**CHƯƠNG 16**

**16.2 Cơ bản về Class**

• 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 ghi nhớ dữ liệu- bia và các chức năng thành viên.

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

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

• Các nhà khai thác, ví dụ 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 đề cập.

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

**16.2.1 Chức năng thành viên**

Các hàm được khai báo trong định nghĩa lớp ( **cấu trúc** là một loại lớp;)được gọi là *thành viên* *chức năng* và có thể được gọi duy nhất cho một biến cụ thể của các loại thích hợp sử dụng tiêu chuẩn cú pháp để truy cập thành viên cấu trúc.

**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 bản sao của một đối tượng của lớp nó. Theo mặc định, bản sao của một đối tượng lớp là bản sao của mỗi thành viên.

**16.2.3 Kiểm soát truy cập**

Có một số lợi ích thu được từ việc hạn chế quyền truy cập vào cấu trúc dữ liệu đối với một danh sách các chức năng đã khai báo.Một lợi thế khác là người dùng tiềm năng cần chỉ kiểm tra các định nghĩa của các hàm thành viên để học cách sử dụng một lớp. Một lợi thế tinh tế hơn, nhưng quan trọng nhất, là rằng việc tập trung vào thiết kế một giao diện tốt chỉ đơn giản là dẫn đến mã tốt hơn bởi vì những suy nghĩ và nếu không thì thời gian dành cho việc gỡ lỗi sẽ được sử dụng cho các mối quan tâm liên quan đến việc sử dụng đúng cách.

Việc bảo vệ dữ liệu riêng tư dựa trên việc hạn chế sử dụng tên thành viên trong lớp. Nó có thể do đó có thể bị phá vỡ bởi thao tác địa chỉ và chuyển đổi kiểu rõ ràng

**16.2.4 Class và Struct**

Xây dựng **class X {...};** được gọi là *định nghĩa lớp*, nó định nghĩa một loại gọi là **X** . Vì lý do lịch sử, định nghĩa lớp là thường được gọi là *khai báo lớp* .

Theo định nghĩa, **struct** là một lớp trong đó các thành viên được mặc định là public; đó là, **struct S {/ \*** ... **\* /};** chỉ đơn giản là viết tắt của

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

**16.2.5 Cấu tạo**

Khi một lớp có một phương thức khởi tạo, tất cả các đối tượng của lớp đó sẽ được khởi tạo bởi một lời gọi phương thức khởi tạo. Vì một phương thức khởi tạo định nghĩa việc khởi tạo cho một lớp, chúng ta có thể sử dụng {}ký hiệu -initializer:

Date today = Date {23,6,1983};  
Date xmas {25,12,1990};

Date release1\_0 {10,12};

Các hàm tạo tuân theo các quy tắc nạp chồng giống như các hàm thông thường. Khi một giá trị đối số được sử dụng để biểu thị '' chọn giá trị mặc định '', giá trị được chọn phải nằm ngoài tập hợp các giá trị có thể có cho đối số.

**16.2.6 Hàm tạo rõ ràng**

Theo mặc định, một hàm tạo được gọi bởi một đối số duy nhất hoạt động như một chuyển đổi ngầm định từ đối số của nó- loại ment đến loại của nó. Chúng ta có thể chỉ định rằng một hàm tạo không được sử dụng như một chuyển đổi *ngầm định* . Một công trình-tor khai báo với từ khóa **explicit** chỉ có thể được sử dụng để khởi tạo và chuyển đổi rõ ràng. **Khởi tạo rõ ràng được gọi là *khởi tạo trực tiếp*.** Sự khác biệt giữa khởi tạo trực tiếp và sao chép được duy trì cho quá trình khởi tạo danh sách.

**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 lại.

Chúng ta có thể giải quyết vấn đề đó bằng cách giới thiệu các đối số mặc định để giảm số lượng hàm tạo.

**16.2.8 Định nghĩa hàm trong lớp**

Một hàm thành viên được định nghĩa trong định nghĩa lớp - thay vì chỉ được khai báo ở đó - được coi là một hàm thành viên nội tuyến. Có nghĩa là, định nghĩa trong lớp của các hàm thành viên dành cho các hàm nhỏ, hiếm khi được sửa đổi, được sử dụng thường xuyên.

**16.2.9 Tính đột biến**

**16.2.9.1 Các chức năng thành viên không đổi**

Khi một hàm thành viên **const** được định nghĩa bên ngoài lớp của nó, thì hậu tố **const** là bắt buộc. Nói cách khác, **const** là một phần của kiểu **Date :: day ()** , **Date :: month ()** và **Date :: year ()** . Một hàm thành viên **const** có thể được gọi cho cả đối tượng **const** và không phải **const** , trong khi một hàm thành viên **const chỉ**có thể được gọi cho các đối tượng không phải **const**. Ví dụ:

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

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

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

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

**cd.add\_year (1);**

**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 mem-ber. 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. Đây thường được gọi là hằng số logic

**16.2.9.3 Có thể thay đổi**

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 **const** sự vật.

**16.2.9.4 Tính đột biến thông qua hướng dẫn**

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 thay đổi dữ liệu trong một đối tượng riêng biệt và truy cập nó một cách gián tiếp.

**16.2.10 Tự tham khảo**

**16.2.11 Quyền truy cập thành viên**

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 bởi áp dụng **->** (mũi tên) vận hành một con trỏ đến một đối tượng của lớp **X**.Nếu chúng ta muốn tham chiếu đến một thành viên nói chung, thay vì một thành viên của một đối tượng cụ thể, chúng ta đủ điều kiện theo tên lớp theo sau là ::.

**16.2.12 [static] Thành viên**

Đây là loại ràng buộc khiến một lớp trở nên vô dụng bên ngoài ngữ cảnh mà nó được viết lần đầu tiên. Một biến là một phần của một lớp, nhưng không phải là một phần của một đối tượng của lớp đó, được gọi là thành viên **tĩnh**. Có chính xác một bản sao của một thành viên **tĩnh**thay vì một bản sao cho mỗi đối tượng, như đối với các thành viên không **tĩnh** thông thường .Tương tự, một chức năng cần quyền truy cập vào các thành viên của lớp, nhưng không cần phải được gọi cho một đối tượng cụ thể, được gọi là một hàm thành viên **tĩnh** .

**16.2.13 Các loại thành viên**

Một *lớp thành viên* (thường được gọi là *lớp lồng nhau* ) có thể tham chiếu đến các kiểu và các thành viên **tĩnh** trong bao bọc của nó lớp. Nó chỉ có thể tham chiếu đến các thành viên không **tĩnh** khi nó được cung cấp cho một đối tượng của lớp bao quanh tham khảo.Các lớp thành viên là một sự tiện lợi về mặt ký hiệu hơn là một tính năng có tầm quan trọng cơ bản.

**16.3 Lớp cụ thể**

Một lớp trừu tượng có nghĩa là được sử dụng như một lớp cơ sở nơi một số hoặc tất cả các hàm được khai báo hoàn toàn là ảo và do đó không thể được khởi tạo. Một lớp cụ thể là một lớp bình thường không có chức năng hoàn toàn ảo và do đó có thể được khởi tạo.

**16.3.1Chức năng thành viên**

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.

**16.3.2 Chức năng của người trợ giúp**

Thông thường, một lớp có một số hàm được liên kết với nó mà không cần được định nghĩa trong lớp bởi vì họ không cần quyền truy cập trực tiếp vào đại diện.

**Chương 17**

**17.1 Hàm khởi tạo và hàm huỷ**

**17.1.1 Hàm khởi tạo**

Hàm khởi tạo là một hàm thành viên đặc biệt của một lớp. Nó sẽ được tự động gọi đến khi một đối tượng của lớp đó được khởi tạo.

**17.1.2 Hàm huỷ**

Hàm huỷ cũng là một hàm thành viên đặc biệt giống như hàm tạo, nó được dùng để phá huỷ hoặc xoá một đối tượng trong lớp.

**17.2 Khởi tạo đối tượng của lớp**

**17.2.1 Hàm khởi tạo không tham số (mặc định)**

Hàm tạo loại này sẽ không truyền vào bất kì một đối số nào.Ví dụ:

class Vector {

public:

Vector(); // Hàm khởi tạo mặc định};

**17.2.2 Hàm khởi tạo sao chép**

Hàm khởi tạo sao chép là một hàm tạo mà tạo một đối tượng bằng việc khởi tạo nó với một đối tượng của cùng lớp đó, mà đã được tạo trước đó.

**17.3 Sao chép và di chuyển**

**17.3.1 Sao chép**

Sao chép một lớp X bằng 2 cách:

+Sao chép hàm tạo: X(const X&)

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

**17.3.2 Sao chép cơ sở**

Đối với mục đích sao chép, một cơ sở là một thành viên: để sao chép một đối tượng của một lớp bạn có thể sao chép cơ sở của nó. Ví dụ:

struct B1 {

B1();

B1(const B1&); };

struct B2 {

B2(int);

B2(const B2&); };

struct D : B1, B2 {

D(int i) :B1{}, B2{i}, m1{}, m2{2∗i} {}

D(const D& a) :B1{a}, B2{a}, m1{a.m1}, m2{a.m2} {}

B1 m1;

B2 m2;};

D d {1}; // construct with int argument

D dd {d}; // copy construct

**17.3.3 Cắt đối tượng**

Cắt đối tượng xảy ra khi một đối tượng lớp có nguồn gốc được gán cho một đối tượng lớp cơ sở, các thuộc tính bổ sung của một đối tượng lớp có nguồn gốc được cắt ra để tạo thành đối tượng lớp cơ sở.

**17.3.4 Di chuyển**

Hàm tạo di chuyển lấy giá trị tham chiếu tới một đối tượng của lớp, và được dùng để hiện thực chuyển quyền sở hữu tài nguyên của đối tượng tham số.

**17.4 Tạo hoạt động mặc định**

**17.4.1 Mặc định rõ ràng**

Vì việc tạo ra các hoạt động mặc định khác có thể bị chặn, nên phải có một cách để lấy lại mặc định. Ngoài ra, một số người muốn xem danh sách đầy đủ các thao tác trong chương trình văn bản ngay cả khi danh sách đầy đủ đó không cần thiết.

Ví dụ:

class gslice {

valarray<siz e\_t> siz e;

valarray<siz e\_t> stride;

valarray<siz e\_t> d1;

public:

gslice() = default;

~gslice() = default;

gslice(const gslice&) = default;

gslice(gslice&&) = default;

gslice& operator=(const gslice&) = default;

gslice& operator=(gslice&&) = default; };

**17.4.2 Default Operations**

Lưu ý rằng giá trị của một đối tượng được chuyển đến của một kiểu dựng sẵn là không thay đổi. Đó là điều đơn giản và nhanh nhất để trình biên dịch thực hiện. Nếu chúng ta muốn một điều gì đó khác được thực hiện cho một thành viên của một lớp, chúng ta phải viết các hoạt động di chuyển của chúng ta cho lớp đó.

**17.4.3 Sử dụng thao tác mặc định**

Phần này trình bày một số ví dụ chứng minh cách sao chép, di chuyển và hủy được liên kết một cách hợp lý. Nếu chúng không được liên kết, các lỗi hiển nhiên khi bạn nghĩ về chúng sẽ không được trình biên dịch bắt gặp.

**17.4.3.1 Trình xây dựng mặc định**

Trình xây dựng mặc định là người xây dựng không lấy bất kỳ đối số nào. Nó không có thông số.

**17.4.3.2 Bất biến**

[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).

17.4.3.3 Tài nguyên bất biến

Nhiều ứng dụng quan trọng và rõ ràng nhất của bất biến liên quan đến quản lý tài nguyên.Ví dụ:

template<class T> class Handle {  
T∗ p;  
public:  
Handle(T∗ pp) :p{pp} { }  
T& operator∗() { return ∗p; }  
 ̃Handle() { delete p; } };

17.4.3.4 Bất biến được chỉ định một phần

Các ví dụ rắc rối dựa trên các bất biến nhưng chỉ thể hiện một phần chúng thông qua các hàm tạo hoặc hàm hủy là hiếm hơn nhưng không phải là chưa từng thấy.Ví dụ:

class Tic\_tac\_toe {  
public:  
Tic\_tac\_toe(): pos(9) {} // always 9 positions  
Tic\_tac\_toe& operator=(const Tic\_tac\_toe& arg){  
for(int i = 0; i<9; ++i)  
pos.at(i) = arg.pos.at(i);  
return ∗this; }  
// ... other operations ...  
enum State { empty, nought, cross };  
private:  
vector<State> pos;  
};

17.4.3.5 Hàm đã xoá

Chúng ta có thể ‘‘ xóa ’’ một hàm; nghĩa là, chúng ta có thể nói rằng một hàm không tồn tại để cố gắng sử dụng nó (một cách ngầm hiểu hoặc rõ ràng) là một lỗi. Cách sử dụng rõ ràng nhất là loại bỏ chức năng. Ví dụ:

class Base {  
// ...  
Base& operator=(const Base&) = delete;// disallow copying  
Base(const Base&) = delete;  
Base& operator=(Base&&) = delete; // disallow moving  
Base(Base&&) = delete;  
};  
Base x1;  
Base x2 {x1}; // error : no copy constructor

**Chương 18: Nạp Chồng Toán Tử**

**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. 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 khái niệm mà các toán tử được sử dụng thông thường không phải là các kiểu tích hợp sẵn trong C ++, vì vậy chúng phải. Ví dụ:

class complex {  
 double re, im;  
 public:  
 complex(double r, double i) :re{r}, im{i} { }  
 complex operator+(complex);  
 complex operator∗(complex);  
};

**18.2 Các chức năng của nhà điều hành**

Các hàm xác định ý nghĩa cho các toán tử sau (§10.3) có thể được khai báo:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| + | − | ∗ | / | % | ˆ | & |
| | | ˜ | ! | = | < | > | += |
| −= | ∗= | /= | %= | ˆ= | &= | |= |
| << | >> | >>= | <<= | == | != | <= |
| >= | && | || | ++ | −− | −>∗ | , |
| −> | [] | () | new | new[] | delete | delete[] |

Người dùng không thể xác định các toán tử sau:  
:: giải quyết phạm vi lựa  
. chọn thành viên   
.∗ lựa chọn thành viên thông qua con trỏ đến thành viên

18.2.1 Toán tử nhị phân và đơn nguyên

Một toán tử nhị phân có thể được định nghĩa bởi một hàm thành viên không tĩnh nhận một đối số hoặc một hàm không phải là bộ nhớ nhận hai đối số. Đối với bất kỳ toán tử nhị phân nào @, aa @ bb có thể được hiểu là aa.operator @ (bb) hoặc toán tử @ (aa, bb). Nếu cả hai đều được xác định, giải pháp quá tải sẽ xác định cách giải thích nào, 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

18.2.2 Ý nghĩa được xác định trước cho toán tử

Ý nghĩa của một số toán tử dựng sẵn được định nghĩa tương đương với một số kết hợp của các toán tử khác trên cùng các đối số. Các quan hệ như vậy không giữ cho các toán tử do người dùng xác định trừ khi người dùng định nghĩa chúng.

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. 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 thuộc loại do người dùng xác định. Đặc biệt, không thể định nghĩa một hàm toán tử hoạt động độc quyền trên con trỏ.

18.2.4 Chuyển đối tượng

Đối với các đối số, chúng ta có hai lựa chọn chính

- Giá trị chuyển tiếp

- Tham chiếu qua

Đối với các đối tượng nhỏ, chẳng hạn như từ một đến bốn từ, gọi theo giá trị thường là một phương án thay thế khả thi và thường là phương án mang lại hiệu suất tốt nhất. Các đối tượng lớn hơn, ta chuyển qua tham chiếu. Đặc biệt, chúng tôi sử dụng const các tham chiếu để truyền các đối tượng lớn không có nghĩa là được sửa đổi bởi hàm được gọi

18.2.5 Các toán tử trong Không gian tên

Toán tử hoặc là một thành viên của một lớp hoặc được định nghĩa trong một số không gian tên. Các toán tử được định nghĩa trong không gian tên có thể được tìm thấy dựa trên kiểu toán hạng của chúng cũng như các hàm có thể được tìm thấy dựa trên kiểu đối số của chúng.  
Lưu ý rằng trong tra cứu toán tử không có ưu tiên nào dành cho các thành viên hơn là những người không phải là thành viên. Điều này khác với tra cứu các hàm được đặt tên . Việc thiếu ẩn các toán tử đảm bảo rằng các toán tử tích hợp không bao giờ không thể truy cập được và người dùng có thể cung cấp các ý nghĩa mới cho một toán tử mà không cần sửa đổi các khai báo lớp hiện có

**18.3 Một loại số phức**

- Toán tử thành viên và không phải thành viên  
- Số học chế độ hỗn hợp  
- Chuyển đổi - Chuyển đổi toán hạng  
- Hằng  
- Hàm Accessor  
- Hàm trợ giúp

**18.4 Chuyển đổi loại**

Việc chuyển đổi kiểu có thể được thực hiện bằng

- Một hàm tạo nhận một đối số duy nhất  
- Toán tử chuyển đổi

18.4.1 Chuyển đổi toán tử

Sử dụng một hàm tạo lấy một đối số duy nhất để chỉ định chuyển đổi kiểu là thuận tiện nhưng có những tác động có thể không mong muốn. Ngoài ra, một hàm tạo không thể chỉ định chuyển đổi ngầm định từ kiểu do người dùng xác định sang kiểu cài sẵn (vì kiểu cài sẵn không phải là lớp) hoặc một chuyển đổi từ một lớp mới sang một lớp đã xác định trước đó (mà không sửa đổi khai báo cho lớp cũ).

Các hàm chuyển đổi dường như đặc biệt hữu ích để xử lý cấu trúc dữ liệu khi việc đọc (được thực thi bởi toán tử chuyển đổi) là nhỏ, trong khi việc gán và khởi tạo rõ ràng là ít tầm thường hơn.

18.4.2 Chuyển đổi toán tử Explicit

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 nó chỉ áp dụng cho việc khởi tạo trực tiếp, trong đó một phương thức khởi tạo tường minh tương đương sẽ được sử dụng.

18.4.3 Khả Nghi

Các quy tắc chuyển đổi không phải là quy tắc đơn giản nhất để thực hiện, cũng không phải là tài liệu đơn giản nhất, cũng không phải là quy tắc chung nhất có thể được nghĩ ra. Tuy nhiên, chúng an toàn hơn đáng kể và độ phân giải kết quả thường ít gây ngạc nhiên hơn so với các giải pháp thay thế. Việc giải quyết sự mơ hồ theo cách thủ công sẽ dễ dàng hơn nhiều so với việc tìm ra lỗi do chuyển đổi không đáng ngờ.

Sự khăng khăng về phân tích từ dưới lên nghiêm ngặt ngụ ý rằng kiểu trả về không được sử dụng trong giải quyết quá tải. Lý do cho sự lựa chọn thiết kế này một phần là phân tích từ dưới lên nghiêm ngặt dễ hiểu hơn và một phần là nó không được coi là công việc của trình biên dịch để quyết định độ chính xác mà lập trình viên có thể muốn bổ sung.

**Chương 19: Toán Tử Đặc Biệt**

**19.1 Giới thiệu**

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

**19.2 Các toán tử đặc biệt**

Các toán tử : [] () −> ++ −− new delete

19.2.1 Toán tử chỉ số con

Một hàm toán tử [] có thể được sử dụng để cung cấp ý nghĩa cho các chỉ số con cho các đối tượng lớp. Đối số thứ hai (chỉ số con) của một hàm toán tử [] có thể thuộc bất kỳ loại nào. Điều này giúp bạn có thể xác định vectơ, mảng kết hợp, v.v.

19.2.2 Gọi hàm

Lời gọi hàm, nghĩa là, biểu thức ký hiệu (danh sách biểu thức), 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. Danh sách đối số cho operator () () được đánh giá và kiểm tra theo quy tắc truyền đối số thông thường. Việc ghi đè toán tử cuộc gọi hàm dường như hữu ích chủ yếu để xác định các kiểu chỉ có một thao tác duy nhất và cho các kiểu mà một thao tác chiếm ưu thế. Gọi toán tử còn được biết là ứng dụng toán tử.

19.2.3 Toán tử tham chiếu

Toán tử tham chiếu, -> (còn được gọi là *mũi tên* toán tử), có thể được định nghĩa là toán tử hậu tố một ngôi. Toán tử -> phải là một hàm thành viên không tĩnh. Nếu được sử dụng, kiểu trả về của nó phải là một con trỏ hoặc một đối tượng của một lớp mà bạn có thể áp dụng ->. Mặc dù có sự giống nhau giữa -> và. (dấu chấm), nhưng không có cách nạp chồng toán tử .(dấu chấm)

19.2.4 Tăng và Giảm

Khi mọi người phát minh ra '' con trỏ thông minh '', họ thường quyết định cung cấp toán tử tăng dần ++ và toán tử giảm dần −− để phản ánh việc sử dụng của các toán tử này cho các kiểu cài sẵn. 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 tiền tố và hậu tốToán tử tăng trước có thể trả về một tham chiếu đến đối tượng của nó. Toán tử tăng sau phải tạo một đối tượng mới để trả về

19.2.5 Phân bổ và Giao dịch

Toán tử new có được 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 định nghĩa lại toán tử toàn cục new () và toán tử delete () hoặc định nghĩa toán tử new () và toán tử delete () cho một lớp cụ thể.

Toán tử thành viên new () s và toán tử delete () s hoàn toàn là các thành viên tĩnh. Do đó, họ không có con trỏ này và không sửa đổi một đối tượng. Chúng cung cấp bộ nhớ mà một phương thức khởi tạo có thể khởi tạo và một phương thức hủy có thể dọn dẹp.

19.2.6 Chữ viết do người dùng định nghĩa

Có bốn loại ký tự có thể được thêm hậu tố để tạo thành một ký tự do người dùng xác định: Một ký tự số nguyên, Một ký tự dấu phẩy động, Một ký tự chuỗi, Một ký tự theo nghĩa đen. Lý do của việc yêu cầu số lượng ký tự là nếu chúng ta muốn có '' một loại chuỗi khác '', chúng ta hầu như luôn muốn biết số lượng ký tự.

Một toán tử theo nghĩa đen chỉ cần một const char∗ đối số (và không có kích thước) có thể được áp dụng cho các ký tự số nguyên và dấu phẩy động. MỘT *toán tử mẫu* là một toán tử theo nghĩa đen nhận đối số của nó như một gói tham số mẫu, chứ không phải là một đối số hàm

**19.3 Một lớp chuỗi**

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

19.3.1 Các hoạt động cần thiết

Lớp string cung cấp tập hợp thông thường của các hàm tạo, một hàm hủy và các hoạt động gán

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 thông thường (nghĩa là sử dụng []), hiệu quả tối đa và đã kiểm tra phạm vi. Nhưng bạn không thể có tất cả các thuộc tính này cùng một lúc.

19.3.3 Đại diện

Đại diện cho String đã được chọn để đáp ứng ba mục tiêu:  
- Để dễ dàng chuyển đổi một chuỗi kiểu C (ví dụ: một chuỗi ký tự) thành Stringvà cho phép dễ dàng truy cập vào các ký tự của một String như một chuỗi kiểu C  
- Để giảm thiểu việc sử dụng cửa hàng miễn phí  
- Để thêm ký tự

19.3.4 Hàm thành viên

Tất cả các String các thành viên cẩn thận không sửa đổi một đại diện mới trước khi họ chắc chắn rằng một đại diện mới có thể được đưa vào. Đặc biệt, họ không delete cho đến sau khi có thể New các hoạt động đã được thực hiện. Trên thực tế String các thành viên cung cấp bảo đảm ngoại lệ mạnh mẽ

19.3.5 Chức năng của người trợ giúp

Để hoàn thành lớp String, Tôi cung cấp một tập hợp các chức năng hữu ích, luồng I / O, hỗ trợ cho phạm vi vòng lặp, so sánh và nối.

ostream& operator<<(ostream& os, const String& s){  
return os << s.c\_str();  
}  
istream& operator>>(istream& is, String& s){  
s = "";   
is>>ws;  
char ch = ' ';  
while(is.get(ch) && !isspace(ch))  
s += ch;  
return is;  
}

19.3.6 Sử dụng chuỗi của chúng tôi  
Chương trình chính chỉ đơn giản thực hiện String toán tử một chút:  
int main(){  
String s ("abcdefghij");  
cout << s << '\n';  
s += 'k';  
s += 'l';  
s += 'm';  
s += 'n';  
cout << s << '\n';  
String s2 = "Hell";  
s2 += " and high water";  
cout << s2 << '\n';  
String s3 = "qwerty";  
s3 = s3;  
String s4 ="the quick brown fox jumped over the lazy dog"; s4 = s4;  
cout << s3 << " " << s4 << "\n";  
cout << s + ". " + s3 + String(". ") + "Horsefeathers\n";  
String buf;  
while (cin>>buf && buf!="quit")  
cout << buf << " " << buf.siz e() << " " << buf.capacity() << '\n';  
}

19.4 Hàm bạn

Một khai báo hàm thành viên bình thường chỉ định ba điều khác biệt về mặt logic:  
Hàm có thể truy cập phần riêng của khai báo lớp.  
Hàm thuộc phạm vi của lớp.  
Hàm phải được gọi trên một đối tượng (có cái này con trỏ).

19.4.1 Tìm bạn

Một người bạn trước đây phải là bao quanh một cách ẩn lớp đang khai báo nó là một friend. Các phạm vi bên ngoài phạm vi không gian tên bao quanh trong cùng không được coi là tên được khai báo đầu tiên là friend. Một hàm friend có thể được tìm thấy thông qua các đối số của nó ngay cả khi nó không được khai báo trong phạm vi kèm theo ngay lập tức

19.4.2 Bạn bè và thành viên

Khi nào chúng ta nên sử dụng hàm kết bạn và khi nào thì hàm thành viên là lựa chọn tốt hơn để chỉ định một hoạt động? Đầu tiên, chúng tôi cố gắng giảm thiểu số lượng hàm truy cập vào biểu diễn 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 '' Có nên là thành viên,tĩnh thành viên, hay một người bạn? " Một số hoạt động phải là thành viên - ví dụ, hàm tạo, hàm hủy và hàm ảo - nhưng thường có một sự lựa chọn. Bởi vì tên thành viên là cục bộ của lớp, một hàm yêu cầu quyền truy cập trực tiếp vào biểu diễn phải là một thành viên trừ khi có lý do cụ thể khiến nó không phải là một thành viên.

**Chương 20: Derived Classes**

**20.2 Các lớp có nguồn gốc**

Cân nhắc xây dựng một chương trình giao dịch với những người được một công ty tuyển dụng. Một chương trình như vậy có thể có cấu trúc dữ liệu như sau:

struct Employee {

string first\_name , family\_name;

char middle\_initial;

Date hiring\_date;

short depar tment;

// ...

};

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

class Employee; // chỉ khai báo, không có định nghĩa

class Manager : public Employee { // error: Nhân viên không được xác định

// ...

};

**20.2.1 Chức năng thành viên**

Các cấu trúc dữ liệu đơn giản, chẳng hạn như Nhân viên và Người quản lý, thực sự không thú vị và thường không đặc biệt hữu ích. Chúng ta cần cung cấp một kiểu thích hợp với một tập hợp các thao tác phù hợp và chúng ta cần làm như vậy mà không bị ràng buộc vào các chi tiết của một biểu diễn cụ thể. Ví dụ:

class Employee {

public:

void print() const;

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

// ...

private:

string first\_name , family\_name;

char middle\_initial;

// ...

};

class Manager : public Employee {

public:

void print() const;

// ...

};

**20.3 Phân cấp lớp**

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

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

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

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

**20.3.1 Trường loại**

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

struct Employee {

enum Empl\_type { man, empl };

Empl\_type type;

Employee() : type{empl} { }

string first\_name , family\_name;

char middle\_initial;

Date hiring\_date;

short depar tment;

// ...

};

struct Manager : public Employee {

Manager() { type = man; }

list<Employee∗> group; // people managed

short level;

// ...

};

**20.3.2 Chức năng ảo**

Các hàm ảo khắc phục các vấn đề với giải pháp trường kiểu bằng cách cho phép lập trình viên khai báo các hàm trong lớp cơ sở có thể được định nghĩa lại trong mỗi lớp dẫn xuất. Trình biên dịch và trình liên kết sẽ đảm bảo sự tương ứng chính xác giữa các đối tượng và các chức năng được áp dụng cho chúng. Ví dụ:

class Employee {

public:

Employee(const string& name, int dept);

virtual void print() const;

// ...

private:

string first\_name , family\_name;

short depar tment;

// ...

};

**20.3.5.1 Trình tạo kế thừa**

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

template<class T>

struct Vector : std::vector<T> {

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

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

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

};

**20.4 Các lớp trừu tượng**

Nhiều lớp giống với lớp Employee ở chỗ chúng hữu ích như chính chúng, làm giao diện cho các lớp dẫn xuất và là một phần của việc triển khai các lớp dẫn xuất. Đối với các lớp như vậy, các kỹ thuật được mô tả trong §20.3.2 là đủ. Tuy nhiên, không phải tất cả các lớp đều tuân theo khuôn mẫu đó. Một số lớp, chẳng hạn như một lớp Shape, đại diện cho các khái niệm trừu tượng mà đối tượng không thể tồn tại. Shape chỉ có ý nghĩa như là cơ sở của một số lớp bắt nguồn từ nó. Điều này có thể được thấy từ thực tế là không thể cung cấp các định nghĩa hợp lý cho các chức năng ảo của nó:

class Shape {

public:

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

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

// ...

};

**20.5 Kiểm soát truy cập**

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

* Nếu nó là private, tên của nó chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của lớp mà nó được khai báo.
* Nếu nó là protected, tên của nó chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của lớp mà nó được khai báo và bởi các hàm thành viên và bạn bè của các lớp dẫn xuất từ lớp này (xem §19.4).
* Nếu nó là public, tên của nó có thể được sử dụng bởi bất kỳ chức năng nào.

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

template<class T>

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

{

Link∗ lnk = get\_free();

lnk−>val = val;

lnk−>next = head;

head = lnk;

}

template<class T>

T List<T>::g et()

{

if (head == 0)

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

Link∗ p= head;

head = p−>next;

p−>next = free;

free = p;

return p−>val;

}

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

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

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

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

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

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

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

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

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

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

class Std\_interface {

public:

virtual void start() = 0;

virtual void suspend() = 0;

virtual void resume() = 0;

virtual void quit() = 0;

virtual void full\_size() = 0;

virtual void small() = 0;

virtual ̃Std\_interface() {}

};

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

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

class Text : public Std\_interface {

public:

void start();

void suspend();

// ...

virtual void print();

private:

vector s;

};

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

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

**Chương 21: CẤU TRÚC PHÂN CẤP LỚP**

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

User Ival\_box

application

value

(via”system”) set\_value() get\_value()

Cách tiếp cận chung là xây dựng một '' hệ thống giao diện người dùng ảo '' để ứng dụng sử dụng. Hệ thống này cung cấp một số dịch vụ được cung cấp bởi các hệ thống giao diện người dùng hiện có. Nó có thể được thực hiện trên nhiều hệ thống khác nhau để đảm bảo tính di động của mã ứng dụng. Đương nhiên, có những cách khác để cách ly một ứng dụng khỏi hệ thống giao diện người dùng.

**21.2.1 Kế thừa triển khai**  
Giải pháp đầu tiên là một cấu trúc phân cấp lớp sử dụng kế thừa triển khai (như thường thấy trong các chương trình cũ hơn). Việc triển khai mặc định của các chức năng là khá cẩu thả và được cung cấp ở đây chủ yếu để minh họa ngữ nghĩa dự định.

**21.2.1.1 Phê bình**

Trong thực tế, chúng ta khó có thể tìm thấy một sơ đồ tiền tố hai chữ cái đơn giản, mạch lạc. Có nhiều khả năng hơn, các thư viện từ các đơn vị cung cấp khác nhau sẽ ở các không gian tên khác nhau và sử dụng các thuật ngữ khác nhau cho các khái niệm tương tự.

**21.2.2Kế thừa giao diện**

[1] Hệ thống giao diện người dùng phải là một chi tiết triển khai được ẩn với những người dùng không muốn biết về nó.  
[2] Lớp Ival\_box không được chứa dữ liệu.  
[3] Không cần biên dịch lại mã sử dụng họ lớp Ival\_box sau khi thay đổi hệ thống giao diện người dùng.  
[4] Ival\_boxes cho các hệ thống giao diện khác nhau sẽ có thể cùng tồn tại trong chương trình của chúng tôi.

**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. 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. Sự cải tiến này trở nên đáng kể khi - không có gì lạ - những phần tóm tắt của chúng tôi không quá khác biệt so với những phần được cung cấp bởi hệ thống được sử dụng để triển khai.

**21.2.3.1 Phê bình**

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 kế thừa thực thi

**21.2.4 Bản địa hóa Tạo đối tượng**

Hầu hết một ứng dụng có thể được viết bằng cách sử dụng Ival\_box giao diện. Hơn nữa, nếu các giao diện có nguồn gốc phát triển để cung cấp nhiều tiện ích hơn so vớiIval\_box, thì hầu hết một ứng dụng có thể được viết bằng cách sử dụng Ival\_box, Ival\_slider, v.v., giao diện. Tuy nhiên, việc tạo các đối tượng phải được thực hiện bằng cách sử dụng các tên dành riêng cho việc triển khai nhưCW\_ival\_dial và BB\_flashing\_ival\_slider. Đối với mỗi giao diện từ Ival\_box họ lớp mà người dùng nên biết, lớpIval\_maker cung cấp một hàm tạo một đối tượng. Một lớp như vậy đôi khi được gọi là *xưởng sản xuất* và các chức năng của nó (hơi gây hiểu lầm) đôi khi được gọi là *các nhà xây dựng ảo.*

**21.3 Thừa kế nhiều**

*Giao diện được chia sẻ*: dẫn đến việc sao chép mã ít hơn bằng cách sử dụng các lớp và làm cho mã đóđồng nhất hơn. Điều này thường được gọi là*đa hình thời gian chạy* hoặc *kế thừa giao diện*.

*Triển khai được chia sẻ*: dẫn đến ít mã hơn và mã triển khai thống nhất hơn. Điều này thường được gọi là*kế thừa thực hiện*

**21.3.1 Nhiều giao diện**

**21.3.2Nhiều lớp triển khai**

Việc sử dụng đa kế thừa để '' gắn kết '' hai lớp không liên quan với nhau như một phần của việc triển khai lớp thứ ba là thô thiển, hiệu quả và tương đối quan trọng, nhưng không thú vị lắm. Về cơ bản, nó giúp lập trình viên không phải viết nhiều hàm chuyển tiếp (để bù đắp cho thực tế là chúng ta chỉ có thể ghi đè các hàm được định nghĩa trong base). Kỹ thuật này không ảnh hưởng đáng kể đến thiết kế tổng thể của một chương trình và đôi khi có thể mâu thuẫn với mong muốn giữ các chi tiết triển khai được ẩn. Tuy nhiên, một kỹ thuật không nhất thiết phải thông minh để trở nên hữu ích.

**21.3.3 Độ phân giải mơ hồ**

Hai lớp cơ sở có thể có:

class Satellite {  
public:virtual Debug\_info get\_debug(); *...*

};

class Displayed {

public:

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

};

Một hàm được khai báo trong một lớp dẫn xuất sẽ ghi đè *tất cả các* các hàm cùng tên và nhập vào các lớp cơ sở của nó. Thông thường, đó chính xác là điều phải làm vì nói chung là một ý tưởng tồi nếu sử dụng cùng một tên cho các phép toán có ngữ nghĩa khác nhau trong một lớp duy nhất. Lý tưởng cho virtual là cho một lệnh gọi để có tác dụng tương tự độc lập với giao diện đã được sử dụng để tìm hàm.

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

Khi mỗi lớp chỉ có một lớp cơ sở trực tiếp, cấu trúc phân cấp lớp sẽ là một cây và một lớp chỉ có thể xuất hiện một lần trong cây. Khi một lớp có thể có nhiều lớp cơ sở, một lớp có thể xuất hiện nhiều lần trong cấu trúc phân cấp kết quả. Một lớp hữu ích như vậy đương nhiên sẽ được sử dụng ở một số nơi trong hệ thống phân cấp lớp. Một hàm ảo của một lớp cơ sở được sao chép có thể được ghi đè bởi một hàm (đơn) trong một lớp dẫn xuất.

**21.3.5 Lớp cơ sở ảo**

Trong đồ thị kế thừa, mọi lớp cơ sở của một tên nhất định được chỉ định là ảo sẽ được biểu diễn bằng một đối tượng duy nhất của lớp đó. Mặt khác, mỗi lớp cơ sở không được chỉ định ảo sẽ có subobject riêng đại diện cho nó.

Tại sao ai đó muốn sử dụng cơ sở ảo chứa dữ liệu? Tôi có thể nghĩ ra ba cách rõ ràng để hai lớp trong hệ thống phân cấp lớp chia sẻ dữ liệu:

[1] Làm cho dữ liệu 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.

**21.3.5.1 Xây dựng cơ sở ảo**

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

struct V {

V(int i); // ... };

struct A {

A(); // ... // default constructor };

struct B : virtual V, virtual A {

B() :V{1} { /\* ... \*/ }; // default constructor ; must initialize base V // ...};

class C : virtual V {

public: C(int i) : V{i} { /\* ... \*/ }; // must initialize base V // ...};

class D : virtual public B, virtual public C { // implicitly gets the virtual base V from B and C // implicitly gets virtual base A from B

public: D() { /\* ... \*/ } // error : no default constructor for C or V

D(int i) :C{i} { /\* ... \*/ }; // error : no default constructor for V

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

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

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

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

Khi sử dụng một lớp trừu tượng (không có bất kỳ dữ liệu được chia sẻ nào) làm giao diện, chúng ta có một lựa chọn:

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

- Tạo lớp giao diện ảo để chia sẻ một đối tượng đơn giản giữa tất cả các lớp trong hệ thống phân

Mặt khác, có thể xây dựng các tình huống hợp lý trong đó việc chia sẻ ngụ ý trong thiết kế cơ sở ảo gây ra sự mơ hồ cho các phôi *từ* lớp cơ sở Tuy nhiên, những sự mơ hồ như vậy rất dễ bị xử lý.

**21.3.6.1 Ghi đè các hàm cơ sở ảo**

Một lớp dẫn xuất có thể ghi đè một hàm ảo của lớp cơ sở ảo trực tiếp hoặc gián tiếp của nó. Đặc biệt, hai lớp khác nhau có thể ghi đè các chức năng ảo khác nhau từ cơ sở ảo. Theo cách đó, một số lớp dẫn xuất có thể đóng góp các triển khai cho giao diện được trình bày bởi một lớp cơ sở ảo. Nếu hai lớp ghi đè một hàm của lớp cơ sở, nhưng không ghi đè lên lớp kia, thì cấu trúc phân cấp lớp là một lỗi. Lý do là không một hàm đơn lẻ nào có thể được sử dụng để cung cấp một ý nghĩa nhất quán cho tất cả các lệnh gọi một cách độc lập với lớp mà chúng sử dụng làm giao diện. Hoặc, bằng cách sử dụng thuật ngữ triển khai, không có bảng hàm ảo nào có thể được xây dựng vì lệnh gọi hàm đó trên đối tượng hoàn chỉnh sẽ không rõ ràng. Một lớp cung cấp một số - nhưng không phải tất cả - việc triển khai cho một lớp cơ sở ảo thường được gọi là *mixin*.