Chapter 18

18.1 Introduction (giới thiệu)

Trong lĩnh vực kỹ thuật-và lĩnh vực phi kỹ thuật, thì việc sử dụng các ký hiệu viết tắc sẽ thuận tiện cho việc trình bày và thảo luận khi các khái niệm này được lặp đi lặp lại nhiều lần.

Ví dụ:

Viết tắt: x+y*z

Đầy đủ: lấy y nhân z và sau đó lấy kết quả cộng với x

Rất khó để đánh giá được tầm quan trọng của các ký hiệu viết tắt trong một bài 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 ý tưởng 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 theo ý muốn của người dùng.

Ví dụ: nếu bạn muốn giải quyết các bài toán phức tạp, ma trận,chuỗi ký tự trong C++. Bạn sẽ tạo ra các lớp để biểu diễn chúng. Và giới hạn các toán tử cần dùng đối với mỗi lớp như vậy sẽ giúp lập trình viên thiết lập những ký hiệu đơn giản để dễ dàng trong việc sử dụng vào trong bài toán khi chỉ cần thực hiện vài thao tác đơn giản.

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

Điều này giúp triển khai đầy đủ những khái niệm cơ bản của bài toán. Dùng một cặp dấu đại điện cho các toán tử + và *

Người dùng định nghĩa như sau: complex::operator +() và complex::operator *() để biểu thị cho phép + và * tương ứng.

Ví dụ: nếu b và c là kiểu complex thì b+c sẽ dược viết b.operator+(c). Chúng ta có thể xem như gần đúng với yêu cầu bài toán

```
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 nó:

```
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}; // syntax error
b = a+complex{1,2}; // OK
g(a,{1,2}); // OK: a function argument is considered an initializer
{a,b} = {b,a}; // syntax error
}
```

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 {} ở mọi nơi 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 lệnh) và cũng đưa ra các thông báo lỗi dẫn đến một hạn chế hơn sử dụng {} trong biểu thức.

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 (toán tử hàm)

Các hàm định nghĩa cho các toán tử được khai bao như sau:

+ − ∗ / %^ &

|˜ ! = < > +=

−= ∗= /= %= ^= &= |=

<< >> >>= <<= == != <=

>= && || ++ −− −>∗ ,

−> [] () new new[] delete delete[]

Người dùng không thể khai báo các toán tử sau

::

.

.*

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 nạp chồng ‘‘ toán tử ’’ vì chúng báo cáo thông tin cơ bản về hoạt động chọn của chúng:

sizeof

alignof

typeid type_info

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

?:

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

Không thể xác định mã thông báo toán tử mới, nhưng bạn có thể sử dụng ký hiệu lệnh gọi hàm khi nhóm các toán tử này là không đầy đủ. Ví dụ, sử dụng pow (), không phải ∗∗. 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 ∗∗ 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, đó là điều tối kỵ 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 toán tử 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; // shor thand
complex d = a.operator+(b); // explicit call
}
```

Với định nghĩa trước đây về phức, hai bộ khởi tạo là đồng nghĩa.

18.2.1 Binary and Unary Operators (toán tử 1 và 2 ngôi)

Toán tử 2 ngôi có thể là hàm thành viên của class và có 1 đối số hoặc không phải là hàm thành viên và sẽ có 2 đối số. Đối với các toán tử 2 ngôi @,aa@bb có thể biểu diễn aa.operator @ (bb) hoặc operator @(aa,bb).

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 không phải hàm thành viên và 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). Đối với bất kỳ toán tử đơn vị hậu tố nào @, aa @ có thể được hiểu là aa.operator @ (int) hoặc operator @ (aa, int). 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% bậc 1 hoặc bậc 3 +.

Ví dụ:

```
class X {
public: // members (with implicit this pointer):
X∗ operator&(); // prefix unary & (address of)
X operator&(X); // binar y & (and)
X operator++(int); // postfix increment (see §19.2.4)
X operator&(X,X); // error : ter nary
X operator/(); // error : unar y /
};
```

Section 18.2.1 Binary and Unary Operators 531

```
// nonmember functions :
X operator−(X); // prefix unary minus
X operator−(X,X); // binar y minus
X operator−−(X&,int); // postfix decrement
```

```
X operator−(); // error : no operand

X operator−(X,X,X); // error : ter nary

X operator%(X);
```

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 toán tử = (§18.2.2), toán tử [] (§19.2.1), toán tử () (§19.2.2) và toán tử−>(§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 lýchính xác như các toán tử 2 ngôi khác.

18.2.2 Predefined Meanings for Operators

Ý nghĩa của một số toán tử cài sẵn được định nghĩa tương đương với một số kết hợp của các toán tử khác toán tử trên các đối số giống nhau. Ví dụ: nếu a là int, ++ a có nghĩa là a + = 1, điều này có nghĩa là a = a + 1. Các quan hệ như vậy không giữ cho các toán tử do người dùng xác định trừ khi người dùng định nghĩa chúng. Vì ví dụ, một trình biên dịch sẽ không tạo ra định nghĩa của Z :: operator + = () từ các định nghĩa của Z :: operator + () và Z :: operator = ().

Các toán tử = (gán), & (tham chiếu) 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 Operators and User-Defined Types

Một hàm toán tử phải là một thành viên hoặc có ít nhất một đối số của kiểu do người dùng xác định

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 toán hạng. 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). 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.

Ví dụ:

```
 enum Day { sun, mon, tue, wed, thu, fri, sat };
Day& operator++(Day& d)
{
return d = (sat==d) ? sun : static_cast<Day>(d+1);
}
```

18.2.4 Passing Objects

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

- Truyền theo tham trị
- Truyền theo tham chiếu (&)

Đối với đối tượng nhỏ mang giá trị từ 1-4 thì dùng tham trị sẽ đạt hiệu quả tối ưu. 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; ABI) và số lần một đối số

được truy cập (hầu như luôn luôn nhanh hơn để truy cập một đối số được truyền bởi tham trị hơn một giá trị được truyền bởi tham chiếu).

Ví dụ:

```
void Point::operator+=(Point delta);    // pass-by-value

Matrix operator+(const Matrix&, const Matrix&); // pass-by-const-reference
```

Đặc biệt, chúng tôi sử dụng tham chiếu const để chuyển các đối tượng lớn không được sửa đổi bởi

được gọi là hàm. Thông thường, một toán tử trả về một kết quả, trả về các đối tượng theo giá trị hoặc không trả về giá trị (void)

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

{

Matrix res {a};

return res+=b;

}
```

Ví dụ trả về chính đối tượng:

```
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;

}
```

18.2.5 Operators in Namespaces

```
namespace std { // simplified std
```

```
class string {
// ...
};
class ostream {
// ...
ostream& operator<<(const char*); // output C-style string
};
extern ostream cout;
ostream& operator<<(ostream&, const string&); // output std::string
} // namespace std
int main()
{
const char* p = "Hello";
std::string s = "world";
std::cout << p << ", " << s << "!\n";
}
```

Lưu ý rằng tôi không làm cho mọi thứ từ std có thể truy cập được 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
```

std::cout.operator<<(p)

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

std::cout << s

```
operator<<(std::cout,s)

std::operator<<(std::ostream&, const std::string&)
```

Các hàm khởi tạo:

```
X operator!(X);

struct Z {

Z operator!(); // does not hide ::operator!()

X f(X x) { /* ... */ return !x; } // invoke ::operator!(X)

int f(int x) { /* ... */ return !x; } // invoke the built-in ! for ints

};
```

Đặc biệt, thư viện iostream tiêu chuẩn định nghĩa << hàm thành viên để xuất ra các kiểu tích hợp sẵn, và

người dùng có thể định nghĩa << để xuất ra các kiểu do người dùng định nghĩa mà không cần sửa đổi lớp ostream (§38.4.2).

18.3 A Complex Number Type

```
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 ().

18.3.1 Member and Nonmember Operators

Tôi muốn giảm thiểu số lượng hàm thao tác trực tiếp với biểu diễn của một đối tượng. Điều này có thể đạt được bằng cách chỉ định nghĩa các toán tử vốn đã sửa đổi giá trị của đối số đầu tiên của chúng, chẳng hạn như + =, trong chính lớp đó. Các toán tử chỉ đơn

giản tạo ra một giá trị mới dựa trên các giá trị của các đối số của chúng, chẳng hạn như +, sau đó được định nghĩa bên ngoài lớp và sử dụng các toán tử thiết yếu trong việc triển khai chúng:

```
class complex {
double re, im;
public:
complex& operator+=(complex a); // needs access to representation
// ...
};
complex operator+(complex a, complex b)
{
return a += b; // access representation through +=
}
```

The arguments to this operator+() are passed by value, so a+b does not modify its operands.

Given these declarations, we can write:

```
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 tổng hợp như + = và ∗ = có xu hướng dễ xác định hơn so với các toán tử " đơn giản " của chúng + và ∗. Điều này làm cho 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 nhu cầu về các biến tạm thời. Ví dụ:

```
inline complex& complex::operator+=(complex a)
        {
        re += a.re;
        im += a.im;
        return *this;
        }
```

18.3.2 Mixed-Mode Arithmetic

```
        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;
        }
        // ...
        };
```

The three variants of operator+() can be defined outside complex:

```
        complex operator+(complex a, complex b)
        {
        return a += b; // calls 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; // built-in integer addition
}
```