**16**

**Lớp**

**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**

**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.