9.3.5 节练习

353

练习 9.29：假定 vec 包含 25 个元素，那么 vec.resize(100)会做什么？如果接下来调用 vec.resize(10)会做什么？

练习 9.30：接受单个参数的 resize 版本对元素类型有什么限制（如果有的话）？

9.3.6 容器操作可能使迭代器失效

向容器中添加元素和从容器中删除元素的操作可能会使指向容器元素的指针、引用或迭代器失效。一个失效的指针、引用或迭代器将不再表示任何元素。使用失效的指针、引用或迭代器是一种严重的程序设计错误，很可能引起与使用未初始化指针一样的问题（参见 2.3.2 节，第 49 页）

在向容器添加元素后：

- 如果容器是 vector 或 string，且存储空间被重新分配，则指向容器的迭代器、指针和引用都会失效。如果存储空间未重新分配，指向插入位置之前的元素的迭代器、指针和引用仍有效，但指向插入位置之后元素的迭代器、指针和引用将会失效。
- 对于 deque，插入到除首尾位置之外的任何位置都会导致迭代器、指针和引用失效。如果在首尾位置添加元素，迭代器会失效，但指向存在的元素的引用和指针不会失效。
- 对于 list 和 forward_list，指向容器的迭代器（包括尾后迭代器和首前迭代器）、指针和引用仍有效。

当我们从一个容器中删除元素后，指向被删除元素的迭代器、指针和引用会失效，这应该不会令人惊讶。毕竟，这些元素都已经被销毁了。当我们删除一个元素后：

- 对于 list 和 forward_list，指向容器其他位置的迭代器（包括尾后迭代器和首前迭代器）、引用和指针仍有效。
- 对于 deque，如果在首尾之外的任何位置删除元素，那么指向被删除元素外其他元素的迭代器、引用或指针也会失效。如果是删除 deque 的尾元素，则尾后迭代器也会失效，但其他迭代器、引用和指针不受影响；如果是删除首元素，这些也不会受影响。
- 对于 vector 和 string，指向被删元素之前元素的迭代器、引用和指针仍有效。注意：当我们删除元素时，尾后迭代器总是会失效。

使用失效的迭代器、指针或引用是严重的运行时错误。

建议：管理迭代器

354

当你使用迭代器（或指向容器元素的引用或指针）时，最小化要求迭代器必须保持有效的程序片段是一个好的方法。

由于向迭代器添加元素和从迭代器删除元素的代码可能会使迭代器失效，因此必须保证每次改变容器的操作之后都正确地重新定位迭代器。这个建议对 vector、string 和 deque 尤为重要。

编写改变容器的循环程序

添加/删除 vector、string 或 deque 元素的循环程序必须考虑迭代器、引用和指针可能失效的问题。程序必须保证每个循环步中都更新迭代器、引用或指针。如果循环中调用的是 insert 或 erase，那么更新迭代器很容易。这些操作都返回迭代器，我们可以用来更新：

```
// 傻瓜循环，删除偶数元素，复制每个奇数元素
vector<int> vi = {0,1,2,3,4,5,6,7,8,9};
auto iter = vi.begin(); // 调用 begin 而不是 cbegin，因为我们要改变 vi
while (iter != vi.end()) {
    if (*iter % 2) {
        iter = vi.insert(iter, *iter); // 复制当前元素
        iter += 2; // 向前移动迭代器，跳过当前元素以及插入到它之前的元素
    } else
        iter = vi.erase(iter);          // 删除偶数元素
        // 不应向前移动迭代器，iter 指向我们删除的元素之后的元素
}
```

此程序删除 vector 中的偶数值元素，并复制每个奇数值元素。我们在调用 insert 和 erase 后都更新迭代器，因为两者都会使迭代器失效。

在调用 erase 后，不必递增迭代器，因为 erase 返回的迭代器已经指向序列中下一个元素。调用 insert 后，需要递增迭代器两次。记住，insert 在给定位置之前插入新元素，然后返回指向新插入元素的迭代器。因此，在调用 insert 后，iter 指向新插入元素，位于我们正在处理的元素之前。我们将迭代器递增两次，恰好越过了新添加的元素和正在处理的元素，指向下一个未处理的元素。

不要保存 end 返回的迭代器

当我们添加/删除 vector 或 string 的元素后，或在 deque 中首元素之外任何位置添加/删除元素后，原来 end 返回的迭代器总是会失效。因此，添加或删除元素的循环程序必须反复调用 end，而不能在循环之前保存 end 返回的迭代器，一直当作容器末尾使用。通常 C++标准库的实现中 end() 操作都很快，部分就是因为这个原因。

例如，考虑这样一个循环，它处理容器中的每个元素，在其后添加一个新元素。我们
355
希望循环能跳过新添加的元素，只处理原有元素。在每步循环之后，我们将定位迭代器，使其指向下一个原有元素。如果我们试图“优化”这个循环，在循环之前保存 end() 返回的迭代器，一直用作容器末尾，就会导致一场灾难：

```
// 灾难：此循环的行为是未定义的
auto begin = v.begin(),
     end = v.end(); // 保存尾迭代器的值是一个坏主意
while (begin != end) {
    // 做一些处理
    // 插入新值，对 begin 重新赋值，否则的话它就会失效
    ++begin; // 向前移动 begin，因为我们想在此元素之后插入元素
    begin = v.insert(begin, 42); // 插入新值
    ++begin; // 向前移动 begin 跳过我们刚刚加入的元素
}
```

此代码的行为是未定义的。在很多标准库实现上，此代码会导致无限循环。问题在于我们将 end 操作返回的迭代器保存在一个名为 end 的局部变量中。在循环体中，我们向容器

中添加了一个元素，这个操作使保存在 end 中的迭代器失效了。这个迭代器不再指向 v 中任何元素，或是 v 中尾元素之后的位置。

如果在一个循环中插入/删除 deque、string 或 vector 中的元素，不要缓存 end 返回的迭代器。

必须在每次插入操作后重新调用 end()，而不能在循环开始前保存它返回的迭代器：

```
// 更安全的方法：在每个循环步添加/删除元素后都重新计算 end
while (begin != v.end()) {
    // 做一些处理
    ++begin; // 向前移动 begin，因为我们想在此元素之后插入元素
    begin = v.insert(begin, 42); // 插入新值
    ++begin; // 向前移动 begin，跳过我们刚刚加入的元素
}
```

9.3.6 节练习

练习 9.31：第 316 页中删除偶数值元素并复制奇数值元素的程序不能用于 list 或 forward_list。为什么？修改程序，使之也能用于这些类型。

练习 9.32：在第 316 页的程序中，向下面语句这样调用 insert 是否合法？如果不合法，为什么？

```
iter = vi.insert(iter, *iter++);
```

练习 9.33：在本节最后一个例子中，如果不将 insert 的结果赋予 begin，将会发生什么？编写程序，去掉此赋值语句，验证你的答案。

练习 9.34：假定 vi 是一个保存 int 的容器，其中有偶数值也有奇数值，分析下面循环的行为，然后编写程序验证你的分析是否正确。

```
iter = vi.begin();
while (iter != vi.end())
    if (*iter % 2)
        iter = vi.insert(iter, *iter);
    ++iter;
```

9.4 vector 对象是如何增长的

为了支持快速随机访问，vector 将元素连续存储——每个元素紧挨着前一个元素存储。通常情况下，我们不必关心一个标准库类型是如何实现的，而只需关心它如何使用。然而，对于 vector 和 string，其部分实现渗透到了接口中。

假定容器中元素是连续存储的，且容器的大小是可变的，考虑向 vector 或 string 中添加元素会发生什么：如果没有空间容纳新元素，容器不可能简单地将它添加到内存中其他位置——因为元素必须连续存储。容器必须分配新的内存空间来保存已有元素和新元素，将已有元素从旧位置移动到新空间中，然后添加新元素，释放旧存储空间。如果我们每添加一个新元素，vector 就执行一次这样的内存分配和释放操作，性能会慢到不可接受。

为了避免这种代价，标准库实现者采用了可以减少容器空间重新分配次数的策略。当

不得不获取新的内存空间时，vector 和 string 的实现通常会分配比新的空间需求更大的内存空间。容器预留这些空间作为备用，可用来保存更多的新元素。这样，就不需要每次添加新元素都重新分配容器的内存空间了。

这种分配策略比每次添加新元素时都重新分配容器内存空间的策略要高效得多。其实际性能也表现得足够好——虽然 vector 在每次重新分配内存空间时都要移动所有元素，但使用此策略后，其扩张操作通常比 list 和 deque 还要快。

管理容量的成员函数

如表 9.10 所示，vector 和 string 类型提供了一些成员函数，允许我们与它的实现中内存分配部分互动。capacity 操作告诉我们容器在不扩张内存空间的情况下可以容纳多少个元素。reserve 操作允许我们通知容器它应该准备保存多少个元素。

表 9.10：容器大小管理操作

shrink_to_fit 只适用于 vector、string 和 deque。	
capacity 和 reserve 只适用于 vector 和 string。	
c.shrink_to_fit()	请将 capacity() 减少为与 size() 相同大小
c.capacity()	不重新分配内存空间的话，c 可以保存多少元素
c.reserve(n)	分配至少能容纳 n 个元素的内存空间

reserve 并不改变容器中元素的数量，它仅影响 vector 预先分配多大的内存空间。

357

只有当需要的内存空间超过当前容量时，reserve 调用才会改变 vector 的容量。如果需求大小大于当前容量，reserve 至少分配与需求一样大的内存空间（可能更大）。

如果需求大小小于或等于当前容量，reserve 什么也不做。特别是，当需求大小小于当前容量时，容器不会退回内存空间。因此，在调用 reserve 之后，capacity 将会大于或等于传递给 reserve 的参数。

这样，调用 reserve 永远也不会减少容器占用的内存空间。类似的，resize 成员函数（参见 9.3.5 节，第 314 页）只改变容器中元素的数目，而不是容器的容量。我们同样不能使用 resize 来减少容器预留的内存空间。

C++ 11

在新标准库中，我们可以调用 shrink_to_fit 来要求 deque、vector 或 string 退回不需要的内存空间。此函数指出我们不再需要任何多余的内存空间。但是，具体的实现可以选择忽略此请求。也就是说，调用 shrink_to_fit 也并不保证一定退回内存空间。

capacity 和 size

理解 capacity 和 size 的区别非常重要。容器的 size 是指它已经保存的元素的数目；而 capacity 则是在不分配新的内存空间的前提下它最多可以保存多少元素。

下面的代码展示了 size 和 capacity 之间的相互作用：

```
vector<int> ivec;
// size 应该为 0；capacity 的值依赖于具体实现
cout << " ivec: size: " << ivec.size()
     << " capacity: "  << ivec.capacity() << endl;
// 向 ivec 添加 24 个元素
```

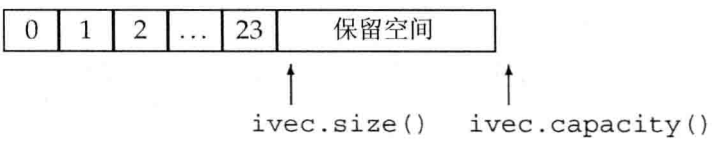
```
for (vector<int>::size_type ix = 0; ix != 24; ++ix)
    ivec.push_back(ix);

// size 应该为 24; capacity 应该大于等于 24，具体值依赖于标准库实现
cout << "ivec: size: " << ivec.size()
     << " capacity: "  << ivec.capacity() << endl
```

当在我们的系统上运行时，这段程序得到如下输出：

```
ivec: size: 0 capacity: 0
ivec: size: 24 capacity: 32
```

我们知道一个空 vector 的 size 为 0，显然在我们的标准库实现中一个空 vector 的 capacity 也为 0。当向 vector 中添加元素时，我们知道 size 与添加的元素数目相等。而 capacity 至少与 size 一样大，具体会分配多少额外空间则视标准库具体实现而定。在我们的标准库实现中，每次添加 1 个元素，共添加 24 个元素，会使 capacity 变为 32。 358

可以想象 ivec 的当前状态如下图所示：



现在可以预分配一些额外空间：

```
ivec.reserve(50); // 将 capacity 至少设定为 50，可能会更大
// size 应该为 24; capacity 应该大于等于 50，具体值依赖于标准库实现
cout << "ivec: size: " << ivec.size()
     << " capacity: " << ivec.capacity() << endl;
```

程序的输出表明 reserve 严格按照我们需求的大小分配了新的空间：

```
ivec: size: 24 capacity: 50
```

接下来可以用光这些预留空间：

```
// 添加元素用光多余容量
while (ivec.size() != ivec.capacity())
    ivec.push_back(0);
// capacity 应该未改变，size 和 capacity 不相等
cout << "ivec: size: " << ivec.size()
     << " capacity: "  << ivec.capacity() << endl;
```

程序输出表明此时我们确实用光了预留空间，size 和 capacity 相等：

```
ivec: size: 50 capacity: 50
```

由于我们只使用了预留空间，因此没有必要为 vector 分配新的空间。实际上，只要没有操作需求超出 vector 的容量，vector 就不能重新分配内存空间。

如果我们现在再添加一个新元素，vector 就不得不重新分配空间：

```
ivec.push_back(42); // 再添加一个元素
// size 应该为 51; capacity 应该大于等于 51，具体值依赖于标准库实现
cout << "ivec: size: " << ivec.size()
     << " capacity: "  << ivec.capacity() << endl;
```

这段程序的输出为

359

```
ivec: size: 51 capacity: 100
```

这表明 vector 的实现采用的策略似乎是在每次需要分配新内存空间时将当前容量翻倍。

可以调用 shrink_to_fit 来要求 vector 将超出当前大小的多余内存退回给系统：

```
ivec.shrink_to_fit(); // 要求归还内存
// size 应该未改变；capacity 的值依赖于具体实现
cout << "ivec: size: " << ivec.size()
     << " capacity: "  << ivec.capacity() << endl;
```

调用 shrink_to_fit 只是一个请求，标准库并不保证退还内存。

每个 vector 实现都可以选择自己的内存分配策略。但是必须遵守的一条原则是：只有当迫不得已时才可以分配新的内存空间。

只有在执行 insert 操作时 size 与 capacity 相等，或者调用 resize 或 reserve 时给定的大小超过当前 capacity，vector 才可能重新分配内存空间。会分配多少超过给定容量的额外空间，取决于具体实现。

虽然不同的实现可以采用不同的分配策略，但所有实现都应遵循一个原则：确保用 push_back 向 vector 添加元素的操作有高效率。从技术角度说，就是通过在一个初始为空的 vector 上调用 n 次 push_back 来创建一个 n 个元素的 vector，所花费的时间不能超过 n 的常数倍。

9.4 节练习

练习 9.35：解释一个 vector 的 capacity 和 size 有何区别。

练习 9.36：一个容器的 capacity 可能小于它的 size 吗？

练习 9.37：为什么 list 或 array 没有 capacity 成员函数？

练习 9.38：编写程序，探究在你的标准库实现中，vector 是如何增长的。

练习 9.39：解释下面程序片段做了什么：

```
vector<string> svec;
svec.reserve(1024);
string word;
while (cin >> word)
    svec.push_back(word);
svec.resize(svec.size()+svec.size()/2);
```

练习 9.40：如果上一题中的程序读入了 256 个词，在 resize 之后容器的 capacity 可能是多少？如果读入了 512 个、1000 个或 1048 个词呢？

360

9.5 额外的 string 操作

除了顺序容器共同的操作之外，string 类型还提供了一些额外的操作。这些操作中的大部分要么是提供 string 类和 C 风格字符数组之间的相互转换，要么是增加了允许我们用下标代替迭代器的版本。

标准库 string 类型定义了大量函数。幸运的是，这些函数使用了重复的模式。由于函数过多，本节初次阅读可能令人心烦，因此读者可能希望快速浏览本节。当你了解 string 支持哪些类型的操作后，就可以在需要使用一个特定操作时回过头来仔细阅读。

9.5.1 构造 string 的其他方法

除了我们在 3.2.1 节（第 76 页）已经介绍过的构造函数，以及与其他顺序容器相同的构造函数（参见表 9.3，第 299 页）外，string 类型还支持另外三个构造函数，如表 9.11 所示。

表 9.11：构造 string 的其他方法

n、len2 和 pos2 都是无符号值	
string s(cp,n)	s 是 cp 指向的数组中前 n 个字符的拷贝。此数组至少应该包含 n 个字符
string s(s2,pos2)	s 是 string s2 从下标 pos2 开始的字符的拷贝。若 pos2>s2.size()，构造函数的行为未定义
string s(s2,pos2,len2)	s 是 string s2 从下标 pos2 开始 len2 个字符的拷贝。若 pos2>s2.size()，构造函数的行为未定义。不管 len2 的值是多少，构造函数至多拷贝 s2.size()-pos2 个字符

这些构造函数接受一个 string 或一个 const char*参数，还接受（可选的）指定拷贝多少个字符的参数。当我们传递给它们的是一个 string 时，还可以给定一个下标来指出从哪里开始拷贝：

```
const char *cp = "Hello World!!!";    // 以空字符结束的数组
char noNull[] = {'H', 'i'};           // 不是以空字符结束
string s1(cp); // 拷贝 cp 中的字符直到遇到空字符; s1 == "Hello World!!!"
string s2(noNull,2);      // 从 noNull 拷贝两个字符; s2 == "Hi"
string s3(noNull);        // 未定义: noNull 不是以空字符结束
string s4(cp + 6, 5);     // 从 cp[6]开始拷贝 5 个字符; s4 == "World"
string s5(s1, 6, 5);      // 从 s1[6]开始拷贝 5 个字符; s5 == "World"
string s6(s1, 6);         // 从 s1[6]开始拷贝，直至 s1 末尾; s6 == "World!!!"
string s7(s1,6,20);       // 正确，只拷贝到 s1 末尾; s7 == "World!!!"
string s8(s1, 16);        // 抛出一个 out_of_range 异常
```

通常当我们从一个 const char*创建 string 时，指针指向的数组必须以空字符结尾，拷贝操作遇到空字符时停止。如果我们还传递给构造函数一个计数值，数组就不必以空字符结尾。如果我们未传递计数值且数组也未以空字符结尾，或者给定计数值大于数组大小，则构造函数的行为是未定义的。 361

当从一个 string 拷贝字符时，我们可以提供一个可选的开始位置和一个计数值。开始位置必须小于或等于给定的 string 的大小。如果位置大于 size，则构造函数抛出一个 out_of_range 异常（参见 5.6 节，第 173 页）。如果我们传递了一个计数值，则从给定位置开始拷贝这么多个字符。不管我们要求拷贝多少个字符，标准库最多拷贝到 string 结尾，不会更多。

substr 操作

substr 操作（参见表 9.12）返回一个 string，它是原始 string 的一部分或全部的拷贝。可以传递给 substr 一个可选的开始位置和计数值：

```
string s("hello world");
string s2 = s.substr(0, 5);      // s2 = hello
string s3 = s.substr(6);         // s3 = world
string s4 = s.substr(6, 11);     // s3 = world
string s5 = s.substr(12);        // 抛出一个 out_of_range 异常
```

如果开始位置超过了 string 的大小，则 substr 函数抛出一个 out_of_range 异常(参见 5.6 节，第 173 页)。如果开始位置加上计数值大于 string 的大小，则 substr 会调整计数值，只拷贝到 string 的末尾。

表 9.12：子字符串操作

s.substr(pos,n)	返回一个 string，包含 s 中从 pos 开始的 n 个字符的拷贝。pos 的默认值为 0。n 的默认值为 s.size()-pos，即拷贝从 pos 开始的所有字符

9.5.1 节练习

练习 9.41：编写程序，从一个 vector<char>初始化一个 string。

练习 9.42：假定你希望每次读取一个字符存入一个 string 中，而且知道最少需要读取 100 个字符，应该如何提高程序的性能？

9.5.2 改变 string 的其他方法

string 类型支持顺序容器的赋值运算符以及 assign、insert 和 erase 操作（参见 9.2.5 节，第 302 页；9.3.1 节，第 306 页；9.3.3 节，第 311 页）。除此之外，它还定义了额外的 insert 和 erase 版本。

除了接受迭代器的 insert 和 erase 版本外，string 还提供了接受下标的版本。下标指出了开始删除的位置，或是 insert 到给定值之前的位置：

```
s.insert(s.size(), 5, '!');  // 在 s 末尾插入 5 个感叹号
s.erase(s.size() - 5, 5);    // 从 s 删除最后 5 个字符
```

362 标准库 string 类型还提供了接受 C 风格字符数组的 insert 和 assign 版本。例如，我们可以将以空字符结尾的字符数组 insert 到或 assign 给一个 string：

```
const char *cp = "Stately, plump Buck";
s.assign(cp, 7);             // s == "Stately"
s.insert(s.size(), cp + 7);  // s == "Stately, plump Buck"
```

此处我们首先通过调用 assign 替换 s 的内容。我们赋予 s 的是从 cp 指向的地址开始的 7 个字符。要求赋值的字符数必须小于或等于 cp 指向的数组中的字符数（不包括结尾的空字符）。

接下来在 s 上调用 insert，我们的意图是将字符插入到 s[size()]处（不存在的）元素之前的位置。在此例中，我们将 cp 开始的 7 个字符（至多到结尾空字符之前）拷贝到 s 中。

我们也可以指定将来自其他 string 或子字符串的字符插入到当前 string 中或赋予当前 string：

```
string s = "some string", s2 = "some other string";
s.insert(0, s2); // 在 s 中位置 0 之前插入 s2 的拷贝
```

```
// 在 s[0]之前插入 s2 中 s2[0]开始的 s2.size()个字符
s.insert(0, s2, 0, s2.size());
```

append 和 replace 函数

string 类定义了两个额外的成员函数：append 和 replace，这两个函数可以改变 string 的内容。表 9.13 描述了这两个函数的功能。append 操作是在 string 末尾进行插入操作的一种简写形式：

```
string s("C++ Primer"), s2 = s; // 将 s 和 s2 初始化为"C++ Primer"
s.insert(s.size(), " 4th Ed."); // s == "C++ Primer 4th Ed."
s2.append(" 4th Ed.");   // 等价方法：将" 4th Ed."追加到 s2; s == s2
```

replace 操作是调用 erase 和 insert 的一种简写形式：

```
// 将"4th"替换为"5th"的等价方法
s.erase(11, 3);                  // s == "C++ Primer Ed."
s.insert(11, "5th");             // s == "C++ Primer 5th Ed."
// 从位置 11 开始，删除 3 个字符并插入"5th"
s2.replace(11, 3, "5th");     // 等价方法：s == s2
```

此例中调用 replace 时，插入的文本恰好与删除的文本一样长。这不是必须的，可以插入一个更长或更短的 string：

```
s.replace(11, 3, "Fifth");   // s == "C++ Primer Fifth Ed."
```

在此调用中，删除了 3 个字符，但在其位置插入了 5 个新字符。

表 9.13：修改 string 的操作

363

s.insert(*pos*,*args*)	在 *pos* 之前插入 *args* 指定的字符。*pos* 可以是一个下标或一个迭代器。接受下标的版本返回一个指向 s 的引用；接受迭代器的版本返回指向第一个插入字符的迭代器
s.erase(pos,len)	删除从位置 pos 开始的 len 个字符。如果 len 被省略，则删除从 pos 开始直至 s 末尾的所有字符。返回一个指向 s 的引用
s.assign(*args*)	将 s 中的字符替换为 *args* 指定的字符。返回一个指向 s 的引用
s.append(*args*)	将 *args* 追加到 s。返回一个指向 s 的引用
s.replace(*range*,*args*)	删除 s 中范围 *range* 内的字符，替换为 *args* 指定的字符。*range* 或者是一个下标和一个长度，或者是一对指向 s 的迭代器。返回一个指向 s 的引用

args 可以是下列形式之一；append 和 assign 可以使用所有形式。

str 不能与 s 相同，迭代器 b 和 e 不能指向 s。

str	字符串 str
str, pos, len	str 中从 pos 开始最多 len 个字
cp, len	从 cp 指向的字符数组的前（最多）len 个字符
cp	cp 指向的以空字符结尾的字符数组
n, c	n 个字符 c
b, e	迭代器 b 和 e 指定的范围内的字符
初始化列表	花括号包围的，以逗号分隔的字符列表

续表

replace 和 insert 所允许的 *args* 形式依赖于 *range* 和 *pos* 是如何指定的。				
replace (pos,len,*args*)	replace (b,e,*args*)	insert (pos,*args*)	insert (iter,*args*)	*args* 可以是
是	是	是	否	str
是	否	是	否	str,pos,len
是	是	是	否	cp,len
是	是	否	否	cp
是	是	是	是	n,c
否	是	否	是	b2,e2
否	是	否	是	初始化列表

改变 string 的多种重载函数

表 9.13 列出的 append、assign、insert 和 replace 函数有多个重载版本。根据我们如何指定要添加的字符和 string 中被替换的部分，这些函数的参数有不同版本。幸运的是，这些函数有共同的接口。

assign 和 append 函数无须指定要替换 string 中哪个部分：assign 总是替换 string 中的所有内容，append 总是将新字符追加到 string 末尾。

replace 函数提供了两种指定删除元素范围的方式。可以通过一个位置和一个长度来指定范围，也可以通过一个迭代器范围来指定。insert 函数允许我们用两种方式指定插入点：用一个下标或一个迭代器。在两种情况下，新元素都会插入到给定下标（或迭代器）之前的位置。

364

可以用好几种方式来指定要添加到 string 中的字符。新字符可以来自于另一个 string，来自于一个字符指针（指向的字符数组），来自于一个花括号包围的字符列表，或者是一个字符和一个计数值。当字符来自于一个 string 或一个字符指针时，我们可以传递一个额外的参数来控制是拷贝部分还是全部字符。

并不是每个函数都支持所有形式的参数。例如，insert 就不支持下标和初始化列表参数。类似的，如果我们希望用迭代器指定插入点，就不能用字符指针指定新字符的来源。

9.5.2 节练习

练习 9.43：编写一个函数，接受三个 string 参数 s、oldVal 和 newVal。使用迭代器及 insert 和 erase 函数将 s 中所有 oldVal 替换为 newVal。测试你的程序，用它替换通用的简写形式，如，将"tho"替换为"though"，将"thru"替换为"through"。

练习 9.44：重写上一题的函数，这次使用一个下标和 replace。

练习 9.45：编写一个函数，接受一个表示名字的 string 参数和两个分别表示前缀（如"Mr."或"Ms."）和后缀（如"Jr."或"III"）的字符串。使用迭代器及 insert 和 append 函数将前缀和后缀添加到给定的名字中，将生成的新 string 返回。

练习 9.46：重写上一题的函数，这次使用位置和长度来管理 string，并只使用 insert。

9.5.3 string 搜索操作

string 类提供了 6 个不同的搜索函数，每个函数都有 4 个重载版本。表 9.14 描述了这些搜索成员函数及其参数。每个搜索操作都返回一个 string::size_type 值，表示匹配发生位置的下标。如果搜索失败，则返回一个名为 string::npos 的 static 成员（参见 7.6 节，第 268 页）。标准库将 npos 定义为一个 const string::size_type 类型，并初始化为值-1。由于 npos 是一个 unsigned 类型，此初始值意味着 npos 等于任何 string 最大的可能大小（参见 2.1.2 节，第 32 页）。

string 搜索函数返回 string:: size_type 值，该类型是一个 unsigned 类型。因此，用一个 int 或其他带符号类型来保存这些函数的返回值不是一个好主意（参见 2.1.2 节，第 33 页）。

find 函数完成最简单的搜索。它查找参数指定的字符串，若找到，则返回第一个匹配位置的下标，否则返回 npos：

```
string name("AnnaBelle");
auto pos1 = name.find("Anna"); // pos1 == 0
```

365

这段程序返回 0，即子字符串"Anna"在"AnnaBelle"中第一次出现的下标。

搜索（以及其他 string 操作）是大小写敏感的。当在 string 中查找子字符串时，要注意大小写：

```
string lowercase("annabelle");
pos1 = lowercase.find("Anna"); // pos1 == npos
```

这段代码会将 pos1 置为 npos，因为 Anna 与 anna 不匹配。

一个更复杂一些的问题是查找与给定字符串中任何一个字符匹配的位置。例如，下面代码定位 name 中的第一个数字：

```
string numbers("0123456789"), name("r2d2");
// 返回 1，即，name 中第一个数字的下标
auto pos = name.find_first_of(numbers);
```

如果是要搜索第一个不在参数中的字符，我们应该调用 find_first_not_of。例如，为了搜索一个 string 中第一个非数字字符，可以这样做：

```
string dept("03714p3");
// 返回 5——字符'p'的下标
auto pos = dept.find_first_not_of(numbers);
```

表 9.14：string 搜索操作

搜索操作返回指定字符出现的下标，如果未找到则返回 npos。	
s.find(*args*)	查找 s 中 *args* 第一次出现的位置
s.rfind(*args*)	查找 s 中 *args* 最后一次出现的位置
s.find_first_of(*args*)	在 s 中查找 *args* 中任何一个字符第一次出现的位置。
s.find_last_of(*args*)	在 s 中查找 *args* 中任何一个字符最后一次出现的位置
s.find_first_not_of(*args*)	在 s 中查找第一个不在 *args* 中的字符
s.find_last_not_of(*args*)	在 s 中查找最后一个不在 *args* 中的字符

续表

args 必须是以下形式之一	
c,pos	从 s 中位置 pos 开始查找字符 c。pos 默认为 0
s2,pos	从 s 中位置 pos 开始查找字符串 s2。pos 默认为 0
cp,pos	从 s 中位置 pos 开始查找指针 cp 指向的以空字符结尾的 C 风格字符串。pos 默认为 0
cp,pos,n	从 s 中位置 pos 开始查找指针 cp 指向的数组的前 n 个字符。pos 和 n 无默认值

指定在哪里开始搜索

我们可以传递给 find 操作一个可选的开始位置。这个可选的参数指出从哪个位置开始进行搜索。默认情况下，此位置被置为 0。一种常见的程序设计模式是用这个可选参数在字符串中循环地搜索子字符串出现的所有位置：

366

```
string::size_type pos = 0;
// 每步循环查找 name 中下一个数
while ((pos = name.find_first_of(numbers, pos))
               != string::npos) {
    cout << "found number at index: " << pos
         << " element is " << name[pos] << endl;
    ++pos; // 移动到下一个字符
}
```

while 的循环条件将 pos 重置为从 pos 开始遇到的第一个数字的下标。只要 find_first_of 返回一个合法下标，我们就打印当前结果并递增 pos。

如果我们忽略了递增 pos，循环就永远也不会终止。为了搞清楚原因，考虑如果不做递增运算会发生什么。在第二步循环中，我们从 pos 指向的字符开始搜索。这个字符是一个数字，因此 find_first_of 会（重复地）返回 pos！

逆向搜索

到现在为止，我们已经用过的 find 操作都是由左至右搜索。标准库还提供了类似的，但由右至左搜索的操作。rfind 成员函数搜索最后一个匹配，即子字符串最靠右的出现位置：

```
string river("Mississippi");
auto first_pos = river.find("is"); // 返回 1
auto last_pos = river.rfind("is"); // 返回 4
```

find 返回下标 1，表示第一个"is"的位置，而 rfind 返回下标 4，表示最后一个"is"的位置。

类似的，find_last 函数的功能与 find_first 函数相似，只是它们返回最后一个而不是第一个匹配：

- find_last_of 搜索与给定 string 中任何一个字符匹配的最后一个字符。
- find_last_not_of 搜索最后一个不出现在给定 string 中的字符。

每个操作都接受一个可选的第二参数，可用来指出从什么位置开始搜索。

9.5.3 节练习

练习 9.47：编写程序，首先查找 string "ab2c3d7R4E6"中的每个数字字符，然后查找其中每个字母字符。编写两个版本的程序，第一个要使用 find_first_of，第二个要使用 find_first_not_of。

练习 9.48：假定 name 和 numbers 的定义如 325 页所示，numbers.find(name) 返回什么？

练习 9.49：如果一个字母延伸到中线之上，如 d 或 f，则称其有上出头部分（ascender）。如果一个字母延伸到中线之下，如 p 或 g，则称其有下出头部分（descender）。编写程序，读入一个单词文件，输出最长的既不包含上出头部分，也不包含下出头部分的单词。

9.5.4 compare 函数

除了关系运算符外（参见 3.2.2 节，第 79 页），标准库 string 类型还提供了一组 compare 函数，这些函数与 C 标准库的 strcmp 函数（参见 3.5.4 节，第 109 页）很相似。类似 strcmp，根据 s 是等于、大于还是小于参数指定的字符串，s.compare 返回 0、正数或负数。

如表 9.15 所示，compare 有 6 个版本。根据我们是要比较两个 string 还是一个 string 与一个字符数组，参数各有不同。在这两种情况下，都可以比较整个或一部分字符串。

367

表 9.15：s.compare 的几种参数形式

s2	比较 s 和 s2
pos1, n1, s2	将 s 中从 pos1 开始的 n1 个字符与 s2 进行比较
pos1, n1, s2, pos2, n2	将 s 中从 pos1 开始的 n1 个字符与 s2 中从 pos2 开始的 n2 个字符进行比较
cp	比较 s 与 cp 指向的以空字符结尾的字符数组
pos1, n1, cp	将 s 中从 pos1 开始的 n1 个字符与 cp 指向的以空字符结尾的字符数组进行比较
pos1, n1, cp, n2	将 s 中从 pos1 开始的 n1 个字符与指针 cp 指向的地址开始的 n2 个字符进行比较

9.5.5 数值转换

字符串中常常包含表示数值的字符。例如，我们用两个字符的 string 表示数值 15——字符'1'后跟字符'5'。一般情况，一个数的字符表示不同于其数值。数值 15 如果保存为 16 位的 short 类型，则其二进制位模式为 0000000000001111，而字符串"15"存为两个 Latin-1 编码的 char，二进制位模式为 0011000100110101。第一个字节表示字符'1'，其八进制值为 061，第二个字节表示'5'，其 Latin-1 编码为八进制值 065。

新标准引入了多个函数，可以实现数值数据与标准库 string 之间的转换：

C++ 11

```
int i = 42;
string s = to_string(i); // 将整数 i 转换为字符表示形式
double d = stod(s);      // 将字符串 s 转换为浮点数
```

368 此例中我们调用 to_string 将 42 转换为其对应的 string 表示，然后调用 stod 将此 string 转换为浮点值。

要转换为数值的 string 中第一个非空白符必须是数值中可能出现的字符：

```
string s2 = "pi = 3.14";
// 转换 s 中以数字开始的第一个子串，结果 d = 3.14
d = stod(s2.substr(s2.find_first_of("+-.0123456789")));
```

在这个 stod 调用中，我们调用了 find_first_of（参见 9.5.3 节，第 325 页）来获得 s 中第一个可能是数值的一部分的字符的位置。我们将 s 中从此位置开始的子串传递给 stod。stod 函数读取此参数，处理其中的字符，直至遇到不可能是数值的一部分的字符。然后它就将找到的这个数值的字符串表示形式转换为对应的双精度浮点值。

string 参数中第一个非空白符必须是符号（+ 或 -）或数字。它可以以 0x 或 0X 开头来表示十六进制数。对那些将字符串转换为浮点值的函数，string 参数也可以以小数点（.）开头，并可以包含 e 或 E 来表示指数部分。对于那些将字符串转换为整型值的函数，根据基数不同，string 参数可以包含字母字符，对应大于数字 9 的数。

如果 string 不能转换为一个数值，这些函数抛出一个 invalid_argument 异常（参见 5.6 节，第 173 页）。如果转换得到的数值无法用任何类型来表示，则抛出一个 out_of_range 异常。

表 9.16：string 和数值之间的转换

to_string(val)	一组重载函数，返回数值 val 的 string 表示。val 可以是任何算术类型（参见 2.1.1 节，第 30 页）。对每个浮点类型和 int 或更大的整型，都有相应版本的 to_string。与往常一样，小整型会被提升（参见 4.11.1 节，第 142 页）
stoi(s, p, b) stol(s, p, b) stoul(s, p, b) stoll(s, p, b) stoull(s, p, b)	返回 s 的起始子串(表示整数内容)的数值，返回值类型分别是 int、long、unsigned long、long long、unsigned long long。b 表示转换所用的基数，默认值为 10。p 是 size_t 指针，用来保存 s 中第一个非数值字符的下标，p 默认为 0，即，函数不保存下标
stof(s, p) stod(s, p) stold(s, p)	返回 s 的起始子串（表示浮点数内容）的数值，返回值类型分别是 float、double 或 long double。参数 p 的作用与整数转换函数中一样

9.5.5 节练习

练习 9.50：编写程序处理一个 vector<string>，其元素都表示整型值。计算 vector 中所有元素之和。修改程序，使之计算表示浮点值的 string 之和。

练习 9.51：设计一个类，它有三个 unsigned 成员，分别表示年、月和日。为其编写构造函数，接受一个表示日期的 string 参数。你的构造函数应该能处理不同数据格式，如 January 1, 1900、1/1/1990、Jan 1 1900 等。

9.6 容器适配器

除了顺序容器外，标准库还定义了三个顺序容器适配器：stack、queue 和 priority_queue。**适配器**（adaptor）是标准库中的一个通用概念。容器、迭代器和函数都有适配器。本质上，一个适配器是一种机制，能使某种事物的行为看起来像另外一种事物一样。一个容器适配器接受一种已有的容器类型，使其行为看起来像一种不同的类型。例如，stack 适配器接受一个顺序容器（除 array 或 forward_list 外），并使其操作起来像一个 stack 一样。表 9.17 列出了所有容器适配器都支持的操作和类型。

369

表 9.17：所有容器适配器都支持的操作和类型

size_type	一种类型，足以保存当前类型的最大对象的大小
value_type	元素类型
container_type	实现适配器的底层容器类型
A a;	创建一个名为 a 的空适配器
A a(c);	创建一个名为 a 的适配器，带有容器 c 的一个拷贝
关系运算符	每个适配器都支持所有关系运算符：==、!=、<、<=、>和>= 这些运算符返回底层容器的比较结果
a.empty()	若 a 包含任何元素，返回 false，否则返回 true
a.size()	返回 a 中的元素数目
swap(a,b) a.swap(b)	交换 a 和 b 的内容，a 和 b 必须有相同类型，包括底层容器类型也必须相同

定义一个适配器

每个适配器都定义两个构造函数：默认构造函数创建一个空对象，接受一个容器的构造函数拷贝该容器来初始化适配器。例如，假定 deq 是一个 deque<int>，我们可以用 deq 来初始化一个新的 stack，如下所示：

```
stack<int> stk(deq); // 从 deq 拷贝元素到 stk
```

默认情况下，stack 和 queue 是基于 deque 实现的，priority_queue 是在 vector 之上实现的。我们可以在创建一个适配器时将一个命名的顺序容器作为第二个类型参数，来重载默认容器类型。

370

```
// 在 vector 上实现的空栈
stack<string, vector<string>> str_stk;
// str_stk2 在 vector 上实现，初始化时保存 svec 的拷贝
stack<string, vector<string>> str_stk2(svec);
```

对于一个给定的适配器，可以使用哪些容器是有限制的。所有适配器都要求容器具有添加和删除元素的能力。因此，适配器不能构造在 array 之上。类似的，我们也不能用 forward_list 来构造适配器，因为所有适配器都要求容器具有添加、删除以及访问尾元素的能力。stack 只要求 push_back、pop_back 和 back 操作，因此可以使用除 array 和 forward_list 之外的任何容器类型来构造 stack。queue 适配器要求 back、push_back、front 和 push_front，因此它可以构造于 list 或 deque 之上，但不能基于 vector 构造。priority_queue 除了 front、push_back 和 pop_back 操作之外还要求随机访问能力，因此它可以构造于 vector 或 deque 之上，但不能基于 list 构造。

栈适配器

stack 类型定义在 stack 头文件中。表 9.18 列出了 stack 所支持的操作。下面的程序展示了如何使用 stack：

```
stack<int> intStack; // 空栈
// 填满栈
for (size_t ix = 0; ix != 10; ++ix)
    intStack.push(ix);          // intStack 保存 0 到 9 十个数
while (!intStack.empty()) {  // intStack 中有值就继续循环
    int value = intStack.top();
    // 使用栈顶值的代码
    intStack.pop(); // 弹出栈顶元素，继续循环
}
```

其中，声明语句

```
stack<int> intStack; // 空栈
```

定义了一个保存整型元素的栈 intStack，初始时为空。for 循环将 10 个元素添加到栈中，这些元素被初始化为从 0 开始连续的整数。while 循环遍历整个 stack，获取 top 值，将其从栈中弹出，直至栈空。

表 9.18：表 9.17 未列出的栈操作

栈默认基于 deque 实现，也可以在 list 或 vector 之上实现。	
s.pop()	删除栈顶元素，但不返回该元素值
s.push(item)	创建一个新元素压入栈顶，该元素通过拷贝或移动 item 而来，或者由 *args* 构造
s.emplace(*args*)	
s.top()	返回栈顶元素，但不将元素弹出栈

371 每个容器适配器都基于底层容器类型的操作定义了自己的特殊操作。我们只可以使用适配器操作，而不能使用底层容器类型的操作。例如，

```
intStack.push(ix); // intStack 保存 0 到 9 十个数
```

此语句试图在 intStack 的底层 deque 对象上调用 push_back。虽然 stack 是基于 deque 实现的，但我们不能直接使用 deque 操作。不能在一个 stack 上调用 push_back，而必须使用 stack 自己的操作——push。

队列适配器

queue 和 priority_queue 适配器定义在 queue 头文件中。表 9.19 列出了它们所支持的操作。

表 9.19：表 9.17 未列出的 queue 和 priority_queue 操作

queue 默认基于 deque 实现，priority_queue 默认基于 vector 实现； queue 也可以用 list 或 vector 实现，priority_queue 也可以用 deque 实现。	
q.pop()	返回 queue 的首元素或 priority_queue 的最高优先级的元素，但不删除此元素
q.front()	返回首元素或尾元素，但不删除此元素
q.back()	只适用于 **queue**

续表

q.top()	返回最高优先级元素，但不删除该元素
	只适用于 priority_queue
q.push(item)	在 queue 末尾或 priority_queue 中恰当的位置创建一个元素，
q.emplace(*args*)	其值为 item，或者由 *args* 构造

标准库 queue 使用一种先进先出（first-in，first-out，FIFO）的存储和访问策略。进入队列的对象被放置到队尾，而离开队列的对象则从队首删除。饭店按客人到达的顺序来为他们安排座位，就是一个先进先出队列的例子。

priority_queue 允许我们为队列中的元素建立优先级。新加入的元素会排在所有优先级比它低的已有元素之前。饭店按照客人预定时间而不是到来时间的早晚来为他们安排座位，就是一个优先队列的例子。默认情况下，标准库在元素类型上使用<运算符来确定相对优先级。我们将在 11.2.2 节（第 378 页）学习如何重载这个默认设置。

9.6 节练习

练习 9.52：使用 stack 处理括号化的表达式。当你看到一个左括号，将其记录下来。当你在一个左括号之后看到一个右括号，从 stack 中 pop 对象，直至遇到左括号，将左括号也一起弹出栈。然后将一个值（括号内的运算结果）push 到栈中，表示一个括号化的（子）表达式已经处理完毕，被其运算结果所替代。

372

小结

标准库容器是模板类型，用来保存给定类型的对象。在一个顺序容器中，元素是按顺序存放的，通过位置来访问。顺序容器有公共的标准接口：如果两个顺序容器都提供一个特定的操作，那么这个操作在两个容器中具有相同的接口和含义。

所有容器（除 array 外）都提供高效的动态内存管理。我们可以向容器中添加元素，而不必担心元素存储在哪里。容器负责管理自身的存储。vector 和 string 都提供更细致的内存管理控制，这是通过它们的 reserve 和 capacity 成员函数来实现的。

很大程度上，容器只定义了极少的操作。每个容器都定义了构造函数、添加和删除元素的操作、确定容器大小的操作以及返回指向特定元素的迭代器的操作。其他一些有用的操作，如排序或搜索，并不是由容器类型定义的，而是由标准库算法实现的，我们将在第10章介绍这些内容。

当我们使用添加和删除元素的容器操作时，必须注意这些操作可能使指向容器中元素的迭代器、指针或引用失效。很多会使迭代器失效的操作，如 insert 和 erase，都会返回一个新的迭代器，来帮助程序员维护容器中的位置。如果循环程序中使用了改变容器大小的操作，就要尤其小心其中迭代器、指针和引用的使用。

术语表

适配器（adaptor） 标准库类型、函数或迭代器，它们接受一个类型、函数或迭代器，使其行为像另外一个类型、函数或迭代器一样。标准库提供了三种顺序容器适配器：stack、queue 和 priority_queue。每个适配器都在其底层顺序容器类型之上定义了一个新的接口。

数组（array） 固定大小的顺序容器。为了定义一个 array，除了元素类型之外还必须给定大小。array 中的元素可以用其位置下标来访问。array 支持快速的随机访问。

begin 容器操作，返回一个指向容器首元素的迭代器，如果容器为空，则返回尾后迭代器。是否返回 const 迭代器依赖于容器的类型。

cbegin 容器操作，返回一个指向容器首元素的 const_iterator，如果容器为空，则返回尾后迭代器。

cend 容器操作，返回一个指向容器尾元素之后（不存在的）的 const_iterator。

容器（container） 保存一组给定类型对象的类型。每个标准库容器类型都是一个模板类型。为了定义一个容器，我们必须指定保存在容器中的元素的类型。除了 array 之外，标准库容器都是大小可变的。

deque 顺序容器。deque 中的元素可以通过位置下标来访问。支持快速的随机访问。deque 各方面都与 vector 类似，唯一的差别是，deque 支持在容器头尾位置的快速插入和删除，而且在两端插入或删除元素都不会导致重新分配空间。

373

end 容器操作，返回一个指向容器尾元素之后（不存在的）元素的迭代器。是否返回 const 迭代器依赖于容器的类型。

forward_list 顺序容器，表示一个单向链表。forward_list 中的元素只能顺序访问。从一个给定元素开始，为了访问另一个元素，我们只能遍历两者之间的所有元素。forward_list 上的迭代器不支持递减运算（--）。forward_list 支持任意位置的快速插入（或删除）操作。与其他容器不同，插入和删除发生在一个给定的

迭代器之后的位置。因此，除了通常的尾后迭代器之外，forward_list 还有一个“首前”迭代器。在添加新元素后，原有的指向 forward_list 的迭代器仍有效。在删除元素后，只有原来指向被删元素的迭代器才会失效。

迭代器范围（iterator range） 由一对迭代器指定的元素范围。第一个迭代器表示序列中第一个元素，第二个迭代器指向最后一个元素之后的位置。如果范围为空，则两个迭代器是相等的（反之亦然，如果两个迭代器不等，则它们表示一个非空范围）。如果范围非空，则必须保证，通过反复递增第一个迭代器，可以到达第二个迭代器。通过递增迭代器，序列中每个元素都能被访问到。

左闭合区间（left-inclusive interval） 值范围，包含首元素，但不包含尾元素。通常表示为[i, j)，表示序列从 i 开始（包含）直至 j 结束（不包含）。

list 顺序容器，表示一个双向链表。list 中的元素只能顺序访问。从一个给定元素开始，为了访问另一个元素，我们只能遍历两者之间的所有元素。list 上的迭代器既支持递增运算（++），也支持递减运算（--）。list 支持任意位置的快速插入（或删除）操作。当加入新元素后，迭代器仍然有效。当删除元素后，只有原来指向被删除元素的迭代器才会失效。

首前迭代器（off-the-beginning iterator） 表示一个 forward_list 开始位置之前（不存在的）元素的迭代器。是 forward_list 的成员函数 before_begin 的返回值。与 end() 迭代器类似，不能被解引用。

尾后迭代器（off-the-end iterator） 表示范围中尾元素之后位置的迭代器。通常被称为“末尾迭代器”（end iterator）。

priority_queue 顺序容器适配器，生成一个队列，插入其中的元素不放在末尾，而是根据特定的优先级排列。默认情况下，优先级用元素类型上的小于运算符确定。

queue 顺序容器适配器，生成一个类型，使我们能将新元素添加到末尾，从头部删除元素。

顺序容器（sequential container） 保存相同类型对象有序集合的类型。顺序容器中的元素通过位置来访问。

stack 顺序容器适配器，生成一个类型，使我们只能在其一端添加和删除元素。

vector 顺序容器。vector 中的元素可以通过位置下标访问。支持快速的随机访问。我们只能在 vector 末尾实现高效的元素添加/删除。向 vector 添加元素可能导致内存空间的重新分配，从而使所有指向 vector 的迭代器失效。在 vector 内部添加（或删除）元素会使所有指向插入（删除）点之后元素的迭代器失效。

第 10 章

泛型算法

内容

标准库容器定义的操作集合惊人得小。标准库并未给每个容器添加大量功能，而是提供了一组算法，这些算法中的大多数都独立于任何特定的容器。这些算法是通用的（generic，或称泛型的）：它们可用于不同类型的容器和不同类型的元素。

泛型算法和关于迭代器的更多细节，构成了本章的主要内容。

376 顺序容器只定义了很少的操作：在多数情况下，我们可以添加和删除元素、访问首尾元素、确定容器是否为空以及获得指向首元素或尾元素之后位置的迭代器。

我们可以想象用户可能还希望做其他很多有用的操作：查找特定元素、替换或删除一个特定值、重排元素顺序等。

标准库并未给每个容器都定义成员函数来实现这些操作，而是定义了一组**泛型算法**（generic algorithm）：称它们为“算法”，是因为它们实现了一些经典算法的公共接口，如排序和搜索；称它们是“泛型的”，是因为它们可以用于不同类型的元素和多种容器类型（不仅包括标准库类型，如 vector 或 list，还包括内置的数组类型），以及我们将看到的，还能用于其他类型的序列。

10.1 概述

大多数算法都定义在头文件 algorithm 中。标准库还在头文件 numeric 中定义了一组数值泛型算法。

一般情况下，这些算法并不直接操作容器，而是遍历由两个迭代器指定的一个元素范围（参见 9.2.1 节，第 296 页）来进行操作。通常情况下，算法遍历范围，对其中每个元素进行一些处理。例如，假定我们有一个 int 的 vector，希望知道 vector 中是否包含一个特定值。回答这个问题最方便的方法是调用标准库算法 find：

```
int val = 42; // 我们将查找的值
// 如果在 vec 中找到想要的元素，则返回结果指向它，否则返回结果为 vec.cend()
auto result = find(vec.cbegin(), vec.cend(), val);
// 报告结果
cout << "The value " << val
     << (result == vec.cend()
           ? " is not present" : " is present") << endl;
```

传递给 find 的前两个参数是表示元素范围的迭代器，第三个参数是一个值。find 将范围中每个元素与给定值进行比较。它返回指向第一个等于给定值的元素的迭代器。如果范围中无匹配元素，则 find 返回第二个参数来表示搜索失败。因此，我们可以通过比较返回值和第二个参数来判断搜索是否成功。我们在输出语句中执行这个检测，其中使用了条件运算符（参见 4.7 节，第 134 页）来报告搜索是否成功。

由于 find 操作的是迭代器，因此我们可以用同样的 find 函数在任何容器中查找值。例如，可以用 find 在一个 string 的 list 中查找一个给定值：

```
string val = "a value"; // 我们要查找的值
// 此调用在 list 中查找 string 元素
auto result = find(lst.cbegin(), lst.cend(), val);
```

类似的，由于指针就像内置数组上的迭代器一样，我们可以用 find 在数组中查找值：

377

```
int ia[] = {27, 210, 12, 47, 109, 83};
int val = 83;
int* result = find(begin(ia), end(ia), val);
```

此例中我们使用了标准库 begin 和 end 函数（参见 3.5.3 节，第 106 页）来获得指向 ia 中首元素和尾元素之后位置的指针，并传递给 find。

还可以在序列的子范围中查找，只需将指向子范围首元素和尾元素之后位置的迭代器

（指针）传递给 find。例如，下面的语句在 ia[1]、ia[2]和 ia[3]中查找给定元素：

```
// 在从 ia[1]开始，直至（但不包含）ia[4]的范围内查找元素
auto result = find(ia + 1, ia + 4, val);
```

算法如何工作

为了弄清这些算法如何用于不同类型的容器，让我们更近地观察一下 find。find 的工作是在一个未排序的元素序列中查找一个特定元素。概念上，find 应执行如下步骤：

1. 访问序列中的首元素。
2. 比较此元素与我们要查找的值。
3. 如果此元素与我们要查找的值匹配，find 返回标识此元素的值。
4. 否则，find 前进到下一个元素，重复执行步骤 2 和 3。
5. 如果到达序列尾，find 应停止。
6. 如果 find 到达序列末尾，它应该返回一个指出元素未找到的值。此值和步骤 3 返回的值必须具有相容的类型。

这些步骤都不依赖于容器所保存的元素类型。因此，只要有一个迭代器可用来访问元素，find 就完全不依赖于容器类型（甚至无须理会保存元素的是不是容器）。

迭代器令算法不依赖于容器，……

在上述 find 函数流程中，除了第 2 步外，其他步骤都可以用迭代器操作来实现：利用迭代器解引用运算符可以实现元素访问；如果发现匹配元素，find 可以返回指向该元素的迭代器；用迭代器递增运算符可以移动到下一个元素；尾后迭代器可以用来判断 find 是否到达给定序列的末尾；find 可以返回尾后迭代器（参见 9.2.1 节，第 296 页）来表示未找到给定元素。

……，但算法依赖于元素类型的操作

虽然迭代器的使用令算法不依赖于容器类型，但大多数算法都使用了一个（或多个）元素类型上的操作。例如，在步骤 2 中，find 用元素类型的==运算符完成每个元素与给定值的比较。其他算法可能要求元素类型支持 < 运算符。不过，我们将会看到，大多数 378 算法提供了一种方法，允许我们使用自定义的操作来代替默认的运算符。

10.1 节练习

练习 10.1：头文件 algorithm 中定义了一个名为 count 的函数，它类似 find，接受一对迭代器和一个值作为参数。count 返回给定值在序列中出现的次数。编写程序，读取 int 序列存入 vector 中，打印有多少个元素的值等于给定值。

练习 10.2：重做上一题，但读取 string 序列存入 list 中。

关键概念：算法永远不会执行容器的操作

泛型算法本身不会执行容器的操作，它们只会运行于迭代器之上，执行迭代器的操作。泛型算法运行于迭代器之上而不会执行容器操作的特性带来了一个令人惊讶但非常必要的编程假定：算法永远不会改变底层容器的大小。算法可能改变容器中保存的元素

的值，也可能在容器内移动元素，但永远不会直接添加或删除元素。

如我们将在 10.4.1 节（第 358 页）所看到的，标准库定义了一类特殊的迭代器，称为插入器（inserter）。与普通迭代器只能遍历所绑定的容器相比，插入器能做更多的事情。当给这类迭代器赋值时，它们会在底层的容器上执行插入操作。因此，当一个算法操作一个这样的迭代器时，迭代器可以完成向容器添加元素的效果，但算法自身永远不会做这样的操作。

10.2 初识泛型算法

标准库提供了超过 100 个算法。幸运的是，与容器类似，这些算法有一致的结构。比起死记硬背全部 100 多个算法，理解此结构可以帮助我们更容易地学习和使用这些算法。在本章中，我们将展示如何使用这些算法，并介绍刻画了这些算法的统一原则。附录 A 按操作方式列出了所有算法。

除了少数例外，标准库算法都对一个范围内的元素进行操作。我们将此元素范围称为“输入范围”。接受输入范围的算法总是使用前两个参数来表示此范围，两个参数分别是指向要处理的第一个元素和尾元素之后位置的迭代器。

虽然大多数算法遍历输入范围的方式相似，但它们使用范围中元素的方式不同。理解算法的最基本的方法就是了解它们是否读取元素、改变元素或是重排元素顺序。

10.2.1 只读算法

379

一些算法只会读取其输入范围内的元素，而从不改变元素。find 就是这样一种算法，我们在 10.1 节练习（第 337 页）中使用的 count 函数也是如此。

另一个只读算法是 accumulate，它定义在头文件 numeric 中。accumulate 函数接受三个参数，前两个指出了需要求和的元素的范围，第三个参数是和的初值。假定 vec 是一个整数序列，则：

```
// 对 vec 中的元素求和，和的初值是 0
int sum = accumulate(vec.cbegin(), vec.cend(), 0);
```

这条语句将 sum 设置为 vec 中元素的和，和的初值被设置为 0。

accumulate 的第三个参数的类型决定了函数中使用哪个加法运算符以及返回值的类型。

算法和元素类型

accumulate 将第三个参数作为求和起点，这蕴含着一个编程假定：将元素类型加到和的类型上的操作必须是可行的。即，序列中元素的类型必须与第三个参数匹配，或者能够转换为第三个参数的类型。在上例中，vec 中的元素可以是 int，或者是 double、long long 或任何其他可以加到 int 上的类型。

下面是另一个例子，由于 string 定义了+运算符，所以我们可以通过调用 accumulate 来将 vector 中所有 string 元素连接起来：

```
string sum = accumulate(v.cbegin(), v.cend(), string(""));
```

此调用将 v 中每个元素连接到一个 string 上，该 string 初始时为空串。注意，我们通过第三个参数显式地创建了一个 string。将空串当做一个字符串字面值传递给第三个参数是不可以的，会导致一个编译错误。

```
// 错误：const char*上没有定义+运算符
string sum = accumulate(v.cbegin(), v.cend(), "");
```

原因在于，如果我们传递了一个字符串字面值，用于保存和的对象的类型将是 const char*。如前所述，此类型决定了使用哪个+运算符。由于 const char*并没有+运算符，此调用将产生编译错误。

对于只读取而不改变元素的算法，通常最好使用 cbegin()和 cend()（参见 9.2.3 节，第 298 页）。但是，如果你计划使用算法返回的迭代器来改变元素的值，就需要使用 begin()和 end()的结果作为参数。

操作两个序列的算法

380

另一个只读算法是 equal，用于确定两个序列是否保存相同的值。它将第一个序列中的每个元素与第二个序列中的对应元素进行比较。如果所有对应元素都相等，则返回 true，否则返回 false。此算法接受三个迭代器：前两个（与以往一样）表示第一个序列中的元素范围，第三个表示第二个序列的首元素：

```
// roster2 中的元素数目应该至少与 roster1 一样多
equal(roster1.cbegin(), roster1.cend(), roster2.cbegin());
```

由于 equal 利用迭代器完成操作，因此我们可以通过调用 equal 来比较两个不同类型的容器中的元素。而且，元素类型也不必一样，只要我们能用==来比较两个元素类型即可。例如，在此例中，roster1 可以是 vector<string>，而 roster2 是 list<const char*>。

但是，equal 基于一个非常重要的假设：它假定第二个序列至少与第一个序列一样长。此算法要处理第一个序列中的每个元素，它假定每个元素在第二个序列中都有一个与之对应的元素。

那些只接受一个单一迭代器来表示第二个序列的算法，都假定第二个序列至少与第一个序列一样长。

10.2.1 节练习

练习 10.3：用 accumulate 求一个 vector<int>中的元素之和。

练习 10.4：假定 v 是一个 vector<double>，那么调用 accumulate(v.cbegin(), v.cend(), 0)有何错误（如果存在的话）？

练习 10.5：在本节对名册（roster）调用 equal 的例子中，如果两个名册中保存的都是 C 风格字符串而不是 string，会发生什么？

10.2.2 写容器元素的算法

一些算法将新值赋予序列中的元素。当我们使用这类算法时，必须注意确保序列原大

小至少不小于我们要求算法写入的元素数目。记住，算法不会执行容器操作，因此它们自身不可能改变容器的大小。

一些算法会自己向输入范围写入元素。这些算法本质上并不危险，它们最多写入与给定序列一样多的元素。

例如，算法 fill 接受一对迭代器表示一个范围，还接受一个值作为第三个参数。fill 将给定的这个值赋予输入序列中的每个元素。

```
fill(vec.begin(), vec.end(), 0); // 将每个元素重置为 0
// 将容器的一个子序列设置为 10
fill(vec.begin(), vec.begin() + vec.size()/2, 10);
```

由于 fill 向给定输入序列中写入数据，因此，只要我们传递了一个有效的输入序列，写入操作就是安全的。

381

关键概念：迭代器参数

一些算法从两个序列中读取元素。构成这两个序列的元素可以来自于不同类型的容器。例如，第一个序列可能保存于一个 vector 中，而第二个序列可能保存于一个 list、deque、内置数组或其他容器中。而且，两个序列中元素的类型也不要求严格匹配。算法要求的只是能够比较两个序列中的元素。例如，对 equal 算法，元素类型不要求相同，但是我们必须能使用==来比较来自两个序列中的元素。

操作两个序列的算法之间的区别在于我们如何传递第二个序列。一些算法，例如 equal，接受三个迭代器：前两个表示第一个序列的范围，第三个表示第二个序列中的首元素。其他算法接受四个迭代器：前两个表示第一个序列的元素范围，后两个表示第二个序列的范围。

用一个单一迭代器表示第二个序列的算法都假定第二个序列至少与第一个一样长。确保算法不会试图访问第二个序列中不存在的元素是程序员的责任。例如，算法 equal 会将其第一个序列中的每个元素与第二个序列中的对应元素进行比较。如果第二个序列是第一个序列的一个子集，则程序会产生一个严重错误——equal 会试图访问第二个序列中末尾之后（不存在）的元素。

算法不检查写操作

一些算法接受一个迭代器来指出一个单独的目的位置。这些算法将新值赋予一个序列中的元素，该序列从目的位置迭代器指向的元素开始。例如，函数 fill_n 接受一个单迭代器、一个计数值和一个值。它将给定值赋予迭代器指向的元素开始的指定个元素。我们可以用 fill_n 将一个新值赋予 vector 中的元素：

```
vector<int> vec; // 空 vector
// 使用 vec，赋予它不同值
fill_n(vec.begin(), vec.size(), 0); // 将所有元素重置为 0
```

函数 fill_n 假定写入指定个元素是安全的。即，如下形式的调用

```
fill_n(dest, n, val)
```

fill_n 假定 dest 指向一个元素，而从 dest 开始的序列至少包含 n 个元素。

382

一个初学者非常容易犯的错误是在一个空容器上调用 fill_n（或类似的写元素的算法）：

```
vector<int> vec; // 空向量
// 灾难：修改 vec 中的 10 个（不存在）元素
fill_n(vec.begin(), 10, 0);
```

这个调用是一场灾难。我们指定了要写入 10 个元素，但 vec 中并没有元素——它是空的。这条语句的结果是未定义的。

向目的位置迭代器写入数据的算法假定目的位置足够大，能容纳要写入的元素。

介绍 back_inserter

一种保证算法有足够元素空间来容纳输出数据的方法是使用**插入迭代器**（insert iterator）。插入迭代器是一种向容器中添加元素的迭代器。通常情况，当我们通过一个迭代器向容器元素赋值时，值被赋予迭代器指向的元素。而当我们通过一个插入迭代器赋值时，一个与赋值号右侧值相等的元素被添加到容器中。

我们将在 10.4.1 节中（第 358 页）详细介绍插入迭代器的内容。但是，为了展示如何用算法向容器写入数据，我们现在将使用**back_inserter**，它是定义在头文件 iterator 中的一个函数。

back_inserter 接受一个指向容器的引用，返回一个与该容器绑定的插入迭代器。当我们通过此迭代器赋值时，赋值运算符会调用 push_back 将一个具有给定值的元素添加到容器中：

```
vector<int> vec; // 空向量
auto it = back_inserter(vec); // 通过它赋值会将元素添加到 vec 中
*it = 42; // vec 中现在有一个元素，值为 42
```

我们常常使用 back_inserter 来创建一个迭代器，作为算法的目的位置来使用。例如：

```
vector<int> vec; // 空向量
// 正确：back_inserter 创建一个插入迭代器，可用来向 vec 添加元素
fill_n(back_inserter(vec), 10, 0); // 添加 10 个元素到 vec
```

在每步迭代中，fill_n 向给定序列的一个元素赋值。由于我们传递的参数是 back_inserter 返回的迭代器，因此每次赋值都会在 vec 上调用 push_back。最终，这条 fill_n 调用语句向 vec 的末尾添加了 10 个元素，每个元素的值都是 0.

拷贝算法

拷贝（copy）算法是另一个向目的位置迭代器指向的输出序列中的元素写入数据的算法。此算法接受三个迭代器，前两个表示一个输入范围，第三个表示目的序列的起始位置。此算法将输入范围中的元素拷贝到目的序列中。传递给 copy 的目的序列至少要包含与输入序列一样多的元素，这一点很重要。

383

我们可以用 copy 实现内置数组的拷贝，如下面代码所示：

```
int a1[] = {0,1,2,3,4,5,6,7,8,9};
int a2[sizeof(a1)/sizeof(*a1)]; // a2 与 a1 大小一样
// ret 指向拷贝到 a2 的尾元素之后的位置
auto ret = copy(begin(a1), end(a1), a2); // 把 a1 的内容拷贝给 a2
```

此例中我们定义了一个名为 a2 的数组，并使用 sizeof 确保 a2 与数组 a1 包含同样多的

元素（参见 4.9 节，第 139 页）。接下来我们调用 copy 完成从 a1 到 a2 的拷贝。在调用 copy 后，两个数组中的元素具有相同的值。

copy 返回的是其目的位置迭代器（递增后）的值。即，ret 恰好指向拷贝到 a2 的尾元素之后的位置。

多个算法都提供所谓的“拷贝”版本。这些算法计算新元素的值，但不会将它们放置在输入序列的末尾，而是创建一个新序列保存这些结果。

例如，replace 算法读入一个序列，并将其中所有等于给定值的元素都改为另一个值。此算法接受 4 个参数：前两个是迭代器，表示输入序列，后两个一个是要搜索的值，另一个是新值。它将所有等于第一个值的元素替换为第二个值：

```
// 将所有值为 0 的元素改为 42
replace(ilst.begin(), ilst.end(), 0, 42);
```

此调用将序列中所有的 0 都替换为 42。如果我们希望保留原序列不变，可以调用 replace_copy。此算法接受额外第三个迭代器参数，指出调整后序列的保存位置：

```
// 使用 back_inserter 按需要增长目标序列
replace_copy(ilst.cbegin(), ilst.cend(),
             back_inserter(ivec), 0, 42);
```

此调用后，ilst 并未改变，ivec 包含 ilst 的一份拷贝，不过原来在 ilst 中值为 0 的元素在 ivec 中都变为 42。

10.2.2 节练习

练习 10.6：编写程序，使用 fill_n 将一个序列中的 int 值都设置为 0。

练习 10.7：下面程序是否有错误？如果有，请改正。

```
(a) vector<int> vec; list<int> lst; int i;
    while (cin >> i)
        lst.push_back(i);
    copy(lst.cbegin(), lst.cend(), vec.begin());

(b) vector<int> vec;
    vec.reserve(10); // reverse 将在 9.4 节（第 318 页）介绍
    fill_n(vec.begin(), 10, 0);
```

练习 10.8：本节提到过，标准库算法不会改变它们所操作的容器的大小。为什么使用 back_inserter 不会使这一断言失效？

10.2.3 重排容器元素的算法

某些算法会重排容器中元素的顺序，一个明显的例子是 sort。调用 sort 会重排输入序列中的元素，使之有序，它是利用元素类型的<运算符来实现排序的。

例如，假定我们想分析一系列儿童故事中所用的词汇。假定已有一个 vector，保存了多个故事的文本。我们希望化简这个 vector，使得每个单词只出现一次，而不管单词在任意给定文档中到底出现了多少次。

为了便于说明问题，我们将使用下面简单的故事作为输入：

```
the quick red fox jumps over the slow red turtle
```

给定此输入，我们的程序应该生成如下 vector：

fox	jumps	over	quick	red	slow	the	turtle

消除重复单词

384

为了消除重复单词，首先将 vector 排序，使得重复的单词都相邻出现。一旦 vector 排序完毕，我们就可以使用另一个称为 unique 的标准库算法来重排 vector，使得不重复的元素出现在 vector 的开始部分。由于算法不能执行容器的操作，我们将使用 vector 的 erase 成员来完成真正的删除操作：

```
void elimDups(vector<string> &words)
{
    // 按字典序排序 words，以便查找重复单词
    sort(words.begin(), words.end());
    // unique 重排输入范围，使得每个单词只出现一次
    // 排列在范围的前部，返回指向不重复区域之后一个位置的迭代器
    auto end_unique = unique(words.begin(), words.end());
    // 使用向量操作 erase 删除重复单词
    words.erase(end_unique, words.end());
}
```

sort 算法接受两个迭代器，表示要排序的元素范围。在此例中，我们排序整个 vector。完成 sort 后，words 的顺序如下所示：

fox	jumps	over	quick	red	red	slow	the	the	turtle

注意，单词 red 和 the 各出现了两次。

使用 unique

words 排序完毕后，我们希望将每个单词都只保存一次。unique 算法重排输入序列，将相邻的重复项“消除”，并返回一个指向不重复值范围末尾的迭代器。调用 unique 后，vector 将变为：

385

fox	jumps	over	quick	red	slow	the	turtle	???	???

end_unique
（最后一个不重复元素之后的位置）

words 的大小并未改变，它仍有 10 个元素。但这些元素的顺序被改变了——相邻的重复元素被“删除”了。我们将删除打引号是因为 unique 并不真的删除任何元素，它只是覆盖相邻的重复元素，使得不重复元素出现在序列开始部分。unique 返回的迭代器指向最后一个不重复元素之后的位置。此位置之后的元素仍然存在，但我们不知道它们的值是什么。

标准库算法对迭代器而不是容器进行操作。因此，算法不能（直接）添加或删除元素。

使用容器操作删除元素

为了真正地删除无用元素，我们必须使用容器操作，本例中使用 erase（参见 9.3.3

节，第 311 页）。我们删除从 end_unique 开始直至 words 末尾的范围内的所有元素。这个调用之后，words 包含来自输入的 8 个不重复的单词。

值得注意的是，即使 words 中没有重复单词，这样调用 erase 也是安全的。在此情况下，unique 会返回 words.end()。因此，传递给 erase 的两个参数具有相同的值：words.end()。迭代器相等意味着传递给 erase 的元素范围为空。删除一个空范围没有什么不良后果，因此程序即使在输入中无重复元素的情况下也是正确的。

10.2.3 节练习

练习 10.9：实现你自己的 elimDups。测试你的程序，分别在读取输入后、调用 unique 后以及调用 erase 后打印 vector 的内容。

练习 10.10：你认为算法不改变容器大小的原因是什么？

10.3 定制操作

很多算法都会比较输入序列中的元素。默认情况下，这类算法使用元素类型的<或==运算符完成比较。标准库还为这些算法定义了额外的版本，允许我们提供自己定义的操作来代替默认运算符。

386 例如，sort 算法默认使用元素类型的<运算符。但可能我们希望的排序顺序与<所定义的顺序不同，或是我们的序列可能保存的是未定义<运算符的元素类型（如 Sales_data）。在这两种情况下，都需要重载 sort 的默认行为。

10.3.1 向算法传递函数

作为一个例子，假定希望在调用 elimDups（参见 10.2.3 节，第 343 页）后打印 vector 的内容。此外还假定希望单词按其长度排序，大小相同的再按字典序排列。为了按长度重排 vector，我们将使用 sort 的第二个版本，此版本是重载过的，它接受第三个参数，此参数是一个**谓词**（predicate）。

谓词

谓词是一个可调用的表达式，其返回结果是一个能用作条件的值。标准库算法所使用的谓词分为两类：**一元谓词**（unary predicate，意味着它们只接受单一参数）和**二元谓词**（binary predicate，意味着它们有两个参数）。接受谓词参数的算法对输入序列中的元素调用谓词。因此，元素类型必须能转换为谓词的参数类型。

接受一个二元谓词参数的 sort 版本用这个谓词代替<来比较元素。我们提供给 sort 的谓词必须满足将在 11.2.2 节（第 378 页）中所介绍的条件。当前，我们只需知道，此操作必须在输入序列中所有可能的元素值上定义一个一致的序。我们在 6.2.2 节（第 189 页）中定义的 isShorter 就是一个满足这些要求的函数，因此可以将 isShorter 传递给 sort。这样做会将元素按大小重新排序：

```
// 比较函数，用来按长度排序单词
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

```
// 按长度由短至长排序 words
sort(words.begin(), words.end(), isShorter);
```

如果 words 包含的数据与 10.2.3 节（第 343 页）中一样，此调用会将 words 重排，使得所有长度为 3 的单词排在长度为 4 的单词之前，然后是长度为 5 的单词，依此类推。

排序算法

在我们将 words 按大小重排的同时，还希望具有相同长度的元素按字典序排列。为了保持相同长度的单词按字典序排列，可以使用 stable_sort 算法。这种稳定排序算法维持相等元素的原有顺序。

通常情况下，我们不关心有序序列中相等元素的相对顺序，它们毕竟是相等的。但是，在本例中，我们定义的“相等”关系表示“具有相同长度”。而具有相同长度的元素，如果看其内容，其实还是各不相同的。通过调用 stable_sort，可以保持等长元素间的字典序： 387

```
elimDups(words); // 将 words 按字典序重排，并消除重复单词
// 按长度重新排序，长度相同的单词维持字典序
stable_sort(words.begin(), words.end(), isShorter);
for (const auto &s : words) // 无须拷贝字符串
    cout << s << " "; // 打印每个元素，以空格分隔
cout << endl;
```

假定在此调用前 words 是按字典序排列的，则调用之后，words 会按元素大小排序，而长度相同的单词会保持字典序。如果我们对原来的 vector 内容运行这段代码，输出为：

```
fox red the over slow jumps quick turtle
```

10.3.1 节练习

练习 10.11：编写程序，使用 stable_sort 和 isShorter 将传递给你的 elimDups 版本的 vector 排序。打印 vector 的内容，验证你的程序的正确性。

练习 10.12：编写名为 compareIsbn 的函数，比较两个 Sales_data 对象的 isbn() 成员。使用这个函数排序一个保存 Sales_data 对象的 vector。

练习 10.13：标准库定义了名为 partition 的算法，它接受一个谓词，对容器内容进行划分，使得谓词为 true 的值会排在容器的前半部分，而使谓词为 false 的值会排在后半部分。算法返回一个迭代器，指向最后一个使谓词为 true 的元素之后的位置。编写函数，接受一个 string，返回一个 bool 值，指出 string 是否有 5 个或更多字符。使用此函数划分 words。打印出长度大于等于 5 的元素。

10.3.2 lambda 表达式

根据算法接受一元谓词还是二元谓词，我们传递给算法的谓词必须严格接受一个或两个参数。但是，有时我们希望进行的操作需要更多参数，超出了算法对谓词的限制。例如，为上一节最后一个练习所编写的程序中，就必须将大小 5 硬编码到划分序列的谓词中。如果在编写划分序列的谓词时，可以不必为每个可能的大小都编写一个独立的谓词，显然更有实际价值。

一个相关的例子是，我们将修改 10.3.1 节（第 345 页）中的程序，求大于等于一个给定长度的单词有多少。我们还会修改输出，使程序只打印大于等于给定长度的单词。

388 我们将此函数命名为 biggies，其框架如下所示：

```
void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    elimDups(words); // 将 words 按字典序排序，删除重复单词
    // 按长度排序，长度相同的单词维持字典序
    stable_sort(words.begin(), words.end(), isShorter);
    // 获取一个迭代器，指向第一个满足 size()>= sz 的元素
    // 计算满足 size >= sz 的元素的数目
    // 打印长度大于等于给定值的单词，每个单词后面接一个空格
}
```

我们的新问题是在 vector 中寻找第一个大于等于给定长度的元素。一旦找到了这个元素，根据其位置，就可以计算出有多少元素的长度大于等于给定值。

我们可以使用标准库 find_if 算法来查找第一个具有特定大小的元素。类似 find（参见 10.1 节，第 336 页），find_if 算法接受一对迭代器，表示一个范围。但与 find 不同的是，find_if 的第三个参数是一个谓词。find_if 算法对输入序列中的每个元素调用给定的这个谓词。它返回第一个使谓词返回非 0 值的元素，如果不存在这样的元素，则返回尾迭代器。

编写一个函数，令其接受一个 string 和一个长度，并返回一个 bool 值表示该 string 的长度是否大于给定长度，是一件很容易的事情。但是，find_if 接受一元谓词——我们传递给 find_if 的任何函数都必须严格接受一个参数，以便能用来自输入序列的一个元素调用它。没有任何办法能传递给它第二个参数来表示长度。为了解决此问题，需要使用另外一些语言特性。

介绍 lambda

我们可以向一个算法传递任何类别的**可调用对象**（callable object）。对于一个对象或一个表达式，如果可以对其使用调用运算符（参见 1.5.2 节，第 21 页），则称它为可调用的。即，如果 e 是一个可调用的表达式，则我们可以编写代码 e(args)，其中 args 是一个逗号分隔的一个或多个参数的列表。

到目前为止，我们使用过的仅有的两种可调用对象是函数和函数指针（参见 6.7 节，第 221 页）。还有其他两种可调用对象：重载了函数调用运算符的类，我们将在 14.8 节（第 506 页）介绍，以及 **lambda 表达式**（lambda expression）。

C++11 一个 lambda 表达式表示一个可调用的代码单元。我们可以将其理解为一个未命名的内联函数。与任何函数类似，一个 lambda 具有一个返回类型、一个参数列表和一个函数体。但与函数不同，lambda 可能定义在函数内部。一个 lambda 表达式具有如下形式

```
[capture list] (parameter list) -> return type { function body }
```

其中，*capture list*（捕获列表）是一个 lambda 所在函数中定义的局部变量的列表（通常为空）；*return type*、*parameter list* 和 *function body* 与任何普通函数一样，分别表示返回类型、参数列表和函数体。但是，与普通函数不同，lambda 必须使用尾置返回（参见 6.3.3 节，第 206 页）来指定返回类型。

我们可以忽略参数列表和返回类型，但必须永远包含捕获列表和函数体

389

```
auto f = [] { return 42; };
```

此例中，我们定义了一个可调用对象 f，它不接受参数，返回 42。

lambda 的调用方式与普通函数的调用方式相同，都是使用调用运算符：

```
cout << f() << endl; // 打印 42
```

在 lambda 中忽略括号和参数列表等价于指定一个空参数列表。在此例中，当调用 f 时，参数列表是空的。如果忽略返回类型，lambda 根据函数体中的代码推断出返回类型。如果函数体只是一个 return 语句，则返回类型从返回的表达式的类型推断而来。否则，返回类型为 void。

如果 lambda 的函数体包含任何单一 return 语句之外的内容，且未指定返回类型，则返回 void。

向 lambda 传递参数

与一个普通函数调用类似，调用一个 lambda 时给定的实参被用来初始化 lambda 的形参。通常，实参和形参的类型必须匹配。但与普通函数不同，lambda 不能有默认参数（参见 6.5.1 节，第 211 页）。因此，一个 lambda 调用的实参数目永远与形参数目相等。一旦形参初始化完毕，就可以执行函数体了。

作为一个带参数的 lambda 的例子，我们可以编写一个与 isShorter 函数完成相同功能的 lambda：

```
[](const string &a, const string &b)
  { return a.size() < b.size();}
```

空捕获列表表明此 lambda 不使用它所在函数中的任何局部变量。lambda 的参数与 isShorter 的参数类似，是 const string 的引用。lambda 的函数体也与 isShorter 类似，比较其两个参数的 size()，并根据两者的相对大小返回一个布尔值。

如下所示，可以使用此 lambda 来调用 stable_sort：

```
// 按长度排序，长度相同的单词维持字典序
stable_sort(words.begin(), words.end(),
            [](const string &a, const string &b)
              { return a.size() < b.size();});
```

当 stable_sort 需要比较两个元素时，它就会调用给定的这个 lambda 表达式。

使用捕获列表

我们现在已经准备好解决原来的问题了——编写一个可以传递给 find_if 的可调用表达式。我们希望这个表达式能将输入序列中每个 string 的长度与 biggies 函数中的 sz 参数的值进行比较。 390

虽然一个 lambda 可以出现在一个函数中，使用其局部变量，但它只能使用那些明确指明的变量。一个 lambda 通过将局部变量包含在其捕获列表中来指出将会使用这些变量。捕获列表指引 lambda 在其内部包含访问局部变量所需的信息。

在本例中，我们的 lambda 会捕获 sz，并只有单一的 string 参数。其函数体会将 string 的大小与捕获的 sz 的值进行比较：

```
[sz](const string &a)
    { return a.size() >= sz; };
```

lambda 以一对[]开始，我们可以在其中提供一个以逗号分隔的名字列表，这些名字都是它所在函数中定义的。

由于此 lambda 捕获 sz，因此 lambda 的函数体可以使用 sz。lambda 不捕获 words，因此不能访问此变量。如果我们给 lambda 提供一个空捕获列表，则代码会编译错误：

```
// 错误：sz 未捕获
[](const string &a)
    { return a.size() >= sz; };
```

一个 lambda 只有在其捕获列表中捕获一个它所在函数中的局部变量，才能在函数体中使用该变量。

调用 find_if

使用此 lambda，我们就可以查找第一个长度大于等于 sz 的元素：

```
// 获取一个迭代器，指向第一个满足 size()>= sz 的元素
auto wc = find_if(words.begin(), words.end(),
    [sz](const string &a)
        { return a.size() >= sz; });
```

这里对 find_if 的调用返回一个迭代器，指向第一个长度不小于给定参数 sz 的元素。如果这样的元素不存在，则返回 words.end()的一个拷贝。

我们可以使用 find_if 返回的迭代器来计算从它开始到 words 的末尾一共有多少个元素（参见 3.4.2 节，第 99 页）：

```
// 计算满足 size >= sz 的元素的数目
auto count = words.end() - wc;
cout << count << " " << make_plural(count, "word", "s")
     << " of length " << sz << " or longer" << endl;
```

我们的输出语句调用 make_plural（参见 6.3.2 节，第 201 页）来输出“word”或“words”，具体输出哪个取决于大小是否等于 1。

391

for_each 算法

问题的最后一部分是打印 words 中长度大于等于 sz 的元素。为了达到这一目的，我们可以使用 for_each 算法。此算法接受一个可调用对象，并对输入序列中每个元素调用此对象：

```
// 打印长度大于等于给定值的单词，每个单词后面接一个空格
for_each(wc, words.end(),
         [](const string &s){cout << s << " ";});
cout << endl;
```

此 lambda 中的捕获列表为空，但其函数体中还是使用了两个名字：s 和 cout，前者是它自己的参数。

捕获列表为空，是因为我们只对 lambda 所在函数中定义的（非 static）变量使用捕获列表。一个 lambda 可以直接使用定义在当前函数之外的名字。在本例中，cout 不是定义在 biggies 中的局部名字，而是定义在头文件 iostream 中。因此，只要在 biggies 出现的作用域中包含了头文件 iostream，我们的 lambda 就可以使用 cout。