**C.18**

**Operator OverLoading**

**18.1 Introduction**

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. Ví dụ, vì quen biết lâu.

x+y∗z

multiply y by z and add the result to x

Giống như hầu hết các ngôn ngữ khác, C ++ hỗ trợ một tập hợp các toán tử cho các kiểu tích hợp của nó. Tuy nhiên, hầu hết các khái niệm mà các toán tử được sử dụng thông thường không phải là các kiểu tích hợp sẵn trong C ++vì vậy chúng phải được biểu diễn dưới dạng các kiểu do người dùng xác định. Ví dụ:

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 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 complex :: 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}; // 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

}

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

. lựa chọn thành viên

. ∗ lựa chọn thành viên thông qua con trỏ đến thành viên

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.

Không thể quá tải ‘‘ toán tử ’’ được đặt tên vì chúng báo cáo thông tin cơ bản về xóa op của chúng:

Kích thước:

**sizeof** kích thước của đối tượng

**alignof** sự liên kết của đối tượng

**typeid** nhập thông tin của một đối tượng

Cuối cùng, toán tử biểu thức điều kiện bậc ba không thể được nạp chồng

**?:** conditional evaluation

Ngoài ra, các ký tự do người dùng định nghĩa được xác định bằng cách sử dụng ký hiệu **Operator** **""**.

Tên của một hàm toán tử là **toán tử** từ khóa được theo sau bởi chính toán tử đó, cho ví dụ, toán tử **<<**. Một hàm toán tử được khai báo và có thể được gọi như bất kỳ hàm nào khác. 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 đồng nghĩa

**18.2.1 Binary and Unary Operators**

Một toán tử nhị phân có thể được xác định bởi một hàm thành viên non-static 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 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)

## }

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

## **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 /**

## **};**

## **// 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); // error : unar y**

Ý nghĩa mặc định của **&&**, **||,** và**, (dấu phẩy)** liên quan đến trình tự: toán hạng đầu tiên được đánh giá trước toán hạng thứ hai (và đối với **&& 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 các toán tử này được xử lý chính xác như các toán tử nhị phân 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 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 ’’;

## **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,()**

## **}**

**18.2.3 Operators and User-Defined Types**

Một hàm toán tử phải là một thành viên hoặc nhận ít nhất một đối số của kiểu do người dùng xác định (các hàm xác định lại toán tử **new** và **delete** không cần).

Một hàm toán tử nhằm chấp nhận một kiểu dựng sẵn 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)**

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);**

## **}**

**18.2.4 Passing Objects**

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

**a = b + c.**

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

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

• Giá trị chuyển tiếp

• Tham khảo qua

Đối với các đối tượng nhỏ, chẳng hạn như từ một đến bốn từ, gọi theo giá trị thường là một phương án thay thế khả thi và thường là phương án mang lại hiệu suất tốt nhất. Tuy nhiên, hiệu suất của việc truyền và sử dụng đối số phụ thuộc vào kiến trúc máy, quy ước giao diện trình biên dịch và số lần một đối số được truy cập. Ví dụ:

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

Các đối tượng lớn hơn,chúng tôi chuyển qua tham chiếu.

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

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 định nghĩa toán tử + = của Matrix như sau:

## **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**

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).

**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";**

### **}**

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

**std::cout << p**

means

**std::cout.operator<<(p)**

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

**std::cout << s**

có nghĩa

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

Hãy xem xét một toán tử nhị phân **@.** Nếu **x** thuộc loại **X** và **y** thuộc loại **Y, x @ y** được giải quyết như sau:

• Nếu **X** là một lớp, hãy tìm **operator@** như một thành viên của X hoặc như một thành viên của một cơ sở của **X**; và

• tìm kiếm các khai báo của **operator @** trong ngữ cảnh xung quanh **x @ y;** và

• nếu **X** được định nghĩa trong không gian tên **N,** hãy tìm các khai báo của **operator @** trong **N;** và

• nếu **Y** được định nghĩa trong không gian tên **M**, hãy tìm các khai báo của **operator @** trong **M.**

Toán hạng của một kiểu do người dùng xác định. Lưu ý rằng trong tra cứu nhà điều hành không có ưu tiên nào dành cho các thành viên hơn là những người không phải là thành viên. Lưu ý rằng trong tra cứu nhà điều hành không có ưu tiên nào dành cho các thành viên hơn là những người không phải là thành viên. Ví dụ:

## **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**

## **};**

**18.3 A Complex Number Type**

Việc triển khai các số phức được trình bày trong §18.1 là quá hạn chế để làm hài lòng bất kỳ ai.

## **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 ta sẽ được cung cấp thêm một số toán tử, chẳng hạn như **==** cho icon 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. Ngoài ra, người sử dụng số học phức tạp phụ thuộc rất nhiều vào các toán tử đến mức định nghĩa của **Complex** đưa vào chơi 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 phức hợp thư viện chuẩn **<double>.**

**18.3.1 Member and Nonmember Operators**

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 +=**

## **}**

Các đối số cho **operator + ()** này được truyền theo giá trị, vì vậy **a + b** không sửa đổi các toán hạng của nó.

## **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)**

## **}**

Các toán tử gán tổng hợp như **+ =** và **∗ =** có xu hướng dễ xác định hơn so với các toán tử '' sim ple '' **+** và **∗** của chúng. 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**

. Chúng tôi 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; // 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()};**

**}**

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**

## **}**

**18.3.3 Conversions**

Để đối phó với các phép gán và khởi tạo các biến **Complex** với đại lượng vô hướng, chúng ta cần chuyển đổi một đại lượng vô hướng (số nguyên hoặc 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**

## **// ... }**

Một hàm tạo nhận một đối số chỉ định một chuyển đổi từ kiểu đối số của nó sang kiểu của hàm tạo:

## **class complex {**

## **double re, im;**

## **public:**

## **complex(double r) :re{r}, im{0} { } // build a complex from a double**

## **// ...**

## **};**

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 gán.

Chúng ta vẫn cần hàm tạo có hai **nhân đôi** và một hàm tạo mặc định khởi tạo **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} { }**

## **// ...**

## **};**

Theo mặc định, việc sao chép các giá trị phức tạp được định nghĩa là sao chép phần thực và phần ảo.Ví dụ:

## **void f() {**

## **complex z;**

## **complex x {1,2};**

## **complex y {x}; // y also has the value {1,2}**

## **z = x; // z also has the value {1,2}**

## **}**

**18.3.3.1 Conversions of Operands**

complex operator+(complex,complex);

complex operator+(complex,double);

complex operator+(double ,complex);

// ...

Ví dụ, lớp ***Complex*** của chúng tôi cung cấp một phương thức khởi tạo để chuyển đổi một ***Double*** thành một ***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; // means operator==(x,y)

## x==3; // means operator==(x,complex(3))

## 3==y; // means operator==(complex(3),y)

## }

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**

## **}**

Vì vậy, bạn có thể gần đúng khái niệm rằng một toán tử yêu cầu một giá trị làm toán hạng bên trái của nó bằng cách đặt toán tử đó thành một thành viên.

## **complex x {4,5}**

## **complex z {sqr t(x)+={1,2}}; // like tmp=sqr t(x), tmp+={1,2}**

**18.3.4 Literals**

class complex {

public:

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

// ...

}

Có thể đi xa hơn và giới thiệu một ký tự do người dùng xác định để hỗ trợ kiểu phức tạp của chúng tôi. Đặc biệt, chúng ta có thể định nghĩa i là một hậu tố có nghĩa là '' tưởng tượng ''. Ví dụ:

## **constexpr complex operator "" i(long double d) // imaginar y literal**

## **{**

## **return {0,d}; // complex is a literal type**

## **}**

Điều này sẽ cho phép chúng ta viết

complex z1 {1.2+12e3i};

## **complex f(double d)**

## **{**

## **auto x {2.3i};**

## **return x+sqrt(d+12e3i)+12e3i;**

## **}**

Ví dụ trên gần tương đương với

## **complex z1 {1.2,12e3};**

## **complex f(double d)**

## **{**

## **complex x {0,2.3};**

## **return x+sqrt(complex{d,12e3})+complex{0,12e3};**

## **}**

**18.3.5 Accessor Functions**

Cho đến nay, chúng tôi đã cung cấp phức hợp lớp chỉ với các hàm tạo và toán tử số học. Điều đó không hoàn toàn đủ để sử dụng thực tế. Đặc biệt, chúng ta thường cần có khả năng kiểm tra và thay đổi giá trị 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; } // ...**

## **};**

Ta 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 là. Đối với nhiều loại, quyền truy cập riêng lẻ (đôi khi được gọi là các hàm get-and-set) là một lời mời dẫn đến thảm họa. Nếu chúng ta không cẩn thận, quyền truy cập cá nhân có thể làm tổn hại đến một bất biến và nó thường làm phức tạp các thay đổi đối với biểu diễn. Tuy nhiên, đối với **complex**, **real ()** và **imag ()** có ý nghĩa về mặt ngữ nghĩa: một số thuật toán được viết rõ ràng nhất nếu chúng có thể thiết lập các phần thực và ảo một cách độc lập.

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

## **{**

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

## **}**

**18.3.6 Helper Functions**

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);**

## **// binar y -, \*, and /**

## **complex operator−(complex); // unar y minus**

## **complex operator+(complex); // unar y plus**

## **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 **imag ()** là yếu tố cần thiết để xác định các phép so sánh. Định nghĩa của hầu hết các hàm trợ giúp sau đây cũng dựa trên **real ()** và **imag ().**

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); // for notational convenience**

## **double imag(complex); // for notational convenience**

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);**

## **// ..**

Từ quan điểm của người dùng, kiểu **complex** được trình bày ở đây gần giống với **complex<double>** được tìm thấy trong **<complex>** trong thư viện chuẩn

**18.4 Type Conversion**

Việc chuyển đổi kiểu có thể được thực hiện bằng

• Một hàm tạo nhận một đối số duy nhất

• Toán tử chuyển đổi

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 , tức là dưới dạng bộ 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, ví dụ: như một đối số của hàm

**18.4.1 Conversion Operators**

Sử dụng một hàm tạo lấy một đối số duy nhất để chỉ định chuyển đổi kiểu là thuận tiện nhưng có những tác động có thể không mong muốn. Ngoài ra, một hàm tạo không thể chỉ định

[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 khai báo cho lớp cũ).

## **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; }**

## **// conversion to int function**

## **};**

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 toán tử 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; } // erro**

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; // no range check (not necessary)**

## **int i = c1+c2; //**

## **i = 64 c1 = c1+c2; // range error: c1 can’t be 64**

## **i = c3−64; //**

## **i = -4 c2 = c3−64; // range error:**

## **c2 can’t be -4**

## **c3 = c4; // no range check (not necessary)**

## **}**

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

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 của **cin.** Giá trị này sau đó có thể được kiểm tra bằng **while.** Tuy nhiên, thường không phải là một ý kiến hay nếu 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, nên tiết kiệm trong việc giới thiệu các nhà khai thác chuyển đổi. 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 ().

Nếu cả chuyển đổi do người dùng xác định và toán tử do người dùng xác định đều được xác định, thì có thể có sự không rõ ràng giữa toán tử do người dùng xác định và toán tử được tích hợp sẵn. Ví dụ:

int operator+(Tiny,Tiny);

## **void f(Tiny t, int i)**

## **{**

## **t+i; // error, ambiguous: ‘‘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 explicit Conversion Operators**

Các toán tử chuyển đổi có xu hướng được xác định để chúng có thể được sử dụng ở mọi nơi. Tuy nhiên, có thể khai báo toán tử chuyển đổi một cách rõ ràng và nó chỉ áp dụng cho việc khởi tạo trực tiếp. Ví dụ:

## **template <typename T, typename D = default\_delete<T>>**

## **class unique\_ptr**

## **{ public:**

## **// ...**

## **explicit operator bool() const noexcept; // does \*this hold a pointer (that is not 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 Inv alid\_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 Ambiguities**

Việc gán giá trị kiểu **V** cho một đối tượng thuộc lớp **X** là hợp pháp nếu có một toán tử gán **X :: operator =** **(Z)** để **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.

Trong một số trường hợp, một giá trị của kiểu mong muốn có thể được xây dựng theo nhiều 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 xem xét nếu không thể giải quyết cuộc gọi mà không có chúng (tức là chỉ sử dụng các chuyển đổi tích hợp 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})!**

## **}**

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.

Sự khăng khăng về phân tích từ dưới lên nghiêm ngặt ngụ ý rằng kiểu trả về không được sử dụng trong giải quyết quá tải. Ví dụ

## **class Quad {**

## **public: Quad(double);**

## **// ...**

## **};**

## **Quad operator+(Quad,Quad);**

## **void f(double a1, double a2)**

## **{**

## **Quad r1 = a1+a2; // double-precision floating-point add**

## **Quad r2 = Quad{a1}+a2; // force quad arithmetic**

## **}**

Lý do cho sự lựa chọn thiết kế này một phần là do phân tích từ dưới lên nghiêm ngặt dễ hiểu hơn và một phần là công việc của trình biên dịch không được coi là quyết định độ chính xác của lập trình viên có thể muốn bổ sung.

Khi đã xác định được kiểu của cả hai phía của một lần khởi tạo hoặc phép gán, thì cả hai kiểu đều được sử dụng để giải quyết việc khởi tạo hoặc gán. Ví dụ:

## **class Real {**

## **public:**

## **operator double();**

## **operator int();**

## **// ...**

## **};**

## **void g(Real a) {**

## **double d = a; // d = a.double();**

## **int i = a; // i = a.int();**

## **d = a; // d = a.double();**

## **i = a; // i = a.int();**

## **}**

Trong những trường hợp này, phân tích kiểu vẫn là từ dưới lên, chỉ với một toán tử duy nhất và các kiểu đối số của nó được xem xét cùng một lúc

**18.5 Advice**

[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 đại diện; §18.3.1.

[6] Ưu tiên các hàm không phải thành viên hơn các thành viên cho các hoạt động không cần quyền truy cập vào biểu diễn; §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 không nhớ 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.

***C19: Special Operators***

**19.1 Introduction**

Quá tải không chỉ dành cho các phép toán số học và logic. Trên thực tế, các nhà khai thác đongs vai trò quan trong trong việc thiết kế của các thùng chứa “con trỏ thông minh” (ví dụ: unique\_ptr và shared), và các lớp khác liên quan đén quản ý tài nguyên.

**19.2 Special Operators**

Toán tử

[] () −> ++ −− new delete

Là điểm đặc biệt từ viêc sử dụng của chúng trong mã đến định nghĩa của lập trình viên khác nhau từ đó được sử dụng cho các toán tử đơn phân và nhị phân thông thường, như +,< và - . Dấu [] (đăng ký) và () (gọi) toán tử là một trong những toán tử hữu ích nhất do người dung kiểm chứng.

**19.2.1 Subscripting**

Một hàm operator[] được sử dụng đẻ cung cấp cho các chỉ số con một ý nghĩa cho các đối tượng lớp. Điều này làm cho nó có thể xác định được vectors, mảng kết hợp, v.v.

Ví dụ:

**struct Assoc {**

**vector<pair<string,int>> vec; // vector of {name,value} pairs**

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

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

**};**

Một Assoc giữ 1 vector của cặp std::pairs. Ví dụ:

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

**// search for s; return a reference to its value if found;**

**// otherwise, make a new pair {s,0} and return a reference to its value**

**{**

**for (auto x : vec)**

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

**vec.push\_back({s,0}); // initial value: 0**

**return vec.back().second; // return last element (§31.2.2)**

**}**

Chúng ta có thể dùng Assoc như sau:

**int main()**

**{**

**Assoc values;**

**string buf;**

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

**for (auto x : values.vec)**

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

**}**

Thư viện tiêu chuẩn map và unordered\_map có thử tự là những phát triển tiếp theo của ý tưởng về mảng với các triển khai ít đơn giản hơn.

Một hàm operator[]() phải là một hàm thành viên không tĩnh.

**19.2.2 Function Call**

Hàm gọi là biểu thức ký hiệu ( danh sách biểu thức ), có thể được hiểu như là một toán hạng nhị phân là toán hạng bên trái và danh sách biểu thức là toán hạng bên phải.

Toán tử gọi, (), can be 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);**

**// ...**

**};**

Việc ghi đè toán tử hàm gọi dường như hữu ích chủ yếu để xác định các kiểu chỉ có một thao tác duy nhất và cho một hoạt động chiếm ưu thế. Toán tử gọi còn được gọi là toán tử ứng dụng. Một đối tượng hoạt động giống như một hàm thường được gọi là một dối tượng giống hàm hoặc đơn giản đó là một đối tượng hàm. Với chức năng này cho phép chúng ta viết mã lấy cách phép toán tầm thường làm tham số. Trong nhiều trường hợp, điều cần thiết là các đối tượng chức năng có thể chứa dữ liệu cần thiết để thực hiện hoạt động của chúng. Ví dụ:

**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; } // them một 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 them số đó vào 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});**

**}**

Lưu ý rằng Add(z) xây dựng một đối tượng bởi for\_each() sử dụng lặp đi lặp lại: Toán tử operater()() được gọi cho mỗi phần tử của dãy. Ví dụ:

**template<typename Iter, typename Fct>**

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

**{**

**while (b != e) f(∗b++);**

**return f;**

**}**

Lưu ý rằng biểu thức về cơ bản là một cú pháp để xác định một đối tượng hàm. Ví dụ:

**void h2(vector<complex>& vec, list<complex>& lst, complex z)**

**{**

**for\_each(vec.begin(),vec.end(),[](complex& a){ a+={2,3}; });**

**for\_each(lst.begin(),lst.end(),[](complex& a){ a+=z; });**

**}**

Trong trường hợp này, mỗi biểu thức lambda tạo ra tương đương với đối tượng hàm Add.

Các cách sử dụng phổ biến khác của toán tử operator () () như một toán tử chuỗi con và như một toán tử chỉ số con cho mảng nhiều chiều. Các toán tử cuộc gọi hàm thường là các mẫu.

**19.2.3 Dereferencing**

Toán tử trỏ đến, -> (còn được gọi là toán tử mũi tên), có thể được định nghĩa như một toán tử sửa lỗi một bậc. Ví dụ:

**class Ptr {**

**// ...**

**X∗ operator−>();**

**};**

Việc biến đổi đối tượng p thành con trỏ p.operator -> () không phụ thuộc vào phần tử m chỉ vào. Đó là ý nghĩa mà toán tử -> () là một toán tử hậu tố một ngôi. Tuy nhiên, không có cú pháp mới nào được giới thiệu, vì vậy tên thành viên vẫn được yêu cầu sau dấu ->. Ví dụ:

**void g(Ptr p)**

**{**

**X∗ q1 = p−>; // syntax error**

**X∗ q2 = p.operator−>(); // OK**

**}**

Nạp chồng phương thức -> chủ yếu hữu ích để tạo ‘‘ con trỏ thông minh ’’, tức là các đối tượng hoạt động giống như con trỏ và ngoài ra còn thực hiện một số hành động bất cứ khi nào một đối tượng được truy cập thông qua chúng. Tiêu chuẩn thư viện ‘‘ con trỏ thông minh ’’ unique\_ptr và shared\_ptr cung cấp toán tử ->. Ví dụ:

**class Disk\_ptr {**

**string identifier;**

**T∗ in\_core\_address;**

**// ...**

**public:**

**Disk\_ptr(const string& s) : identifier{s}, in\_core\_address{nullptr} { }**

**̃Disk\_ptr() { write\_to\_disk(in\_core\_address,identifier); }**

**T∗ operator−>()**

**{**

**if (in\_core\_address == nullptr)**

**in\_core\_address = read\_from\_disk(identifier);**

**return in\_core\_address;**

**}**

**};**

Disk\_ptr có thể được sử dụng như thế này:

**struct Rec {**

**string name;**

**// ...**

**};**

**void update(const string& s)**

**{**

**Disk\_ptr<Rec> p {s};**

**p−>name = "Roscoe";**

**// ...**

**}**

Đối với con trỏ thông thường, việc sử dụng -> đồng nghĩa với một số cách sử dụng một ngôi ∗ và []. Cho một lớp Y mà ->, ∗ và [] có nghĩa mặc định của chúng và một Y ∗ được gọi là p, thì:

**p−>m == (∗p).m // đúng**

**(∗p).m == p[0].m // đúng**

**p−>m == p[0].m // đúng**

**template<typename T>**

**class Ptr {**

**Y∗ p;**

**public:**

**Y∗ operator−>() { return p; } // quyền tri cập thành viên**

**Y& operator∗() { return ∗p; } // trỏ đến truy cập toàn bộ đối tượng**

**Y& operator[](int i) { return p[i]; } // quyền truy cập phần tử truy cập**

**// ...**

**};**

Toán tử -> phải là một hàm thành viên không tĩnh. Nếu được sử dụng, kiểu trả về của nó phải là một con trỏ hoặc một đối tượng của một lớp mà bạn có thể áp dụng ->. Phần thân của hàm thành viên lớp mẫu chỉ được kiểm tra nếu hàm được sử dụng, vì vậy chúng ta có thể định nghĩa toán tử -> () mà không cần lo lắng về các kiểu, chẳng hạn như Ptr <int>, mà -> không có lý. Mặc dù có sự giống nhau giữa -> và. (dấu chấm), không có cách nạp chồng toán tử. (dấu chấm).

**19.2.4 Increment and Decrement**

Khi mọi người phát minh ra '' con trỏ thông minh '', họ thường quyết định cung cấp toán tử tăng ++ và toán tử giảm −− để phản ánh việc sử dụng các toán tử này cho các kiểu tích hợp. Điều này đặc biệt rõ ràng và cần thiết khi mục đích là thay thế một loại con trỏ thông thường bằng một loại '' con trỏ thông minh '' có cùng ngữ nghĩa, ngoại trừ việc nó thêm một chút kiểm tra lỗi thời gian chạy. Ví dụ:

**void f1(X a) // traditional use**

**{**

**X v[200];**

**X∗ p = &v[0];**

**p−−;**

**∗p = a; // oops: p out of range, uncaught**

**++p;**

**∗p = a; // OK**

**}**

Ở đây, chúng ta có thể muốn thay thế X ∗ bằng một đối tượng của lớp Ptr <X> chỉ có thể được tham chiếu nếu nó thực sự trỏ đến X. Chúng ta cũng muốn đảm bảo rằng p chỉ có thể được tăng và giảm dần nếu nó trỏ đến một đối tượng trong một mảng và các phép toán tăng và giảm mang lại một đối tượng trong mảng đó. Ví dụ:

**void f2(Ptr<X> a) // checked**

**{**

**X v[200];**

**Ptr<X> p(&v[0],v);**

**p−−;**

**∗p = a; // run-time error: p out of range**

**++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 tăng và giảm tiền tố và hậu tố cho Ptr <T>. VÍ dụ:

**template<typename T>**

**class Ptr {**

**T∗ ptr;**

**T∗ array;**

**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++(); // prefix**

**Ptr operator++(int); // postfix**

**Ptr& operator−−(); // prefix**

**Ptr operator−−(int); // postfix**

**T& operator∗(); // prefix**

**};**

Đối số int được sử dụng để chỉ ra rằng hàm sẽ được gọi cho ứng dụng hậu tố của ++. Đối số chỉ đơn giản là một giả được sử dụng để phân biệt giữa tiền tố và ứng dụng sau sửa chữa. Đối số giả chỉ được sử dụng cho hậu tố '' lẻ '' ++ và −−.

Xem xét việc bỏ qua hậu tố ++ và −− trong một thiết kế. Nó không chỉ khác lạ về mặt cú pháp, nó có xu hướng khó triển khai hơn một chút so với các phiên bản sau sửa lỗi, kém hiệu quả hơn và ít được sử dụng hơn. Ví dụ:

**template<typename T>**

**Ptr& Ptr<T>::operator++() // return the current object after incrementing**

**{**

**// ... check that ptr+1 can be pointed to ...**

**return ∗++ptr;**

**}**

**template<typename T>**

**Ptr Ptr<T>::operator++(int) // increment and return a Ptr with the old value**

**{**

**// ... check that ptr+1 can be pointed to ...**

**Ptr<T> old {ptr,array,sz};**

**++ptr;**

**return old;**

**}**

Toán tử tăng trước có thể trả về một tham chiếu đến đối tượng của nó. Toán tử tăng sau phải tạo một đối tượng mới để trả về.

Sử dụng Ptr, như ví dụ sau:

**void f3(T a) // checked**

**{**

**T v[200];**

**Ptr<T> p(&v[0],v,200);**

**p.operator−−(0); // suffix: p--**

**p.operator∗() = a; // run-time error: p out of range**

**p.operator++(); // prefix: ++p**

**p.operator∗() = a; // OK**

**}**

**19.2.5 Allocation and Deallocation**

Toán tử mới có được bộ nhớ của nó bằng cách gọi một toán tử new (). Tương tự, toán tử xóa giải phóng bộ nhớ của nó bằng cách gọi một toán tử xóa (). Người dùng có thể định nghĩa lại toán tử toàn cục new () và toán tử delete () hoặc định nghĩa toán tử new () và toán tử delete () cho một lớp cụ thể.

Sử dụng bí danh kiểu thư viện tiêu chuẩn kích thước t cho các kích thước, khai báo của các phiên bản toàn cầu trông giống như sau:

**void∗ operator new(siz e\_t); // use for individual object**

**void∗ operator new[](siz e\_t); // use for array**

**void operator delete(void∗, siz e\_t); // use for individual object**

**void operator delete[](void∗, siz e\_t); // use for array**

Nghĩa là, khi new cần bộ nhớ trên kho lưu trữ miễn phí cho một đối tượng kiểu X, nó sẽ gọi toán tử new (sizeof (X)). Tương tự, khi new cần bộ nhớ trên kho lưu trữ miễn phí cho một mảng N đối tượng kiểu X, nó gọi toán tử new [] (N ∗ siz eof (X)). Một biểu thức mới có thể yêu cầu nhiều bộ nhớ hơn được chỉ ra bởi kích thước N ∗ của (X), nhưng nó sẽ luôn làm như vậy về số lượng ký tự (tức là một số byte). Thay thế toán tử toàn cục new () và toán tử delete () không dành cho những người yếu đuối và không được khuyến khích.

Một cách tiếp cận chọn lọc hơn và thường tốt hơn là cung cấp các hoạt động này cho một lớp cụ thể. Lớp này có thể là cơ sở cho nhiều lớp dẫn xuất. Ví dụ:

**class Employee {**

**public:**

**// ...**

**void∗ operator new(siz e\_t);**

**void operator delete(void∗, siz e\_t);**

**void∗ operator new[](siz e\_t);**

**void operator delete[](void∗, siz e\_t);**

**};**

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

**void∗ Employee::operator new(siz e\_t s)**

**{**

**// allocate s bytes of memory and return a pointer to it**

**}**

**void Employee::operator delete(void∗ p, size\_t s)**

**{**

**if (p) { // delete only if p!=0;**

**// assume p points to s bytes of memory allocated by Employee::operator new()**

**// and free that memory for reuse**

**}**

**}**

Việc sử dụng đối số kích thước t bí ẩn cho đến nay đã trở nên hiển nhiên. Nó là kích thước của đối tượng bị xóa. Xóa Employee thuần túy cung cấp một giá trị đối số có kích thước là (Employee); xóa một Trình quản lý có nguồn gốc từ Nhân viên không có toán tử riêng delete () cho một giá trị đối số là sizeof (Trình quản lý). Điều này cho phép trình phân bổ theo lớp cụ thể tránh lưu trữ thông tin kích thước với mỗi lần phân bổ. Tuy nhiên, làm như vậy khó cải thiện đáng kể tốc độ và mức tiêu thụ bộ nhớ của bộ cấp phát đa năng.

**Employee∗ p = new Manager;**

**// ...**

**delete p;**

Về nguyên tắc, việc phân bổ giao dịch sau đó được thực hiện bởi hàm hủy (biết kích thước của lớp của nó).

**19.2.6 User-defined Literals**

C ++ cung cấp các ký tự cho nhiều kiểu tích hợp sẵn :

**123 // int**

**1.2 // double**

**1.2F // float**

**'a' // char**

**1ULL // unsigned long long**

**0xD0 // hexadecimal unsigned**

**"as" // C-style string (const char[3])**

Ngoài ra, chúng ta có thể định nghĩa các chữ cho các kiểu do người dùng xác định và các dạng chữ mới cho các kiểu dựng sẵn. Ví dụ:

**"Hi!"s // string, not ‘‘zero-ter minated array of char’’**

**1.2i // imaginar y**

**101010111000101b // binar y**

**123s // seconds**

**123.56km // not miles! (units)**

**1234567890123456789012345678901234567890x // extended-precision**

Tên của toán tử theo nghĩa đen là toán tử "" theo sau là hậu tố. Ví dụ:

# **constexpr complex<double> operator"" i(long double d) // imaginar y literal**

# **{**

# **return {0,d}; // complex is a literal type**

# **}**

# **std::string operator"" s(const char∗ p, size\_t n) // std::string literal**

# **{**

# **return string{p,n}; // requires free-store allocation**

# **}**

Hai toán tử này lần lượt xác định các hậu tố i và s. Sử dụng constexpr để cho phép đánh giá thời gian biên dịch. Ví dụ:

**template<typename T> void f(const T&);**

**void g()**

**{**

**f("Hello"); // pass pointer to char\***

**f("Hello"s); // pass (five-character) string object**

**f("Hello\n"s); // pass (six-character) string object**

**auto z = 2+1i; // complex{2,1}**

**}**

Cơ bản là sau khi phân tích cú pháp những gì có thể là một chữ, trình biên dịch luôn kiểm tra hậu tố. Cơ chế ký tự do người dùng xác định chỉ đơn giản cho phép người dùng chỉ định một hậu tố mới và xác định những gì sẽ được thực hiện với ký tự trước nó. Không thể xác định lại ý nghĩa của một hậu tố nghĩa đen có sẵn hoặc để tăng cường cú pháp của các ký tự.

Có bốn loại ký tự có thể được nối với nhau để tạo thành một ký tự được xác định đã sử dụng:

* Một ký tự số nguyên: được chấp nhận bởi toán tử ký tự lấy một giá trị dài không dấu hoặc đối số const char ∗ hoặc bởi toán tử ký tự mẫu, ví dụ: 123m hoặc 2345678901234567890X
* Một ký tự dấu phẩy động: được chấp nhận bởi toán tử ký tự lấy một đối số kép dài hoặc const char ∗ hoặc bởi toán tử ký tự mẫu, ví dụ: 12345678901234567890.976543210x hoặc 3,99s
* Một chuỗi ký tự: được chấp nhận bởi một toán tử ký tự lấy một cặp đối số (const char ∗, size\_t), ví dụ: "string" s và R "(Foo \ bar)" \_ path.
* Một ký tự chữ: được chấp nhận bởi toán tử ký tự lấy đối số ký tự kiểu char, wchar\_t, char16\_t hoặc char32\_t, ví dụ: 'f'\_runic hoặc u'BEEF'\_w.

Ví dụ: chúng ta có thể xác định một toán tử chữ để thu thập các chữ số cho các giá trị số nguyên không thể được biểu diễn trong bất kỳ kiểu số nguyên tích hợp nào:

**Bignum operator"" x(const char∗ p)**

**{**

**return Bignum(p);**

**}**

**void f(Bignum);**

**f(123456789012345678901234567890123456789012345x);**

Ở đây, chuỗi kiểu C "123456789012345678901234567890123456789012345" được chuyển cho toán tử "" x ().Lưu ý rằng tôi không đặt những chữ số đó trong dấu ngoặc kép.

Để chuyển một chuỗi kiểu C từ văn bản nguồn chương trình thành một toán tử chữ, chúng tôi yêu cầu cả chuỗi và số ký tự của nó. Ví dụ:

**string operator"" s(const char∗ p, size\_t n);**

**string s12 = "one two"s; // calls operator ""("one two",7)**

**string s22 = "two\ntwo"s; // calls operator ""("two\ntwo",7)**

**string sxx = R"(two\ntwo)"s; // calls operator ""("two\\ntwo",8)**

Trong chuỗi thô, "\ n" đại diện cho hai ký tự '\' và 'n'.

Có thể áp dụng toán tử chữ chỉ lấy một đối số const char ∗ (và không có kích thước) cho các chữ số nguyên và dấu phẩy động. Ví dụ:

**string operator"" SS(const char∗ p);**

**string s12 = "one two"SS;**

**string s13 = 13SS;**

Một toán tử chữ chuyển đổi các giá trị số thành chuỗi có thể khá khó hiểu.

Một toán tử theo mẫu là một toán tử theo nghĩa đen nhận đối số của nó như một gói tham số mẫu, chứ không phải là một đối số của hàm. Ví dụ:

**template<char...>**

**constexpr int operator"" \_b3();**

Do đó, chúng tôi nhận được:

**201\_b3 // means operator"" b3<’2’,’0’,’1’>(); so 9\*2+0\*3+1 == 19**

**241\_b3 // means operator"" b3<’2’,’4’,’1’>(); so error: 4 isn’t a ternar y digit**

Để xác định toán tử "" \_b3 (), chúng ta cần một số hàm trợ giúp:

**constexpr int ipow(int x, int n) // x to the nth power for n>=0**

**{**

**return (n>0) ? x∗ipow(n−1) : 1;**

**}**

**template<char c> // handle the single ternar y digit case**

**constexpr int b3\_helper()**

**{**

**static\_assert(c<'3',"not a ternar y digit");**

**return c;**

**}**

**template<char c, char... tail> // peel off one ternar y digit**

**constexpr int b3\_helper()**

**{**

**static\_assert(c<'3',"not a ternar y digit");**

**return ipow(3,siz eof...(tail))∗(c−'0')+b3\_helper(tail...);**

**}**

Do đó, chúng ta có thể xác định toán tử chữ cơ sở 3 của chúng ta:

**template<char... chars>**

**constexpr int operator"" \_b3() // base 3, i.e., ternar y**

**{**

**return b3\_helper(chars...);**

**}**

Nhiều hậu tố sẽ ngắn (ví dụ: s cho std :: string, i cho ảo, m cho mét và x cho mở rộng), vì vậy các cách sử dụng khác nhau có thể dễ dàng xung đột. Sử dụng không gian tên để ngăn đụng độ:

**namespace Numerics {**

**// ...**

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

**namespace literals {**

**Bignum operator"" x(char const∗);**

**}**

**// ...**

**}**

**using namespace Numerics::literals;**

Thư viện tiêu chuẩn dự trữ tất cả các hậu tố không bắt đầu bằng dấu gạch dưới ban đầu, vì vậy hãy xác định các hậu tố của bạn bắt đầu bằng dấu gạch dưới hoặc có nguy cơ bị hỏng mã trong tương lai:

**123km // reser ved by the standard librar y**

**123\_km // available for your use**

**19.3 A String Class ( Một lớp chuỗi)**

Lớp chuỗi tương đối đơn giản được trình bày trong phần này minh họa một số kỹ thuật hữu ích cho việc thiết kế và triển khai các lớp bằng cách sử dụng các toán tử được định nghĩa theo quy ước. Chuỗi cung cấp ngữ nghĩa giá trị, quyền truy cập đã kiểm tra và chưa kiểm tra vào các ký tự, luồng I / O, hỗ trợ vòng lặp phạm vi cho, phép toán bình đẳng và toán tử nối. Tôi cũng đã thêm một ký tự String, mà std :: string không (chưa) có.

Để cho phép khả năng tương tác đơn giản với các chuỗi kiểu C (bao gồm các ký tự chuỗi ), tôi biểu diễn các chuỗi dưới dạng mảng ký tự có kết thúc bằng không. Đối với chủ nghĩa hiện thực, ta 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 t self, chứ không phải trên cửa hàng miễn phí. Điều này tối ưu hóa việc sử dụng chuỗi cho các chuỗi nhỏ. 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 khả thi và việc phân bổ và phân bổ kho lưu trữ tự do tương đối tốn kém.

Để cho phép các Chuỗi '' phát triển '' một cách hiệu quả bằng cách thêm các ký tự vào cuối, ta triển khai một sơ đồ để giữ thêm không gian cho sự phát triển như vậy tương tự như sơ đồ được sử dụng cho vectơ. Điều này làm cho String trở thành mục tiêu phù hợp cho nhiều dạng đầu vào khác nhau.

**19.3.1 Essential Operations**

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

**class String {**

**public:**

**String(); // default constructor : x{""}**

**explicit String(const char∗ p); // constructor from C-style string: x{"Euler"}**

**String(const String&); // copy constr uctor**

**String& operator=(const String&); // copy assignment**

**String(String&& x); // move constr uctor**

**String& operator=(String&& x); // move assignment**

**̃String() { if (short\_max<sz) delete[] ptr; } // destructor**

**// ...**

**};**

Chuỗi này có ngữ nghĩa giá trị. Nghĩa là, sau một phép gán s1 = s2, hai chuỗi s1 và s2 hoàn toàn khác biệt và những thay đổi tiếp theo đối với chuỗi này không ảnh hưởng đến chuỗi kia. Điều đó có nghĩa là để những thay đổi đối với s2 sau s1 = s2 cũng ảnh hưởng đến giá trị của s1. Tuy nhiên, để ngữ nghĩa giá trị phù hợp với túi tiền, chúng ta cần chuyển Chuỗi bằng tham chiếu khi chúng ta không cần bản sao và triển khai ngữ nghĩa di chuyển để tối ưu hóa lợi nhuận.

Lưu ý rằng nó yêu cầu các phiên bản do người dùng xác định của các hoạt động sao chép và di chuyển.

**19.3.2 Access to Characters**

Việc thiết kế các toán tử truy cập cho một chuỗi là một chủ đề khó vì truy cập lý tưởng là bằng ký hiệu thông thường (nghĩa là sử dụng []), hiệu quả tối đa và được kiểm tra phạm vi. Bạn không thể có tất cả các thuộc tính này cùng một lúc. Ở đây, ta 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 các phép toán được kiểm tra phạm vi tại ():

**class String {**

**public:**

**// ...**

**char& operator[](int n) { return ptr[n]; } // unchecked element access**

**char operator[](int n) const { return ptr[n]; }**

**char& at(int n) { check(n); return ptr[n]; } // range-checked element access**

**char at(int n) const { check(n); return ptr[n]; }**

**String& operator+=(char c); // add c at end**

**const char∗ c\_str() { return ptr; } // C-style string access**

**const char∗ c\_str() const { return ptr; }**

**int size() const { return sz; } // number of elements**

**int capacity() const // elements plus available space**

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

**// ...**

**};**

Ý tưởng là sử dụng [] cho mục đích sử dụng thông thường. Ví dụ:

**int hash(const String& s)**

**{**

**int h {s[0]};**

**for (int i {1}; i!=s.size(); i++) h ˆ= s[i]>>1; // unchecked access to s**

**return h;**

**}**

Ở đây, sử dụng dấu kiểm tại () sẽ là thừa vì chúng tôi chỉ truy cập chính xác s từ 0 đến s.size () - 1.

Chúng ta có thể sử dụng tại () nơi chúng ta thấy có thể xảy ra sai sót. Ví dụ:

**void print\_in\_order(const String& s,const vector<int>& index)**

**{**

**for (x : index) cout << s.at(x) << ‘\n';**

**}**

Giả định rằng mọi người sẽ sử dụng at () nhất quán ở những nơi có thể mắc lỗi là quá lạc quan, vì vậy một số triển khai của std :: string (từ đó quy ước [] / at () được mượn) cũng kiểm tra [].Tuy nhiên, đối với các tác vụ thao tác chuỗi nghiêm trọng, việc kiểm tra phạm vi đối với mỗi quyền truy cập ký tự có thể áp đặt chi phí khá đáng chú ý.

* + 1. **Representation**

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

* Để dễ dàng chuyển đổi một chuỗi kiểu C (ví dụ: một chuỗi ký tự) thành một Chuỗi và để cho phép dễ dàng truy cập vào các ký tự của Chuỗi dưới dạng chuỗi kiểu C
* Để giảm thiểu việc sử dụng cửa hàng miễn phí
* Để làm cho việc 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 str ing ngắn

size()==sz là số phần tử

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

ptr trỏ đến đầu chuỗi ký tự

chuỗi ký tự được giữ nguyên số 0:ptr[size()]==0;

Điều này cho phép chúng tôi sử dụng các hàm chuỗi thư viện C 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;

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

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

**public:**

**// ...**

**private:**

**static const int short\_max = 15;**

**int sz; // number of characters**

**char∗ ptr;**

**union {**

**int space; // unused allocated space**

**char ch[shor t\_max+1]; // leave space for terminating 0**

**};**

**void check(int n) const // range check**

**{**

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

**throw std::out\_of\_rang e("String::at()");**

**}**

**// ancillar y member functions:**

**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 Chuỗi, 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 cấp thêm dung lượng để mở rộng. Thành viên được đặt tên khoảng trắng là số ký tự như vậy.

Trong cả hai trường hợp, số phần tử được giữ trong sz và chúng tôi xem xét sz, để xác định sơ đồ triển khai 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 hàm truy cập không cần phải kiểm tra xem biểu diễn nào được sử dụng; chúng chỉ đơn giản là sử dụng ptr. Chỉ các hàm tạo, phép gán, phép di chuyển và hàm hủy mới phải quan tâm đến hai lựa chọn thay thế.

Chúng ta chỉ sử dụng mảng ch khi sz <= short\_max và không gian nguyên chỉ khi! (Sz <= short\_max). Do đó, sẽ rất lãng phí nếu phân bổ không gian cho cả ch và không gian trong một đối tượng Chuỗi. Để tránh lãng phí như vậy, tôi sử dụng công đoàn. Đặc biệt, tôi đã sử dụng một dạng liên hiệp được gọi là liên hiệp ẩn danh, được thiết kế đặc biệt để cho phép một lớp quản lý các đại diện thay thế của các đối tượng. Chỉ một thành viên có thể được sử dụng tại bất kỳ thời điểm nào, nhưng nếu không chúng được truy cập và sử dụng chính xác như thể chúng là các thành viên riêng biệt của phạm vi bao quanh liên minh ẩn danh. Nhiệm vụ của lập trình viên là đảm bảo rằng chúng không bao giờ bị lạm dụng. Ví dụ, tất cả các hàm thành viên của Chuỗi sử dụng khoảng trắng phải đảm bảo rằng nó thực sự là không gian đã được thiết lập chứ không phải ch. Điều đó được thực hiện bằng cách xem sz <= short\_max. Nói cách khác, Shape là (trong số những thứ khác) là một liên hợp phân biệt với sz <= short\_max là đối tượng phân biệt.

* + - 1. **Ancillary Functions**

Ngoài các chức năng dành cho mục đích sử dụng chung, ta thấy rằng mã của mình trở nên sạch hơn khi tôi cung cấp ba hàm phụ trợ là '' khối xây dựng '' để giúp ta thực hiện việc sao chép lại hơi phức tạp và giảm thiểu việc sao chép mã. Hai trong số đó cần truy cập vào biểu diễn của String, vì vậy ta đã biến họ thành thành viên. Tuy nhiên, tôi đã đặt họ thành thành viên riêng tư vì họ không đại diện cho các hoạt động thường hữu ích và an toàn để sử dụng. Đối với nhiều lớp thú vị, việc triển khai không chỉ là biểu diễn cộng với các hàm công khai. Các chức năng phụ trợ có thể dẫn đến ít trùng lặp mã hơn, thiết kế tốt hơn và khả năng bảo trì được cải thiện.

Hàm đầu tiên di chuyển các ký tự vào bộ nhớ mới được cấp phát:

**char∗ expand(const char∗ ptr, int n) // expand into free store**

**{**

**char∗ p = new char[n];**

**strcpy(p,ptr); // §43.4**

**return p;**

**}**

Hàm này không truy cập vào biểu diễn chuỗi, vì vậy tôi không đặt nó làm thành viên.

Hàm thực thi thứ hai được sử dụng bởi các hoạt động sao chép để cung cấp cho một Chuỗi bản sao của các thành viên của chuỗi khác:

**void String::copy\_from(const String& x)**

**// make \*this a copy of x**

**{**

**if (x.sz<=short\_max) { // copy \*this**

**memcpy(this,&x,siz eof(x)); // §43.5**

**ptr = ch;**

**}**

**else { // copy the elements**

**ptr = expand(x.ptr,x.sz+1);**

**sz = x.sz;**

**space = 0;**

**}**

**}**

Bất kỳ quá trình dọn dẹp cần thiết nào đối với Chuỗi mục tiêu là nhiệm vụ của các trình gọi copy\_from (); copy\_from () ghi đè mục tiêu của nó một cách vô điều kiện. Ta sử dụng thư viện chuẩn memcpy () để sao chép các byte của nguồn vào đích. Đó là một chức năng cấp thấp và đôi khi khá khó chịu. Nó chỉ nên được sử dụng khi không có đối tượng nào có hàm tạo hoặc hàm hủy trong bộ nhớ đã sao chép vì memcpy () không biết gì về kiểu. Cả hai hoạt động sao chép chuỗi đều sử dụng copy\_from ().

Chức năng tương ứng cho các hoạt động di chuyển là:

**void String::move\_from(String& x)**

**{**

**if (x.sz<=short\_max) { // copy \*this**

**memcpy(this,&x,siz eof(x)); // §43.5**

**ptr = ch;**

**}**

**else { // grab the elements**

**ptr = x.ptr;**

**sz = x.sz;**

**space = x.space;**

**x.ptr = x.ch; // x = ""**

**x.sz = 0;**

**x.ch[0]=0;**

**}**

**}**

Ta cũng có thể đã sử dụng memcpy () trong trường hợp chuỗi dài, nhưng vì biểu diễn chuỗi dài chỉ sử dụng một phần biểu diễn của String, tôi quyết định sao chép riêng từng thành viên đã sử dụng.

* + 1. **Member Functions**

Hàm tạo mặc định xác định một Chuỗi trống:

**String::String() // default constructor : x{""}**

**: sz{0}, ptr{ch} // ptr points to elements, ch is an initial location (§19.3.3)**

**{**

**ch[0] = 0; // terminating 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 ký tự 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); // copy characters into ptr from p**

**}**

Nếu đối số là một chuỗi ngắn, ptr được đặt để trỏ đến ch; nếu không, không gian được phân bổ trên cửa hàng miễn phí. Trong cả hai trường hợp, các ký tự được sao chép từ chuỗi đối số vào bộ nhớ được quản lý bởi Chuỗi.

Hàm tạo sao chép chỉ cần sao chép biểu diễn của các đối số của nó:

**String::String(const String& x) // copy constr uctor**

**{**

**copy\_from(x); // copy representation from x**

**}**

Tương tự, hàm tạo di chuyển di chuyển biểu diễn từ nguồn của nó (và có thể đặt đối số của nó là chuỗi trống):

**String::String(String&& x) // move constr uctor**

**{**

**move\_from(x);**

**}**

Giống như hàm tạo bản sao, phép gán bản sao sử dụng copy\_from () để sao chép biểu diễn của 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 gặp rắc rối với việc tự chuyển nhượng (ví dụ: s = s):

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

**{**

**if (this==&x) return ∗this; // deal with self-assignment**

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

**copy\_from(x);**

**delete[] p;**

**return ∗this;**

**}**

Phép toán Chuỗi phức tạp nhất về mặt logic là + =, thêm một ký tự vào cuối chuỗi, tăng kích thước của nó lên một:

**String& String::operator+=(char c)**

**{**

**if (sz==short\_max) { // expand to long string**

**int n = sz+sz+2; // double the allocation (+2 because of the terminating 0)**

**ptr = expand(ptr,n);**

**space = n−sz−2;**

**}**

**else if (short\_max<sz) {**

**if (space==0) { // expand in free store**

**int n = sz+sz+2; // double the allocation (+2 because of the terminating 0)**

**char∗ p = expand(ptr,n);**

**delete[] ptr;**

**ptr = p;**

**space = n−sz−2;**

**}**

**else**

**−−space;**

**}**

**ptr[sz] = c; // add c at end**

**ptr[++sz] = 0; // increase size and set terminator**

**return ∗this;**

**}**

Có rất nhiều điều đang diễn ra ở đây: toán tử + = () phải theo dõi xem biểu diễn nào (ngắn hay dài) được sử dụng và liệu có thêm không gian để mở rộng hay không. Nếu cần thêm không gian, thì expand () được gọi để phân bổ không gian đó và di chuyển các ký tự cũ vào không gian mới. Nếu có một phân bổ cũ cần xóa, nó sẽ được trả về để + = có thể xóa nó. Sau khi có đủ dung lượng, việc đặt ký tự mới c vào đó và thêm số 0 tận cùng là điều nhỏ.

Lưu ý tính toán bộ nhớ khả dụng cho không gian. Trong số tất cả các quá trình triển khai Chuỗi mất nhiều thời gian nhất để làm đúng: một phép tính nhỏ lộn xộn dễ xảy ra lỗi từng lỗi một. Hằng số 2 lặp đi lặp lại đó có cảm giác khủng khiếp giống như một '' hằng số ma thuật ''.

Tất cả các thành viên của Chuỗi lưu ý không sửa đổi một đại diện mới trước khi họ chắc chắn rằng một đại diện mới có thể được đưa vào. Đặc biệt, họ không xóa cho đến khi thực hiện xong bất kỳ thao tác mới nào có thể. Nếu bạn không thích loại mã khó hiểu được trình bày như một phần của quá trình triển khai Chuỗi, chỉ cần sử dụng std :: string. Ở một mức độ lớn, các cơ sở thư viện đứng tồn tại để giúp chúng ta không phải lập trình ở mức lev thấp này hầu hết thời gian. Tuy nhiên, một khi bài tập được thực hiện, một kết quả sẽ là sự đánh giá cao những gì tiêu chuẩn mang lại và mong muốn không duy trì phiên bản của riêng bạn.

* + 1. **Helper Functions**

Để hoàn thành Chuỗi lớp, tôi cung cấp một tập hợp các hàm 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. Tất cả những điều này phản ánh các lựa chọn thiết kế được sử dụng cho std :: string. 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();**

**}**

**istream& operator>>(istream& is, String& s)**

**{**

**s = "";**

**is>>ws;**

**char ch = ' ';**

**while(is.get(ch) && !isspace(ch))**

**s += ch;**

**return is;**

**}**

Ta 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);**

**}**

Để hỗ trợ vòng lặp range-for, chúng ta cần begin () và end (). Một lần nữa, chúng ta có thể cung cấp các hàm đó dưới dạng các hàm tự do (nonmember) mà không cần truy cập trực tiếp vào việc triển khai Chuỗi:

**char∗ begin(String& x) // C-string-style access**

**{**

**return x.c\_str();**

**}**

**char∗ end(String& x)**

**{**

**return x.c\_str()+x.size();**

**}**

**const char∗ begin(const String& x)**

**{**

**return x.c\_str();**

**}**

**const char∗ end(const String& x)**

**{**

**return x.c\_str()+x.size();**

**}**

Với hàm thành viên + = có thêm một ký tự ở cuối, các toán tử nối dễ dàng được cung cấp dưới dạng các hàm không phải là bộ nhớ:

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

**{**

**for (auto x : b)**

**a+=x;**

**return a;**

**}**

**String operator+(const String& a, const String& b)// concatenation**

**{**

**String res {a};**

**res += b;**

**return res;**

**}**

Việc sử dụng + = này được hiểu là toán tử + = (s, String ("Gunnar")). Ta có thể cung cấp một Chuỗi :: operator + = (const char ∗) hiệu quả hơn, nhưng tôi không biết liệu hiệu suất thêm vào có đáng giá trong mã thế giới thực hay không. Trong những trường hợp như vậy, tôi cố gắng tiết chế và đưa ra thiết kế tối giản. Có thể làm một điều gì đó không phải là một lý do chính đáng để làm điều đó.

Tương tự, ta không thể cố tối ưu hóa + = bằng cách tính đến kích thước của chuỗi nguồn.

Thêm s làm hậu tố theo nghĩa đen của chuỗi có nghĩa là Chuỗi là không đáng kể:

**String operator"" \_s(const char∗ p, size\_t)**

**{**

**return String{p};**

**}**

Bây giờ chúng ta có thể viết:

**void f(const char∗); // C-style string**

**void f(const String&); // our string**

**void g()**

**{**

**f("Madden's"); // f(const char\*)**

**f("Christopher's"\_s); // f(const String&);**

**}**

* + 1. **Using Our String**

Chương trình chính chỉ đơn giản là thực hành các toán tử chuỗi một chút:

**int main()**

**{**

**String s ("abcdefghij");**

**cout << s << '\n';**

**s += 'k';**

**s += 'l';**

**s += 'm';**

**s += 'n';**

**cout << s << '\n';**

**String s2 = "Hell";**

**s2 += " and high water";**

**cout << s2 << '\n';**

**String s3 = "qwerty";**

**s3 = s3;**

**String s4 ="the quick bro wn fox jumped over the lazy dog";**

**s4 = s4;**

**cout << s3 << " " << s4 << "\n";**

**cout << s + ". " + s3 + String(". ") + "Horsefeathers\n";**

**String buf;**

**while (cin>>buf && buf!="quit")**

**cout << buf << " " << buf.siz e() << " " << buf.capacity() << '\n';**

**}**

Chuỗi này thiếu nhiều tính năng mà bạn có thể coi là quan trọng hoặc thậm chí là cần thiết. Tuy nhiên, về những gì nó thực hiện thì nó gần giống với std :: string và minh họa các kỹ thuật được sử dụng để triển khai chuỗi thư viện chuẩn.

* 1. **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 tư 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 static, chúng ta chỉ có thể cung cấp cho nó hai thuộc tính đầu tiên. Bằng cách khai báo một hàm không phải thành viên là bạn, chúng ta chỉ có thể cấp cho nó thuộc tính đầu tiên. Có nghĩa là, một hàm friend được khai báo được cấp quyền truy cập vào việc thực thi một lớp giống như một hàm thành viên nhưng không phụ thuộc vào lớp đó.

Ví dụ:

**constexpr rc\_max {4}; // row and column size**

**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 Vector và Ma trận. Điều đó sẽ cho phép các kỹ thuật triển khai 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 kết bạn có thể được đặt trong phần riêng tư hoặc phần công khai của một khai báo lớp; không quan trọng ở đâu. Giống như một hàm thành viên, một hàm friend được khai báo rõ ràng trong phần khai báo của lớp mà nó là bạn. Do đó, nó là một phần của giao diện đó cũng như một chức năng thành viên. Một hàm thành viên của một lớp có thể là bạn của lớp khác. Ví dụ:

**class List\_iterator {**

**// ...**

**int∗ next();**

**};**

**class List {**

**friend int∗ List\_iterator::next();**

**// ...**

**};**

Có một cách viết tắt để biến tất cả các chức năng của lớp này trở thành bạn của lớp khác. Ví dụ:

**class List {**

**friend class List\_iterator;**

**// ...**

**};**

Khai báo một lớp mà một người bạn cấp quyền truy cập vào mọi chức năng của lớp đó. Điều đó ngụ ý rằng chúng ta không thể biết tập hợp các hàm có thể truy cập biểu diễn của lớp cấp chỉ bằng cách nhìn vào chính lớp đó. Trong trường hợp này, khai báo lớp bạn khác với khai báo hàm thành viên và hàm bạn bè. Rõ ràng, các lớp học bạn bè nên được sử dụng một cách thận trọng và chỉ để diễn đạt các khái niệm có mối liên hệ chặt chẽ với nhau.

Có thể biến đối số mẫu thành một Friend:

**template<typename T>**

**class X {**

**friend T;**

**friend class T; // redundant ‘‘class’’**

**// ...**

**};**

Thông thường, có sự lựa chọn giữa việc đặt một lớp thành thành viên (một lớp lồng nhau) hoặc một người bạn không phải thành viên.

**19.4.1 Finding Friends**

Một hàm bạn phải được khai báo trước đó trong một phạm vi bao quanh hoặc được xác định trong phạm vi không phải lớp ngay lập tức bao quanh lớp đang khai báo nó là friend. Các phạm vi bên ngoài phạm vi không gian tên bao quanh trong cùng không được coi là tên được khai báo đầu tiên là friend.

**class C1 { }; // will become friend of N::C**

**void f1(); // will become friend of N::C**

**namespace N {**

**class C2 { }; // will become friend of C**

**void f2() { } // will become friend of C**

**class C {**

**int x;**

**public:**

**friend class C1; // OK (previously defined)**

**friend void f1();**

**friend class C3; // OK (defined in enclosing namespace)**

**friend void f3();**

**friend class C4; // First declared in N and assumed to be in N**

**friend void f4();**

**};**

**class C3 {}; // friend of C**

**void f3() { C x; x.x = 1; } // OK: friend of C**

**} // namespace N**

**class C4 { }; // not friend of N::C**

**void f4() { N::C x; x.x = 1; } // error : x is private and f4() is not a friend of N::C**

Một hàm friend có thể được tìm thấy thông qua các đối số của nó (§14.2.4) ngay cả khi nó không được khai báo trong phạm vi kèm theo ngay lập tức. Ví dụ:

**void f(Matrix& m)**

**{**

**inver t(m); // Matrix’s friend invert()**

**}**

Do đó, một hàm friend nên được khai báo rõ ràng trong một phạm vi bao quanh hoặc lấy một đối số của lớp của nó hoặc một lớp có nguồn gốc từ đó. Nếu không, hàm friend không thể được gọi. Ví dụ:

**// no f() in this scope**

**class X {**

**friend void f(); // useless**

**friend void h(const X&); // can be found through its argument**

**};**

**void g(const X& x)**

**{**

**f(); // no f() in scope**

**h(x); // X’s friend h()**

**}**

**19.4.2 Friends and Members**

Đầu tiên, chúng tôi cố gắng giảm thiểu số lượng các hàm truy cập vào biểu diễn của một lớp và cố gắng làm cho tập 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 nó có thực sự cần quyền truy cập không? ’’ Thông thường, tập hợp các hàm cần truy cập nhỏ hơn so với mức chúng tôi mong muốn lúc đầu. Một số hoạt động phải là thành viên - ví dụ: hàm tạo, hàm hủy và hàm ảo - nhưng thường có một sự lựa chọn. Bởi vì tên thành viên là cục bộ của lớp, một hàm yêu cầu quyền truy cập trực tiếp vào biểu diễn phải là một thành viên trừ khi có lý do cụ thể khiến nó không phải là một thành viên.

Hãy xem xét một lớp X cung cấp các cách thay thế để trình bày một phép toán:

**class X {**

**// ...**

**X(int);**

**int m1(); // member**

**int m2() const;**

**friend int f1(X&); // friend, not member**

**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 nào do người dùng xác định được áp dụng cho toán hạng ngoài cùng bên trái của a hoặc -> . Ví dụ:

**void g()**

**{**

**99.m1(); // error : X(99).m1() not tried**

**99.m2(); // error : X(99).m2() not tried**

**}**

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 liên quan. Tuy nhiên, các chuyển đổi có thể được áp dụng cho các đối số của f2 () và f3 ():

**void h()**

**{**

**f1(99); // error : f1(X(99)) not tried: non-const X& argument**

**f2(99); // OK: f2(X(99)); const X& argument**

**f3(99); // OK: f3(X(99)); X argument**

**}**

Do đó, một hoạt động sửa đổi trạng thái của một đối tượng lớp phải là một thành viên hoặc một hàm nhận 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 thành viên cho các kiểu do người dùng xác định. Ngược lại, nếu muốn chuyển đổi kiểu ngầm định cho tất cả các toán hạng của một phép toán, thì hàm thực thi nó phải là một hàm không phải là một hàm không phải là một hàm có tham số const hoặc một đối số không phải là tham chiếu. Tuy nhiên, các toán tử như vậy thường cần quyền truy cập vào các biểu diễn của lớp toán hạng của chúng. Do đó, các toán tử nhị phân là nguồn phổ biến nhất của các hàm friend.

Trừ khi các chuyển đổi kiểu được xác định, dường như không có lý do thuyết phục nào để chọn một thành viên thay vì một người bạn đang lấy đối số tham chiếu hoặc ngược lại. Trong một số trường hợp, lập trình viên có thể thích một cú pháp gọi hơn cú pháp khác.

Tất cả những thứ khác được coi là bình đẳng, thực hiện các hoạt động cần truy cập trực tiếp vào một biểu diễn dưới dạng các hàm thành viên:

* Không thể biết một ngày nào đó ai đó sẽ xác định một 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 thức 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 các hàm không phải là thành phần.
* Nếu chúng ta đã xác định một thành viên f () và sau đó chúng ta cảm thấy cần phải có một thành viên không phải f (x), chúng ta có thể chỉ cần xác định nó có nghĩa là x.f ().

Ngược lại, các phép toán không cần truy cập trực tiếp vào một biểu diễn thường được biểu diễn tốt nhất dưới dạng các hàm không phải là phần tử, có thể trong một không gian tên làm cho mối quan hệ của chúng với lớp rõ rang.

**19.5 Advice**

[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 số con và để lựa chọn dựa trên nhiều giá trị; §19.2.2.

[3] Sử dụng toán tử -> () để hủy 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 phân bổ 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 các bài tập của riêng bạn; §19.3.

[10] Sử dụng hàm friend nếu bạn cần một hàm nonmember để có quyền truy cập vào biểu diễn của một lớp (ví dụ: để cải thiện ký hiệu hoặc để truy cập 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 một lớp; §19.4.2.