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++ 11 432

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：

```
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

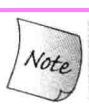
在一个不允许重复关键字的关联容器中查找一个元素是一件很简单的事情——元素要么在容器中，要么不在。但对于允许重复关键字的容器来说，过程就更为复杂：在容器中可能有很多元素具有给定的关键字。如果一个 multimap 或 multiset 中有多个元素具有给定关键字，则这些元素在容器中会相邻存储。

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。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：

438

```
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。

done!

done!

第 12 章

动态内存

内容

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：

451
```
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
```

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()	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
p.reset(q)	
p.reset(q, d)	

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
}
```