使用 map

一个经典的使用关联数组的例子是单词计数程序：

```
// 统计每个单词在输入中出现的次数
map<string, size_t> word_count;  // string 到 size_t 的空 map
string word;
while (cin >> word)
    ++word_count[word];          // 提取 word 的计数器并将其加 1
for (const auto &w : word_count) // 对 map 中的每个元素
    // 打印结果
    cout << w.first << " occurs " << w.second
         << ((w.second > 1) ? " times" : " time") << endl;
```

此程序读取输入，报告每个单词出现多少次。

类似顺序容器，关联容器也是模板（参见 3.3 节，第 86 页）。为了定义一个 map，我们必须指定关键字和值的类型。在此程序中，map 保存的每个元素中，关键字是 string 类型，值是 size_t 类型（参见 3.5.2 节，第 103 页）。当对 word_count 进行下标操作时，我们使用一个 string 作为下标，获得与此 string 相关联的 size_t 类型的计数器。

while 循环每次从标准输入读取一个单词。它使用每个单词对 word_count 进行下标操作。如果 word 还未在 map 中，下标运算符会创建一个新元素，其关键字为 word，值为 0。不管元素是否是新创建的，我们将其值加 1。

一旦读取完所有输入，范围 for 语句（参见 3.2.3 节，第 81 页）就会遍历 map，打印每个单词和对应的计数器。当从 map 中提取一个元素时，会得到一个 pair 类型的对象，我们将在 11.2.3 节（第 379 页）介绍它。简单来说，pair 是一个模板类型，保存两个名为 first 和 second 的（公有）数据成员。map 所使用的 pair 用 first 成员保存关键字，用 second 成员保存对应的值。因此，输出语句的效果是打印每个单词及其关联的计数器。

如果我们对本节第一段中的文本（指英文版中的文本）运行这个程序，输出将会是：

```
Although occurs 1 time
Before occurs 1 time
an occurs 1 time
and occurs 1 time
...
```

使用 set

422

上一个示例程序的一个合理扩展是：忽略常见单词，如"the"、"and"、"or"等。我们可以使用 set 保存想忽略的单词，只对不在集合中的单词统计出现次数：

```
// 统计输入中每个单词出现的次数
map<string, size_t> word_count; // string 到 size_t 的空 map
set<string> exclude = {"The", "But", "And", "Or", "An", "A",
                       "the", "but", "and", "or", "an", "a"};
string word;
while (cin >> word)
    // 只统计不在 exclude 中的单词
    if (exclude.find(word) == exclude.end())
        ++word_count[word];     // 获取并递增 word 的计数器
```

与其他容器类似，set 也是模板。为了定义一个 set，必须指定其元素类型，本例中是 string。与顺序容器类似，可以对一个关联容器的元素进行列表初始化（参见 9.2.4 节，第 300 页）。集合 exclude 中保存了 12 个我们想忽略的单词。

此程序与前一个程序的重要不同是，在统计每个单词出现次数之前，我们检查单词是否在忽略集合中，这是在 if 语句中完成的：

```
// 只统计不在 exclude 中的单词
if (exclude.find(word) == exclude.end())
```

find 调用返回一个迭代器。如果给定关键字在 set 中，迭代器指向该关键字。否则，find 返回尾后迭代器。在此程序中，仅当 word 不在 exclude 中时我们才更新 word 的计数器。

如果用此程序处理与之前相同的输入，输出将会是：

```
Although occurs 1 time
Before occurs 1 time
are occurs 1 time
as occurs 1 time
...
```

11.1 节练习

练习 11.1：描述 map 和 vector 的不同。

练习 11.2：分别给出最适合使用 list、vector、deque、map 以及 set 的例子。

练习 11.3：编写你自己的单词计数程序。

练习 11.4：扩展你的程序，忽略大小写和标点。例如，"example. "、"example,"和"Example"应该递增相同的计数器。

423

11.2 关联容器概述

关联容器（有序的和无序的）都支持 9.2 节（第 294 页）中介绍的普通容器操作（列于表 9.2，第 295 页）。关联容器不支持顺序容器的位置相关的操作，例如 push_front 或 push_back。原因是关联容器中元素是根据关键字存储的，这些操作对关联容器没有意义。而且，关联容器也不支持构造函数或插入操作这些接受一个元素值和一个数量值的操作。

除了与顺序容器相同的操作之外，关联容器还支持一些顺序容器不支持的操作（参见表 11.7，第 388 页）和类型别名（参见表 11.3，第 381 页）。此外，无序容器还提供一些用来调整哈希性能的操作，我们将在 11.4 节（第 394 页）中介绍。

关联容器的迭代器都是双向的（参见 10.5.1 节，第 365 页）。

11.2.1 定义关联容器

如前所示，当定义一个 map 时，必须既指明关键字类型又指明值类型；而定义一个 set 时，只需指明关键字类型，因为 set 中没有值。每个关联容器都定义了一个默认构

造函数，它创建一个指定类型的空容器。我们也可以将关联容器初始化为另一个同类型容器的拷贝，或是从一个值范围来初始化关联容器，只要这些值可以转化为容器所需类型就可以。在新标准下，我们也可以对关联容器进行值初始化：

C++ 11

```
map<string, size_t> word_count; // 空容器
// 列表初始化
set<string> exclude = {"the", "but", "and", "or", "an", "a",
                       "The", "But", "And", "Or", "An", "A"};
// 三个元素；authors 将姓映射为名
map<string, string> authors = { {"Joyce", "James"},
                                {"Austen", "Jane"},
                                {"Dickens", "Charles"} };
```

与以往一样，初始化器必须能转换为容器中元素的类型。对于 set，元素类型就是关键字类型。

当初始化一个 map 时，必须提供关键字类型和值类型。我们将每个关键字-值对包围在花括号中：

{*key*, *value*}

来指出它们一起构成了 map 中的一个元素。在每个花括号中，关键字是第一个元素，值是第二个。因此，authors 将姓映射到名，初始化后它包含三个元素。

初始化 multimap 或 multiset

一个 map 或 set 中的关键字必须是唯一的，即，对于一个给定的关键字，只能有一个元素的关键字等于它。容器 multimap 和 multiset 没有此限制，它们都允许多个元素具有相同的关键字。例如，在我们用来统计单词数量的 map 中，每个单词只能有一个元素。另一方面，在一个词典中，一个特定单词则可具有多个与之关联的词义。

424

下面的例子展示了具有唯一关键字的容器与允许重复关键字的容器之间的区别。首先，我们将创建一个名为 ivec 的保存 int 的 vector，它包含 20 个元素：0 到 9 每个整数有两个拷贝。我们将使用此 vector 初始化一个 set 和一个 multiset：

```
// 定义一个有 20 个元素的 vector，保存 0 到 9 每个整数的两个拷贝
vector<int> ivec;
for (vector<int>::size_type i = 0; i != 10; ++i) {
    ivec.push_back(i);
    ivec.push_back(i); // 每个数重复保存一次
}
// iset 包含来自 ivec 的不重复的元素；miset 包含所有 20 个元素
set<int> iset(ivec.cbegin(), ivec.cend());
multiset<int> miset(ivec.cbegin(), ivec.cend());
cout << ivec.size() << endl;      // 打印出 20
cout << iset.size() << endl;      // 打印出 10
cout << miset.size() << endl;     // 打印出 20
```

即使我们用整个 ivec 容器来初始化 iset，它也只含有 10 个元素：对应 ivec 中每个不同的元素。另一方面，miset 有 20 个元素，与 ivec 中的元素数量一样多。

11.2.1 节练习

练习 11.5：解释 map 和 set 的区别。你如何选择使用哪个？

练习 11.6：解释 set 和 list 的区别。你如何选择使用哪个？

练习 11.7：定义一个 map，关键字是家庭的姓，值是一个 vector，保存家中孩子（们）的名。编写代码，实现添加新的家庭以及向已有家庭中添加新的孩子。

练习 11.8：编写一个程序，在一个 vector 而不是一个 set 中保存不重复的单词。使用 set 的优点是什么？



11.2.2 关键字类型的要求

关联容器对其关键字类型有一些限制。对于无序容器中关键字的要求，我们将在 11.4 节（第 396 页）中介绍。对于有序容器——map、multimap、set 以及 multiset，关键字类型必须定义元素比较的方法。默认情况下，标准库使用关键字类型的<运算符来比较两个关键字。在集合类型中，关键字类型就是元素类型；在映射类型中，关键字类型是
425 元素的第一部分的类型。因此，11.2 节（第 377 页）中 word_count 的关键字类型是 string。类似的，exclude 的关键字类型也是 string。



传递给排序算法的可调用对象（参见 10.3.1 节，第 344 页）必须满足与关联容器中关键字一样的类型要求。

有序容器的关键字类型

可以向一个算法提供我们自己定义的比较操作（参见 10.3 节，第 344 页），与之类似，也可以提供自己定义的操作来代替关键字上的<运算符。所提供的操作必须在关键字类型上定义一个**严格弱序**（strict weak ordering）。可以将严格弱序看作“小于等于”，虽然实际定义的操作可能是一个复杂的函数。无论我们怎样定义比较函数，它必须具备如下基本性质：

- 两个关键字不能同时“小于等于”对方；如果 k1“小于等于”k2，那么 k2 绝不能“小于等于”k1。
- 如果 k1“小于等于”k2，且 k2“小于等于”k3，那么 k1 必须“小于等于”k3。
- 如果存在两个关键字，任何一个都不“小于等于”另一个，那么我们称这两个关键字是“等价”的。如果 k1“等价于”k2，且 k2“等价于”k3，那么 k1 必须“等价于”k3。

如果两个关键字是等价的（即，任何一个都不“小于等于”另一个），那么容器将它们视作相等来处理。当用作 map 的关键字时，只能有一个元素与这两个关键字关联，我们可以用两者中任意一个来访问对应的值。



在实际编程中，重要的是，如果一个类型定义了“行为正常”的<运算符，则它可以用作关键字类型。

使用关键字类型的比较函数

用来组织一个容器中元素的操作的类型也是该容器类型的一部分。为了指定使用自定义的操作，必须在定义关联容器类型时提供此操作的类型。如前所述，用尖括号指出要定

义哪种类型的容器，自定义的操作类型必须在尖括号中紧跟着元素类型给出。

在尖括号中出现的每个类型，就仅仅是一个类型而已。当我们创建一个容器（对象）时，才会以构造函数参数的形式提供真正的比较操作（其类型必须与在尖括号中指定的类型相吻合）。

例如，我们不能直接定义一个 Sales_data 的 multiset，因为 Sales_data 没有<运算符。但是，可以用 10.3.1 节练习（第 345 页）中的 compareIsbn 函数来定义一个 multiset。此函数在 Sales_data 对象的 ISBN 成员上定义了一个严格弱序。函数 compareIsbn 应该像下面这样定义

```
bool compareIsbn(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() < rhs.isbn();
}
```

426

为了使用自己定义的操作，在定义 multiset 时我们必须提供两个类型：关键字类型 Sales_data，以及比较操作类型——应该是一种函数指针类型（参见 6.7 节，第 221 页），可以指向 compareIsbn。当定义此容器类型的对象时，需要提供想要使用的操作的指针。在本例中，我们提供一个指向 compareIsbn 的指针：

```
// bookstore 中多条记录可以有相同的 ISBN
// bookstore 中的元素以 ISBN 的顺序进行排列
multiset<Sales_data, decltype(compareIsbn)*>
    bookstore(compareIsbn);
```

此处，我们使用 decltype 来指出自定义操作的类型。记住，当用 decltype 来获得一个函数指针类型时，必须加上一个*来指出我们要使用一个给定函数类型的指针（参见 6.7 节，第 223 页）。用 compareIsbn 来初始化 bookstore 对象，这表示当我们向 bookstore 添加元素时，通过调用 compareIsbn 来为这些元素排序。即，bookstore 中的元素将按它们的 ISBN 成员的值排序。可以用 compareIsbn 代替&compareIsbn 作为构造函数的参数，因为当我们使用一个函数的名字时，在需要的情况下它会自动转化为一个指针（参见 6.7 节，第 221 页）。当然，使用&compareIsbn 的效果也是一样的。

11.2.2 节练习

练习 11.9：定义一个 map，将单词与一个行号的 list 关联，list 中保存的是单词所出现的行号。

练习 11.10：可以定义一个 vector<int>::iterator 到 int 的 map 吗？list<int>::iterator 到 int 的 map 呢？对于两种情况，如果不能，解释为什么。

练习 11.11：不使用 decltype 重新定义 bookstore。

11.2.3 pair 类型

在介绍关联容器操作之前，我们需要了解名为 **pair** 的标准库类型，它定义在头文件 utility 中。

一个 pair 保存两个数据成员。类似容器，pair 是一个用来生成特定类型的模板。当创建一个 pair 时，我们必须提供两个类型名，pair 的数据成员将具有对应的类型。两个类型不要求一样：

```
pair<string, string> anon;          // 保存两个 string
pair<string, size_t> word_count;    // 保存一个 string 和一个 size_t
pair<string, vector<int>> line;     // 保存 string 和 vector<int>
```

427 pair 的默认构造函数对数据成员进行值初始化（参见 3.3.1 节，第 88 页）。因此，anon 是一个包含两个空 string 的 pair，line 保存一个空 string 和一个空 vector。word_count 中的 size_t 成员值为 0，而 string 成员被初始化为空。

我们也可以为每个成员提供初始化器：

```
pair<string, string> author{"James", "Joyce"};
```

这条语句创建一个名为 author 的 pair，两个成员被初始化为"James"和"Joyce"。

与其他标准库类型不同，pair 的数据成员是 public 的（参见 7.2 节，第 240 页）。两个成员分别命名为 first 和 second。我们用普通的成员访问符号（参见 1.5.2 节，第 20 页）来访问它们，例如，在第 375 页的单词计数程序的输出语句中我们就是这么做的：

```
// 打印结果
cout << w.first << " occurs " << w.second
     << ((w.second > 1) ? " times" : " time") << endl;
```

此处，w 是指向 map 中某个元素的引用。map 的元素是 pair。在这条语句中，我们首先打印关键字——元素的 first 成员，接着打印计数器——second 成员。标准库只定义了有限的几个 pair 操作，表 11.2 列出了这些操作。

表 11.2：pair 上的操作

`pair<T1, T2> p;`	p 是一个 pair，两个类型分别为 T1 和 T2 的成员都进行了值初始化（参见 3.3.1 节，第 88 页）
`pair<T1, T2> p(v1, v2)`	p 是一个成员类型为 T1 和 T2 的 pair；first 和 second 成员分别用 v1 和 v2 进行初始化
`pair<T1,T2>p = {v1,v2};`	等价于 p (v1,v2)
`make_pair(v1, v2)`	返回一个用 v1 和 v2 初始化的 pair。pair 的类型从 v1 和 v2 的类型推断出来
`p.first`	返回 p 的名为 first 的（公有）数据成员
`p.second`	返回 p 的名为 second 的（公有）数据成员
`p1` *relop* `p2`	关系运算符（<、>、<=、>=）按字典序定义：例如，当 p1.first < p2.first 或!(p2.first < p1.first) && p1.second < p2.second 成立时，p1 < p2 为 true。关系运算利用元素的< 运算符来实现
`p1 == p2`	当 first 和 second 成员分别相等时，两个 pair 相等。相等性判断利用元素的==运算符实现
`p1 != p2`	

创建 pair 对象的函数

C++ 11 想象有一个函数需要返回一个 pair。在新标准下，我们可以对返回值进行列表初始化（参见 6.3.2 节，第 203 页）

428
```
pair<string, int>
process(vector<string> &v)
{
    // 处理 v
```

```
    if (!v.empty())
        return {v.back(), v.back().size()}; // 列表初始化
    else
        return pair<string, int>(); // 隐式构造返回值
}
```

若 v 不为空，我们返回一个由 v 中最后一个 string 及其大小组成的 pair。否则，隐式构造一个空 pair，并返回它。

在较早的C++版本中，不允许用花括号包围的初始化器来返回 pair 这种类型的对象，必须显式构造返回值：

```
if (!v.empty())
    return pair<string, int>(v.back(), v.back().size());
```

我们还可以用 make_pair 来生成 pair 对象，pair 的两个类型来自于 make_pair 的参数：

```
if (!v.empty())
    return make_pair(v.back(), v.back().size());
```

11.2.3 节练习

练习 11.12：编写程序，读入 string 和 int 的序列，将每个 string 和 int 存入一个 pair 中，pair 保存在一个 vector 中。

练习 11.13：在上一题的程序中，至少有三种创建 pair 的方法。编写此程序的三个版本，分别采用不同的方法创建 pair。解释你认为哪种形式最易于编写和理解，为什么？

练习 11.14：扩展你在 11.2.1 节练习（第 378 页）中编写的孩子姓到名的 map，添加一个 pair 的 vector，保存孩子的名和生日。

11.3 关联容器操作

除了表 9.2（第 295 页）中列出的类型，关联容器还定义了表 11.3 中列出的类型。这些类型表示容器关键字和值的类型。

表 11.3：关联容器额外的类型别名

key_type	此容器类型的关键字类型
mapped_type	每个关键字关联的类型；只适用于 map
value_type	对于 set，与 key_type 相同 对于 map，为 pair<const key_type, mapped_type>

对于 set 类型，**key_type** 和 **value_type** 是一样的；set 中保存的值就是关键字。在一个 map 中，元素是关键字-值对。即，每个元素是一个 pair 对象，包含一个关键字和一个关联的值。由于我们不能改变一个元素的关键字，因此这些 pair 的关键字部分是 const 的：

```
set<string>::value_type v1;       // v1 是一个 string
set<string>::key_type v2;         // v2 是一个 string
map<string, int>::value_type v3; // v3 是一个 pair<const string, int>
map<string, int>::key_type v4;    // v4 是一个 string
map<string, int>::mapped_type v5;// v5 是一个 int
```

429

与顺序容器一样（参见 9.2.2 节，第 297 页），我们使用作用域运算符来提取一个类型的成员——例如，map<string, int>::key_type。

只有 map 类型（unordered_map、unordered_multimap、multimap 和 map）才定义了 **mapped_type**。

11.3.1 关联容器迭代器

当解引用一个关联容器迭代器时，我们会得到一个类型为容器的 value_type 的值的引用。对 map 而言，value_type 是一个 pair 类型，其 first 成员保存 const 的关键字，second 成员保存值：

```
// 获得指向 word_count 中一个元素的迭代器
auto map_it = word_count.begin();
// *map_it 是指向一个 pair<const string, size_t>对象的引用
cout << map_it->first;          // 打印此元素的关键字
cout << " " << map_it->second;  // 打印此元素的值
map_it->first = "new key";      // 错误：关键字是 const 的
++map_it->second; // 正确：我们可以通过迭代器改变元素
```



必须记住，一个 map 的 value_type 是一个 pair，我们可以改变 pair 的值，但不能改变关键字成员的值。

set 的迭代器是 const 的

虽然 set 类型同时定义了 iterator 和 const_iterator 类型，但两种类型都只允许只读访问 set 中的元素。与不能改变一个 map 元素的关键字一样，一个 set 中的关键字也是 const 的。可以用一个 set 迭代器来读取元素的值，但不能修改：

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
set<int>::iterator set_it = iset.begin();
if (set_it != iset.end()) {
    *set_it = 42;              // 错误：set 中的关键字是只读的
    cout << *set_it << endl; // 正确：可以读关键字
}
```

430

遍历关联容器

map 和 set 类型都支持表 9.2（第 295 页）中的 begin 和 end 操作。与往常一样，我们可以用这些函数获取迭代器，然后用迭代器来遍历容器。例如，我们可以编写一个循环来打印第 375 页中单词计数程序的结果，如下所示：

```
// 获得一个指向首元素的迭代器
auto map_it = word_count.cbegin();
// 比较当前迭代器和尾后迭代器
while (map_it != word_count.cend()) {
    // 解引用迭代器，打印关键字-值对
    cout << map_it->first << " occurs "
         << map_it->second << " times" << endl;
    ++map_it; // 递增迭代器，移动到下一个元素
}
```

while 的循环条件和循环中的迭代器递增操作看起来很像我们之前编写的打印一个 vector

或一个 string 的程序。我们首先初始化迭代器 map_it，让它指向 word_count 中的首元素。只要迭代器不等于 end，就打印当前元素并递增迭代器。输出语句解引用 map_it 来获得 pair 的成员，否则与我们之前的程序一样。



本程序的输出是按字典序排列的。当使用一个迭代器遍历一个 map、multimap、set 或 multiset 时，迭代器按关键字升序遍历元素。

关联容器和算法

我们通常不对关联容器使用泛型算法（参见第 10 章）。关键字是 const 这一特性意味着不能将关联容器传递给修改或重排容器元素的算法，因为这类算法需要向元素写入值，而 set 类型中的元素是 const 的，map 中的元素是 pair，其第一个成员是 const 的。

关联容器可用于只读取元素的算法。但是，很多这类算法都要搜索序列。由于关联容器中的元素不能通过它们的关键字进行（快速）查找，因此对其使用泛型搜索算法几乎总是个坏主意。例如，我们将在 11.3.5 节（第 388 页）中看到，关联容器定义了一个名为 find 的成员，它通过一个给定的关键字直接获取元素。我们可以用泛型 find 算法来查找一个元素，但此算法会进行顺序搜索。使用关联容器定义的专用的 find 成员会比调用泛型 find 快得多。

在实际编程中，如果我们真要对一个关联容器使用算法，要么是将它当作一个源序列，要么当作一个目的位置。例如，可以用泛型 copy 算法将元素从一个关联容器拷贝到另一个序列。类似的，可以调用 inserter 将一个插入器绑定（参见 10.4.1 节，第 358 页）到一个关联容器。通过使用 inserter，我们可以将关联容器当作一个目的位置来调用另一个算法。

11.3.1 节练习

431

练习 11.15：对一个 int 到 vector<int>的 map，其 mapped_type、key_type 和 value_type 分别是什么？

练习 11.16：使用一个 map 迭代器编写一个表达式，将一个值赋予一个元素。

练习 11.17：假定 c 是一个 string 的 multiset，v 是一个 string 的 vector，解释下面的调用。指出每个调用是否合法：

```
copy(v.begin(), v.end(), inserter(c, c.end()));
copy(v.begin(), v.end(), back_inserter(c));
copy(c.begin(), c.end(), inserter(v, v.end()));
copy(c.begin(), c.end(), back_inserter(v));
```

练习 11.18：写出第 382 页循环中 map_it 的类型，不要使用 auto 或 decltype。

练习 11.19：定义一个变量，通过对 11.2.2 节（第 378 页）中的名为 bookstore 的 multiset 调用 begin()来初始化这个变量。写出变量的类型，不要使用 auto 或 decltype。

11.3.2 添加元素

关联容器的 insert 成员（见表 11.4，第 384 页）向容器中添加一个元素或一个元素范围。由于 map 和 set（以及对应的无序类型）包含不重复的关键字，因此插入一个已

存在的元素对容器没有任何影响：

```
vector<int> ivec = {2,4,6,8,2,4,6,8};          // ivec 有 8 个元素
set<int> set2;                                  // 空集合
set2.insert(ivec.cbegin(), ivec.cend());        // set2 有 4 个元素
set2.insert({1,3,5,7,1,3,5,7});                 // set2 现在有 8 个元素
```

insert 有两个版本，分别接受一对迭代器，或是一个初始化器列表，这两个版本的行为类似对应的构造函数（参见 11.2.1 节，第 376 页）——对于一个给定的关键字，只有第一个带此关键字的元素才被插入到容器中。

向 map 添加元素

对一个 map 进行 insert 操作时，必须记住元素类型是 pair。通常，对于想要插入的数据，并没有一个现成的 pair 对象。可以在 insert 的参数列表中创建一个 pair：

```
// 向 word_count 插入 word 的 4 种方法
word_count.insert({word, 1});
word_count.insert(make_pair(word, 1));
word_count.insert(pair<string, size_t>(word, 1));
word_count.insert(map<string, size_t>::value_type(word, 1));
```

C++ 11 如我们所见，在新标准下，创建一个 pair 最简单的方法是在参数列表中使用花括号初始化。也可以调用 make_pair 或显式构造 pair。最后一个 insert 调用中的参数：

432

```
map<string, size_t>::value_type(s, 1)
```

构造一个恰当的 pair 类型，并构造该类型的一个新对象，插入到 map 中。

表 11.4：关联容器 insert 操作

c.insert(v) c.emplace(*args*)	v 是 value_type 类型的对象；*args* 用来构造一个元素 对于 map 和 set，只有当元素的关键字不在 c 中时才插入（或构造）元素。函数返回一个 pair，包含一个迭代器，指向具有指定关键字的元素，以及一个指示插入是否成功的 bool 值。 对于 multimap 和 multiset，总会插入（或构造）给定元素，并返回一个指向新元素的迭代器
c.insert(b, e) c.insert(il)	b 和 e 是迭代器，表示一个 c::value_type 类型值的范围；il 是这种值的花括号列表。函数返回 void 对于 map 和 set，只插入关键字不在 c 中的元素。对于 multimap 和 multiset，则会插入范围中的每个元素
c.insert(p, v) c.emplace(p, *args*)	类似 insert(v)（或 emplace(*args*)），但将迭代器 p 作为一个提示，指出从哪里开始搜索新元素应该存储的位置。返回一个迭代器，指向具有给定关键字的元素

检测 insert 的返回值

insert（或 emplace）返回的值依赖于容器类型和参数。对于不包含重复关键字的容器，添加单一元素的 insert 和 emplace 版本返回一个 pair，告诉我们插入操作是否成功。pair 的 first 成员是一个迭代器，指向具有给定关键字的元素；second 成员是一个 bool 值，指出元素是插入成功还是已经存在于容器中。如果关键字已在容器中，则 insert 什么事情也不做，且返回值中的 bool 部分为 false。如果关键字不存在，元

素被插入容器中，且 bool 值为 true。

作为一个例子，我们用 insert 重写单词计数程序：

```
// 统计每个单词在输入中出现次数的一种更烦琐的方法
map<string, size_t> word_count; // 从 string 到 size_t 的空 map
string word;
while (cin >> word) {
    // 插入一个元素，关键字等于 word，值为 1;
    // 若 word 已在 word_count 中，insert 什么也不做
    auto ret = word_count.insert({word, 1});
    if (!ret.second)              // word 已在 word_count 中
        ++ret.first->second;      // 递增计数器
}
```

对于每个 word，我们尝试将其插入到容器中，对应的值为 1。若 word 已在 map 中，则什么都不做，特别是与 word 相关联的计数器的值不变。若 word 还未在 map 中，则此 string 对象被添加到 map 中，且其计数器的值被置为 1。 433

if 语句检查返回值的 bool 部分，若为 false，则表明插入操作未发生。在此情况下，word 已存在于 word_count 中，因此必须递增此元素所关联的计数器。

展开递增语句

在这个版本的单词计数程序中，递增计数器的语句很难理解。通过添加一些括号来反映出运算符的优先级（参见 4.1.2 节，第 121 页），会使表达式更容易理解一些：

```
++((ret.first)->second); // 等价的表达式
```

下面我们一步一步来解释此表达式：

ret 保存 insert 返回的值，是一个 pair。

ret.first 是 pair 的第一个成员，是一个 map 迭代器，指向具有给定关键字的元素。

ret.first-> 解引用此迭代器，提取 map 中的元素，元素也是一个 pair。

ret.first->second map 中元素的值部分。

++ret.first->second 递增此值。

再回到原来完整的递增语句，它提取匹配关键字 word 的元素的迭代器，并递增与我们试图插入的关键字相关联的计数器。

如果读者使用的是旧版本的编译器，或者是在阅读新标准推出之前编写的代码，ret 的声明和初始化可能复杂些：

```
pair<map<string, size_t>::iterator, bool> ret =
            word_count.insert(make_pair(word, 1));
```

应该容易看出这条语句定义了一个 pair，其第二个类型为 bool 类型。第一个类型理解起来有点儿困难，它是一个在 map<string, size_t>类型上定义的 iterator 类型。

向 multiset 或 multimap 添加元素

我们的单词计数程序依赖于这样一个事实：一个给定的关键字只能出现一次。这样，任意给定的单词只有一个关联的计数器。我们有时希望能添加具有相同关键字的多个元素。例如，可能想建立作者到他所著书籍题目的映射。在此情况下，每个作者可能有多个

条目，因此我们应该使用 multimap 而不是 map。由于一个 multi 容器中的关键字不必唯一，在这些类型上调用 insert 总会插入一个元素：

434
```
multimap<string, string> authors;
// 插入第一个元素，关键字为 Barth, John
authors.insert({"Barth, John", "Sot-Weed Factor"});
// 正确：添加第二个元素，关键字也是 Barth, John
authors.insert({"Barth, John", "Lost in the Funhouse"});
```

对允许重复关键字的容器，接受单个元素的 insert 操作返回一个指向新元素的迭代器。这里无须返回一个 bool 值，因为 insert 总是向这类容器中加入一个新元素。

11.3.2 节练习

练习 11.20：重写 11.1 节练习（第 376 页）的单词计数程序，使用 insert 代替下标操作。你认为哪个程序更容易编写和阅读？解释原因。

练习 11.21：假定 word_count 是一个 string 到 size_t 的 map，word 是一个 string，解释下面循环的作用：

```
while (cin >> word)
    ++word_count.insert({word, 0}).first->second;
```

练习 11.22：给定一个 map<string, vector<int>>，对此容器的插入一个元素的 insert 版本，写出其参数类型和返回类型。

练习 11.23：11.2.1 节练习（第 378 页）中的 map 以孩子的姓为关键字，保存他们的名的 vector，用 multimap 重写此 map。

11.3.3 删除元素

关联容器定义了三个版本的 erase，如表 11.5 所示。与顺序容器一样，我们可以通过传递给 erase 一个迭代器或一个迭代器对来删除一个元素或者一个元素范围。这两个版本的 erase 与对应的顺序容器的操作非常相似：指定的元素被删除，函数返回 void。

关联容器提供一个额外的 erase 操作，它接受一个 key_type 参数。此版本删除所有匹配给定关键字的元素（如果存在的话），返回实际删除的元素的数量。我们可以用此版本在打印结果之前从 word_count 中删除一个特定的单词：

```
// 删除一个关键字，返回删除的元素数量
if (word_count.erase(removal_word))
    cout << "ok: " << removal_word << " removed\n";
else cout << "oops: " << removal_word << " not found!\n";
```

对于保存不重复关键字的容器，erase 的返回值总是 0 或 1。若返回值为 0，则表明想要删除的元素并不在容器中

435 对允许重复关键字的容器，删除元素的数量可能大于 1：

```
auto cnt = authors.erase("Barth, John");
```

如果 authors 是我们在 11.3.2 节（第 386 页）中创建的 multimap，则 cnt 的值为 2。

表 11.5：从关联容器删除元素	
c.erase(k)	从 c 中删除每个关键字为 k 的元素。返回一个 size_type 值，指出删除的元素的数量
c.erase(p)	从 c 中删除迭代器 p 指定的元素。p 必须指向 c 中一个真实元素，不能等于 c.end()。返回一个指向 p 之后元素的迭代器，若 p 指向 c 中的尾元素，则返回 c.end()
c.erase(b, e)	删除迭代器对 b 和 e 所表示的范围中的元素。返回 e

11.3.4 map 的下标操作



map 和 unordered_map 容器提供了下标运算符和一个对应的 at 函数（参见 9.3.2 节，第 311 页），如表 11.6 所示。set 类型不支持下标，因为 set 中没有与关键字相关联的“值”。元素本身就是关键字，因此“获取与一个关键字相关联的值”的操作就没有意义了。我们不能对一个 multimap 或一个 unordered_multimap 进行下标操作，因为这些容器中可能有多个值与一个关键字相关联。

类似我们用过的其他下标运算符，map 下标运算符接受一个索引（即，一个关键字），获取与此关键字相关联的值。但是，与其他下标运算符不同的是，如果关键字并不在 map 中，会为它创建一个元素并插入到 map 中，关联值将进行值初始化（参见 3.3.1 节，第 88 页）。

例如，如果我们编写如下代码

```
map <string, size_t> word_count; // empty map
// 插入一个关键字为 Anna 的元素，关联值进行值初始化；然后将 1 赋予它
word_count["Anna"] = 1;
```

将会执行如下操作：

- 在 word_count 中搜索关键字为 Anna 的元素，未找到。
- 将一个新的关键字-值对插入到 word_count 中。关键字是一个 const string，保存 Anna。值进行值初始化，在本例中意味着值为 0。
- 提取出新插入的元素，并将值 1 赋予它。

由于下标运算符可能插入一个新元素，我们只可以对非 const 的 map 使用下标操作。 436



对一个 map 使用下标操作，其行为与数组或 vector 上的下标操作很不相同：使用一个不在容器中的关键字作为下标，会添加一个具有此关键字的元素到 map 中。

表 11.6：map 和 unordered_map 的下标操作	
c[k]	返回关键字为 k 的元素；如果 k 不在 c 中，添加一个关键字为 k 的元素，对其进行值初始化
c.at(k)	访问关键字为 k 的元素，带参数检查；若 k 不在 c 中，抛出一个 out_of_range 异常（参见 5.6 节，第 173 页）

使用下标操作的返回值

map 的下标运算符与我们用过的其他下标运算符的另一个不同之处是其返回类型。通

常情况下，解引用一个迭代器所返回的类型与下标运算符返回的类型是一样的。但对 map 则不然：当对一个 map 进行下标操作时，会获得一个 mapped_type 对象；但当解引用一个 map 迭代器时，会得到一个 value_type 对象（参见 11.3 节，第 381 页）。

与其他下标运算符相同的是，map 的下标运算符返回一个左值（参见 4.1.1 节，第 121 页）。由于返回的是一个左值，所以我们既可以读也可以写元素：

```
cout << word_count["Anna"];      // 用 Anna 作为下标提取元素；会打印出 1
++word_count["Anna"];            // 提取元素，将其增 1
cout << word_count["Anna"];      // 提取元素并打印它；会打印出 2
```



与 vector 与 string 不同，map 的下标运算符返回的类型与解引用 map 迭代器得到的类型不同。

如果关键字还未在 map 中，下标运算符会添加一个新元素，这一特性允许我们编写出异常简洁的程序，例如单词计数程序中的循环（参见 11.1 节，第 375 页）。另一方面，有时只是想知道一个元素是否已在 map 中，但在不存在时并不想添加元素。在这种情况下，就不能使用下标运算符。

11.3.4 节练习

练习 11.24：下面的程序完成什么功能？

```
map<int, int> m;
m[0] = 1;
```

练习 11.25：对比下面程序与上一题程序

```
vector<int> v;
v[0] = 1;
```

练习 11.26：可以用什么类型来对一个 map 进行下标操作？下标运算符返回的类型是什么？请给出一个具体例子——即，定义一个 map，然后写出一个可以用来对 map 进行下标操作的类型以及下标运算符将会返回的类型。

11.3.5 访问元素

关联容器提供多种查找一个指定元素的方法，如表 11.7 所示。应该使用哪个操作依赖于我们要解决什么问题。如果我们所关心的只不过是一个特定元素是否已在容器中，可能 find 是最佳选择。对于不允许重复关键字的容器，可能使用 find 还是 count 没什么区别。但对于允许重复关键字的容器，count 还会做更多的工作：如果元素在容器中，它还会统计有多少个元素有相同的关键字。如果不需要计数，最好使用 find：

437

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
iset.find(1);    // 返回一个迭代器，指向 key == 1 的元素
iset.find(11);   // 返回一个迭代器，其值等于 iset.end()
iset.count(1);   // 返回 1
iset.count(11);  // 返回 0
```

表 11.7：在一个关联容器中查找元素的操作

lower_bound 和 upper_bound 不适用于无序容器。
下标和 at 操作只适用于非 const 的 map 和 unordered_map。

续表

c.find(k)	返回一个迭代器，指向第一个关键字为 k 的元素，若 k 不在容器中，则返回尾后迭代器
c.count(k)	返回关键字等于 k 的元素的数量。对于不允许重复关键字的容器，返回值永远是 0 或 1
c.lower_bound(k)	返回一个迭代器，指向第一个关键字不小于 k 的元素
c.upper_bound(k)	返回一个迭代器，指向第一个关键字大于 k 的元素
c.equal_range(k)	返回一个迭代器 pair，表示关键字等于 k 的元素的范围。若 k 不存在，pair 的两个成员均等于 c.end()

对 map 使用 find 代替下标操作

对 map 和 unordered_map 类型，下标运算符提供了最简单的提取元素的方法。但是，如我们所见，使用下标操作有一个严重的副作用：如果关键字还未在 map 中，下标操作会插入一个具有给定关键字的元素。这种行为是否正确完全依赖于我们的预期是什么。例如，单词计数程序依赖于这样一个特性：使用一个不存在的关键字作为下标，会插入一个新元素，其关键字为给定关键字，其值为 0。也就是说，下标操作的行为符合我们的预期。

但有时，我们只是想知道一个给定关键字是否在 map 中，而不想改变 map。这样就不能使用下标运算符来检查一个元素是否存在，因为如果关键字不存在的话，下标运算符会插入一个新元素。在这种情况下，应该使用 find：

```
if (word_count.find("foobar") == word_count.end())
    cout << "foobar is not in the map" << endl;
```

在 multimap 或 multiset 中查找元素

在一个不允许重复关键字的关联容器中查找一个元素是一件很简单的事情——元素要么在容器中，要么不在。但对于允许重复关键字的容器来说，过程就更为复杂：在容器中可能有很多元素具有给定的关键字。如果一个 multimap 或 multiset 中有多个元素具有给定关键字，则这些元素在容器中会相邻存储。

例如，给定一个从作者到著作题目的映射，我们可能想打印一个特定作者的所有著作。 438
可以用三种不同方法来解决这个问题。最直观的方法是使用 find 和 count：

```
string search_item("Alain de Botton");          // 要查找的作者
auto entries = authors.count(search_item);      // 元素的数量
auto iter = authors.find(search_item);          // 此作者的第一本书
// 用一个循环查找此作者的所有著作
while(entries) {
    cout << iter->second << endl;               // 打印每个题目
        ++iter;                                 // 前进到下一本书
        --entries;                              // 记录已经打印了多少本书
}
```

首先调用 count 确定此作者共有多少本著作，并调用 find 获得一个迭代器，指向第一个关键字为此作者的元素。for 循环的迭代次数依赖于 count 的返回值。特别是，如果 count 返回 0，则循环一次也不执行。



当我们遍历一个 multimap 或 multiset 时，保证可以得到序列中所有具有给定关键字的元素。

一种不同的，面向迭代器的解决方法

我们还可以用 lower_bound 和 upper_bound 来解决此问题。这两个操作都接受一个关键字，返回一个迭代器。如果关键字在容器中，lower_bound 返回的迭代器将指向第一个具有给定关键字的元素，而 upper_bound 返回的迭代器则指向最后一个匹配给定关键字的元素之后的位置。如果元素不在 multimap 中，则 lower_bound 和 upper_bound 会返回相等的迭代器——指向一个不影响排序的关键字插入位置。因此，用相同的关键字调用 lower_bound 和 upper_bound 会得到一个迭代器范围(参见 9.2.1 节，第 296 页)，表示所有具有该关键字的元素的范围。

439 当然，这两个操作返回的迭代器可能是容器的尾后迭代器。如果我们查找的元素具有容器中最大的关键字，则此关键字的 upper_bound 返回尾后迭代器。如果关键字不存在，且大于容器中任何关键字，则 lower_bound 返回的也是尾后迭代器。



lower_bound 返回的迭代器可能指向一个具有给定关键字的元素，但也可能不指向。如果关键字不在容器中，则 lower_bound 会返回关键字的第一个安全插入点——不影响容器中元素顺序的插入位置。

使用这两个操作，我们可以重写前面的程序：

```
// authors 和 search_item 的定义，与前面的程序一样
// beg 和 end 表示对应此作者的元素的范围
for (auto beg = authors.lower_bound(search_item),
          end = authors.upper_bound(search_item);
     beg != end; ++beg)
    cout << beg->second << endl; // 打印每个题目
```

此程序与使用 count 和 find 的版本完成相同的工作，但更直接。对 lower_bound 的调用将 beg 定位到第一个与 search_item 匹配的元素（如果存在的话）。如果容器中没有这样的元素，beg 将指向第一个关键字大于 search_item 的元素，有可能是尾后迭代器。upper_bound 调用将 end 指向最后一个匹配指定关键字的元素之后的元素。这两个操作并不报告关键字是否存在，重要的是它们的返回值可作为一个迭代器范围（参见 9.2.1 节，第 296 页）。

如果没有元素与给定关键字匹配，则 lower_bound 和 upper_bound 会返回相等的迭代器——都指向给定关键字的插入点，能保持容器中元素顺序的插入位置。

假定有多个元素与给定关键字匹配，beg 将指向其中第一个元素。我们可以通过递增 beg 来遍历这些元素。end 中的迭代器会指出何时完成遍历——当 beg 等于 end 时，就表明已经遍历了所有匹配给定关键字的元素了。

由于这两个迭代器构成一个范围，我们可以用一个 for 循环来遍历这个范围。循环可能执行零次，如果存在给定作者的话，就会执行多次，打印出该作者的所有项。如果给定作者不存在，beg 和 end 相等，循环就一次也不会执行。否则，我们知道递增 beg 最终会使它到达 end，在此过程中我们就会打印出与此作者关联的每条记录。



如果 lower_bound 和 upper_bound 返回相同的迭代器，则给定关键字不在容器中。

equal_range 函数

解决此问题的最后一种方法是三种方法中最直接的：不必再调用 upper_bound 和

lower_bound，直接调用 equal_range 即可。此函数接受一个关键字，返回一个迭代器 pair。若关键字存在，则第一个迭代器指向第一个与关键字匹配的元素，第二个迭代器指向最后一个匹配元素之后的位置。若未找到匹配元素，则两个迭代器都指向关键字可以插入的位置。 440

可以用 equal_range 来再次修改我们的程序：

```
// authors 和 search_item 的定义，与前面的程序一样
// pos 保存迭代器对，表示与关键字匹配的元素范围
for (auto pos = authors.equal_range(search_item);
     pos.first != pos.second; ++pos.first)
    cout << pos.first->second << endl; // 打印每个题目
```

此程序本质上与前一个使用 upper_bound 和 lower_bound 的程序是一样的。不同之处就是，没有用局部变量 beg 和 end 来保存元素范围，而是使用了 equal_range 返回的 pair。此 pair 的 first 成员保存的迭代器与 lower_bound 返回的迭代器是一样的，second 保存的迭代器与 upper_bound 的返回值是一样的。因此，在此程序中，pos.first 等价于 beg，pos.second 等价于 end。

11.3.5 节练习

练习 11.27：对于什么问题你会使用 count 来解决？什么时候你又会选择 find 呢？

练习 11.28：对一个 string 到 int 的 vector 的 map，定义并初始化一个变量来保存在其上调用 find 所返回的结果。

练习 11.29：如果给定的关键字不在容器中，upper_bound、lower_bound 和 equal_range 分别会返回什么？

练习 11.30：对于本节最后一个程序中的输出表达式，解释运算对象 pos.first->second 的含义。

练习 11.31：编写程序，定义一个作者及其作品的 multimap。使用 find 在 multimap 中查找一个元素并用 erase 删除它。确保你的程序在元素不在 map 中时也能正常运行。

练习 11.32：使用上一题定义的 multimap 编写一个程序，按字典序打印作者列表和他们的作品。

11.3.6 一个单词转换的 map

我们将以一个程序结束本节的内容，它将展示 map 的创建、搜索以及遍历。这个程序的功能是这样的：给定一个 string，将它转换为另一个 string。程序的输入是两个文件。第一个文件保存的是一些规则，用来转换第二个文件中的文本。每条规则由两部分组成：一个可能出现在输入文件中的单词和一个用来替换它的短语。表达的含义是，每当第一个单词出现在输入中时，我们就将它替换为对应的短语。第二个输入文件包含要转换的文本。

如果单词转换文件的内容如下所示： 441

```
brb be right back
k okay?
y why
r are
```

```
u you
pic picture
thk thanks!
l8r later
```

我们希望转换的文本为

```
where r u
y dont u send me a pic
k thk l8r
```

则程序应该生成这样的输出：

```
where are you
why dont you send me a picture
okay? thanks! later
```

单词转换程序

我们的程序将使用三个函数。函数 word_transform 管理整个过程。它接受两个 ifstream 参数：第一个参数应绑定到单词转换文件，第二个参数应绑定到我们要转换的文本文件。函数 buildMap 会读取转换规则文件，并创建一个 map，用于保存每个单词到其转换内容的映射。函数 transform 接受一个 string，如果存在转换规则，返回转换后的内容。

我们首先定义 word_transform 函数。最重要的部分是调用 buildMap 和 transform：

```
void word_transform(ifstream &map_file, ifstream &input)
{
    auto trans_map = buildMap(map_file); // 保存转换规则
    string text;                         // 保存输入中的每一行
    while (getline(input, text)) {       // 读取一行输入
        istringstream stream(text);      // 读取每个单词
        string word;
        bool firstword = true;           // 控制是否打印空格
        while (stream >> word) {
            if (firstword)
                firstword = false;
            else
                cout << " ";             // 在单词间打印一个空格
            // transform返回它的第一个参数或其转换之后的形式
            cout << transform(word, trans_map); // 打印输出
        }
        cout << endl;                    // 完成一行的转换
    }
}
```

442 函数首先调用 buildMap 来生成单词转换 map，我们将它保存在 trans_map 中。函数的剩余部分处理输入文件。while 循环用 getline 一行一行地读取输入文件。这样做的目的是使得输出中的换行位置能和输入文件中一样。为了从每行中获取单词，我们使用了一个嵌套的 while 循环，它用一个 istringstream（参见 8.3 节，第 287 页）来处理当前行中的每个单词。

在输出过程中，内层 while 循环使用一个 bool 变量 firstword 来确定是否打印

一个空格。它通过调用 transform 来获得要打印的单词。transform 的返回值或者是 word 中原来的 string，或者是 trans_map 中指出的对应的转换内容。

建立转换映射

函数 buildMap 读入给定文件，建立起转换映射。

```
map<string, string> buildMap(ifstream &map_file)
{
    map<string, string> trans_map; // 保存转换规则
    string key;      // 要转换的单词
    string value;    // 替换后的内容
    // 读取第一个单词存入 key 中，行中剩余内容存入 value
    while (map_file >> key && getline(map_file, value))
        if (value.size() > 1) // 检查是否有转换规则
            trans_map[key] = value.substr(1); // 跳过前导空格
        else
            throw runtime_error("no rule for " + key);
    return trans_map;
}
```

map_file 中的每一行对应一条规则。每条规则由一个单词和一个短语组成，短语可能包含多个单词。我们用>>读取要转换的单词，存入 key 中，并调用 getline 读取这一行中的剩余内容存入 value。由于 getline 不会跳过前导空格（参见 3.2.2 节，第 78 页），需要我们来跳过单词和它的转换内容之间的空格。在保存转换规则之前，检查是否获得了一个以上的字符。如果是，调用 substr（参见 9.5.1 节，第 321 页）来跳过分隔单词及其转换短语之间的前导空格，并将得到的子字符串存入 trans_map。

注意，我们使用下标运算符来添加关键字-值对。我们隐含地忽略了一个单词在转换文件中出现多次的情况。如果真的有单词出现多次，循环会将最后一个对应短语存入 trans_map。当 while 循环结束后，trans_map 中将保存着用来转换输入文本的规则。

生成转换文本

函数 transform 进行实际的转换工作。其参数是需要转换的 string 的引用和转换规则 map。如果给定 string 在 map 中，transform 返回相应的短语。否则，transform 直接返回原 string：

```
const string &
transform(const string &s, const map<string, string> &m)
{
    // 实际的转换工作；此部分是程序的核心
    auto map_it = m.find(s);
    // 如果单词在转换规则 map 中
    if (map_it != m.cend())
        return map_it->second;    // 使用替换短语
    else
        return s;                 // 否则返回原 string
}
```

443

函数首先调用 find 来确定给定 string 是否在 map 中。如果存在，则 find 返回一个指向对应元素的迭代器。否则，find 返回尾后迭代器。如果元素存在，我们解引用迭代器，获得一个保存关键字和值的 pair（参见 11.3 节，第 381 页），然后返回成员 second，即

用来替代 s 的内容。

11.3.6 节练习

练习 11.33：实现你自己版本的单词转换程序。

练习 11.34：如果你将 transform 函数中的 find 替换为下标运算符，会发生什么情况？

练习 11.35：在 buildMap 中，如果进行如下改写，会有什么效果？

```
    trans_map[key] = value.substr(1);
改为 trans_map.insert({key, value.substr(1)})
```

练习 11.36：我们的程序并没有检查输入文件的合法性。特别是，它假定转换规则文件中的规则都是有意义的。如果文件中的某一行包含一个关键字、一个空格，然后就结束了，会发生什么？预测程序的行为并进行验证，再与你的程序进行比较。

11.4 无序容器

新标准定义了 4 个**无序关联容器**（unordered associative container）。这些容器不是使用比较运算符来组织元素，而是使用一个哈希函数（hash function）和关键字类型的==运算符。在关键字类型的元素没有明显的序关系的情况下，无序容器是非常有用的。在某些应用中，维护元素的序代价非常高昂，此时无序容器也很有用。

C++ 11

虽然理论上哈希技术能获得更好的平均性能，但在实际中想要达到很好的效果还需要进行一些性能测试和调优工作。因此，使用无序容器通常更为简单（通常也会有更好的性能）。

444



如果关键字类型固有就是无序的，或者性能测试发现问题可以用哈希技术解决，就可以使用无序容器。

使用无序容器

除了哈希管理操作之外，无序容器还提供了与有序容器相同的操作（find、insert 等）。这意味着我们曾用于 map 和 set 的操作也能用于 unordered_map 和 unordered_set。类似的，无序容器也有允许重复关键字的版本。

因此，通常可以用一个无序容器替换对应的有序容器，反之亦然。但是，由于元素未按顺序存储，一个使用无序容器的程序的输出（通常）会与使用有序容器的版本不同。

例如，可以用 unordered_map 重写最初的单词计数程序（参见 11.1 节，第 375 页）：

```
// 统计出现次数，但单词不会按字典序排列
unordered_map<string, size_t> word_count;
string word;
while (cin >> word)
    ++word_count[word];           // 提取并递增 word 的计数器
for (const auto &w : word_count) // 对 map 中的每个元素
    // 打印结果
    cout << w.first << " occurs " << w.second
         << ((w.second > 1) ? " times" : " time") << endl;
```

此程序与原程序的唯一区别是 word_count 的类型。如果在相同的输入数据上运行此版本，会得到这样的输出：

```
containers. occurs 1 time
use occurs 1 time
can occurs 1 time
examples occurs 1 time
. . .
```

对于每个单词，我们将得到相同的计数结果。但单词不太可能按字典序输出。

管理桶

无序容器在存储上组织为一组桶，每个桶保存零个或多个元素。无序容器使用一个哈希函数将元素映射到桶。为了访问一个元素，容器首先计算元素的哈希值，它指出应该搜索哪个桶。容器将具有一个特定哈希值的所有元素都保存在相同的桶中。如果容器允许重复关键字，所有具有相同关键字的元素也都会在同一个桶中。因此，无序容器的性能依赖于哈希函数的质量和桶的数量和大小。

对于相同的参数，哈希函数必须总是产生相同的结果。理想情况下，哈希函数还能将 445
每个特定的值映射到唯一的桶。但是，将不同关键字的元素映射到相同的桶也是允许的。当一个桶保存多个元素时，需要顺序搜索这些元素来查找我们想要的那个。计算一个元素的哈希值和在桶中搜索通常都是很快的操作。但是，如果一个桶中保存了很多元素，那么查找一个特定元素就需要大量比较操作。

无序容器提供了一组管理桶的函数，如表 11.8 所示。这些成员函数允许我们查询容器的状态以及在必要时强制容器进行重组。

表 11.8：无序容器管理操作

桶接口	
c.bucket_count()	正在使用的桶的数目
c.max_bucket_count()	容器能容纳的最多的桶的数量
c.bucket_size(n)	第 n 个桶中有多少个元素
c.bucket(k)	关键字为 k 的元素在哪个桶中
桶迭代	
local_iterator	可以用来访问桶中元素的迭代器类型
const_local_iterator	桶迭代器的 const 版本
c.begin(n), c.end(n)	桶 n 的首元素迭代器和尾后迭代器
c.cbegin(n), c.cend(n)	与前两个函数类似，但返回 const_local_iterator
哈希策略	
c.load_factor()	每个桶的平均元素数量，返回 float 值
c.max_load_factor()	c 试图维护的平均桶大小，返回 float 值。c 会在需要时添加新的桶，以使得 load_factor<=max_load_ factor
c.rehash(n)	重组存储，使得 bucket_count>=n 且 bucket_count>size/max_load_factor
c.reserve(n)	重组存储，使得 c 可以保存 n 个元素且不必 rehash

无序容器对关键字类型的要求

默认情况下，无序容器使用关键字类型的==运算符来比较元素，它们还使用一个hash<key_type>类型的对象来生成每个元素的哈希值。标准库为内置类型（包括指针）提供了 **hash** 模板。还为一些标准库类型，包括 string 和我们将要在第 12 章介绍的智能指针类型定义了 hash。因此，我们可以直接定义关键字是内置类型（包括指针类型）、string 还是智能指针类型的无序容器。

但是，我们不能直接定义关键字类型为自定义类类型的无序容器。与容器不同，不能直接使用哈希模板，而必须提供我们自己的 hash 模板版本。我们将在 16.5 节（第 626 页）中介绍如何做到这一点。

我们不使用默认的 hash，而是使用另一种方法，类似于为有序容器重载关键字类型
446
的默认比较操作（参见 11.2.2 节，第 378 页）。为了能将 Sale_data 用作关键字，我们需要提供函数来替代==运算符和哈希值计算函数。我们从定义这些重载函数开始：

```
size_t hasher(const Sales_data &sd)
{
    return hash<string>()(sd.isbn());
}
bool eqOp(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn();
}
```

我们的 hasher 函数使用一个标准库 hash 类型对象来计算 ISBN 成员的哈希值，该 hash 类型建立在 string 类型之上。类似的，eqOp 函数通过比较 ISBN 号来比较两个 Sales_data。

我们使用这些函数来定义一个 unordered_multiset

```
using SD_multiset = unordered_multiset<Sales_data,
                    decltype(hasher)*, decltype(eqOp)*>;
// 参数是桶大小、哈希函数指针和相等性判断运算符指针
SD_multiset bookstore(42, hasher, eqOp);
```

为了简化 bookstore 的定义，首先为 unordered_multiset 定义了一个类型别名（参见 2.5.1 节，第 60 页），此集合的哈希和相等性判断操作与 hasher 和 eqOp 函数有着相同的类型。通过使用这种类型，在定义 bookstore 时可以将我们希望它使用的函数的指针传递给它。

如果我们的类定义了==运算符，则可以只重载哈希函数：

```
// 使用 FooHash 生成哈希值；Foo 必须有==运算符
unordered_set<Foo, decltype(FooHash)*> fooSet(10, FooHash);
```

11.4 节练习

练习 11.37：一个无序容器与其有序版本相比有何优势？有序版本有何优势？

练习 11.38：用 unordered_map 重写单词计数程序（参见 11.1 节，第 375 页）和单词转换程序（参见 11.3.6 节，第 391 页）。

小结

447

关联容器支持通过关键字高效查找和提取元素。对关键字的使用将关联容器和顺序容器区分开来，顺序容器中是通过位置访问元素的。

标准库定义了 8 个关联容器，每个容器

- 是一个 map 或是一个 set。map 保存关键字-值对；set 只保存关键字。
- 要求关键字唯一或不要求。
- 保持关键字有序或不保证有序。

有序容器使用比较函数来比较关键字，从而将元素按顺序存储。默认情况下，比较操作是采用关键字类型的<运算符。无序容器使用关键字类型的==运算符和一个 hash<key_type>类型的对象来组织元素。

允许重复关键字的容器的名字中都包含 multi；而使用哈希技术的容器的名字都以 unordered 开头。例如，set 是一个有序集合，其中每个关键字只可以出现一次；unordered_multiset 则是一个无序的关键字集合，其中关键字可以出现多次。

关联容器和顺序容器有很多共同的元素。但是，关联容器定义了一些新操作，并对一些和顺序容器和关联容器都支持的操作重新定义了含义或返回类型。操作的不同反映出关联容器使用关键字的特点。

有序容器的迭代器通过关键字有序访问容器中的元素。无论在有序容器中还是在无序容器中，具有相同关键字的元素都是相邻存储的。

术语表

关联数组（associative array） 元素通过关键字而不是位置来索引的数组。我们称这样的数组将一个关键字映射到其关联的值。

关联容器（associative container） 类型，保存对象的集合，支持通过关键字的高效查找。

hash 特殊的标准库模板，无序容器用它来管理元素的位置。

哈希函数（hash function） 将给定类型的值映射到整形（size_t）值的函数。相等的值必须映射到相同的整数；不相等的值应尽可能映射到不同整数。

key_type 关联容器定义的类型，用来保存和提取值的关键字的类型。对于一个 map，key_type 是用来索引 map 的类型。对于 set，key_type 和 value_type 是一样的。

map 关联容器类型，定义了一个关联数组。类似 vector，map 是一个类模板。但是，一个 map 要用两个类型来定义：关键字的类型和关联的值的类型。在一个 map 中，一个给定关键字只能出现一次。每个关键字关联一个特定的值。解引用一个 map 迭代器会生成一个 pair，它保存一个 const 关键字及其关联的值。

mapped_type 映射类型定义的类型，就是映射中关键字关联的值的类型。 448

multimap 关联容器类型，类似 map，不同之处在于，在一个 multimap 中，一个给定的关键字可以出现多次。multimap 不支持下标操作。

multiset 保存关键字的关联容器类型。在一个 multiset 中，一个给定关键字可以出现多次。

pair 类型，保存名为 first 和 second 的 public 数据成员。pair 类型是模板类型，接受两个类型参数，作为其成员的类型。

set 保存关键字的关联容器。在一个 set 中，一个给定的关键字只能出现一次。

严格弱序（strict weak ordering） 关联容器所使用的关键字间的关系。在一个严格弱序中，可以比较任意两个值并确定哪个更小。若任何一个都不小于另一个，则认为两个值相等。

无序容器（unordered container） 关联容器，用哈希技术而不是比较操作来存储和访问元素。这类容器的性能依赖于哈希函数的质量。

unordered_map 保存关键字-值对的容器，不允许重复关键字。

unordered_multimap 保存关键字-值对的容器，允许重复关键字。

unordered_multiset 保存关键字的容器，允许重复关键字。

unordered_set 保存关键字的容器，不允许重复关键字。

value_type 容器中元素的类型。对于 set 和 multiset，value_type 和 key_type 是一样的。对于 map 和 multimap，此类型是一个 pair，其 first 成员类型为 const key_type，second 成员类型为 mapped_type。

***运算符** 解引用运算符。当应用于 map、set、multimap 或 multiset 的迭代器时，会生成一个 value_type 值。注意，对 map 和 multimap，value_type 是一个 pair。

[ ]运算符 下标运算符。只能用于 map 和 unordered_map 类型的非 const 对象。对于映射类型，[]接受一个索引，必须是一个 key_type 值（或者是能转换为 key_type 的类型）。生成一个 mapped_type 值。

第 12 章

动态内存

内容

12.1 动态内存与智能指针 400
12.2 动态数组 423
12.3 使用标准库：文本查询程序 430
小结 436
术语表 436

到目前为止，我们编写的程序中所使用的对象都有着严格定义的生存期。全局对象在程序启动时分配，在程序结束时销毁。对于局部自动对象，当我们进入其定义所在的程序块时被创建，在离开块时销毁。局部 `static` 对象在第一次使用前分配，在程序结束时销毁。

除了自动和 `static` 对象外，C++还支持动态分配对象。动态分配的对象的生存期与它们在哪里创建是无关的，只有当显式地被释放时，这些对象才会销毁。

动态对象的正确释放被证明是编程中极其容易出错的地方。为了更安全地使用动态对象，标准库定义了两个智能指针类型来管理动态分配的对象。当一个对象应该被释放时，指向它的智能指针可以确保自动地释放它。

450 我们的程序到目前为止只使用过静态内存或栈内存。静态内存用来保存局部 static 对象（参见 6.6.1 节，第 185 页）、类 static 数据成员（参见 7.6 节，第 268 页）以及定义在任何函数之外的变量。栈内存用来保存定义在函数内的非 static 对象。分配在静态或栈内存中的对象由编译器自动创建和销毁。对于栈对象，仅在其定义的程序块运行时才存在；static 对象在使用之前分配，在程序结束时销毁。

除了静态内存和栈内存，每个程序还拥有一个内存池。这部分内存被称作**自由空间**（free store）或**堆**（heap）。程序用堆来存储**动态分配**（dynamically allocate）的对象——即，那些在程序运行时分配的对象。动态对象的生存期由程序来控制，也就是说，当动态对象不再使用时，我们的代码必须显式地销毁它们。



虽然使用动态内存有时是必要的，但众所周知，正确地管理动态内存是非常棘手的。

12.1 动态内存与智能指针

在 C++中，动态内存的管理是通过一对运算符来完成的：**new**，在动态内存中为对象分配空间并返回一个指向该对象的指针，我们可以选择对对象进行初始化；**delete**，接受一个动态对象的指针，销毁该对象，并释放与之关联的内存。

动态内存的使用很容易出问题，因为确保在正确的时间释放内存是极其困难的。有时我们会忘记释放内存，在这种情况下就会产生内存泄漏；有时在尚有指针引用内存的情况下我们就释放了它，在这种情况下就会产生引用非法内存的指针。

C++ 11 为了更容易（同时也更安全）地使用动态内存，新的标准库提供了两种**智能指针**（smart pointer）类型来管理动态对象。智能指针的行为类似常规指针，重要的区别是它负责自动释放所指向的对象。新标准库提供的这两种智能指针的区别在于管理底层指针的方式：**shared_ptr** 允许多个指针指向同一个对象；**unique_ptr** 则“独占”所指向的对象。标准库还定义了一个名为 **weak_ptr** 的伴随类，它是一种弱引用，指向 shared_ptr 所管理的对象。这三种类型都定义在 memory 头文件中。

12.1.1 shared_ptr 类

类似 vector，智能指针也是模板（参见 3.3 节，第 86 页）。因此，当我们创建一个 C++ 11 智能指针时，必须提供额外的信息——指针可以指向的类型。与 vector 一样，我们在尖括号内给出类型，之后是所定义的这种智能指针的名字：

```
shared_ptr<string> p1;        // shared_ptr，可以指向 string
shared_ptr<list<int>> p2;     // shared_ptr，可以指向 int 的 list
```

451

默认初始化的智能指针中保存着一个空指针（参见 2.3.2 节，第 48 页）。在 12.1.3 节中（见第 412 页），我们将介绍初始化智能指针的其他方法。

智能指针的使用方式与普通指针类似。解引用一个智能指针返回它指向的对象。如果在一个条件判断中使用智能指针，效果就是检测它是否为空：

```
// 如果 p1 不为空，检查它是否指向一个空 string
if (p1 && p1->empty())
    *p1 = "hi"; // 如果 p1 指向一个空 string，解引用 p1，将一个新值赋予 string
```

表 12.1 列出了 shared_ptr 和 unique_ptr 都支持的操作。只适用于 shared_ptr 的

操作列于表 12.2 中。

表 12.1：shared_ptr 和 unique_ptr 都支持的操作

`shared_ptr<T> sp` `unique_ptr<T> up`	空智能指针，可以指向类型为 T 的对象
`p`	将 p 用作一个条件判断，若 p 指向一个对象，则为 true
`*p`	解引用 p，获得它指向的对象
`p->mem`	等价于 `(*p).mem`
`p.get()`	返回 p 中保存的指针。要小心使用，若智能指针释放了其对象，返回的指针所指向的对象也就消失了
`swap(p, q)` `p.swap(q)`	交换 p 和 q 中的指针

表 12.2：shared_ptr 独有的操作

`make_shared<T>(args)`	返回一个 shared_ptr，指向一个动态分配的类型为 T 的对象。使用 *args* 初始化此对象
`shared_ptr<T>p(q)`	p 是 shared_ptr q 的拷贝；此操作会递增 q 中的计数器。q 中的指针必须能转换为 T*（参见 4.11.2 节，第 143 页）
`p = q`	p 和 q 都是 shared_ptr，所保存的指针必须能相互转换。此操作会递减 p 的引用计数，递增 q 的引用计数；若 p 的引用计数变为 0，则将其管理的原内存释放
`p.unique()`	若 `p.use_count()` 为 1，返回 true；否则返回 false
`p.use_count()`	返回与 p 共享对象的智能指针数量；可能很慢，主要用于调试

make_shared 函数

最安全的分配和使用动态内存的方法是调用一个名为 make_shared 的标准库函数。此函数在动态内存中分配一个对象并初始化它，返回指向此对象的 shared_ptr。与智能指针一样，make_shared 也定义在头文件 memory 中。

当要用 make_shared 时，必须指定想要创建的对象的类型。定义方式与模板类相同，在函数名之后跟一个尖括号，在其中给出类型：

```
// 指向一个值为 42 的 int 的 shared_ptr
shared_ptr<int> p3 = make_shared<int>(42);
// p4 指向一个值为"9999999999"的 string
shared_ptr<string> p4 = make_shared<string>(10, '9');
// p5 指向一个值初始化的(参见 3.3.1 节，第 88 页)int，即，值为 0
shared_ptr<int> p5 = make_shared<int>();
```

类似顺序容器的 emplace 成员（参见 9.3.1 节，第 308 页），make_shared 用其参数来构造给定类型的对象。例如，调用 make_shared<string>时传递的参数必须与 string 的某个构造函数相匹配，调用 make_shared<int>时传递的参数必须能用来初始化一个 int，依此类推。如果我们不传递任何参数，对象就会进行值初始化（参见 3.3.1 节，第 88 页）。

当然，我们通常用 auto（参见 2.5.2 节，第 61 页）定义一个对象来保存 make_shared 的结果，这种方式较为简单：

```
// p6 指向一个动态分配的空 vector<string>
auto p6 = make_shared<vector<string>>();
```

shared_ptr 的拷贝和赋值

当进行拷贝或赋值操作时，每个 shared_ptr 都会记录有多少个其他 shared_ptr 指向相同的对象：

```
auto p = make_shared<int>(42); // p 指向的对象只有 p 一个引用者
auto q(p); // p 和 q 指向相同对象，此对象有两个引用者
```

452 我们可以认为每个 shared_ptr 都有一个关联的计数器，通常称其为**引用计数**（reference count）。无论何时我们拷贝一个 shared_ptr，计数器都会递增。例如，当用一个 shared_ptr 初始化另一个 shared_ptr，或将它作为参数传递给一个函数（参见 6.2.1 节，第 188 页）以及作为函数的返回值（参见 6.3.2 节，第 201 页）时，它所关联的计数器就会递增。当我们给 shared_ptr 赋予一个新值或是 shared_ptr 被销毁（例如一个局部的 shared_ptr 离开其作用域（参见 6.1.1 节，第 184 页））时，计数器就会递减。

一旦一个 shared_ptr 的计数器变为 0，它就会自动释放自己所管理的对象：

```
auto r = make_shared<int>(42); // r 指向的 int 只有一个引用者
r = q; // 给 r 赋值，令它指向另一个地址
       // 递增 q 指向的对象的引用计数
       // 递减 r 原来指向的对象的引用计数
       // r 原来指向的对象已没有引用者，会自动释放
```

此例中我们分配了一个 int，将其指针保存在 r 中。接下来，我们将一个新值赋予 r。在此情况下，r 是唯一指向此 int 的 shared_ptr，在把 q 赋给 r 的过程中，此 int 被自动释放。



到底是用一个计数器还是其他数据结构来记录有多少指针共享对象，完全由标准库的具体实现来决定。关键是智能指针类能记录有多少个 shared_ptr 指向相同的对象，并能在恰当的时候自动释放对象。

shared_ptr 自动销毁所管理的对象……

当指向一个对象的最后一个 shared_ptr 被销毁时，shared_ptr 类会自动销毁此对象。它是通过另一个特殊的成员函数——**析构函数**（destructor）完成销毁工作的。类似于构造函数，每个类都有一个析构函数。就像构造函数控制初始化一样，析构函数控制此类型的对象销毁时做什么操作。

453 析构函数一般用来释放对象所分配的资源。例如，string 的构造函数（以及其他 string 成员）会分配内存来保存构成 string 的字符。string 的析构函数就负责释放这些内存。类似的，vector 的若干操作都会分配内存来保存其元素。vector 的析构函数就负责销毁这些元素，并释放它们所占用的内存。

shared_ptr 的析构函数会递减它所指向的对象的引用计数。如果引用计数变为 0，shared_ptr 的析构函数就会销毁对象，并释放它占用的内存。

……shared_ptr 还会自动释放相关联的内存

当动态对象不再被使用时，shared_ptr 类会自动释放动态对象，这一特性使得动态内存的使用变得非常容易。例如，我们可能有一个函数，它返回一个 shared_ptr，指向

一个 Foo 类型的动态分配的对象，对象是通过一个类型为 T 的参数进行初始化的：

```
// factory返回一个shared_ptr，指向一个动态分配的对象
shared_ptr<Foo> factory(T arg)
{
    // 恰当地处理arg
    // shared_ptr负责释放内存
    return make_shared<Foo>(arg);
}
```

由于 factory 返回一个 shared_ptr，所以我们可以确保它分配的对象会在恰当的时刻被释放。例如，下面的函数将 factory 返回的 shared_ptr 保存在局部变量中：

```
void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // 使用p
} // p离开了作用域，它指向的内存会被自动释放掉
```

由于 p 是 use_factory 的局部变量，在 use_factory 结束时它将被销毁（参见 6.1.1 节，第 184 页）。当 p 被销毁时，将递减其引用计数并检查它是否为 0。在此例中，p 是唯一引用 factory 返回的内存的对象。由于 p 将要销毁，p 指向的这个对象也会被销毁，所占用的内存会被释放。

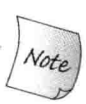
454

但如果有其他 shared_ptr 也指向这块内存，它就不会被释放掉：

```
void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // 使用p
    return p; // 当我们返回p时，引用计数进行了递增操作
} // p离开了作用域，但它指向的内存不会被释放掉
```

在此版本中，use_factory 中的 return 语句向此函数的调用者返回一个 p 的拷贝。拷贝一个 shared_ptr 会增加所管理对象的引用计数值。现在当 p 被销毁时，它所指向的内存还有其他使用者。对于一块内存，shared_ptr 类保证只要有任何 shared_ptr 对象引用它，它就不会被释放掉。

由于在最后一个 shared_ptr 销毁前内存都不会释放，保证 shared_ptr 在无用之后不再保留就非常重要了。如果你忘记了销毁程序不再需要的 shared_ptr，程序仍会正确执行，但会浪费内存。share_ptr 在无用之后仍然保留的一种可能情况是，你将 shared_ptr 存放在一个容器中，随后重排了容器，从而不再需要某些元素。在这种情况下，你应该确保用 erase 删除那些不再需要的 shared_ptr 元素。



如果你将 shared_ptr 存放于一个容器中，而后不再需要全部元素，而只使用其中一部分，要记得用 erase 删除不再需要的那些元素。

使用了动态生存期的资源的类

程序使用动态内存出于以下三种原因之一：

1. 程序不知道自己需要使用多少对象
2. 程序不知道所需对象的准确类型

3．程序需要在多个对象间共享数据

容器类是出于第一种原因而使用动态内存的典型例子，我们将在第 15 章看到出于第二种原因而使用动态内存的例子。在本节中，我们将定义一个类，它使用动态内存是为了让多个对象能共享相同的底层数据。

到目前为止，我们使用过的类中，分配的资源都与对应对象生存期一致。例如，每个 vector“拥有”其自己的元素。当我们拷贝一个 vector 时，原 vector 和副本 vector 中的元素是相互分离的：

455

```
vector<string> v1; // 空 vector
{ // 新作用域
    vector<string> v2 = {"a", "an", "the"};
    v1 = v2; // 从 v2 拷贝元素到 v1 中
} // v2 被销毁，其中的元素也被销毁
  // v1 有三个元素，是原来 v2 中元素的拷贝
```

由一个 vector 分配的元素只有当这个 vector 存在时才存在。当一个 vector 被销毁时，这个 vector 中的元素也都被销毁。

但某些类分配的资源具有与原对象相独立的生存期。例如，假定我们希望定义一个名为 Blob 的类，保存一组元素。与容器不同，我们希望 Blob 对象的不同拷贝之间共享相同的元素。即，当我们拷贝一个 Blob 时，原 Blob 对象及其拷贝应该引用相同的底层元素。

一般而言，如果两个对象共享底层的数据，当某个对象被销毁时，我们不能单方面地销毁底层数据：

```
Blob<string> b1; // 空 Blob
{ // 新作用域
    Blob<string> b2 = {"a", "an", "the"};
    b1 = b2; // b1 和 b2 共享相同的元素
} // b2 被销毁了，但 b2 中的元素不能销毁
  // b1 指向最初由 b2 创建的元素
```

在此例中，b1 和 b2 共享相同的元素。当 b2 离开作用域时，这些元素必须保留，因为 b1 仍然在使用它们。



使用动态内存的一个常见原因是允许多个对象共享相同的状态。

定义 StrBlob 类

最终，我们会将 Blob 类实现为一个模板，但我们直到 16.1.2 节（第 583 页）才会学习模板的相关知识。因此，现在我们先定义一个管理 string 的类，此版本命名为 StrBlob。

实现一个新的集合类型的最简单方法是使用某个标准库容器来管理元素。采用这种方法，我们可以借助标准库类型来管理元素所使用的内存空间。在本例中，我们将使用 vector 来保存元素。

但是，我们不能在一个 Blob 对象内直接保存 vector，因为一个对象的成员在对象销毁时也会被销毁。例如，假定 b1 和 b2 是两个 Blob 对象，共享相同的 vector。如果此 vector 保存在其中一个 Blob 中——例如 b2 中，那么当 b2 离开作用域时，此 vector 也将被销毁，也就是说其中的元素都将不复存在。为了保证 vector 中的元素继续存在，

我们将 vector 保存在动态内存中。

为了实现我们所希望的数据共享，我们为每个 StrBlob 设置一个 shared_ptr 来管理动态分配的 vector。此 shared_ptr 的成员将记录有多少个 StrBlob 共享相同的 vector，并在 vector 的最后一个使用者被销毁时释放 vector。

我们还需要确定这个类应该提供什么操作。当前，我们将实现一个 vector 操作的小的子集。我们会修改访问元素的操作（如 front 和 back）：在我们的类中，如果用户试图访问不存在的元素，这些操作会抛出一个异常。 456

我们的类有一个默认构造函数和一个构造函数，接受单一的 initializer_list<string>类型参数（参见 6.2.6 节，第 198 页）。此构造函数可以接受一个初始化器的花括号列表。

```
class StrBlob {
public:
    typedef std::vector<std::string>::size_type size_type;
    StrBlob();
    StrBlob(std::initializer_list<std::string> il);
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // 添加和删除元素
    void push_back(const std::string &t) {data->push_back(t);}
    void pop_back();
    // 元素访问
    std::string& front();
    std::string& back();
private:
    std::shared_ptr<std::vector<std::string>> data;
    // 如果 data[i]不合法，抛出一个异常
    void check(size_type i, const std::string &msg) const;
};
```

在此类中，我们实现了 size、empty 和 push_back 成员。这些成员通过指向底层 vector 的 data 成员来完成它们的工作。例如，对一个 StrBlob 对象调用 size()会调用 data->size()，依此类推。

StrBlob 构造函数

两个构造函数都使用初始化列表（参见 7.1.4 节，第 237 页）来初始化其 data 成员，令它指向一个动态分配的 vector。默认构造函数分配一个空 vector：

```
StrBlob::StrBlob(): data(make_shared<vector<string>>()) { }
StrBlob::StrBlob(initializer_list<string> il):
                data(make_shared<vector<string>>(il)) { }
```

接受一个 initializer_list 的构造函数将其参数传递给对应的 vector 构造函数（参见 2.2.1 节，第 39 页）。此构造函数通过拷贝列表中的值来初始化 vector 的元素。

元素访问成员函数

pop_back、front 和 back 操作访问 vector 中的元素。这些操作在试图访问元素之前必须检查元素是否存在。由于这些成员函数需要做相同的检查操作，我们为 StrBlob 定义了一个名为 check 的 private 工具函数，它检查一个给定索引是否在合法范围内。 457

除了索引，check 还接受一个 string 参数，它会将此参数传递给异常处理程序，这个 string 描述了错误内容：

```
void StrBlob::check(size_type i, const string &msg) const
{
    if (i >= data->size())
        throw out_of_range(msg);
}
```

pop_back 和元素访问成员函数首先调用 check。如果 check 成功，这些成员函数继续利用底层 vector 的操作来完成自己的工作：

```
string& StrBlob::front()
{
    // 如果 vector 为空，check 会抛出一个异常
    check(0, "front on empty StrBlob");
    return data->front();
}
string& StrBlob::back()
{
    check(0, "back on empty StrBlob");
    return data->back();
}
void StrBlob::pop_back()
{
    check(0, "pop_back on empty StrBlob");
    data->pop_back();
}
```

front 和 back 应该对 const 进行重载（参见 7.3.2 节，第 247 页），这些版本的定义留作练习。

StrBlob 的拷贝、赋值和销毁

类似 Sales_data 类，StrBlob 使用默认版本的拷贝、赋值和销毁成员函数来对此类型的对象进行这些操作（参见 7.1.5 节，第 239 页）。默认情况下，这些操作拷贝、赋值和销毁类的数据成员。我们的 StrBlob 类只有一个数据成员，它是 shared_ptr 类型。因此，当我们拷贝、赋值或销毁一个 StrBlob 对象时，它的 shared_ptr 成员会被拷贝、赋值或销毁。

如前所见，拷贝一个 shared_ptr 会递增其引用计数；将一个 shared_ptr 赋予另一个 shared_ptr 会递增赋值号右侧 shared_ptr 的引用计数，而递减左侧 shared_ptr 的引用计数。如果一个 shared_ptr 的引用计数变为 0，它所指向的对象会被自动销毁。因此，对于由 StrBlob 构造函数分配的 vector，当最后一个指向它的 StrBlob 对象被销毁时，它会随之被自动销毁。

458

12.1.1 节练习

练习 12.1：在此代码的结尾，b1 和 b2 各包含多少个元素？

```
StrBlob b1;
{
    StrBlob b2 = {"a", "an", "the"};
```

```
    b1 = b2;
    b2.push_back("about");
}
```

练习 12.2：编写你自己的 StrBlob 类，包含 const 版本的 front 和 back。

练习 12.3：StrBlob 需要 const 版本的 push_back 和 pop_back 吗？如果需要，添加进去。否则，解释为什么不需要。

练习 12.4：在我们的 check 函数中，没有检查 i 是否大于 0。为什么可以忽略这个检查？

练习 12.5：我们未编写接受一个 initializer_list explicit（参见 7.5.4 节，第 264 页）参数的构造函数。讨论这个设计策略的优点和缺点。

12.1.2 直接管理内存

C++语言定义了两个运算符来分配和释放动态内存。运算符 new 分配内存，delete 释放 new 分配的内存。

相对于智能指针，使用这两个运算符管理内存非常容易出错，随着我们逐步详细介绍这两个运算符，这一点会更为清楚。而且，自己直接管理内存的类与使用智能指针的类不同，它们不能依赖类对象拷贝、赋值和销毁操作的任何默认定义（参见 7.1.4 节，第 237 页）。因此，使用智能指针的程序更容易编写和调试。



在学习第 13 章之前，除非使用智能指针来管理内存，否则不要分配动态内存。

使用 new 动态分配和初始化对象

在自由空间分配的内存是无名的，因此 **new** 无法为其分配的对象命名，而是返回一个指向该对象的指针：

```
int *pi = new int; // pi 指向一个动态分配的、未初始化的无名对象
```

此 new 表达式在自由空间构造一个 int 型对象，并返回指向该对象的指针。

默认情况下，动态分配的对象是默认初始化的（参见 2.2.1 节，第 40 页），这意味着内置类型或组合类型的对象的值将是未定义的，而类类型对象将用默认构造函数进行初始化：

```
string *ps = new string; // 初始化为空 string
int *pi = new int;       // pi 指向一个未初始化的 int
```

459

我们可以使用直接初始化方式（参见 3.2.1 节，第 76 页）来初始化一个动态分配的对象。我们可以使用传统的构造方式（使用圆括号），在新标准下，也可以使用列表初始化（使用花括号）：




```
int *pi = new int(1024);              // pi 指向的对象的值为 1024
string *ps = new string(10, '9');     // *ps 为"9999999999"
// vector 有 10 个元素，值依次从 0 到 9

vector<int> *pv = new vector<int>{0,1,2,3,4,5,6,7,8,9};
```

也可以对动态分配的对象进行值初始化（参见 3.3.1 节，第 88 页），只需在类型名之

后跟一对空括号即可：

```
string *ps1 = new string;      // 默认初始化为空 string
string *ps = new string();     // 值初始化为空 string
int *pi1 = new int;            // 默认初始化；*pi1 的值未定义
int *pi2 = new int();          // 值初始化为 0；*pi2 为 0
```

对于定义了自己的构造函数（参见 7.1.4 节，第 235 页）的类类型（例如 string）来说，要求值初始化是没有意义的；不管采用什么形式，对象都会通过默认构造函数来初始化。但对于内置类型，两种形式的差别就很大了；值初始化的内置类型对象有着良好定义的值，而默认初始化的对象的值则是未定义的。类似的，对于类中那些依赖于编译器合成的默认构造函数的内置类型成员，如果它们未在类内被初始化，那么它们的值也是未定义的（参见 7.1.4 节，第 236 页）。

Best Practices 出于与变量初始化相同的原因，对动态分配的对象进行初始化通常是个好主意。

如果我们提供了一个括号包围的初始化器，就可以使用 auto（参见 2.5.2 节，第 61 页）从此初始化器来推断我们想要分配的对象的类型。但是，由于编译器要用初始化器的类型来推断要分配的类型，只有当括号中仅有单一初始化器时才可以使用 auto：

C++ 11

```
auto p1 = new auto(obj);       // p 指向一个与 obj 类型相同的对象
                               // 该对象用 obj 进行初始化
auto p2 = new auto{a,b,c};     // 错误：括号中只能有单个初始化器
```

p1 的类型是一个指针，指向从 obj 自动推断出的类型。若 obj 是一个 int，那么 p1 就是 int*；若 obj 是一个 string，那么 p1 是一个 string*；依此类推。新分配的对象用 obj 的值进行初始化。

动态分配的 const 对象

用 new 分配 const 对象是合法的：

```
// 分配并初始化一个 const int
const int *pci = new const int(1024);
// 分配并默认初始化一个 const 的空 string
const string *pcs = new const string;
```

460 类似其他任何 const 对象，一个动态分配的 const 对象必须进行初始化。对于一个定义了默认构造函数（参见 7.1.4 节，第 236 页）的类类型，其 const 动态对象可以隐式初始化，而其他类型的对象就必须显式初始化。由于分配的对象是 const 的，new 返回的指针是一个指向 const 的指针（参见 2.4.2 节，第 56 页）。

内存耗尽

虽然现代计算机通常都配备大容量内存，但是自由空间被耗尽的情况还是有可能发生。一旦一个程序用光了它所有可用的内存，new 表达式就会失败。默认情况下，如果 new 不能分配所要求的内存空间，它会抛出一个类型为 bad_alloc（参见 5.6 节，第 173 页）的异常。我们可以改变使用 new 的方式来阻止它抛出异常：

```
// 如果分配失败，new 返回一个空指针
int *p1 = new int; // 如果分配失败，new 抛出 std::bad_alloc
int *p2 = new (nothrow) int; // 如果分配失败，new 返回一个空指针
```

我们称这种形式的 new 为**定位 new**（placement new），其原因我们将在 19.1.2 节（第 729 页）中解释。定位 new 表达式允许我们向 new 传递额外的参数。在此例中，我们传递给它一个由标准库定义的名为 nothrow 的对象。如果将 nothrow 传递给 new，我们的意图是告诉它不能抛出异常。如果这种形式的 new 不能分配所需内存，它会返回一个空指针。bad_alloc 和 nothrow 都定义在头文件 new 中。

释放动态内存

为了防止内存耗尽，在动态内存使用完毕后，必须将其归还给系统。我们通过 **delete 表达式**（delete expression）来将动态内存归还给系统。delete 表达式接受一个指针，指向我们想要释放的对象：

```
delete p; // p 必须指向一个动态分配的对象或是一个空指针
```

与 new 类型类似，delete 表达式也执行两个动作：销毁给定的指针指向的对象；释放对应的内存。

指针值和 delete

我们传递给 delete 的指针必须指向动态分配的内存，或者是一个空指针（参见 2.3.2 节，第 48 页）。释放一块并非 new 分配的内存，或者将相同的指针值释放多次，其行为是未定义的：

```
int i, *pi1 = &i, *pi2 = nullptr;
double *pd = new double(33), *pd2 = pd;
delete i;    // 错误：i 不是一个指针
delete pi1;  // 未定义：pi1 指向一个局部变量
delete pd;   // 正确
delete pd2;  // 未定义：pd2 指向的内存已经被释放了
delete pi2;  // 正确：释放一个空指针总是没有错误的
```

对于 delete i 的请求，编译器会生成一个错误信息，因为它知道 i 不是一个指针。执行 delete pi1 和 pd2 所产生的错误则更具潜在危害：通常情况下，编译器不能分辨一个指针指向的是静态还是动态分配的对象。类似的，编译器也不能分辨一个指针所指向的内存是否已经被释放了。对于这些 delete 表达式，大多数编译器会编译通过，尽管它们是错误的。 461

虽然一个 const 对象的值不能被改变，但它本身是可以被销毁的。如同任何其他动态对象一样，想要释放一个 const 动态对象，只要 delete 指向它的指针即可：

```
const int *pci = new const int(1024);
delete pci; // 正确：释放一个 const 对象
```

动态对象的生存期直到被释放时为止

如 12.1.1 节（第 402 页）所述，由 shared_ptr 管理的内存在最后一个 shared_ptr 销毁时会被自动释放。但对于通过内置指针类型来管理的内存，就不是这样了。对于一个由内置指针管理的动态对象，直到被显式释放之前它都是存在的。

返回指向动态内存的指针（而不是智能指针）的函数给其调用者增加了一个额外负担——调用者必须记得释放内存：

```
// factory 返回一个指针，指向一个动态分配的对象
Foo* factory(T arg)
```

```
{
    // 视情况处理 arg
    return new Foo(arg); // 调用者负责释放此内存
}
```

类似我们之前定义的 factory 函数（参见 12.1.1 节，第 403 页），这个版本的 factory 分配一个对象，但并不 delete 它。factory 的调用者负责在不需要此对象时释放它。不幸的是，调用者经常忘记释放对象：

```
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p 但不 delete 它
} // p 离开了它的作用域，但它所指向的内存没有被释放!
```

此处，use_factory 函数调用 factory，后者分配一个类型为 Foo 的新对象。当 use_factory 返回时，局部变量 p 被销毁。此变量是一个内置指针，而不是一个智能指针。

与类类型不同，内置类型的对象被销毁时什么也不会发生。特别是，当一个指针离开其作用域时，它所指向的对象什么也不会发生。如果这个指针指向的是动态内存，那么内存将不会被自动释放。



由内置指针（而不是智能指针）管理的动态内存在被显式释放前一直都会存在。

462 在本例中，p 是指向 factory 分配的内存的唯一指针。一旦 use_factory 返回，程序就没有办法释放这块内存了。根据整个程序的逻辑，修正这个错误的正确方法是在 use_factory 中记得释放内存：

```
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p
    delete p; // 现在记得释放内存，我们已经不需要它了
}
```

还有一种可能，我们的系统中的其他代码要使用 use_factory 所分配的对象，我们就应该修改此函数，让它返回一个指针，指向它分配的内存：

```
Foo* use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p
    return p; // 调用者必须释放内存
}
```

小心：动态内存的管理非常容易出错

使用 new 和 delete 管理动态内存存在三个常见问题：

1. 忘记 delete 内存。忘记释放动态内存会导致人们常说的“内存泄漏”问题，因为这种内存永远不可能被归还给自由空间了。查找内存泄露错误是非常困难的，因为通常应用程序运行很长时间后，真正耗尽内存时，才能检测到这种错误。

2. 使用已经释放掉的对象。通过在释放内存后将指针置为空，有时可以检测出这

种错误。

3．同一块内存释放两次。当有两个指针指向相同的动态分配对象时，可能发生这种错误。如果对其中一个指针进行了 delete 操作，对象的内存就被归还给自由空间了。如果我们随后又 delete 第二个指针，自由空间就可能被破坏。

相对于查找和修正这些错误来说，制造出这些错误要简单得多。

Best Practices 坚持只使用智能指针，就可以避免所有这些问题。对于一块内存，只有在没有任何智能指针指向它的情况下，智能指针才会自动释放它。

delete 之后重置指针值……

当我们 delete 一个指针后，指针值就变为无效了。虽然指针已经无效，但在很多机器上指针仍然保存着（已经释放了的）动态内存的地址。在 delete 之后，指针就变成了人们所说的**空悬指针**（dangling pointer），即，指向一块曾经保存数据对象但现在已经无效的内存的指针。 463

未初始化指针（参见 2.3.2 节，第 49 页）的所有缺点空悬指针也都有。有一种方法可以避免空悬指针的问题：在指针即将要离开其作用域之前释放掉它所关联的内存。这样，在指针关联的内存被释放掉之后，就没有机会继续使用指针了。如果我们需要保留指针，可以在 delete 之后将 nullptr 赋予指针，这样就清楚地指出指针不指向任何对象。

……这只是提供了有限的保护

动态内存的一个基本问题是可能有多个指针指向相同的内存。在 delete 内存之后重置指针的方法只对这个指针有效，对其他任何仍指向（已释放的）内存的指针是没有作用的。例如：

```
int *p(new int(42)); // p 指向动态内存
auto q = p;          // p 和 q 指向相同的内存
delete p;            // p 和 q 均变为无效
p = nullptr;         // 指出 p 不再绑定到任何对象
```

本例中 p 和 q 指向相同的动态分配的对象。我们 delete 此内存，然后将 p 置为 nullptr，指出它不再指向任何对象。但是，重置 p 对 q 没有任何作用，在我们释放 p 所指向的（同时也是 q 所指向的！）内存时，q 也变为无效了。在实际系统中，查找指向相同内存的所有指针是异常困难的。

12.1.2 节练习

练习 12.6：编写函数，返回一个动态分配的 int 的 vector。将此 vector 传递给另一个函数，这个函数读取标准输入，将读入的值保存在 vector 元素中。再将 vector 传递给另一个函数，打印读入的值。记得在恰当的时刻 delete vector。

练习 12.7：重做上一题，这次使用 shared_ptr 而不是内置指针。

练习 12.8：下面的函数是否有错误？如果有，解释错误原因。

```
bool b() {
    int* p = new int;
    // ...
```

```
    return p;
}
```

练习 12.9：解释下面代码执行的结果：

```
int *q = new int(42), *r = new int(100);
r = q;
auto q2 = make_shared<int>(42), r2 = make_shared<int>(100);
r2 = q2;
```

464

12.1.3 shared_ptr 和 new 结合使用

如前所述，如果我们不初始化一个智能指针，它就会被初始化为一个空指针。如表 12.3 所示，我们还可以用 new 返回的指针来初始化智能指针：

```
shared_ptr<double> p1; // shared_ptr 可以指向一个 double
shared_ptr<int> p2(new int(42)); // p2 指向一个值为 42 的 int
```

接受指针参数的智能指针构造函数是 explicit 的（参见 7.5.4 节，第 265 页）。因此，我们不能将一个内置指针隐式转换为一个智能指针，必须使用直接初始化形式（参见 3.2.1 节，第 76 页）来初始化一个智能指针：

```
shared_ptr<int> p1 = new int(1024);   // 错误：必须使用直接初始化形式
shared_ptr<int> p2(new int(1024));    // 正确：使用了直接初始化形式
```

p1 的初始化隐式地要求编译器用一个 new 返回的 int*来创建一个 shared_ptr。由于我们不能进行内置指针到智能指针间的隐式转换，因此这条初始化语句是错误的。出于相同的原因，一个返回 shared_ptr 的函数不能在其返回语句中隐式转换一个普通指针：

```
shared_ptr<int> clone(int p) {
    return new int(p); // 错误：隐式转换为 shared_ptr<int>
}
```

我们必须将 shared_ptr 显式绑定到一个想要返回的指针上：

```
shared_ptr<int> clone(int p) {
    //正确：显式地用 int*创建 shared_ptr<int>
    return shared_ptr<int>(new int(p));
}
```

默认情况下，一个用来初始化智能指针的普通指针必须指向动态内存，因为智能指针默认使用 delete 释放它所关联的对象。我们可以将智能指针绑定到一个指向其他类型的资源的指针上，但是为了这样做，必须提供自己的操作来替代 delete。我们将在 12.1.4 节（第 415 页）介绍如何定义自己的释放操作。

表 12.3：定义和改变 shared_ptr 的其他方法

shared_ptr<T> p(q)	p 管理内置指针 q 所指向的对象；q 必须指向 new 分配的内存，且能够转换为 T*类型
shared_ptr<T> p(u)	p 从 unique_ptr u 那里接管了对象的所有权；将 u 置为空
shared_ptr<T> p(q, d)	p 接管了内置指针 q 所指向的对象的所有权。q 必须能转换为 T*类型（参见 4.11.2 节，第 143 页）。p 将使用可调用对象 d（参见 10.3.2 节，第 346 页）来代替 delete

续表

shared_ptr<T> p(p2, d)	如表 12.2 所示，p 是 shared_ptr p2 的拷贝，唯一的区别是 p 将用可调用对象 d 来代替 delete
p.reset() p.reset(q) p.reset(q, d)	若 p 是唯一指向其对象的 shared_ptr，reset 会释放此对象。若传递了可选的参数内置指针 q，会令 p 指向 q，否则会将 p 置为空。若还传递了参数 d，将会调用 d 而不是 delete 来释放 q

不要混合使用普通指针和智能指针……



shared_ptr 可以协调对象的析构，但这仅限于其自身的拷贝（也是 shared_ptr）之间。这也是为什么我们推荐使用 make_shared 而不是 new 的原因。这样，我们就能在分配对象的同时就将 shared_ptr 与之绑定，从而避免了无意中将同一块内存绑定到多个独立创建的 shared_ptr 上。

考虑下面对 shared_ptr 进行操作的函数：

```
// 在函数被调用时 ptr 被创建并初始化
void process(shared_ptr<int> ptr)
{
    // 使用 ptr
} // ptr 离开作用域，被销毁
```

process 的参数是传值方式传递的，因此实参会被拷贝到 ptr 中。拷贝一个 shared_ptr 会递增其引用计数，因此，在 process 运行过程中，引用计数值至少为 2。当 process 结束时，ptr 的引用计数会递减，但不会变为 0。因此，当局部变量 ptr 被销毁时，ptr 指向的内存不会被释放。 465

使用此函数的正确方法是传递给它一个 shared_ptr：

```
shared_ptr<int> p(new int(42)); // 引用计数为 1
process(p); // 拷贝 p 会递增它的引用计数；在 process 中引用计数值为 2
int i = *p; // 正确：引用计数值为 1
```

虽然不能传递给 process 一个内置指针，但可以传递给它一个（临时的）shared_ptr，这个 shared_ptr 是用一个内置指针显式构造的。但是，这样做很可能会导致错误：

```
int *x(new int(1024));          // 危险：x 是一个普通指针，不是一个智能指针
process(x); // 错误：不能将 int*转换为一个 shared_ptr<int>
process(shared_ptr<int>(x)); // 合法的，但内存会被释放！
int j = *x; // 未定义的：x 是一个空悬指针！
```

在上面的调用中，我们将一个临时 shared_ptr 传递给 process。当这个调用所在的表达式结束时，这个临时对象就被销毁了。销毁这个临时变量会递减引用计数，此时引用计数就变为 0 了。因此，当临时对象被销毁时，它所指向的内存会被释放。

但 x 继续指向（已经释放的）内存，从而变成一个空悬指针。如果试图使用 x 的值，其行为是未定义的。

当将一个 shared_ptr 绑定到一个普通指针时，我们就将内存的管理责任交给了这个 shared_ptr。一旦这样做了，我们就不应该再使用内置指针来访问 shared_ptr 所指向的内存了。

466



使用一个内置指针来访问一个智能指针所负责的对象是很危险的，因为我们无法知道对象何时会被销毁。



……也不要使用 get 初始化另一个智能指针或为智能指针赋值

智能指针类型定义了一个名为 get 的函数（参见表 12.1），它返回一个内置指针，指向智能指针管理的对象。此函数是为了这样一种情况而设计的：我们需要向不能使用智能指针的代码传递一个内置指针。使用 get 返回的指针的代码不能 delete 此指针。

虽然编译器不会给出错误信息，但将另一个智能指针也绑定到 get 返回的指针上是错误的：

```
shared_ptr<int> p(new int(42)); // 引用计数为 1
int *q = p.get(); // 正确：但使用 q 时要注意，不要让它管理的指针被释放
{ // 新程序块
// 未定义：两个独立的 shared_ptr 指向相同的内存
shared_ptr<int>(q);
} // 程序块结束，q 被销毁，它指向的内存被释放
int foo = *p; // 未定义：p 指向的内存已经被释放了
```

在本例中，p 和 q 指向相同的内存。由于它们是相互独立创建的，因此各自的引用计数都是 1。当 q 所在的程序块结束时，q 被销毁，这会导致 q 指向的内存被释放。从而 p 变成一个空悬指针，意味着当我们试图使用 p 时，将发生未定义的行为。而且，当 p 被销毁时，这块内存会被第二次 delete。



get 用来将指针的访问权限传递给代码，你只有在确定代码不会 delete 指针的情况下，才能使用 get。特别是，永远不要用 get 初始化另一个智能指针或者为另一个智能指针赋值。

其他 shared_ptr 操作

shared_ptr 还定义了其他一些操作，参见表 12.2 和表 12.3 所示。我们可以用 reset 来将一个新的指针赋予一个 shared_ptr：

```
p = new int(1024);          // 错误：不能将一个指针赋予 shared_ptr
p.reset(new int(1024));     // 正确：p 指向一个新对象
```

与赋值类似，reset 会更新引用计数，如果需要的话，会释放 p 指向的对象。reset 成员经常与 unique 一起使用，来控制多个 shared_ptr 共享的对象。在改变底层对象之前，我们检查自己是否是当前对象仅有的用户。如果不是，在改变之前要制作一份新的拷贝：

```
if (!p.unique())
    p.reset(new string(*p)); // 我们不是唯一用户；分配新的拷贝
*p += newVal; // 现在我们知道自己是唯一的用户，可以改变对象的值
```

467

12.1.3 节练习

练习 12.10：下面的代码调用了第 413 页中定义的 process 函数，解释此调用是否正确。如果不正确，应如何修改？

```
shared_ptr<int> p(new int(42));
```

```
process(shared_ptr<int>(p));
```

练习 12.11：如果我们像下面这样调用 process，会发生什么？

```
process(shared_ptr<int>(p.get()));
```

练习 12.12：p 和 q 的定义如下，对于接下来的对 process 的每个调用，如果合法，解释它做了什么，如果不合法，解释错误原因：

```
auto p = new int();
auto sp = make_shared<int>();
(a) process(sp);
(b) process(new int());
(c) process(p);
(d) process(shared_ptr<int>(p));
```

练习 12.13：如果执行下面的代码，会发生什么？

```
auto sp = make_shared<int>();
auto p = sp.get();
delete p;
```

12.1.4 智能指针和异常



5.6.2 节（第 175 页）中介绍了使用异常处理的程序能在异常发生后令程序流程继续，我们注意到，这种程序需要确保在异常发生后资源能被正确地释放。一个简单的确保资源被释放的方法是使用智能指针。

如果使用智能指针，即使程序块过早结束，智能指针类也能确保在内存不再需要时将其释放，：

```
void f()
{
    shared_ptr<int> sp(new int(42)); // 分配一个新对象
    //这段代码抛出一个异常，且在 f 中未被捕获
}  // 在函数结束时 shared_ptr 自动释放内存
```

函数的退出有两种可能，正常处理结束或者发生了异常，无论哪种情况，局部对象都会被销毁。在上面的程序中，sp 是一个 shared_ptr，因此 sp 销毁时会检查引用计数。在此例中，sp 是指向这块内存的唯一指针，因此内存会被释放掉。

与之相对的，当发生异常时，我们直接管理的内存是不会自动释放的。如果使用内置指针管理内存，且在 new 之后在对应的 delete 之前发生了异常，则内存不会被释放：

```
void f()
{
    int *ip = new int(42);    // 动态分配一个新对象
    // 这段代码抛出一个异常，且在 f 中未被捕获
    delete ip;                // 在退出之前释放内存
}
```

468

如果在 new 和 delete 之间发生异常，且异常未在 f 中被捕获，则内存就永远不会被释放了。在函数 f 之外没有指针指向这块内存，因此就无法释放它了。

智能指针和哑类

包括所有标准库类在内的很多 C++类都定义了析构函数（参见 12.1.1 节，第 402 页），负责清理对象使用的资源。但是，不是所有的类都是这样良好定义的。特别是那些为 C 和 C++两种语言设计的类，通常都要求用户显式地释放所使用的任何资源。

那些分配了资源，而又没有定义析构函数来释放这些资源的类，可能会遇到与使用动态内存相同的错误——程序员非常容易忘记释放资源。类似的，如果在资源分配和释放之间发生了异常，程序也会发生资源泄漏。

与管理动态内存类似，我们通常可以使用类似的技术来管理不具有良好定义的析构函数的类。例如，假定我们正在使用一个 C 和 C++都使用的网络库，使用这个库的代码可能是这样的：

```
struct destination;                    // 表示我们正在连接什么
struct connection;                     // 使用连接所需的信息
connection connect(destination*);      // 打开连接
void disconnect(connection);           // 关闭给定的连接
void f(destination &d /* 其他参数 */)
{
    // 获得一个连接；记住使用完后要关闭它
    connection c = connect(&d);
    // 使用连接
    // 如果我们在 f 退出前忘记调用 disconnect，就无法关闭 c 了
}
```

如果 connection 有一个析构函数，就可以在 f 结束时由析构函数自动关闭连接。但是，connection 没有析构函数。这个问题与我们上一个程序中使用 shared_ptr 避免内存泄漏几乎是等价的。使用 shared_ptr 来保证 connection 被正确关闭，已被证明是一种有效的方法。

使用我们自己的释放操作

469 默认情况下，shared_ptr 假定它们指向的是动态内存。因此，当一个 shared_ptr 被销毁时，它默认地对它管理的指针进行 delete 操作。为了用 shared_ptr 来管理一个 connection，我们必须首先定义一个函数来代替 delete。这个**删除器**（deleter）函数必须能够完成对 shared_ptr 中保存的指针进行释放的操作。在本例中，我们的删除器必须接受单个类型为 connection*的参数：

```
void end_connection(connection *p) { disconnect(*p); }
```

当我们创建一个 shared_ptr 时，可以传递一个（可选的）指向删除器函数的参数（参见 6.7 节，第 221 页）：

```
void f(destination &d /* 其他参数 */)
{
    connection c = connect(&d);
    shared_ptr<connection> p(&c, end_connection);
    // 使用连接
    // 当 f 退出时（即使是由于异常而退出），connection 会被正确关闭
}
```

当 p 被销毁时，它不会对自己保存的指针执行 delete，而是调用 end_connection。接下来，end_connection 会调用 disconnect，从而确保连接被关闭。如果 f 正常退出，那么 p 的销毁会作为结束处理的一部分。如果发生了异常，p 同样会被销毁，从而连接被关闭。

注意：智能指针陷阱

智能指针可以提供对动态分配的内存安全而又方便的管理，但这建立在正确使用的前提下。为了正确使用智能指针，我们必须坚持一些基本规范：

- 不使用相同的内置指针值初始化（或 reset）多个智能指针。
- 不 delete get()返回的指针。
- 不使用 get()初始化或 reset 另一个智能指针。
- 如果你使用 get()返回的指针，记住当最后一个对应的智能指针销毁后，你的指针就变为无效了。
- 如果你使用智能指针管理的资源不是 new 分配的内存，记住传递给它一个删除器（参见 12.1.4 节，第 415 页和 12.1.5 节，第 419 页）。

12.1.4 节练习

练习 12.14：编写你自己版本的用 shared_ptr 管理 connection 的函数。

练习 12.15：重写第一题的程序，用 lambda（参见 10.3.2 节，第 346 页）代替 end_connection 函数。

12.1.5 unique_ptr

470

一个 unique_ptr“拥有”它所指向的对象。与 shared_ptr 不同，某个时刻只能有一个 unique_ptr 指向一个给定对象。当 unique_ptr 被销毁时，它所指向的对象也被销毁。表 12.4 列出了 unique_ptr 特有的操作。与 shared_ptr 相同的操作列在表 12.1（第 401 页）中。

C++ 11

与 shared_ptr 不同，没有类似 make_shared 的标准库函数返回一个 unique_ptr。当我们定义一个 unique_ptr 时，需要将其绑定到一个 new 返回的指针上。类似 shared_ptr，初始化 unique_ptr 必须采用直接初始化形式：

```
unique_ptr<double> p1; // 可以指向一个 double 的 unique_ptr
unique_ptr<int> p2(new int(42)); // p2 指向一个值为 42 的 int
```

由于一个 unique_ptr 拥有它指向的对象，因此 unique_ptr 不支持普通的拷贝或赋值操作：

```
unique_ptr<string> p1(new string("Stegosaurus"));
unique_ptr<string> p2(p1);    // 错误：unique_ptr 不支持拷贝
unique_ptr<string> p3;
p3 = p2;                      // 错误：unique_ptr 不支持赋值
```

表 12.4：unique_ptr 操作（另参见表 12.1，第 401 页）	
unique_ptr<T> u1	空 unique_ptr，可以指向类型为 T 的对象。u1 会使用 delete 来释放它的指针；u2 会使用一个类型为 D 的可调用对象来释放它的指针
unique_ptr<T, D> u2	
unique_ptr<T, D> u(d)	空 unique_ptr，指向类型为 T 的对象，用类型为 D 的对象 d 代替 delete
u = nullptr	释放 u 指向的对象，将 u 置为空
u.release()	u 放弃对指针的控制权，返回指针，并将 u 置为空
u.reset()	释放 u 指向的对象
u.reset(q)	如果提供了内置指针 q，令 u 指向这个对象；否则将 u 置为空
u.reset(nullptr)	

虽然我们不能拷贝或赋值 unique_ptr，但可以通过调用 release 或 reset 将指针的所有权从一个（非 const）unique_ptr 转移给另一个 unique：

```
// 将所有权从 p1（指向 string Stegosaurus）转移给 p2
unique_ptr<string> p2(p1.release()); // release 将 p1 置为空
unique_ptr<string> p3(new string("Trex"));
// 将所有权从 p3 转移给 p2
p2.reset(p3.release()); // reset 释放了 p2 原来指向的内存
```

release 成员返回 unique_ptr 当前保存的指针并将其置为空。因此，p2 被初始化为 p1 原来保存的指针，而 p1 被置为空。

471 reset 成员接受一个可选的指针参数，令 unique_ptr 重新指向给定的指针。如果 unique_ptr 不为空，它原来指向的对象被释放。因此，对 p2 调用 reset 释放了用 "Stegosaurus"初始化的 string 所使用的内存，将 p3 对指针的所有权转移给 p2，并将 p3 置为空。

调用 release 会切断 unique_ptr 和它原来管理的对象间的联系。release 返回的指针通常被用来初始化另一个智能指针或给另一个智能指针赋值。在本例中，管理内存的责任简单地从一个智能指针转移给另一个。但是，如果我们不用另一个智能指针来保存 release 返回的指针，我们的程序就要负责资源的释放：

```
p2.release();              // 错误：p2 不会释放内存，而且我们丢失了指针
auto p = p2.release();     // 正确，但我们必须记得 delete(p)
```

传递 unique_ptr 参数和返回 unique_ptr

不能拷贝 unique_ptr 的规则有一个例外：我们可以拷贝或赋值一个将要被销毁的 unique_ptr。最常见的例子是从函数返回一个 unique_ptr：

```
unique_ptr<int> clone(int p) {
    // 正确：从 int*创建一个 unique_ptr<int>
    return unique_ptr<int>(new int(p));
}
```

还可以返回一个局部对象的拷贝：

```
unique_ptr<int> clone(int p) {
    unique_ptr<int> ret(new int (p));
    // ...
    return ret;
}
```

对于两段代码，编译器都知道要返回的对象将要被销毁。在此情况下，编译器执行一种特殊的“拷贝”，我们将在 13.6.2 节（第 473 页）中介绍它。

向后兼容：auto_ptr

标准库的较早版本包含了一个名为 auto_ptr 的类，它具有 unique_ptr 的部分特性，但不是全部。特别是，我们不能在容器中保存 auto_ptr，也不能从函数中返回 auto_ptr。

虽然 auto_ptr 仍是标准库的一部分，但编写程序时应该使用 unique_ptr。

向 unique_ptr 传递删除器

类似 shared_ptr，unique_ptr 默认情况下用 delete 释放它指向的对象。与 shared_ptr 一样，我们可以重载一个 unique_ptr 中默认的删除器（参见 12.1.4 节，第 415 页）。但是，unique_ptr 管理删除器的方式与 shared_ptr 不同，其原因我们将在 16.1.6 节（第 599 页）中介绍。 472

重载一个 unique_ptr 中的删除器会影响到 unique_ptr 类型以及如何构造（或 reset）该类型的对象。与重载关联容器的比较操作（参见 11.2.2 节，第 378 页）类似，我们必须在尖括号中 unique_ptr 指向类型之后提供删除器类型。在创建或 reset 一个这种 unique_ptr 类型的对象时，必须提供一个指定类型的可调用对象（删除器）：

```
// p 指向一个类型为 objT 的对象，并使用一个类型为 delT 的对象释放 objT 对象
// 它会调用一个名为 fcn 的 delT 类型对象
unique_ptr<objT, delT> p (new objT, fcn);
```

作为一个更具体的例子，我们将重写连接程序，用 unique_ptr 来代替 shared_ptr，如下所示：

```
void f(destination &d /* 其他需要的参数 */)
{
    connection c = connect(&d); // 打开连接
    // 当 p 被销毁时，连接将会关闭
    unique_ptr<connection, decltype(end_connection)*>
        p(&c, end_connection);
    // 使用连接
    // 当 f 退出时（即使是由于异常而退出），connection 会被正确关闭
}
```

在本例中我们使用了 decltype（参见 2.5.3 节，第 62 页）来指明函数指针类型。由于 decltype(end_connection) 返回一个函数类型，所以我们必须添加一个*来指出我们正在使用该类型的一个指针（参见 6.7 节，第 223 页）。

12.1.5 节练习

练习 12.16：如果你试图拷贝或赋值 unique_ptr，编译器并不总是能给出易于理解的错误信息。编写包含这种错误的程序，观察编译器如何诊断这种错误。

练习 12.17：下面的 unique_ptr 声明中，哪些是合法的，哪些可能导致后续的程序错误？解释每个错误的问题在哪里。

```
int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef unique_ptr<int> IntP;
(a) IntP p0(ix);                    (b) IntP p1(pi);
(c) IntP p2(pi2);                   (d) IntP p3(&ix);
(e) IntP p4(new int(2048));         (f) IntP p5(p2.get());
```

练习 12.18：shared_ptr 为什么没有 release 成员？

12.1.6 weak_ptr

473

C++ 11

weak_ptr（见表 12.5）是一种不控制所指向对象生存期的智能指针，它指向由一个 shared_ptr 管理的对象。将一个 weak_ptr 绑定到一个 shared_ptr 不会改变 shared_ptr 的引用计数。一旦最后一个指向对象的 shared_ptr 被销毁，对象就会被释放。即使有 weak_ptr 指向对象，对象也还是会被释放，因此，weak_ptr 的名字抓住了这种智能指针“弱”共享对象的特点。

表 12.5：weak_ptr

weak_ptr<T> w	空 weak_ptr 可以指向类型为 T 的对象
weak_ptr<T> w(sp)	与 shared_ptr sp 指向相同对象的 weak_ptr。T 必须能转换为 sp 指向的类型
w = p	p 可以是一个 shared_ptr 或一个 weak_ptr。赋值后 w 与 p 共享对象
w.reset()	将 w 置为空
w.use_count()	与 w 共享对象的 shared_ptr 的数量
w.expired()	若 w.use_count() 为 0，返回 true，否则返回 false
w.lock()	如果 expired 为 true，返回一个空 shared_ptr；否则返回一个指向 w 的对象的 shared_ptr

当我们创建一个 weak_ptr 时，要用一个 shared_ptr 来初始化它：

```
auto p = make_shared<int>(42);
weak_ptr<int> wp(p); // wp 弱共享 p; p 的引用计数未改变
```

本例中 wp 和 p 指向相同的对象。由于是弱共享，创建 wp 不会改变 p 的引用计数；wp 指向的对象可能被释放掉。

由于对象可能不存在，我们不能使用 weak_ptr 直接访问对象，而必须调用 lock。此函数检查 weak_ptr 指向的对象是否仍存在。如果存在，lock 返回一个指向共享对象的 shared_ptr。与任何其他 shared_ptr 类似，只要此 shared_ptr 存在，它所指向的底层对象也就会一直存在。例如：

```
if (shared_ptr<int> np = wp.lock()) { // 如果 np 不为空则条件成立
    // 在 if 中，np 与 p 共享对象
}
```

在这段代码中，只有当 lock 调用返回 true 时我们才会进入 if 语句体。在 if 中，使用 np 访问共享对象是安全的。

核查指针类

作为 weak_ptr 用途的一个展示，我们将为 StrBlob 类定义一个伴随指针类。我们

的指针类将命名为 StrBlobPtr，会保存一个 weak_ptr，指向 StrBlob 的 data 成员，这是初始化时提供给它的。通过使用 weak_ptr，不会影响一个给定的 StrBlob 所指向的 vector 的生存期。但是，可以阻止用户访问一个不再存在的 vector 的企图。 474

StrBlobPtr 会有两个数据成员：wptr，或者为空，或者指向一个 StrBlob 中的 vector；curr，保存当前对象所表示的元素的下标。类似它的伴随类 StrBlob，我们的指针类也有一个 check 成员来检查解引用 StrBlobPtr 是否安全：

```
// 对于访问一个不存在元素的尝试，StrBlobPtr 抛出一个异常
class StrBlobPtr {
public:
    StrBlobPtr(): curr(0) { }
    StrBlobPtr(StrBlob &a, size_t sz = 0):
            wptr(a.data), curr(sz) { }
    std::string& deref() const;
    StrBlobPtr& incr(); // 前缀递增
private:
    // 若检查成功，check 返回一个指向 vector 的 shared_ptr
    std::shared_ptr<std::vector<std::string>>
        check(std::size_t, const std::string&) const;
    // 保存一个 weak_ptr，意味着底层 vector 可能会被销毁
    std::weak_ptr<std::vector<std::string>> wptr;
    std::size_t curr; // 在数组中的当前位置
};
```

默认构造函数生成一个空的 StrBlobPtr。其构造函数初始化列表（参见 7.1.4 节，第 237 页）将 curr 显式初始化为 0，并将 wptr 隐式初始化为一个空 weak_ptr。第二个构造函数接受一个 StrBlob 引用和一个可选的索引值。此构造函数初始化 wptr，令其指向给定 StrBlob 对象的 shared_ptr 中的 vector，并将 curr 初始化为 sz 的值。我们使用了默认参数（参见 6.5.1 节，第 211 页），表示默认情况下将 curr 初始化为第一个元素的下标。我们将会看到，StrBlob 的 end 成员将会用到参数 sz。

值得注意的是，我们不能将 StrBlobPtr 绑定到一个 const StrBlob 对象。这个限制是由于构造函数接受一个非 const StrBlob 对象的引用而导致的。

StrBlobPtr 的 check 成员与 StrBlob 中的同名成员不同，它还要检查指针指向的 vector 是否还存在：

```
std::shared_ptr<std::vector<std::string>>
StrBlobPtr::check(std::size_t i, const std::string &msg) const
{
    auto ret = wptr.lock(); // vector 还存在吗?
    if (!ret)
        throw std::runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw std::out_of_range(msg);
    return ret; // 否则，返回指向 vector 的 shared_ptr
}
```

由于一个 weak_ptr 不参与其对应的 shared_ptr 的引用计数，StrBlobPtr 指向的 vector 可能已经被释放了。如果 vector 已销毁，lock 将返回一个空指针。在本例中，任何 vector 的引用都会失败，于是抛出一个异常。否则，check 会检查给定索引，如果索引值合法，check 返回从 lock 获得的 shared_ptr。 475

指针操作

我们将在第 14 章学习如何定义自己的运算符。现在，我们将定义名为 deref 和 incr 的函数，分别用来解引用和递增 StrBlobPtr。

deref 成员调用 check，检查使用 vector 是否安全以及 curr 是否在合法范围内：

```
std::string& StrBlobPtr::deref() const
{
    auto p = check(curr, "dereference past end");
    return (*p)[curr]; // (*p)是对象所指向的 vector
}
```

如果 check 成功，p 就是一个 shared_ptr，指向 StrBlobPtr 所指向的 vector。表达式(*p)[curr]解引用 shared_ptr 来获得 vector，然后使用下标运算符提取并返回 curr 位置上的元素。

incr 成员也调用 check：

```
// 前缀递增：返回递增后的对象的引用
StrBlobPtr& StrBlobPtr::incr()
{
    // 如果 curr 已经指向容器的尾后位置，就不能递增它
    check(curr, "increment past end of StrBlobPtr");
    ++curr; // 推进当前位置
    return *this;
}
```

当然，为了访问 data 成员，我们的指针类必须声明为 StrBlob 的 friend（参见 7.3.4 节，第 250 页）。我们还要为 StrBlob 类定义 begin 和 end 操作，返回一个指向它自身的 StrBlobPtr：

```
// 对于 StrBlob 中的友元声明来说，此前置声明是必要的
class StrBlobPtr;
class StrBlob {
    friend class StrBlobPtr;
    // 其他成员与 12.1.1 节（第 405 页）中声明相同
    // 返回指向首元素和尾后元素的 StrBlobPtr
    StrBlobPtr begin() { return StrBlobPtr(*this); }
    StrBlobPtr end()
        { auto ret = StrBlobPtr(*this, data->size());
          return ret; }
};
```

476

12.1.6 节练习

练习 12.19：定义你自己版本的 StrBlobPtr，更新 StrBlob 类，加入恰当的 friend 声明及 begin 和 end 成员。

练习 12.20：编写程序，逐行读入一个输入文件，将内容存入一个 StrBlob 中，用一个 StrBlobPtr 打印出 StrBlob 中的每个元素。

练习 12.21：也可以这样编写 StrBlobPtr 的 deref 成员：

```
std::string& deref() const
```

```
{ return (*check(curr, "dereference past end"))[curr]; }
```

你认为哪个版本更好？为什么？

练习 12.22：为了能让 StrBlobPtr 使用 const StrBlob，你觉得应该如何修改？定义一个名为 ConstStrBlobPtr 的类，使其能够指向 const StrBlob。



12.2 动态数组

new 和 delete 运算符一次分配/释放一个对象，但某些应用需要一次为很多对象分配内存的功能。例如，vector 和 string 都是在连续内存中保存它们的元素，因此，当容器需要重新分配内存时（参见 9.4 节，第 317 页），必须一次性为很多元素分配内存。

为了支持这种需求，C++语言和标准库提供了两种一次分配一个对象数组的方法。C++语言定义了另一种 new 表达式语法，可以分配并初始化一个对象数组。标准库中包含一个名为 allocator 的类，允许我们将分配和初始化分离。使用 allocator 通常会提供更好的性能和更灵活的内存管理能力，原因我们将在 12.2.2 节（第 427 页）中解释。

很多（可能是大多数）应用都没有直接访问动态数组的需求。当一个应用需要可变数量的对象时，我们在 StrBlob 中所采用的方法几乎总是更简单、更快速并且更安全的——即，使用 vector（或其他标准库容器）。如我们将在 13.6 节（第 470 页）中看到的，使用标准库容器的优势在新标准下更为显著。在支持新标准的标准库中，容器操作比之前的版本要快速得多。



大多数应用应该使用标准库容器而不是动态分配的数组。使用容器更为简单、更不容易出现内存管理错误并且可能有更好的性能。

如前所述，使用容器的类可以使用默认版本的拷贝、赋值和析构操作（参见 7.1.5 节，第 239 页）。分配动态数组的类则必须定义自己版本的操作，在拷贝、复制以及销毁对象时管理所关联的内存。



直到学习完第 13 章，不要在类内的代码中分配动态内存。

12.2.1 new 和数组



为了让 new 分配一个对象数组，我们要在类型名之后跟一对方括号，在其中指明要分配的对象的数目。在下例中，new 分配要求数量的对象并（假定分配成功后）返回指向第一个对象的指针：

477

```
// 调用 get_size 确定分配多少个 int
int *pia = new int[get_size()]; // pia 指向第一个 int
```

方括号中的大小必须是整型，但不必是常量。

也可以用一个表示数组类型的类型别名（参见 2.5.1 节，第 60 页）来分配一个数组，这样，new 表达式中就不需要方括号了：

```
typedef int arrT[42];      // arrT 表示 42 个 int 的数组类型
int *p = new arrT;         // 分配一个 42 个 int 的数组；p 指向第一个 int
```

在本例中，new 分配一个 int 数组，并返回指向第一个 int 的指针。即使这段代码中没

有方括号，编译器执行这个表达式时还是会用 new[]。即，编译器执行如下形式：

```
int *p = new int[42];
```

分配一个数组会得到一个元素类型的指针

虽然我们通常称 new T[]分配的内存为“动态数组”，但这种叫法某种程度上有些误导。当用 new 分配一个数组时，我们并未得到一个数组类型的对象，而是得到一个数组元素类型的指针。即使我们使用类型别名定义了一个数组类型，new 也不会分配一个数组类型的对象。在上例中，我们正在分配一个数组的事实甚至都是不可见的——连[*num*]都没有。new 返回的是一个元素类型的指针。

由于分配的内存并不是一个数组类型，因此不能对动态数组调用 begin 或 end（参见 3.5.3 节，第 106 页）。这些函数使用数组维度（回忆一下，维度是数组类型的一部分）来返回指向首元素和尾后元素的指针。出于相同的原因，也不能用范围 for 语句来处理（所谓的）动态数组中的元素。

C++ 11



要记住我们所说的动态数组并不是数组类型，这是很重要的。

初始化动态分配对象的数组

默认情况下，new 分配的对象，不管是单个分配的还是数组中的，都是默认初始化的。可以对数组中的元素进行值初始化（参见 3.3.1 节，第 88 页），方法是在大小之后跟一对空括号。

```
int *pia = new int[10];           // 10 个未初始化的 int
int *pia2 = new int[10]();        // 10 个值初始化为 0 的 int
string *psa = new string[10];     // 10 个空 string
string *psa2 = new string[10](); // 10 个空 string
```

478 在新标准中，我们还可以提供一个元素初始化器的花括号列表：

C++ 11

```
// 10 个 int 分别用列表中对应的初始化器初始化
int *pia3 = new int[10]{0,1,2,3,4,5,6,7,8,9};
// 10 个 string，前 4 个用给定的初始化器初始化，剩余的进行值初始化
string *psa3 = new string[10]{"a", "an", "the", string(3,'x')};
```

与内置数组对象的列表初始化（参见 3.5.1 节，第 102 页）一样，初始化器会用来初始化动态数组中开始部分的元素。如果初始化器数目小于元素数目，剩余元素将进行值初始化。如果初始化器数目大于元素数目，则 new 表达式失败，不会分配任何内存。在本例中，new 会抛出一个类型为 bad_array_new_length 的异常。类似 bad_alloc，此类型定义在头文件 new 中。

C++ 11

虽然我们用空括号对数组中元素进行值初始化，但不能在括号中给出初始化器，这意味着不能用 auto 分配数组（参见 12.1.2 节，第 407 页）。

动态分配一个空数组是合法的

可以用任意表达式来确定要分配的对象的数目：

```
size_t n = get_size();    // get_size 返回需要的元素的数目
int* p = new int[n];      // 分配数组保存元素
for (int* q = p; q != p + n; ++q)
     /* 处理数组 */ ;
```

这产生了一个有意思的问题：如果 get_size 返回 0，会发生什么？答案是代码仍能正常工作。虽然我们不能创建一个大小为 0 的静态数组对象，但当 n 等于 0 时，调用 new[n] 是合法的：

```
char arr[0];              // 错误：不能定义长度为 0 的数组
char *cp = new char[0];   // 正确：但 cp 不能解引用
```

当我们用 new 分配一个大小为 0 的数组时，new 返回一个合法的非空指针。此指针保证与 new 返回的其他任何指针都不相同。对于零长度的数组来说，此指针就像尾后指针一样（参见 3.5.3 节，第 106 页），我们可以像使用尾后迭代器一样使用这个指针。可以用此指针进行比较操作，就像上面循环代码中那样。可以向此指针加上（或从此指针减去）0，也可以从此指针减去自身从而得到 0。但此指针不能解引用——毕竟它不指向任何元素。

在我们假想的循环中，若 get_size 返回 0，则 n 也是 0，new 会分配 0 个对象。for 循环中的条件会失败（p 等于 q+n，因为 n 为 0）。因此，循环体不会被执行。

释放动态数组

为了释放动态数组，我们使用一种特殊形式的 delete——在指针前加上一个空方括号对：

```
delete p;          // p 必须指向一个动态分配的对象或为空
delete [] pa;      // pa 必须指向一个动态分配的数组或为空
```

479

第二条语句销毁 pa 指向的数组中的元素，并释放对应的内存。数组中的元素按逆序销毁，即，最后一个元素首先被销毁，然后是倒数第二个，依此类推。

当我们释放一个指向数组的指针时，空方括号对是必需的：它指示编译器此指针指向一个对象数组的第一个元素。如果我们在 delete 一个指向数组的指针时忽略了方括号（或者在 delete 一个指向单一对象的指针时使用了方括号），其行为是未定义的。

回忆一下，当我们使用一个类型别名来定义一个数组类型时，在 new 表达式中不使用[]。即使是这样，在释放一个数组指针时也必须使用方括号：

```
typedef int arrT[42];     // arrT 是 42 个 int 的数组的类型别名
int *p = new arrT;        // 分配一个 42 个 int 的数组；p 指向第一个元素
delete [] p;              // 方括号是必需的，因为我们当初分配的是一个数组
```

不管外表如何，p 指向一个对象数组的首元素，而不是一个类型为 arrT 的单一对象。因此，在释放 p 时我们必须使用[]。



如果我们在 delete 一个数组指针时忘记了方括号，或者在 delete 一个单一对象的指针时使用了方括号，编译器很可能不会给出警告。我们的程序可能在执行过程中在没有任何警告的情况下行为异常。

智能指针和动态数组

标准库提供了一个可以管理 new 分配的数组的 unique_ptr 版本。为了用一个 unique_ptr 管理动态数组，我们必须在对象类型后面跟一对空方括号：

```
// up 指向一个包含 10 个未初始化 int 的数组
unique_ptr<int[]> up(new int[10]);
up.release(); // 自动用 delete[] 销毁其指针
```

类型说明符中的方括号（<int[]>）指出 up 指向一个 int 数组而不是一个 int。由于 up 指向一个数组，当 up 销毁它管理的指针时，会自动使用 delete[]。

指向数组的 unique_ptr 提供的操作与我们在 12.1.5 节（第 417 页）中使用的那些操作有一些不同，我们在表 12.6 中描述了这些操作。当一个 unique_ptr 指向一个数组时，我们不能使用点和箭头成员运算符。毕竟 unique_ptr 指向的是一个数组而不是单个对象，因此这些运算符是无意义的。另一方面，当一个 unique_ptr 指向一个数组时，我们可以使用下标运算符来访问数组中的元素：

```
for (size_t i = 0; i != 10; ++i)
    up[i] = i; // 为每个元素赋予一个新值
```

480

表 12.6：指向数组的 unique_ptr

指向数组的 unique_ptr 不支持成员访问运算符（点和箭头运算符）。	
其他 unique_ptr 操作不变。	
unique_ptr<T[]> u	u 可以指向一个动态分配的数组，数组元素类型为 T
unique_ptr<T[]> u(p)	u 指向内置指针 p 所指向的动态分配的数组。p 必须能转换为类型 T*（参见 4.11.2 节，第 143 页）
u[i]	返回 u 拥有的数组中位置 i 处的对象
	u 必须指向一个数组

与 unique_ptr 不同，shared_ptr 不直接支持管理动态数组。如果希望使用 shared_ptr 管理一个动态数组，必须提供自己定义的删除器：

```
// 为了使用 shared_ptr，必须提供一个删除器
shared_ptr<int> sp(new int[10], [](int *p) { delete[] p; });
sp.reset(); // 使用我们提供的 lambda 释放数组，它使用 delete[]
```

本例中我们传递给 shared_ptr 一个 lambda（参见 10.3.2 节，第 346 页）作为删除器，它使用 delete[]释放数组。

如果未提供删除器，这段代码将是未定义的。默认情况下，shared_ptr 使用 delete 销毁它指向的对象。如果此对象是一个动态数组，对其使用 delete 所产生的问题与释放一个动态数组指针时忘记[]产生的问题一样（参见 12.2.1 节，第 425 页）。

shared_ptr 不直接支持动态数组管理这一特性会影响我们如何访问数组中的元素：

```
// shared_ptr 未定义下标运算符，并且不支持指针的算术运算
for (size_t i = 0; i != 10; ++i)
    *(sp.get() + i) = i; // 使用 get 获取一个内置指针
```

shared_ptr 未定义下标运算符，而且智能指针类型不支持指针算术运算。因此，为了访问数组中的元素，必须用 get 获取一个内置指针，然后用它来访问数组元素。

12.2.1 节练习

练习 12.23：编写一个程序，连接两个字符串字面常量，将结果保存在一个动态分配的 char 数组中。重写这个程序，连接两个标准库 string 对象。

练习 12.24：编写一个程序，从标准输入读取一个字符串，存入一个动态分配的字符数组中。描述你的程序如何处理变长输入。测试你的程序，输入一个超出你分配的数组长度的字符串。

练习 12.25：给定下面的 new 表达式，你应该如何释放 pa？

```
int *pa = new int[10];
```

12.2.2 allocator 类



new 有一些灵活性上的局限，其中一方面表现在它将内存分配和对象构造组合在了一起。类似的，delete 将对象析构和内存释放组合在了一起。我们分配单个对象时，通常希望将内存分配和对象初始化组合在一起。因为在这种情况下，我们几乎肯定知道对象应有什么值。

481

当分配一大块内存时，我们通常计划在这块内存上按需构造对象。在此情况下，我们希望将内存分配和对象构造分离。这意味着我们可以分配大块内存，但只在真正需要时才真正执行对象创建操作（同时付出一定开销）。

一般情况下，将内存分配和对象构造组合在一起可能会导致不必要的浪费。例如：

```
string *const p = new string[n]; // 构造 n 个空 string
string s;
string *q = p;                   // q 指向第一个 string
while (cin >> s && q != p + n)
    *q++ = s;                    // 赋予*q 一个新值
const size_t size = q - p;       // 记住我们读取了多少个 string
// 使用数组
delete[] p; // p 指向一个数组；记得用 delete[] 来释放
```

new 表达式分配并初始化了 n 个 string。但是，我们可能不需要 n 个 string，少量 string 可能就足够了。这样，我们就可能创建了一些永远也用不到的对象。而且，对于那些确实要使用的对象，我们也在初始化之后立即赋予了它们新值。每个使用到的元素都被赋值了两次：第一次是在默认初始化时，随后是在赋值时。

更重要的是，那些没有默认构造函数的类就不能动态分配数组了。

allocator 类

标准库 **allocator** 类定义在头文件 memory 中，它帮助我们将内存分配和对象构造分离开来。它提供一种类型感知的内存分配方法，它分配的内存是原始的、未构造的。表 12.7 概述了 allocator 支持的操作。在本节中，我们将介绍这些 allocator 操作。在 13.5 节（第 464 页），我们将看到如何使用这个类的典型例子。

类似 vector，allocator 是一个模板（参见 3.3 节，第 86 页）。为了定义一个 allocator 对象，我们必须指明这个 allocator 可以分配的对象类型。当一个 allocator 对象分配内存时，它会根据给定的对象类型来确定恰当的内存大小和对齐位置：

```
allocator<string> alloc;              // 可以分配 string 的 allocator 对象
auto const p = alloc.allocate(n);     // 分配 n 个未初始化的 string
```

这个 allocate 调用为 n 个 string 分配了内存。

482

表 12.7：标准库 allocator 类及其算法	
`allocator<T> a`	定义了一个名为 a 的 allocator 对象，它可以为类型为 T 的对象分配内存
`a.allocate(n)`	分配一段原始的、未构造的内存，保存 n 个类型为 T 的对象
`a.deallocate(p, n)`	释放从 T*指针 p 中地址开始的内存，这块内存保存了 n 个类型为 T 的对象；p 必须是一个先前由 allocate 返回的指针，且 n 必须是 p 创建时所要求的大小。在调用 deallocate 之前，用户必须对每个在这块内存中创建的对象调用 destroy
`a.construct(p, args)`	p 必须是一个类型为 T*的指针，指向一块原始内存；*arg* 被传递给类型为 T 的构造函数，用来在 p 指向的内存中构造一个对象
`a.destroy(p)`	p 为 T*类型的指针，此算法对 p 指向的对象执行析构函数（参见 12.1.1 节，第 402 页）

allocator 分配未构造的内存

allocator 分配的内存是未构造的（unconstructed）。我们按需要在此内存中构造对象。在新标准库中，construct 成员函数接受一个指针和零个或多个额外参数，在给定位置构造一个元素。额外参数用来初始化构造的对象。类似 make_shared 的参数（参见 12.1.1 节，第 401 页），这些额外参数必须是与构造的对象的类型相匹配的合法的初始化器：

C++ 11

```
auto q = p; // q 指向最后构造的元素之后的位置
alloc.construct(q++);              // *q 为空字符串
alloc.construct(q++, 10, 'c');     // *q 为 cccccccccc
alloc.construct(q++, "hi");        // *q 为 hi!
```

在早期版本的标准库中，construct 只接受两个参数：指向创建对象位置的指针和一个元素类型的值。因此，我们只能将一个元素拷贝到未构造空间中，而不能用元素类型的任何其他构造函数来构造一个元素。

还未构造对象的情况下就使用原始内存是错误的：

```
cout << *p << endl; // 正确：使用 string 的输出运算符
cout << *q << endl; // 灾难：q 指向未构造的内存！
```



为了使用 allocate 返回的内存，我们必须用 construct 构造对象。使用未构造的内存，其行为是未定义的。

当我们用完对象后，必须对每个构造的元素调用 destroy 来销毁它们。函数 destroy 接受一个指针，对指向的对象执行析构函数（参见 12.1.1 节，第 402 页）：

483

```
while (q != p)
    alloc.destroy(--q); // 释放我们真正构造的 string
```

在循环开始处，q 指向最后构造的元素之后的位置。我们在调用 destroy 之前对 q 进行了递减操作。因此，第一次调用 destroy 时，q 指向最后一个构造的元素。最后一步循环中我们 destroy 了第一个构造的元素，随后 q 将与 p 相等，循环结束。



我们只能对真正构造了的元素进行 destroy 操作。

一旦元素被销毁后，就可以重新使用这部分内存来保存其他 string，也可以将其归

还给系统。释放内存通过调用 deallocate 来完成：

```
alloc.deallocate(p, n);
```

我们传递给 deallocate 的指针不能为空，它必须指向由 allocate 分配的内存。而且，传递给 deallocate 的大小参数必须与调用 allocated 分配内存时提供的大小参数具有一样的值。

拷贝和填充未初始化内存的算法

标准库还为 allocator 类定义了两个伴随算法，可以在未初始化内存中创建对象。表 12.8 描述了这些函数，它们都定义在头文件 memory 中。

表 12.8：allocator 算法

这些函数在给定目的位置创建元素，而不是由系统分配内存给它们。

uninitialized_copy(b,e,b2)	从迭代器 b 和 e 指出的输入范围中拷贝元素到迭代器 b2 指定的未构造的原始内存中。b2 指向的内存必须足够大，能容纳输入序列中元素的拷贝
uninitialized_copy_n(b,n,b2)	从迭代器 b 指向的元素开始，拷贝 n 个元素到 b2 开始的内存中
uninitialized_fill(b,e,t)	在迭代器 b 和 e 指定的原始内存范围中创建对象，对象的值均为 t 的拷贝
uninitialized_fill_n(b,n,t)	从迭代器 b 指向的内存地址开始创建 n 个对象。b 必须指向足够大的未构造的原始内存，能够容纳给定数量的对象

作为一个例子，假定有一个 int 的 vector，希望将其内容拷贝到动态内存中。我们将分配一块比 vector 中元素所占用空间大一倍的动态内存，然后将原 vector 中的元素拷贝到前一半空间，对后一半空间用一个给定值进行填充：

```
// 分配比 vi 中元素所占用空间大一倍的动态内存
auto p = alloc.allocate(vi.size() * 2);
// 通过拷贝 vi 中的元素来构造从 p 开始的元素
auto q = uninitialized_copy(vi.begin(), vi.end(), p);
// 将剩余元素初始化为 42
uninitialized_fill_n(q, vi.size(), 42);
```

484

类似拷贝算法（参见 10.2.2 节，第 341 页），uninitialized_copy 接受三个迭代器参数。前两个表示输入序列，第三个表示这些元素将要拷贝到的目的空间。传递给 uninitialized_copy 的目的位置迭代器必须指向未构造的内存。与 copy 不同，uninitialized_copy 在给定目的位置构造元素。

类似 copy，uninitialized_copy 返回（递增后的）目的位置迭代器。因此，一次 uninitialized_copy 调用会返回一个指针，指向最后一个构造的元素之后的位置。在本例中，我们将此指针保存在 q 中，然后将 q 传递给 uninitialized_fill_n。此函数类似 fill_n（参见 10.2.2 节，第 340 页），接受一个指向目的位置的指针、一个计数和一个值。它会在目的位置指针指向的内存中创建给定数目个对象，用给定值对它们进行初始化。

12.2.2 节练习

练习 12.26：用 allocator 重写第 427 页中的程序。

12.3 使用标准库：文本查询程序

我们将实现一个简单的文本查询程序，作为标准库相关内容学习的总结。我们的程序允许用户在一个给定文件中查询单词。查询结果是单词在文件中出现的次数及其所在行的列表。如果一个单词在一行中出现多次，此行只列出一次。行会按照升序输出——即，第 7 行会在第 9 行之前显示，依此类推。

例如，我们可能读入一个包含本章内容（指英文版中的文本）的文件，在其中寻找单词 element。输出结果的前几行应该是这样的：

```
element occurs 112 times
    (line 36) A set element contains only a key;
    (line 158) operator creates a new element
    (line 160) Regardless of whether the element
    (line 168) When we fetch an element from a map, we
    (line 214) If the element is not found, find returns
```

接下来还有大约 100 行，都是单词 element 出现的位置。

12.3.1 文本查询程序设计

485

开始一个程序的设计的一种好方法是列出程序的操作。了解需要哪些操作会帮助我们分析出需要什么样的数据结构。从需求入手，我们的文本查询程序需要完成如下任务：

- 当程序读取输入文件时，它必须记住单词出现的每一行。因此，程序需要逐行读取输入文件，并将每一行分解为独立的单词
- 当程序生成输出时，
  - 它必须能提取每个单词所关联的行号
  - 行号必须按升序出现且无重复
  - 它必须能打印给定行号中的文本。

利用多种标准库设施，我们可以很漂亮地实现这些要求：

- 我们将使用一个 vector<string>来保存整个输入文件的一份拷贝。输入文件中的每行保存为 vector 中的一个元素。当需要打印一行时，可以用行号作为下标来提取行文本。
- 我们使用一个 istringstream（参见 8.3 节，第 287 页）来将每行分解为单词。
- 我们使用一个 set 来保存每个单词在输入文本中出现的行号。这保证了每行只出现一次且行号按升序保存。
- 我们使用一个 map 来将每个单词与它出现的行号 set 关联起来。这样我们就可以方便地提取任意单词的 set。

我们的解决方案还使用了 shared_ptr，原因稍后进行解释。

数据结构

虽然我们可以用 vector、set 和 map 来直接编写文本查询程序，但如果定义一个更

为抽象的解决方案，会更为有效。我们将从定义一个保存输入文件的类开始，这会令文件查询更为容易。我们将这个类命名为 TextQuery，它包含一个 vector 和一个 map。vector 用来保存输入文件的文本，map 用来关联每个单词和它出现的行号的 set。这个类将会有一个用来读取给定输入文件的构造函数和一个执行查询的操作。

查询操作要完成的任务非常简单：查找 map 成员，检查给定单词是否出现。设计这个函数的难点是确定应该返回什么内容。一旦找到了一个单词，我们需要知道它出现了多少次、它出现的行号以及每行的文本。

返回所有这些内容的最简单的方法是定义另一个类，可以命名为 QueryResult，来保存查询结果。这个类会有一个 print 函数，完成结果打印工作。

在类之间共享数据

486

我们的 QueryResult 类要表达查询的结果。这些结果包括与给定单词关联的行号的 set 和这些行对应的文本。这些数据都保存在 TextQuery 类型的对象中。

由于 QueryResult 所需要的数据都保存在一个 TextQuery 对象中，我们就必须确定如何访问它们。我们可以拷贝行号的 set，但这样做可能很耗时。而且，我们当然不希望拷贝 vector，因为这可能会引起整个文件的拷贝，而目标只不过是为了打印文件的一小部分而已（通常会是这样）。

通过返回指向 TextQuery 对象内部的迭代器（或指针），我们可以避免拷贝操作。但是，这种方法开启了一个陷阱：如果 TextQuery 对象在对应的 QueryResult 对象之前被销毁，会发生什么？在此情况下，QueryResult 就将引用一个不再存在的对象中的数据。

对于 QueryResult 对象和对应的 TextQuery 对象的生存期应该同步这一观察结果，其实已经暗示了问题的解决方案。考虑到这两个类概念上“共享”了数据，可以使用 shared_ptr（参见 12.1.1 节，第 400 页）来反映数据结构中的这种共享关系。

使用 TextQuery 类

当我们设计一个类时，在真正实现成员之前先编写程序使用这个类，是一种非常有用的方法。通过这种方法，可以看到类是否具有我们所需要的操作。例如，下面的程序使用了 TextQuery 和 QueryResult 类。这个函数接受一个指向要处理的文件的 ifstream，并与用户交互，打印给定单词的查询结果

```
void runQueries(ifstream &infile)
{
    // infile 是一个 ifstream，指向我们要处理的文件
    TextQuery tq(infile); // 保存文件并建立查询 map
    // 与用户交互：提示用户输入要查询的单词，完成查询并打印结果
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        // 若遇到文件尾或用户输入了'q'时循环终止
        if (!(cin >> s) || s == "q") break;
        // 指向查询并打印结果
        print(cout, tq.query(s)) << endl;
    }
}
```

我们首先用给定的 ifstream 初始化一个名为 tq 的 TextQuery 对象。TextQuery 的构造函数读取输入文件，保存在 vector 中，并建立单词到所在行号的 map。

while（无限）循环提示用户输入一个要查询的单词，并打印出查询结果，如此往复。循环条件检测字面常量 true（参见 2.1.3 节，第 37 页），因此永远成功。循环的退出是通
487 过 if 语句中的 break（参见 5.5.1 节，第 170 页）实现的。此 if 语句检查输入是否成功。如果成功，它再检查用户是否输入了 q。输入失败或用户输入了 q 都会使循环终止。一旦用户输入了要查询的单词，我们要求 tq 查找这个单词，然后调用 print 打印搜索结果。

12.3.1 节练习

练习 12.27：TextQuery 和 QueryResult 类只使用了我们已经介绍过的语言和标准库特性。不要提前看后续章节内容，只用已经学到的知识对这两个类编写你自己的版本。

练习 12.28：编写程序实现文本查询，不要定义类来管理数据。你的程序应该接受一个文件，并与用户交互来查询单词。使用 vector、map 和 set 容器来保存来自文件的数据并生成查询结果。

练习 12.29：我们曾经用 do while 循环来编写管理用户交互的循环（参见 5.4.4 节，第 169 页）。用 do while 重写本节程序，解释你倾向于哪个版本，为什么。



12.3.2 文本查询程序类的定义

我们以 TextQuery 类的定义开始。用户创建此类的对象时会提供一个 istream，用来读取输入文件。这个类还提供一个 query 操作，接受一个 string，返回一个 QueryResult 表示 string 出现的那些行。

设计类的数据成员时，需要考虑与 QueryResult 对象共享数据的需求。QueryResult 类需要共享保存输入文件的 vector 和保存单词关联的行号的 set。因此，这个类应该有两个数据成员：一个指向动态分配的 vector（保存输入文件）的 shared_ptr 和一个 string 到 shared_ptr<set>的 map。map 将文件中每个单词关联到一个动态分配的 set 上，而此 set 保存了该单词所出现的行号。

为了使代码更易读，我们还会定义一个类型成员（参见 7.3.1 节，第 243 页）来引用行号，即 string 的 vector 中的下标：

```
class QueryResult; // 为了定义函数 query 的返回类型，这个定义是必需的
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
private:
    std::shared_ptr<std::vector<std::string>> file; // 输入文件
    // 每个单词到它所在的行号的集合的映射
    std::map<std::string,
             std::shared_ptr<std::set<line_no>>> wm;
};
```

488 这个类定义最困难的部分是解开类名。与往常一样，对于可能置于头文件中的代码，在使用标准库名字时要加上 std::（参见 3.1 节，第 74 页）。在本例中，我们反复使用了 std::，

使得代码开始可能有些难读。例如，

```
std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
```

如果写成下面的形式可能就更好理解一些

```
map<string, shared_ptr<set<line_no>>> wm;
```

TextQuery 构造函数

TextQuery 的构造函数接受一个 ifstream，逐行读取输入文件：

```
// 读取输入文件并建立单词到行号的映射
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
    string text;
    while (getline(is, text)) {          // 对文件中每一行
        file->push_back(text);           // 保存此行文本
        int n = file->size() - 1;        // 当前行号
        istringstream line(text);        // 将行文本分解为单词
        string word;
        while (line >> word) {           // 对行中每个单词
            // 如果单词不在 wm 中，以之为下标在 wm 中添加一项
            auto &lines = wm[word];  // lines 是一个 shared_ptr
            if (!lines) // 在我们第一次遇到这个单词时，此指针为空
                lines.reset(new set<line_no>); // 分配一个新的 set
            lines->insert(n);            // 将此行号插入 set 中
        }
    }
}
```

构造函数的初始化器分配一个新的 vector 来保存输入文件中的文本。我们用 getline 逐行读取输入文件，并存入 vector 中。由于 file 是一个 shared_ptr，我们用->运算符解引用 file 来提取 file 指向的 vector 对象的 push_back 成员。

接下来我们用一个 istringstream（参见 8.3 节，第 287 页）来处理刚刚读入的一行中的每个单词。内层 while 循环用 istringstream 的输入运算符来从当前行读取每个单词，存入 word 中。在 while 循环内，我们用 map 下标运算符提取与 word 相关联的 shared_ptr<set>，并将 lines 绑定到此指针。注意，lines 是一个引用，因此改变 lines 也会改变 wm 中的元素。

若 word 不在 map 中，下标运算符会将 word 添加到 wm 中（参见 11.3.4 节，第 387 页），与 word 关联的值进行值初始化。这意味着，如果下标运算符将 word 添加到 wm 中，lines 将是一个空指针。如果 lines 为空，我们分配一个新的 set，并调用 reset 更新 lines 引用的 shared_ptr，使其指向这个新分配的 set。

不管是否创建了一个新的 set，我们都调用 insert 将当前行号添加到 set 中。由于 lines 是一个引用，对 insert 的调用会将新元素添加到 wm 中的 set 中。如果一个给定单词在同一行中出现多次，对 insert 的调用什么都不会做。 489

QueryResult 类

QueryResult 类有三个数据成员：一个 string，保存查询单词；一个 shared_ptr，指向保存输入文件的 vector；一个 shared_ptr，指向保存单词出现行号的 set。它唯

一的一个成员函数是一个构造函数，初始化这三个数据成员：

```
class QueryResult {
friend std::ostream& print(std::ostream&, const QueryResult&);
public:
QueryResult(std::string s,
            std::shared_ptr<std::set<line_no>> p,
            std::shared_ptr<std::vector<std::string>> f):
    sought(s), lines(p), file(f) { }
private:
    std::string sought; // 查询单词
    std::shared_ptr<std::set<line_no>> lines;          // 出现的行号
    std::shared_ptr<std::vector<std::string>> file;    // 输入文件
};
```

构造函数的唯一工作是将参数保存在对应的数据成员中，这是在其初始化器列表中完成的（参见 7.1.4 节，第 237 页）。

query 函数

query 函数接受一个 string 参数，即查询单词，query 用它来在 map 中定位对应的行号 set。如果找到了这个 string，query 函数构造一个 QueryResult，保存给定 string、TextQuery 的 file 成员以及从 wm 中提取的 set。

唯一的问题是：如果给定 string 未找到，我们应该返回什么？在这种情况下，没有可返回的 set。为了解决此问题，我们定义了一个局部 static 对象，它是一个指向空的行号 set 的 shared_ptr。当未找到给定单词时，我们返回此对象的一个拷贝：

```
QueryResult
TextQuery::query(const string &sought) const
{
    // 如果未找到 sought，我们将返回一个指向此 set 的指针
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // 使用 find 而不是下标运算符来查找单词，避免将单词添加到 wm 中！
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // 未找到
    else
        return QueryResult(sought, loc->second, file);
}
```

490 ### 打印结果

print 函数在给定的流上打印出给定的 QueryResult 对象：

```
ostream &print(ostream & os, const QueryResult &qr)
{
    // 如果找到了单词，打印出现次数和所有出现的位置
    os << qr.sought << " occurs " << qr.lines->size() << " "
       << make_plural(qr.lines->size(), "time", "s") << endl;
    // 打印单词出现的每一行
    for (auto num : *qr.lines) // 对 set 中每个单词
        // 避免行号从 0 开始给用户带来的困惑
        os << "\t(line " << num + 1 << ") "
```

```
            << *(qr.file->begin() + num) << endl;
    return os;
}
```

我们调用 qr.lines 指向的 set 的 size 成员来报告单词出现了多少次。由于 set 是一个 shared_ptr，必须解引用 lines。调用 make_plural（参见 6.3.2 节，第 201 页）来根据大小是否等于 1 打印 time 或 times。

在 for 循环中，我们遍历 lines 所指向的 set。for 循环体打印行号，并按人们习惯的方式调整计数值。set 中的数值就是 vector 中元素的下标，从 0 开始编号。但大多数用户认为第一行的行号应该是 1，因此我们对每个行号都加上 1，转换为人们更习惯的形式。

我们用行号从 file 指向的 vector 中提取一行文本。回忆一下，当给一个迭代器加上一个数时，会得到 vector 中相应偏移之后位置的元素（参见 3.4.2 节，第 99 页）。因此，file->begin()+num 即为 file 指向的 vector 中第 num 个位置的元素。

注意此函数能正确处理未找到单词的情况。在此情况下，set 为空。第一条输出语句会注意到单词出现了 0 次。由于*res.lines 为空，for 循环一次也不会执行。

12.3.2 节练习

练习 12.30：定义你自己版本的 TextQuery 和 QueryResult 类，并执行 12.3.1 节（第 431 页）中的 runQueries 函数。

练习 12.31：如果用 vector 代替 set 保存行号，会有什么差别？哪种方法更好？为什么？

练习 12.32：重写 TextQuery 和 QueryResult 类，用 StrBlob 代替 vector<string> 保存输入文件。

练习 12.33：在第 15 章中我们将扩展查询系统，在 QueryResult 类中将会需要一些额外的成员。添加名为 begin 和 end 的成员，返回一个迭代器，指向一个给定查询返回的行号的 set 中的位置。再添加一个名为 get_file 的成员，返回一个 shared_ptr，指向 QueryResult 对象中的文件。

491

小结

在C++中，内存是通过new表达式分配，通过delete表达式释放的。标准库还定义了一个allocator类来分配动态内存块。

分配动态内存的程序应负责释放它所分配的内存。内存的正确释放是非常容易出错的地方：要么内存永远不会被释放，要么在仍有指针引用它时就被释放了。新的标准库定义了智能指针类型——shared_ptr、unique_ptr和weak_ptr，可令动态内存管理更为安全。对于一块内存，当没有任何用户使用它时，智能指针会自动释放它。现代C++程序应尽可能使用智能指针。

术语表

allocator 标准库类，用来分配未构造的内存。

空悬指针（dangling pointer） 一个指针，指向曾经保存一个对象但现在已释放的内存。众所周知，空悬指针引起的程序错误非常难以调试。

delete 释放new分配的内存。delete p释放对象，delete []p释放p指向的数组。p可以为空，或者指向new分配的内存。

释放器（deleter） 传递给智能指针的函数，用来代替delete释放指针绑定的对象。

析构函数（destructor） 特殊的成员函数，负责在对象离开作用域或被释放时完成清理工作。

动态分配的（dynamically allocated） 在自由空间中分配的对象。在自由空间中分配的对象直到被显式释放或程序结束才会销毁。

自由空间（free store） 程序可用的内存池，保存动态分配的对象。

堆（heap） 自由空间的同义词。

new 从自由空间分配内存。new T分配并构造一个类型为T的对象，并返回一个指向该对象的指针。如果T是一个数组类型，new返回一个指向数组首元素的指针。类似的，new [n] T分配n个类型为T的对象，并返回指向数组首元素的指针。默认情况下，分配的对象进行默认初始化。我们也可以提供可选的初始化器。

定位new（placement new） 一种new表达式形式，接受一些额外的参数，在new关键字后面的括号中给出。例如，new (nothrow) int告诉new不要抛出异常。

引用计数（reference count） 一个计数器，记录有多少用户共享一个对象。智能指针用它来判断什么时候释放所指向的对象是安全的。

shared_ptr 提供所有权共享的智能指针：对共享对象来说，当最后一个指向它的shared_ptr被销毁时会被释放。

智能指针（smart pointer） 标准库类型，行为类似指针，但可以检查什么时候使用指针是安全的。智能指针类型负责在恰当的时候释放内存。

unique_ptr 提供独享所有权的智能指针：当unique_ptr被销毁时，它指向的对象被释放。unique_ptr不能直接拷贝或赋值。

weak_ptr 一种智能指针，指向由shared_ptr管理的对象。在确定是否应释放对象时，shared_ptr并不把weak_ptr统计在内。

第III部分
类设计者的工具

内容

第 13 章　拷贝控制 439
第 14 章　操作重载与类型转换 489
第 15 章　面向对象程序设计 525
第 16 章　模板与泛型编程 577

类是C++的核心概念。我们已经从第7章开始详细介绍了如何定义类。第7章涵盖了使用类的所有基本知识：类作用域、数据隐藏以及构造函数，还介绍了类的一些重要特性：成员函数、隐式 `this` 指针、友元以及 `const`、`static` 和 `mutable` 成员。在第III部分中，我们将延伸类的有关话题的讨论，将介绍拷贝控制、重载运算符、继承和模板。

如前所述，在C++中，我们通过定义构造函数来控制在类类型的对象初始化时做什么。类还可以控制在对象拷贝、赋值、移动和销毁时做什么。在这方面，C++与其他语言是不同的，其他很多语言都没有给予类设计者控制这些操作的能力。第13章将介绍这些内容。本章还会介绍新标准引入的两个重要概念：右值引用和移动操作。

第14章介绍运算符重载，这种机制允许内置运算符作用于类类型的运算对象。这样，我们创建的类型直观上就可以像内置类型一样使用，运算符重载是C++借以实现这一目的的方法之一。

类可以重载的运算符中有一种特殊的运算符——函数调用运算符。对于重载了这种运

算符的类，我们可以“调用”其对象，就好像它们是函数一样。新标准库中提供了一些设施，使得不同类型的可调用对象可以以一种一致的方式来使用，我们也将介绍这部分内容。

第14章最后将介绍另一种特殊类型的类成员函数——转换运算符。这些运算符定义了类类型对象的隐式转换机制。编译器应用这种转换机制的场合与原因都与内置类型转换是一样的。

第III部分的最后两章将介绍C++如何支持面向对象编程和泛型编程。

第 15 章介绍继承和动态绑定。继承和动态绑定与数据抽象一起构成了面向对象编程的基础。继承令关联类型的定义更为简单，而动态绑定可以帮助我们编写类型无关的代码，可以忽略具有继承关系的类型之间的差异。

第 16 章介绍函数模板和类模板。模板可以让我们写出类型无关的通用类和函数。新标准引入了一些模板相关的新特性：可变参数模板、模板类型别名以及控制实例化的新方法。

编写我们自己的面向对象的或是泛型的类型需要对 C++有深刻的理解。幸运的是，我们无须掌握如何构建面向对象和泛型类型的细节也可以使用它们。例如，标准库中广泛使用了我们将在第 15 章和第 16 章中学习的技术，虽然我们已经使用过标准库类型和算法，但实际上我们并不了解它们是如何实现的。

因此，读者应该明白第III部分涉及的是相当深入的内容。编写模板或面向对象的类要求对 C++的基本知识和基本类的定义有着深刻的理解。

第 13 章
拷贝控制

内容

13.1 拷贝、赋值与销毁 440
13.2 拷贝控制和资源管理 452
13.3 交换操作 457
13.4 拷贝控制示例 460
13.5 动态内存管理类 464
13.6 对象移动 470
小结 486
术语表 486

如我们在第 7 章所见，每个类都定义了一个新类型和在此类型对象上可执行的操作。在本章中，我们还将学到，类可以定义构造函数，用来控制在创建此类型对象时做什么。

在本章中，我们还将学习类如何控制该类型对象拷贝、赋值、移动或销毁时做什么。类通过一些特殊的成员函数控制这些操作，包括：拷贝构造函数、移动构造函数、拷贝赋值运算符、移动赋值运算符以及析构函数。

496 当定义一个类时，我们显式地或隐式地指定在此类型的对象拷贝、移动、赋值和销毁时做什么。一个类通过定义五种特殊的成员函数来控制这些操作，包括：**拷贝构造函数**（copy constructor）、**拷贝赋值运算符**（copy-assignment operator）、**移动构造函数**（move constructor）、**移动赋值运算符**（move-assignment operator）和**析构函数**（destructor）。拷贝和移动构造函数定义了当用同类型的另一个对象初始化本对象时做什么。拷贝和移动赋值运算符定义了将一个对象赋予同类型的另一个对象时做什么。析构函数定义了当此类型对象销毁时做什么。我们称这些操作为**拷贝控制操作**（copy control）。

如果一个类没有定义所有这些拷贝控制成员，编译器会自动为它定义缺失的操作。因此，很多类会忽略这些拷贝控制操作（参见 7.1.5 节，第 239 页）。但是，对一些类来说，依赖这些操作的默认定义会导致灾难。通常，实现拷贝控制操作最困难的地方是首先认识到什么时候需要定义这些操作。



在定义任何 C++类时，拷贝控制操作都是必要部分。对初学 C++的程序员来说，必须定义对象拷贝、移动、赋值或销毁时做什么，这常常令他们感到困惑。这种困扰很复杂，因为如果我们不显式定义这些操作，编译器也会为我们定义，但编译器定义的版本的行为可能并非我们所想。

13.1 拷贝、赋值与销毁

我们将以最基本的操作——拷贝构造函数、拷贝赋值运算符和析构函数作为开始。我们在 13.6 节（第 470 页）中将介绍移动操作（新标准所引入的操作）。

13.1.1 拷贝构造函数

如果一个构造函数的第一个参数是自身类类型的引用，且任何额外参数都有默认值，则此构造函数是拷贝构造函数。

```
class Foo {
public:
    Foo();              // 默认构造函数
    Foo(const Foo&);    // 拷贝构造函数
    // ...
};
```

拷贝构造函数的第一个参数必须是一个引用类型，原因我们稍后解释。虽然我们可以定义一个接受非 const 引用的拷贝构造函数，但此参数几乎总是一个 const 的引用。拷贝构造函数在几种情况下都会被隐式地使用。因此，拷贝构造函数通常不应该是 explicit 的（参见 7.5.4 节，第 265 页）。

497 **合成拷贝构造函数**

如果我们没有为一个类定义拷贝构造函数，编译器会为我们定义一个。与合成默认构造函数（参见 7.1.4 节，第 235 页）不同，即使我们定义了其他构造函数，编译器也会为我们合成一个拷贝构造函数。

如我们将在 13.1.6 节（第 450 页）中所见，对某些类来说，**合成拷贝构造函数**（synthesized copy constructor）用来阻止我们拷贝该类类型的对象。而一般情况，合成的拷贝构造函数会将其参数的成员逐个拷贝到正在创建的对象中（参见 7.1.5 节，第 239 页）。编译器从给

定对象中依次将每个非 static 成员拷贝到正在创建的对象中。

每个成员的类型决定了它如何拷贝：对类类型的成员，会使用其拷贝构造函数来拷贝；内置类型的成员则直接拷贝。虽然我们不能直接拷贝一个数组（参见 3.5.1 节，第 102 页），但合成拷贝构造函数会逐元素地拷贝一个数组类型的成员。如果数组元素是类类型，则使用元素的拷贝构造函数来进行拷贝。

作为一个例子，我们的 Sales_data 类的合成拷贝构造函数等价于：

```
class Sales_data {
public:
    // 其他成员和构造函数的定义，如前
    // 与合成的拷贝构造函数等价的拷贝构造函数的声明
   Sales_data(const Sales_data&);
private:
    std::string bookNo;
    int units_sold = 0;
    double revenue = 0.0;
};
// 与 Sales_data 的合成的拷贝构造函数等价
Sales_data::Sales_data(const Sales_data &orig):
    bookNo(orig.bookNo),          // 使用 string 的拷贝构造函数
    units_sold(orig.units_sold), // 拷贝 orig.units_sold
    revenue(orig.revenue)         // 拷贝 orig.revenue
    {    }                        // 空函数体
```

拷贝初始化

现在，我们可以完全理解直接初始化和拷贝初始化之间的差异了（参见 3.2.1 节，第 76 页）：

```
string dots(10, '.');                  // 直接初始化
string s(dots);                        // 直接初始化
string s2 = dots;                      // 拷贝初始化
string null_book = "9-999-99999-9";    // 拷贝初始化
string nines = string(100, '9');       // 拷贝初始化
```

当使用直接初始化时，我们实际上是要求编译器使用普通的函数匹配（参见 6.4 节，第 209 页）来选择与我们提供的参数最匹配的构造函数。当我们使用**拷贝初始化**（copy initialization）时，我们要求编译器将右侧运算对象拷贝到正在创建的对象中，如果需要的话还要进行类型转换（参见 7.5.4 节，第 263 页）。

拷贝初始化通常使用拷贝构造函数来完成。但是，如我们将在 13.6.2 节（第 473 页）所见，如果一个类有一个移动构造函数，则拷贝初始化有时会使用移动构造函数而非拷贝构造函数来完成。但现在，我们只需了解拷贝初始化何时发生，以及拷贝初始化是依靠拷贝构造函数或移动构造函数来完成的就可以了。

498

拷贝初始化不仅在我们用=定义变量时会发生，在下列情况下也会发生

- 将一个对象作为实参传递给一个非引用类型的形参
- 从一个返回类型为非引用类型的函数返回一个对象
- 用花括号列表初始化一个数组中的元素或一个聚合类中的成员（参见 7.5.5 节，第 266 页）

某些类类型还会对它们所分配的对象使用拷贝初始化。例如，当我们初始化标准库容器或是调用其 insert 或 push 成员（参见 9.3.1 节，第 306 页）时，容器会对其元素进行拷贝初始化。与之相对，用 emplace 成员创建的元素都进行直接初始化（参见 9.3.1 节，第 308 页）。

参数和返回值

在函数调用过程中，具有非引用类型的参数要进行拷贝初始化（参见 6.2.1 节，第 188 页）。类似的，当一个函数具有非引用的返回类型时，返回值会被用来初始化调用方的结果（参见 6.3.2 节，第 201 页）。

拷贝构造函数被用来初始化非引用类类型参数，这一特性解释了为什么拷贝构造函数自己的参数必须是引用类型。如果其参数不是引用类型，则调用永远也不会成功——为了调用拷贝构造函数，我们必须拷贝它的实参，但为了拷贝实参，我们又需要调用拷贝构造函数，如此无限循环。

拷贝初始化的限制

如前所述，如果我们使用的初始化值要求通过一个 explicit 的构造函数来进行类型转换（参见 7.5.4 节，第 265 页），那么使用拷贝初始化还是直接初始化就不是无关紧要的了：

```
vector<int> v1(10);  // 正确：直接初始化
vector<int> v2 = 10; // 错误：接受大小参数的构造函数是 explicit 的
void f(vector<int>); // f 的参数进行拷贝初始化
f(10); // 错误：不能用一个 explicit 的构造函数拷贝一个实参
f(vector<int>(10));  // 正确：从一个 int 直接构造一个临时 vector
```

直接初始化 v1 是合法的，但看起来与之等价的拷贝初始化 v2 则是错误的，因为 vector 的接受单一大小参数的构造函数是 explicit 的。出于同样的原因，当传递一个实参或从函数返回一个值时，我们不能隐式使用一个 explicit 构造函数。如果我们希望使用一个 explicit 构造函数，就必须显式地使用，像此代码中最后一行那样。

499 ### 编译器可以绕过拷贝构造函数

在拷贝初始化过程中，编译器可以（但不是必须）跳过拷贝/移动构造函数，直接创建对象。即，编译器被允许将下面的代码

```
string null_book = "9-999-99999-9"; // 拷贝初始化
```

改写为

```
string null_book("9-999-99999-9"); // 编译器略过了拷贝构造函数
```

但是，即使编译器略过了拷贝/移动构造函数，但在这个程序点上，拷贝/移动构造函数必须是存在且可访问的（例如，不能是 private 的）。

13.1.1 节练习

练习 13.1：拷贝构造函数是什么？什么时候使用它？

练习 13.2：解释为什么下面的声明是非法的：

```
Sales_data::Sales_data(Sales_data rhs);
```

练习 13.3：当我们拷贝一个 StrBlob 时，会发生什么？拷贝一个 StrBlobPtr 呢？

练习 13.4：假定 Point 是一个类类型，它有一个 public 的拷贝构造函数，指出下面程序片段中哪些地方使用了拷贝构造函数：

```
Point global;
Point foo_bar(Point arg)
{
    Point local = arg, *heap = new Point(global);
    *heap = local;
    Point pa[ 4 ] = { local, *heap };
    return *heap;
}
```

练习 13.5：给定下面的类框架，编写一个拷贝构造函数，拷贝所有成员。你的构造函数应该动态分配一个新的 string（参见 12.1.2 节，第 407 页），并将对象拷贝到 ps 指向的位置，而不是 ps 本身的位置。

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
private:
    std::string *ps;
    int i;
};
```

13.1.2 拷贝赋值运算符



与类控制其对象如何初始化一样，类也可以控制其对象如何赋值：

500

```
Sales_data trans, accum;
trans = accum; // 使用 Sales_data 的拷贝赋值运算符
```

与拷贝构造函数一样，如果类未定义自己的拷贝赋值运算符，编译器会为它合成一个。

重载赋值运算符

在介绍合成赋值运算符之前，我们需要了解一点儿有关**重载运算符**（overloaded operator）的知识，详细内容将在第 14 章中进行介绍。

重载运算符本质上是函数，其名字由 operator 关键字后接表示要定义的运算符的符号组成。因此，赋值运算符就是一个名为 operator=的函数。类似于任何其他函数，运算符函数也有一个返回类型和一个参数列表。

重载运算符的参数表示运算符的运算对象。某些运算符，包括赋值运算符，必须定义为成员函数。如果一个运算符是一个成员函数，其左侧运算对象就绑定到隐式的 this 参数（参见 7.1.2 节，第 231 页）。对于一个二元运算符，例如赋值运算符，其右侧运算对象作为显式参数传递。

拷贝赋值运算符接受一个与其所在类相同类型的参数：

```
class Foo {
public:
```

```
    Foo& operator=(const Foo&); // 赋值运算符
    // ...
};
```

为了与内置类型的赋值（参见 4.4 节，第 129 页）保持一致，赋值运算符通常返回一个指向其左侧运算对象的引用。另外值得注意的是，标准库通常要求保存在容器中的类型要具有赋值运算符，且其返回值是左侧运算对象的引用。

Best Practices 赋值运算符通常应该返回一个指向其左侧运算对象的引用。

合成拷贝赋值运算符

与处理拷贝构造函数一样，如果一个类未定义自己的拷贝赋值运算符，编译器会为它生成一个**合成拷贝赋值运算符**（synthesized copy-assignment operator）。类似拷贝构造函数，对于某些类，合成拷贝赋值运算符用来禁止该类型对象的赋值（参见 13.1.6 节，第 450 页）。如果拷贝赋值运算符并非出于此目的，它会将右侧运算对象的每个非 `static` 成员赋予左侧运算对象的对应成员，这一工作是通过成员类型的拷贝赋值运算符来完成的。对于数组类型的成员，逐个赋值数组元素。合成拷贝赋值运算符返回一个指向其左侧运算对象的引用。

501 作为一个例子，下面的代码等价于 `Sales_data` 的合成拷贝赋值运算符：

```
// 等价于合成拷贝赋值运算符
Sales_data&
Sales_data::operator=(const Sales_data &rhs)
{
    bookNo = rhs.bookNo;          // 调用 string::operator=
    units_sold = rhs.units_sold; // 使用内置的 int 赋值
    revenue = rhs.revenue;        // 使用内置的 double 赋值
    return *this;                 // 返回一个此对象的引用
}
```

13.1.2 节练习

练习 13.6：拷贝赋值运算符是什么？什么时候使用它？合成拷贝赋值运算符完成什么工作？什么时候会生成合成拷贝赋值运算符？

练习 13.7：当我们将一个 `StrBlob` 赋值给另一个 `StrBlob` 时，会发生什么？赋值 `StrBlobPtr` 呢？

练习 13.8：为 13.1.1 节（第 443 页）练习 13.5 中的 `HasPtr` 类编写赋值运算符。类似拷贝构造函数，你的赋值运算符应该将对象拷贝到 `ps` 指向的位置。



13.1.3 析构函数

析构函数执行与构造函数相反的操作：构造函数初始化对象的非 `static` 数据成员，还可能做一些其他工作；析构函数释放对象使用的资源，并销毁对象的非 `static` 数据成员。

析构函数是类的一个成员函数，名字由波浪号接类名构成。它没有返回值，也不接受参数：

```
class Foo {
public:
    ~Foo(); // 析构函数
```

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

由于析构函数不接受参数，因此它不能被重载。对一个给定类，只会有唯一一个析构函数。

析构函数完成什么工作

如同构造函数有一个初始化部分和一个函数体（参见 7.5.1 节，第 257 页），析构函数也有一个函数体和一个析构部分。在一个构造函数中，成员的初始化是在函数体执行之前完成的，且按照它们在类中出现的顺序进行初始化。在一个析构函数中，首先执行函数体，然后销毁成员。成员按初始化顺序的逆序销毁。 502

在对象最后一次使用之后，析构函数的函数体可执行类设计者希望执行的任何收尾工作。通常，析构函数释放对象在生存期分配的所有资源。

在一个析构函数中，不存在类似构造函数中初始化列表的东西来控制成员如何销毁，析构部分是隐式的。成员销毁时发生什么完全依赖于成员的类型。销毁类类型的成员需要执行成员自己的析构函数。内置类型没有析构函数，因此销毁内置类型成员什么也不需要做。



隐式销毁一个内置指针类型的成员不会 delete 它所指向的对象。

与普通指针不同，智能指针（参见 12.1.1 节，第 402 页）是类类型，所以具有析构函数。因此，与普通指针不同，智能指针成员在析构阶段会被自动销毁。

什么时候会调用析构函数

无论何时一个对象被销毁，就会自动调用其析构函数：

- 变量在离开其作用域时被销毁。
- 当一个对象被销毁时，其成员被销毁。
- 容器（无论是标准库容器还是数组）被销毁时，其元素被销毁。
- 对于动态分配的对象，当对指向它的指针应用 delete 运算符时被销毁（参见 12.1.2 节，第 409 页）。
- 对于临时对象，当创建它的完整表达式结束时被销毁。

由于析构函数自动运行，我们的程序可以按需要分配资源，而（通常）无须担心何时释放这些资源。

例如，下面代码片段定义了四个 Sales_data 对象：

```
{ // 新作用域
    // p 和 p2 指向动态分配的对象
    Sales_data *p = new Sales_data;        // p 是一个内置指针
    auto p2 = make_shared<Sales_data>(); // p2 是一个 shared_ptr
    Sales_data item(*p);       // 拷贝构造函数将*p 拷贝到 item 中
    vector<Sales_data> vec;    // 局部对象
    vec.push_back(*p2);        // 拷贝 p2 指向的对象
    delete p;                  // 对 p 指向的对象执行析构函数
} // 退出局部作用域；对 item、p2 和 vec 调用析构函数
  // 销毁 p2 会递减其引用计数；如果引用计数变为 0，对象被释放
  // 销毁 vec 会销毁它的元素
```

503 每个 Sales_data 对象都包含一个 string 成员，它分配动态内存来保存 bookNo 成员中的字符。但是，我们的代码唯一需要直接管理的内存就是我们直接分配的 Sales_data 对象。我们的代码只需直接释放绑定到 p 的动态分配对象。

其他 Sales_data 对象会在离开作用域时被自动销毁。当程序块结束时，vec、p2 和 item 都离开了作用域，意味着在这些对象上分别会执行 vector、shared_ptr 和 Sales_data 的析构函数。vector 的析构函数会销毁我们添加到 vec 的元素。shared_ptr 的析构函数会递减 p2 指向的对象的引用计数。在本例中，引用计数会变为 0，因此 shared_ptr 的析构函数会 delete p2 分配的 Sales_data 对象。

在所有情况下，Sales_data 的析构函数都会隐式地销毁 bookNo 成员。销毁 bookNo 会调用 string 的析构函数，它会释放用来保存 ISBN 的内存。



当指向一个对象的引用或指针离开作用域时，析构函数不会执行。

合成析构函数

当一个类未定义自己的析构函数时，编译器会为它定义一个**合成析构函数**（synthesized destructor）。类似拷贝构造函数和拷贝赋值运算符，对于某些类，合成析构函数被用来阻止该类型的对象被销毁（参见 13.1.6 节，第 450 页）。如果不是这种情况，合成析构函数的函数体就为空。

例如，下面的代码片段等价于 Sales_data 的合成析构函数：

```
class Sales_data {
public:
    // 成员会被自动销毁，除此之外不需要做其他事情
     ~Sales_data() { }
    // 其他成员的定义，如前
};
```

在（空）析构函数体执行完毕后，成员会被自动销毁。特别的，string 的析构函数会被调用，它将释放 bookNo 成员所用的内存。

认识到析构函数体自身并不直接销毁成员是非常重要的。成员是在析构函数体之后隐含的析构阶段中被销毁的。在整个对象销毁过程中，析构函数体是作为成员销毁步骤之外的另一部分而进行的。

13.1.3 节练习

练习 13.9：析构函数是什么？合成析构函数完成什么工作？什么时候会生成合成析构函数？

练习 13.10：当一个 StrBlob 对象销毁时会发生什么？一个 StrBlobPtr 对象销毁时呢？

练习 13.11：为前面练习中的 HasPtr 类添加一个析构函数。

练习 13.12：在下面的代码片段中会发生几次析构函数调用？

```
bool fcn(const Sales_data *trans, Sales_data accum)
{
    Sales_data item1(*trans), item2(accum);
```

```
        return item1.isbn() != item2.isbn();
    }
```

练习 13.13：理解拷贝控制成员和构造函数的一个好方法是定义一个简单的类，为该类定义这些成员，每个成员都打印出自己的名字：

```
struct X {
    X() {std::cout << "X()" << std::endl;}
    X(const X&) {std::cout << "X(const X&)" << std::endl;}
};
```

给 X 添加拷贝赋值运算符和析构函数，并编写一个程序以不同方式使用 X 的对象：将它们作为非引用和引用参数传递；动态分配它们；将它们存放于容器中；诸如此类。观察程序的输出，直到你确认理解了什么时候会使用拷贝控制成员，以及为什么会使用它们。当你观察程序输出时，记住编译器可以略过对拷贝构造函数的调用。

13.1.4 三/五法则



如前所述，有三个基本操作可以控制类的拷贝操作：拷贝构造函数、拷贝赋值运算符和析构函数。而且，在新标准下，一个类还可以定义一个移动构造函数和一个移动赋值运算符，我们将在 13.6 节（第 470 页）中介绍这些内容。

C++语言并不要求我们定义所有这些操作：可以只定义其中一个或两个，而不必定义所有。但是，这些操作通常应该被看作一个整体。通常，只需要其中一个操作，而不需要定义所有操作的情况是很少见的。 504

需要析构函数的类也需要拷贝和赋值操作

当我们决定一个类是否要定义它自己版本的拷贝控制成员时，一个基本原则是首先确定这个类是否需要一个析构函数。通常，对析构函数的需求要比对拷贝构造函数或赋值运算符的需求更为明显。如果这个类需要一个析构函数，我们几乎可以肯定它也需要一个拷贝构造函数和一个拷贝赋值运算符。

我们在练习中用过的 HasPtr 类是一个好例子（参见 13.1.1 节，第 443 页）。这个类在构造函数中分配动态内存。合成析构函数不会 delete 一个指针数据成员。因此，此类需要定义一个析构函数来释放构造函数分配的内存。

应该怎么做可能还有点儿不清晰，但基本原则告诉我们，HasPtr 也需要一个拷贝构造函数和一个拷贝赋值运算符。

如果我们为 HasPtr 定义一个析构函数，但使用合成版本的拷贝构造函数和拷贝赋值运算符，考虑会发生什么： 505

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
    ~HasPtr() { delete ps; }
    // 错误：HasPtr 需要一个拷贝构造函数和一个拷贝赋值运算符
    // 其他成员的定义，如前
};
```

在这个版本的类定义中，构造函数中分配的内存将在 HasPtr 对象销毁时被释放。但不幸的是，我们引入了一个严重的错误！这个版本的类使用了合成的拷贝构造函数和拷贝

赋值运算符。这些函数简单拷贝指针成员，这意味着多个 HasPtr 对象可能指向相同的内存：

```
HasPtr f(HasPtr hp)        // HasPtr 是传值参数，所以将被拷贝
{
    HasPtr ret = hp;       // 拷贝给定的 HasPtr
    // 处理 ret
    return ret;            // ret 和 hp 被销毁
}
```

当 f 返回时，hp 和 ret 都被销毁，在两个对象上都会调用 HasPtr 的析构函数。此析构函数会 delete ret 和 hp 中的指针成员。但这两个对象包含相同的指针值。此代码会导致此指针被 delete 两次，这显然是一个错误（参见 12.1.2 节，第 411 页）。将要发生什么是未定义的。

此外，f 的调用者还会使用传递给 f 的对象：

```
HasPtr p("some values");
f(p);              // 当 f 结束时，p.ps 指向的内存被释放
HasPtr q(p);       // 现在 p 和 q 都指向无效内存!
```

p（以及 q）指向的内存不再有效，在 hp（或 ret!）销毁时它就被归还给系统了。



如果一个类需要自定义析构函数，几乎可以肯定它也需要自定义拷贝赋值运算符和拷贝构造函数。

需要拷贝操作的类也需要赋值操作，反之亦然

虽然很多类需要定义所有（或是不需要定义任何）拷贝控制成员，但某些类所要完成的工作，只需要拷贝或赋值操作，不需要析构函数。

作为一个例子，考虑一个类为每个对象分配一个独有的、唯一的序号。这个类需要一个拷贝构造函数为每个新创建的对象生成一个新的、独一无二的序号。除此之外，这个拷贝构造函数从给定对象拷贝所有其他数据成员。这个类还需要自定义拷贝赋值运算符来避免将序号赋予目的对象。但是，这个类不需要自定义析构函数。

506

这个例子引出了第二个基本原则：如果一个类需要一个拷贝构造函数，几乎可以肯定它也需要一个拷贝赋值运算符。反之亦然——如果一个类需要一个拷贝赋值运算符，几乎可以肯定它也需要一个拷贝构造函数。然而，无论是需要拷贝构造函数还是需要拷贝赋值运算符都不必然意味着也需要析构函数。

13.1.4 节练习

练习 13.14：假定 numbered 是一个类，它有一个默认构造函数，能为每个对象生成一个唯一的序号，保存在名为 mysn 的数据成员中。假定 numbered 使用合成的拷贝控制成员，并给定如下函数：

```
void f (numbered s) { cout << s.mysn << endl; }
```

则下面代码输出什么内容？

```
numbered a, b = a, c = b;
f(a); f(b); f(c);
```

练习 13.15：假定 numbered 定义了一个拷贝构造函数，能生成一个新的序号。这会改变上一题中调用的输出结果吗？如果会改变，为什么？新的输出结果是什么？

练习 13.16：如果 f 中的参数是 const numbered&，将会怎样？这会改变输出结果吗？如果会改变，为什么？新的输出结果是什么？

练习 13.17：分别编写前三题中所描述的 numbered 和 f，验证你是否正确预测了输出结果。

13.1.5 使用=default

我们可以通过将拷贝控制成员定义为=default来显式地要求编译器生成合成的版本（参见 7.1.4 节，第 237 页）：

C++ 11

```
class Sales_data {
public:
    // 拷贝控制成员；使用 default
    Sales_data() = default;
    Sales_data(const Sales_data&) = default;
    Sales_data& operator=(const Sales_data &);
    ~Sales_data() = default;
    // 其他成员的定义，如前
};
Sales_data& Sales_data::operator=(const Sales_data&) = default;
```

当我们在类内用=default 修饰成员的声明时，合成的函数将隐式地声明为内联的（就像任何其他类内声明的成员函数一样）。如果我们不希望合成的成员是内联函数，应该只对成员的类外定义使用=default，就像对拷贝赋值运算符所做的那样。

507



我们只能对具有合成版本的成员函数使用=default（即，默认构造函数或拷贝控制成员）。

13.1.6 阻止拷贝



大多数类应该定义默认构造函数、拷贝构造函数和拷贝赋值运算符，无论是隐式地还是显式地。

虽然大多数类应该定义（而且通常也的确定义了）拷贝构造函数和拷贝赋值运算符，但对某些类来说，这些操作没有合理的意义。在此情况下，定义类时必须采用某种机制阻止拷贝或赋值。例如，iostream 类阻止了拷贝，以避免多个对象写入或读取相同的 IO 缓冲。为了阻止拷贝，看起来可能应该不定义拷贝控制成员。但是，这种策略是无效的：如果我们的类未定义这些操作，编译器为它生成合成的版本。

定义删除的函数

在新标准下，我们可以通过将拷贝构造函数和拷贝赋值运算符定义为**删除的函数**（deleted function）来阻止拷贝。删除的函数是这样一种函数：我们虽然声明了它们，但不能以任何方式使用它们。在函数的参数列表后面加上=delete 来指出我们希望将它定义为删除的：

C++ 11

```
struct NoCopy {
    NoCopy() = default;       // 使用合成的默认构造函数
    NoCopy(const NoCopy&) = delete;                // 阻止拷贝
    NoCopy &operator=(const NoCopy&) = delete;     // 阻止赋值
```

```
    ~NoCopy() = default;       // 使用合成的析构函数
    // 其他成员
};
```

=delete 通知编译器（以及我们代码的读者），我们不希望定义这些成员。

与=default 不同，=delete 必须出现在函数第一次声明的时候，这个差异与这些声明的含义在逻辑上是吻合的。一个默认的成员只影响为这个成员而生成的代码，因此=default 直到编译器生成代码时才需要。而另一方面，编译器需要知道一个函数是删除的，以便禁止试图使用它的操作。

与=default 的另一个不同之处是，我们可以对任何函数指定=delete（我们只能对
508
编译器可以合成的默认构造函数或拷贝控制成员使用=default）。虽然删除函数的主要用途是禁止拷贝控制成员，但当我们希望引导函数匹配过程时，删除函数有时也是有用的。

析构函数不能是删除的成员

值得注意的是，我们不能删除析构函数。如果析构函数被删除，就无法销毁此类型的对象了。对于一个删除了析构函数的类型，编译器将不允许定义该类型的变量或创建该类的临时对象。而且，如果一个类有某个成员的类型删除了析构函数，我们也不能定义该类的变量或临时对象。因为如果一个成员的析构函数是删除的，则该成员无法被销毁。而如果一个成员无法被销毁，则对象整体也就无法被销毁了。

对于删除了析构函数的类型，虽然我们不能定义这种类型的变量或成员，但可以动态分配这种类型的对象。但是，不能释放这些对象：

```
struct NoDtor {
    NoDtor() = default;  // 使用合成默认构造函数
    ~NoDtor() = delete;  // 我们不能销毁 NoDtor 类型的对象
};
NoDtor nd; // 错误：NoDtor 的析构函数是删除的
NoDtor *p = new NoDtor();// 正确：但我们不能 delete p
delete p; // 错误：NoDtor 的析构函数是删除的
```



对于析构函数已删除的类型，不能定义该类型的变量或释放指向该类型动态分配对象的指针。

合成的拷贝控制成员可能是删除的

如前所述，如果我们未定义拷贝控制成员，编译器会为我们定义合成的版本。类似的，如果一个类未定义构造函数，编译器会为其合成一个默认构造函数（参见 7.1.4 节，第 235 页）。对某些类来说，编译器将这些合成的成员定义为删除的函数：

- 如果类的某个成员的析构函数是删除的或不可访问的（例如，是 private 的），则类的合成析构函数被定义为删除的。
- 如果类的某个成员的拷贝构造函数是删除的或不可访问的，则类的合成拷贝构造函数被定义为删除的。如果类的某个成员的析构函数是删除的或不可访问的，则类合成的拷贝构造函数也被定义为删除的。
- 如果类的某个成员的拷贝赋值运算符是删除的或不可访问的，或是类有一个 const 的或引用成员，则类的合成拷贝赋值运算符被定义为删除的。
- 如果类的某个成员的析构函数是删除的或不可访问的，或是类有一个引用成员，它没有类内初始化器（参见 2.6.1 节，第 65 页），或是类有一个 const 成员，它没有

类内初始化器且其类型未显式定义默认构造函数，则该类的默认构造函数被定义为删除的。

本质上，这些规则的含义是：如果一个类有数据成员不能默认构造、拷贝、复制或销毁，则对应的成员函数将被定义为删除的。 509

一个成员有删除的或不可访问的析构函数会导致合成的默认和拷贝构造函数被定义为删除的，这看起来可能有些奇怪。其原因是，如果没有这条规则，我们可能会创建出无法销毁的对象。

对于具有引用成员或无法默认构造的 const 成员的类，编译器不会为其合成默认构造函数，这应该不奇怪。同样不出人意料的规则是：如果一个类有 const 成员，则它不能使用合成的拷贝赋值运算符。毕竟，此运算符试图赋值所有成员，而将一个新值赋予一个 const 对象是不可能的。

虽然我们可以将一个新值赋予一个引用成员，但这样做改变的是引用指向的对象的值，而不是引用本身。如果为这样的类合成拷贝赋值运算符，则赋值后，左侧运算对象仍然指向与赋值前一样的对象，而不会与右侧运算对象指向相同的对象。由于这种行为看起来并不是我们所期望的，因此对于有引用成员的类，合成拷贝赋值运算符被定义为删除的。

我们将在 13.6.2 节（第 476 页）、15.7.2 节（第 553 页）及 19.6 节（第 751 页）中介绍导致类的拷贝控制成员被定义为删除函数的其他原因。



本质上，当不可能拷贝、赋值或销毁类的成员时，类的合成拷贝控制成员就被定义为删除的。

private 拷贝控制

在新标准发布之前，类是通过将其拷贝构造函数和拷贝赋值运算符声明为 private 的来阻止拷贝：

```
class PrivateCopy {
    // 无访问说明符；接下来的成员默认为 private 的；参见 7.2 节（第 240 页）
    // 拷贝控制成员是 private 的，因此普通用户代码无法访问
    PrivateCopy(const PrivateCopy&);
    PrivateCopy &operator=(const PrivateCopy&);
    // 其他成员
public:
    PrivateCopy() = default; // 使用合成的默认构造函数
    ~PrivateCopy(); // 用户可以定义此类型的对象，但无法拷贝它们
};
```

由于析构函数是 public 的，用户可以定义 PrivateCopy 类型的对象。但是，由于拷贝构造函数和拷贝赋值运算符是 private 的，用户代码将不能拷贝这个类型的对象。但是，友元和成员函数仍旧可以拷贝对象。为了阻止友元和成员函数进行拷贝，我们将这些拷贝控制成员声明为 private 的，但并不定义它们。

声明但不定义一个成员函数是合法的（参见 6.1.2 节，第 186 页），对此只有一个例外，我们将在 15.2.1 节（第 528 页）中介绍。试图访问一个未定义的成员将导致一个链接时错误。通过声明（但不定义）private 的拷贝构造函数，我们可以预先阻止任何拷贝该类型对象的企图：试图拷贝对象的用户代码将在编译阶段被标记为错误；成员函数或友元函数中的拷贝操作将会导致链接时错误。 510

Best Practices 希望阻止拷贝的类应该使用=delete 来定义它们自己的拷贝构造函数和拷贝赋值运算符，而不应该将它们声明为 private 的。

13.1.6 节练习

练习 13.18：定义一个 Employee 类，它包含雇员的姓名和唯一的雇员证号。为这个类定义默认构造函数，以及接受一个表示雇员姓名的 string 的构造函数。每个构造函数应该通过递增一个 static 数据成员来生成一个唯一的证号。

练习 13.19：你的 Employee 类需要定义它自己的拷贝控制成员吗？如果需要，为什么？如果不需要，为什么？实现你认为 Employee 需要的拷贝控制成员。

练习 13.20：解释当我们拷贝、赋值或销毁 TextQuery 和 QueryResult 类（参见 12.3 节，第 430 页）对象时会发生什么。

练习 13.21：你认为 TextQuery 和 QueryResult 类需要定义它们自己版本的拷贝控制成员吗？如果需要，为什么？如果不需要，为什么？实现你认为这两个类需要的拷贝控制操作。

13.2 拷贝控制和资源管理

通常，管理类外资源的类必须定义拷贝控制成员。如我们在 13.1.4 节（第 447 页）中所见，这种类需要通过析构函数来释放对象所分配的资源。一旦一个类需要析构函数，那么它几乎肯定也需要一个拷贝构造函数和一个拷贝赋值运算符。

为了定义这些成员，我们首先必须确定此类型对象的拷贝语义。一般来说，有两种选择：可以定义拷贝操作，使类的行为看起来像一个值或者像一个指针。

类的行为像一个值，意味着它应该也有自己的状态。当我们拷贝一个像值的对象时，副本和原对象是完全独立的。改变副本不会对原对象有任何影响，反之亦然。

行为像指针的类则共享状态。当我们拷贝一个这种类的对象时，副本和原对象使用相同的底层数据。改变副本也会改变原对象，反之亦然。

在我们使用过的标准库类中，标准库容器和 string 类的行为像一个值。而不出意外的，shared_ptr 类提供类似指针的行为，就像我们的 StrBlob 类（参见 12.1.1 节，第 405 页）一样，IO 类型和 unique_ptr 不允许拷贝或赋值，因此它们的行为既不像值也不像指针。

511

为了说明这两种方式，我们会为练习中的 HasPtr 类定义拷贝控制成员。首先，我们将令类的行为像一个值；然后重新实现类，使它的行为像一个指针。

我们的 HasPtr 类有两个成员，一个 int 和一个 string 指针。通常，类直接拷贝内置类型（不包括指针）成员；这些成员本身就是值，因此通常应该让它们的行为像值一样。我们如何拷贝指针成员决定了像 HasPtr 这样的类是具有类值行为还是类指针行为。

13.2 节练习

练习 13.22：假定我们希望 HasPtr 的行为像一个值。即，对于对象所指向的 string

成员，每个对象都有一份自己的拷贝。我们将在下一节介绍拷贝控制成员的定义。但是，你已经学习了定义这些成员所需的所有知识。在继续学习下一节之前，为 HasPtr 编写拷贝构造函数和拷贝赋值运算符。

13.2.1 行为像值的类



为了提供类值的行为，对于类管理的资源，每个对象都应该拥有一份自己的拷贝。这意味着对于 ps 指向的 string，每个 HasPtr 对象都必须有自己的拷贝。为了实现类值行为，HasPtr 需要

- 定义一个拷贝构造函数，完成 string 的拷贝，而不是拷贝指针
- 定义一个析构函数来释放 string
- 定义一个拷贝赋值运算符来释放对象当前的 string，并从右侧运算对象拷贝 string

类值版本的 HasPtr 如下所示

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
    // 对 ps 指向的 string，每个 HasPtr 对象都有自己的拷贝
    HasPtr(const HasPtr &p):
        ps(new std::string(*p.ps)), i(p.i) { }
    HasPtr& operator=(const HasPtr &);
    ~HasPtr() { delete ps; }
private:
    std::string *ps;
    int      i;
};
```

我们的类足够简单，在类内就已定义了除赋值运算符之外的所有成员函数。第一个构造函数接受一个（可选的）string 参数。这个构造函数动态分配它自己的 string 副本，并将指向 string 的指针保存在 ps 中。拷贝构造函数也分配它自己的 string 副本。析构函数对指针成员 ps 执行 delete，释放构造函数中分配的内存。

512

类值拷贝赋值运算符

赋值运算符通常组合了析构函数和构造函数的操作。类似析构函数，赋值操作会销毁左侧运算对象的资源。类似拷贝构造函数，赋值操作会从右侧运算对象拷贝数据。但是，非常重要的一点是，这些操作是以正确的顺序执行的，即使将一个对象赋予它自身，也保证正确。而且，如果可能，我们编写的赋值运算符还应该是异常安全的——当异常发生时能将左侧运算对象置于一个有意义的状态（参见 5.6.2 节，第 175 页）。

在本例中，通过先拷贝右侧运算对象，我们可以处理自赋值情况，并能保证在异常发生时代码也是安全的。在完成拷贝后，我们释放左侧运算对象的资源，并更新指针指向新分配的 string：

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    auto newp = new string(*rhs.ps); // 拷贝底层 string
    delete ps;        // 释放旧内存
```

```
    ps = newp;          // 从右侧运算对象拷贝数据到本对象
    i = rhs.i;
    return *this;       // 返回本对象
}
```

在这个赋值运算符中，非常清楚，我们首先进行了构造函数的工作：newp 的初始化器等价于 HasPtr 的拷贝构造函数中 ps 的初始化器。接下来与析构函数一样，我们 delete 当前 ps 指向的 string。然后就只剩下拷贝指向新分配的 string 的指针，以及从 rhs 拷贝 int 值到本对象了。

关键概念：赋值运算符

当你编写赋值运算符时，有两点需要记住：

- 如果将一个对象赋予它自身，赋值运算符必须能正确工作。
- 大多数赋值运算符组合了析构函数和拷贝构造函数的工作。

当你编写一个赋值运算符时，一个好的模式是先将右侧运算对象拷贝到一个局部临时对象中。当拷贝完成后，销毁左侧运算对象的现有成员就是安全的了。一旦左侧运算对象的资源被销毁，就只剩下将数据从临时对象拷贝到左侧运算对象的成员中了。

513 为了说明防范自赋值操作的重要性，考虑如果赋值运算符如下编写将会发生什么

```
// 这样编写赋值运算符是错误的!
HasPtr&
HasPtr::operator=(const HasPtr &rhs)
{
    delete ps; // 释放对象指向的 string
    // 如果 rhs 和*this 是同一个对象，我们就将从已释放的内存中拷贝数据!
    ps = new string(*(rhs.ps));
    i = rhs.i;
    return *this;
}
```

如果 rhs 和本对象是同一个对象，delete ps 会释放*this 和 rhs 指向的 string。接下来，当我们在 new 表达式中试图拷贝*(rhs.ps)时，就会访问一个指向无效内存的指针，其行为和结果是未定义的。



对于一个赋值运算符来说，正确工作是非常重要的，即使是将一个对象赋予它自身，也要能正确工作。一个好的方法是在销毁左侧运算对象资源之前拷贝右侧运算对象。

13.2.1 节练习

练习 13.23：比较上一节练习中你编写的拷贝控制成员和这一节中的代码。确定你理解了你的代码和我们的代码之间的差异（如果有的话）。

练习 13.24：如果本节中的 HasPtr 版本未定义析构函数，将会发生什么？如果未定义拷贝构造函数，将会发生什么？

练习 13.25：假定希望定义 StrBlob 的类值版本，而且希望继续使用 shared_ptr，

这样我们的 StrBlobPtr 类就仍能使用指向 vector 的 weak_ptr 了。你修改后的类将需要一个拷贝构造函数和一个拷贝赋值运算符，但不需要析构函数。解释拷贝构造函数和拷贝赋值运算符必须要做什么。解释为什么不需要析构函数。

练习 13.26：对上一题中描述的 StrBlob 类，编写你自己的版本。

13.2.2 定义行为像指针的类



对于行为类似指针的类，我们需要为其定义拷贝构造函数和拷贝赋值运算符，来拷贝指针成员本身而不是它指向的 string。我们的类仍然需要自己的析构函数来释放接受 string 参数的构造函数分配的内存（参见 13.1.4 节，第 447 页）。但是，在本例中，析构函数不能单方面地释放关联的 string。只有当最后一个指向 string 的 HasPtr 销毁时，它才可以释放 string。

令一个类展现类似指针的行为的最好方法是使用 shared_ptr 来管理类中的资源。拷贝（或赋值）一个 shared_ptr 会拷贝（赋值）shared_ptr 所指向的指针。514 shared_ptr 类自己记录有多少用户共享它所指向的对象。当没有用户使用对象时，shared_ptr 类负责释放资源。

但是，有时我们希望直接管理资源。在这种情况下，使用**引用计数**（reference count）（参见 12.1.1 节，第 402 页）就很有用了。为了说明引用计数如何工作，我们将重新定义 HasPtr，令其行为像指针一样，但我们不使用 shared_ptr，而是设计自己的引用计数。

引用计数

引用计数的工作方式如下：

- 除了初始化对象外，每个构造函数（拷贝构造函数除外）还要创建一个引用计数，用来记录有多少对象与正在创建的对象共享状态。当我们创建一个对象时，只有一个对象共享状态，因此将计数器初始化为 1。
- 拷贝构造函数不分配新的计数器，而是拷贝给定对象的数据成员，包括计数器。拷贝构造函数递增共享的计数器，指出给定对象的状态又被一个新用户所共享。
- 析构函数递减计数器，指出共享状态的用户少了一个。如果计数器变为 0，则析构函数释放状态。
- 拷贝赋值运算符递增右侧运算对象的计数器，递减左侧运算对象的计数器。如果左侧运算对象的计数器变为 0，意味着它的共享状态没有用户了，拷贝赋值运算符就必须销毁状态。

唯一的难题是确定在哪里存放引用计数。计数器不能直接作为 HasPtr 对象的成员。下面的例子说明了原因：

```
HasPtr p1("Hiya!");
HasPtr p2(p1);    // p1 和 p2 指向相同的 string
HasPtr p3(p1);    // p1、p2 和 p3 都指向相同的 string
```

如果引用计数保存在每个对象中，当创建 p3 时我们应该如何正确更新它呢？可以递增 p1 中的计数器并将其拷贝到 p3 中，但如何更新 p2 中的计数器呢？

解决此问题的一种方法是将计数器保存在动态内存中。当创建一个对象时，我们也分配一个新的计数器。当拷贝或赋值对象时，我们拷贝指向计数器的指针。使用这种方法，副本和原对象都会指向相同的计数器。

定义一个使用引用计数的类

通过使用引用计数，我们就可以编写类指针的 HasPtr 版本了：

515
```
class HasPtr {
public:
    // 构造函数分配新的 string 和新的计数器，将计数器置为 1
    HasPtr(const std::string &s = std::string()):
      ps(new std::string(s)), i(0), use(new std::size_t(1)) {}
    // 拷贝构造函数拷贝所有三个数据成员，并递增计数器
    HasPtr(const HasPtr &p):
        ps(p.ps), i(p.i), use(p.use) { ++*use; }
    HasPtr& operator=(const HasPtr&);
    ~HasPtr();
private:
    std::string *ps;
    int i;
    std::size_t *use; // 用来记录有多少个对象共享*ps 的成员
};
```

在此，我们添加了一个名为 use 的数据成员，它记录有多少对象共享相同的 string。接受 string 参数的构造函数分配新的计数器，并将其初始化为 1，指出当前有一个用户使用本对象的 string 成员。

类指针的拷贝成员“篡改”引用计数

当拷贝或赋值一个 HasPtr 对象时，我们希望副本和原对象都指向相同的 string。即，当拷贝一个 HasPtr 时，我们将拷贝 ps 本身，而不是 ps 指向的 string。当我们进行拷贝时，还会递增该 string 关联的计数器。

（我们在类内定义的）拷贝构造函数拷贝给定 HasPtr 的所有三个数据成员。这个构造函数还递增 use 成员，指出 ps 和 p.ps 指向的 string 又有了一个新的用户。

析构函数不能无条件地 delete ps——可能还有其他对象指向这块内存。析构函数应该递减引用计数，指出共享 string 的对象少了一个。如果计数器变为 0，则析构函数释放 ps 和 use 指向的内存：

```
HasPtr::~HasPtr()
{
    if (--*use == 0) {    // 如果引用计数变为 0
        delete ps;        // 释放 string 内存
        delete use;       // 释放计数器内存
    }
}
```

拷贝赋值运算符与往常一样执行类似拷贝构造函数和析构函数的工作。即，它必须递增右侧运算对象的引用计数（即，拷贝构造函数的工作），并递减左侧运算对象的引用计数，在必要时释放使用的内存（即，析构函数的工作）。

而且与往常一样，赋值运算符必须处理自赋值。我们通过先递增 rhs 中的计数然后
516 再递减左侧运算对象中的计数来实现这一点。通过这种方法，当两个对象相同时，在我们检查 ps（及 use）是否应该释放之前，计数器就已经被递增过了：

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
```

```
    ++*rhs.use; // 递增右侧运算对象的引用计数
    if (--*use == 0) {    // 然后递减本对象的引用计数
        delete ps;        // 如果没有其他用户
        delete use;       // 释放本对象分配的成员
    }
    ps = rhs.ps;          // 将数据从 rhs 拷贝到本对象
    i = rhs.i;
    use = rhs.use;
    return *this;         // 返回本对象
}
```

13.2.2 节练习

练习 13.27：定义你自己的使用引用计数版本的 HasPtr。

练习 13.28：给定下面的类，为其实现一个默认构造函数和必要的拷贝控制成员。

```
(a) class TreeNode {
    private:
        std::string value;
        int         count;
        TreeNode    *left;
        TreeNode    *right;
    };
```

```
(b) class BinStrTree {
    private:
        TreeNode *root;
    };
```

13.3 交换操作

除了定义拷贝控制成员，管理资源的类通常还定义一个名为 swap 的函数（参见 9.2.5 节，第 303 页）。对于那些与重排元素顺序的算法（参见 10.2.3 节，第 342 页）一起使用的类，定义 swap 是非常重要的。这类算法在需要交换两个元素时会调用 swap。

如果一个类定义了自己的 swap，那么算法将使用类自定义版本。否则，算法将使用标准库定义的 swap。虽然与往常一样我们不知道 swap 是如何实现的，但理论上很容易理解，为了交换两个对象我们需要进行一次拷贝和两次赋值。例如，交换两个类值 HasPtr 对象（参见 13.2.1 节，第 453 页）的代码可能像下面这样：

```
HasPtr temp = v1;     // 创建 v1 的值的一个临时副本
v1 = v2;              // 将 v2 的值赋予 v1
v2 = temp;            // 将保存的 v1 的值赋予 v2
```

这段代码将原来 v1 中的 string 拷贝了两次——第一次是 HasPtr 的拷贝构造函数将 v1 拷贝给 temp，第二次是赋值运算符将 temp 赋予 v2。将 v2 赋予 v1 的语句还拷贝了原来 v2 中的 string。如我们所见，拷贝一个类值的 HasPtr 会分配一个新 string 并将其拷贝到 HasPtr 指向的位置。

517

理论上，这些内存分配都是不必要的。我们更希望 swap 交换指针，而不是分配 string 的新副本。即，我们希望这样交换两个 HasPtr：

```
string *temp = v1.ps;     // 为 v1.ps 中的指针创建一个副本
v1.ps = v2.ps;            // 将 v2.ps 中的指针赋予 v1.ps
v2.ps = temp;             // 将保存的 v1.ps 中原来的指针赋予 v2.ps
```

编写我们自己的 swap 函数

可以在我们的类上定义一个自己版本的 swap 来重载 swap 的默认行为。swap 的典型实现如下：

```
class HasPtr {
    friend void swap(HasPtr&, HasPtr&);
    // 其他成员定义，与 13.2.1 节（第 453 页）中一样
};
inline
void swap(HasPtr &lhs, HasPtr &rhs)
{
    using std::swap;
    swap(lhs.ps, rhs.ps);      // 交换指针，而不是 string 数据
    swap(lhs.i, rhs.i);        // 交换 int 成员
}
```

我们首先将 swap 定义为 friend，以便能访问 HasPtr 的（private 的）数据成员。由于 swap 的存在就是为了优化代码，我们将其声明为 inline 函数（参见 6.5.2 节，第 213 页）。swap 的函数体对给定对象的每个数据成员调用 swap。我们首先 swap 绑定到 rhs 和 lhs 的对象的指针成员，然后是 int 成员。



与拷贝控制成员不同，swap 并不是必要的。但是，对于分配了资源的类，定义 swap 可能是一种很重要的优化手段。

swap 函数应该调用 swap，而不是 std::swap

此代码中有一个很重要的微妙之处：虽然这一点在这个特殊的例子中并不重要，但在一般情况下它非常重要——swap 函数中调用的 swap 不是 std::swap。在本例中，数据成员是内置类型的，而内置类型是没有特定版本的 swap 的，所以在本例中，对 swap 的调用会调用标准库 std::swap。

但是，如果一个类的成员有自己类型特定的 swap 函数，调用 std::swap 就是错误的了。例如，假定我们有另一个命名为 Foo 的类，它有一个类型为 HasPtr 的成员 h。如
518 果我们未定义 Foo 版本的 swap，那么就会使用标准库版本的 swap。如我们所见，标准库 swap 对 HasPtr 管理的 string 进行了不必要的拷贝。

我们可以为 Foo 编写一个 swap 函数，来避免这些拷贝。但是，如果这样编写 Foo 版本的 swap：

```
void swap(Foo &lhs, Foo &rhs)
{
    // 错误：这个函数使用了标准库版本的 swap，而不是 HasPtr 版本
    std::swap(lhs.h, rhs.h);
    // 交换类型 Foo 的其他成员
}
```

此编码会编译通过，且正常运行。但是，使用此版本与简单使用默认版本的 swap 并没有任何性能差异。问题在于我们显式地调用了标准库版本的 swap。但是，我们不希望使用 std 中的版本，我们希望调用为 HasPtr 对象定义的版本。

正确的 swap 函数如下所示：

```
void swap(Foo &lhs, Foo &rhs)
```

```
{
    using std::swap;
    swap(lhs.h, rhs.h); // 使用 HasPtr 版本的 swap
    // 交换类型 Foo 的其他成员
}
```

每个 swap 调用应该都是未加限定的。即，每个调用都应该是 swap，而不是 std::swap。如果存在类型特定的 swap 版本，其匹配程度会优于 std 中定义的版本，原因我们将在16.3 节（第 616 页）中进行解释。因此，如果存在类型特定的 swap 版本，swap 调用会与之匹配。如果不存在类型特定的版本，则会使用 std 中的版本（假定作用域中有 using 声明）。

非常仔细的读者可能会奇怪为什么 swap 函数中的 using 声明没有隐藏 HasPtr 版本 swap 的声明（参见 6.4.1 节，第 210 页）。我们将在 18.2.3 节（第 706 页）中解释为什么这段代码能正常工作。

在赋值运算符中使用 swap

定义 swap 的类通常用 swap 来定义它们的赋值运算符。这些运算符使用了一种名为**拷贝并交换**（copy and swap）的技术。这种技术将左侧运算对象与右侧运算对象的一个副本进行交换：

```
// 注意 rhs 是按值传递的，意味着 HasPtr 的拷贝构造函数
// 将右侧运算对象中的 string 拷贝到 rhs
HasPtr& HasPtr::operator=(HasPtr rhs)
{
    // 交换左侧运算对象和局部变量 rhs 的内容
    swap(*this, rhs);      // rhs 现在指向本对象曾经使用的内存
    return *this;          // rhs 被销毁，从而 delete 了 rhs 中的指针
}
```

在这个版本的赋值运算符中，参数并不是一个引用，我们将右侧运算对象以传值方式传递给了赋值运算符。因此，rhs 是右侧运算对象的一个副本。参数传递时拷贝 HasPtr 的操作会分配该对象的 string 的一个新副本。 519

在赋值运算符的函数体中，我们调用 swap 来交换 rhs 和*this 中的数据成员。这个调用将左侧运算对象中原来保存的指针存入 rhs 中，并将 rhs 中原来的指针存入*this 中。因此，在 swap 调用之后，*this 中的指针成员将指向新分配的 string——右侧运算对象中 string 的一个副本。

当赋值运算符结束时，rhs 被销毁，HasPtr 的析构函数将执行。此析构函数 delete rhs 现在指向的内存，即，释放掉左侧运算对象中原来的内存。

这个技术的有趣之处是它自动处理了自赋值情况且天然就是异常安全的。它通过在改变左侧运算对象之前拷贝右侧运算对象保证了自赋值的正确，这与我们在原来的赋值运算符中使用的方法是一致的（参见 13.2.1 节，第 453 页）。它保证异常安全的方法也与原来的赋值运算符实现一样。代码中唯一可能抛出异常的是拷贝构造函数中的 new 表达式。如果真发生了异常，它也会在我们改变左侧运算对象之前发生。



使用拷贝和交换的赋值运算符自动就是异常安全的，且能正确处理自赋值。

13.3 节练习

练习 13.29：解释 swap(HasPtr&, HasPtr&) 中对 swap 的调用不会导致递归循环。

练习 13.30：为你的类值版本的 HasPtr 编写 swap 函数，并测试它。为你的 swap 函数添加一个打印语句，指出函数什么时候执行。

练习 13.31：为你的 HasPtr 类定义一个<运算符，并定义一个 HasPtr 的 vector。为这个 vector 添加一些元素，并对它执行 sort。注意何时会调用 swap。

练习 13.32：类指针的 HasPtr 版本会从 swap 函数受益吗？如果会，得到了什么益处？如果不是，为什么？

13.4 拷贝控制示例

虽然通常来说分配资源的类更需要拷贝控制，但资源管理并不是一个类需要定义自己的拷贝控制成员的唯一原因。一些类也需要拷贝控制成员的帮助来进行簿记工作或其他操作。

作为类需要拷贝控制来进行簿记操作的例子，我们将概述两个类的设计，这两个类可能用于邮件处理应用中。两个类命名为 Message 和 Folder，分别表示电子邮件（或者
520 其他类型的）消息和消息目录。每个 Message 对象可以出现在多个 Folder 中。但是，任意给定的 Message 的内容只有一个副本。这样，如果一条 Message 的内容被改变，则我们从它所在的任何 Folder 来浏览此 Message 时，都会看到改变后的内容。

为了记录 Message 位于哪些 Folder 中，每个 Message 都会保存一个它所在 Folder 的指针的 set，同样的，每个 Folder 都保存一个它包含的 Message 的指针的 set。图 13.1 说明了这种设计思路。

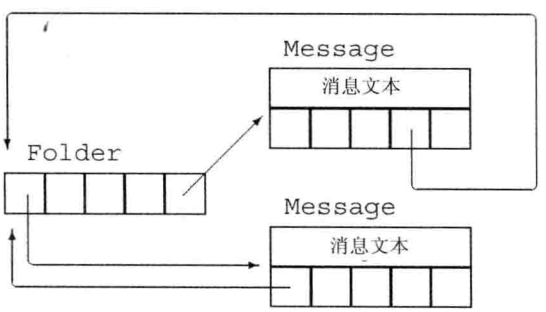


图 13.1：Message 和 Folder 类设计

我们的 Message 类会提供 save 和 remove 操作，来向一个给定 Folder 添加一条 Message 或是从中删除一条 Message。为了创建一个新的 Message，我们会指明消息内容，但不会指出 Folder。为了将一条 Message 放到一个特定 Folder 中，我们必须调用 save。

当我们拷贝一个 Message 时，副本和原对象将是不同的 Message 对象，但两个 Message 都出现在相同的 Folder 中。因此，拷贝 Message 的操作包括消息内容和 Folder 指针 set 的拷贝。而且，我们必须在每个包含此消息的 Folder 中都添加一个指向新创建的 Message 的指针。

当我们销毁一个 Message 时，它将不复存在。因此，我们必须从包含此消息的所有

Folder 中删除指向此 Message 的指针。

当我们将一个 Message 对象赋予另一个 Message 对象时，左侧 Message 的内容会被右侧 Message 的内容所替代。我们还必须更新 Folder 集合，从原来包含左侧 Message 的 Folder 中将它删除，并将它添加到包含右侧 Message 的 Folder 中。

观察这些操作，我们可以看到，析构函数和拷贝赋值运算符都必须从包含一条 Message 的所有 Folder 中删除它。类似的，拷贝构造函数和拷贝赋值运算符都要将一个 Message 添加到给定的一组 Folder 中。我们将定义两个 private 的工具函数来完成这些工作。



拷贝赋值运算符通常执行拷贝构造函数和析构函数中也要做的工作。这种情况下，公共的工作应该放在 private 的工具函数中完成。

Folder 类也需要类似的拷贝控制成员，来添加或删除它保存的 Message。 521

我们将 Folder 类的设计和实现留作练习。但是，我们将假定 Folder 类包含名为 addMsg 和 remMsg 的成员，分别完成在给定 Folder 对象的消息集合中添加和删除 Message 的工作。

Message 类

根据上述设计，我们可以编写 Message 类，如下所示：

```
class Message {
    friend class Folder;
public:
    // folders 被隐式初始化为空集合
    explicit Message(const std::string &str = ""):
        contents(str) { }
    // 拷贝控制成员，用来管理指向本 Message 的指针
    Message(const Message&);                    // 拷贝构造函数
    Message& operator=(const Message&);         // 拷贝赋值运算符
    ~Message();                                 // 析构函数
    // 从给定 Folder 集合中添加/删除本 Message
    void save(Folder&);
    void remove(Folder&);
private:
    std::string contents;           // 实际消息文本
    std::set<Folder*> folders;      // 包含本 Message 的 Folder
    // 拷贝构造函数、拷贝赋值运算符和析构函数所使用的工具函数
    // 将本 Message 添加到指向参数的 Folder 中
    void add_to_Folders(const Message&);
    // 从 folders 中的每个 Folder 中删除本 Message
    void remove_from_Folders();
};
```

这个类定义了两个数据成员：contents，保存消息文本；folders，保存指向本 Message 所在 Folder 的指针。接受一个 string 参数的构造函数将给定 string 拷贝给 contents，并将 folders（隐式）初始化为空集。由于此构造函数有一个默认参数，因此它也被当作 Message 的默认构造函数（参见 7.5.1 节，第 260 页）。

save 和 remove 成员

除拷贝控制成员外，Message 类只有两个公共成员：save，将本 Message 存放在给定 Folder 中；remove，删除本 Message：

```
void Message::save(Folder &f)
{
    folders.insert(&f);   // 将给定 Folder 添加到我们的 Folder 列表中
    f.addMsg(this);       // 将本 Message 添加到 f 的 Message 集合中
}
```

522

```
void Message::remove(Folder &f)
{
    folders.erase(&f);    // 将给定 Folder 从我们的 Folder 列表中删除
    f.remMsg(this);       // 将本 Message 从 f 的 Message 集合中删除
}
```

为了保存（或删除）一个 Message，需要更新本 Message 的 folders 成员。当 save 一个 Message 时，我们应保存一个指向给定 Folder 的指针；当 remove 一个 Message 时，我们要删除此指针。

这些操作还必须更新给定的 Folder。更新一个 Folder 的任务是由 Folder 类的 addMsg 和 remMsg 成员来完成的，分别添加和删除给定 Message 的指针。

Message 类的拷贝控制成员

当我们拷贝一个 Message 时，得到的副本应该与原 Message 出现在相同的 Folder 中。因此，我们必须遍历 Folder 指针的 set，对每个指向原 Message 的 Folder 添加一个指向新 Message 的指针。拷贝构造函数和拷贝赋值运算符都需要做这个工作，因此我们定义一个函数来完成这个公共操作：

```
// 将本 Message 添加到指向 m 的 Folder 中
void Message::add_to_Folders(const Message &m)
{
    for (auto f : m.folders) // 对每个包含 m 的 Folder
        f->addMsg(this);     // 向该 Folder 添加一个指向本 Message 的指针
}
```

此例中我们对 m.folders 中每个 Folder 调用 addMsg。函数 addMsg 会将本 Message 的指针添加到每个 Folder 中。

Message 的拷贝构造函数拷贝给定对象的数据成员：

```
Message::Message(const Message &m):
    contents(m.contents), folders(m.folders)
{
    add_to_Folders(m); // 将本消息添加到指向 m 的 Folder 中
}
```

并调用 add_to_Folders 将新创建的 Message 的指针添加到每个包含原 Message 的 Folder 中。

Message 的析构函数

当一个 Message 被销毁时，我们必须从指向此 Message 的 Folder 中删除它。拷贝赋值运算符也要执行此操作，因此我们会定义一个公共函数来完成此工作：

```
// 从对应的 Folder 中删除本 Message
void Message::remove_from_Folders()
{
    for (auto f : folders)    // 对 folders 中每个指针
        f->remMsg(this);      // 从该 Folder 中删除本 Message
}
```

函数 remove_from_Folders 的实现类似 add_to_Folders，不同之处是它调用 remMsg 来删除当前 Message 而不是调用 addMsg 来添加 Message。 523

有了 remove_from_Folders 函数，编写析构函数就很简单了：

```
Message::~Message()
{
    remove_from_Folders();
}
```

调用 remove_from_Folders 确保没有任何 Folder 保存正在销毁的 Message 的指针。编译器自动调用 string 的析构函数来释放 contents，并自动调用 set 的析构函数来清理集合成员使用的内存。

Message 的拷贝赋值运算符

与大多数赋值运算符相同，我们的 Message 类的拷贝赋值运算符必须执行拷贝构造函数和析构函数的工作。与往常一样，最重要的是我们要组织好代码结构，使得即使左侧和右侧运算对象是同一个 Message，拷贝赋值运算符也能正确执行。

在本例中，我们先从左侧运算对象的 folders 中删除此 Message 的指针，然后再将指针添加到右侧运算对象的 folders 中，从而实现了自赋值的正确处理：

```
Message& Message::operator=(const Message &rhs)
{
    // 通过先删除指针再插入它们来处理自赋值情况
    remove_from_Folders();    // 更新已有 Folder
    contents = rhs.contents; // 从 rhs 拷贝消息内容
    folders = rhs.folders;    // 从 rhs 拷贝 Folder 指针
    add_to_Folders(rhs);      // 将本 Message 添加到那些 Folder 中
    return *this;
}
```

如果左侧和右侧运算对象是相同的 Message，则它们具有相同的地址。如果我们在 add_to_Folders 之后调用 remove_from_Folders，就会将此 Message 从它所在的所有 Folder 中删除。

Message 的 swap 函数

标准库中定义了 string 和 set 的 swap 版本（参见 9.2.5 节，第 303 页）。因此，如果为我们的 Message 类定义它自己的 swap 版本，它将从中受益。通过定义一个 Message 特定版本的 swap，我们可以避免对 contents 和 folders 成员进行不必要的拷贝。

但是，我们的 swap 函数必须管理指向被交换 Message 的 Folder 指针。在调用 swap(m1,m2) 之后，原来指向 m1 的 Folder 现在必须指向 m2，反之亦然。

我们通过两遍扫描 folders 中每个成员来正确处理 Folder 指针。第一遍扫描将 Message 从它们所在的 Folder 中删除。接下来我们调用 swap 来交换数据成员。最后

对 folders 进行第二遍扫描来添加交换过的 Message：

524
```
void swap(Message &lhs, Message &rhs)
{
    using std::swap; // 在本例中严格来说并不需要，但这是一个好习惯
    // 将每个消息的指针从它（原来）所在 Folder 中删除
    for (auto f: lhs.folders)
        f->remMsg(&lhs);
    for (auto f: rhs.folders)
        f->remMsg(&rhs);
    // 交换 contents 和 Folder 指针 set
    swap(lhs.folders, rhs.folders);          // 使用 swap(set&, set&)
    swap(lhs.contents, rhs.contents);        // swap(string&, string&)
    // 将每个 Message 的指针添加到它的（新）Folder 中
    for (auto f: lhs.folders)
        f->addMsg(&lhs);
    for (auto f: rhs.folders)
        f->addMsg(&rhs);
}
```

13.4 节练习

练习 13.33：为什么 Message 的成员 save 和 remove 的参数是一个 Folder&？为什么我们不将参数定义为 Folder 或是 const Folder&？

练习 13.34：编写本节所描述的 Message。

练习 13.35：如果 Message 使用合成的拷贝控制成员，将会发生什么？

练习 13.36：设计并实现对应的 Folder 类。此类应该保存一个指向 Folder 中包含的 Message 的 set。

练习 13.37：为 Message 类添加成员，实现向 folders 添加或删除一个给定的 Folder*。这两个成员类似 Folder 类的 addMsg 和 remMsg 操作。

练习 13.38：我们并未使用拷贝和交换方式来设计 Message 的赋值运算符。你认为其原因是什么？

13.5 动态内存管理类

某些类需要在运行时分配可变大小的内存空间。这种类通常可以（并且如果它们确实可以的话，一般应该）使用标准库容器来保存它们的数据。例如，我们的 StrBlob 类使用一个 vector 来管理其元素的底层内存。

但是，这一策略并不是对每个类都适用；某些类需要自己进行内存分配。这些类一般来说必须定义自己的拷贝控制成员来管理所分配的内存。

525 例如，我们将实现标准库 vector 类的一个简化版本。我们所做的一个简化是不使用模板，我们的类只用于 string。因此，它被命名为 StrVec。

StrVec 类的设计

回忆一下，vector 类将其元素保存在连续内存中。为了获得可接受的性能，vector

预先分配足够的内存来保存可能需要的更多元素（参见 9.4 节，第 317 页）。vector 的每个添加元素的成员函数会检查是否有空间容纳更多的元素。如果有，成员函数会在下一个可用位置构造一个对象。如果没有可用空间，vector 就会重新分配空间：它获得新的空间，将已有元素移动到新空间中，释放旧空间，并添加新元素。

我们在 StrVec 类中使用类似的策略。我们将使用一个 allocator 来获得原始内存（参见 12.2.2 节，第 427 页）。由于 allocator 分配的内存是未构造的，我们将在需要添加新元素时用 allocator 的 construct 成员在原始内存中创建对象。类似的，当我们需要删除一个元素时，我们将使用 destroy 成员来销毁元素。

每个 StrVec 有三个指针成员指向其元素所使用的内存：

- elements，指向分配的内存中的首元素
- first_free，指向最后一个实际元素之后的位置
- cap，指向分配的内存末尾之后的位置

图 13.2 说明了这些指针的含义。

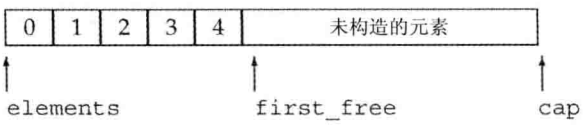


图 13.2：StrVec 内存分配策略

除了这些指针之外，StrVec 还有一个名为 alloc 的静态成员，其类型为 allocator<string>。alloc 成员会分配 StrVec 使用的内存。我们的类还有 4 个工具函数：

- alloc_n_copy 会分配内存，并拷贝一个给定范围中的元素。
- free 会销毁构造的元素并释放内存。
- chk_n_alloc 保证 StrVec 至少有容纳一个新元素的空间。如果没有空间添加新元素，chk_n_alloc 会调用 reallocate 来分配更多内存。
- reallocate 在内存用完时为 StrVec 分配新内存。

虽然我们关注的是类的实现，但我们也将定义 vector 接口中的一些成员。

StrVec 类定义

526

有了上述实现概要，我们现在可以定义 StrVec 类，如下所示：

```
// 类 vector 类内存分配策略的简化实现
class StrVec {
public:
    StrVec(): // allocator 成员进行默认初始化
      elements(nullptr), first_free(nullptr), cap(nullptr) { }
    StrVec(const StrVec&);                  // 拷贝构造函数
    StrVec &operator=(const StrVec&);       // 拷贝赋值运算符
    ~StrVec();                              // 析构函数
    void push_back(const std::string&);     // 拷贝元素
    size_t size() const { return first_free - elements; }
    size_t capacity() const { return cap - elements; }
    std::string *begin() const { return elements; }
    std::string *end() const { return first_free; }
```

```
    // ...
private:
    Static std::allocator<std::string> alloc; // 分配元素
    // 被添加元素的函数所使用
    void chk_n_alloc()
        { if (size() == capacity()) reallocate(); }
    // 工具函数，被拷贝构造函数、赋值运算符和析构函数所使用
    std::pair<std::string*, std::string*> alloc_n_copy
        (const std::string*, const std::string*);
    void free();                // 销毁元素并释放内存
    void reallocate();          // 获得更多内存并拷贝已有元素
    std::string *elements;      // 指向数组首元素的指针
    std::string *first_free;    // 指向数组第一个空闲元素的指针
    std::string *cap;           // 指向数组尾后位置的指针
};
```

类体定义了多个成员：

- 默认构造函数(隐式地)默认初始化 alloc 并(显式地)将指针初始化为 nullptr，表明没有元素。
- size 成员返回当前真正在使用的元素的数目，等于 first_free-elements。
- capacity 成员返回 StrVec 可以保存的元素的数量，等价于 cap-elements。
- 当没有空间容纳新元素，即 cap==first_free 时，chk_n_alloc 会为 StrVec 重新分配内存。
- begin 和 end 成员分别返回指向首元素（即 elements）和最后一个构造的元素之后位置（即 first_free）的指针。

使用 construct

函数 push_back 调用 chk_n_alloc 确保有空间容纳新元素。如果需要，
527 chk_n_alloc 会调用 reallocate。当 chk_n_alloc 返回时，push_back 知道必有空间容纳新元素。它要求其 allocator 成员来 construct 新的尾元素：

```
void StrVec::push_back(const string& s)
{
    chk_n_alloc(); // 确保有空间容纳新元素
    // 在 first_free 指向的元素中构造 s 的副本
    alloc.construct(first_free++, s);
}
```

当我们用 allocator 分配内存时，必须记住内存是未构造的(参见 12.2.2 节，第 428 页)。为了使用此原始内存，我们必须调用 construct，在此内存中构造一个对象。传递给 construct 的第一个参数必须是一个指针，指向调用 allocate 所分配的未构造的内存空间。剩余参数确定用哪个构造函数来构造对象。在本例中，只有一个额外参数，类型为 string，因此会使用 string 的拷贝构造函数。

值得注意的是，对 construct 的调用也会递增 first_free，表示已经构造了一个新元素。它使用前置递增（参见 4.5 节，第 131 页），因此这个调用会在 first_free 当前值指定的地址构造一个对象，并递增 first_free 指向下一个未构造的元素。

alloc_n_copy 成员

我们在拷贝或赋值 StrVec 时，可能会调用 alloc_n_copy 成员。类似 vector，我们的 StrVec 类有类值的行为（参见 13.2.1 节，第 453 页）。当我们拷贝或赋值 StrVec 时，必须分配独立的内存，并从原 StrVec 对象拷贝元素至新对象。

alloc_n_copy 成员会分配足够的内存来保存给定范围的元素，并将这些元素拷贝到新分配的内存中。此函数返回一个指针的 pair（参见 11.2.3 节，第 379 页），两个指针分别指向新空间的开始位置和拷贝的尾后的位置：

```
pair<string*, string*>
StrVec::alloc_n_copy(const string *b, const string *e)
{
    // 分配空间保存给定范围中的元素
    auto data = alloc.allocate(e - b);
    // 初始化并返回一个 pair，该 pair 由 data 和 uninitialized_copy 的返回值构成
    return {data, uninitialized_copy(b, e, data)};
}
```

alloc_n_copy 用尾后指针减去首元素指针，来计算需要多少空间。在分配内存之后，它必须在此空间中构造给定元素的副本。

它是在返回语句中完成拷贝工作的，返回语句中对返回值进行了列表初始化（参见 6.3.2 节，第 203 页）。返回的 pair 的 first 成员指向分配的内存的开始位置；second 成员则是 uninitialized_copy（参见 12.2.2 节，第 429 页）的返回值，此值是一个指针，指向最后一个构造元素之后的位置。 528

free 成员

free 成员有两个责任：首先 destroy 元素，然后释放 StrVec 自己分配的内存空间。for 循环调用 allocator 的 destroy 成员，从构造的尾元素开始，到首元素为止，逆序销毁所有元素：

```
void StrVec::free()
{
    // 不能传递给 deallocate 一个空指针，如果 elements 为 0，函数什么也不做
    if (elements) {
        // 逆序销毁旧元素
        for (auto p = first_free; p != elements; /* 空 */)
            alloc.destroy(--p);
        alloc.deallocate(elements, cap - elements);
    }
}
```

destroy 函数会运行 string 的析构函数。string 的析构函数会释放 string 自己分配的内存空间。

一旦元素被销毁，我们就调用 deallocate 来释放本 StrVec 对象分配的内存空间。我们传递给 deallocate 的指针必须是之前某次 allocate 调用所返回的指针。因此，在调用 deallocate 之前我们首先检查 elements 是否为空。

拷贝控制成员

实现了 alloc_n_copy 和 free 成员后，为我们的类实现拷贝控制成员就很简单了。

拷贝构造函数调用 alloc_n_copy：

```
StrVec::StrVec(const StrVec &s)
{
    // 调用 alloc_n_copy 分配空间以容纳与 s 中一样多的元素
    auto newdata = alloc_n_copy(s.begin(), s.end());
    elements = newdata.first;
    first_free = cap = newdata.second;
}
```

并将返回结果赋予数据成员。alloc_n_copy 的返回值是一个指针的 pair。其 first 成员指向第一个构造的元素，second 成员指向最后一个构造的元素之后的位置。由于 alloc_n_copy 分配的空间恰好容纳给定的元素，cap 也指向最后一个构造的元素之后的位置。

析构函数调用 free：

```
StrVec::~StrVec() { free(); }
```

拷贝赋值运算符在释放已有元素之前调用 alloc_n_copy，这样就可以正确处理自赋值了：

529

```
StrVec &StrVec::operator=(const StrVec &rhs)
{
    // 调用 alloc_n_copy 分配内存，大小与 rhs 中元素占用空间一样多
    auto data = alloc_n_copy(rhs.begin(), rhs.end());
    free();
    elements = data.first;
    first_free = cap = data.second;
    return *this;
}
```

类似拷贝构造函数，拷贝赋值运算符使用 alloc_n_copy 的返回值来初始化它的指针。

在重新分配内存的过程中移动而不是拷贝元素

在编写 reallocate 成员函数之前，我们稍微思考一下此函数应该做什么。它应该

- 为一个新的、更大的 string 数组分配内存
- 在内存空间的前一部分构造对象，保存现有元素
- 销毁原内存空间中的元素，并释放这块内存

观察这个操作步骤，我们可以看出，为一个 StrVec 重新分配内存空间会引起从旧内存空间到新内存空间逐个拷贝 string。虽然我们不知道 string 的实现细节，但我们知道 string 具有类值行为。当拷贝一个 string 时，新 string 和原 string 是相互独立的。改变原 string 不会影响到副本，反之亦然。

由于 string 的行为类似值，我们可以得出结论，每个 string 对构成它的所有字符都会保存自己的一份副本。拷贝一个 string 必须为这些字符分配内存空间，而销毁一个 string 必须释放所占用的内存。

拷贝一个 string 就必须真的拷贝数据，因为通常情况下，在我们拷贝了一个 string 之后，它就会有两个用户。但是，如果是 reallocate 拷贝 StrVec 中的 string，则在拷贝之后，每个 string 只有唯一的用户。一旦将元素从旧空间拷贝到了新空间，我们就会立即销毁原 string。

因此，拷贝这些 string 中的数据是多余的。在重新分配内存空间时，如果我们能避免分配和释放 string 的额外开销，StrVec 的性能会好得多。

移动构造函数和 std::move

通过使用新标准库引入的两种机制，我们就可以避免 string 的拷贝。首先，有一些标准库类，包括 string，都定义了所谓的“移动构造函数”。关于 string 的移动构造函数如何工作的细节，以及有关实现的任何其他细节，目前都尚未公开。但是，我们知道，移动构造函数通常是将资源从给定对象“移动”而不是拷贝到正在创建的对象。而且我们知道标准库保证“移后源”（moved-from）string 仍然保持一个有效的、可析构的状态。对于 string，我们可以想象每个 string 都有一个指向 char 数组的指针。可以假定 string 的移动构造函数进行了指针的拷贝，而不是为字符分配内存空间然后拷贝字符。

C++ 11

530

我们使用的第二个机制是一个名为 **move** 的标准库函数，它定义在 utility 头文件中。目前，关于 move 我们需要了解两个关键点。首先，当 reallocate 在新内存中构造 string 时，它必须调用 move 来表示希望使用 string 的移动构造函数，原因我们将在 13.6.1 节（第 470 页）中解释。如果它漏掉了 move 调用，将会使用 string 的拷贝构造函数。其次，我们通常不为 move 提供一个 using 声明（参见 3.1 节，第 74 页），原因将在 18.2.3 节（第 706 页）中解释。当我们使用 move 时，直接调用 std::move 而不是 move。

reallocate 成员

了解了这些知识，现在就可以编写 reallocate 成员了。首先调用 allocate 分配新内存空间。我们每次重新分配内存时都会将 StrVec 的容量加倍。如果 StrVec 为空，我们将分配容纳一个元素的空间：

```
void StrVec::reallocate()
{
    // 我们将分配当前大小两倍的内存空间
    auto newcapacity = size() ? 2 * size() : 1;
    // 分配新内存
    auto newdata = alloc.allocate(newcapacity);
    // 将数据从旧内存移动到新内存
    auto dest = newdata;      // 指向新数组中下一个空闲位置
    auto elem = elements;     // 指向旧数组中下一个元素
    for (size_t i = 0; i != size(); ++i)
        alloc.construct(dest++, std::move(*elem++));
    free(); // 一旦我们移动完元素就释放旧内存空间
    // 更新我们的数据结构，执行新元素
    elements = newdata;
    first_free = dest;
    cap = elements + newcapacity;
}
```

for 循环遍历每个已有元素，并在新内存空间中 construct 一个对应元素。我们使用 dest 指向构造新 string 的内存，使用 elem 指向原数组中的元素。我们每次用后置递增运算将 dest（和 elem）推进到各自数组中的下一个元素。

construct 的第二个参数（即，确定使用哪个构造函数的参数（参见 12.2.2 节，第 428 页））是 move 返回的值。调用 move 返回的结果会令 construct 使用 string 的移

动构造函数。由于我们使用了移动构造函数，这些 string 管理的内存将不会被拷贝。相反，我们构造的每个 string 都会从 elem 指向的 string 那里接管内存的所有权。

531 在元素移动完毕后，我们调用 free 销毁旧元素并释放 StrVec 原来使用的内存。string 成员不再管理它们曾经指向的内存；其数据的管理职责已经转移给新 StrVec 内存中的元素了。我们不知道旧 StrVec 内存中的 string 包含什么值，但我们保证对它们执行 string 的析构函数是安全的。

剩下的就是更新指针，指向新分配并已初始化过的数组了。first_free 和 cap 指针分别被设置为指向最后一个构造的元素之后的位置及指向新分配空间的尾后位置。

13.5 节练习

练习 13.39：编写你自己版本的 StrVec，包括自己版本的 reserve、capacity（参见 9.4 节，第 318 页）和 resize（参见 9.3.5 节，第 314 页）。

练习 13.40：为你的 StrVec 类添加一个构造函数，它接受一个 initializer_list<string>参数。

练习 13.41：在 push_back 中，我们为什么在 construct 调用中使用前置递增运算？如果使用后置递增运算的话，会发生什么？

练习 13.42：在你的 TextQuery 和 QueryResult 类（参见 12.3 节，第 431 页）中用你的 StrVec 类代替 vector<string>，以此来测试你的 StrVec 类。

练习 13.43：重写 free 成员，用 for_each 和 lambda（参见 10.3.2 节，第 346 页）来代替 for 循环 destroy 元素。你更倾向于哪种实现，为什么？

练习 13.44：编写标准库 string 类的简化版本，命名为 String。你的类应该至少有一个默认构造函数和一个接受 C 风格字符串指针参数的构造函数。使用 allocator 为你的 String 类分配所需内存。

13.6 对象移动

新标准的一个最主要的特性是可以移动而非拷贝对象的能力。如我们在 13.1.1 节（第 440 页）中所见，很多情况下都会发生对象拷贝。在其中某些情况下，对象拷贝后就立即被销毁了。在这些情况下，移动而非拷贝对象会大幅度提升性能。

如我们已经看到的，我们的 StrVec 类是这种不必要的拷贝的一个很好的例子。在重新分配内存的过程中，从旧内存将元素拷贝到新内存是不必要的，更好的方式是移动元素。使用移动而不是拷贝的另一个原因源于 IO 类或 unique_ptr 这样的类。这些类都包含不能被共享的资源（如指针或 IO 缓冲）。因此，这些类型的对象不能拷贝但可以移动。

532 在旧 C++标准中，没有直接的方法移动对象。因此，即使不必拷贝对象的情况下，我们也不得不拷贝。如果对象较大，或者是对象本身要求分配内存空间（如 string），进行不必要的拷贝代价非常高。类似的，在旧版本的标准库中，容器中所保存的类必须是可拷贝的。但在新标准中，我们可以用容器保存不可拷贝的类型，只要它们能被移动即可。



标准库容器、string 和 shared_ptr 类既支持移动也支持拷贝。IO 类和 unique_ptr 类可以移动但不能拷贝。



13.6.1 右值引用

为了支持移动操作，新标准引入了一种新的引用类型——**右值引用**（rvalue reference）。所谓右值引用就是必须绑定到右值的引用。我们通过&&而不是&来获得右值引用。如我们将要看到的，右值引用有一个重要的性质——只能绑定到一个将要销毁的对象。因此，我们可以自由地将一个右值引用的资源“移动”到另一个对象中。

C++ 11

回忆一下，左值和右值是表达式的属性（参见 4.1.1 节，第 121 页）。一些表达式生成或要求左值，而另外一些则生成或要求右值。一般而言，一个左值表达式表示的是一个对象的身份，而一个右值表达式表示的是对象的值。

类似任何引用，一个右值引用也不过是某个对象的另一个名字而已。如我们所知，对于常规引用（为了与右值引用区分开来，我们可以称之为**左值引用**（lvalue reference）），我们不能将其绑定到要求转换的表达式、字面常量或是返回右值的表达式（参见 2.3.1 节，第 46 页）。右值引用有着完全相反的绑定特性：我们可以将一个右值引用绑定到这类表达式上，但不能将一个右值引用直接绑定到一个左值上：

```
int i = 42;
int &r = i;                 // 正确：r 引用 i
int &&rr = i;               // 错误：不能将一个右值引用绑定到一个左值上
int &r2 = i * 42;           // 错误：i*42 是一个右值
const int &r3 = i * 42;     // 正确：我们可以将一个 const 的引用绑定到一个右值上
int &&rr2 = i * 42;         // 正确：将 rr2 绑定到乘法结果上
```

返回左值引用的函数，连同赋值、下标、解引用和前置递增/递减运算符，都是返回左值的表达式的例子。我们可以将一个左值引用绑定到这类表达式的结果上。

返回非引用类型的函数，连同算术、关系、位以及后置递增/递减运算符，都生成右值。我们不能将一个左值引用绑定到这类表达式上，但我们可以将一个 const 的左值引用或者一个右值引用绑定到这类表达式上。

左值持久：右值短暂

533

考察左值和右值表达式的列表，两者相互区别之处就很明显了：左值有持久的状态，而右值要么是字面常量，要么是在表达式求值过程中创建的临时对象。

由于右值引用只能绑定到临时对象，我们得知

- 所引用的对象将要被销毁
- 该对象没有其他用户

这两个特性意味着：使用右值引用的代码可以自由地接管所引用的对象的资源。



右值引用指向将要被销毁的对象。因此，我们可以从绑定到右值引用的对象“窃取”状态。

变量是左值

变量可以看作只有一个运算对象而没有运算符的表达式，虽然我们很少这样看待变

量。类似其他任何表达式，变量表达式也有左值/右值属性。变量表达式都是左值。带来的结果就是，我们不能将一个右值引用绑定到一个右值引用类型的变量上，这有些令人惊讶：

```
int &&rr1 = 42;  // 正确：字面常量是右值
int &&rr2 = rr1; // 错误：表达式 rr1 是左值!
```

其实有了右值表示临时对象这一观察结果，变量是左值这一特性并不令人惊讶。毕竟，变量是持久的，直至离开作用域时才被销毁。



变量是左值，因此我们不能将一个右值引用直接绑定到一个变量上，即使这个变量是右值引用类型也不行。

标准库 move 函数

虽然不能将一个右值引用直接绑定到一个左值上，但我们可以显式地将一个左值转换为对应的右值引用类型。我们还可以通过调用一个名为 **move** 的新标准库函数来获得绑定到左值上的右值引用，此函数定义在头文件 utility 中。move 函数使用了我们将在 16.2.6 节（第 610 页）中描述的机制来返回给定对象的右值引用。

C++ 11

```
int &&rr3 = std::move(rr1); // ok
```

move 调用告诉编译器：我们有一个左值，但我们希望像一个右值一样处理它。我们必须认识到，调用 move 就意味着承诺：除了对 rr1 赋值或销毁它外，我们将不再使用它。在调用 move 之后，我们不能对移后源对象的值做任何假设。

534



我们可以销毁一个移后源对象，也可以赋予它新值，但不能使用一个移后源对象的值。

如前所述，与大多数标准库名字的使用不同，对 move（参见 13.5 节，第 469 页）我们不提供 using 声明（参见 3.1 节，第 74 页）。我们直接调用 std::move 而不是 move，其原因将在 18.2.3 节（第 707 页）中解释。



使用 move 的代码应该使用 std::move 而不是 move。这样做可以避免潜在的名字冲突。

13.6.1 节练习

练习 13.45：解释右值引用和左值引用的区别。

练习 13.46：什么类型的引用可以绑定到下面的初始化器上？

```
int f();
vector<int> vi(100);
int? r1 = f();
int? r2 = vi[0];
int? r3 = r1;
int? r4 = vi[0] * f();
```

练习 13.47：对你在练习 13.44（13.5 节，第 470 页）中定义的 String 类，为它的拷贝构造函数和拷贝赋值运算符添加一条语句，在每次函数执行时打印一条信息。

练习 13.48：定义一个 vector<String>并在其上多次调用 push_back。运行你的程序，并观察 String 被拷贝了多少次。

13.6.2 移动构造函数和移动赋值运算符



类似 string 类（及其他标准库类），如果我们自己的类也同时支持移动和拷贝，那么也能从中受益。为了让我们自己的类型支持移动操作，需要为其定义移动构造函数和移动赋值运算符。这两个成员类似对应的拷贝操作，但它们从给定对象“窃取”资源而不是拷贝资源。

类似拷贝构造函数，移动构造函数的第一个参数是该类类型的一个引用。不同于拷贝构造函数的是，这个引用参数在移动构造函数中是一个右值引用。与拷贝构造函数一样，任何额外的参数都必须有默认实参。

C++ 11

除了完成资源移动，移动构造函数还必须确保移后源对象处于这样一个状态——销毁它是无害的。特别是，一旦资源完成移动，源对象必须不再指向被移动的资源——这些资源的所有权已经归属新创建的对象。

作为一个例子，我们为 StrVec 类定义移动构造函数，实现从一个 StrVec 到另一个 StrVec 的元素移动而非拷贝：

535

```
StrVec::StrVec(StrVec &&s) noexcept // 移动操作不应抛出任何异常
  // 成员初始化器接管 s 中的资源
  : elements(s.elements), first_free(s.first_free), cap(s.cap)
{
    // 令 s 进入这样的状态——对其运行析构函数是安全的
    s.elements = s.first_free = s.cap = nullptr;
}
```

我们将简短解释 noexcept（它通知标准库我们的构造函数不抛出任何异常），但让我们先分析一下此构造函数完成什么工作。

与拷贝构造函数不同，移动构造函数不分配任何新内存；它接管给定的 StrVec 中的内存。在接管内存之后，它将给定对象中的指针都置为 nullptr。这样就完成了从给定对象的移动操作，此对象将继续存在。最终，移后源对象会被销毁，意味着将在其上运行析构函数。StrVec 的析构函数在 first_free 上调用 deallocate。如果我们忘记了改变 s.first_free，则销毁移后源对象就会释放掉我们刚刚移动的内存。

移动操作、标准库容器和异常



由于移动操作“窃取”资源，它通常不分配任何资源。因此，移动操作通常不会抛出任何异常。当编写一个不抛出异常的移动操作时，我们应该将此事通知标准库。我们将看到，除非标准库知道我们的移动构造函数不会抛出异常，否则它会认为移动我们的类对象时可能会抛出异常，并且为了处理这种可能性而做一些额外的工作。

一种通知标准库的方法是在我们的构造函数中指明 noexcept。noexcept 是新标准引入的，我们将在 18.1.4 节（第 690 页）中讨论更多细节。目前重要的是要知道，noexcept 是我们承诺一个函数不抛出异常的一种方法。我们在一个函数的参数列表后指定 noexcept。在一个构造函数中，noexcept 出现在参数列表和初始化列表开始的冒号之间：

C++ 11

```
class StrVec {
```

```
public:
    StrVec(StrVec&&) noexcept; // 移动构造函数
    // 其他成员的定义，如前
};
StrVec::StrVec(StrVec &&s) noexcept : /* 成员初始化器 */
{ /* 构造函数体 */ }
```

我们必须在类头文件的声明中和定义中（如果定义在类外的话）都指定 noexcept。



不抛出异常的移动构造函数和移动赋值运算符必须标记为 noexcept。

536 搞清楚为什么需要 noexcept 能帮助我们深入理解标准库是如何与我们自定义的类型交互的。我们需要指出一个移动操作不抛出异常，这是因为两个相互关联的事实：首先，虽然移动操作通常不抛出异常，但抛出异常也是允许的；其次，标准库容器能对异常发生时其自身的行为提供保障。例如，vector 保证，如果我们调用 push_back 时发生异常，vector 自身不会发生改变。

现在让我们思考 push_back 内部发生了什么。类似对应的 StrVec 操作（参见 13.5 节，第 466 页），对一个 vector 调用 push_back 可能要求为 vector 重新分配内存空间。当重新分配 vector 的内存时，vector 将元素从旧空间移动到新内存中，就像我们在 reallocate 中所做的那样（参见 13.5 节，第 469 页）。

如我们刚刚看到的那样，移动一个对象通常会改变它的值。如果重新分配过程使用了移动构造函数，且在移动了部分而不是全部元素后抛出了一个异常，就会产生问题。旧空间中的移动源元素已经被改变了，而新空间中未构造的元素可能尚不存在。在此情况下，vector 将不能满足自身保持不变的要求。

另一方面，如果 vector 使用了拷贝构造函数且发生了异常，它可以很容易地满足要求。在此情况下，当在新内存中构造元素时，旧元素保持不变。如果此时发生了异常，vector 可以释放新分配的（但还未成功构造的）内存并返回。vector 原有的元素仍然存在。

为了避免这种潜在问题，除非 vector 知道元素类型的移动构造函数不会抛出异常，否则在重新分配内存的过程中，它就必须使用拷贝构造函数而不是移动构造函数。如果希望在 vector 重新分配内存这类情况下对我们自定义类型的对象进行移动而不是拷贝，就必须显式地告诉标准库我们的移动构造函数可以安全使用。我们通过将移动构造函数（及移动赋值运算符）标记为 noexcept 来做到这一点。

移动赋值运算符

移动赋值运算符执行与析构函数和移动构造函数相同的工作。与移动构造函数一样，如果我们的移动赋值运算符不抛出任何异常，我们就应该将它标记为 noexcept。类似拷贝赋值运算符，移动赋值运算符必须正确处理自赋值：

```
StrVec &StrVec::operator=(StrVec &&rhs) noexcept
{
    // 直接检测自赋值
    if (this != &rhs) {
        free();                    // 释放已有元素
        elements = rhs.elements; // 从 rhs 接管资源
        first_free = rhs.first_free;
```