

第 10 章

泛型算法

内容

10.1 概述	336
10.2 初识泛型算法	338
10.3 定制操作	344
10.4 再探迭代器	357
10.5 泛型算法结构	365
10.6 特定容器算法	369
小结	371
术语表	371

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` 将范围内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。如果范围内无匹配元素，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。

< 378

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，但它们使用范围内元素的方式不同。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` 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的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。

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

< 385

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：

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

< 386

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：

```
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);
    // 获得一个迭代器, 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

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

389

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`。



捕获列表只用于局部非 static 变量，lambda 可以直接使用局部 static 变量和在它所在函数之外声明的名字。

完整的 biggies

到目前为止，我们已经解决了程序的所有细节，下面就是完整的程序：

```
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(),
                 [] (const string &a, const string &b)
                     { return a.size() < b.size(); });
    // 获得一个迭代器，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; });
    // 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;
    // 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;
}
```

10.3.2 节练习

392

练习 10.14: 编写一个 lambda，接受两个 int，返回它们的和。

练习 10.15: 编写一个 lambda，捕获它所在函数的 int，并接受一个 int 参数。lambda 应该返回捕获的 int 和 int 参数的和。

练习 10.16: 使用 lambda 编写你自己版本的 biggies。

练习 10.17: 重写 10.3.1 节练习 10.12(第 345 页)的程序，在对 sort 的调用中使用 lambda 来代替函数 compareIsbn。

练习 10.18: 重写 biggies，用 partition 替代 find_if。我们在 10.3.1 节练习 10.13(第 345 页) 中介绍了 partition 算法。

练习 10.19: 用 stable_partition 重写前一题的程序，与 stable_sort 类似，在划分后的序列中维持原有元素的顺序。

10.3.3 lambda 捕获和返回

当定义一个 lambda 时，编译器生成一个与 lambda 对应的新的（未命名的）类类型。我们将在 14.8.1 节（第 507 页）介绍这种类是如何生成的。目前，可以这样理解，当向一个函数传递一个 lambda 时，同时定义了一个新类型和该类型的一个对象：传递的参数就

是此编译器生成的类类型的未命名对象。类似的，当使用 `auto` 定义一个用 `lambda` 初始化的变量时，定义了一个从 `lambda` 生成的类型的对象。

默认情况下，从 `lambda` 生成的类都包含一个对应该 `lambda` 所捕获的变量的数据成员。类似任何普通类的数据成员，`lambda` 的数据成员也在 `lambda` 对象创建时被初始化。

值捕获

类似参数传递，变量的捕获方式也可以是值或引用。表 10.1（第 352 页）列出了几种不同的构造捕获列表的方式。到目前为止，我们的 `lambda` 采用值捕获的方式。与传值参数类似，采用值捕获的前提是变量可以拷贝。与参数不同，被捕获的变量的值是在 `lambda` 创建时拷贝，而不是调用时拷贝：

```
void fcn1()
{
    size_t v1 = 42; // 局部变量
    // 将 v1 拷贝到名为 f 的可调用对象
    auto f = [v1] { return v1; };
    v1 = 0;
    auto j = f(); // j 为 42; f 保存了我们创建它时 v1 的拷贝
}
```

由于被捕获变量的值是在 `lambda` 创建时拷贝，因此随后对其修改不会影响到 `lambda` 内对应的值。

393 引用捕获

我们定义 `lambda` 时可以采用引用方式捕获变量。例如：

```
void fcn2()
{
    size_t v1 = 42; // 局部变量
    // 对象 f2 包含 v1 的引用
    auto f2 = [&v1] { return v1; };
    v1 = 0;
    auto j = f2(); // j 为 0; f2 保存 v1 的引用，而非拷贝
}
```

`v1` 之前的 `&` 指出 `v1` 应该以引用方式捕获。一个以引用方式捕获的变量与其他任何类型的引用的行为类似。当我们在 `lambda` 函数体内使用此变量时，实际上使用的是引用所绑定的对象。在本例中，当 `lambda` 返回 `v1` 时，它返回的是 `v1` 指向的对象的值。

引用捕获与返回引用（参见 6.3.2 节，第 201 页）有着相同的问题和限制。如果我们采用引用方式捕获一个变量，就必须确保被引用的对象在 `lambda` 执行的时候是存在的。`lambda` 捕获的都是局部变量，这些变量在函数结束后就不复存在了。如果 `lambda` 可能在函数结束后执行，捕获的引用指向的局部变量已经消失。

引用捕获有时是必要的。例如，我们可能希望 `biggies` 函数接受一个 `ostream` 的引用，用来输出数据，并接受一个字符作为分隔符：

```
void biggies(vector<string> &words,
             vector<string>::size_type sz,
             ostream &os = cout, char c = ' ')
{
    // 与之前例子一样的重排 words 的代码
```

```
// 打印 count 的语句改为打印到 os
for_each(words.begin(), words.end(),
         [&os, c](const string &s) { os << s << c; });
}
```

我们不能拷贝 `ostream` 对象（参见 8.1.1 节，第 279 页），因此捕获 `os` 的唯一方法就是捕获其引用（或指向 `os` 的指针）。

当我们向一个函数传递一个 `lambda` 时，就像本例中调用 `for_each` 那样，`lambda` 会立即执行。在此情况下，以引用方式捕获 `os` 没有问题，因为当 `for_each` 执行时，`biggies` 中的变量是存在的。

我们也可以从一个函数返回 `lambda`。函数可以直接返回一个可调用对象，或者返回一个类对象，该类含有可调用对象的数据成员。如果函数返回一个 `lambda`，则与函数不能返回一个局部变量的引用类似，此 `lambda` 也不能包含引用捕获。



WARNING

当以引用方式捕获一个变量时，必须保证在 `lambda` 执行时变量是存在的。

< 394

建议：尽量保持 `lambda` 的变量捕获简单化

一个 `lambda` 捕获从 `lambda` 被创建（即，定义 `lambda` 的代码执行时）到 `lambda` 自身执行（可能有多次执行）这段时间内保存的相关信息。确保 `lambda` 每次执行的时候这些信息都有预期的意义，是程序员的责任。

捕获一个普通变量，如 `int`、`string` 或其他非指针类型，通常可以采用简单的值捕获方式。在此情况下，只需关注变量在捕获时是否有我们所需的值就可以了。

如果我们捕获一个指针或迭代器，或采用引用捕获方式，就必须确保在 `lambda` 执行时，绑定到迭代器、指针或引用的对象仍然存在。而且，需要保证对象具有预期的值。在 `lambda` 从创建到它执行的这段时间内，可能有代码改变绑定的对象的值。也就是说，在指针（或引用）被捕获的时刻，绑定的对象的值是我们所期望的，但在 `lambda` 执行时，该对象的值可能已经完全不同了。

一般来说，我们应该尽量减少捕获的数据量，来避免潜在的捕获导致的问题。而且，如果可能的话，应该避免捕获指针或引用。

隐式捕获

除了显式列出我们希望使用的来自所在函数的变量之外，还可以让编译器根据 `lambda` 体中的代码来推断我们要使用哪些变量。为了指示编译器推断捕获列表，应在捕获列表中写一个`&`或`=`。`&`告诉编译器采用捕获引用方式，`=`则表示采用值捕获方式。例如，我们可以重写传递给 `find_if` 的 `lambda`:

```
// sz 为隐式捕获，值捕获方式
wc = find_if(words.begin(), words.end(),
              [=](const string &s)
                  { return s.size() >= sz; });
```

如果我们希望对一部分变量采用值捕获，对其他变量采用引用捕获，可以混合使用隐式捕获和显式捕获:

```
void biggies(vector<string> &words,
```

```

        vector<string>::size_type sz,
        ostream &os = cout, char c = ' ')
    {
        // 其他处理与前例一样
        // os 隐式捕获, 引用捕获方式; c 显式捕获, 值捕获方式
        for_each(words.begin(), words.end(),
                  [&, c](const string &s) { os << s << c; });
        // os 显式捕获, 引用捕获方式; c 隐式捕获, 值捕获方式
        for_each(words.begin(), words.end(),
                  [=, &os](const string &s) { os << s << c; });
    }
}

```

395> 当我们混合使用隐式捕获和显式捕获时, 捕获列表中的第一个元素必须是一个&或=。此符号指定了默认捕获方式为引用或值。

当混合使用隐式捕获和显式捕获时, 显式捕获的变量必须使用与隐式捕获不同的方式。即, 如果隐式捕获是引用方式(使用了&), 则显式捕获命名变量必须采用值方式, 因此不能在其名字前使用&。类似的, 如果隐式捕获采用的是值方式(使用了=), 则显式捕获命名变量必须采用引用方式, 即, 在名字前使用&。

表 10.1: lambda 捕获列表

[]	空捕获列表。lambda 不能使用所在函数中的变量。一个 lambda 只有捕获变量后才能使用它们
[names]	names 是一个逗号分隔的名字列表, 这些名字都是 lambda 所在函数的局部变量。默认情况下, 捕获列表中的变量都被拷贝。名字前如果使用了&, 则采用引用捕获方式
[&]	隐式捕获列表, 采用引用捕获方式。lambda 体中所使用的来自所在函数的实体都采用引用方式使用
[=]	隐式捕获列表, 采用值捕获方式。lambda 体将拷贝所使用的来自所在函数的实体的值
[&, identifier_list]	identifier_list 是一个逗号分隔的列表, 包含 0 个或多个来自所在函数的变量。这些变量采用值捕获方式, 而任何隐式捕获的变量都采用引用方式捕获。identifier_list 中的名字前面不能使用&
[=, identifier_list]	identifier_list 中的变量都采用引用方式捕获, 而任何隐式捕获的变量都采用值方式捕获。identifier_list 中的名字不能包括 this, 且这些名字之前必须使用&

可变 lambda

默认情况下, 对于一个值被拷贝的变量, lambda 不会改变其值。如果我们希望能改变一个被捕获的变量的值, 就必须在参数列表首加上关键字 mutable。因此, 可变 lambda 能省略参数列表:

```

void fcn3()
{
    size_t v1 = 42; // 局部变量
    // f 可以改变它所捕获的变量的值
    auto f = [v1] () mutable { return ++v1; };
    v1 = 0;
    auto j = f(); // j 为 43
}

```

一个引用捕获的变量是否（如往常一样）可以修改依赖于此引用指向的是一个 `const` 类型还是一个非 `const` 类型：

```
void fcn4()
{
    size_t v1 = 42; // 局部变量
    // v1 是一个非 const 变量的引用
    // 可以通过 f2 中的引用来改变它
    auto f2 = [&v1] { return ++v1; };
    v1 = 0;
    auto j = f2(); // j 为 1
}
```

< 396

指定 lambda 返回类型

到目前为止，我们所编写的 `lambda` 都只包含单一的 `return` 语句。因此，我们还未遇到必须指定返回类型的情况。默认情况下，如果一个 `lambda` 体包含 `return` 之外的任何语句，则编译器假定此 `lambda` 返回 `void`。与其他返回 `void` 的函数类似，被推断返回 `void` 的 `lambda` 不能返回值。

下面给出了一个简单的例子，我们可以使用标准库 `transform` 算法和一个 `lambda` 来将一个序列中的每个负数替换为其绝对值：

```
transform(vi.begin(), vi.end(), vi.begin(),
         [](int i) { return i < 0 ? -i : i; });
```

函数 `transform` 接受三个迭代器和一个可调用对象。前两个迭代器表示输入序列，第三个迭代器表示目的位置。算法对输入序列中每个元素调用可调用对象，并将结果写到目的位置。如本例所示，目的位置迭代器与表示输入序列开始位置的迭代器可以是相同的。当输入迭代器和目的迭代器相同时，`transform` 将输入序列中每个元素替换为可调用对象操作该元素得到的结果。

在本例中，我们传递给 `transform` 一个 `lambda`，它返回其参数的绝对值。`lambda` 体是单一的 `return` 语句，返回一个条件表达式的结果。我们无须指定返回类型，因为可以根据条件运算符的类型推断出来。

但是，如果我们将程序改写为看起来是等价的 `if` 语句，就会产生编译错误：

```
// 错误：不能推断 lambda 的返回类型
transform(vi.begin(), vi.end(), vi.begin(),
         [](int i) { if (i < 0) return -i; else return i; });
```

编译器推断这个版本的 `lambda` 返回类型为 `void`，但它返回了一个 `int` 值。

当我们需要为一个 `lambda` 定义返回类型时，必须使用尾置返回类型（参见 6.3.3 节，第 206 页）：

```
transform(vi.begin(), vi.end(), vi.begin(),
         [](int i) -> int
         { if (i < 0) return -i; else return i; });
```

在此例中，传递给 `transform` 的第四个参数是一个 `lambda`，它的捕获列表是空的，接受单一 `int` 参数，返回一个 `int` 值。它的函数体是一个返回其参数的绝对值的 `if` 语句。

C++
11

397

10.3.3 节练习

练习 10.20: 标准库定义了一个名为 `count_if` 的算法。类似 `find_if`，此函数接受一对迭代器，表示一个输入范围，还接受一个谓词，会对输入范围中每个元素执行。`count_if` 返回一个计数值，表示谓词有多少次为真。使用 `count_if` 重写我们程序中统计有多少单词长度超过 6 的部分。

练习 10.21: 编写一个 `lambda`，捕获一个局部 `int` 变量，并递减变量值，直至它变为 0。一旦变量变为 0，再调用 `lambda` 应该不再递减变量。`lambda` 应该返回一个 `bool` 值，指出捕获的变量是否为 0。



10.3.4 参数绑定

对于那种只在一两个地方使用的简单操作，`lambda` 表达式是最有用的。如果我们需要在很多地方使用相同的操作，通常应该定义一个函数，而不是多次编写相同的 `lambda` 表达式。类似的，如果一个操作需要很多语句才能完成，通常使用函数更好。

如果 `lambda` 的捕获列表为空，通常可以用函数来代替它。如前面章节所示，既可以用一个 `lambda`，也可以用函数 `isShorter` 来实现将 `vector` 中的单词按长度排序。类似的，对于打印 `vector` 内容的 `lambda`，编写一个函数来替换它也是很容易的事情，这个函数只需接受一个 `string` 并在标准输出上打印它即可。

但是，对于捕获局部变量的 `lambda`，用函数来替换它就不是那么容易了。例如，我们在 `find_if` 调用中的 `lambda` 比较一个 `string` 和一个给定大小。我们可以很容易地编写一个完成同样工作的函数：

```
bool check_size(const string &s, string::size_type sz)
{
    return s.size() >= sz;
}
```

但是，我们不能用这个函数作为 `find_if` 的一个参数。如前文所示，`find_if` 接受一个一元谓词，因此传递给 `find_if` 的可调用对象必须接受单一参数。`biggies` 传递给 `find_if` 的 `lambda` 使用捕获列表来保存 `sz`。为了用 `check_size` 来代替此 `lambda`，必须解决如何向 `sz` 形参传递一个参数的问题。

标准库 `bind` 函数

 我们可以解决向 `check_size` 传递一个长度参数的问题，方法是使用一个新的名为 `bind` 的标准库函数，它定义在头文件 `functional` 中。可以将 `bind` 函数看作一个通用的函数适配器（参见 9.6 节，第 329 页），它接受一个可调用对象，生成一个新的可调用对象来“适应”原对象的参数列表。

398

调用 `bind` 的一般形式为：

```
auto newCallable = bind(callable, arg_list);
```

其中，`newCallable` 本身是一个可调用对象，`arg_list` 是一个逗号分隔的参数列表，对应给定的 `callable` 的参数。即，当我们调用 `newCallable` 时，`newCallable` 会调用 `callable`，并传递给它 `arg_list` 中的参数。

`arg_list` 中的参数可能包含形如 `_n` 的名字，其中 `n` 是一个整数。这些参数是“占位符”，

表示 *newCallable* 的参数，它们占据了传递给 *newCallable* 的参数的“位置”。数值 *n* 表示生成的可调用对象中参数的位置：*_1* 为 *newCallable* 的第一个参数，*_2* 为第二个参数，依此类推。

绑定 *check_size* 的 *sz* 参数

作为一个简单的例子，我们将使用 *bind* 生成一个调用 *check_size* 的对象，如下所示，它用一个定值作为其大小参数来调用 *check_size*：

```
// check6 是一个可调用对象，接受一个 string 类型的参数  
// 并用此 string 和值 6 来调用 check_size  
auto check6 = bind(check_size, _1, 6);
```

此 *bind* 调用只有一个占位符，表示 *check6* 只接受单一参数。占位符出现在 *arg_list* 的第一个位置，表示 *check6* 的此参数对应 *check_size* 的第一个参数。此参数是一个 *const string&*。因此，调用 *check6* 必须传递给它一个 *string* 类型的参数，*check6* 会将此参数传递给 *check_size*。

```
string s = "hello";  
bool b1 = check6(s); // check6(s) 会调用 check_size(s, 6)
```

使用 *bind*，我们可以将原来基于 *lambda* 的 *find_if* 调用：

```
auto wc = find_if(words.begin(), words.end(),  
                   [sz](const string &a)
```

替换为如下使用 *check_size* 的版本：

```
auto wc = find_if(words.begin(), words.end(),  
                   bind(check_size, _1, sz));
```

此 *bind* 调用生成一个可调用对象，将 *check_size* 的第二个参数绑定到 *sz* 的值。当 *find_if* 对 *words* 中的 *string* 调用这个对象时，这些对象会调用 *check_size*，将给定的 *string* 和 *sz* 传递给它。因此，*find_if* 可以有效地对输入序列中每个 *string* 调用 *check_size*，实现 *string* 的大小与 *sz* 的比较。

使用 *placeholders* 名字

399

名字 *_n* 都定义在一个名为 *placeholders* 的命名空间中，而这个命名空间本身定义在 *std* 命名空间（参见 3.1 节，第 74 页）中。为了使用这些名字，两个命名空间都要写上。与我们的其他例子类似，对 *bind* 的调用代码假定之前已经恰当地使用了 *using* 声明。例如，*_1* 对应的 *using* 声明为：

```
using std::placeholders::_1;
```

此声明说明我们要使用的名字 *_1* 定义在命名空间 *placeholders* 中，而此命名空间又定义在命名空间 *std* 中。

对每个占位符名字，我们都必须提供一个单独的 *using* 声明。编写这样的声明很烦人，也很容易出错。可以使用另外一种不同形式的 *using* 语句（详细内容将在 18.2.2 节（第 702 页）中介绍），而不是分别声明每个占位符，如下所示：

```
using namespace namespace_name;
```

这种形式说明希望所有来自 *namespace_name* 的名字都可以在我们的程序中直接使用。例如：

```
using namespace std::placeholders;
```

使得由 placeholders 定义的所有名字都可用。与 bind 函数一样，placeholders 命名空间也定义在 functional 头文件中。

bind 的参数

如前文所述，我们可以用 bind 修正参数的值。更一般的，可以用 bind 绑定给定可调用对象中的参数或重新安排其顺序。例如，假定 f 是一个可调用对象，它有 5 个参数，则下面对 bind 的调用：

```
// g 是一个有两个参数的可调用对象
auto g = bind(f, a, b, _2, c, _1);
```

生成一个新的可调用对象，它有两个参数，分别用占位符_2 和_1 表示。这个新的可调用对象将它自己的参数作为第三个和第五个参数传递给 f。f 的第一个、第二个和第四个参数分别被绑定到给定的值 a、b 和 c 上。

传递给 g 的参数按位置绑定到占位符。即，第一个参数绑定到_1，第二个参数绑定到_2。因此，当我们调用 g 时，其第一个参数将被传递给 f 作为最后一个参数，第二个参数将被传递给 f 作为第三个参数。实际上，这个 bind 调用会将

```
g(_1, _2)
```

映射为

```
f(a, b, _2, c, _1)
```

即，对 g 的调用会调用 f，用 g 的参数代替占位符，再加上绑定的参数 a、b 和 c。例如，调用 g(X, Y) 会调用

```
f(a, b, Y, c, X)
```

400> 用 bind 重排参数顺序

下面是用 bind 重排参数顺序的一个具体例子，我们可以用 bind 颠倒 isShorter 的含义：

```
// 按单词长度由短至长排序
sort(words.begin(), words.end(), isShorter);
// 按单词长度由长至短排序
sort(words.begin(), words.end(), bind(isShorter, _2, _1));
```

在第一个调用中，当 sort 需要比较两个元素 A 和 B 时，它会调用 isShorter(A, B)。在第二个对 sort 的调用中，传递给 isShorter 的参数被交换过来了。因此，当 sort 比较两个元素时，就好像调用 isShorter(B, A) 一样。

绑定引用参数

默认情况下，bind 的那些不是占位符的参数被拷贝到 bind 返回的可调用对象中。但是，与 lambda 类似，有时对有些绑定的参数我们希望以引用方式传递，或是要绑定参数的类型无法拷贝。

例如，为了替换一个引用方式捕获 ostream 的 lambda：

```
// os 是一个局部变量，引用一个输出流
// c 是一个局部变量，类型为 char
for_each(words.begin(), words.end(),
```

```
[&os, c](const string &s) { os << s << c; });
```

可以很容易地编写一个函数，完成相同的工作：

```
ostream &print(ostream &os, const string &s, char c)
{
    return os << s << c;
}
```

但是，不能直接用 bind 来代替对 os 的捕获：

```
// 错误：不能拷贝 os
for_each(words.begin(), words.end(), bind(print, os, _1, ' '));
```

原因在于 bind 拷贝其参数，而我们不能拷贝一个 ostream。如果我们希望传递给 bind 一个对象而又不拷贝它，就必须使用标准库 **ref** 函数：

```
for_each(words.begin(), words.end(),
         bind(print, ref(os), _1, ' '));
```

函数 ref 返回一个对象，包含给定的引用，此对象是可以拷贝的。标准库中还有一个 **cref** 函数，生成一个保存 const 引用的类。与 bind 一样，函数 ref 和 cref 也定义在头文件 functional 中。

向后兼容：参数绑定

401

旧版本 C++ 提供的绑定函数参数的语言特性限制更多，也更复杂。标准库定义了两个分别名为 bind1st 和 bind2nd 的函数。类似 bind，这两个函数接受一个函数作为参数，生成一个新的可调用对象，该对象调用给定函数，并将绑定的参数传递给它。但是，这些函数分别只能绑定第一个或第二个参数。由于这些函数局限太强，在新标准中已被弃用（deprecated）。所谓被弃用的特性就是在新版本中不再支持的特性。新的 C++ 程序应该使用 bind。

10.3.4 节练习

练习 10.22：重写统计长度小于等于 6 的单词数量的程序，使用函数代替 lambda。

练习 10.23：bind 接受几个参数？

练习 10.24：给定一个 string，使用 bind 和 check_size 在一个 int 的 vector 中查找第一个大于 string 长度的值。

练习 10.25：在 10.3.2 节（第 349 页）的练习中，编写了一个使用 partition 的 biggies 版本。使用 check_size 和 bind 重写此函数。

10.4 再探迭代器

除了为每个容器定义的迭代器之外，标准库在头文件 iterator 中还定义了额外几种迭代器。这些迭代器包括以下几种。

- **插入迭代器**（insert iterator）：这些迭代器被绑定到一个容器上，可用来向容器插入元素。
- **流迭代器**（stream iterator）：这些迭代器被绑定到输入或输出流上，可用来遍历所

关联的 IO 流。

- **反向迭代器 (reverse iterator)**: 这些迭代器向后而不是向前移动。除了 `forward_list` 之外的标准库容器都有反向迭代器。
- **移动迭代器 (move iterator)**: 这些专用的迭代器不是拷贝其中的元素，而是移动它们。我们将在 13.6.2 节（第 480 页）介绍移动迭代器。



10.4.1 插入迭代器

插入器是一种迭代器适配器（参见 9.6 节，第 329 页），它接受一个容器，生成一个迭代器，能实现向给定容器添加元素。当我们通过一个插入迭代器进行赋值时，该迭代器调用容器操作来向给定容器的指定位置插入一个元素。表 10.2 列出了这种迭代器支持的操作。

表 10.2: 插入迭代器操作

<code>it = t</code>	在 <code>it</code> 指定的当前位置插入值 <code>t</code> 。假定 <code>c</code> 是 <code>it</code> 绑定的容器，依赖于插入迭代器的不同种类，此赋值会分别调用 <code>c.push_back(t)</code> 、 <code>c.push_front(t)</code> 或 <code>c.insert(t,p)</code> ，其中 <code>p</code> 为传递给 <code>inserter</code> 的迭代器位置
<code>*it, ++it, it++</code>	这些操作虽然存在，但不会对 <code>it</code> 做任何事情。每个操作都返回 <code>it</code>

402

插入器有三种类型，差异在于元素插入的位置：

- **back_inserter** (参见 10.2.2 节，第 341 页) 创建一个使用 `push_back` 的迭代器。
- **front_inserter** 创建一个使用 `push_front` 的迭代器。
- **inserter** 创建一个使用 `insert` 的迭代器。此函数接受第二个参数，这个参数必须是一个指向给定容器的迭代器。元素将被插入到给定迭代器所表示的元素之前。



只有在容器支持 `push_front` 的情况下，我们才可以使用 `front_inserter`。类似的，只有在容器支持 `push_back` 的情况下，我们才能使用 `back_inserter`。

理解插入器的工作过程是很重要的：当调用 `inserter(c, iter)` 时，我们得到一个迭代器，接下来使用它时，会将元素插入到 `iter` 原来所指向的元素之前的位置。即，如果 `it` 是由 `inserter` 生成的迭代器，则下面这样的赋值语句

```
*it = val;
```

其效果与下面代码一样

```
it = c.insert(it, val); // it 指向新加入的元素
++it; // 递增 it 使它指向原来的元素
```

`front_inserter` 生成的迭代器的行为与 `inserter` 生成的迭代器完全不一样。当我们使用 `front_inserter` 时，元素总是插入到容器第一个元素之前。即使我们传递给 `inserter` 的位置原来指向第一个元素，只要我们在此元素之前插入一个新元素，此元素就不再是容器的首元素了：

```
list<int> lst = {1,2,3,4};
list<int> lst2, lst3; // 空 list
```

```
// 拷贝完成之后，lst2 包含 4 3 2 1
copy(lst.cbegin(), lst.cend(), front_inserter(lst2));
// 拷贝完成之后，lst3 包含 1 2 3 4
copy(lst.cbegin(), lst.cend(), inserter(lst3, lst3.begin()));
```

当调用 `front_inserter(c)` 时，我们得到一个插入迭代器，接下来会调用 `push_front`。当每个元素被插入到容器 `c` 中时，它变为 `c` 的新的首元素。因此，`front_inserter` 生成的迭代器会将插入的元素序列的顺序颠倒过来，而 `inserter` 和 `back_inserter` 则不会。

10.4.1 节练习

403

练习 10.26：解释三种插入迭代器的不同之处。

练习 10.27：除了 `unique`（参见 10.2.3 节，第 343 页）之外，标准库还定义了名为 `unique_copy` 的函数，它接受第三个迭代器，表示拷贝不重复元素的目的位置。编写一个程序，使用 `unique_copy` 将一个 `vector` 中不重复的元素拷贝到一个初始为空的 `list` 中。

练习 10.28：一个 `vector` 中保存 1 到 9，将其拷贝到三个其他容器中。分别使用 `inserter`、`back_inserter` 和 `front_inserter` 将元素添加到三个容器中。对每种 `inserter`，估计输出序列是怎样的，运行程序验证你的估计是否正确。

10.4.2 iostream 迭代器



虽然 `iostream` 类型不是容器，但标准库定义了可以用于这些 IO 类型对象的迭代器（参见 8.1 节，第 278 页）。`istream_iterator`（参见表 10.3）读取输入流，`ostream_iterator`（参见表 10.4 节，第 361 页）向一个输出流写数据。这些迭代器将它们对应的流当作一个特定类型的元素序列来处理。通过使用流迭代器，我们可以用泛型算法从流对象读取数据以及向其写入数据。

istream_iterator 操作

当创建一个流迭代器时，必须指定迭代器将要读写的对象类型。一个 `istream_iterator` 使用 `>>` 来读取流。因此，`istream_iterator` 要读取的类型必须定义了输入运算符。当创建一个 `istream_iterator` 时，我们可以将它绑定到一个流。当然，我们还可以默认初始化迭代器，这样就创建了一个可以当作尾后值使用的迭代器。

```
istream_iterator<int> int_it(cin); // 从 cin 读取 int
istream_iterator<int> int_eof; // 尾后迭代器
ifstream in("afile");
istream_iterator<string> str_it(in); // 从 "afile" 读取字符串
```

下面是一个用 `istream_iterator` 从标准输入读取数据，存入一个 `vector` 的例子：

```
istream_iterator<int> in_iter(cin); // 从 cin 读取 int
istream_iterator<int> eof; // istream 尾后迭代器
while (in_iter != eof) // 当有数据可供读取时
    // 后置递增运算读取流，返回迭代器的旧值
    // 解引用迭代器，获得从流读取的前一个值
    vec.push_back(*in_iter++);
```

此循环从 `cin` 读取 `int` 值，保存在 `vec` 中。在每个循环步中，循环体代码检查 `in_iter` 是否等于 `eof`。`eof` 被定义为空的 `istream_iterator`，从而可以当作尾后迭代器来使用。对于一个绑定到流的迭代器，一旦其关联的流遇到文件尾或遇到 IO 错误，迭代器的值就与尾后迭代器相等。

此程序最困难的部分是传递给 `push_back` 的参数，其中用到了解引用运算符和后置递增运算符。该表达式的计算过程与我们之前写过的其他结合解引用和后置递增运算的表达式一样（参见 4.5 节，第 131 页）。后置递增运算会从流中读取下一个值，向前推进，但返回的是迭代器的旧值。迭代器的旧值包含了从流中读取的前一个值，对迭代器进行解引用就能获得此值。

我们可以将程序重写为如下形式，这体现了 `istream_iterator` 更有用的地方：

```
istream_iterator<int> in_iter(cin), eof; // 从 cin 读取 int
vector<int> vec(in_iter, eof); // 从迭代器范围构造 vec
```

本例中我们用一对表示元素范围的迭代器来构造 `vec`。这两个迭代器是 `istream_iterator`，这意味着元素范围是通过从关联的流中读取数据获得的。这个构造函数从 `cin` 中读取数据，直至遇到文件尾或者遇到一个不是 `int` 的数据为止。从流中读取的数据被用来构造 `vec`。

表 10.3: `istream_iterator` 操作

<code>istream_iterator<T> in(is);</code>	<code>in</code> 从输入流 <code>is</code> 读取类型为 <code>T</code> 的值
<code>istream_iterator<T> end;</code>	读取类型为 <code>T</code> 的值的 <code>istream_iterator</code> 迭代器，表示尾后位置
<code>in1 == in2</code>	<code>in1</code> 和 <code>in2</code> 必须读取相同类型。如果它们都是尾后迭代器，或绑定到相同的输入，则两者相等
<code>in1 != in2</code>	<code>in1</code> 和 <code>in2</code> 必须读取相同类型。如果它们都是尾后迭代器，或绑定到相同的输入，则两者相等
<code>*in</code>	返回从流中读取的值
<code>in->mem</code>	与 <code>(*in).mem</code> 的含义相同
<code>++in, in++</code>	使用元素类型所定义的 <code>>></code> 运算符从输入流中读取下一个值。与以往一样，前置版本返回一个指向递增后迭代器的引用，后置版本返回旧值

使用算法操作流迭代器

由于算法使用迭代器操作来处理数据，而流迭代器又至少支持某些迭代器操作，因此我们至少可以用某些算法来操作流迭代器。我们在 10.5.1 节（第 365 页）会看到如何分辨哪些算法可以用于流迭代器。下面是一个例子，我们可以用一对 `istream_iterator` 来调用 `accumulate`：

```
istream_iterator<int> in(cin), eof;
cout << accumulate(in, eof, 0) << endl;
```

此调用会计算出从标准输入读取的值的和。如果输入为：

```
23 109 45 89 6 34 12 90 34 23 56 23 8 89 23
```

则输出为 664。

405 > `istream_iterator` 允许使用懒惰求值

当我们把一个 `istream_iterator` 绑定到一个流时，标准库并不保证迭代器立即从流读取数据。具体实现可以推迟从流中读取数据，直到我们使用迭代器时才真正读取。标

准库中的实现所保证的是，在我们第一次解引用迭代器之前，从流中读取数据的操作已经完成了。对于大多数组程序来说，立即读取还是推迟读取没什么差别。但是，如果我们创建了一个 `istream_iterator`，没有使用就销毁了，或者我们正在从两个不同的对象同步读取同一个流，那么何时读取可能就很重要了。

`ostream_iterator` 操作

我们可以对任何具有输出运算符（`<<`运算符）的类型定义 `ostream_iterator`。当创建一个 `ostream_iterator` 时，我们可以提供（可选的）第二参数，它是一个字符串，在输出每个元素后都会打印此字符串。此字符串必须是一个 C 风格字符串（即，一个字符串字面常量或者一个指向以空字符结尾的字符数组的指针）。必须将 `ostream_iterator` 绑定到一个指定的流，不允许空的或表示尾后位置的 `ostream_iterator`。

表 10.4: `ostream_iterator` 操作

<code>ostream_iterator<T> out(os);</code>	<code>out</code> 将类型为 <code>T</code> 的值写到输出流 <code>os</code> 中
<code>ostream_iterator<T> out(os, d);</code>	<code>out</code> 将类型为 <code>T</code> 的值写到输出流 <code>os</code> 中，每个值后面都输出一个 <code>d</code> 。 <code>d</code> 指向一个空字符结尾的字符串数组
<code>out = val</code>	用 <code><<</code> 运算符将 <code>val</code> 写入到 <code>out</code> 所绑定的 <code>ostream</code> 中。 <code>val</code> 的类型必须与 <code>out</code> 可写的类型兼容
<code>*out, ++out, out++</code>	这些运算符是存在的，但不对 <code>out</code> 做任何事情。每个运算符都返回 <code>out</code>

我们可以用 `ostream_iterator` 来输出值的序列：

```
ostream_iterator<int> out_iter(cout, " ");
for (auto e : vec)
    *out_iter++ = e; // 赋值语句实际上将元素写到 cout
cout << endl;
```

此程序将 `vec` 中的每个元素写到 `cout`，每个元素后加一个空格。每次向 `out_iter` 赋值时，写操作就会被提交。

值得注意的是，当我们向 `out_iter` 赋值时，可以忽略解引用和递增运算。即，循环可以重写成下面的样子：

```
for (auto e : vec)
    out_iter = e; // 赋值语句将元素写到 cout
cout << endl;
```

运算符*和++实际上对 `ostream_iterator` 对象不做任何事情，因此忽略它们对我们的程序没有任何影响。但是，推荐第一种形式。在这种写法中，流迭代器的使用与其他迭代器的使用保持一致。如果想将此循环改为操作其他迭代器类型，修改起来非常容易。而且，对于读者来说，此循环的行为也更为清晰。

可以通过调用 `copy` 来打印 `vec` 中的元素，这比编写循环更为简单：

```
copy(vec.begin(), vec.end(), out_iter);
cout << endl;
```

406

使用流迭代器处理类类型

我们可以为任何定义了输入运算符 (`>>`) 的类型创建 `istream_iterator` 对象。类似的，只要类型有输出运算符 (`<<`)，我们就可以为其定义 `ostream_iterator`。由于 `Sales_item` 既有输入运算符也有输出运算符，因此可以使用 IO 迭代器重写 1.6 节（第 21 页）中的书店程序：

```
istream_iterator<Sales_item> item_iter(cin), eof;
ostream_iterator<Sales_item> out_iter(cout, "\n");
// 将第一笔交易记录存在 sum 中，并读取下一条记录
Sales_item sum = *item_iter++;
while (item_iter != eof) {
    // 如果当前交易记录（存在 item_iter 中）有着相同的 ISBN 号
    if (item_iter->isbn() == sum.isbn())
        sum += *item_iter++; // 将其加到 sum 上并读取下一条记录
    else {
        out_iter = sum; // 输出 sum 当前值
        sum = *item_iter++; // 读取下一条记录
    }
}
out_iter = sum; // 记得打印最后一组记录的和
```

此程序使用 `item_iter` 从 `cin` 读取 `Sales_item` 交易记录，并将和写入 `cout`，每个结果后面都跟一个换行符。定义了自己的迭代器后，我们就可以用 `item_iter` 读取第一条交易记录，用它的值来初始化 `sum`：

```
// 将第一条交易记录保存在 sum 中，并读取下一条记录
Sales_item sum = *item_iter++;
```

此处，我们对 `item_iter` 执行后置递增操作，对结果进行解引用操作。这个表达式读取下一条交易记录，并用之前保存在 `item_iter` 中的值来初始化 `sum`。

`while` 循环会反复执行，直至在 `cin` 上遇到文件尾为止。在 `while` 循环体中，我们检查 `sum` 与刚刚读入的记录是否对应同一本书。如果两者的 ISBN 不同，我们将 `sum` 赋予 `out_iter`，这将会打印 `sum` 的当前值，并接着打印一个换行符。在打印了前一本书的交易金额之和后，我们将最近读入的交易记录的副本赋予 `sum`，并递增迭代器，这将读取下一条交易记录。循环会这样持续下去，直至遇到错误或文件尾。在退出之前，记住要打印输入中最后一本书的交易金额之和。

407

10.4.2 节练习

练习 10.29: 编写程序，使用流迭代器读取一个文本文件，存入一个 `vector` 中的 `string` 里。

练习 10.30: 使用流迭代器、`sort` 和 `copy` 从标准输入读取一个整数序列，将其排序，并将结果写到标准输出。

练习 10.31: 修改前一题的程序，使其只打印不重复的元素。你的程序应使用 `unique_copy`（参见 10.4.1 节，第 359 页）。

练习 10.32: 重写 1.6 节（第 21 页）中的书店程序，使用一个 `vector` 保存交易记录，使用不同算法完成处理。使用 `sort` 和 10.3.1 节（第 345 页）中的 `compareIsbn` 函数来排序交易记录，然后使用 `find` 和 `accumulate` 求和。

练习 10.33: 编写程序，接受三个参数：一个输入文件和两个输出文件的文件名。输入文件保存的应该是整数。使用 `istream_iterator` 读取输入文件。使用 `ostream_iterator` 将奇数写入第一个输出文件，每个值之后都跟一个空格。将偶数写入第二个输出文件，每个值都独占一行。

10.4.3 反向迭代器

反向迭代器就是在容器中从尾元素向首元素反向移动的迭代器。对于反向迭代器，递增（以及递减）操作的含义会颠倒过来。递增一个反向迭代器 (`++it`) 会移动到前一个元素；递减一个迭代器 (`--it`) 会移动到下一个元素。

除了 `forward_list` 之外，其他容器都支持反向迭代器。我们可以通过调用 `rbegin`、`rend`、`crbegin` 和 `crend` 成员函数来获得反向迭代器。这些成员函数返回指向容器尾元素和首元素之前一个位置的迭代器。与普通迭代器一样，反向迭代器也有 `const` 和非 `const` 版本。

图 10.1 显示了一个名为 `vec` 的假设的 `vector` 上的 4 种迭代器：

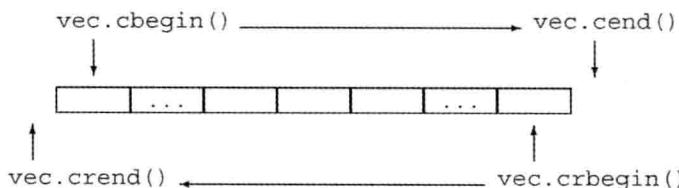


图 10.1：比较 `cbegin/cend` 和 `crbegin/crend`

下面的循环是一个使用反向迭代器的例子，它按逆序打印 `vec` 中的元素：

```
vector<int> vec = {0,1,2,3,4,5,6,7,8,9};  
// 从尾元素到首元素的反向迭代器  
for (auto r_iter = vec.crbegin(); // 将 r_iter 绑定到尾元素  
      r_iter != vec.crend(); // crend 指向首元素之前的位置  
      ++r_iter) // 实际是递减，移动到前一个元素  
    cout << *r_iter << endl; // 打印 9, 8, 7, ... 0
```

408

虽然颠倒递增和递减运算符的含义可能看起来令人混淆，但这样做使我们可以用算法透明地向前或向后处理容器。例如，可以通过向 `sort` 传递一对反向迭代器来将 `vector` 整理为递减序：

```
sort(vec.begin(), vec.end()); // 按“正常序”排序 vec  
// 按逆序排序：将最小元素放在 vec 的末尾  
sort(vec.rbegin(), vec.rend());
```

反向迭代器需要递减运算符

不必惊讶，我们只能从既支持`++`也支持`--`的迭代器来定义反向迭代器。毕竟反向迭代器的目的是在序列中反向移动。除了 `forward_list` 之外，标准容器上的其他迭代器都既支持递增运算又支持递减运算。但是，流迭代器不支持递减运算，因为不可能在一个流中反向移动。因此，不可能从一个 `forward_list` 或一个流迭代器创建反向迭代器。

反向迭代器和其他迭代器间的关系

假定有一个名为 `line` 的 `string`，保存着一个逗号分隔的单词列表，我们希望打印



`line` 中的第一个单词。使用 `find` 可以很容易地完成这一任务：

```
// 在一个逗号分隔的列表中查找第一个元素
auto comma = find(line.cbegin(), line.cend(), ',');
cout << string(line.cbegin(), comma) << endl;
```

如果 `line` 中有逗号，那么 `comma` 将指向这个逗号；否则，它将等于 `line.cend()`。当我们打印从 `line.cbegin()` 到 `comma` 之间的内容时，将打印到逗号为止的字符，或者打印整个 `string`（如果其中不含逗号的话）。

如果希望打印最后一个单词，可以改用反向迭代器：

```
// 在一个逗号分隔的列表中查找最后一个元素
auto rcomma = find(line.crbegin(), line.crend(), ',');
```

由于我们将 `crbegin()` 和 `crend()` 传递给 `find`，`find` 将从 `line` 的最后一个字符开始向前搜索。当 `find` 完成后，如果 `line` 中有逗号，则 `rcomma` 指向最后一个逗号——即，它指向反向搜索中找到的第一个逗号。如果 `line` 中没有逗号，则 `rcomma` 指向 `line.crend()`。

当我们试图打印找到的单词时，最有意思的部分就来了。看起来下面的代码是显然的方法

```
// 错误：将逆序输出单词的字符
cout << string(line.crbegin(), rcomma) << endl;
```

409> 但它会生成错误的输出结果。例如，如果我们的输入是

FIRST,MIDDLE,LAST

则这条语句会打印 `TSAL!`

图 10.2 说明了问题所在：我们使用的是反向迭代器，会反向处理 `string`。因此，上述输出语句从 `crbegin` 开始反向打印 `line` 中内容。而我们希望按正常顺序打印从 `rcomma` 开始到 `line` 末尾间的字符。但是，我们不能直接使用 `rcomma`。因为它是一个反向迭代器，意味着它会反向朝着 `string` 的开始位置移动。需要做的是，将 `rcomma` 转换回一个普通迭代器，能在 `line` 中正向移动。我们通过调用 `reverse_iterator` 的 `base` 成员函数来完成这一转换，此成员函数会返回其对应的普通迭代器：

```
// 正确：得到一个正向迭代器，从逗号开始读取字符直到 line 末尾
cout << string(rcomma.base(), line.cend()) << endl;
```

给定和之前一样的输入，这条语句会如我们的预期打印出 `LAST`。



图 10.2：反向迭代器和普通迭代器间的关系

图 10.2 中的对象显示了普通迭代器与反向迭代器之间的关系。例如，`rcomma` 和 `rcomma.base()` 指向不同的元素，`line.crbegin` 和 `line.cend()` 也是如此。这些不同保证了元素范围无论是正向处理还是反向处理都是相同的。

从技术上讲，普通迭代器与反向迭代器的关系反映了左闭合区间（参见 9.2.1 节，第 296 页）的特性。关键点在于 [line.cbegin(), rcomma] 和 [rcomma.base(), line.cend()] 指向 line 中相同的元素范围。为了实现这一点，rcomma 和 rcomma.base() 必须生成相邻位置而不是相同位置，cbegin() 和 cend() 也是如此。



反向迭代器的目的是表示元素范围，而这些范围是不对称的，这导致一个重要的结果：当我们从一个普通迭代器初始化一个反向迭代器，或是给一个反向迭代器赋值时，结果迭代器与原迭代器指向的并不是相同的元素。

10.4.3 节练习

410

练习 10.34： 使用 reverse_iterator 逆序打印一个 vector。

练习 10.35： 使用普通迭代器逆序打印一个 vector。

练习 10.36： 使用 find 在一个 int 的 list 中查找最后一个值为 0 的元素。

练习 10.37： 给定一个包含 10 个元素的 vector，将位置 3 到 7 之间的元素按逆序拷贝到一个 list 中。

10.5 泛型算法结构



任何算法最基本的特性是它要求其迭代器提供哪些操作。某些算法，如 find，只要求通过迭代器访问元素、递增迭代器以及比较两个迭代器是否相等这些能力。其他一些算法，如 sort，还要求读、写和随机访问元素的能力。算法所要求的迭代器操作可以分为 5 个迭代器类别 (iterator category)，如表 10.5 所示。每个算法都会对它的每个迭代器参数指明须提供哪类迭代器。

表 10.5：迭代器类别

输入迭代器	只读，不写；单遍扫描，只能递增
输出迭代器	只写，不读；单遍扫描，只能递增
前向迭代器	可读写；多遍扫描，只能递增
双向迭代器	可读写；多遍扫描，可递增递减
随机访问迭代器	可读写，多遍扫描，支持全部迭代器运算

第二种算法分类的方式（如我们在本章开始所做的）是按照是否读、写或是重排序列中的元素来分类。附录 A 按这种分类方法列出了所有算法。

算法还共享一组参数传递规范和一组命名规范，我们在介绍迭代器类别之后将介绍这些内容。

10.5.1 5 类迭代器



类似容器，迭代器也定义了一组公共操作。一些操作所有迭代器都支持，另外一些只有特定类别的迭代器才支持。例如，ostream_iterator 只支持递增、解引用和赋值。vector、string 和 deque 的迭代器除了这些操作外，还支持递减、关系和算术运算。

迭代器是按它们所提供的操作来分类的，而这种分类形成了一种层次。除了输出迭代

器之外，一个高层类别的迭代器支持低层类别迭代器的所有操作。

C++ 标准指明了泛型和数值算法的每个迭代器参数的最小类别。例如，`find` 算法在一个序列上进行一遍扫描，对元素进行只读操作，因此至少需要输入迭代器。`replace` 函数需要一对迭代器，至少是前向迭代器。类似的，`replace_copy` 的前两个迭代器参数也要求至少是前向迭代器。其第三个迭代器表示目的位置，必须至少是输出迭代器。其他的例子类似。对每个迭代器参数来说，其能力必须与规定的最小类别至少相当。向算法传递一个能力更差的迭代器会产生错误。



对于向一个算法传递错误类别的迭代器的问题，很多编译器不会给出任何警告或提示。

迭代器类别

输入迭代器 (input iterator): 可以读取序列中的元素。一个输入迭代器必须支持

- 用于比较两个迭代器的相等和不相等运算符 (`==`、`!=`)
- 用于推进迭代器的前置和后置递增运算 (`++`)
- 用于读取元素的解引用运算符 (`*`)；解引用只会出现在赋值运算符的右侧
- 箭头运算符 (`->`)，等价于 `(*it).member`，即，解引用迭代器，并提取对象的成员

输入迭代器只用于顺序访问。对于一个输入迭代器，`*it++` 保证是有效的，但递增它可能导致所有其他指向流的迭代器失效。其结果就是，不能保证输入迭代器的状态可以保存下来并用来访问元素。因此，输入迭代器只能用于单遍扫描算法。算法 `find` 和 `accumulate` 要求输入迭代器；而 `istream_iterator` 是一种输入迭代器。

输出迭代器 (output iterator): 可以看作输入迭代器功能上的补集——只写而不读元素。输出迭代器必须支持

- 用于推进迭代器的前置和后置递增运算 (`++`)
- 解引用运算符 (`*`)，只出现在赋值运算符的左侧（向一个已经解引用的输出迭代器赋值，就是将值写入它所指向的元素）

我们只能向一个输出迭代器赋值一次。类似输入迭代器，输出迭代器只能用于单遍扫描算法。用作目的位置的迭代器通常都是输出迭代器。例如，`copy` 函数的第三个参数就是输出迭代器。`ostream_iterator` 类型也是输出迭代器。

前向迭代器 (forward iterator): 可以读写元素。这类迭代器只能在序列中沿一个方向移动。前向迭代器支持所有输入和输出迭代器的操作，而且可以多次读写同一个元素。因此，我们可以保存前向迭代器的状态，使用前向迭代器的算法可以对序列进行多遍扫描。算法 `replace` 要求前向迭代器，`forward_list` 上的迭代器是前向迭代器。

双向迭代器 (bidirectional iterator): 可以正向/反向读写序列中的元素。除了支持所有前向迭代器的操作之外，双向迭代器还支持前置和后置递减运算符 (`--`)。算法 `reverse` 要求双向迭代器，除了 `forward_list` 之外，其他标准库都提供符合双向迭代器要求的迭代器。

随机访问迭代器 (random-access iterator): 提供在常量时间内访问序列中任意元素的能力。此类迭代器支持双向迭代器的所有功能，此外还支持表 3.7 (第 99 页) 中的操作：

- 用于比较两个迭代器相对位置的关系运算符 (<、<=、>和>=)
- 迭代器和一个整数值的加减运算 (+、+=、-和-=)，计算结果是迭代器在序列中前进（或后退）给定整数个元素后的位置
- 用于两个迭代器上的减法运算符 (-)，得到两个迭代器的距离
- 下标运算符 (iter[n])，与*(iter[n])等价

算法 `sort` 要求随机访问迭代器。`array`、`deque`、`string` 和 `vector` 的迭代器都是随机访问迭代器，用于访问内置数组元素的指针也是。

10.5.1 节练习

练习 10.38：列出 5 个迭代器类别，以及每类迭代器所支持的操作。

练习 10.39：`list` 上的迭代器属于哪类？`vector` 呢？

练习 10.40：你认为 `copy` 要求哪类迭代器？`reverse` 和 `unique` 呢？

10.5.2 算法形参模式



在任何其他算法分类之上，还有一组参数规范。理解这些参数规范对学习新算法很有帮助——通过理解参数的含义，你可以将注意力集中在算法所做的操作上。大多数算法具有如下 4 种形式之一：

```
alg(beg, end, other args);  
alg(beg, end, dest, other args);  
alg(beg, end, beg2, other args);  
alg(beg, end, beg2, end2, other args);
```

其中 `alg` 是算法的名字，`beg` 和 `end` 表示算法所操作的输入范围。几乎所有算法都接受一个输入范围，是否有其他参数依赖于要执行的操作。这里列出了常见的一种——`dest`、`beg2` 和 `end2`，都是迭代器参数。顾名思义，如果用到了这些迭代器参数，它们分别承担指定目的位置和第二个范围的角色。除了这些迭代器参数，一些算法还接受额外的、非迭代器的特定参数。

413

接受单个目标迭代器的算法

`dest` 参数是一个表示算法可以写入的目的位置的迭代器。算法假定 (assume)：按其需要写入数据，不管写入多少个元素都是安全的。



WARNING

向输出迭代器写入数据的算法都假定目标空间足够容纳写入的数据。

如果 `dest` 是一个直接指向容器的迭代器，那么算法将输出数据写到容器中已存在的元素内。更常见的原因是，`dest` 被绑定到一个插入迭代器（参见 10.4.1 节，第 358 页）或是一个 `ostream_iterator`（参见 10.4.2 节，第 359 页）。插入迭代器会将新元素添加到容器中，因而保证空间是足够的。`ostream_iterator` 会将数据写入到一个输出流，同样不管要写入多少个元素都没有问题。

接受第二个输入序列的算法

接受单独的 `beg2` 或是接受 `beg2` 和 `end2` 的算法用这些迭代器表示第二个输入范围。这些算法通常使用第二个范围中的元素与第一个输入范围结合来进行一些运算。

如果一个算法接受 `beg2` 和 `end2`, 这两个迭代器表示第二个范围。这类算法接受两个完整指定的范围: `[beg, end)` 表示的范围和 `[beg2 end2)` 表示的第二个范围。

只接受单独的 `beg2`(不接受 `end2`)的算法将 `beg2` 作为第二个输入范围中的首元素。此范围的结束位置未指定, 这些算法假定从 `beg2` 开始的范围与 `beg` 和 `end` 所表示的范围至少一样大。



接受单独 `beg2` 的算法假定从 `beg2` 开始的序列与 `beg` 和 `end` 所表示的范围至少一样大。



10.5.3 算法命名规范

除了参数规范, 算法还遵循一套命名和重载规范。这些规范处理诸如: 如何提供一个操作代替默认的`<`或`==`运算符以及算法是将输出数据写入输入序列还是一个分离的目的位置等问题。

一些算法使用重载形式传递一个谓词

414 接受谓词参数来代替`<`或`==`运算符的算法, 以及那些不接受额外参数的算法, 通常都是重载的函数。函数的一个版本用元素类型的运算符来比较元素; 另一个版本接受一个额外谓词参数, 来代替`<`或`==`:

```
unique(beg, end);           // 使用 == 运算符比较元素
unique(beg, end, comp);    // 使用 comp 比较元素
```

两个调用都重新整理给定序列, 将相邻的重复元素删除。第一个调用使用元素类型的`==`运算符来检查重复元素; 第二个则调用 `comp` 来确定两个元素是否相等。由于两个版本的函数在参数个数上不相等, 因此具体应该调用哪个版本不会产生歧义(参见 6.4 节, 第 208 页)。

_if 版本的算法

接受一个元素值的算法通常有另一个不同名的(不是重载的)版本, 该版本接受一个谓词(参见 10.3.1 节, 第 344 页)代替元素值。接受谓词参数的算法都有附加的 `_if` 前缀:

```
find(beg, end, val);        // 查找输入范围内 val 第一次出现的位置
find_if(beg, end, pred);   // 查找第一个令 pred 为真的元素
```

这两个算法都在输入范围内查找特定元素第一次出现的位置。算法 `find` 查找一个指定值; 算法 `find_if` 查找使得 `pred` 返回非零值的元素。

这两个算法提供了命名上差异的版本, 而非重载版本, 因为两个版本的算法都接受相同数目的参数。因此可能产生重载歧义, 虽然很罕见, 但为了避免任何可能的歧义, 标准库选择提供不同名字的版本而不是重载。

区分拷贝元素的版本和不拷贝的版本

默认情况下, 重排元素的算法将重排后的元素写回给定的输入序列中。这些算法还提供另一个版本, 将元素写到一个指定的输出目的位置。如我们所见, 写到额外目的空间的

算法都在名字后面附加一个_copy (参见 10.2.2 节, 第 341 页):

```
reverse(beg, end);           // 反转输入范围中元素的顺序
reverse_copy(beg, end, dest); // 将元素按逆序拷贝到 dest
```

一些算法同时提供_copy 和_if 版本。这些版本接受一个目的位置迭代器和一个谓词:

```
// 从 v1 中删除奇数元素
remove_if(v1.begin(), v1.end(),
           [](int i) { return i % 2; });

// 将偶数元素从 v1 拷贝到 v2; v1 不变
remove_copy_if(v1.begin(), v1.end(), back_inserter(v2),
               [](int i) { return i % 2; });
```

两个算法都调用了 lambda (参见 10.3.2 节, 第 346 页) 来确定元素是否为奇数。在第一个调用中, 我们从输入序列中将奇数元素删除。在第二个调用中, 我们将非奇数 (亦即偶数) 元素从输入范围拷贝到 v2 中。

10.5.3 节练习

415

练习 10.41: 仅根据算法和参数的名字, 描述下面每个标准库算法执行什么操作:

```
replace(beg, end, old_val, new_val);
replace_if(beg, end, pred, new_val);
replace_copy(beg, end, dest, old_val, new_val);
replace_copy_if(beg, end, dest, pred, new_val);
```

10.6 特定容器算法

与其他容器不同, 链表类型 `list` 和 `forward_list` 定义了几个成员函数形式的算法, 如表 10.6 所示。特别是, 它们定义了独有的 `sort`、`merge`、`remove`、`reverse` 和 `unique`。通用版本的 `sort` 要求随机访问迭代器, 因此不能用于 `list` 和 `forward_list`, 因为这两个类型分别提供双向迭代器和前向迭代器。

链表类型定义的其他算法的通用版本可以用于链表, 但代价太高。这些算法需要交换输入序列中的元素。一个链表可以通过改变元素间的链接而不是真的交换它们的值来快速“交换”元素。因此, 这些链表版本的算法的性能比对应的通用版本好得多。



对于 `list` 和 `forward_list`, 应该优先使用成员函数版本的算法而不是通用算法。

表 10.6: `list` 和 `forward_list` 成员函数版本的算法

这些操作都返回 `void`

<code>lst.merge(lst2)</code>	将来自 <code>lst2</code> 的元素合并入 <code>lst</code> 。 <code>lst</code> 和 <code>lst2</code> 都必须是有序的。
<code>lst.merge(lst2, comp)</code>	元素将从 <code>lst2</code> 中删除。在合并之后, <code>lst2</code> 变为空。第一个版本使用<运算符; 第二个版本使用给定的比较操作
<code>lst.remove(val)</code>	调用 <code>erase</code> 删除掉与给定值相等 (<code>==</code>) 或令一元谓词为真的每个元素
<code>lst.remove_if(pred)</code>	
<code>lst.reverse()</code>	反转 <code>lst</code> 中元素的顺序

续表

<code>lst.sort()</code>	使用<或给定比较操作排序元素
<code>lst.sort(comp)</code>	
<code>lst.unique()</code>	调用 <code>erase</code> 删除同一个值的连续拷贝。第一个版本使用==；第二个版本使用给定的二元谓词
<code>lst.unique(pred)</code>	

📚 splice 成员

416 链表类型还定义了 `splice` 算法，其描述见表 10.7。此算法是链表数据结构所特有的，因此不需要通用版本。

表 10.7: `list` 和 `forward_list` 的 `splice` 成员函数的参数

<code>lst.splice(args)</code> 或 <code>f1st.splice_after(args)</code>	
<code>(p, lst2)</code>	<code>p</code> 是一个指向 <code>lst</code> 中元素的迭代器，或一个指向 <code>f1st</code> 首前位置的迭代器。函数将 <code>lst2</code> 的所有元素移动到 <code>lst</code> 中 <code>p</code> 之前的位置或是 <code>f1st</code> 中 <code>p</code> 之后的位置。将元素从 <code>lst2</code> 中删除。 <code>lst2</code> 的类型必须与 <code>lst</code> 或 <code>f1st</code> 相同，且不能是同一个链表
<code>(p, lst2, p2)</code>	<code>p2</code> 是一个指向 <code>lst2</code> 中位置的有效迭代器。将 <code>p2</code> 指向的元素移动到 <code>lst</code> 中，或将 <code>p2</code> 之后的元素移动到 <code>f1st</code> 中。 <code>lst2</code> 可以是与 <code>lst</code> 或 <code>f1st</code> 相同的链表
<code>(p, lst2, b, e)</code>	<code>b</code> 和 <code>e</code> 必须表示 <code>lst2</code> 中的合法范围。将给定范围中的元素从 <code>lst2</code> 移动到 <code>lst</code> 或 <code>f1st</code> 。 <code>lst2</code> 与 <code>lst</code> (或 <code>f1st</code>) 可以是相同的链表，但 <code>p</code> 不能指向给定范围内元素

链表特有的操作会改变容器

多数链表特有的算法都与其通用版本很相似，但不完全相同。链表特有版本与通用版本间的一个至关重要的区别是链表版本会改变底层的容器。例如，`remove` 的链表版本会删除指定的元素。`unique` 的链表版本会删除第二个和后继的重复元素。

类似的，`merge` 和 `splice` 会销毁其参数。例如，通用版本的 `merge` 将合并的序列写到一个给定的目的迭代器；两个输入序列是不变的。而链表版本的 `merge` 函数会销毁给定的链表——元素从参数指定的链表中删除，被合并到调用 `merge` 的链表对象中。在 `merge` 之后，来自两个链表中的元素仍然存在，但它们都已在同一个链表中。

10.6 节练习

练习 10.42：使用 `list` 代替 `vector` 重新实现 10.2.3 节（第 343 页）中的去除重复单词的程序。

小结

< 417

标准库定义了大约 100 个类型无关的对序列进行操作的算法。序列可以是标准库容器类型中的元素、一个内置数组或者是（例如）通过读写一个流来生成的。算法通过在迭代器上进行操作来实现类型无关。多数算法接受的前两个参数是一对迭代器，表示一个元素范围。额外的迭代器参数可能包括一个表示目的位置的输出迭代器，或是表示第二个输入范围的另一个或另一对迭代器。

根据支持的操作不同，迭代器可分为五类：输入、输出、前向、双向以及随机访问迭代器。如果一个迭代器支持某个迭代器类别所要求的操作，则属于该类别。

如同迭代器根据操作分类一样，传递给算法的迭代器参数也按照所要求的操作进行分类。仅读取序列的算法只要求输入迭代器操作。写入数据到目的位置迭代器的算法只要求输出迭代器操作，依此类推。

算法从不直接改变它们所操作的序列的大小。它们会将元素从一个位置拷贝到另一个位置，但不会直接添加或删除元素。

虽然算法不能向序列添加元素，但插入迭代器可以做到。一个插入迭代器被绑定到一个容器上。当我们将一个容器元素类型的值赋予一个插入迭代器时，迭代器会将该值添加到容器中。

容器 `forward_list` 和 `list` 对一些通用算法定义了自己特有的版本。与通用算法不同，这些链表特有版本会修改给定的链表。

术语表

back_inserter 这是一个迭代器适配器，它接受一个指向容器的引用，生成一个插入迭代器，该插入迭代器用 `push_back` 向指定容器添加元素。

双向迭代器（bidirectional iterator） 支持前向迭代器的所有操作，还具有用`--`在序列中反向移动的能力。

二元谓词（binary predicate） 接受两个参数的谓词。

bind 标准库函数，将一个或多个参数绑定到一个可调用表达式。`bind` 定义在头文件 `functional` 中。

可调用对象（callable object） 可以出现在调用运算符左边的对象。函数指针、`lambda` 以及重载了函数调用运算符的类的对象都是可调用对象。

捕获列表（capture list） `lambda` 表达式的

一部分，指出 `lambda` 表达式可以访问所在上下文中哪些变量。

cref 标准库函数，返回一个可拷贝的对象，其中保存了一个指向不可拷贝类型的 `const` 对象的引用。

前向迭代器（forward iterator） 可以读写元素，但不必支持`--`的迭代器。

front_inserter 迭代器适配器，给定一个容器，生成一个用 `push_front` 向容器开始位置添加元素的插入迭代器。

泛型算法（generic algorithm） 类型无关的算法。

输入迭代器（input iterator） 可以读但不能写序列中元素的迭代器。

插入迭代器（insert iterator） 迭代器适配器，生成一个迭代器，该迭代器使用容器操作向给定容器添加元素。

< 418

插入器 (insertter) 迭代器适配器，接受一个迭代器和一个指向容器的引用，生成一个插入迭代器，该插入迭代器用 `insert` 在给定迭代器指向的元素之前的位置添加元素。

istream_iterator 读取输入流的流迭代器。

迭代器类别 (iterator category) 根据所支持的操作对迭代器进行的分类组织。迭代器类别形成一个层次，其中更强大的类别支持更弱类别的所有操作。算法使用迭代器类别来指出迭代器参数必须支持哪些操作。只要迭代器达到所要求的最小类别，它就可以用于算法。例如，一些算法只要求输入迭代器。这类算法可处理除只满足输出迭代器要求的迭代器之外的任何迭代器。而要求随机访问迭代器的算法只能用于支持随机访问操作的迭代器。

lambda 表达式 (lambda expression) 可调用的代码单元。一个 `lambda` 类似一个未命名的内联函数。一个 `lambda` 以一个捕获列表开始，此列表允许 `lambda` 访问所在函数中的变量。类似函数，`lambda` 有一个（可能为空的）参数列表、一个返回类型和一个函数体。`lambda` 可以忽略返回类型。如果函数体是一个单一的 `return` 语句，返回类型就从返回对象的类型推断。否则，忽略的返回类型默认为 `void`。

移动迭代器 (move iterator) 迭代器适配器，生成一个迭代器，该迭代器移动而不是拷贝元素。移动迭代器将在第 13 章中进行介绍。

ostream_iterator 写输出流的迭代器。

输出迭代器 (output iterator) 可以写元素，但不必具有读元素能力的迭代器。

谓词 (predicate) 返回可以转换为 `bool` 类型的值的函数。泛型算法通常用来检测元素。标准库使用的谓词是一元（接受一个参数）或二元（接受两个参数）的。

随机访问迭代器 (random-access iterator) 支持双向迭代器的所有操作再加上比较迭代器值的关系运算符、下标运算符和迭代器上的算术运算，因此支持随机访问元素。

ref 标准库函数，从一个指向不能拷贝的类型的对象的引用生成一个可拷贝的对象。

反向迭代器 (reverse iterator) 在序列中反向移动的迭代器。这些迭代器交换了++ 和 -- 的含义。

流迭代器 (stream iterator) 可以绑定到一个流的迭代器。

一元谓词 (unary predicate) 接受一个参数的谓词。