Nguyễn Thành Liêm MSSV : 6151071065

**CHUYÊN ĐỀ : DỊCH SÁCH C++**

**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.

ký hiệu. Xem xét:

class complex { // very simplifie complex

double re, im;

public:

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

complex operator+(complex);

complex operator∗(complex);

};

Điều này xác định một cách triển khai đơn giản của khái niệm số phức. Một phức hợp được gửi đi bởi một cặp số dấu phẩy động có độ chính xác kép được thao tác bởi các toán tử + và ∗.Lập trình viên định nghĩa phức hợp :: operator + () và complex :: operator ∗ () để cung cấp ý nghĩa cho +và ∗, tương ứng. Ví dụ: nếu b và c thuộc loại phức tạp, b + c có nghĩa là b. Điều hành + (c). Chúng ta có thể bây giờ gần đúng với cách giải thích thông thường của các biểu thức phức tạp:

void f() {

complex a = complex{1,3.1};

complex b {1.2, 2};

complex c {b};

a = b+c;

b = b+c∗a;

c=a∗b+complex(1,2);

}

Các quy tắc ưu tiên thông thường được giữ nguyên, vì vậy câu lệnh thứ hai có nghĩa là b = b + (c ∗ a), không phải b = (b + c) ∗ a.Lưu ý rằng ngữ pháp C ++ được viết để ký hiệu {} chỉ có thể được sử dụng cho các trình khởi tạo và ở phía bên phải của bài tập:

void g(complex a, complex b) {

a = {1,2}; // OK: right hand side of assignment

a += {1,2}; // OK: right hand side of assignment

b = a+{1,2}; // lỗi cú pháp

b = a+complex{1,2}; // OK

g(a,{1,2}); // OK :một đối số hàm được coi là một trình khởi tạo

{a,b} = {b,a}; // lỗi cú pháp

}

Dường như không có lý do cơ bản nào để không sử dụng {} ở nhiều nơi hơn, nhưng các vấn đề kỹ thuật của viết một ngữ pháp cho phép {} ev ery ở đâu trong một biểu thức (ví dụ: làm thế nào bạn biết nếu một {sau một dấu chấm phẩy là phần bắt đầu của một biểu thức hay một khối?) và cũng đưa ra các thông báo lỗi tốt dẫn đến một hạn chế hơn sử dụng {} trong biểu thức

Nhiều ứng dụng rõ ràng nhất của việc nạp chồng toán tử là đối với các kiểu số. Tuy nhiên,

tính hữu ích của các toán tử do người dùng xác định không bị giới hạn đối với các kiểu số. Ví dụ, thiết kế của

các giao diện chung và trừu tượng thường dẫn đến việc sử dụng các toán tử như ->, [] và ()

**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).

Không thể xác định mã thông báo toán tử mới, nhưng bạn có thể sử dụng ký hiệu lệnh gọi hàm khi tập hợp các toán tử này là không đầy đủ. Ví dụ, sử dụng **pow ()**, không phải ∗∗. Những hạn chế này có vẻ

Các quy tắc hà khắc, nhưng linh hoạt hơn có thể dễ dẫn đến sự mơ hồ. Ví dụ, xác định một toán hạng tor ∗∗ có nghĩa là lũy thừa có vẻ là một nhiệm vụ hiển nhiên và dễ dàng, nhưng hãy nghĩ lại. Tôi có nên ràng buộc

ở bên trái (như ở Fortran) hay bên phải (như ở Algol)? Biểu thức **a ∗∗** **p** có được hiểu là

**a ∗ (∗ p)** hay là **(a) ∗∗ (p)**? Có giải pháp cho tất cả các câu hỏi kỹ thuật như vậy. Tuy nhiên, điều đáng chú ý nhất là nếu áp dụng các quy tắc kỹ thuật tinh tế sẽ dẫn đến mã dễ đọc và dễ bảo trì hơn. Nếu trong

nghi ngờ, sử dụng một chức năng được đặt tên.

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.

**18.2.4 Chuyển đối tượng**

Khi chúng tôi xác định một toán tử, chúng tôi thường muốn cung cấp một ký hiệu thông thường, ví dụ:

**a = b + c.**

Do đó, chúng tôi có các lựa chọn hạn chế về cách truyền các đối số cho hàm toán tử và cách nó trả về giá trị của nó. Ví dụ: chúng tôi không thể yêu cầu các đối số con trỏ và yêu cầu các nhà soạn thảo chuyên nghiệp sử dụng toán tử address-of hoặc trả về một con trỏ và mong đợi người dùng bỏ qua nó:

**∗ a = & b + & c** không được chấp nhận.

Đối với các đối số, chúng ta có hai lựa chọn chính (§12.2):

• Giá trị chuyển qua

• Tham chiếu chuyển qua

Đối với các đối tượng nhỏ, chẳng hạn như một đến bốn từ, gọi theo giá trị thường là một sự thay thế khả thi và thường là một thứ mang lại hiệu suất tốt nhất. Tuy nhiên, hiệu suất của việc truyền và sử dụng đối số phụ thuộc vào về kiến ​​trúc máy, quy ước giao diện trình biên dịch (Giao diện nhị phân ứng dụng; ABIs) và số lần một đối số được truy cập (hầu như luôn luôn nhanh hơn để truy cập một đối số được truyền bởi giá trị hơn một giá trị được truyền bởi tham chiếu). Ví dụ: giả sử rằng một **point** được biểu diễn dưới dạng cặp **int**:

void Point::operator+=(Point delta); // giá trị chuyển qua

Các đối tượng lớn hơn, chúng tôi chuyển qua tham chiếu. Ví dụ, bởi vì một Ma trận (một ma trận đơn giản của nhân đôi; §17.5.1) rất có thể lớn hơn một vài từ, chúng tôi sử dụng tham chiếu chuyển qua:

Matrix operator+(const Matrix&, const Matrix&); // tham chiếu chuyển qua

Đặc biệt, chúng tôi sử dụng tham chiếu **const** để chuyển các đối tượng lớn không được sửa đổi bởi được gọi là hàm (§12.2.1).

Thông thường, một toán tử trả về một kết quả. Trả lại một con trỏ hoặc một tham chiếu đến một tham chiếu mới được tạo đối tượng thường là một ý tưởng rất tồi: sử dụng một con trỏ đưa ra các vấn đề về ký hiệu và tham chiếu đến một đối tượng trên cửa hàng miễn phí (cho dù bằng con trỏ hay bằng tham chiếu) dẫn đến quản lý bộ nhớ các vấn đề. Thay vào đó, hãy trả về các đối tượng theo giá trị. Đối với các đối tượng lớn, chẳng hạn như **Ma trận**, hãy xác định các toán hạng di chuyển để thực hiện việc chuyển các giá trị như vậy một cách hiệu quả (§3.3.2, §17.5.2). Ví dụ:

Matrix operator+(const Matrix& a, const Matrix& b) // return-by-value

{

Matrix res {a};

return res+=b;

}

Lưu ý rằng các toán tử trả về một trong các đối tượng đối số của chúng có thể - và thường là - trả về một tham chiếu. Ví dụ: chúng ta có thể xác định toán tử **+ =** của **Ma trận** như thế này:

Matrix& Matrix::operator+=(const Matrix& a) // return-by-reference

{

if (dim[0]!=a.dim[0] || dim[1]!=a.dim[1])

throw std::exception("bad Matrix += argument");

double∗ p = elem;

double∗ q = a.elem;

double∗ end = p+dim[0]∗dim[1];

while(p!=end)

∗p++ += ∗q++

return ∗this;

}

Điều này đặc biệt phổ biến đối với các hàm toán tử được triển khai với tư cách là thành viên.

Nếu một hàm chỉ đơn giản là chuyển một đối tượng cho một hàm khác, thì một đối số tham chiếu rvalue sẽ được sử dụng (§17.4.3, §23.5.2.1, §28.6.3).

**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.

**18.3.2 Số học chế độ hỗn hợp**

Để đối phó với **2 + z,** trong đó **z** là **một phức**, chúng ta cần xác định toán tử + để chấp nhận các toán hạng khác nhau các loại. Theo thuật ngữ Fortran, chúng ta cần số học ở chế độ hỗn hợp. Chúng ta có thể đạt được điều đó đơn giản bằng cách thêm các phiên bản thích hợp của các toán tử:

class complex {

double re, im;

public:

complex& operator+=(complex a)

{

re += a.re;

im += a.im;

return ∗this;

}

complex& operator+=(double a)

{

re += a;

return ∗this;

}

// ...

};

Ba biến thể của **toán tử + ()** có thể được định nghĩa bên ngoài **complex**:

complex operator+(complex a, complex b)

{

return a += b; // gọi complex::operator+=(complex)

}

complex operator+(complex a, double b)

{

return {a.real()+b,a.imag()};

}

complex operator+(double a, complex b)

{

return {a+b.real(),b.imag()};

}

Các hàm truy cập **real ()** và **virtual ()** được định nghĩa trong §18.3.6.

Với các khai báo **+** này, chúng ta có thể viết:

void f(complex x, complex y)

{

auto r1 = x+y; // calls operator+(complex,complex)

auto r2 = x+2; // calls operator+(complex,double)

auto r3 = 2+x; // calls operator+(double,complex)

auto r4 = 2+3; // phép cộng số nguyên cài sẵn

}

Tôi đã thêm phép cộng số nguyên cho đầy đủ.

**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)

}

Có thể có lý do để thích xác định các chức năng riêng biệt. Ví dụ, trong một số trường hợp,

Khi một số biến thể của một hàm hoặc một toán tử tồn tại, trình biên dịch phải chọn '' đúng '' biến thể dựa trên các loại đối số và các chuyển đổi có sẵn (tiêu chuẩn và do người dùng xác định).

Trừ khi một kết quả phù hợp nhất tồn tại, một biểu thức không rõ ràng và là một lỗi (xem §12.3).

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.

Có thể đi xa hơn và giới thiệu một ký tự do người dùng xác định (§19.2.6) để hỗ trợ loại **com-plex** của chúng tôi. Đặc biệt, chúng ta có thể định nghĩa i là một hậu tố có nghĩa là '' phần ảo ''. Ví dụ:

constexpr complex <double> operator "" i (long double d) // chữ phần ảo

{

return {0, d}; // complex là một kiểu 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.

Nói chung, điều khôn ngoan là nên tiết kiệm trong việc giới thiệu các nhà khai thác chuyển đổi. Khi được sử dụng quá mức, chúng sẽ dẫn đến sự mơ hồ. Những sự mơ hồ như vậy được trình biên dịch bắt gặp, nhưng chúng có thể gây phiền toái để giải quyết. Có lẽ ý tưởng tốt nhất ban đầu là thực hiện chuyển đổi bằng các hàm được đặt tên, chẳng hạn như **X :: make\_int ().**

Nếu một hàm như vậy trở nên đủ phổ biến để làm cho việc sử dụng rõ ràng không phù hợp, nó có thể được thay thế bằng một toán tử chuyển đổi **X :: operator int ().**

int operator+(Tiny,Tiny);

void f(Tiny t, int i)

{

t+i; // error, không rỏ rang ‘‘operator+(t,Tiny(i))’’ or ‘‘int(t)+i’’?

}

Do đó, tốt nhất là nên dựa vào chuyển đổi do người dùng xác định hoặc toán tử do người dùng xác định cho một loại 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).

**18.4.3 Tính không rõ ràng**

Việc gán giá trị kiểu **V** cho một đối tượng thuộc lớp **X** là hợp pháp nếu có toán tử gán

**X :: operator = (Z)** sao cho **V** là **Z** hoặc có một chuyển đổi duy nhất của **V** thành **Z.** Quá trình khởi tạo được xử lý tương đương.

Trong một số trường hợp, giá trị của kiểu mong muốn có thể được xây dựng bằng cách sử dụng lặp lại các hàm tạo hoặc toán tử chuyển đổi. Điều này phải được xử lý bằng các chuyển đổi rõ ràng; chỉ một cấp độ do người dùng xác định chuyển đổi ngầm là hợp pháp. Trong một số trường hợp, giá trị của kiểu mong muốn có thể được xây dựng trong nhiều hơn một cách; những trường hợp như vậy là bất hợp pháp. Ví dụ:

class X { /\* ... \*/ X(int); X(const char∗); };

class Y { /\* ... \*/ Y(int); };

class Z { /\* ... \*/ Z(X); };

X f(X);

Y f(Y);

Z g(Z);

void k1()

{

f(1); // error : ambiguous f(X(1)) or f(Y(1))?

f(X{1}); // OK

f(Y{1}); // OK

g("Mack"); // error : two user-defined conversions needed; g(Z{X{"Mack"}}) not tried

g(X{"Doc"}); // OK: g(Z{X{"Doc"}})

g(Z{"Suzy"}); // OK: g(Z{X{"Suzy"}})

}

Các chuyển đổi do người dùng xác định chỉ được coi là nếu không thể giải quyết cuộc gọi mà không có chúng (tức là sử dụng chỉ các chuyển đổi được tạo sẵn). Ví dụ:

class XX { /\* ... \*/ XX(int); };

void h(double);

void h(XX);

void k2()

{

h(1); // h(double{1}) or h(XX{1})? h(double{1})!

}

Gọi h (1) có nghĩa là **h (double (1))** vì phương án đó chỉ sử dụng một chuyển đổi tiêu chuẩn chứ không phải một chuyển đổi do người dùng xác định (§12.3).

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.

**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 &);

};

Một **Assoc** giữ một vectơ gồm các cặp **std ::pair**. Việc triển khai sử dụng cùng một cách tầm thường và không hiệu quả phương pháp tìm kiếm như trong §7.7:

int & Assoc :: operator [] (const string & s)

// tìm kiếm s; trả về một tham chiếu đến giá trị của nó nếu được tìm thấy;

// nếu không, hãy tạo một cặp {s, 0} mới và trả về một tham chiếu đến giá trị của nó

{

for (auto x: vec)

if (s == x.first) return x.second;

vec.push\_back ({s, 0}); // giá trị ban đầu: 0

return vec.back ().second; // trả về phần tử cuối cùng (§31.2.2)

}

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. Việc ghi đè toán tử cuộc gọi hàm dường như hữu ích chủ yếu để xác định các loại chỉ có một thao tác duy nhất và dành cho các loại mà một thao tác chiếm ưu thế.Gọi toán tử còn được gọi là toán tử ứng dụng.

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

};

Một đối tượng của lớp Add được khởi tạo bằng một số phức và khi được gọi bằng cách sử dụng (), nó sẽ thêm vào số cho đối số của nó. Ví dụ:

void h (vector <complex> & vec, list <complex> & lst, complex z)

{

for\_each (vec.begin (), vec.end (), Add {2,3});

for\_each (lst.begin (), lst.end (), Add{z});

}

Điều này sẽ thêm phức {2,3} vào mọi phần tử của vectơ và z vào mọi phần tử của danh sách. Ghi chú rằng Thêm {z} xây dựng một đối tượng được for\_each () sử dụng lặp đi lặp lại: Toán tử Thêm {z} () () được gọi là cho mỗi phần tử của dãy.

Tất cả điều này đều hoạt động vì for\_each là một mẫu áp dụng () cho đối số thứ ba của nó mà không cần quan tâm chính xác đối số thứ ba đó thực sự là gì:

template <typename Iter, typename Fct>

Fct for\_each (Iter b, Iter e, Fct f)

{

while (b! = e) f (∗ b ++);

treturn f;

}

Thoạt nhìn, kỹ thuật này có vẻ bí truyền, nhưng nó rất đơn giản, hiệu quả và cực kỳ hữu ích

(§3.4.3, §33.4).

**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 Phân bổ và Giao dịch**

**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 Trình bày**

Biểu diễn cho Chuỗi được chọn để đáp ứng ba mục tiêu:

• Để dễ dàng chuyển đổi chuỗi kiểu C (ví dụ: chuỗi ký tự) thành Chuỗi và cho phép dễ dàng

truy cập vào các ký tự của một chuỗi dưới dạng một chuỗi kiểu C

• Để giảm thiểu việc sử dụng cửa hàng miễn phí

• Để thêm các ký tự vào cuối một Chuỗi hiệu quả

Kết quả rõ ràng là phức tạp hơn một biểu diễn {pointer, size} đơn giản, nhưng thực tế hơn nhiều:

Class String {

/ \*

Một chuỗi đơn giản thực hiện tối ưu hóa string ngắn

size () == sz là số phần tử

if size () <= short\_max, các ký tự được giữ trong chính đối tượng String;

nếu không thì cửa hàng miễn phí được sử dụng.

ptr trỏ đến đầu chuỗi ký tự

dãy ký tự được giữ nguyên bằng không: ptr [size ()] == 0;

điều này cho phép chúng ta sử dụng các hàm C libraryry string và dễ dàng trả về một chuỗi kiểu C: c\_str ()

Để cho phép bổ sung hiệu quả các ký tự vào cuối, Chuỗi phát triển bằng cách tăng gấp đôi phân bổ của nó;

dung lượng () là lượng không gian có sẵn cho các ký tự

(không bao gồm số 0 kết thúc): sz + dấu cách

\* /

public:

// ...

private:

static const int short\_max = 15;

int sz; // số ký tự

char ∗ ptr;

unlion {

int space; // không gian được cấp phát chưa sử dụng

char ch [shor t\_max + 1]; // để lại khoảng trống để kết thúc 0

};

void check (int n) const // kiểm tra phạm vi

{

if (n <0 || sz <= n)

throw std :: out\_of\_range ("String :: at ()");

}

// hàm thành viên ancillary:

void copy\_from (const String & x);

void move\_from (string & x);

};

Điều này hỗ trợ những gì được gọi là tối ưu hóa chuỗi ngắn bằng cách sử dụng hai biểu diễn chuỗi:

• Nếu sz <= short\_max, các ký tự được lưu trữ trong chính đối tượng String, trong mảng có tên ch.

• Nếu! (Sz <= short\_max), các ký tự được lưu trữ trên cửa hàng miễn phí và chúng tôi có thể phân bổ them không gian để mở rộng. Thành viên được đặt tên khoảng trắng là số ký tự như vậy.

Trong cả hai trường hợp, số lượng phần tử được giữ trong sz và chúng tôi xem xét sz, để xác định lược đồ ẩn ý nào được sử dụng cho một chuỗi nhất định.

Trong cả hai trường hợp, ptr trỏ đến các phần tử. Điều này là cần thiết cho hiệu suất: các chức năng truy cập

không cần phải kiểm tra xem biểu diễn nào được sử dụng; họ chỉ đơn giản là sử dụng ptr. Chỉ các hàm tạo, gán ments, di chuyển và hủy (§19.3.4) mới phải quan tâm đến hai lựa chọn thay thế.

Chúng ta chỉ sử dụng mảng ch khi sz <= short\_max và không gian số nguyên chỉ khi! (Sz <= short\_max). Theo trình tự, sẽ thật lãng phí nếu phân bổ không gian cho cả ch và khoảng trắng trong một đối tượng Chuỗi. Tránh lãng phí như vậy, tôi sử dụng một công đoàn (§8.3). Đặc biệt, tôi đã sử dụng một hình thức công đoàn được gọi là công đoàn ẩn danh

(§8.3.2), được thiết kế đặc biệt để cho phép một lớp quản lý các đại diện thay thế của

các đối tượng. Tất cả các thành viên của một liên minh ẩn danh được phân bổ trong cùng một bộ nhớ, bắt đầu từ cùng địa chỉ. Chỉ một thành viên có thể được sử dụng bất kỳ lúc nào, nhưng nếu không, họ sẽ được truy cập và được sử dụng chính xác như thể chúng là các thành viên riêng biệt của phạm vi xung quanh liên minh ẩn danh. Nó là công việc của lập trình viên là đảm bảo rằng chúng không bao giờ bị lạm dụng. Ví dụ, tất cả các chức năng thành viên của Chuỗi sử dụng không gian phải đảm bảo rằng đó thực sự là không gian đã được thiết lập và không phải ch. Đó là được thực hiện bằng cách xem sz <= short\_max. Nói cách khác, Hình dạng (trong số những thứ khác) là một hợp nhất với sz <= short\_max làm phân biệt

**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*