ: hàm tạo, hàm hủy và các hàm ảo (§3.2.3, §17.2.5) - 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);

};

Các hàm thành viên chỉ có thể được gọi cho các đối tượng thuộc lớp của chúng; không có chuyển đổi do người dùng xác định là áp dụng cho toán hạng ngoài cùng bên trái của a. hoặc **->** (nhưng xem §19.2.3). Ví dụ:

void g ()

{

99.m1 (); // error: X (99) .m1 () không được thử

99.m2 (); // error: X (99) .m2 () không được thử

}

Hàm tổng thể **f1 ()** có một thuộc tính tương tự vì các chuyển đổi ngầm định không được sử dụng cho các đối số tham chiếu không phải const (§7.7). Tuy nhiên, các chuyển đổi có thể được áp dụng cho các đối số của **f2 ()** và **f3 ():**

void h ()

{

f1 (99); // lỗi: f1 (X (99)) không được thử: non-const X & đối số

f2 (99); // Được: f2 (X (99)); const X & đối số

f3 (99); // Được: f3 (X (99)); Đối số X

}

Do đó, một hoạt động sửa đổi trạng thái của một đối tượng lớp phải là một thành viên hoặc một hàm sử dụng đối số tham chiếu không phải **const** (hoặc đối số con trỏ không **const**).

Các toán tử sửa đổi một toán hạng (ví dụ: **=, ∗ =** và **++**) được xác định một cách tự nhiên nhất là các thành viên cho loại do người dùng xác định. Ngược lại, nếu muốn chuyển đổi kiểu ngầm định cho tất cả các toán hạng của một toán hạng, thì hàm thực thi nó phải là một hàm không phải là một hàm khác, lấy tham chiếu const đối số hoặc một đối số không tham chiếu. Điều này thường xảy ra đối với các toán tử triển khai chức năng không yêu cầu toán hạng giá trị khi áp dụng cho các kiểu cơ bản (ví dụ: +, - và ||). Tuy vậy,

các toán tử như vậy thường cần quyền truy cập vào các biểu diễn của lớp toán hạng của chúng. Do đó, nhị phân toán tử là nguồn phổ biến nhất của các hàm bạn bè.

Trừ khi các chuyển đổi loại được xác định, dường như không có lý do thuyết phục nào để chọn thành viên hơn một người bạn tham gia đối số hoặc ngược lại. Trong một số trường hợp, lập trình viên có thể có ưu tiên cho một cú pháp cuộc gọi hơn một cú pháp khác.

Tất cả những thứ khác được coi là bình đẳng, hãy triển khai các hoạt động cần truy cập trực tiếp vào một đại diện dưới dạng các chức năng thành viên:

• Không thể biết một ngày nào đó ai đó sẽ định nghĩa toán tử chuyển đổi.

• Cú pháp gọi hàm thành viên làm cho người dùng rõ ràng rằng đối tượng có thể được sửa đổi;

một đối số tham chiếu ít rõ ràng hơn nhiều.

• Các biểu thức trong phần thân của một thành viên có thể ngắn hơn đáng kể so với các biểu tượng expres tương đương trong một hàm toàn cục; một hàm nonmember phải sử dụng một đối số rõ ràng, trong khi thành viên có thể sử dụng điều này một cách ngầm định.

• Tên thành viên là cục bộ của một lớp, vì vậy chúng có xu hướng ngắn hơn tên của những người không phải là hàm thành viên.

• Nếu chúng ta đã xác định một thành viên f () và sau đó chúng ta cảm thấy cần phải có một f (x) khác, chúng ta có thể chỉ cần xác định nó có nghĩa là x.f ().

Ngược lại, các hoạt động không cần truy cập trực tiếp vào một biểu diễn thường được biểu diễn tốt nhất dưới dạng các hàm không phải là bộ nhớ, có thể trong một không gian tên tạo nên mối quan hệ của chúng với lớp rõ ràng (§18.3.6).

**19.5 Lời khuyên**

*[1] Sử dụng toán tử [] () để lập chỉ số và 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 val; §19.2.2.*

*[3] Sử dụng toán tử -> () để hủy bỏ tham chiếu ‘‘ 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] Định nghĩa toán tử thành viên new () và toán tử thành viên delete () để kiểm soát việc cấp phát và hủy cấp phát 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 chức năng kết bạn nếu bạn cần một chức năng khác 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*

## CHƯƠNG 20: CÁC LỚP CÓ NGUỒN GỐC.

20.1 GIỚI THIỆU

• Các lớp có nguồn gốc

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

• Phân cấp lớp

Nhập trường; Chức năng ảo; Trình độ chuyên môn rõ ràng; Kiểm soát Ghi đè; sử dụng 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

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

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);*

*}*

20.2.1 Chức năng thành viên

Ví dụ:

nhân viên lớp {

công cộng:

void print () const;

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

// ...

riêng:

string first\_name, family\_name;

char middle\_initial;

// ...

};

quản lý lớp: nhân viên công cộng {

công cộng:

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

Trình quản lý :: 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

// ...

}

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.

V.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*

V.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 fol-

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.

20.3.2 CHỨC NĂNG ẢO

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 {*

*công cộng:*

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

*void print () const;*

*// ...*

*riêng:*

*danh sách <Nhân viên ∗> nhóm;*

*mức độ ngắn;*

*// ...*

*};*

*void Manager :: print () const*

*{*

*Nhân viên :: print ();*

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

*// ...*

*}*

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**

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 SOAT GHI DE

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 {*

*công cộng:*

*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 {*

*công cộng:*

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

*// ...*

*};*

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

*thí dụ:*

*class For\_statement final: public Node {*

*công cộng:*

*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)*

*}*

20.3.5.1 TRINH TẠO KẾ THỪA

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

*mẫu <lớp T>*

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

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

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

*void check (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:

*mẫu <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

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:

*lớp Expr {*

*công cộng:*

*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 {*

*công cộng:*

*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

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 {*

*công cộng:*

*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:

*lớp Shape {// lớp trừu tượng*

*công cộng:*

*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 {*

*công cộng:*

*ghi đè void xoay (int) {}*

*void draw () const override;*

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

*Circle (Điểm p, int r);*

*riêng:*

*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*

*công cộng:*

*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;*

*công cộng:*

*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 {*

*công cộng:*

*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*

*};*

20.5 KIỂM SOÁT TRUY CẬP

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:

*công cộng:*

*được bảo vệ:*

*riêng:*

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:*

*mẫu <lớp T>*

*danh sách lớp học {*

*công cộng:*

*void insert (T);*

*T get ();*

*// ...*

*riêng:*

*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:*

*mẫu <lớp T>*

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

*{*

*Liên kết ∗ lnk = get\_free ();*

*lnk−> val = val;*

*lnk−> tiếp theo = đầu;*

*đầu = lnk;*

*}*

*mẫu <lớp T>*

*Danh sách T <T> :: 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:*

*mẫu <lớp T>*

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

*{*

*if (miễn phí == 0) {*

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

*}*

*Liên kết ∗ p = miễn phí;*

*miễn phí = free−> tiếp theo;*

*trả lại p;*

*}*

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

*} mẫu <lớp T>*

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

*{*

*// ...*

*}*

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

*mẫu <typename T>*

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

*{*

*Danh sách <T> :: Liên kết ∗ q = 0; // 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:

*mẫu <lớp T>*

*struct Link2 {*

*T val;*

*Link2 ∗ tiếp theo;*

*};*

*mẫu <lớp T>*

*danh sách lớp học {*

*riêng:*

*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;*

*mẫu <lớp T>*

*liên kết lớp3 {*

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

*T val;*

*Liên kết3 ∗ tiếp theo;*

*};*

*mẫu <lớp T>*

*danh sách lớp học {*

*riêng:*

*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:

*lớp S {*

*công cộng:*

*int m1;*

*công cộng:*

*int m2;*

*};*

20.5.1 THÀNH VIÊN ĐƯỢC BẢO VỆ

Khi thiết kế hệ thống phân cấp lớp, đôi khi chúng tôi cung cấp các hàm được thiết kế để 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 {*

*công cộng:*

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

*// ...*

*được bảo vệ:*

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

*// ...*

*};*

*class Circular\_buffer: public Buffer {*

*công cộng:*

*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

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

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:

*lớp X: public B {/ \* ... \* /};*

*lớp Y: protected B {/ \* ... \* /};*

*lớp 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:

*lớp 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** 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:

*cấu trúc B {*

*int m;*

*tĩnh int sm;*

*// ...*

*};*

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

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

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

*D12 ∗ pd = mới 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. Vì

thí dụ:

*lớp X1: public B {/ \* ... \* /};*

*lớp X2: public B {/ \* ... \* /};*

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

*XX ∗ pxx = mới 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:*

*lớp B {*

*riêng:*

*int a;*

*được bảo vệ:*

*int b;*

*công cộng:*

*int c;*

*};*

*lớp D: công cộng B {*

*công cộng:*

*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

**20.6 Con trỏ tới Membia**

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**

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:

*lớp Std\_interface {*

*công cộng:*

*void ảo start () = 0;*

*void ảo 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**

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**

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 {*

*công cộng:*

*void start ();*

*void đình chỉ ();*

*// ...*

*void in ảo ();*

*riêng:*

*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

Te xt :: 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**

[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.

## **CHƯƠNG 21: Lớp Phân Cấp**

**21.1 Giới thiệu**

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**

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**

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:

*lớp Ival\_box {*

*được bảo vệ:*

*int val;*

*int thấp, cao;*

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

*công cộng:*

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

*virtual int get\_value () {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.

**21.2.1.1 Phê bình**

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**

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:

*lớp Ival\_box {*

*công cộng:*

*ảo int get\_value () = 0;*

*void set\_value ảo (int i) = 0;*

*void reset\_value ảo (int i) = 0;*

*dấu nhắc void ảo () = 0;*

*bool ảo was\_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:

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

*công cộng:*

*Ival\_slider (int, int);*

*Ghi đè ̃Ival\_slider ();*

*int get\_value () ghi đè;*

*ghi đè void set\_value (int i);*

*// ...*

*được bảo vệ:*

*// ... các hàm ghi đè các hàm ảo BBwidget*

*// ví dụ: BBwidget :: draw (), BBwidget :: mouse1hit () ...*

*riêng:*

*// ... 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)*

*{*

*// ...*

*xóa 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:

*lớp Ival\_box {/ \* ... \* /};*

*lớp Ival\_slider*

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

*lớp Ival\_dial*

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

*lớp Flashing\_ival\_slider*

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

*lớp 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*

**21.2.3 Triển khai Thay thế**

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

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

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

*lớp Ival\_slider*

*: Ival\_box công khai, BBwidget được bảo vệ {/ \* ... \* /}; // cho BB*

*lớp Ival\_slider*

*: public Ival\_box, CWwidget được bảo vệ {/ \* ... \* /}; // 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:

*lớp Ival\_box {/ \* ... \* /};*

*lớp BB\_ival\_slider*

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

*lớp CW\_ival\_slider*

*: public Ival\_box, CWwidget được bảo vệ {/ \* ... \* /};*

*// ...*

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:

*lớp Ival\_box {/ \* ... \* /};*

*lớp Ival\_slider*

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

*lớp BB\_ival\_slider*

*: public Ival\_slider, BBwidget được bảo vệ {/ \* ... \* /};*

*lớp CW\_ival\_slider*

*: public Ival\_slider, CWwidget được bảo vệ {/ \* ... \* /};*

*// ...*

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*

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

*lớp CW\_ival\_slider*

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

hoặc bằng đồ thị:

*Ival\_box*

*BBslider CWslider Ival\_slider*

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

*BBwidget CWwidg et*

**VI.21.2.3.1 Phê bình**

Thiết kế lớp trừu tượng linh hoạt và gần như đơn giản để xử lý như thiết kế tương đương dựa trên 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**

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:

*lớp Ival\_maker {*

*công cộng:*

*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*

*công cộng:*

*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*

*công cộng:*

*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**

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**

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**

Ví dụ về vệ tinh sẽ là đá, mảnh vỡ từ các phương tiện vũ trụ cũ, vệ tinh liên lạc, và Interna- Trạm vũ trụ tional. 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, lớp Comm\_sat:*

*class Comm\_sat: vệ tinh công cộng, hiển thị công khai {*

*công cộng:*

*// ...*

*};*

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 {*

*công cộng:*

*tâm Pos ảo () const = 0; // Trung tâm của lực hấp dẫn*

*// ...*

*};*

*lớp được hiển thị {*

*công cộng:*

*ảo void draw () = 0;*

*// ...*

*};*

*class Comm\_sat: vệ tinh công cộng, hiển thị công khai {*

*công cộng:*

*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**

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

*vệ tinh lớp {*

*công cộng:*

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

*// ...*

*};*

*lớp được hiển thị {*

*công cộng:*

*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 {*

*công cộng:*

*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:

*lớp Telstar: public Comm\_sat {*

*công cộng:*

*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:

*lớp Satellite: public SimObj {*

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

*công cộng:*

*virtual Debug\_info get\_debug (); // gọi SimObj :: DBinf () và giải nén thông tin*

*// ...*

*};*

*lớp 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*

*công cộng:*

*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 {*

*công cộng:*

*void draw (); // hình ảnh hiển thị*

*// ...*

*};*

*cao bồi lớp {*

*công cộng:*

*void draw (); // rút súng từ bao da*

*// ...*

*};*

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

*// ...*

*};*

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 {*

*công cộng:*

*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.

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

Khi mỗi lớp chỉ có một lớp cơ sở trực tiếp, cấu trúc phân cấp lớp sẽ là một cây và một lớp 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 {*

*công cộng:*

*ghi đè void write ();*

*// ...*

*};*

*Class Receiver: public Stocking {*

*công cộng:*

*ghi đè void write ();*

*// ...*

*};*

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

*công cộng:*

*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*

**VI.21.3.5 Lớp cơ sở ảo**

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 {*

*công cộng:*

*Lưu trữ (const string & s); // lưu trữ trong tệp có tên s*

*ảo void read () = 0;*

*void ghi () = 0;*

*ảo ̃Storable ();*

*được bảo vệ:*

*chuỗi tên\_tệp;*

*Stocking (const Stocking &) = xóa;*

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

*};*

Với 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 {*

*công cộng:*

*ghi đè void write ();*

*// ...*

*};*

*Class Receiver: public virtual Stocking {*

*công cộng:*

*ghi đè void write ();*

*// ...*

*};*

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

*công cộng:*

*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.

**21.3.5.1 Xây dựng cơ sở ảo**

Sử dụng các đế ảo, bạn có thể tạo các mạng phức tạp. Đương nhiên, chúng tôi muốn giữ 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);*

*// ...*

*};*

*cấu trúc 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*

*// ...*

*};*

*lớp C: ảo V {*

*công cộng:*

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

*// ...*

*};*

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

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

*// ngầm nhận cơ sở ảo A từ B*

*công cộng:*

*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**

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 {*

*công cộng:*

*// vật liệu cơ bản*

*ảo void draw ();*

*};*

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

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

*// nội dung đường viền*

*được bảo vệ:*

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

*công cộng:*

*ghi đè void draw ();*

*};*

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

*// menu thứ*

*được bảo vệ:*

*void own\_draw (); // hiển thị menu*

*công cộng:*

*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*

*được bảo vệ:*

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

*công cộng:*

*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**

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:

*lớp 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:

*lớp BB\_ival\_slider*

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

*lớp Popup\_ival\_slider*

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

*lớp 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:

*lớp BB\_ival\_slider*

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

*lớp Popup\_ival\_slider*

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

*lớp 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\_popup\_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.

**3.6.1 Ghi đè các hàm cơ sở ảo**

Một lớp dẫn xuất có thể ghi đè một hàm ảo của lớp cơ sở ảo trực tiếp hoặc gián tiếp của nó. 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.

# II)CHƯƠNG TRÌNH ỨNG DỤNG

1. **Lí do chọn đề tài:**

Hiện nay số lượng sinh viên nhập học vào các trường học rất là đông,dẫn đến việc quản lí sinh viên gặp rất nhiều khó khăn và bất cập trong quá tình quản lí,dẫn tới việc có nhiều sai sót trong quá trình quản lí và xảy ra nhiều vấn đề trong qua trong quá trình tìm kiếm, lưu trữ danh sách các sinh viên.

Để giải quyết vấn đề này nhóm chúng em đã xây dựng một chương trình quản lí sinh viên với mong muốn sẽ giải quyết được phần nào những bất cập trong quản lí sinh viên của các trường học.

Mô

1. **Nội dung lí thuyết:**

**2.1) Lập trình hướng đối tượng là gì:**

là một kỹ thuật lập trình cho phép lập trình viên tạo ra các đối tượng trong code trừu tượng hóa các đối tượng.

**2.2) Các ưu điểm của lập trình hướng đối tượng:**

- Dựa trên nguyên lý kế thừa, trong quá trình mô tả các lớp có thể loại bỏ những chương trình bị lặp, dư. Và có thể mở rộng khả năng sử dụng các lớp mà không cần thực hiện lại. Tối ưu và tái sử dụng code hiệu quả.

- Đảm bảo rút ngắn thời gian xây dựng hệ thống và tăng năng suất thực hiện.

**2.3) Tính đóng gói (Encapsulation):**

Là cách để che dấu những tính chất xử lý bên trong của đối tượng, những đối tượng khác không thể tác động trực tiếp làm thay đổi trạng thái chỉ có thể tác động thông qua các method public của đối tượng đó.

khi ta muốn thay đổi các tính chất (properties) không thể tương tác trực tiếp với properties mà phải thông qua các method public được định nghĩa bên trong class.

VD: class SinhVien

{

protected:

string mssv;

string HoTen;

int Ngay, Thang, Nam;

int Khoa;

string DiaChi;

long long HocPhi;

public:

SinhVien();

~SinhVien();

virtual void Nhap();

virtual void Xuat();

virtual void TinhHocPhi()=0;

float getHocPhi();

string getTen();

string getmssv();

string getDiaChi();

};

2.4Tính kế thừa (Inheritance):

- Cho phép kế thừa lại những tính năng mà một đối tượng khác đã có, giúp tránh việc code lặp dư thừa mà chỉ xử lý công việc tương tự.

- Các lớp con kế thừa toàn bộ thành phần của lớp cha và không cần định nghĩa lại.

VD: Chúng tạo ra class SinhVienkithuat kế thừa class SinhVien ở ví dụ trên

class SinhVienKiThuat : public SinhVien

{

private:

int SoTinChi;

public:

SinhVienKiThuat();

~SinhVienKiThuat();

void Nhap();

void Xuat();

void TinhHocPhi();

};

2.5) Tính đa hình (Polymorphism)

Là một đối tượng thuộc các lớp khác nhau có thể hiểu cùng một thông điệp theo cách khác nhau.

class NhanVien

{

protected:

string hoTen;

float luong;

public:

NhanVien()

{

this->hoTen = "";

this->luong = 0.0;

}

void nhap()

{

cout << "Ho ten: ";

cin.ignore();

getline(cin, this->hoTen);

}

void xuat()

{

cout << "Ho ten: ";

cout << this->hoTen << endl;

}

void tinhLuong() {}

};

**2.6) Tính trừu tượng:**

- Tính trừu tượng là một tiến trình ẩn các chi tiết trình triển khai, chỉ hiển thị tính năng tới người dùng. Tính trừu tượng cho phép bạn loại bỏ tính chất phức tạp của đối tượng bằng cách chỉ đưa ra các thuộc tính và phương thức cần thiết của đối tượng.

- Tính trừu tượng giúp tập trung vào những cốt lõi cần thiết của đối tượng thay vì quan tâm đến cách mà nó thực hiện.

**3) Cấu trúc dữ liệu-Hàm vector:**

Vector trong C++ giống dynamic array (mảng động) nhưng có khả năng tự động thay đổi kích thước khi một phần tử được chèn hoặc xóa tùy thuộc vào nhu cầu của tác vụ được thực thi, với việc lưu trữ của chúng sẽ được vùng chứa tự động xử lý. Các phần tử vector được đặt trong contiguous storage (bộ nhớ liền kề) để chúng có thể được truy cập và duyệt qua bằng cách sử dụng iterator.

**Khái Quát chương trình:**

**Các thư viện:**

#include <iostream>

#include <vector>

#include <iomanip>

#include <Windows.h>

#include<string.h>

**Các class đã khai báo:**

class SinhVien

class SinhVienKiThuat

class SinhVienKinhTe

class QuanLySinhVien

**Các chức năng trong chương trình:**

**Nhập thông tin sinh viên:**

void SinhVien::Nhap()

{

cout << "\t\t\t\tNhap ma so sinh vien: ";

cin.ignore(256, '\n');

getline(cin, mssv);

cout << "\t\t\t\tNhap ho ten sinh vien: ";

fflush(stdin);

getline(cin, HoTen);

cout<<"\t\t\t\tNhap nam sinh: ";

cin >> Nam;

**Xuất thông tin sinh viên:**

void SinhVien::Nhap()

{

cout << "\t\t\t\tNhap ma so sinh vien: ";

cin.ignore(256, '\n');

getline(cin, mssv);

cout << "\t\t\t\tNhap ho ten sinh vien: ";

fflush(stdin);

getline(cin, HoTen);

cout<<"\t\t\t\tNhap nam sinh: ";

cin >> Nam;

**Sắp xếp sinh viên:**

void QuanLySinhVien :: SapXepTangDan()

{

if(SV.size()==0){

cout << "\n\t\t\t\t\t\*\*\*\*\*\*\*\*\*\* Danh sach rong \*\*\*\*\*\*\*\*\*" << endl;

}

else

{

for (int i=0; i < SV.size()-1; i++)

{

for (int j = i + 1; j < SV.size(); j++)

{

if(SV.at(i)->getHocPhi() > SV.at(j)->getHocPhi())

{

SinhVien \*t= SV.at(i);

SV.at(i)=SV.at(j);

SV.at(j)=t;

}

}

}

XuatDS();

}

}

void QuanLySinhVien :: SapXepGiamDan()

{

if(SV.size()==0)

{

cout << "\n\t\t\t\t\t\*\*\*\*\*\*\*\*\*\* Danh sach rong \*\*\*\*\*\*\*\*\*\*" << endl;

}

else

{

for (int i=0; i < SV.size()-1; i++)

{

for (int j = i + 1; j < SV.size(); j++)

{

if(SV.at(i)->getHocPhi() < SV.at(j)->getHocPhi())

{

SinhVien \*t= SV.at(i);

SV.at(i)=SV.at(j);

SV.at(j)=t;

}

}

}

XuatDS();

}

}

**Tìm kiếm thông tin sinh viên:**

void QuanLySinhVien::TimTen()

{

if(SV.size() == 0)

{

cout << "\n\t\t\t\t\t\*\*\*\*\*\*\*\* Danh sach rong \*\*\*\*\*\*\*\*" << endl;

}

else

{

SinhVien \*ten;

string timten;

cout << "\n\t\t\t\tNhap ten sinh vien can tim: ";

fflush(stdin);

getline(cin, timten);

for(int i=0; i<SV.size(); i++)

{

if(SV.at(i)->getTen() == timten)

{

ten = SV.at(i);

cout << "\n\n\t\t\t\t\tTHONG TIN TEN SINH VIEN CAN TIM"<<endl;

ten->Xuat();

break;

}

else

{

cout << "\n\t\t\t\tKhong co ten sinh vien trong danh sach can tim. Vui long nhap lai" << endl;

break;

}

}

}

}

**Xóa sinh viên:**

void QuanLySinhVien::XoaSinhVien()

{

if(SV.size()==0)

{

cout << "\n\t\t\t\t\t\*\*\*\*\*\*\*\*\*\* Danh sach rong \*\*\*\*\*\*\*\*\*\*" << endl;

}

else{

int n=SV.size();

string TayXoa;

cout<<"\n\n\t\t\tNhap ma so sinh vien can xoa : ";

fflush(stdin);

getline(cin,TayXoa);

for(int i=0; i<SV.size()-1;i++){

if(SV.at(i)->getMSSV()==TayXoa){

for(;i<SV.size()-1;i++){

for(int j=i+1;j<SV.size();j++){

SV.at(i)=SV.at(i+1);

}

}

}

n=n-1;

this->SV.at(i)->Xuat();

}

}

}

**Hướng dẫn sử dụng chương trình:**

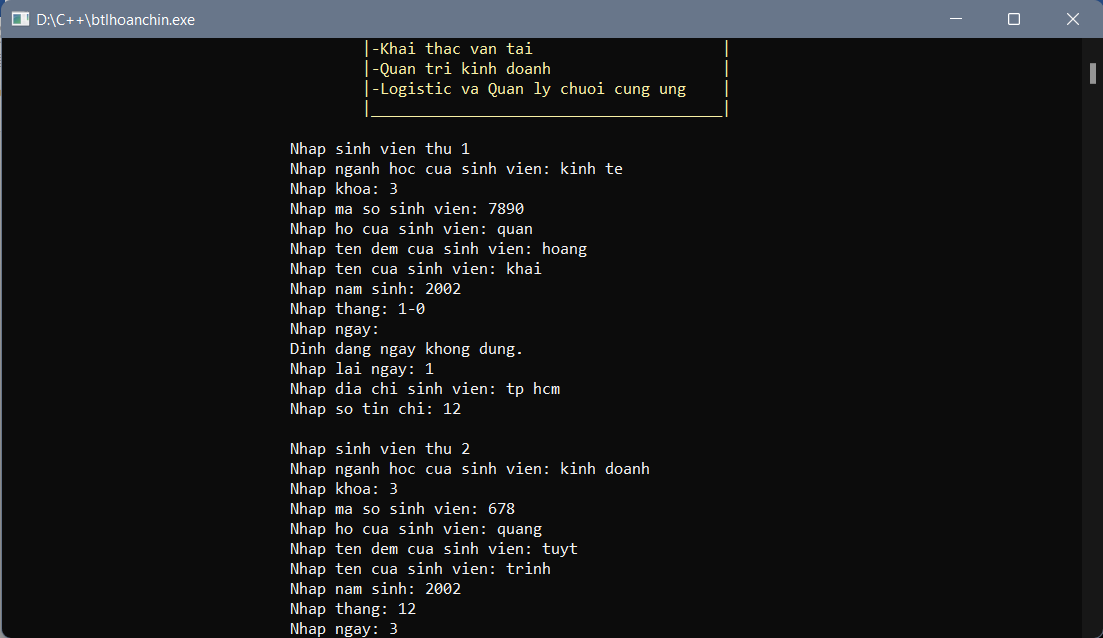
Khi chạy chương trình:



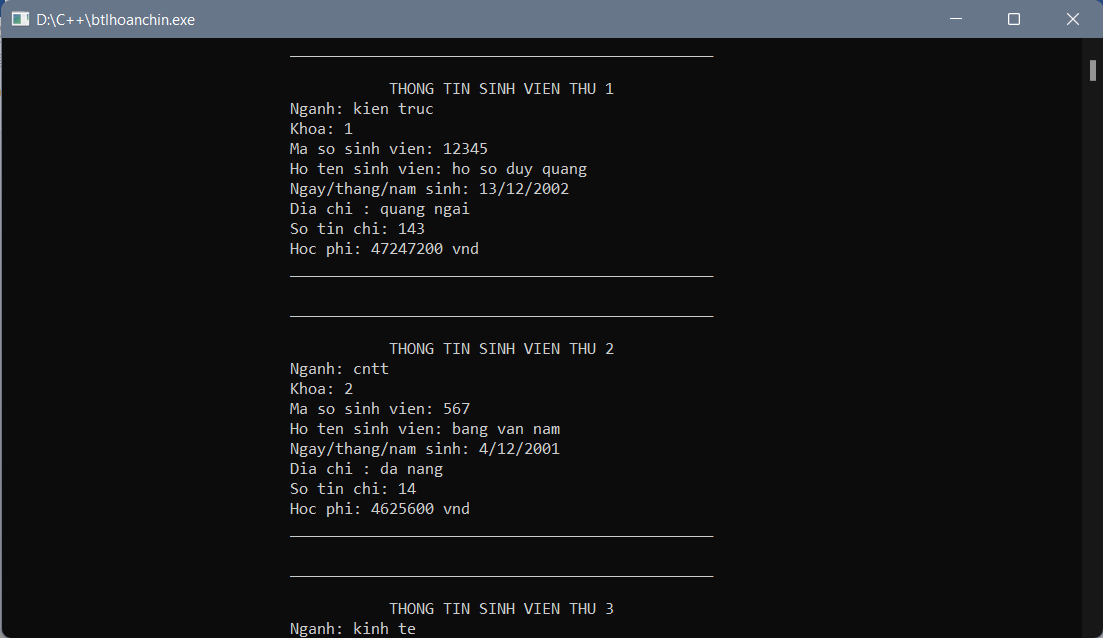
Chon chức năng từ 1 đến 8:

Chức năng 1:Bạn chọn số 1 từ bàn phím r nhấn phím enter

Nhập danh sách sinh viên gồm (số lượng sinh viên kĩ thuật, số lượng nhân viên kinh tế,(nhập ngành học sinh viên,nhập khoa, nhập mã số sinh viên,nhập tên của sinh viên , nhập ngày tháng năm của sinh viên,nhập số tinh chỉ của sinh viên)



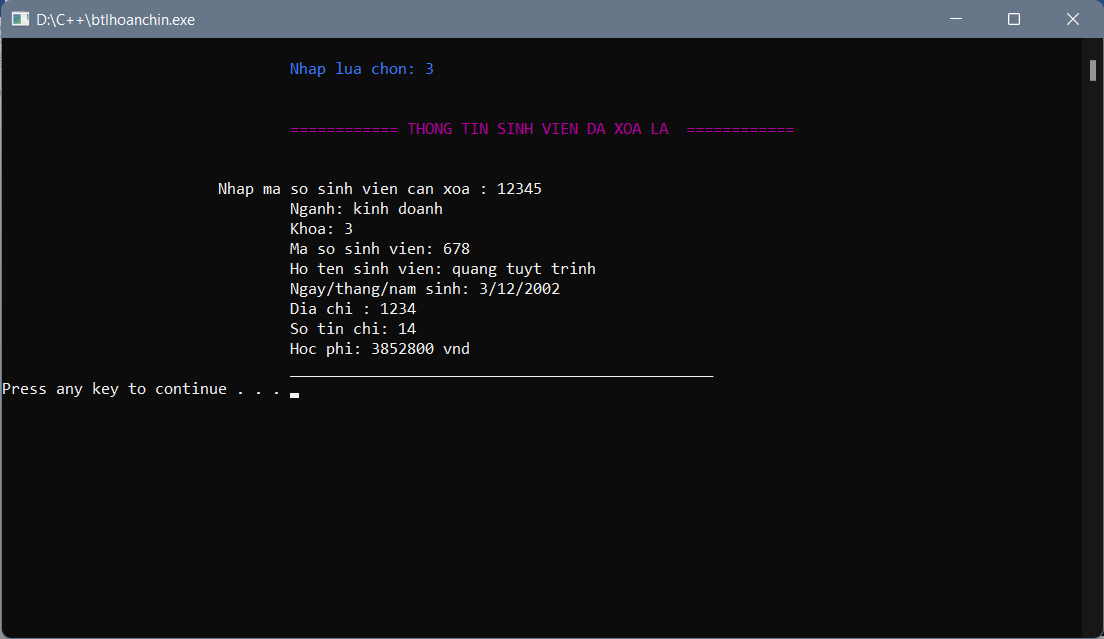
Chức năng 2: Bạn nhập số 2 từ bàn phím rồi nhấn phím enter



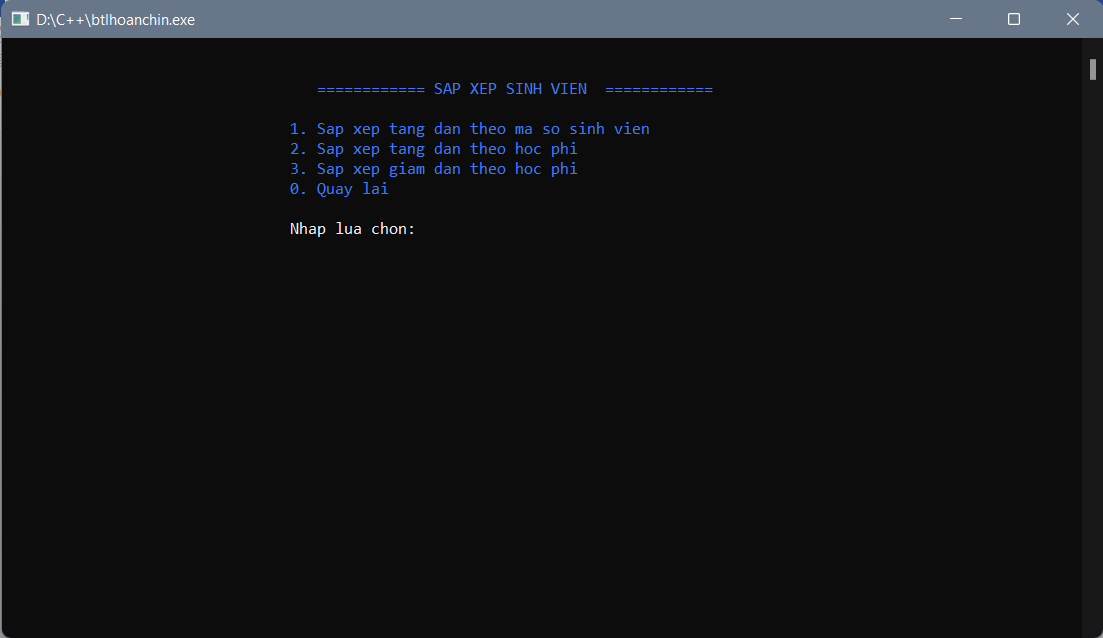
Nó sẽ in thông tin các sinh viên

Chức năng 3: bạn chọn số 3 rồi nhấn phím enter

Sau đó bạn chỉ cần nhập mã số sinh viên cần xóa

nó sẽ hiện ra thông tin nhân viên cần xóa và xóa đi nó.

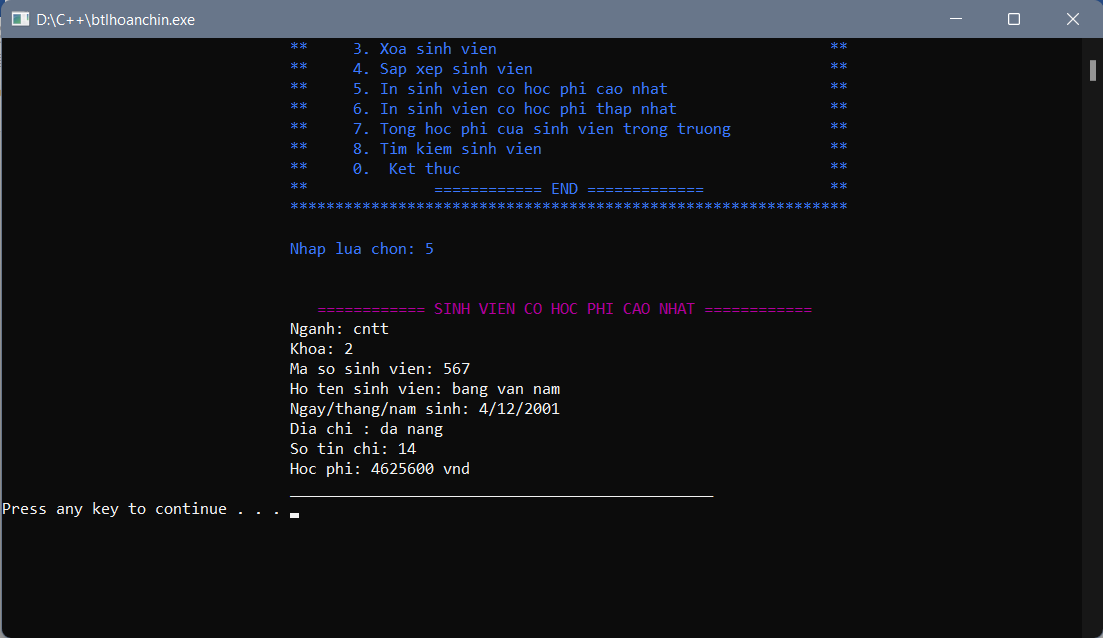
Chức năng 4: bạn chỉ cần nhấn số 4 và nhấn phím enter nó sẽ hiện ra sắp xếp sinh viên theo mong muốn của bạn



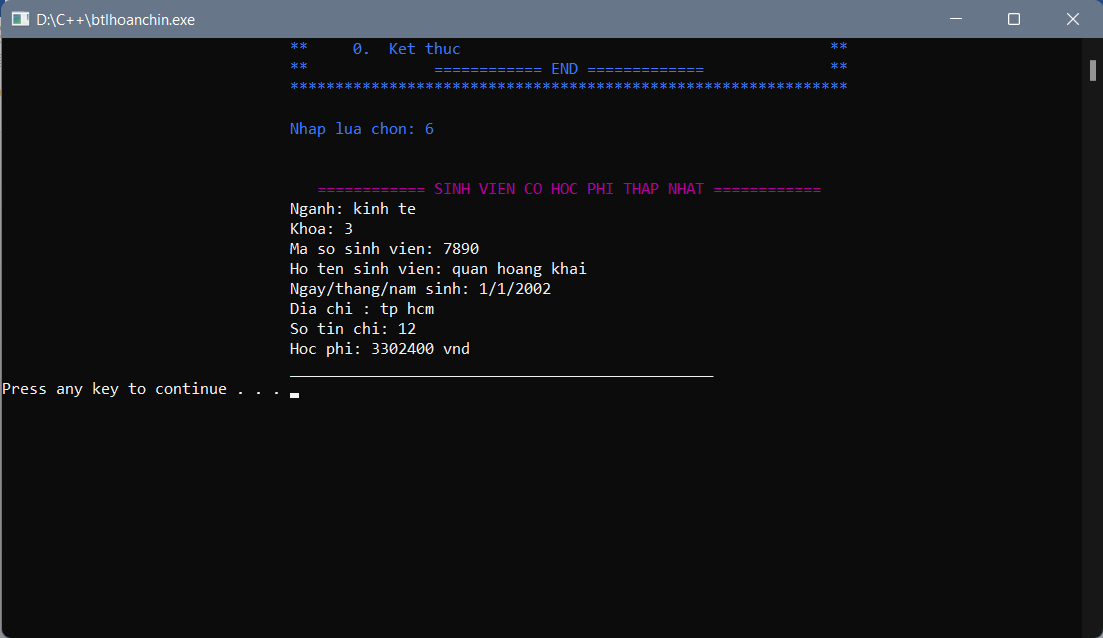
Chọn chức năng bạn muốn sắp xếp

Chức năng 5:

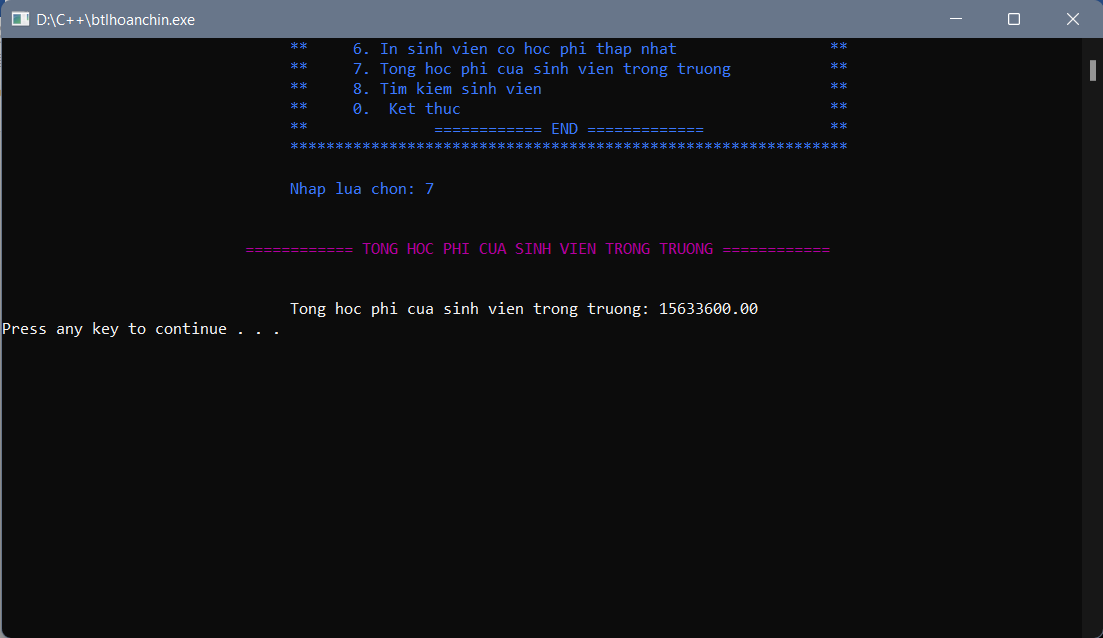
In thông tin ra sinh viên có học phí cao nhất:



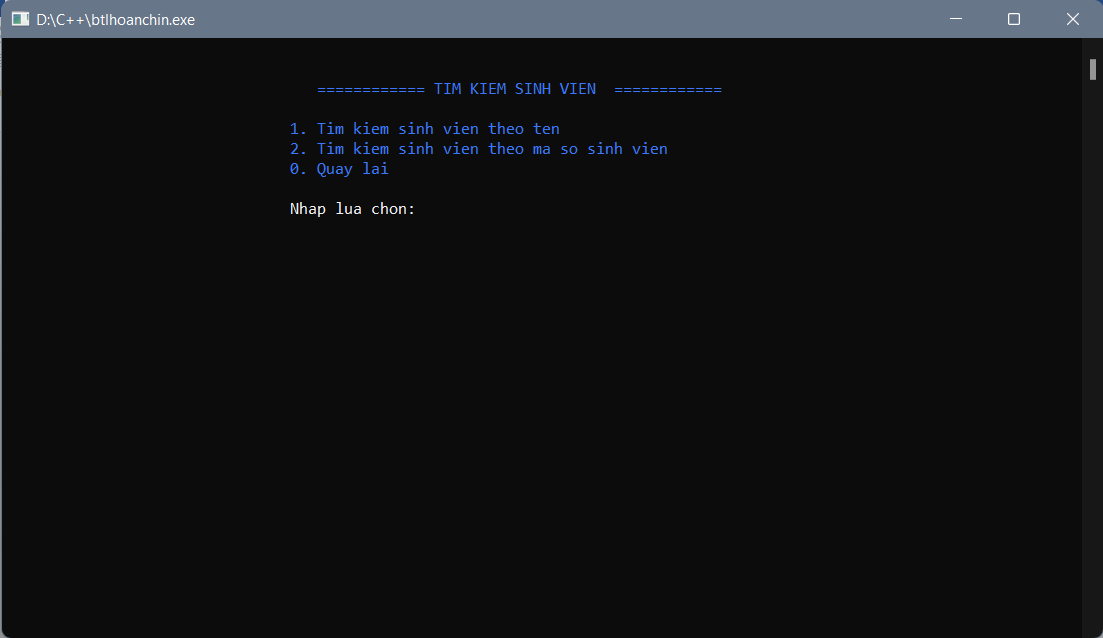
Chức năng 6:khi bạn chọn nó sẽ in ra sinh viên có học phí thấp nhất



Chức năng 7 Nó sẽ tổng kết học phí của sinh viên toàn trường



Chức năng 8:Chúng ta sẽ tìm kiếm sinh viên mã số sinh viên, tên sinh viên:



Cuối cùng khi chúng ta chọn 0 nó sẽ kết thúc chương trình.