Chapter 19

19. 1 giới thiệu

chồng toán tử không chỉ dành lo phép toán số học và logic
Trên thực tế, toán tử 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 là đặc biệt trong việc lập bản đồ từ việc sử dụng trong mã cho định nghĩa của lập trình viên khác nhau
Một chút từ việc dùng cho các toán tử đơn ngôi thông thường và nhị phân, như +, <, và˜ (§18. 2. 3).Các toán tử [] (subscript) và () (call) là một trong những toán tử do người dùng định nghĩa hữu ích nhất.

19.2.1 Subscripting []

Một hàm toán tử [] có thể được sử dụng để cung cấp ý nghĩa cho các chỉ số con cho các đối tượng lớp. Đối số thứ hai (chỉ số con) của một hàm toán tử [] có thể thuộc bất kỳ kiểu nào. Điều này giúp bạn 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&);
};
```

Sự kết hợp giữa 1 vecto của std::pairs. Việc triển khai sử dụng cùng một phương pháp tìm kiếm tầm thường và không hiệu quả 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 kết hợp 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 đầy đủ hơn. Toán tử [] () phải là một hàm thành viên không tĩnh.

19.2.2 Function Call (gọi hàm)

Lời gọi hàm, nghĩa là, biểu thức ký hiệu (danh sách biểu thức), có thể được hiểu là một phép toán nhị phân với biểu thức là toán hạng bên trái và danh sách biểu thức là toán hạng bên phải. Gọi oán tử (), có thể được nạp chồng theo cách giống như các toán tử khác có thể.

```
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 operator()() việc chồng hàm khi gọi toán tử dường như hữu ích chủ yếu để xác định các kiểu chỉ có một thao tác duy nhất và cho các kiểu mà một thao tác chiếm ưu thế. Gọi toán tử còn được gọi là ứng dụng toán tử.

Rõ ràng nhất và cũng là quan trọng nhất, việc sử dụng toán tử () là để cung cấp cú pháp gọi hàm thông thường 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). Các đối tượng chức năng như vậy cho phép chúng ta viết code lấy các 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, chúng ta có thể định nghĩa một lớp bằng toán tử () () để thêm một giá trị được lưu trữ vào đối số của nó:

```
class Add {
complex val;
public:
Add(complex c) :val{c} { } // lưu giá trị
Add(double r, double i) :val{{r,i}} { }
void operator()(complex& c) const { c += val; } // thêm giá trị cho đối số
};
```

Một đối tượng của class Add được khởi tạo bằng một số bất kỳ và khi được gọi bằng cách sử dụng (), nó sẽ thêm 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});
}
```

Điều này sẽ thêm {2,3} vào mọi phần tử của vectơ và z vào mọi phần tử của danh sách. Lưu ý rằng Thêm {z} xây dựng một đối tượng được sử dụng nhiều lần "for_each(): Add{z}'s operator()()" được gọi cho mỗi phần tử của dãy.

Tất cả đều hoạt động bởi 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++);
return f;
}
```

kỹ thuật này có vẻ bí mật, nhưng nó đơn giản, hiệu quả và cực kỳ hữu ích (§3.4.3, §33.4).

```
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 operator () () là một toán tử chuỗi con và như một toán tử chỉ số con cho mảng nhiều chiều (§29.2.2, §40.5.2).

Một toán tử () () phải là một hàm thành viên không tĩnh.

Các toán tử gọi hàm thường là các mẫu (§29.2.2, §33.5.3).

19.2.3 Dereferencing (toán tử mũi tên)

Được định nghĩa là toán tử hậu tố.Ví dụ:

```
class Ptr {
// ...
X∗ operator−>();
};
```

Các đối tượng của class Ptr có thể được sử dụng để truy cập các thành viên của lớp X theo cách giống với cách con trỏ được sử dụng. Ví dụ:

```
void f(Ptr p)
{
p−>m = 7; // (p.operator->())->m = 7
}
```

Việc biến đổi đối tượng p thành con trỏ p.operator -> () không phụ thuộc vào thành viên m được trỏ tới. Đó là 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
}
```

chúng ta có thể định nghĩa một lớp Disk_ptr để truy cập các đối tượng được lưu trữ trên đĩa. Phương thức khởi tạo của Disk_ptr lấy một tên có thể được sử dụng để tìm đối tượng trên đĩa, Disk_ptr :: operator -> () đưa đối tượng vào bộ nhớ chính khi được truy cập thông qua Disk_ptr của nó và cuối cùng trình hủy của Disk_ptr ghi đối tượng đã cập nhật trở lại đĩa:

```
template<typename T>
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}; // get Disk_ptr for s
p−>name = "Roscoe"; // update s; if necessary, first retrieve from disk
// ...
} // p’s destr uctor wr ites back to disk
```

Đối với con trỏ thông thường, việc sử dụng -> đồng nghĩa với một số cách sử dụng một ngôi ∗ và []. Cho một lớp học

Y mà ->, ∗, và [] có nghĩa mặc định của chúng và một Y ∗ được gọi là p, khi đó:

```
p−>m == (∗p).m // is true
(∗p).m == p[0].m // is true
p−>m == p[0].m // is true
```

Người dùng có thể tự định nghĩa nhưng không đảm bảo như những cung cấp có sẵn:

```
template<typename T>
class Ptr {
Y∗ p;
public:
Y∗ operator−>() { return p; } // dereference to access member
Y& operator∗() { return ∗p; } // dereference to access whole object
Y& operator[](int i) { return p[i]; } // dereference to access element
// ...
};
```

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 (§26.2.1), 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.

19.2.4 Increment and Decrement (tăng giảm)

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: hãy xem xét một chương trình truyền thống rắc rối:

```
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
}
```

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 tôi cũng muốn đảm bảo rằng p có thể được tăng và giảm chỉ khi nó trỏ đến một đối tượng trong một mảng và các phép toán tăng và giảm mang lại một đối tượng trong mảng đó. Đó là, chúng tôi muốn một cái gì đó như thế này:

```
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 đặc biệt nhất trong số các toán tử C ++ ở chỗ chúng có thể được sử dụng như cả 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∗ 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
};
```

Xem xét việc bỏ qua hậu tố ++ và −− trong một thiết kế. Chúng không chỉ kỳ quặc về mặt cú pháp, chúng có xu hướng khó triển khai hơn một chút so với các phiên bản hậu tố, kém hiệu quả hơn và ít thường xuyên hơn đã sử dụng. 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.

```
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(phân bố và giao dịch)

Toán tử new (§11.2.3) lấy bộ nhớ của nó bằng cách gọi một operator new (). Tương tự, toán tử xóa giải phóng bộ nhớ của nó bằng cách gọi một operator delete (). Người dùng có thể định nghĩa lại operator new () và operator delete () hoặc định nghĩa operator new () và operator delete () 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 các phiên bản chung trông giống như sau:

```
void∗ operator new(siz e_t); // sử dụng cho từng đối tượng
void∗ operator new[](siz e_t); // dùng cho mảng
void operator delete(void∗, siz e_t); // sử dụng cho từng đối tượng
void operator delete[](void∗, siz e_t); // dùng cho mảng
// for more versions, see §11.2.4
```

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: chúng ta có thể muốn của một lớp Employee cung cấp một trình phân bổ và phân bổ giao dịch chuyên biệt cho chính nó và tất cả các lớp dẫn xuất của nó:

```
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);
};
```

operator new()s và operator delete()s 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 hủy.

```
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; see §11.2, §11.2.3
// assume p points to s bytes of memory allocated by Employee::operator new()
// and free that memory for reuse
}
}
```

Làm thế nào để một trình biên dịch biết cách cung cấp kích thước phù hợp cho toán tử delete ()? Kiểu được chỉ định trong thao tác xóa khớp với kiểu của đối tượng đang bị xóa. Nếu chúng ta xóa một đối tượng thông qua một con trỏ đến một lớp cơ sở, thì lớp cơ sở đó phải có một virtual(ảo) (§17.2.5) để có kích thước chính xác được cung cấp:

```
Employee∗ p = new Manager; // potential trouble (the exact type is lost)
```

```
// ...
delete p;
```

19.2.6 User-defined Literals(chữ viết)

C ++ cung cấp các ký tự cho nhiều kiểu tích hợp sẵn (§6.2.6):

```
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)
12345678901234567890123456789012345678901234567890x // 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. Tôi sử dụng const để kích hoạt thời gian biên dịch đánh giá. Với những điều đó, chúng ta có thể viết:

```
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}

}
```

Một ký tự số nguyên (§6.2.4.1): được chấp nhận bởi toán tử ký tự lấy một giá trị dài không dấu hoặc đối số const char ∗ hoặc bởi một toán tử ký tự mẫu, ví dụ: 123m hoặc 123456789012345678 90X

Một ký tự dấu phẩy động (§6.2.5.1): được chấp nhận bởi toán tử ký tự lấy một dấu kép dài hoặc một đối số const char ∗ hoặc bởi một toán tử mẫu, chẳng hạn, 123456789012345678 90,976543210x hoặc 3,99 giây

Một chuỗi ký tự (§7.3.2): được chấp nhận bởi một toán tử ký tự nhận một cặp (const char ∗, size_t) của

các đối số, ví dụ: "string" s và R"(Foo\bar)"_path

Ví dụ:

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

{

return Bignum(p);

}

void f(Bignum);

f(12345678901234567890123456789012345678901234567890123456789012345x);
```

Để chuyển một chuỗi C kiểu string từ văn bản nguồn chương trình thành một toán tử chữ, chúng tôi yêu cầu cả hai 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)
```

Một toán tử nghĩa đen chỉ nhận một đối số const char ∗ (và không có kích thước) có thể được áp dụng cho số nguyên và các ký tự dấu phẩy động. Ví dụ:

```
string operator"" SS(const char∗ p); // cảnh báo: điều này sẽ không xảy ra như mong đợi

string s12 = "one two"SS; // error: không có toán tử ký tự áp dụng

string s13 = 13SS; // OK, nhưng tại sao mọi người lại làm như vậy?


template<char...>

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

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 (§28.7.3) và x cho mở rộng), vì vậy các mục đích 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;
```

19.3 A String Class(lớp chuỗi)

19.3.1 Essential Operations (cơ bản)

Class String cung cấp tập hợp thông thường của các hàm tạo, một hàm hủy và các hoạt động gán (§17.1):

```
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. 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 dây. Howev er, 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 Access to Characters

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 các hoạt động được kiểm tra at() operate

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

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 ^= s[i]>>1; // quyền truy cập bỏ chọn vào s
return h;
}
```

Sử dụng at() nơi chúng tôi thấy có thể xảy ra sai lầm. Ví dụ:

```
void print_in_order(const String& s,const vector<int>& index)
{
for (x : index) cout << s.at(x) << '\n';
}
```

19.3.3 Representation

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 {
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);
};
```

19.3.3.1 Ancillary Functions

Hàm đầu tiên như vậy 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;
}
```

Chức năng triển khai thứ hai được sử dụng bởi các hoạt động sao chép để cung cấp cho một string một bản sao của thành viên của người 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;
}
}
```

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 { // lấy các phần tử
ptr = x.ptr;
sz = x.sz;
space = x.space;
x.ptr = x.ch; // x = ""
x.sz = 0;
x.ch[0]=0;
}
}
```

19.3.4 Member Functions

Hàm tạo mặc định xác định một string 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 C kiểu string 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
}
```

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

```
String::String(const String& x) // sao chép cấu trúc uctor
{
copy_from(x); // sao chép biểu diễn từ 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;
}
```

Nhiệm vụ di chuyển Chuỗi sẽ xóa cửa hàng miễn phí của mục tiêu (nếu có) và sau đó di chuyển:

```
String& String::operator=(String&& x)
{
if (this==&x) return ∗this; // deal with self-assignment (x = move(x) is insanity)
if (short_max<sz) delete[] ptr; // delete target
move_from(x); // does not throw
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;
}
```

19.3.6 Using Our String(chuỗi tự định nghĩa)

```
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 (Chương 36) và minh họa các kỹ thuật được sử dụng để triển khai chuỗi thư viện chuẩn.

19.4 Friends(bạn)

chúng ta có thể xác định một toán tử nhân Matrix với một Vector. Đương nhiên, Vector và Matrix ẩn các biểu diễn tương ứng của chúng và cung cấp một tập hợp hoàn chỉnh để thao tác 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 có thể đọc và ghi toàn bộ biểu diễn của cả Matrix và Vectơ. Để tránh điều này, chúng tôi tuyên bố operator∗ friend của cả 2:

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

Có thể coi đối số mẫu là một người bạn:

```
template<typename T>
class X {
friend T;
friend class T; // redundant ‘‘class’’
// ...
};
```