

```
// 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

<code>s.insert(pos,args)</code>	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
<code>s.erase(pos,len)</code>	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
<code>s.assign(args)</code>	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
<code>s.append(args)</code>	将 args 追加到 s。返回一个指向 s 的引用
<code>s.replace(range,args)</code>	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
<i>args</i> 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。	
<code>str</code>	字符串 str
<code>str, pos, len</code>	str 中从 pos 开始最多 len 个字
<code>cp, len</code>	从 cp 指向的字符数组的前(最多) len 个字符
<code>cp</code>	cp 指向的以空字符结尾的字符数组
<code>n, c</code>	n 个字符 c
<code>b, e</code>	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。

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）。

< 365

`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
```

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`。

<code>s.find(args)</code>	查找 <code>s</code> 中 <code>args</code> 第一次出现的位置
<code>s.rfind(args)</code>	查找 <code>s</code> 中 <code>args</code> 最后一次出现的位置
<code>s.find_first_of(args)</code>	在 <code>s</code> 中查找 <code>args</code> 中任何一个字符第一次出现的位置。
<code>s.find_last_of(args)</code>	在 <code>s</code> 中查找 <code>args</code> 中任何一个字符最后一次出现的位置
<code>s.find_first_not_of(args)</code>	在 <code>s</code> 中查找第一个不在 <code>args</code> 中的字符
<code>s.find_last_not_of(args)</code>	在 <code>s</code> 中查找最后一个不在 <code>args</code> 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

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



```
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

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

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

<code>size_type</code>	一种类型，足以保存当前类型的最大对象的大小
<code>value_type</code>	元素类型
<code>container_type</code>	实现适配器的底层容器类型
<code>A a;</code>	创建一个名为 a 的空适配器
<code>A a(c);</code>	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： <code>==</code> 、 <code>!=</code> 、 <code><</code> 、 <code><=</code> 、 <code>></code> 和 <code>>=</code> 这些运算符返回底层容器的比较结果
<code>a.empty()</code>	若 a 包含任何元素，返回 <code>false</code> ，否则返回 <code>true</code>
<code>a.size()</code>	返回 a 中的元素数目
<code>swap(a,b)</code>	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
<code>a.swap(b)</code>	

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 <code>deque</code> 实现，也可以在 <code>list</code> 或 <code>vector</code> 之上实现。	
<code>s.pop()</code>	删除栈顶元素，但不返回该元素值
<code>s.push(item)</code>	创建一个新元素压入栈顶，该元素通过拷贝或移动 <code>item</code> 而来，或者由 <code>args</code> 构造
<code>s.emplace(args)</code>	由 <code>args</code> 构造
<code>s.top()</code>	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

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

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， 其值为 item，或者由 args 构造
q.emplace(args)	

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。

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

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（*insertter*）。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`



`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。

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

$$[\text{capture list}] (\text{parameter list}) \rightarrow \text{return type} \{ \text{function body} \}$$

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。

建议：尽量保持 `lambda` 的变量捕获简单化

< 394

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

C++ 11 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。

404> 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

当我们将一个 `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;
```

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：

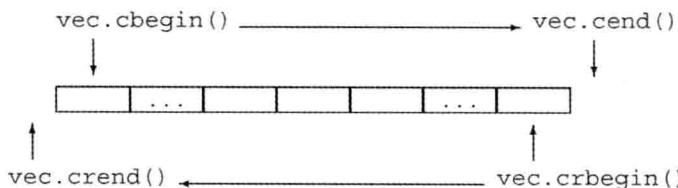


图 10.1：比较 `cbegin/cend` 和 `crbegin/crend`

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`。

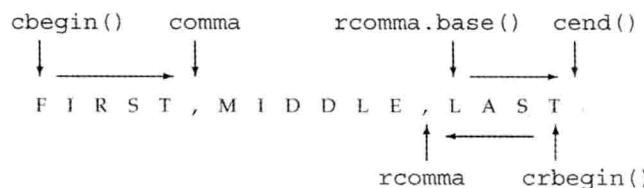


图 10.2：反向迭代器和普通迭代器间的关系

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。



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。

WARNING

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。更常见的情况是，`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

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);           // 反转输入范围内元素的顺序
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 (inserter) 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。

第 11 章

关联容器

内容

11.1 使用关联容器.....	374
11.2 关联容器概述.....	376
11.3 关联容器操作.....	381
11.4 无序容器.....	394
小结	397
术语表.....	397

关联容器和顺序容器有着根本的不同：关联容器中的元素是按关键字来保存和访问的。与之相对，顺序容器中的元素是按它们在容器中的位置来顺序保存和访问的。

虽然关联容器的很多行为与顺序容器相同，但其不同之处反映了关键字的作用。

420

关联容器支持高效的关键字查找和访问。两个主要的关联容器 (associative-container) 类型是 **map** 和 **set**。**map** 中的元素是一些关键字-值 (key-value) 对：关键字起到索引的作用，值则表示与索引相关联的数据。**set** 中每个元素只包含一个关键字；**set** 支持高效的关键字查询操作——检查一个给定关键字是否在 **set** 中。例如，在某些文本处理过程中，可以用一个 **set** 来保存想要忽略的单词。字典则是一个很好的使用 **map** 的例子：可以将单词作为关键字，将单词释义作为值。

标准库提供 8 个关联容器，如表 11.1 所示。这 8 个容器的不同体现在三个维度上：每个容器（1）或者是一个 **set**，或者是一个 **map**；（2）或者要求不重复的关键字，或者允许重复关键字；（3）按顺序保存元素，或无序保存。允许重复关键字的容器的名字中都包含单词 **multi**；不保持关键字按顺序存储的容器的名字都以单词 **unordered** 开头。因此一个 **unordered_multi_set** 是一个允许重复关键字，元素无序保存的集合，而一个 **set** 则是一个要求不重复关键字，有序存储的集合。无序容器使用哈希函数来组织元素，我们将在 11.4 节（第 394 页）中详细介绍有关哈希函数的更多内容。

类型 **map** 和 **multimap** 定义在头文件 **map** 中；**set** 和 **multiset** 定义在头文件 **set** 中；无序容器则定义在头文件 **unordered_map** 和 **unordered_set** 中。

表 11.1：关联容器类型

按关键字有序保存元素	
map	关联数组：保存关键字-值对
set	关键字即值，即只保存关键字的容器
multimap	关键字可重复出现的 map
multiset	关键字可重复出现的 set
无序集合	
unordered_map	用哈希函数组织的 map
unordered_set	用哈希函数组织的 set
unordered_multimap	哈希组织的 map ：关键字可以重复出现
unordered_multiset	哈希组织的 set ：关键字可以重复出现



11.1 使用关联容器

虽然大多数程序员都熟悉诸如 **vector** 和 **list** 这样的数据结构，但他们中很多人从未使用过关联数据结构。在学习标准库关联容器类型的详细内容之前，我们首先来看一个如何使用这类容器的例子，这对后续学习很有帮助。

map 是关键字-值对的集合。例如，可以将一个人的名字作为关键字，将其电话号码作为值。我们称这样的数据结构为“将名字映射到电话号码”。**map** 类型通常被称为**关联数组 (associative array)**。关联数组与“正常”数组类似，不同之处在于其下标不必是整数。

421

我们通过一个关键字而不是位置来查找值。给定一个名字到电话号码的 **map**，我们可以使用一个人的名字作为下标来获取此人的电话号码。

与之相对，**set** 就是关键字的简单集合。当只是想知道一个值是否存在时，**set** 是最有用的。例如，一个企业可以定义一个名为 **bad_checks** 的 **set** 来保存那些曾经开过空头支票的人的名字。在接受一张支票之前，可以查询 **bad_checks** 来检查顾客的名字是否在其中。

使用 map

一个经典的使用关联数组的例子是单词计数程序：

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

此程序读取输入，报告每个单词出现多少次。

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

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

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

如果我们对本节第一段中的文本（指英文版中的文本）运行这个程序，输出将会是：

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

使用 set

422

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

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

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

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

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

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

如果用此程序处理与之前相同的输入，输出将会是：

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

11.1 节练习

练习 11.1：描述 `map` 和 `vector` 的不同。

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

练习 11.3：编写你自己的单词计数程序。

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

423 11.2 关联容器概述

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

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

关联容器的迭代器都是双向的（参见 10.5.1 节，第 365 页）。



11.2.1 定义关联容器

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

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

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

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

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

```
{key, value}
```

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

初始化 `multimap` 或 `multiset`

一个 `map` 或 `set` 中的关键字必须是唯一的，即，对于一个给定的关键字，只能有一个元素的关键字等于它。容器 `multimap` 和 `multiset` 没有此限制，它们都允许多个元素具有相同的关键字。例如，在我们用来统计单词数量的 `map` 中，每个单词只能有一个元素。另一方面，在一个词典中，一个特定单词则可具有多个与之关联的词义。 424

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

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

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

11.2.1 节练习

练习 11.5：解释 map 和 set 的区别。你如何选择使用哪个？

练习 11.6：解释 set 和 list 的区别。你如何选择使用哪个？

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

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



11.2.2 关键字类型的要求

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



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

有序容器的关键字类型

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

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

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



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

使用关键字类型的比较函数

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

义哪种类型的容器，自定义的操作类型必须在尖括号中紧跟着元素类型给出。

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

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

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

426

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

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

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

11.2.2 节练习

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

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

练习 11.11： 不使用 `decltype` 重新定义 `bookstore`。

11.2.3 pair 类型

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

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

```

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

```

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

我们也可以为每个成员提供初始化器：

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

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

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

```

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

```

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

表 11.2: pair 上的操作

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

创建 pair 对象的函数

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

```

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

```

```

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

```

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

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

```

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

```

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

```

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

```

11.2.3 节练习

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

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

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

11.3 关联容器操作

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

表 11.3：关联容器额外的类型别名

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

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

```

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

```

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

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

11.3.1 关联容器迭代器

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

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



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

set 的迭代器是 `const` 的

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

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

430 遍历关联容器

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

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

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

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



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

关联容器和算法

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

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

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

11.3.1 节练习

< 431

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

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

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

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

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

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

11.3.2 添加元素

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

存在的元素对容器没有任何影响：

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

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

向 map 添加元素

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

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

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

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

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

表 11.4：关联容器 `insert` 操作

<code>c.insert(v)</code>	<code>v</code> 是 <code>value_type</code> 类型的对象； <code>args</code> 用来构造一个元素
<code>c.emplace(args)</code>	对于 <code>map</code> 和 <code>set</code> ，只有当元素的关键字不在 <code>c</code> 中时才插入（或构造）元素。函数返回一个 <code>pair</code> ，包含一个迭代器，指向具有指定关键字的元素，以及一个指示插入是否成功的 <code>bool</code> 值。 对于 <code>multimap</code> 和 <code>multiset</code> ，总会插入（或构造）给定元素，并返回一个指向新元素的迭代器
<code>c.insert(b, e)</code>	<code>b</code> 和 <code>e</code> 是迭代器，表示一个 <code>c::value_type</code> 类型值的范围； <code>i1</code> 是这种值的花括号列表。函数返回 <code>void</code>
<code>c.insert(i1)</code>	对于 <code>map</code> 和 <code>set</code> ，只插入关键字不在 <code>c</code> 中的元素。对于 <code>multimap</code> 和 <code>multiset</code> ，则会插入范围中的每个元素
<code>c.insert(p, v)</code>	类似 <code>insert(v)</code> （或 <code>emplace(args)</code> ），但将迭代器 <code>p</code> 作为一个提示，指出从哪里开始搜索新元素应该存储的位置。返回一个迭代器，指向具有给定关键字的元素
<code>c.emplace(p, args)</code>	

检测 `insert` 的返回值

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

素被插入容器中，且 bool 值为 true。

作为一个例子，我们用 insert 重写单词计数程序：

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

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

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

展开递增语句

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

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

下面我们一步一步来解释此表达式：

ret 保存 insert 返回的值，是一个 pair。

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

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

ret.first->second map 中元素的值部分。

++ret.first->second 递增此值。

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

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

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

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

向 multiset 或 multimap 添加元素

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

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

434 >

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

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

11.3.2 节练习

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

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

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

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

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

11.3.3 删除元素

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

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

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

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

435 > 对允许重复关键字的容器，删除元素的数量可能大于 1：

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

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

表 11.5: 从关联容器删除元素

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

11.3.4 map 的下标操作



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

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

例如，如果我们编写如下代码

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

将会执行如下操作：

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

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

436



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

表 11.6: `map` 和 `unordered_map` 的下标操作

<code>c[k]</code>	返回关键字为 k 的元素；如果 k 不在 c 中，添加一个关键字为 k 的元素，对其进行值初始化
<code>c.at(k)</code>	访问关键字为 k 的元素，带参数检查；若 k 不在 c 中，抛出一个 <code>out_of_range</code> 异常（参见 5.6 节，第 173 页）

使用下标操作的返回值

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

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

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

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



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

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

11.3.4 节练习

练习 11.24：下面的程序完成什么功能？

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

练习 11.25：对比下面程序与上一题程序

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

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

11.3.5 访问元素

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

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

表 11.7：在一个关联容器中查找元素的操作

lower_bound 和 upper_bound 不适用于无序容器。

下标和 at 操作只适用于非 const 的 map 和 unordered_map。

续表

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

对 map 使用 find 代替下标操作

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

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

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

在 multimap 或 multiset 中查找元素

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

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

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

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



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

一种不同的，面向迭代器的解决方法

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

439 >

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



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

使用这两个操作，我们可以重写前面的程序：

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

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

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

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

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



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

equal_range 函数

解决此问题的最后一一种方法是三种方法中最直接的：不必再调用 `upper_bound` 和

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

<440

可以用 `equal_range` 来再次修改我们的程序:

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

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

11.3.5 节练习

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

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

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

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

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

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

11.3.6 一个单词转换的 map

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

如果单词转换文件的内容如下所示:

<441

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

```

u you
pic picture
thk thanks!
18r later

```

我们希望转换的文本为

```

where r u
y dont u send me a pic
k thk 18r

```

则程序应该生成这样的输出：

```

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

```

单词转换程序

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

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

```

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

```

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

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

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

建立转换映射

函数 `buildMap` 读入给定文件, 建立起转换映射。

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

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

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

生成转换文本

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

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

< 443

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

用来替代 s 的内容。

11.3.6 节练习

练习 11.33: 实现你自己版本的单词转换程序。

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

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

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

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



11.4 无序容器

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

C++
11

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

444 >



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

使用无序容器

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

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

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

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

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

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

对于每个单词，我们将得到相同的计数结果。但单词不太可能按字典序输出。

管理桶

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

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

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

<445

表 11.8：无序容器管理操作

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

无序容器对关键字类型的要求

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

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

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

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

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

我们使用这些函数来定义一个`unordered_multiset`

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

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

如果我们的类定义了`==`运算符，则可以只重载哈希函数：

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

11.4 节练习

练习 11.37：一个无序容器与其有序版本相比有何优势？有序版本有何优势？

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

小结

447

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

标准库定义了 8 个关联容器，每个容器

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

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

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

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

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

术语表

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

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

hash 特殊的标准库模板，无序容器用它来管理元素的位置。

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

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

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

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

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

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

448

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

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

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

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

unordered_map 保存关键字-值对的容器，不允许重复关键字。

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

unordered_multiset 保存关键字的容器，

允许重复关键字。

unordered_set 保存关键字的容器，不允许重复关键字。

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

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

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

第 12 章

动态内存

内容

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

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

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

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

450

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

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



虽然使用动态内存有时是必要的，但众所周知，正确地管理动态内存是非常棘手的。

12.1 动态内存与智能指针

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

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

C++ 11

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



12.1.1 shared_ptr 类

C++ 11

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

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

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

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

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

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

操作列于表 12.2 中。

表 12.1: `shared_ptr` 和 `unique_ptr` 都支持的操作

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

表 12.2: `shared_ptr` 独有的操作

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

make_shared 函数

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

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

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

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

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

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

shared_ptr 的拷贝和赋值

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

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

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

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

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

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



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

shared_ptr 自动销毁所管理的对象……

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

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

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

……shared_ptr 还会自动释放相关联的内存

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

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

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

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

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

由于 `p` 是 `use_factory` 的局部变量，在 `use_factory` 结束时它将被销毁（参见 6.1.1 ◀454 节，第 184 页）。当 `p` 被销毁时，将递减其引用计数并检查它是否为 0。在此例中，`p` 是唯一引用 `factory` 返回的内存的对象。由于 `p` 将要销毁，`p` 指向的这个对象也会被销毁，所占用的内存会被释放。

但如果其他 `shared_ptr` 也指向这块内存，它就不会被释放掉：

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

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

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



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

使用了动态生存期的资源的类

程序使用动态内存出于以下三种原因之一：

1. 程序不知道自己需要使用多少对象
2. 程序不知道所需对象的准确类型

3. 程序需要在多个对象间共享数据

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

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

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

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

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

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

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

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



使用动态内存的一个常见原因是允许多个对象共享相同的状态。

定义 `StrBlob` 类

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

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

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

我们将 `vector` 保存在动态内存中。

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

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

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

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

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

StrBlob 构造函数

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

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

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

元素访问成员函数

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

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

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

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

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

string& StrBlob::back()
{
    check(0, "back on empty StrBlob");
    return data->back();
}

void StrBlob::pop_back()
{
    check(0, "pop_back on empty StrBlob");
    data->pop_back();
}
```

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

StrBlob 的拷贝、赋值和销毁

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

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

12.1.1 节练习

练习 12.1：在此代码的结尾，`b1` 和 `b2` 各包含多少个元素？

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

```

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

```

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

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

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

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

12.1.2 直接管理内存

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

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



在学习第 13 章之前，除非使用智能指针来管理内存，否则不要分配动态内存。

使用 new 动态分配和初始化对象

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

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

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

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

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

459

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

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

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

C++
11

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

后跟一对空括号即可：

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

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

Best Practices

出于与变量初始化相同的原因，对动态分配的对象进行初始化通常是个好主意。

C++ 11 如果我们提供了一个括号包围的初始化器，就可以使用 `auto`（参见 2.5.2 节，第 61 页）从此初始化器来推断我们想要分配的对象的类型。但是，由于编译器要用初始化器的类型来推断要分配的类型，只有当括号中仅有单一初始化器时才可以使用 `auto`：

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

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

动态分配的 `const` 对象

用 `new` 分配 `const` 对象是合法的：

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

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

内存耗尽

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

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

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

释放动态内存

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

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

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

指针值和 delete

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

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

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

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

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

动态对象的生存期直到被释放时为止

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

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

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