

下情况：对于大部分应用，使用 vector 容器是最好的。原因在于，标准库的实现者使用这样的内存分配策略：以最小的代价连续存储元素。由此而带来的访问元素的便利弥补了其存储代价。

为了使 vector 容器实现快速的内存分配，其实际分配的容量要比当前所需的空间多一些。vector 容器预留了这些额外的存储区，用于存放新添加的元素。于是，不必为每个新元素重新分配容器。所分配的额外内存容量的确切数目因库的实现不同而不同。比起每添加一个新元素就必须重新分配一次容器，这个分配策略带来显著的效率。事实上，其性能非常好，因此在实际应用中，比起 list 和 deque 容器，vector 的增长效率通常会更高。

capacity 和 reserve 成员

vector 容器处理内存分配的细节是其实现的一部分。然而，该实现部分是由 vector 的接口支持的。vector 类提供了两个成员函数：capacity 和 reserve，使程序员可与 vector 容器内存分配的实现部分交互工作。capacity 操作获取在容器需要分配更多的存储空间之前能够存储的元素总数，而 reserve 操作则告诉 vector 容器应该预留多少个元素的存储空间。



弄清楚容器的 capacity（容量）与 size（长度）的区别非常重要。size 指容器当前拥有的元素个数，而 capacity 则指容器在必须分配新存储空间之前可以存储的元素总数。

为了说明 size 和 capacity 的交互作用，考虑下面的程序：

```
vector<int> ivec;
// size should be zero; capacity is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
// give ivec 24 elements
for (vector<int>::size_type ix = 0; ix != 24; ++ix)
    ivec.push_back(ix);

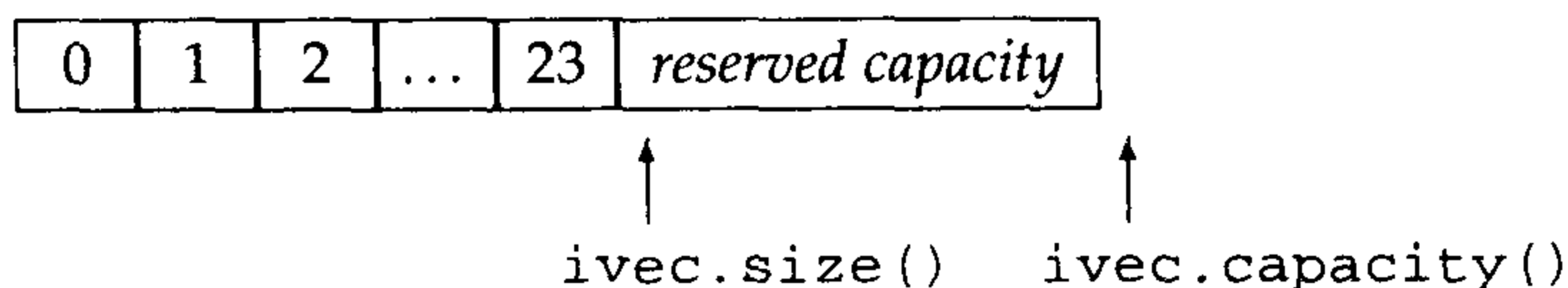
// size should be 24; capacity will be >= 24 and is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

在我们的系统上运行该程序时，得到以下输出结果：

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

331

由此可见，空 vector 容器的 size 是 0，而标准库显然将其 capacity 也设置为 0。当程序员在 vector 中插入元素时，容器的 size 就是所添加的元素个数，而其 capacity 则必须至少等于 size，但通常比 size 值更大。在上述程序中，一次添加一个元素，共添加了 24 个元素，结果其 capacity 为 32。ivec 容器的当前状态如下图所示：



现在，可如下预留额外的存储空间：

```
ivec.reserve(50); // sets capacity to at least 50; might be more
// size should be 24; capacity will be >= 50 and is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

正如下面的输出结果所示，该操作只改变了容器的 capacity，而其 size 不变：

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

下面的程序将预留的容量用完：

```
// add elements to use up the excess capacity
while (ivec.size() != ivec.capacity())
    ivec.push_back(0);
// size should be 50; capacity should be unchanged
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

由于在该程序中，只使用了预留的容量，因此 vector 不必做任何内存分配工作。事实上，只要有剩余的容量，vector 就不必为其元素重新分配存储空间。

其输出结果表明：此时我们已经耗尽了预留的容量，该容器的 size 和 capacity 值相等：

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

此时，如果要添加新的元素，vector 必须为自己重新分配存储空间：

```
ivec.push_back(42); // add one more element
// size should be 51; capacity will be >= 51 and is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

这段程序的输出：

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

332

表明：每当 vector 容器不得不分配新的存储空间时，以加倍当前容量的分配策略实现重新分配。



vector 的每种实现都可自由地选择自己的内存分配策略。然而，它们都必须提供 reserve 和 capacity 函数，而且必须是到必要时才分配新的内存空间。分配多少内存取决于其实现方式。不同的库采用不同的策略实现。

此外，每种实现都要求遵循以下原则：确保 push_back 操作高效地在 vector 中添加元素。从技术上来说，在原来为空的 vector 容器上 n 次调用 push_back 函数，从而创建拥有 n 个元素的 vector 容器，其执行时间永远不能超过 n 的常量倍。

习题

习题 9.29 解释 vector 的容量和长度之间的区别。为什么在连续存储元素的容器中需要支持“容量”的概念？而非连续的容器，如 list，则不需要。

习题 9.30 编写程序研究标准库为 `vector` 对象提供的内存分配策略。

习题 9.31 容器的容量可以比其长度小吗？在初始时或插入元素后，容量是否恰好等于所需要的长度？为什么？

习题 9.32 解释下面程序实现的功能：

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

如果该程序读入了 256 个单词，在调整大小后，该容器的容量可能是多少？如果读入 512，或 1000，或 1048 个单词呢？

9.5 容器的选用

在前面的章节中可见，分配连续存储元素的内存空间会影响内存分配策略和容器对象的开销。通过巧妙的实现技巧，标准库的实现者已经最小化了内存分配的开销。元素是否连续存储还会显著地影响：

- 在容器的中间位置添加或删除元素的代价。
- 执行容器元素的随机访问的代价。

333

程序使用这些操作的程度将决定应该选择哪种类型的容器。`vector` 和 `deque` 容器提供了对元素的快速随机访问，但付出的代价是，在容器的任意位置插入或删除元素，比在容器尾部插入和删除的开销更大。`list` 类型在任何位置都能快速插入和删除，但付出的代价是元素的随机访问开销较大。

1. 插入操作如何影响容器的选择

`list` 容器表示不连续的内存区域，允许向前和向后逐个遍历元素。在任何位置都可高效地 `insert` 或 `erase` 一个元素。插入或删除 `list` 容器中的一个元素不需要移动任何其他元素。另一方面，`list` 容器不支持随机访问，访问某个元素要求遍历所涉及的其他元素。

对于 `vector` 容器，除了容器尾部外，其他任何位置上的插入（或删除）操作都要求移动被插入（或删除）元素右边所有的元素。例如，假设有一个拥有 50 个元素的 `vector` 容器，我们希望删除其中的第 23 号元素，则 23 号元素后面的所有元素都必须向前移动一个位置。否则，`vector` 容器上将会留下一个空位（hole），而 `vector` 容器的元素就不再是连续存放的了。

`deque` 容器拥有更加复杂的数据结构。从 `deque` 队列的两端插入和删除元素都非常快。在容器中间插入或删除付出的代价将更高。`deque` 容器同时提供了 `list` 和 `vector` 的一些性质：

- 与 `vector` 容器一样，在 `deque` 容器的中间 `insert` 或 `erase` 元素效率比较低。
- 不同于 `vector` 容器，`deque` 容器提供高效地在其首部实现 `insert` 和 `erase` 的操作，就像在容器尾部的一样。
- 与 `vector` 容器一样而不同于 `list` 容器的是，`deque` 容器支持对所有元素的随机访问。

- 在 deque 容器首部或尾部插入元素不会使任何迭代器失效，而在首部或尾部删除元素则只会使指向被删除元素的迭代器失效。在 deque 容器的任何其他位置的插入和删除操作将使指向该容器元素的所有迭代器都失效。

2. 元素的访问如何影响容器的选择

vector 和 deque 容器都支持对其元素实现高效的随机访问。也就是说，我们可以高效地先访问 5 号元素，然后访问 15 号元素，接着访问 7 号元素，等等。由于 vector 容器的每次访问都是距离其起点的固定偏移，因此其随机访问非常有效率。在 list 容器中，上述跳跃访问会变得慢很多。在 list 容器的元素之间移动的唯一方法是顺序跟随指针。从 5 号元素移动到 15 号元素必须遍历它们之间所有的元素。

334



提示 通常来说，除非找到选择使用其他容器的更好理由，否则 vector 容器都是最佳选择。

3. 选择容器的提示

下面列举了一些选择容器类型的法则：

- (1) 如果程序要求随机访问元素，则应使用 vector 或 deque 容器。
- (2) 如果程序必须在容器的中间位置插入或删除元素，则应采用 list 容器。
- (3) 如果程序不是在容器的中间位置，而是在容器首部或尾部插入或删除元素，则应采用 deque 容器。
- (4) 如果只需在读取输入时在容器的中间位置插入元素，然后需要随机访问元素，则可考虑在输入时将元素读入到一个 list 容器，接着对此容器重新排序，使其适合顺序访问，然后将排序后的 list 容器复制到一个 vector 容器。

如果程序既需要随机访问又必须在容器的中间位置插入或删除元素，那应该怎么办呢？

此时，选择何种容器取决于下面两种操作付出的相对代价：随机访问 list 容器元素的代价，以及在 vector 或 deque 容器中插入/删除元素时复制元素的代价。通常来说，应用中占优势的操作（程序中更多使用的是访问操作还是插入/删除操作）将决定应该选择什么类型的容器。

决定使用哪种容器可能要求剖析各种容器类型完成应用所要求的各类操作的性能。



最佳实践 如果无法确定某种应用应该采用哪种容器，则编写代码时尝试只使用 vector 和 list 容器都提供的操作：使用迭代器，而不是下标，并且避免随机访问元素。这样编写代码，在必要时，可很方便地将程序从使用 vector 容器修改为使用 list 容器。

习题

- 习题 9.33** 对于下列程序任务，采用哪种容器（vector、deque 还是 list）实现最合适？解释选择的理由。如果无法说明采用某种容器比另一种容器更好的原因，请解释为什么无法说明？
- 从一个文件中读入未知数目的单词，以生成英文句子。
 - 读入固定数目的单词，在输入时将它们按字母顺序插入到容器中。下一章将介绍更适合处理此类问题的关联容器。

- (c) 读入未知数目的单词。总是在容器尾部插入新单词，从容器首部删除下一个值。
- (d) 从一个文件中读入未知数目的整数。对这些整数排序，然后把它们输出到标准输出设备。

9.6 再谈 string 类型

3.2 节介绍了 string 类型，表 9-12 扼要重述了在该节中介绍的 string 操作。

表 9-12 3.2 节介绍的 string 操作	
string s;	定义一个新的空 string 对象，命名为 s
string s(cp);	定义一个新的 string 对象，用 cp 所指向的（以空字符 null 结束的）C 风格字符串初始化该对象
string s(s2);	定义一个新的 string 对象，并将它初始化为 s2 的副本
is >> s;	从输入流 is 中读取一个以空白字符分隔的字符串，写入 s
os << s;	将 s 写到输出流 os 中
getline(is, s)	从输入流 is 中读取一行字符，写入 s
s1 + s2	把 s1 和 s2 串接起来，产生一个新的 string 对象
s1 += s2	将 s2 拼接在 s1 后面
关系操作符	相等运算（==和!=）以及关系运算（<、<=、>和>=）都可用于 string 对象的比较，等效于（区分大小写的）字典次序的比较

除了已经使用过的操作外，string 类型还支持大多数顺序容器操作。在某些方面，可将 string 类型视为字符容器。除了一些特殊操作，string 类型提供与 vector 容器相同的操作。string 类型与 vector 容器不同的是，它不支持以栈方式操纵容器：在 string 类型中不能使用 front、back 和 pop_back 操作。

335

string 支持的容器操作有：

- 表 9-5 列出的 typedef，包括迭代器类型。
- 表 9-2 列出的容器构造函数，但是不包括只需一个长度参数的构造函数。
- 表 9-7 列出的 vector 容器所提供的添加元素的操作。注意：无论 vector 容器还是 string 类型都不支持 push_front 操作。
- 表 9-8 列出的长度操作。
- 表 9-9 列出的下标和 at 操作；但 string 类型不提供该表列出的 back 或 front 操作。
- 表 9-6 列出的 begin 和 end 操作。
- 表 9-10 列出的 erase 和 clear 操作；但是 string 类型不提供 pop_back 或 pop_front 操作。
- 表 9-11 列出的赋值操作。
- 与 vector 容器的元素一样，string 的字符也是连续存储的。因此，string 类型支持 9.4 节描述的 capacity 和 reserve 操作。

string 类型提供容器操作意味着可将操纵 vector 对象的程序改写为操纵 string 对象。例如，以下程序使用迭代器将一个 string 对象的字符以每次一行的方式输出到标准输出设备：

```
string s("Hiya!");
string::iterator iter = s.begin();
while (iter != s.end())
    cout << *iter++ << endl; // postfix increment: print old value
```

336

不要奇怪，这段代码看上去几乎与 5.5 节中输出 `vector<int>` 容器元素的程序一模一样。

除了共享容器的操作外，`string` 类型还支持其他本类型特有的操作。在本节剩下的篇幅中，我们将回顾这些 `string` 类型特有的操作，包括与其他容器相关操作的补充版本，以及全新的函数。`string` 类型提供的补充函数将在 9.6.2 节介绍。

`string` 类型为某些容器操作提供补充版本，以支持 `string` 特有的、不为其他容器共享的属性。例如，好几种操作允许指定指向字符数组的指针参数。无论字符串是否以空字符结束，这些操作都支持标准库 `string` 对象与字符数组之间的紧密交互作用。其他版本则使程序员只能使用下标而不能使用迭代器。这些版本只能通过位置操纵元素：指定起始位置，在某些情况下还需指定一个计数器，由此指定要操纵的某个元素或一段元素。

习题

习题 9.34 使用迭代器将 `string` 对象中的字符都改为大写字母。

习题 9.35 使用迭代器寻找和删除 `string` 对象中所有的大写字符。

习题 9.36 编写程序用 `vector<char>` 容器初始化 `string` 对象。

习题 9.37 假设希望一次读取一个字符并写入 `string` 对象，而且已知需要读入至少 100 个字符，考虑应该如何提高程序的性能？

337



提示

`string` 库定义了大量使用重复模式的函数。由于该类型支持的函数非常多，初次阅读本节时会觉得精神疲累。

读者可跳过 9.6 节剩下的内容。一旦知道了有哪些操作可以使用，就可以在编写需要使用这种操作的程序时，才回来阅读其细节。

9.6.1 构造 `string` 对象的其他方法

`string` 类支持表 9-2 所列出的几乎所有构造函数，只有一个例外：`string` 不支持带有单个容器长度作为参数的构造函数。创建 `string` 对象时：不提供任何参数，则得到空的 `string` 对象；也可将新对象初始化为另一个 `string` 对象的副本；或用一对迭代器初始化；或者使用一个计数器和一个字符初始化：

```
string s1; // s1 is the empty string
string s2(5, 'a'); // s2 == "aaaaa"
string s3(s2); // s3 is a copy of s2
string s4(s3.begin(),
          s3.begin() + s3.size() / 2); // s4 == "aa"
```

除了上述构造函数之外，`string` 类型还提供了三种其他方式创建该类对象（表 9-13）。在前面的章节中，已经使用过只有一个指针参数的构造函数，该指针指向以空字符结束的字符数组

中的第一个元素。另一种构造函数需要一个指向字符数组元素的指针和一个标记要复制多少个字符的计数器作参数。由于该构造函数带有一个计数器，因此数组不必以空字符结束：

```
char *cp = "Hiya";           // null-terminated array
char c_array[] = "World!!!!"; // null-terminated
char no_null[] = {'H', 'i'}; // not null-terminated
string s1(cp);                // s1 == "Hiya"
string s2(c_array, 5);        // s2 == "World"
string s3(c_array + 5, 4);    // s3 == "!!!!"
string s4(no_null);           // runtime error: no_null not null-terminated
string s5(no_null, 2);        // ok: s5 == "Hi"
```

使用只有一个指针参数的构造函数定义 s1，该指针指向以空字符结束的数组中的第一个字符。这个数组的所有字符，但不包括结束符 null，都被复制到新创建的 string 对象中。

表 9-13 构造 string 对象的其他方法	
string s(cp, n)	创建一个 string 对象，它被初始化为 cp 所指向数组的前 n 个元素的副本
string s(s2, pos2)	创建一个 string 对象，它被初始化为一个已存在的 string 对象 s2 中从下标 pos2 开始的字符的副本 如果 pos2 > s2.size()，则该操作未定义
string s(s2, pos2, len2)	创建一个 string 对象，它被初始化为 s2 中从下标 pos2 开始的 len2 个字符的副本。如果 pos2 > s2.size()，则该操作未定义 无论 len2 的值是多少，最多只能复制 s2.size() - pos2 个字符
注意：n、len2 和 pos2 都是 unsigned 值	

而 s2 的初始化式则通过第二种构造函数实现，它的参数包括一个指针和一个计数器。在这个例子中，从参数指针指向的那个字符开始，连续复制第二个参数指定数目的字符。因此，s2 是 c_array 数组前 5 个字符的副本。记住，将数组作为参数传递时，数组将自动转换为指向其第一个元素的指针。当然，并没有限制非得传递指向数组起点的指针不可。通过给 s3 的构造函数传递指向 c_array 数组中第一个感叹号字符的指针，s3 被初始化为存储 4 个感叹号字符的 string 对象。

s4 和 s5 的初始化式并不是 C 风格字符串。其中，s4 的定义是错误的。调用这种形式的初始化，其参数必须是以空字符结束的数组。将不包含 null 的数组传递给此构造函数将导致编译器无法检测的严重错误（4.3 节），此类错误在运行时将会发生什么状况并未定义。

s5 的初始化则是正确的：初始化式包含了一个计数器，以说明要复制多少个字符用于初始化。该计数器的值必须小于数组的长度，此时，无论数组是否以空字符结束，都没什么关系。

用子串做初始化式

另一对构造函数使程序员可以在创建 string 对象时将其初始化为另一个 string 对象的子串。

```
string s6(s1, 2);           // s6 == "ya"
string s7(s1, 0, 2);        // s7 == "Hi"
string s8(s1, 0, 8);        // s8 == "Hiya"
```

第一个语句的两个参数指定了要复制的 string 对象及其复制的起点。在两个参数的构造函数版

本中，复制 `string` 对象实参中从指定位置到其末尾的所有字符，用于初始化新创建的 `string` 对象。还可以为此类构造函数提供第三个参数，用于指定复制字符的个数。在本例中，我们从指定位置开始复制指定数目（最多为 `string` 对象的长度）的字符数。例如，创建 `s7` 时，从 `s1` 中下标为 0 的位置开始复制两个字符；而创建 `s8` 时，只复制了 4 个字符，而并不是要求的 8 个字符。无论要求复制多少个字符，标准库最多只能复制数目与 `string` 对象长度相等的字符。

9.6.2 修改 `string` 对象的其他方法

339

`string` 类型支持的许多容器操作在操作时都以迭代器为基础。例如，`erase` 操作需要一个迭代器或一段迭代器范围作其参数，用于指定从容器中删除的元素。类似地，所有版本的 `insert` 函数的第一个参数都是一个指向插入位置之后的迭代器，而新插入的元素值则由其他参数指定。尽管 `string` 类型支持这些基于迭代器的操作，它同样也提供以下标为基础的操作。下标用于指定 `erase` 操作的起始元素，或在其前面 `insert` 适当值的元素。表 9-14 列出了 `string` 类型和容器类型共有的操作；而表 9-15 则列出了 `string` 类型特有的操作。

表 9-14 与容器共有的 <code>string</code> 操作	
<code>s.insert(p, t)</code>	在迭代器 <code>p</code> 指向的元素之前插入一个值为 <code>t</code> 的新元素。返回指向新插入元素的迭代器
<code>s.insert(p, n, t)</code>	在迭代器 <code>p</code> 指向的元素之前插入 <code>n</code> 个值为 <code>t</code> 的新元素。返回 <code>void</code>
<code>s.insert(p, b, e)</code>	在迭代器 <code>p</code> 指向的元素之前插入迭代器 <code>b</code> 和 <code>e</code> 标记范围内所有的元素。返回 <code>void</code>
<code>s.assign(b, e)</code>	用迭代器 <code>b</code> 和 <code>e</code> 标记范围内的元素替换 <code>s</code> 。对于 <code>string</code> 类型，该操作返回 <code>s</code> ；对于容器类型，则返回 <code>void</code>
<code>s.assign(n, t)</code>	用值为 <code>t</code> 的 <code>n</code> 个副本替换 <code>s</code> 。对于 <code>string</code> 类型，该操作返回 <code>s</code> ；对于容器类型，则返回 <code>void</code>
<code>s.erase(p)</code>	删除迭代器 <code>p</code> 指向的元素。返回一个迭代器，指向被删除元素后面的元素
<code>s.erase(b, e)</code>	删除迭代器 <code>b</code> 和 <code>e</code> 标记范围内所有的元素。返回一个迭代器，指向被删除元素段后面的第一个元素

表 9-15 <code>string</code> 类型特有的版本	
<code>s.insert(pos, n, c)</code>	在下标为 <code>pos</code> 的元素之前插入 <code>n</code> 个字符 <code>c</code>
<code>s.insert(pos, s2)</code>	在下标为 <code>pos</code> 的元素之前插入 <code>string</code> 对象 <code>s2</code> 的副本
<code>s.insert(pos, s2, pos2, len)</code>	在下标为 <code>pos</code> 的元素之前插入 <code>s2</code> 中从下标 <code>pos2</code> 开始的 <code>len</code> 个字符
<code>s.insert(pos, cp, len)</code>	在下标为 <code>pos</code> 的元素之前插入 <code>cp</code> 所指向数组的前 <code>len</code> 个字符
<code>s.insert(pos, cp)</code>	在下标为 <code>pos</code> 的元素之前插入 <code>cp</code> 所指向的以空字符结束的字符串副本
<code>s.assign(s2)</code>	用 <code>s2</code> 的副本替换 <code>s</code>
<code>s.assign(s2, pos2, len)</code>	用 <code>s2</code> 中从下标 <code>pos2</code> 开始的 <code>len</code> 个字符副本替换 <code>s</code>
<code>s.assign(cp, len)</code>	用 <code>cp</code> 所指向数组的前 <code>len</code> 个字符副本替换 <code>s</code>
<code>s.assign(cp)</code>	用 <code>cp</code> 所指向的以空字符结束的字符串副本替换 <code>s</code>
<code>s.erase(pos, len)</code>	删除从下标 <code>pos</code> 开始的 <code>len</code> 个字符
除非特殊声明，上述所有操作都返回 <code>s</code> 的引用	

1. 基于位置的实参

`string` 类型为这些操作提供本类型特有的版本，它们接受的实参类似于在前一节介绍的补充构造函数。程序员可通过这些操作基于位置处理 `string` 对象，并/或使用指向字符数组的指针而不是 `string` 对象作实参。

例如，所有容器都允许程序员指定一对迭代器，用于标记删除（`erase`）的元素范围。对于 `string` 类型，还允许通过为 `erase` 函数传递一个起点位置和删除元素的数目，来指定删除的范围。假设 `s` 至少有 5 个元素，下面的语句用于删除 `s` 的最后 5 个字符：

```
s.erase(s.size() - 5, 5); // erase last five characters from s
```

类似地，对于容器类型，可在迭代器指向的元素之前插入（`insert`）指定数目的新值。而对于 `string` 类型，系统还允许使用下标而不是迭代器指定插入位置：

```
s.insert(s.size(), 5, '!'); // insert five exclamation points at end of s
```

2. 指定新的内容

在 `string` 对象中 `insert` 或 `assign` 的字符可来自于字符数组或另一个 `string` 对象。例如，以空字符结束的字符数组可以用作 `insert` 或 `assign` 到 `string` 对象的内容：

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

类似地，可如下所示将一个 `string` 对象的副本插入到另一个 `string` 对象中：

```
s = "some string";
s2 = "some other string";
// 3 equivalent ways to insert all the characters from s2 at beginning of s
// insert iterator range before s.begin()
s.insert(s.begin(), s2.begin(), s2.end());
// insert copy of s2 before position 0 in s
s.insert(0, s2);
// insert s2.size() characters from s2 starting at s2[0] before s[0]
s.insert(0, s2, 0, s2.size());
```

9.6.3 只适用于 `string` 类型的操作

`string` 类型提供了容器类型不支持的其他几种操作，如表 9-16 所示：

- `substr` 函数，返回当前 `string` 对象的子串。
- `append` 和 `replace` 函数，用于修改 `string` 对象。
- 一系列 `find` 函数，用于查找 `string` 对象。

表 9-16 子串操作

表 9-16 子串操作	
<code>s.substr(pos, n)</code>	返回一个 <code>string</code> 类型的字符串，它包含 <code>s</code> 中从下标 <code>pos</code> 开始的 <code>n</code> 个字符
<code>s.substr(pos)</code>	返回一个 <code>string</code> 类型的字符串，它包含从下标 <code>pos</code> 开始到 <code>s</code> 末尾的所有字符
<code>s.substr()</code>	返回 <code>s</code> 的副本

1. substr 操作

使用 `substr` 操作可在指定 `string` 对象中检索需要的子串。我们可以给 `substr` 函数传递查找的起点和一个计数器。该函数将生成一个新的 `string` 对象，包含原目标 `string` 对象从指定位置开始的若干个字符（字符数目由计数器决定，但最多只能到原 `string` 对象的最后一个字符）：

```
string s("hello world");
// return substring of 5 characters starting at position 6
string s2 = s.substr(6, 5); // s2 = world
```

可选择另一种方法实现相同的功能：

```
// return substring from position 6 to the end of s
string s3 = s.substr(6); // s3 = world
```

2. append 和 replace 函数

`string` 类型提供了 6 个 `append` 重载函数版本和 10 个 `replace` 版本（见表 9-17）。`append` 和 `replace` 函数使用了相同的参数集合实现重载。这些参数如表 9-18 所示，用于指定在 `string` 对象中添加的字符。对于 `append` 操作，字符将添加在 `string` 对象的末尾。而 `replace` 函数则将这些字符插入到指定位置，从而替换 `string` 对象中一段已存在的字符。

表 9-17 修改 `string` 对象的操作（`args` 在表 9-18 中定义）

<code>s.append(args)</code>	将 <code>args</code> 串接在 <code>s</code> 后面。返回 <code>s</code> 的引用
<code>s.replace(pos, len, args)</code>	删除 <code>s</code> 中从下标 <code>pos</code> 开始的 <code>len</code> 个字符，用 <code>args</code> 指定的字符替换之。返回 <code>s</code> 的引用 在这个版本中， <code>args</code> 不能为 <code>b2</code> , <code>e2</code>
<code>s.replace(b, e, args)</code>	删除迭代器 <code>b</code> 和 <code>e</code> 标记的范围内所有的字符，用 <code>args</code> 替换之。返回 <code>s</code> 的引用 在这个版本中， <code>args</code> 不能为 <code>s2</code> , <code>pos2</code> , <code>len2</code>

表 9-18 `append` 和 `replace` 操作的参数：`args`

<code>s2</code>	<code>string</code> 类型的字符串 <code>s2</code>
<code>s2, pos2, len2</code>	字符串 <code>s2</code> 中从下标 <code>pos2</code> 开始的 <code>len2</code> 个字符
<code>cp</code>	指针 <code>cp</code> 指向的以空字符结束的数组
<code>cp, len2</code>	<code>cp</code> 指向的以空字符结束的数组中前 <code>len2</code> 个字符
<code>n, c</code>	字符 <code>c</code> 的 <code>n</code> 个副本
<code>b2, e2</code>	迭代器 <code>b2</code> 和 <code>e2</code> 标记的范围内所有字符

`append` 操作提供了在字符串尾部插入的捷径：

```
string s("C++ Primer"); // initialize s to "C++ Primer"
s.append(" 3rd Ed."); // s == "C++ Primer 3rd Ed."
// equivalent to s.append(" 3rd Ed.")
s.insert(s.size(), " 3rd Ed.");
```

`replace` 操作用于删除一段指定范围的字符，然后在删除位置插入一组新字符，等效于调用 `erase` 和 `insert` 函数。

`string` 类型为 `replace` 操作提供了 10 个不同版本，其差别在于以不同的方式指定要删除

的字符和要插入的新字符。前两个参数应指定删除的元素范围，可用迭代器对实现，也可用一个下标和一个计数器实现。其他的参数则用于指定插入的新字符。

可将 `replace` 视为删除一些字符然后在同一位置插入其他内容的捷径：

342

```
// starting at position 11, erase 3 characters and then insert "4th"
s.replace(11, 3, "4th");           // s == "C++ Primer 4th Ed."
// equivalent way to replace "3rd" by "4th"
s.erase(11, 3);                   // s == "C++ Primer Ed."
s.insert(11, "4th");               // s == "C++ Primer 4th Ed."
```

注解

不要求删除的文本长度与插入的相同。

上述例子调用了 `replace` 函数，插入的文本恰好与删除的文本长度相同。我们也可以插入更长或更短的字符串：

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

在这个例子中，删除了 3 个字符，但在同一个位置却插入了 6 个字符。

9.6.4 string 类型的查找操作

`string` 类提供了 6 种查找函数（表 9-19），每种函数以不同形式的 `find` 命名。这些操作全都返回 `string::size_type` 类型的值，以下标形式标记查找匹配所发生的位置；或者返回一个名为 `string::npos` 的特殊值，说明查找没有匹配。`string` 类将 `npos` 定义为保证大于任何有效下标的值。

343

表 9-19 string 类型的查找操作（其参数 args 在表 9-20 中定义）	
<code>s.find(args)</code>	在 <code>s</code> 中查找 <code>args</code> 的第一次出现
<code>s.rfind(args)</code>	在 <code>s</code> 中查找 <code>args</code> 的最后一次出现
<code>s.find_first_of(args)</code>	在 <code>s</code> 中查找 <code>args</code> 的任意字符的第一次出现
<code>s.find_last_of(args)</code>	在 <code>s</code> 中查找 <code>args</code> 的任意字符的最后一次出现
<code>s.find_first_not_of(args)</code>	在 <code>s</code> 中查找第一个不属于 <code>args</code> 的字符
<code>s.find_last_not_of(args)</code>	在 <code>s</code> 中查找最后一个不属于 <code>args</code> 的字符

每种查找操作都有 4 个重载版本，每个版本使用不同的参数集合。表 9-20 列出了查找操作使用的不同参数形式。基本上，这些操作的不同之处在于查找的到底是单个字符、另一个 `string` 字符串、C 风格的以空字符结束的字符串，还是用字符数组给出的特定数目的字符集合。

表 9-20 string 类型提供的 find 操作的参数	
<code>c, pos</code>	在 <code>s</code> 中，从下标 <code>pos</code> 标记的位置开始，查找字符 <code>c</code> 。 <code>pos</code> 的默认值为 0
<code>s2, pos</code>	在 <code>s</code> 中，从下标 <code>pos</code> 标记的位置开始，查找 <code>string</code> 对象 <code>s2</code> 。 <code>pos</code> 的默认值为 0
<code>cp, pos</code>	在 <code>s</code> 中，从下标 <code>pos</code> 标记的位置开始，查找指针 <code>cp</code> 所指向的 C 风格的以空字符结束的字符串。 <code>pos</code> 的默认值为 0
<code>cp, pos, n</code>	在 <code>s</code> 中，从下标 <code>pos</code> 标记的位置开始，查找指针 <code>cp</code> 所指向数组的前 <code>n</code> 个字符。 <code>pos</code> 和 <code>n</code> 都没有默认值

344

1. 精确匹配的查找

最简单的查找操作是 `find` 函数，用于寻找实参指定的内容。如果找到的话，则返回第一次匹配的下标值；如果找不到，则返回 `npos`：

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

返回 0，这是子串 "Anna" 位于字符串 "AnnaBelle" 中的下标。



默认情况下，`find` 操作（以及其他处理字符的 `string` 操作）使用内置操作符比较 `string` 字符串中的字符。因此，这些操作（以及其他 `string` 操作）都区分字母的大小写。

以下程序寻找 `string` 对象中的某个值，字母的大小写影响了程序结果：

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

这段代码使 `pos1` 的值为 `npos`——字符串 Anna 与 anna 不匹配。



`find` 操作的返回类型是 `string::size_type`，请使用该类型的对象存储 `find` 的返回值。

2. 查找任意字符

如果在查找字符串时希望匹配任意指定的字符，则实现起来稍微复杂一点。例如，下面的程序要在 `name` 中寻找并定位第一个数字：

```
string numerics("0123456789");
string name("r2d2");
string::size_type pos = name.find_first_of(numerics);
cout << "found number at index: " << pos
      << " element is " << name[pos] << endl;
```

在这个例子中，`pos` 的值被设置为 1（记住，`string` 对象的元素下标从 0 开始计数）。

3. 指定查找的起点

程序员可以给 `find` 操作传递一个可选的起点位置实参，用于指定开始查找的下标位置，该位置实参的默认值为 0。通常的编程模式是使用这个可选的实参循环查找 `string` 对象中所有的匹配。下面的程序重写了查找 "r2d2" 的程序，以便找出 `name` 字符串中出现的所有数字：

```
string::size_type pos = 0;
// each trip reset pos to the next instance in name
while ((pos = name.find_first_of(numerics, pos))
       != string::npos) {
    cout << "found number at index: " << pos
          << " element is " << name[pos] << endl;
    ++pos; // move to the next character
}
```

在这个例子中，首先将 `pos` 初始化为 0，使第一次循环从 0 号元素开始查找 `name`。`while` 的循环条件实现两个功能：从当前 `pos` 位置开始查找，并将找到的第一个数字出现的下标值赋给 `pos`。

当 `find_first_of` 函数返回有效的下标值时，输出此次查找的结果，并且让 `pos` 加 1。

如果漏掉了循环体末尾让 `pos` 加 1 的语句，那么循环永远都不会结束。考虑没有该操作时，会发生什么情况？第二次循环时，从 `pos` 标记的位置开始查找，而此时 `pos` 标记的就是一个数字，于是 `find_first_of` 函数将（不断重复地）返回同一个 `pos` 值。



`pos` 的值必须加 1，以确保下一次循环从刚找到的数字后面开始查找下一个数字。

345

4. 寻找不匹配点

除了寻找匹配的位置外，还可以调用 `find_first_not_of` 函数查找第一个与实参不匹配的位置。例如，如果要在 `string` 对象中寻找第一个非数字字符，可以如下编写程序：

```
string numbers("0123456789");
string dept("03714p3");
// returns 5, which is the index to the character 'p'
string::size_type pos = dept.find_first_not_of(numbers);
```

5. 反向查找

迄今为止，我们使用的所有 `find` 操作都是从左向右查找的。除此之外，标准库还提供了一组类似的从右向左查找 `string` 对象的操作。`rfind` 成员函数用于寻找最后一个——也就是最右边的——指定子串出现的位置：

```
string river("Mississippi");
string::size_type first_pos = river.find("is"); // returns 1
string::size_type last_pos = river.rfind("is"); // returns 4
```

`find` 函数返回下标 1，标记 `river` 字符串中第一个 "is" 的出现位置；而 `rfind` 函数则返回下标 4，标记最后一个 "is" 的出现位置。

6. `find_last` 函数

`find_last` 函数类似对应的 `find_first` 函数，唯一的差别在于 `find_last` 函数返回最后一个匹配的位置，而并不是第一个。

- `find_last_of` 函数查找与目标字符串的任意字符匹配的最后一个字符。
- `find_last_not_of` 函数查找最后一个不能跟目标字符串的任何字符匹配的字符。

这两个操作都提供第二个参数，这个参数是可选的，用于指定在 `string` 对象中开始查找的位置。

习题

习题 9.38 已知有如下 `string` 对象：

```
"ab2c3d7R4E6"
```

编写程序寻找该字符串中所有的数字字符，然后再寻找所有的字母字符。以两种版本编写该程序：第一个版本使用 `find_first_of` 函数，而第二个版本则使用 `find_first_not_of` 函数。

习题 9.39 已知有如下 `string` 对象：

```
string line1 = "We were her pride of 10 she named us:";
string line2 = "Benjamin, Phoenix, the Prodigal"
string line3 = "and perspicacious pacific Suzanne";

string sentence = line1 + ' ' + line2 + ' ' + line3;
```

编写程序计算 `sentence` 中有多少个单词，并指出其中最长和最短的单词。如果有多个最长或最短的单词，则将它们全部输出。

9.6.5 `string` 对象的比较

正如在 3.2.3 节所看到的，`string` 类型定义了所有关系操作符，使程序员可以比较两个 `string` 对象是否相等（`==`）、不等（`!=`），以及实现小于或大于（`<`、`<=`、`>`、`>=`）运算。`string` 对象采用字典顺序比较，也就是说，`string` 对象的比较与大小写敏感的字典顺序比较相同：

```
string cobol_program_crash("abend");
string cplus_program_crash("abort");
```

这里的 `cobol_program_crash` 小于 `cplus_program_crash`。对于两个 `string` 对象，关系操作符逐个字符地进行比较，直到比较到某个位置上，两个 `string` 对象对应的字符不相同为止。`string` 对象的整个比较依赖于不相同字符之间的比较。在本例中，第一个不相等的字符是 'e' 和 'o'。由于在英文字母表中，'e' 出现得比 'o' 早（即 'e' 小于 'o'），于是 "abend" 小于 "abort"。如果要比较的两个 `string` 对象长度不相同，而且一个 `string` 对象是另一个 `string` 对象的子串，则较短的 `string` 对象小于较长的 `string` 对象。

compare 函数

除了关系操作符，`string` 类型还提供了一组 `compare` 操作（表 9-21），用于实现字典顺序的比较。这些操作的结果类似于 C 语言中的库函数 `strcmp`（4.3 节）。假设有语句：

```
s1.compare (args);
```

表 9-21 `string` 类型的 `compare` 操作

<code>s.compare(s2)</code>	比较 <code>s</code> 和 <code>s2</code>
<code>s.compare(pos1, n1, s2)</code>	让 <code>s</code> 中从 <code>pos1</code> 下标位置开始的 <code>n1</code> 个字符与 <code>s2</code> 做比较
<code>s.compare(pos1, n1, s2, pos2, n2)</code>	让 <code>s</code> 中从 <code>pos1</code> 下标位置开始的 <code>n1</code> 个字符与 <code>s2</code> 中从 <code>pos2</code> 下标位置开始的 <code>n2</code> 个字符做比较
<code>s.compare(cp)</code>	比较 <code>s</code> 和 <code>cp</code> 所指向的以空字符结束的字符串
<code>s.compare(pos1, n1, cp)</code>	让 <code>s</code> 中从 <code>pos1</code> 下标位置开始的 <code>n1</code> 个字符与 <code>cp</code> 所指向的字符串做比较
<code>s.compare(pos1, n1, cp, n2)</code>	让 <code>s</code> 中从 <code>pos1</code> 下标位置开始的 <code>n1</code> 个字符与 <code>cp</code> 所指向字符串的前 <code>n2</code> 个字符做比较

`compare` 函数返回下面列出的三种可能值之一：

- (1) 正数，此时 `s1` 大于 `args` 所代表的 `string` 对象。
- (2) 负数，此时 `s1` 小于 `args` 所代表的 `string` 对象。

(3) 0, 此时 s1 恰好等于 *args* 所代表的 string 对象。

例如:

```
// returns a negative value
cobol_program_crash.compare(cplusplus_program_crash);
// returns a positive value
cplusplus_program_crash.compare(cobol_program_crash);
```

347

compare 操作提供了 6 种重载函数版本, 以方便程序员实现一个或两个 string 对象的子串的比较, 以及 string 对象与字符数组或其中某一部分的比较:

```
char second_ed[] = "C++ Primer, 2nd Edition";
string third_ed("C++ Primer, 3rd Edition");
string fourth_ed("C++ Primer, 4th Edition");
// compares C++ library string to C-style string
fourth_ed.compare(second_ed); // ok, second_ed is null-terminated
// compare substrings of fourth_ed and third_ed
fourth_ed.compare(fourth_ed.find("4th"), 3,
                  third_ed, third_ed.find("3rd"), 3);
```

我们对第二个 compare 函数的调用更感兴趣。这个调用使用了具有 5 个参数的 compare 函数版本。先调用 find 函数找到子串 "4th" 的起点。让从此起点开始的 3 个字符与 third_ed 的子串做比较。而 third_ed 的子串起始于 find 函数找到的 "3rd" 的起点, 同样取其随后的 3 个字符参加比较。可见这个语句本质上比较的是 "4th" 和 "3rd"。

习题

习题 9.40 编写程序接收下列两个 string 对象:

```
string q1("When lilacs last in the dooryard bloom'd");
string q2("The child is father of the man");
```

然后使用 assign 和 append 操作, 创建 string 对象:

```
string sentence("The child is in the dooryard");
```

习题 9.41 已知有如下 string 对象:

```
string generic1("Dear Ms Daisy:");
string generic2("MrsMsMissPeople");
```

编写程序实现下面函数:

```
string greet(string form, string lastname, string title,
             string::size_type pos, int length);
```

该函数使用 replace 操作实现以下功能: 对于字符串 form, 将其中的 Daisy 替换为 lastname, 将其中的 Ms 替换为字符串 generic2 中从 pos 下标开始的 length 个字符。例如, 下面的语句:

```
string lastName("AnnaP");
string salute = greet(generic1, lastName, generic2, 5, 4);
```

将返回字符串:

```
Dear Miss AnnaP:
```

9.7 容器适配器

除了顺序容器，标准库还提供了三种顺序容器适配器：queue、priority_queue 和 stack。适配器（adaptor）是标准库中通用的概念，包括容器适配器、迭代器适配器和函数适配器。本质上，适配器是使一事物的行为类似于另一事物的行为的一种机制。容器适配器让一种已存在的容器类型采用另一种不同的抽象类型的工作方式实现。例如，stack（栈）适配器可使任何一种顺序容器以栈的方式工作。表 9-22 列出了所有容器适配器通用的操作和类型。

348

表 9-22 适配器通用的操作和类型	
size_type	一种类型，足以存储此适配器类型最大对象的长度
value_type	元素类型
container_type	基础容器的类型，适配器在此容器类型上实现
A a;	创建一个新的空适配器，命名为 a
A a(c);	创建一个名为 a 的新适配器，初始化为容器 c 的副本
关系操作符	所有适配器都支持全部关系操作符：==、!=、<、<=、>、>=

使用适配器时，必须包含相关的头文件：

```
#include <stack>      // stack adaptor
#include <queue>      // both queue and priority_queue adaptors
```

1. 适配器的初始化

所有适配器都定义了两个构造函数：默认构造函数用于创建空对象，而带一个容器参数的构造函数将参数容器的副本作为其基础值。例如，假设 deq 是 deque<int>类型的容器，则可用 deq 初始化一个新的栈，如下所示：

```
stack<int> stk(deq);      // copies elements from deq into stk
```

2. 覆盖基础容器类型

默认的 stack 和 queue 都基于 deque 容器实现，而 priority_queue 则在 vector 容器上实现。在创建适配器时，通过将一个顺序容器指定为适配器的第二个类型实参，可覆盖其关联的基础容器类型：

```
// empty stack implemented on top of vector
stack< string, vector<string> > str_stk;
// str_stk2 is implemented on top of vector and holds a copy of svec
stack<string, vector<string> > str_stk2(svec);
```

349

对于给定的适配器，其关联的容器必须满足一定的约束条件。stack 适配器所关联的基础容器可以是任意一种顺序容器类型。因此，stack 栈可以建立在 vector、list 或者 deque 容器之上。而 queue 适配器要求其关联的基础容器必须提供 push_front 运算，因此只能建立在 list 容器上，而不能建立在 vector 容器上。priority_queue 适配器要求提供随机访问功能，因此可建立在 vector 或 deque 容器上，但不能建立在 list 容器上。

3. 适配器的关系运算

两个相同类型的适配器可以做相等、不等、小于、大于、小于等于以及大于等于关系比较，只要基础元素类型支持等于和小于操作符即可。这些关系运算由元素依次比较来实现。第一对不相等的元素将决定两者之间的小于或大于关系。

9.7.1 栈适配器

表 9-23 列出了栈提供的所有操作。

表 9-23 栈容器适配器支持的操作	
s.empty()	如果栈为空，则返回 true，否则返回 false
s.size()	返回栈中元素的个数
s.pop()	删除栈顶元素，但不返回其值
s.top()	返回栈顶元素的值，但不删除该元素
s.push(item)	在栈顶压入新元素

下面的程序使用了这 5 个栈操作：

```
// number of elements we'll put in our stack
const stack<int>::size_type stk_size = 10;
stack<int> intStack; // empty stack
// fill up the stack
int ix = 0;
while (intStack.size() != stk_size)
    // use postfix increment; want to push old value onto intStack
    intStack.push(ix++); // intStack holds 0...9 inclusive
int error_cnt = 0;
// look at each value and pop it off the stack
while (intStack.empty() == false) {
    int value = intStack.top();
    // read the top element of the stack
    if (value != --ix) {
        cerr << "oops! expected " << ix
            << " received " << value << endl;
        ++error_cnt;
    }
    intStack.pop(); // pop the top element, and repeat
}
cout << "Our program ran with "
    << error_cnt << " errors!" << endl;
```

350

声明语句：

```
stack<int> intStack; // empty stack
```

将 intStack 定义为一个存储整型元素的空栈。第一个 while 循环在该栈中添加了 stk_size 个元素，元素初值是从 0 开始依次递增 1 的整数。第二个 while 循环迭代遍历整个栈，检查其栈顶 (top) 的元素值，然后栈顶元素出栈，直到栈变空为止。

所有容器适配器都根据其基础容器类型所支持的操作来定义自己的操作。默认情况下，栈适

配器建立在 deque 容器上，因此采用 deque 提供的操作来实现栈功能。例如，执行下面的语句：

```
// use postfix increment; want to push old value onto intStack
intStack.push(ix++); // intStack holds 0...9 inclusive
```

这个操作通过调用 push_back 操作实现，而该 push_back 操作由 intStack 所基于的 deque 对象提供。尽管栈是以 deque 容器为基础实现的，但是程序员不能直接访问 deque 所提供的操作。例如，不能在栈上调用 push_back 函数，而是必须使用栈所提供的名为 push 的操作。

9.7.2 队列和优先级队列

标准库队列使用了先进先出（FIFO）的存储和检索策略。进入队列的对象被放置在尾部，下一个被取出的元素则取自队列的首部。标准库提供了两种风格的队列：FIFO 队列（FIFO queue，简称 queue），以及优先级队列（priority queue）。

priority_queue 允许用户为队列中存储的元素设置优先级。这种队列不是直接将新元素放置在队列尾部，而是放在比它优先级低的元素前面。标准库默认使用元素类型的 < 操作符来确定它们之间的优先级关系。

351

优先级队列的一个实例是机场行李检查队列。30 分钟后即将离港的航班的乘客通常会被移到队列前面，以便他们能在飞机起飞前完成检查过程。使用优先级队列的程序示例是操作系统的调度表，它决定在大量等待进程中下一个要执行的进程。

要使用这两种队列，必须包含 queue 头文件。表 9-24 列出了队列和优先级队列所提供的的所有操作。

表 9-24 队列和优先级队列支持的操作	
q.empty()	如果队列为空，则返回 true，否则返回 false
q.size()	返回队列中元素的个数
q.pop()	删除队首元素，但不返回其值
q.front()	返回队首元素的值，但不删除该元素 该操作只适用于队列
q.back()	返回队尾元素的值，但不删除该元素 该操作只适用于队列
q.top()	返回具有最高优先级的元素值，但不删除该元素 该操作只适用于优先级队列
q.push(item)	对于 queue，在队尾压入一个新元素 对于 priority_queue，在基于优先级的适当位置插入新元素

习题

- 习题 9.42 编写程序读入一系列单词，并将它们存储在 stack 对象中。
- 习题 9.43 使用 stack 对象处理带圆括号的表达式。遇到左圆括号时，将其标记下来。然后在遇到右圆括号时，弹出 stack 对象中这两边括号之间的相关元素（包括左圆括号）。接着在 stack 对象中压入一个值，用以表明这个用一对圆括号括起来的表达式已经被替换。

352

小结

C++标准库定义了一系列顺序容器类型。容器是用于存储某种给定类型对象的模板类型。在顺序容器中，所有元素根据其位置排列和访问。顺序容器共享一组通用的已标准化的接口：如果两种顺序容器都提供某一操作，那么该操作具有相同的接口和含义。所有容器都提供（有效的）动态内存管理。程序员在容器中添加元素时，不必操心元素存放在哪里。容器自己实现其存储管理。

最经常使用的容器类型是 `vector`，它支持对元素的快速随机访问。可高效地在 `vector` 容器尾部添加和删除元素，而在其他任何位置上的插入或删除运算则要付出比较昂贵的代价。`deque` 类与 `vector` 相似，但它还支持在 `deque` 首部的快速插入和删除运算。`list` 类只支持元素的顺序访问，但在 `list` 内部任何位置插入和删除元素都非常快速。

容器定义的操作非常少，只定义了构造函数、添加或删除元素的操作、设置容器长度的操作以及返回指向特殊元素的迭代器的操作。其他一些有用的操作，如排序、查找，则不是由容器类型定义，而是由第11章介绍的标准算法定义。

在容器中添加或删除元素可能会使已存在的迭代器失效。当混合使用迭代器操作和容器操作时，必须时刻留意给定的容器操作是否会使迭代器失效。许多使一个迭代器失效的操作，例如 `insert` 或 `erase`，将返回一个新的迭代器，让程序员保留容器中的一个位置。使用改变容器长度的容器操作的循环必须非常小心其迭代器的使用。

术语

adaptor（适配器） 一种标准库类型、函数或迭代器，使某种标准库类型、函数或迭代器的行为类似于另外一种标准库类型、函数或迭代器。系统提供了三种顺序容器适配器：`stack`（栈）、`queue`（队列）以及 `priority_queue`（优先级队列）。所有的适配器都会在其基础顺序容器上定义一个新的接口。

begin（begin 操作） 一种容器操作。如果容器中有元素，该操作返回指向容器中第一个元素的迭代器；如果容器为空，则返回超出末端迭代器。

container（容器） 一种存储给定类型对象集合的类型。所有标准库容器类型都是模板类型。定义容器时，必须指定在该容器中存储的元素是什么类型。标准库容器具有可变的长度。

deque（双端队列） 一种顺序容器。`deque` 中存储的元素通过其下标位置访问。该容器类型在很多方面与 `vector` 一样，唯一的不同是 `deque` 类型支持在容器首部快速地插入新元素，就像在尾部插入一样，而且无论在容器的哪一端插入或删除都不会引起元素的重新定位。

end（end 操作） 一种容器操作，返回指向容器的超出末端的下一位置的迭代器。

invalidated iterator（无效迭代器） 指向不再存在的元素的迭代器。无效迭代器的使用未定义，可能会导致严重的运行时错误。

iterator（迭代器） 一种类型，其操作支持遍历和检查容器元素的操作。所有标准库容器都定义了表 9-5 列出的 4 种迭代器，与之共同工

作。标准库迭代器都支持解引用 (*) 操作符和箭头 (->) 操作符, 用于检查迭代器指向的元素值。它们还支持前置和后置的自增 (++)、自减操作符 (--), 以及相等 (==) 和不等 (!=) 操作符。

iterator range (迭代器范围) 由一对迭代器标记的一段元素范围。第一个迭代器指向序列中的第一个元素, 而第二个迭代器则指向该范围中的最后一个元素的下一位置。如果这段范围为空, 则这两个迭代器相等 (反之亦然——如果这两个迭代器相等, 则它们标记一个空范围)。如果这段范围非空, 则对第一个迭代器重复做自增运算, 必然能到达第二个迭代器。通过这个对迭代器进行自增的过程, 即可处理该序列中所有的元素。

left-inclusive interval (左闭合区间) 一段包含第一个元素但不包含最后一个元素的范围。一般表示为 [i, j], 意味着该序列从 i 开始 (包括 i) 一直到 j, 但不包含 j。

list (列表) 一种顺序容器。list 中的元素只能顺序访问——从给定元素开始, 要获取另一个元素, 则必须通过自增或自减迭代器的操作遍历这两个元素之间的所有元素。list 容器支持在容器的任何位置实现快速插入 (或删

除) 运算。新元素的插入不会影响 list 中的其他元素。插入元素时, 迭代器保持有效; 删除元素时, 只有指向该元素的迭代器失效。

priority_queue (优先级队列) 一种顺序容器适配器。在这种队列中, 新元素不是在队列尾部插入, 而是根据指定的优先级级别插入。默认情况下, 元素的优先级由元素类型的小于操作符决定。

queue (队列) 一种顺序容器适配器。在这种队列中, 保证只在队尾插入新元素, 而且只在队首删除元素。

sequential container (顺序容器) 以有序集合的方式存储单一类型对象的类型。顺序容器中的元素可通过下标访问。

stack (栈) 一种顺序容器适配器, 这种类型只能在一端插入和删除元素。

vector (向量) 一种顺序容器。vector 中的元素通过其位置下标访问。可通过调用 push_back 或 insert 函数在 vector 中添加元素。在 vector 中添加元素可能会导致重新为容器分配内存空间, 也可能使所有的迭代器失效。在 vector 容器中间添加 (或删除) 元素将使所有指向插入 (或删除) 点后面的元素的迭代器失效。

353
354

第 10 章

关联容器

目录

10.1 引言: pair 类型	306
10.2 关联容器	308
10.3 map 类型	309
10.4 set 类型	319
10.5 multimap 和 multiset 类型	322
10.6 容器的综合应用: 文本查询程序	325
小结	332
术语	332

本章将继续介绍标准库容器类型的另一项内容——关联容器。关联容器和顺序容器的本质差别在于：关联容器通过键（key）存储和读取元素，而顺序容器则通过元素在容器中的位置顺序存储和访问元素。

虽然关联容器的大部分行为与顺序容器相同，但其独特之处在于支持键的使用。本章涵盖了关联容器的相关内容，并完善和扩展了一个使用顺序容器和关联容器的例子。

关联容器（associative container）支持通过键来高效地查找和读取元素。两个基本的关联容器类型是 map 和 set（表 10-1）。map 的元素以键-值（key-value）对的形式组织：键用作元素在 map 中的索引，而值则表示所存储和读取的数据。set 仅包含一个键，并有效地支持关于某个键是否存在的查询。

表 10-1 关联容器类型	
map	关联数组：元素通过键来存储和读取
set	大小可变的集合，支持通过键实现的快速读取
multimap	支持同一个键多次出现的 map 类型
multiset	支持同一个键多次出现的 set 类型

一般来说，如果希望有效地存储不同值的集合，那么使用 set 容器比较合适，而 map 容器则更适用于需要存储（乃至修改）每个键所关联的值的的情况。在做某种文本处理时，可使用 set 保存要忽略的单词。而字典则是 map 的一种很好的应用：单词本身是键，而它的解释说明则是值。set 和 map 类型的对象所包含的元素都具有不同的键，不允许为同一个键添加第二个元素。如果一个键必须对应多个实例，则需使用 multimap 或 multiset 类型（表 10-1），这两种类型允许多个元素拥有相同的键。

关联容器支持很多顺序容器也提供的相同操作，此外，还提供管理或使用键的特殊操作。下面的小节将详细讨论关联容器类型及其操作，最后以一个用容器实现的小型文本查询程序结束本章。

10.1 引言：pair 类型

在开始介绍关联容器之前，必须先了解一种与之相关的简单的标准库类型——pair 类型（表 10-2），该类型在 utility 头文件中定义。

表 10-2 pair 类型提供的操作	
pair<T1, T2> p1;	创建一个空的 pair 对象，它的两个元素分别是 T1 和 T2 类型，采用值初始化（3.3.1 节）
pair<T1, T2> p1(v1, v2);	创建一个 pair 对象，它的两个元素分别是 T1 和 T2 类型，其中 first 成员初始化为 v1，而 second 成员初始化为 v2
make_pair(v1, v2)	以 v1 和 v2 值创建一个新的 pair 对象，其元素类型分别是 v1 和 v2 的类型
p1 < p2	两个 pair 对象之间的小于运算，其定义遵循字典次序：如果 p1.first<p2.first 或者 !(p2.first<p1.first)&& p1.second<p2.second，则返回 true
p1 == p2	如果两个 pair 对象的 first 和 second 成员依次相等，则这两个对象相等。该运算使用其元素的==操作符
p.first	返回 p 中名为 first 的（公有）数据成员
p.second	返回 p 的名为 second 的（公有）数据成员

1. pair 的创建和初始化

pair 包含两个数据值。与容器一样，pair 也是一种模板类型。但又与之前介绍的容器不同，

在创建 pair 对象时，必须提供两个类型名：pair 对象所包含的两个数据成员各自对应的类型名字，这两个类型不必相同。

```
pair<string, string> anon;           // holds two strings
pair<string, int> word_count;        // holds a string and an int
pair<string, vector<int> > line;     // holds string and vector<int>
```

356

如果在创建 pair 对象时不提供初始化式，则调用默认构造函数对其成员采用值初始化。于是，anon 是包含两个空 string 类型成员的 pair 对象，line 则存储一个空的 string 类型对象和一个空的 vector 类型对象。word_count 中的 int 型成员获得 0 值，而 string 成员则初始化为空 string 对象。

当然，也可在定义时为每个成员提供初始化式：

```
pair<string, string> author("James", "Joyce");
```

创建一个名为 author 的 pair 对象，它的两个成员都是 string 类型，分别初始化为字符串 "James" 和 "Joyce"。

pair 类型的使用相当繁琐，因此，如果需要定义多个相同的 pair 类型对象，可考虑利用 typedef 简化其声明（2.6 节）：

```
typedef pair<string, string> Author;
Author proust("Marcel", "Proust");
Author joyce("James", "Joyce");
```

2. pair 对象的操作

与其他标准库类型不同，对于 pair 类，可以直接访问其数据成员：其成员都是公有的，分别命名为 first 和 second。只需使用普通的点操作符（1.5.2 节）——成员访问标志即可访问其成员：

```
string firstBook;
// access and test the data members of the pair
if (author.first == "James" && author.second == "Joyce")
    firstBook = "Stephen Hero";
```

标准库只为 pair 类型定义了表 10-2 所列出的数量有限的操作。

357

3. 生成新的 pair 对象

除了构造函数，标准库还定义了一个 make_pair 函数，由传递给它的两个实参生成一个新的 pair 对象。可如下使用该函数来创建新的 pair 对象，并赋给已存在的 pair 对象：

```
pair<string, string> next_auth;
string first, last;
while (cin >> first >> last) {
    // generate a pair from first and last
    next_auth = make_pair(first, last);
    // process next_auth...
}
```

这个循环处理一系列的作者信息：在 while 循环条件中读入的作者名字作为实参，调用 make_pair 函数生成一个新的 pair 对象。此操作等价于下面更复杂的操作：

```
// use pair constructor to make first and last into a pair
next_auth = pair<string, string>(first, last);
```

由于 pair 的数据成员是公有的，因而可如下直接地读取输入：

```
pair<string, string> next_auth;
// read directly into the members of next_auth
while (cin >> next_auth.first >> next_auth.second) {
    // process next_auth...
}
```

习题

习题 10.1 编写程序读入一系列 string 和 int 型数据，将每一组存储在一个 pair 对象中，然后将这些 pair 对象存储在 vector 容器里。

习题 10.2 在前一题中，至少可使用三种方法创建 pair 对象。编写三个版本的程序，分别采用不同的方法来创建 pair 对象。你认为哪一种方法更易于编写和理解，为什么？

10.2 关联容器

关联容器共享大部分——但并非全部——的顺序容器操作。关联容器不提供 front、push_front、pop_front、back、push_back 以及 pop_back 操作。

358

顺序容器和关联容器公共的操作包括下面的几种：

- 表 9-2 描述的前三种构造函数：

```
C<T> c;           // creates an empty container
// c2 must be same type as c1
C<T> c1(c2);      // copies elements from c2 into c1
// b and e are iterators denoting a sequence
C<T> c(b, e);     // copies elements from the sequence into c
```

关联容器不能通过容器大小来定义，因为这样的话就无法知道键所对应的值是什么。

- 9.3.4 节中描述的关系运算。
- 表 9-6 列出的 begin、end、rbegin 和 rend 操作。
- 表 9-5 列出的类型别名 (typedef)。注意，对于 map 容器，value_type 并非元素的类型，而是描述键及其关联值类型的 pair 类型。10.3.2 节将详细解释 map 中的类型别名。
- 表 9-11 中描述的 swap 和赋值操作。但关联容器不提供 assign 函数。
- 表 9-10 列出的 clear 和 erase 操作，但关联容器的 erase 运算返回 void 类型。
- 表 9-8 列出的关于容器大小的操作。但 resize 函数不能用于关联容器。

根据键排列元素

除了上述列出的操作之外，关联容器还提供了其他的操作。而对于顺序容器也提供的相同操作，关联容器也重新定义了这些操作的含义或返回类型，其中的差别在于关联容器中使用了键。

注解

“容器元素根据键的次序排列”这一事实就是一个重要的结论：在迭代遍历关联容器时，我们可确保按键的顺序访问元素，而与元素在容器中的存放位置完全无关。

习题

习题 10.3 描述关联容器和顺序容器的差别。

习题 10.4 举例说明 list、vector、deque、map 以及 set 类型分别适用的情况。

359

10.3 map 类型

map 是键-值对的集合。map 类型通常可理解为**关联数组**：可使用键作为下标来获取一个值，正如内置数组类型一样。而关联的本质在于元素的值与某个特定的键相关联，而并非通过元素在数组中的位置来获取。

10.3.1 map 对象的定义

要使用 map 对象，则必须包含 map 头文件。在定义 map 对象时，必须分别指明键和值的类型（表 10-3）：

```
// count number of times each word occurs in the input
map<string, int> word_count; // empty map from string to int
```

这个语句定义了一个名为 word_count 的 map 对象，由 string 类型的键索引，关联的值则为 int 型。

表 10-3 map 的构造函数

map<k, v> m;	创建一个名为 m 的空 map 对象，其键和值的类型分别为 k 和 v
map<k, v> m(m2);	创建 m2 的副本 m，m 与 m2 必须有相同的键类型和值类型
map<k, v> m(b, e);	创建 map 类型的对象 m，存储迭代器 b 和 e 标记的范围内所有元素的副本。元素的类型必须能转换为 pair<const k, v>

键类型的约束

在使用关联容器时，它的键不但有一个类型，而且还有一个相关的比较函数。默认情况下，标准库使用键类型定义的 < 操作符来实现键的比较。15.8.3 节将介绍如何重写默认的操作符，并提供自定义的操作符函数。

所用的比较函数必须在键类型上定义**严格弱排序**（strict weak ordering）。所谓的严格弱排序可理解为键类型数据上的“小于”关系，虽然实际上可以选择将比较函数设计得更复杂。但无论这样的比较函数如何定义，当用于一个键与自身的比较时，肯定会导致 false 结果。此外，在比较两个键时，不能出现相互“小于”的情况，而且，如果 k1 “小于” k2，k2 “小于” k3，则 k1 必然“小于” k3。对于两个键，如果它们相互之间都不存在“小于”关系，则容器将之视为相同的键。用做 map 对象的键时，可使用任意一个键值来访问相应的元素。



在实际应用中，键类型必须定义 < 操作符，而且该操作符应能“正确地工作”，这一点很重要。

360

例如，在书店问题中，可增加一个名为 ISBN 的类型，封装与国际标准图书编号 (ISBN) 相关的规则。在我们的实现中，国际标准图书编号是 string 类型，可做比较运算以确定编号之间的大小关系。因此，ISBN 类型可以支持 < 运算。假设我们已经定义了这样的类型，则可定义一个 map 容器对象，以便高效地查找书店中存放的某本书。

```
map<ISBN, Sales_item> bookstore;
```

该语句定义了一个名为 bookstore 的 map 对象，以 ISBN 类型的对象为索引，其所有元素都存储了一个关联的 Sales_item 类类型实例。

注解

对于键类型，唯一的约束就是必须支持 < 操作符，至于是否支持其他的关系或相等运算，则不作要求。

习题

- 习题 10.5 定义一个 map 对象，将单词与一个 list 对象关联起来，该 list 对象存储对应的单词可能出现的行号。
- 习题 10.6 可否定义一个 map 对象以 vector<int>::iterator 为键关联 int 型对象？如果以 list<int>::iterator 关联 int 型对象呢？或者，以 pair<int, string> 关联 int？对于每种情况，如果不允许，请解释其原因。

10.3.2 map 定义的类型

map 对象的元素是键-值对，也即每个元素包含两个部分：键以及由键关联的值。map 的 value_type 就反映了这个事实（表 10-4）。该类型比前面介绍的容器所使用的元素类型要复杂得多：value_type 是存储元素的键以及值的 pair 类型，而且键为 const。例如，word_count 数组的 value_type 为 pair<const string, int> 类型。

表 10-4 map 类定义的类型	
map<K, V>::key_type	在 map 容器中，用做索引的键的类型
map<K, V>::mapped_type	在 map 容器中，键所关联的值的类型
map<K, V>::value_type	一个 pair 类型，它的 first 元素具有 const map<K, V>::key_type 类型，而 second 元素则为 map<K, V>::mapped_type 类型

361

注解

在学习 map 的接口时，需谨记 value_type 是 pair 类型，它的值成员可以修改，但键成员不能修改。

1. map 迭代器进行解引用将产生 pair 类型的对象

对迭代器进行解引用时，将获得一个引用，指向容器中一个 value_type 类型的值。对于 map 容器，其 value_type 是 pair 类型：

```
// get an iterator to an element in word_count
map<string, int>::iterator map_it = word_count.begin();
// *map_it is a reference to a pair<const string, int> object
```

```

cout << map_it->first;           // prints the key for this element
cout << " " << map_it->second;   // prints the value of the element
map_it->first = "new key";       // error: key is const
++map_it->second;               // ok: we can change value through an iterator

```

对迭代器进行解引用将获得一个 pair 对象，它的 first 成员存放键，为 const，而 second 成员则存放值。

2. map 容器额外定义的类型别名 (typedef)

map 类额外定义了两种类型：key_type 和 mapped_type，以获得键或值的类型。对于 word_count，其 key_type 是 string 类型，而 mapped_type 则是 int 型。如同顺序容器 (9.3.1 节) 一样，可使用作用域操作符 (scope operator) 来获取类型成员，如 map<string,int>::key_type。

习题

习题 10.7 对于以 int 型对象为索引关联 vector<int> 型对象的 map 容器，它的 mapped_type、key_type 和 value_type 分别是什么？

习题 10.8 编写一个表达式，使用 map 的迭代器给其元素赋值。

10.3.3 给 map 添加元素

定义了 map 容器后，下一步工作就是在容器中添加键-值元素对。该项工作可使用 insert 成员实现；或者，先用下标操作符获取元素，然后给获取的元素赋值。在这两种情况下，一个给定的键只能对应于一个元素这一事实影响了这些操作的行为。

10.3.4 使用下标访问 map 对象

如下编写程序时：

```

map <string, int> word_count; // empty map
// insert default initialized element with key Anna; then assign 1 to its value
word_count["Anna"] = 1;

```

将发生以下事情：

- (1) 在 word_count 中查找键为 Anna 的元素，没有找到。
- (2) 将一个新的键-值对插入到 word_count 中。它的键是 const string 类型的对象，保存 Anna。而它的值则采用值初始化，这就意味着在本例中值为 0。
- (3) 将这个新的键-值对插入到 word_count 中。
- (4) 读取新插入的元素，并将它的值赋为 1。



使用下标访问 map 与使用下标访问数组或 vector 的行为截然不同：用下标访问不存在的元素将导致在 map 容器中添加一个新的元素，它的键即为该下标值。

如同其他下标操作符一样，map 的下标也使用索引（其实就是键）来获取该键所关联的值。如果该键已在容器中，则 map 的下标运算与 vector 的下标运算行为相同：返回该键所关联的值。

只有在所查找的键不存在时, map 容器才为该键创建一个新的元素, 并将它插入到此 map 对象中。此时, 所关联的值采用值初始化: 类类型的元素用默认构造函数初始化, 而内置类型的元素则初始化为 0。

1. 下标操作符返回值的使用

通常来说, 下标操作符返回左值。它返回的左值是特定键所关联的值。可如下读或写元素:

```
cout << word_count["Anna"];    // fetch element indexed by Anna; prints 1
++word_count["Anna"];          // fetch the element and add one to it
cout << word_count["Anna"];    // fetch the element and print it; prints 2
```



有别于 vector 或 string 类型, map 下标操作符返回的类型与对 map 迭代器进行解引用获得的类型不相同。

显然, map 迭代器返回 value_type 类型的值——包含 const key_type 和 mapped_type 类型成员的 pair 对象; 下标操作符则返回一个 mapped_type 类型的值。

2. 下标行为的编程意义

对于 map 容器, 如果下标所表示的键在容器中不存在, 则添加新元素, 这一特性可使程序惊人地简练:

```
// count number of times each word occurs in the input
map<string, int> word_count; // empty map from string to int
string word;
while (cin >> word)
    ++word_count[word];
```

363

这段程序创建一个 map 对象, 用来记录每个单词出现的次数。while 循环每次从标准输入读取一个单词。如果这是一个新的单词, 则在 word_count 中添加以该单词为索引的新元素。如果读入的单词已在 map 对象中, 则将它所对应的值加 1。

其中最有趣的是, 在单词第一次出现时, 会在 word_count 中创建并插入一个以该单词为索引的新元素, 同时将它的值初始化为 0。然后其值立即加 1, 所以每次在 map 中添加新元素时, 所统计的出现次数正好从 1 开始。

习题

习题 10.9 编写程序统计并输出所读入的单词出现的次数。

习题 10.10 解释下面程序的功能:

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

比较上一程序和下面程序的行为

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

习题 10.11 哪些类型可用做 map 容器对象的下标? 下标操作符返回的又是什么类型? 给出一个具体例子说明, 即定义一个 map 对象, 指出哪些类型可用作其下标, 以及下标操作符返回的类型。

10.3.5 map::insert 的使用

map 容器的 insert 成员（表 10-5）与顺序容器（9.3.3 节）的类似，但有一点要注意：必须考虑键的作用。键影响了实参的类型：插入单个元素的 insert 版本使用键-值 pair 类型的参数。类似地，对于参数为一对迭代器的版本，迭代器必须指向键-值 pair 类型的元素。另一个差别则是：map 容器的接受单个值的 insert 版本的返回类型。本节的后续部分将详细阐述这一特性。

表 10-5 map 容器提供的 insert 操作

m.insert(e)	e 是一个用在 m 上的 value_type 类型的值。如果键(e.first)不在 m 中，则插入一个值为 e.second 的新元素；如果该键在 m 中已存在，则保持 m 不变。该函数返回一个 pair 类型对象，包含指向键为 e.first 的元素的 map 迭代器，以及一个 bool 类型的对象，表示是否插入了该元素
m.insert(beg, end)	beg 和 end 是标记元素范围的迭代器，其中的元素必须为 m.value_type 类型的键-值对。对于该范围内的所有元素，如果它的键在 m 中不存在，则将该键及其关联的值插入到 m。返回 void 类型
m.insert(iter, e)	e 是一个用在 m 上的 value_type 类型的值。如果键(e.first)不在 m 中，则创建新元素，并以迭代器 iter 为起点搜索新元素存储的位置。返回一个迭代器，指向 m 中具有给定键的元素

1. 以 insert 代替下标运算

使用下标给 map 容器添加新元素时，元素的值部分将采用值初始化。通常，我们会立即为其赋值，其实就是对同一个对象进行初始化并赋值。而插入元素的另一个方法是：直接使用 insert 成员，其语法更紧凑：

```
// if Anna not already in word_count, inserts new element with value 1
word_count.insert(map<string, int>::value_type("Anna", 1));
```

这个 insert 函数版本的实参：

```
map<string, int>::value_type("Anna", 1)
```

是一个新创建的 pair 对象，将直接插入到 map 容器中。谨记 value_type 是 pair<const K, V> 类型的同义词，K 为键类型，而 V 是键所关联的值的类型。insert 的实参创建了一个适当的 pair 类型新对象，该对象将插入到 map 容器。在添加新的 map 元素时，使用 insert 成员可避免使用下标操作符所带来的副作用：不必要的初始化。

传递给 insert 的实参相当笨拙。可用两种方法简化：使用 make_pair

```
word_count.insert(make_pair("Anna", 1));
```

或使用 typedef：

```
typedef map<string, int>::value_type valType;
word_count.insert(valType("Anna", 1));
```

这两种方法都使调用变得简单，提高了程序的可读性。

2. 检测 insert 的返回值

map 对象中一个给定键只对应一个元素。如果试图插入的元素所对应的键已在容器中，则 insert 将不做任何操作。含有一个或一对迭代器形参的 insert 函数版本并不说明是否有或有多

少个元素插入到容器中。

但是, 带有一个键-值 pair 形参的 insert 版本将返回一个值: 包含一个迭代器和一个 bool 值的 pair 对象, 其中迭代器指向 map 中具有相应键的元素, 而 bool 值则表示是否插入了该元素。如果该键已在容器中, 则其关联的值保持不变, 返回的 bool 值为 false; 如果该键不在容器中, 则插入新元素, 且 bool 值为 true。在这两种情况下, 迭代器都将指向具有给定键的元素。下面是使用 insert 重写的单词统计程序:

365

```
// count number of times each word occurs in the input
map<string, int> word_count; // empty map from string to int
string word;
while (cin >> word) {
    // inserts element with key equal to word and value 1;
    // if word already in word_count, insert does nothing
    pair<map<string, int>::iterator, bool> ret =
        word_count.insert(make_pair(word, 1));
    if (!ret.second) // word already in word_count
        ++ret.first->second; // increment counter
}
```

对于每个单词, 都尝试 insert 它, 并将它的值赋为 1。if 语句检测 insert 函数返回值中的 bool 值。如果该值为 false, 则表示没有做插入操作, 按 word 索引的元素已在 word_count 中存在。此时, 将该元素所关联的值加 1。

3. 语法展开

ret 的定义和自增运算可能比较难解释:

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

首先, 应该很容易看出我们定义的是一个 pair 对象, 它的 second 成员为 bool 类型。而它的 first 成员则比较难理解, 这是 map<string, int> 容器所定义的迭代器类型。

根据操作符的优先级次序 (5.10.1 节), 可如下从添加圆括号开始理解自增操作:

```
++((ret.first)->second); // equivalent expression
```

下面对这个表达式一步步地展开解释:

- ret 存储 insert 函数返回的 pair 对象。该 pair 对象的 first 成员是一个 map 迭代器, 指向插入的键。
- ret.first 从 insert 返回的 pair 对象中获取 map 迭代器。
- ret.first->second 对该迭代器进行解引用, 获得一个 value_type 类型的对象。这个对象同样是 pair 类型的, 它的 second 成员即为我们所添加的元素的值部分。
- ++ret.first->second 实现该值的自增运算。

归结起来, 这个自增语句获取指向按 word 索引的元素的迭代器, 并将该元素的值加 1。

366

习题

习题 10.12 重写 10.3.4 节习题的单词统计程序, 要求使用 insert 函数代替下标运算。你认为哪个程序更容易编写和阅读? 请解释原因。

习题 10.13 假设有 `map<string, vector<int>>` 类型，指出在该容器中插入一个元素的 `insert` 函数应具有的参数类型和返回值类型。

10.3.6 查找并读取 `map` 中的元素

下标操作符给出了读取一个值的最简单方法：

```
map<string,int> word_count;
int occurs = word_count["foobar"];
```

但是，使用下标存在一个很危险的副作用：如果该键不在 `map` 容器中，那么下标操作会插入一个具有该键的新元素。

这样的行为是否正确取决于程序员的意愿。在这个例子中，如果 "foobar" 不存在，则在 `map` 中插入具有该键的新元素，其关联的值为 0。在这种情况下，`occurs` 获得 0 值。

我们的单词统计程序的确是要通过下标引用一个不存在的元素来实现新元素的插入，并将其关联的值初始化为 0。然而，大多数情况下，我们只想知道某元素是否存在，而当该元素不存在时，并不想做插入运算。对于这种应用，则不能使用下标操作符来判断元素是否存在。

`map` 容器提供了两个操作：`count` 和 `find`，用于检查某个键是否存在而不会插入该键（见表 10-6）。

表 10-6 不修改 `map` 对象的查询操作

<code>m.count(k)</code>	返回 <code>m</code> 中 <code>k</code> 的出现次数
<code>m.find(k)</code>	如果 <code>m</code> 容器中存在按 <code>k</code> 索引的元素，则返回指向该元素的迭代器。如果不存在，则返回超出末端迭代器（3.4 节）

1. 使用 `count` 检查 `map` 对象中某键是否存在

对于 `map` 对象，`count` 成员的返回值只能是 0 或 1。`map` 容器只允许一个键对应一个实例，所以 `count` 可有效地表明一个键是否存在。而对于 `multimap` 容器，`count` 的返回值将有更多的用途，相关内容将会在 10.5 节中介绍。如果返回值非 0，则可以使用下标操作符来获取该键所关联的值，而不必担心这样做会在 `map` 中插入新元素：

```
int occurs = 0;
if (word_count.count("foobar"))
    occurs = word_count["foobar"];
```

当然，在执行 `count` 后再使用下标操作符，实际上是对元素作了两次查找。如果希望当元素存在时就使用它，则应该用 `find` 操作。

2. 读取元素而又不插入该元素

`find` 操作返回指向元素的迭代器，如果元素不存在，则返回 `end` 迭代器：

```
int occurs = 0;
map<string,int>::iterator it = word_count.find("foobar");
if (it != word_count.end())
    occurs = it->second;
```

如果希望当具有指定键的元素存在时，就获取该元素的引用，否则就不在容器中创建新的元素，那么应该使用 `find`。

习题

习题 10.14 map 容器的 count 和 find 运算有何区别?

习题 10.15 你认为 count 适合用于解决哪一类问题? 而 find 呢?

习题 10.16 定义并初始化一个变量, 用来存储调用键为 string、值为 vector<int>的 map 对象的 find 函数的返回结果。

10.3.7 从 map 对象中删除元素

从 map 容器中删除元素的 erase 操作有三种变化形式 (表 10-7)。与顺序容器一样, 可向 erase 传递一个或一对迭代器, 来删除单个元素或一段范围内的元素。其删除功能类似于顺序容器, 但有一点不同: map 容器的 erase 操作返回 void, 而顺序容器的 erase 操作则返回一个迭代器, 指向被删除元素后面的元素。

表 10-7 从 map 对象中删除元素

m.erase(k)	删除 m 中键为 k 的元素。返回 size_type 类型的值, 表示删除的元素个数
m.erase(p)	从 m 中删除迭代器 p 所指向的元素。p 必须指向 m 中确实存在的元素, 而且不能等于 m.end()。返回 void 类型
m.erase(b, e)	从 m 中删除一段范围内的元素, 该范围由迭代器对 b 和 e 标记。b 和 e 必须标记 m 中的一段有效范围: 即 b 和 e 都必须指向 m 中的元素或最后一个元素的下一位置。而且, b 和 e 要么相等 (此时删除的范围为空), 要么 b 所指向的元素必须出现在 e 所指向的元素之前。返回 void 类型

除此之外, map 类型还提供了一种额外的 erase 操作, 其参数是 key_type 类型的值, 如果拥有该键的元素存在, 则删除该元素。对于单词统计程序, 可使用这个版本的 erase 函数来删除 word_count 中指定的单词, 然后输出被删除的单词:

```
// erase of a key returns number of elements removed
if (word_count.erase(removal_word))
    cout << "ok: " << removal_word << " removed\n";
else cout << "oops: " << removal_word << " not found!\n";
```

erase 函数返回被删除元素的个数。对于 map 容器, 该值必然是 0 或 1。如果返回 0, 则表示欲删除的元素在 map 中不存在。

10.3.8 map 对象的迭代遍历

与其他容器一样, map 同样提供 begin 和 end 运算, 以生成用于遍历整个容器的迭代器。例如, 可如下将 10.3.4 节建立的 map 容器 word_count 的内容输出:

```
// get iterator positioned on the first element
map<string, int>::const_iterator
    map_it = word_count.begin();
// for each element in the map
while (map_it != word_count.end()) {
    // print the element key, value pairs
    cout << map_it->first << " occurs "
        << map_it->second << " times" << endl;
    ++map_it; // increment iterator to denote the next element
}
```

while 循环的条件判断以及循环体中迭代器的自增都与输出 vector 或 string 容器内容的程序非常相像。首先，初始化 map_it 迭代器，使之指向 word_count 的第一个元素。只要该迭代器不等于 end 的值，就输出当前元素并给迭代器加 1。这段程序的循环体要比前面类似的程序更加复杂，原因在于对于 map 的每个元素都必须分别输出它的键和值。



这个单词统计程序依据字典顺序输出单词。在使用迭代器遍历 map 容器时，迭代器指向的元素按键的升序排列。

10.3.9 “单词转换” map 对象

下面的程序说明如何创建、查找和迭代遍历一个 map 对象，我们将以此结束本节内容。这个程序求解的问题是：给出一个 string 对象，把它转换为另一个 string 对象。本程序的输入是两个文件。第一个文件包括了若干单词对，每对的第一个单词将出现在输入的字符串中，而第二个单词则是用于输出。本质上，这个文件提供的是单词转换的集合——在遇到第一个单词时，应该将之替换为第二个单词。第二个文件则提供了需要转换的文本。如果单词转换文件的内容是：

369

```
'em      them
cuz      because
gratz    grateful
i        I
nah      no
pos      supposed
sez      said
tanx     thanks
wuz      was
```

而要转换的文本是：

```
nah i sez tanx cuz i wuz pos to
not cuz i wuz gratz
```

则程序将产生如下输出结果：

```
no I said thanks because I was supposed to
not because I was grateful
```

单词转换程序

下面给出的解决方案是将单词转换文件的内容存储在一个 map 容器中，将被替换的单词作为键，而用作替换的单词则作为其相应的值。接着读取输入，查找输入的第一个单词是否对应有转换。若有，则实现转换，然后输出其转换后的单词，否则，直接输出原词。

该程序的主函数需有两个实参（7.2.6 节）：单词转换文件的名称以及需要转换的文件名。程序执行时，首先检查实参的个数。第一个实参 argv[0] 是命令名，而执行该程序所需要的两个文件名参数则分别存储在 argv[1] 及 argv[2] 中。

如果 argv[1] 的值合法，则调用 open_file 函数（8.4.3 节）打开单词转换文件。假设 open 操作成功，则读入“单词转换对”。以“转换对”中的第一个单词为键，第二个为值，调用 insert 函数在容器中插入新元素。while 循环结束后，trans_map 容器对象包含了转换输入文本所需的数据。而如果该实参有问题，则抛出异常（6.13 节）并结束程序的运行。

接下来, 调用 `open_file` 打开要转换的文件。第二个 `while` 循环使用 `getline` 函数逐行读入该文件。因为程序每次读入一行, 从而可在输出文件的相同位置进行换行。然后在内嵌的 `while` 循环中使用 `istringstream` 将每一行中的单词提取出来。这部分程序与 8.5 节的程序框架类似。

内层的 `while` 循环检查每个单词, 判断它是否在转换的 `map` 对象中出现。如果在, 则从该 `map` 对象中取出对应的值替代此单词。最后, 无论是否做了转换, 都输出该单词。同时, 程序使用 `bool` 值 `firstword` 判断是否需要输出空格。如果当前处理的是这一行的第一个单词, 则无须输出空格。

370

```

/*
 * A program to transform words.
 * Takes two arguments: The first is name of the word transformation file
 *                       The second is name of the input to transform
 */
int main(int argc, char **argv)
{
    // map to hold the word transformation pairs:
    // key is the word to look for in the input; value is word to use in the output
    map<string, string> trans_map;
    string key, value;
    if (argc != 3)
        throw runtime_error("wrong number of arguments");
    // open transformation file and check that open succeeded
    ifstream map_file;
    if (!open_file(map_file, argv[1]))
        throw runtime_error("no transformation file");
    // read the transformation map and build the map
    while (map_file >> key >> value)
        trans_map.insert(make_pair(key, value));
    // ok, now we're ready to do the transformations
    // open the input file and check that the open succeeded
    ifstream input;
    if (!open_file(input, argv[2]))
        throw runtime_error("no input file");
    string line;    // hold each line from the input
    // read the text to transform it a line at a time
    while (getline(input, line)) {
        istringstream stream(line);    // read the line a word at a time
        string word;
        bool firstword = true;    // controls whether a space is printed
        while (stream >> word) {
            // ok: the actual mapwork, this part is the heart of the program
            map<string, string>::const_iterator map_it =
                trans_map.find(word);
            // if this word is in the transformation map
            if (map_it != trans_map.end())
                // replace it by the transformation value in the map
                word = map_it->second;
            if (firstword)
                firstword = false;
            else
                cout << " ";    // print space between words
            cout << word;
        }
        cout << endl;    // done with this line of input
    }
    return 0;
}

```

371

习题

习题 10.17 上述转换程序使用了 `find` 函数来查找单词：

```
map<string, string>::const_iterator map_it =
    trans_map.find(word);
```

你认为这个程序为什么要使用 `find` 函数？如果使用下标操作符又会怎么样？

习题 10.18 定义一个 `map` 对象，其元素的键是家族姓氏，而值则是存储该家族孩子名字的 `vector` 对象。为这个 `map` 容器输入至少六个条目。通过基于家族姓氏的查询检测你的程序，查询应输出该家族所有孩子的名字。

习题 10.19 把上一题的 `map` 对象再扩展一下，使其 `vector` 对象存储 `pair` 类型的对象，记录每个孩子的名字和生日。相应地修改程序，测试修改后的测试程序以检查所编写的 `map` 是否正确。

习题 10.20 列出至少三种可以使用 `map` 类型的应用。为每种应用定义 `map` 对象，并指出如何插入和读取元素。

10.4 set 类型

`map` 容器是键-值对的集合，好比以人名为键的地址和电话号码。相反地，`set` 容器只是单纯的键的集合。例如，某公司可能定义了一个名为 `bad_checks` 的 `set` 容器，用于记录曾经给本公司发空头支票的客户。当只想知道一个值是否存在时，使用 `set` 容器是最适合的。例如，在接收一张支票前，该公司可能想查询 `bad_checks` 对象，看看该客户的名字是否存在。

除了两种例外情况，`set` 容器支持大部分的 `map` 操作，包括下面几种：

- 10.2 节列出的所有通用的容器操作。
- 表 10-3 描述的构造函数。
- 表 10-5 描述的 `insert` 操作。
- 表 10-6 描述的 `count` 和 `find` 操作。
- 表 10-7 描述的 `erase` 操作。

两种例外包括：`set` 不支持下标操作符，而且没有定义 `mapped_type` 类型。在 `set` 容器中，`value_type` 不是 `pair` 类型，而是与 `key_type` 相同的类型。它们指的都是 `set` 中存储的元素类型。这一差别也体现了 `set` 存储的元素仅仅是键，而没有所关联的值。与 `map` 一样，`set` 容器存储的键也必须唯一，而且不能修改。

372

习题

习题 10.21 解释 `map` 和 `set` 容器的差别，以及它们各自适用的情况。

习题 10.22 解释 `set` 和 `list` 容器的差别，以及它们各自适用的情况。

10.4.1 set 容器的定义和使用

为了使用 `set` 容器，必须包含 `set` 头文件。`set` 支持的操作基本上与 `map` 提供的相同。

与 map 容器一样, set 容器的每个键都只能对应一个元素。以一段范围的元素初始化 set 对象, 或在 set 对象中插入一组元素时, 对于每个键, 事实上都只添加了一个元素:

```
// define a vector with 20 elements, holding two copies of each number from 0 to 9
vector<int> ivec;
for (vector<int>::size_type i = 0; i != 10; ++i) {
    ivec.push_back(i);
    ivec.push_back(i); // duplicate copies of each number
}
// iset holds unique elements from ivec
set<int> iset(ivec.begin(), ivec.end());
cout << ivec.size() << endl; // prints 20
cout << iset.size() << endl; // prints 10
```

首先创建了一个名为 ivec 的 int 型 vector 容器, 存储 20 个元素: 0~9 (包括 9) 中每个整数都出现了两次。然后用 ivec 中所有的元素初始化一个 int 型的 set 容器。则这个 set 容器仅有 10 个元素: ivec 中不相同的各个元素。

1. 在 set 中添加元素

可使用 insert 操作在 set 中添加元素:

```
set<string> set1; // empty set
set1.insert("the"); // set1 now has one element
set1.insert("and"); // set1 now has two elements
```

另一种用法是, 调用 insert 函数时, 提供一对迭代器实参, 插入其标记范围内所有的元素。该版本的 insert 函数类似于形参为一对迭代器的构造函数——对于一个键, 仅插入一个元素:

```
set<int> iset2; // empty set
iset2.insert(ivec.begin(), ivec.end()); // iset2 has 10 elements
```

与 map 容器的操作一样, 带有一个键参数的 insert 版本返回 pair 类型对象, 包含一个迭代器和一个 bool 值, 迭代器指向拥有该键的元素, 而 bool 值表明是否添加了元素。使用迭代器对的 insert 版本返回 void 类型。

2. 从 set 中获取元素

set 容器不提供下标操作符。为了通过键从 set 中获取元素, 可使用 find 运算。如果只需简单地判断某个元素是否存在, 同样可以使用 count 运算, 返回 set 中该键对应的元素个数。当然, 对于 set 容器, count 的返回值只能是 1 (该元素存在) 或 0 (该元素不存在):

```
iset.find(1) // returns iterator that refers to the element with key == 1
iset.find(11) // returns iterator == iset.end()
iset.count(1) // returns 1
iset.count(11) // returns 0
```

正如不能修改 map 中元素的键部分一样, set 中的键也为 const。在获得指向 set 中某元素的迭代器后, 只能对其做读操作, 而不能做写操作:

```
// set_it refers to the element with key == 1
set<int>::iterator set_it = iset.find(1);
*set_it = 11; // error: keys in a set are read-only
cout << *set_it << endl; // ok: can read the key
```

10.4.2 创建“单词排除”集

10.3.7 节的程序从 map 对象 word_count 中删除一个指定的单词。可将这个操作扩展为删除指定文件中所有的单词（即该文件记录的是排除集）。也即，我们的单词统计程序只对那些不在排除集中的单词进行统计。使用 set 和 map 容器，可以简单而直接地实现该功能：

```
void restricted_wc(ifstream &remove_file,
                  map<string, int> &word_count)
{
    set<string> excluded; // set to hold words we'll ignore
    string remove_word;
    while (remove_file >> remove_word)
        excluded.insert(remove_word);
    // read input and keep a count for words that aren't in the exclusion set
    string word;
    while (cin >> word)
        // increment counter only if the word is not in excluded
        if (!excluded.count(word))
            ++word_count[word];
}
```

这个程序类似 10.3.4 节的单词统计程序。其差别在于不需要费力地统计常见的单词。

该函数首先读取传递进来的文件，该文件列出了所有被排除的单词。读入这些单词并存储在一个名为 excluded 的 set 容器中。第一个 while 循环完成时，该 set 对象包含了输入文件中的所有单词。

接下来的程序类似原来的单词统计程序。关键的区别在于：在统计每个单词之前，先检查该单词是否出现在排除集中。第二个 while 循环里的 if 语句实现了该功能：

```
// increment counter only if the word is not in excluded
if (!excluded.count(word))
```

如果该单词出现在排除集 excluded 中，则调用 count 将返回 1，否则返回 0。对 count 的返回值做“非”运算，则当该 word 不在 excluded 中时，条件测试成功，此时，修改该单词在 map 中对应的值。

与单词统计程序原来的版本一样，需要使用下标操作符的性质：如果某键尚未在 map 容器中出现，则将该元素插入容器。所以语句

```
++word_count[word];
```

的效果是：如果 word 还没出现过，则将它插入到 word_count 中，并在插入元素后，将它关联的值初始化为 0。然后不管是否插入了新元素，相应元素的值都加 1。

习题

习题 10.23 编写程序将被排除的单词存储在 vector 对象中，而不是存储在 set 对象中。请指出使用 set 的好处。

习题 10.24 编写程序通过删除单词尾部的 's' 生成该单词的非复数版本。同时，建立一个单词排除集，用于识别以 's' 结尾、但这个结尾的 's' 又不能删除的单词。例如，放在该排除集

中的单词可能有 success 和 class。使用这个排除集编写程序，删除输入单词的复数后缀，而如果输入的是排除集中的单词，则保持该单词不变。

习题 10.25 定义一个 vector 容器，存储你在未来六个月里要阅读的书，再定义一个 set，用于记录你已经看过的书名。编写程序从 vector 中为你选择一本没有读过而现在要读的书。当它为你返回选中的书名后，应该将该书名放入记录已读书目的 set 中。如果实际上你把这本书放在一边没有看，则本程序应该支持从已读书目的 set 中删除该书的记录。在虚拟的六个月后，输出已读书目和还没有读的书目。

10.5 multimap 和 multiset 类型

map 和 set 容器中，一个键只能对应一个实例。而 multiset 和 multimap 类型则允许一个键对应多个实例。例如，在电话簿中，每个人可能有单独的电话号码列表。在作者的文章集中，每位作者可能有单独的文章标题列表。multimap 和 multiset 类型与相应的单元素版本具有相同的头文件定义：分别是 map 和 set 头文件。

375

multimap 和 multiset 所支持的操作分别与 map 和 set 的操作相同，只有一个例外：multimap 不支持下标运算。不能对 multimap 对象使用下标操作，因为在这类容器中，某个键可能对应多个值。为了顺应一个键可以对应多个值这一性质，map 和 multimap，或 set 和 multiset 中相同的操作都以不同的方式做出了一定的修改。在使用 multimap 或 multiset 时，对于某个键，必须做好处理多个值的准备，而非只有单一的值。

10.5.1 元素的添加和删除

表 10-5 描述的 insert 操作和表 10-7 描述的 erase 操作同样适用于 multimap 以及 multiset 容器，实现元素的添加和删除。

由于键不要求是唯一的，因此每次调用 insert 总会添加一个元素。例如，可如下定义一个 multimap 容器对象将作者映射到他们所写的书的书名上。这样的映射可为一个作者存储多个条目：

```
// adds first element with key Barth
authors.insert(make_pair(
    string("Barth, John"),
    string("Sot-Weed Factor")));
// ok: adds second element with key Barth
authors.insert(make_pair(
    string("Barth, John"),
    string("Lost in the Funhouse")));
```

带有一个键参数的 erase 版本将删除拥有该键的所有元素，并返回删除元素的个数。而带有一个或一对迭代器参数的版本只删除指定的元素，并返回 void 类型：

```
multimap<string, string> authors;
string search_item("Kazuo Ishiguro");
// erase all elements with this key; returns number of elements removed
multimap<string, string>::size_type cnt =
    authors.erase(search_item);
```

10.5.2 在 `multimap` 和 `multiset` 中查找元素

注意到，关联容器 `map` 和 `set` 的元素是按顺序存储的。而 `multimap` 和 `multiset` 也一样。因此，在 `multimap` 和 `multiset` 容器中，如果某个键对应多个实例，则这些实例在容器中将相邻存放。



迭代遍历 `multimap` 或 `multiset` 容器时，可保证依次返回特定键所关联的所有元素。

376

在 `map` 或 `set` 容器中查找一个元素很简单——该元素要么在要么不在容器中。但对于 `multimap` 或 `multiset`，该过程就复杂多了：某键对应的元素可能出现多次。例如，假设有作者与书名的映射，我们可能希望找到并输出某个作者写的所有书的书名。

事实证明，上述问题可用三种策略解决。而且三种策略都基于一个事实——在 `multimap` 中，同一个键所关联的元素必然相邻存放。

首先介绍第一种策略：仅使用前面介绍过的函数。但这种方法要编写比较多的代码，所以我们将继续探索更简洁的方法。

1. 使用 `find` 和 `count` 操作

使用 `find` 和 `count` 可有效地解决刚才的问题。`count` 函数求出某键出现的次数，而 `find` 操作则返回一个迭代器，指向第一个拥有正在查找的键的实例：

```
// author we'll look for
string search_item("Alain de Botton");
// how many entries are there for this author
typedef multimap<string, string>::size_type sz_type;
sz_type entries = authors.count(search_item);
// get iterator to the first entry for this author
multimap<string, string>::iterator iter =
    authors.find(search_item);
// loop through the number of entries there are for this author
for (sz_type cnt = 0; cnt != entries; ++cnt, ++iter)
    cout << iter->second << endl; // print each title
```

首先，调用 `count` 确定某作者所写的书籍数目，然后调用 `find` 获得指向第一个该键所关联的元素的迭代器。`for` 循环迭代的次数依赖于 `count` 返回的值。在特殊情况下，如果 `count` 返回 0 值，则该循环永不执行。

2. 与众不同的面向迭代器的解决方案

另一个更优雅简洁的方法是使用两个未曾见过的关联容器的操作：`lower_bound` 和 `upper_bound`。表 10-8 列出的这些操作适用于所有的关联容器，也可用于普通的 `map` 和 `set` 容器，但更常用于 `multimap` 和 `multiset`。所有这些操作都需要传递一个键，并返回一个迭代器。

表 10-8 返回迭代器的关联容器操作

<code>m.lower_bound(k)</code>	返回一个迭代器，指向键不小于 <code>k</code> 的第一个元素
<code>m.upper_bound(k)</code>	返回一个迭代器，指向键大于 <code>k</code> 的第一个元素
<code>m.equal_range(k)</code>	返回一个迭代器的 <code>pair</code> 对象 它的 <code>first</code> 成员等价于 <code>m.lower_bound(k)</code> 。而 <code>second</code> 成员则等价于 <code>m.upper_bound(k)</code>

在同一个键上调用 `lower_bound` 和 `upper_bound`，将产生一个迭代器范围（9.2.1 节），指示出该键所关联的所有元素。如果该键在容器中存在，则会获得两个不同的迭代器：`lower_bound` 返回的迭代器指向该键关联的第一个实例，而 `upper_bound` 返回的迭代器则指向最后一个实例的下一位置。如果该键不在 `multimap` 中，这两个操作将返回同一个迭代器，指向依据元素的排列顺序该键应该插入的位置。

377

当然，这些操作返回的也可能是容器自身的超出末端迭代器。如果所查找的元素拥有 `multimap` 容器中最大的键，那么在该键上调用 `upper_bound` 将返回超出末端迭代器。如果所查找的键不存在，而且比 `multimap` 容器中所有的键都大，则 `lower_bound` 也将返回超出末端迭代器。



`lower_bound` 返回的迭代器不一定指向拥有特定键的元素。如果该键不在容器中，则 `lower_bound` 返回在保持容器元素顺序的前提下该键应被插入的第一个位置。

使用这些操作，可如下重写程序：

```
// definitions of authors and search_item as above
// beg and end denote range of elements for this author
typedef multimap<string, string>::iterator authors_it;
authors_it beg = authors.lower_bound(search_item),
           end = authors.upper_bound(search_item);
// loop through the number of entries there are for this author
while (beg != end) {
    cout << beg->second << endl; // print each title
    ++beg;
}
```

这个程序实现的功能与前面使用 `count` 和 `find` 的程序相同，但任务的实现更直接。调用 `lower_bound` 定位 `beg` 迭代器，如果键 `search_item` 在容器中存在，则使 `beg` 指向第一个与之匹配的元素。如果容器中没有这样的元素，那么 `beg` 将指向第一个键比 `search_item` 大的元素。调用 `upper_bound` 设置 `end` 迭代器，使之指向拥有该键的最后一个元素的下一位置。



这两个操作不会说明键是否存在，其关键之处在于返回值给出了迭代器范围。

若该键没有关联的元素，则 `lower_bound` 和 `upper_bound` 返回相同的迭代器：都指向同一个元素或同时指向 `multimap` 的超出末端位置。它们都指向在保持容器元素顺序的前提下该键应被插入的位置。

如果该键所关联的元素存在，那么 `beg` 将指向满足条件的元素中的第一个。可对 `beg` 做自增运算遍历拥有该键的所有元素。当迭代器累加至 `end` 标志时，表示已遍历了所有这些元素。当 `beg` 等于 `end` 时，表示已访问所有与该键关联的元素。

假设这些迭代器标记某个范围，可使用同样的 `while` 循环遍历该范围。该循环执行 0 次或多次，输出指定作者所写的所有书的书名（如果有的话）。如果没有相关的元素，那么 `beg` 和 `end` 相等，循环永不执行。否则，不断累加 `beg` 将最终到达 `end`，在这个过程中可输出该作者所关联的记录。

378

3. `equal_range` 函数

事实上，解决上述问题更直接的方法是：调用 `equal_range` 函数来取代调用 `upper_bound`

和 `lower_bound` 函数。`equal_range` 函数返回存储一对迭代器的 `pair` 对象。如果该值存在，则 `pair` 对象中的第一个迭代器指向该键关联的第一个实例，第二个迭代器指向该键关联的最后一个实例的下一位置。如果找不到匹配的元素，则 `pair` 对象中的两个迭代器都将指向此键应该插入的位置。

使用 `equal_range` 函数再次修改程序：

```
// definitions of authors and search_item as above
// pos holds iterators that denote range of elements for this key
pair<authors_it, authors_it>
    pos = authors.equal_range(search_item);
// loop through the number of entries there are for this author
while (pos.first != pos.second) {
    cout << pos.first->second << endl; // print each title
    ++pos.first;
}
```

这个程序段与前面使用 `upper_bound` 和 `lower_bound` 的程序基本上是相同的。本程序不用局部变量 `beg` 和 `end` 来记录迭代器范围，而是直接使用 `equal_range` 返回的 `pair` 对象。该 `pair` 对象的 `first` 成员存储 `lower_bound` 函数返回的迭代器，而 `second` 成员则记录 `upper_bound` 函数返回的迭代器。

因此，本程序的 `pos.first` 等价于前一方法中的 `beg`，而 `pos.second` 等价于 `end`。

习题

- 习题 10.26 编写程序建立作者及其作品的 `multimap` 容器。使用 `find` 函数在 `multimap` 中查找元素，并调用 `erase` 将其删除。当所寻找的元素不存在时，确保你的程序依然能正确执行。
- 习题 10.27 重复上一题所编写的程序，但这一次要求使用 `equal_range` 函数获取迭代器，然后删除一段范围内的元素。
- 习题 10.28 沿用上一题中的 `multimap` 容器，编写程序以下面的格式按姓名首字母的顺序输出作者名字：

```
Author Names Beginning with 'A':
Author, book, book, ...
...
Author Names Beginning with 'B':
...
```

- 习题 10.29 解释本节最后一个程序的输出表达式使用的操作数 `pos.first->second` 的含义。

10.6 容器的综合应用：文本查询程序

我们将实现一个简单的文本查询程序来结束本章。

我们的程序将读取用户指定的任意文本文件，然后允许用户从该文件中查找单词。查询的结果是该单词出现的次数，并列出行号。如果某单词在同一行中多次出现，程序将只显示该行一次。行号按升序显示，即第 7 行应该在第 9 行之前输出，依此类推。

例如，以本章的内容作为文件输入，然后查找单词“element”。输出的前几行应为：

```
element occurs 125 times
(line 62) element with a given key.
(line 64) second element with the same key.
(line 153) element |==| operator.
(line 250) the element type.
(line 398) corresponding element.
```

后面省略了大约 120 行。

10.6.1 查询程序的设计

设计程序的一个良好习惯是首先将程序所涉及的操作列出来。明确需要提供的操作有助于建立需要的数据结构和实现这些行为。从需求出发，我们的程序需要支持如下任务：

(1) 它必须允许用户指明要处理的文件名字。程序将存储该文件的内容，以便输出每个单词所在的原始行。

(2) 它必须将每一行分解为各个单词，并记录每个单词所在的所有行。在输出行号时，应保证以升序输出，并且不重复。

(3) 对特定单词的查询将返回出现该单词的所有行的行号。

(4) 输出某单词所在的行文本时，程序必须能根据给定的行号从输入文件中获取相应的行。

1. 数据结构

我们将用一个简单的类 `TextQuery` 实现这个程序。再加上几种容器的配合使用，就可相当巧妙地满足上述要求。

(1) 使用一个 `vector<string>` 类型的对象存储整个输入文件的副本。输入文件的每一行是该 `vector` 对象的一个元素。因而，在希望输出某一行时，只需以行号为下标获取该行所在的元素即可。

(2) 将每个单词所在的行号存储在一个 `set` 容器对象中。使用 `set` 就可确保每行只有一个条目，而且行号将自动按升序排列。

(3) 使用一个 `map` 容器将每个单词与一个 `set` 容器对象关联起来，该 `set` 容器对象记录此单词所在的行号。

综上所述，我们定义的 `TextQuery` 类将有两个数据成员：储存输入文件的 `vector` 对象，以及一个 `map` 容器对象，该对象关联每个输入的单词以及记录该单词所在行号的 `set` 容器对象。

2. 操作

对于类还要求有良好的接口。然而，一个重要的设计策略首先要确定：查询函数需返回存储一组行号的 `set` 对象。这个返回类型应该如何设计呢？

事实上，查询的过程相当简单：使用下标访问 `map` 对象获取关联的 `set` 对象即可。唯一的问题是如何返回所找到的 `set` 对象。安全的设计方案是返回该 `set` 对象的副本。但如此一来，就意味着要复制 `set` 中的每一个元素。如果处理的是一个相当庞大的文件，则复制 `set` 对象的代价会非常昂贵。其他可行的方法包括：返回一个 `pair` 对象，存储一对指向 `set` 中元素的迭代器；或者返回 `set` 对象的 `const` 引用。为简单起见，我们在这里采用返回副本的方法，但注意：如果在实际应用中复制代价太大，需重新考虑其实现方法。

第一、第三和第四个任务是使用这个类的程序员将执行的动作。第二个任务则是类的内部任务。将这四个任务映射为类的成员函数，则类的接口需提供下列三个 public 函数：

- `read_file` 成员函数，其形参为一个 `ifstream&` 类型对象。该函数每次从文件中读入一行，并将它保存在 `vector` 容器中。输入完毕后，`read_file` 将创建关联每个单词及其所在行号的 `map` 容器。
- `run_query` 成员函数，其形参为一个 `string` 类型对象，返回一个 `set` 对象，该 `set` 对象包含出现该 `string` 对象的所有行的行号。
- `text_line` 成员函数，其形参为一个行号，返回输入文本中该行号对应的文本行。

381

无论 `run_query` 还是 `text_line` 都不会修改调用此函数的对象，因此，可将这两个操作定义为 `const` 成员函数（7.7.1 节）。

为实现 `read_file` 功能，还需定义两个 `private` 函数来读取输入文本和创建 `map` 容器：

- `store_file` 函数读入文件，并将文件内容存储在 `vector` 容器对象中。
- `build_map` 函数将每一行分解为各个单词，创建 `map` 容器对象，同时记录每个单词出现的行号。

10.6.2 TextQuery 类

经过前面的设计后，现在可以编写 `TextQuery` 类了：

```
class TextQuery {
public:
    // typedef to make declarations easier
    typedef std::vector<std::string>::size_type line_no;
    /* interface:
     *   read_file builds internal data structures for the given file
     *   run_query finds the given word and returns set of lines on which it appears
     *   text_line returns a requested line from the input file
     */
    void read_file(std::ifstream &is)
        { store_file(is); build_map(); }
    std::set<line_no> run_query(const std::string&) const;
    std::string text_line(line_no) const;
private:
    // utility functions used by read_file
    void store_file(std::ifstream&); // store input file
    void build_map(); // associated each word with a set of line numbers
    // remember the whole input file
    std::vector<std::string> lines_of_text;
    // map word to set of the lines on which it occurs
    std::map< std::string, std::set<line_no> > word_map;
};
```

这个类直接反映了我们的设计策略。唯一没有提及的是使用 `typedef` 为 `vector` 的 `size_type` 定义了一个别名。



基于 3.1 节所提及的原因，这个类的定义在引用标准库内容时必须完整地使用 `std::` 限定符。

`read_file` 函数在类的内部定义。该函数首先调用 `store_file` 读取并保存输入文件，然后

调用 `build_map` 创建关联单词与行号的 `map` 容器。该类的其他函数将在 10.6.4 节定义。首先，我们编写一个程序，使用这个类来解决文本查询问题。

习题

习题 10.30 `TextQuery` 类的成员函数仅使用了前面介绍过的内容。先别查看后面章节，请自己编写这些成员函数。提示：唯一棘手的是 `run_query` 函数在行号集合 `set` 为空时应返回什么值？解决方法是构造并返回一个新的（临时）`set` 对象。

10.6.3 `TextQuery` 类的使用

下面的主程序 `main` 使用 `TextQuery` 对象实现简单的用户查询会话。这段程序的主要工作是实现与用户的互动：提示输入下一个要查询的单词，然后调用 `print_results` 函数（将在下面定义）输出结果。

```
// program takes single argument specifying the file to query
int main(int argc, char **argv)
{
    // open the file from which user will query words
    ifstream infile;
    if (argc < 2 || !open_file(infile, argv[1])) {
        cerr << "No input file!" << endl;
        return EXIT_FAILURE;
    }
    TextQuery tq;
    tq.read_file(infile); // builds query map
    // iterate with the user: prompt for a word to find and print results
    // loop indefinitely; the loop exit is inside the while
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        cin >> s;
        // stop if hit eof on input or a 'q' is entered
        if (!cin || s == "q") break;
        // get the set of line numbers on which this word appears
        set<TextQuery::line_no> locs = tq.run_query(s);
        // print count and all occurrences, if any
        print_results(locs, s, tq);
    }
    return 0;
}
```

1. 引子

程序首先检查 `argv[1]` 是否合法，然后调用 `open_file` 函数（8.4.3 节）打开以 `main` 函数实参形式给出的文件。检查流以判断输入文件是否正确。如果不正确，就给出适当的提示信息并结束程序的运行，返回 `EXIT_FAILURE`（7.3.2 节）说明发生了错误。

一旦文件成功打开，建立支持查询的 `map` 容器就相当简单。定义一个局部变量 `tq` 来保存该输入文件和所关联的数据结构：

```
TextQuery tq;
tq.read_file(infile); // builds query map
```

tq 调用 read_file 操作, 并将由 open_file 打开的文件传递给此函数。

read_file 完成后, tq 存储了两个数据结构: 保存输入文件的 vector 对象, 以及关联单词和行号的 map 容器对象。map 容器为输入文件中的每个单词建立唯一的元素, 由每个单词关联的 set 容器记录了该单词出现的行号。

2. 实现查询

为了使用户在每次会话时都能查询多个单词, 我们将提示语句也置于 while 循环中:

```
// iterate with the user: prompt for a word to find and print results
// loop indefinitely; the loop exit is inside the while
while (true) {
    cout << "enter word to look for, or q to quit: ";
    string s;
    cin >> s;
    // stop if hit eof on input or a 'q' is entered
    if (!cin || s == "q") break;
    // get the set of line numbers on which this word appears
    set<TextQuery::line_no> locs = tq.run_query(s);
    // print count and all occurrences, if any
    print_results(locs, s, tq);
}
```

while 循环条件为布尔字面值 true, 这就意味着循环条件总是成立。在检查 cin 和读入 s 的值后, 由紧跟的 break 语句跳出循环。具体说来, 当 cin 遇到错误或文件结束, 或者用户输入 q 时, 循环结束。

每次要查找一个单词时, 访问 tq 获取记录该单词出现的行号的 set 对象。将 set 对象、要查找的单词和 TextQuery 对象作为参数传递给 print_results 函数, 该函数输出查询结果。

3. 输出结果

现在只剩下 print_results 函数的定义:

```
void print_results(const set<TextQuery::line_no>& locs,
                  const string& sought, const TextQuery &file)
{
    // if the word was found, then print count and all occurrences
    typedef set<TextQuery::line_no> line_nums;
    line_nums::size_type size = locs.size();
    cout << "\n" << sought << " occurs "
         << size << " "
         << make_plural(size, "time", "s") << endl;
    // print each line in which the word appeared
    line_nums::const_iterator it = locs.begin();
    for ( ; it != locs.end(); ++it) {
        cout << "\t(line "
             << (*it) + 1 << ") "
             << file.text_line(*it) << endl;
    }
}
```

函数首先使用 typedef 简化记录行号的 set 容器对象的使用。输出时, 首先给出查询到的匹配个数, 即 set 对象的大小。然后调用 make_plural (7.3.2 节), 根据 size 是否为 1 输出 “time” 或 “times”。

这段程序最复杂的部分是处理 `locs` 对象的 `for` 循环，用于输出找到该单词的行号。其唯一的微妙之处是记得将行号修改为更友好的形式输出。为了与 C++ 的容器和数组下标编号匹配，在储存文本时，我们以行号 0 存储第一行。但考虑到很多用户会默认第一行的行号为 1，所以输出行号时，相应地在所存储的行号上加 1 使之转换为更通用的形式。

习题

习题 10.31 如果没有找到要查询的单词，`main` 函数输出什么？

10.6.4 编写成员函数

现在给没有在类内定义的成员函数编写定义。

1. 存储输入文件

第一个任务是读入需要查询的文件。使用 `string` 和 `vector` 容器提供的操作，可以很简便地实现这个任务：

```
// read input file: store each line as element in lines_of_text
void TextQuery::store_file(ifstream &is)
{
    string textline;
    while (getline(is, textline))
        lines_of_text.push_back(textline);
}
```

385

由于我们希望每次存储文件的一行内容，因此使用 `getline` 读取输入，每读入一行就将其添加到名为 `lines_of_text` 的 `vector` 对象中。

2. 建立单词 `map` 容器

`vector` 容器中的每个元素就是一行文本。要建立一个从单词关联到行号的 `map` 容器，必须将每行分解为各个单词。再次使用 8.5 节描述的 `istringstream`：

```
// finds whitespace-separated words in the input vector
// and puts the word in word_map along with the line number
void TextQuery::build_map()
{
    // process each line from the input vector
    for (line_no line_num = 0;
         line_num != lines_of_text.size();
         ++line_num)
    {
        // we'll use line to read the text a word at a time
        istringstream line(lines_of_text[line_num]);
        string word;
        while (line >> word)
        {
            // add this line number to the set;
            // subscript will add word to the map if it's not already there
            word_map[word].insert(line_num);
        }
    }
}
```

`for` 循环以每次一行的迭代过程遍历 `lines_of_text`。首先将 `istringstream` 对象 `line` 与当前行绑定起来，然后使用 `istringstream` 的输入操作符读入该行中的每个单词。回顾此类输入操作符，与其他 `istream` 操作符一样，将忽略空白符号。因此，`while` 循环将 `line` 中以空白符

分隔的单词读取出来。

这个函数的结尾部分类似前面的单词统计程序。将 word 用做 map 容器的下标。如果 word 在 word_map 容器对象中不存在，那么下标操作符将该 word 添加到此容器中，并将其关联的值初始化为空的 set。不管是否添加了 word，下标运算都返回一个 set 对象。然后调用 insert 函数在该 set 对象中添加当前行号。如果某单词在同一行中重复出现，那么 insert 函数的调用将不做任何操作。

3. 支持查询

run_query 函数实现真正的查询功能：

```
set<TextQuery::line_no>
TextQuery::run_query(const string &query_word) const
{
    //Note: must use find and not subscript the map directly
    //to avoid adding words to word_map!
    map<string, set<line_no> >::const_iterator
        loc = word_map.find(query_word);
    if (loc == word_map.end())
        return set<line_no>(); // not found, return empty set
    else
        // fetch and return set of line numbers for this word
        return loc->second;
}
```

386

run_query 函数带有指向 const string 类型对象的引用参数，并以这个参数作为下标来访问 word_map 对象。假设成功找到这个 string，那么该函数返回关联此 string 的 set 对象，否则返回一个空的 set 对象。

4. run_query 返回值的使用

运行 run_query 函数后，将获得一组所查找的单词出现的行号。除了输出该单词的出现次数之外，还需要输出出现该单词的每一行。这就是 text_line 函数实现的功能：

```
string TextQuery::text_line(line_no line) const
{
    if (line < lines_of_text.size())
        return lines_of_text[line];
    throw std::out_of_range("line number out of range");
}
```

该函数带有一个行号参数，返回该行号所对应的输入文本行。由于上述代码使用了 TextQuery 类，因此不能直接输出（因为 lines_of_text 是私有的），应该首先检查我们要查询的行是否位于合法范围内。如果是，则返回相应的行，否则，抛出 out_of_range 异常。

习题

习题 10.32 重新实现文本查询程序，使用 vector 容器代替 set 对象来存储行号。注意，由于行以升序出现，因此只有在当前行号不是 vector 容器对象中的最后一个元素时，才能将新行号添加到 vector 中。这两种实现方法的性能特点 and 设计特点分别是什么？你觉得哪一种解决方法更好？为什么？

习题 10.33 TextQuery::text_line 函数为什么不检查它的参数是否为负数？

387

小结

关联容器的元素按键排序和访问。关联容器支持通过键高效地查找和读取元素。键的使用，使关联容器区别于顺序容器，顺序容器的元素是根据位置访问的。

`map` 和 `multimap` 类型存储的元素是键-值对。它们使用在 `utility` 头文件中定义的标准库 `pair` 类，来表示这些键-值对元素。对 `map` 或 `multimap` 迭代器进行解引用将获得 `pair` 类型的值。`pair` 对象的 `first` 成员是一个 `const` 键，而 `second` 成员则是该键所关联的值。`set` 和 `multiset` 类型则专门用于存储键。在 `map` 和 `set` 类型中，一个键只能关联一个元素。而 `multimap` 和 `multiset` 类型则允许多个元素拥有相同的键。

关联容器共享了顺序容器的许多操作。除此之外，关联容器还定义了一些新的操作，并对某些顺序容器同样提供的操作重新定义了其含义或返回类型，这些操作的差别体现了关联容器中键的使用。

关联容器的元素可用迭代器访问。标准库保证迭代器按照键的次序访问元素。`begin` 操作将获得拥有最小键的元素，对此迭代器做自增运算则可以按非降序依次访问各个元素。

术语

associative array (关联数组) 由键而不是位置来索引元素的数组。通常描述为：此类数组将键映射到其关联的值上。

associative container (关联容器) 存储对象集合的类型，支持通过键的高效查询。

key_type 关联容器定义的类型，表示该容器在存储或读取值时所使用的键的类型。对于 `map` 容器，`key_type` 是用于索引该容器的类型。对于 `set` 容器，`key_type` 与 `value_type` 相同。

map 定义关联数组的关联容器类型。与 `vector` 容器一样，`map` 也是类模板。但是，`map` 容器定义了两类型：键类型及其关联的值类型。在 `map` 中，每个键只能出现一次，并关联某一具体的值。对 `map` 容器的迭代器进行解引用将获得一个 `pair` 对象，该对象存储了一个 `const` 键和它所关联的值。

mapped_type `map` 或 `multimap` 容器定义的类型，表示在 `map` 容器中存储的值的类型。

multimap 类似 `map` 的关联容器。在 `multimap` 容器中，一个键可以出现多次。

multiset 只存储键的关联容器类型。在 `multiset` 容器中，一个键可以出现多次。

pair 一种类型，有两个 `public` 数据成员，分别名为 `first` 和 `second`。`pair` 类型是带有两个类型形参的模板类型，它的类型形参用作数据成员的类型。

set 只存储键的关联容器。在 `set` 容器中，一个键只能出现一次。

strict weak ordering (严格弱排序) 关联容器所使用的键之间的比较关系。在这种关系下，任意两个元素都可比较，并能确定两者之间谁比谁小。如果两个值都不比对方小，则这两个值相等。详见 10.3.1 节。

value_type 存储在容器中的元素的类型。对于 `set` 和 `multiset` 容器，`value_type` 与 `key_type` 相同。而对于 `map` 和 `multimap` 容器，该类型为 `pair` 类型，它的 `first` 成员是

`const key_type` 类型，`second` 成员则是 `mapped_type` 类型。

*** operator (解引用操作符)** 用于 `map`、`set`、`multimap` 或 `multiset` 迭代器时，解引用操作符将生成一个 `value_type` 类型的值。注意，对于 `map` 和 `multimap` 容器，`value_type` 是

`pair` 类型。

[] operator (下标操作符) 下标操作符。对 `map` 容器使用下标操作符时，`[]` 中的索引必须是 `key_type` 类型（或者是可以转换为 `key_type` 的类型）的值；该运算生成 `mapped_type` 类型的值。

388
?
389

第 11 章

泛型算法

目录

11.1 概述.....	336
11.2 初窥算法.....	339
11.3 再谈迭代器.....	347
11.4 泛型算法的结构.....	358
11.5 容器特有的算法.....	361
小结.....	362
术语.....	363

标准库容器定义的操作非常少。标准库没有给容器添加大量的功能函数，而是选择提供一组算法，这些算法大都不依赖特定的容器类型，是“泛型”的，可作用在不同类型的容器和不同类型的元素上。

泛型算法以及对迭代器更详尽的描述，组成了本章的主题。

标准容器 (the standard container) 定义了很少的操作。大部分容器都支持添加和删除元素；访问第一个和最后一个元素；获取容器的大小，并在某些情况下重设容器的大小；以及获取指向第一个元素和最后一个元素的下一位置的迭代器。

可以想像，用户可能还希望对容器元素进行更多其他有用的操作：也许需要给顺序容器排序，或者查找某个特定的元素，或者查找最大或最小的元素，等等。标准库并没有为每种容器类型都定义实现这些操作的成员函数，而是定义了一组泛型算法 (generic algorithm)：因为它们实现共同的操作，所以称之为“算法”；而“泛型”指的是它们可以操作在多种容器类型上——不但可作用于 `vector` 或 `list` 这些标准库类型，还可用在内置数组类型、甚至其他类型的序列上，这些我们将在本章的后续内容中了解。自定义的容器类型只要与标准库兼容，同样可以使用这些泛型算法。

大多数算法是通过遍历由两个迭代器标记的一段元素来实现其功能。典型情况下，算法在遍历一段元素范围时，操纵其中的每一个元素。算法通过迭代器访问元素，这些迭代器标记了要遍历的元素范围。

11.1 概述

假设有一个 `int` 型的 `vector` 对象，名为 `vec`，我们想知道其中是否包含某个特定值。解决这个问题最简单的方法是使用标准库提供的 `find` 运算：

```
// value we'll look for
int search_value = 42;
// call find to see if that value is present
vector<int>::const_iterator result =
    find(vec.begin(), vec.end(), search_value);
// report the result
cout << "The value " << search_value
    << (result == vec.end()
        ? " is not present" : " is present")
    << endl;
```

使用两个迭代器和一个值调用 `find` 函数，检查两个迭代器实参标记范围内的每一个元素。只要找到与给定值相等的元素，`find` 就会返回指向该元素的迭代器。如果没有匹配的元素，`find` 就返回它的第二个迭代器实参，表示查找失败。于是，只要检查该函数的返回值是否与它的第二个实参相等，就可得知元素是否找到了。我们在输出语句中使用条件操作符 (5.7 节) 实现这个检查并报告是否找到了给定值。

392

由于 `find` 运算是基于迭代器的，因此可在任意容器中使用相同的 `find` 函数查找值。例如，可在一个名为 `lst` 的 `int` 型 `list` 对象上，使用 `find` 函数查找一个值：

```
// call find to look through elements in a list
list<int>::const_iterator result =
    find(lst.begin(), lst.end(), search_value);
cout << "The value " << search_value
    << (result == lst.end()
        ? " is not present" : " is present")
```

```
<< endl;
```

除了 `result` 的类型和传递给 `find` 的迭代器类型之外，这段代码与使用 `find` 在 `vector` 对象中查找元素的程序完全相同。

类似地，由于指针的行为与作用在内置数组上的迭代器一样，因此也可以使用 `find` 来搜索数组：

```
int ia[6] = {27, 210, 12, 47, 109, 83};
int search_value = 83;
int *result = find(ia, ia + 6, search_value);
cout << "The value " << search_value
      << (result == ia + 6
          ? " is not present" : " is present")
      << endl;
```

这里给 `find` 函数传递了两个指针：指向 `ia` 数组中第一个元素的指针，以及指向 `ia` 数组起始位置之后第 6 个元素的指针（即 `ia` 的最后一个元素的下一位置）。如果返回的指针等于 `ia+6`，那么搜索不成功；否则，返回的指针指向找到的值。

如果需要传递一个子区间，则传递指向这个子区间的第一个元素以及最后一个元素的下一位置的迭代器（或指针）。例如，在下面对 `find` 函数的调用中，只搜索了 `ia[1]` 和 `ia[2]`：

```
// only search elements ia[1] and ia[2]
int *result = find(ia + 1, ia + 3, search_value);
```

1. 算法如何工作

每个泛型算法的实现都独立于单独的容器。这些算法还是大而不全的，并且不依赖于容器存储的元素类型。为了知道算法如何工作，让我们深入了解 `find` 操作。该操作的任务是在一个未排序的元素集合中查找特定的元素。从概念上看，`find` 必须包含以下步骤：

- (1) 顺序检查每个元素。
- (2) 如果当前元素等于要查找的值，那么返回指向该元素的迭代器。
- (3) 否则，检查下一个元素，重复步骤 2，直到找到这个值，或者检查完所有的元素为止。
- (4) 如果已经到达集合末尾，而且还未找到该值，则返回某个值，指明要查找的值在这个集合中不存在。

393

2. 标准算法固有地独立于类型

这种算法，正如我们所指出的，与容器的类型无关：在前面的描述中，没有任何内容依赖于容器类型。这种算法只在一点上隐式地依赖元素类型：必须能够对元素做比较运算。该算法的明确要求如下：

- (1) 需要某种遍历集合的方式：能够从一个元素向前移到下一个元素。
- (2) 必须能够知道是否到达了集合的末尾。
- (3) 必须能够对容器中的每一个元素与被查找的元素进行比较。
- (4) 需要一个类型来指出元素在容器中的位置，或者表示找不到该元素。

3. 迭代器将算法和容器绑定起来

泛型算法用迭代器来解决第一个要求：遍历容器。所有迭代器都支持自增操作符，从一个元

素定位到下一个元素，并提供解引用操作符访问元素的值。除了 11.3.5 节将介绍的一个例外情况之外，迭代器还支持相等和不等操作符，用于判断两个迭代器是否相等。

大多数情况下，每个算法都需要使用（至少）两个迭代器来指出该算法操纵的元素范围。第一个迭代器指向第一个元素，而第二个迭代器则指向最后一个元素的下一位置。第二个迭代器所指向的元素[有时被称为**超出末端迭代器**（off-the-end iterator）]本身不是要操作的元素，而被用作终止遍历的哨兵（sentinel）。

使用超出末端迭代器还可以很方便地处理第四个要求，只要以此迭代器为返回值，即可表示没有找到要查找的元素。如果要查找的值未找到，则返回超出末端迭代器；否则，返回的迭代器指向匹配的元素。

第三个要求——元素值的比较，有两种解决方法。默认情况下，find 操作要求元素类型定义了相等（==）操作符，算法使用这个操作符比较元素。如果元素类型不支持相等（==）操作符，或者打算用不同的测试方法来比较元素，则可使用第二个版本的 find 函数。这个版本需要一个额外的参数：实现元素比较的函数名字。

这些算法从不使用容器操作，因而其实现与类型无关，元素的所有访问和遍历都通过迭代器实现。实际的容器类型未知（甚至所处理的元素是否存储在容器中也是未知的）。

标准库提供了超过 100 种算法。与容器一样，算法有着一致的结构。比起死记全部一百多种算法，了解算法的设计可使我们更容易学习和使用它们。本章除了举例说明这些算法的使用之外，还将描述标准库算法的统一原理。附录根据操作分类列出了所有的算法。

394

习题

习题 11.1 algorithm 头文件定义了一个名为 count 的函数，其功能类似于 find。这个函数使用一对迭代器和一个值做参数，返回这个值出现次数的统计结果。编写程序读取一系列 int 型数据，并将它们存储到 vector 对象中，然后统计某个指定的值出现了多少次。

习题 11.2 重复前面的程序，但是，将读入的值存储到一个 string 类型的 list 对象中。

关键概念：算法永不执行容器提供的操作

泛型算法本身从不执行容器操作，只是单独依赖迭代器和迭代器操作实现。算法基于迭代器及其操作实现，而并非基于容器操作。这个事实也许比较意外，但本质上暗示了：使用“普通”的迭代器时，算法从不修改基础容器的大小。正如我们所看到的，算法也许会改变存储在容器中的元素的值，也许会在容器内移动元素，但是，算法从不直接添加或删除元素。

11.3.1 节将介绍标准库提供的另一种特殊的迭代器类：插入器（inserter），除了用于遍历其所绑定的序列之外，还可实现更多的功能。在给这类迭代器赋值时，在基础容器上将执行插入运算。如果算法操纵这类迭代器，迭代器将可能导致在容器中添加元素。但是，算法本身从不这么做。

11.2 初窥算法

在研究算法标准库的结构之前，先看一些例子。上一节已经介绍了 `find` 函数的用法；本节将要使用其他的一些算法。使用泛型算法必须包含 `algorithm` 头文件：

```
#include <algorithm>
```

标准库还定义了一组泛化的算术算法（`generalized numeric algorithm`），其命名习惯与泛型算法相同。使用这些算法则必须包含 `numeric` 头文件：

```
#include <numeric>
```

除了少数例外情况，所有算法都在一段范围内的元素上操作，我们将这段范围称为“输入范围（`input range`）”。带有输入范围参数的算法总是使用头两个形参标记该范围。这两个形参是分别指向要处理的第一个元素和最后一个元素的下一位置的迭代器。

尽管大多数算法对输入范围的操作是类似的，但在该范围内如何操纵元素却有所不同。理解算法的最基本方法是了解该算法是否读元素、写元素或者对元素进行重新排序。在本节的余下内容中，将会观察到每种算法的例子。

395

11.2.1 只读算法

许多算法只会读取其输入范围内的元素，而不会写这些元素。`find` 就是一个这样的算法。另一个简单的只读算法是 `accumulate`，该算法在 `numeric` 头文件中定义。假设 `vec` 是一个 `int` 型的 `vector` 对象，下面的代码：

```
// sum the elements in vec starting the summation with the value 42
int sum = accumulate(vec.begin(), vec.end(), 42);
```

将 `sum` 设置为 `vec` 的元素之和再加上 42。`accumulate` 带有三个形参。头两个形参指定要累加的元素范围。第三个形参则是累加的初值。`accumulate` 函数将它的一个内部变量设置为指定的初值，然后在此初值上累加输入范围内所有元素的值。`accumulate` 算法返回累加的结果，其返回类型就是其第三个实参的类型。



用于指定累加起始值的第三个实参是必要的，因为 `accumulate` 对将要累加的元素类型一无所知，因此，除此之外，没有别的办法创建合适的起始值或者关联的类型。

`accumulate` 对要累加的元素类型一无所知，这个事实有两层含义。首先，调用该函数时必须传递一个起始值，否则，`accumulate` 将不知道使用什么起始值。其次，容器内的元素类型必须与第三个实参的类型匹配，或者可转换为第三个实参的类型。在 `accumulate` 内部，第三个实参用作累加的起点；容器内的元素按顺序连续累加到总和之中。因此，必须能够将元素类型加到总和类型上。

考虑下面的例子，可以使用 `accumulate` 把 `string` 型的 `vector` 容器中的元素连接起来：

```
// concatenate elements from v and store in sum
string sum = accumulate(v.begin(), v.end(), string(""));
```

这个函数调用的效果是：从空字符串开始，把 vec 里的每个元素连接成一个字符串。注意：程序显式地创建了一个 string 对象，用作该函数调用的第三个实参。传递一个字符串字面值，将会导致编译时错误。因为此时，累加和的类型将是 const char*，而 string 的加法操作符（3.2.3 节）所使用的操作数则分别是 string 和 const char* 类型，加法的结果将产生一个 string 对象，而不是 const char* 指针。

find_first_of 的使用

396

除了 find 之外，标准库还定义了一些更复杂的查找算法。当中的一部分类似 string 类的 find 操作（9.6.4 节），其中一个就是 find_first_of 函数。这个算法带有两对迭代器参数来标记两段元素范围，在第一段范围内查找与第二段范围中任意元素匹配的元素，然后返回一个迭代器，指向第一个匹配的元素。如果找不到匹配元素，则返回第一个范围的 end 迭代器。假设 roster1 和 roster2 是两个存放名字的 list 对象，可使用 find_first_of 统计有多少个名字同时出现在这两个列表中：

```
// program for illustration purposes only:
// there are much faster ways to solve this problem
size_t cnt = 0;
list<string>::iterator it = roster1.begin();
// look in roster1 for any name also in roster2
while ((it = find_first_of(it, roster1.end(),
                          roster2.begin(), roster2.end()))
       != roster1.end()) {
    ++cnt;
    // we got a match, increment it to look in the rest of roster1
    ++it;
}
cout << "Found " << cnt
     << " names on both rosters" << endl;
```

调用 find_first_of 查找 roster2 中的每个元素是否与第一个范围内的元素匹配，也就是在 it 到 roster1.end() 范围内查找一个元素。该函数返回此范围内第一个同时存在于第二个范围中的元素。在 while 的第一次循环中，遍历整个 roster1 范围。第二次以及后续的循环迭代则只考虑 roster1 中尚未匹配的部分。

循环条件检查 find_first_of 的返回值，判断是否找到匹配的名字。如果找到一个匹配，则使计数器加 1，同时给 it 加 1，使它指向 roster1 中的下一个元素。很明显可知，当不再有任何匹配时，find_first_of 返回 roster1.end()，完成统计。

关键概念：迭代器实参类型

通常，泛型算法都是在标记容器（或其他序列）内的元素范围的迭代器上操作的。标记范围的两个实参类型必须精确匹配，而迭代器本身必须标记一个范围：它们必须指向同一个容器中的元素（或者超出容器末端的下一位置），并且如果两者不相等，则第一个迭代器通过不断地自增，必须可以到达第二个迭代器。

有些算法，例如 find_first_of，带有两对迭代器参数。每对迭代器中，两个实参的类

型必须精确匹配，但不要求两对之间的类型匹配。特别是，元素可存储在不同类型的序列中，只要这两个序列的元素可以比较即可。

在上述程序中，roster1 和 roster2 的类型不必精确匹配：roster1 可以是 list 对象，而 roster2 则可以是 vector 对象、deque 对象或者是其他后面要学到的序列。只要这两个序列的元素可使用相等 (==) 操作符进行比较即可。如果 roster1 是 list<string> 对象，则 roster2 可以是 vector<char*> 对象，因为 string 标准库为 string 对象与 char* 对象定义了相等 (==) 操作符。

397

习题

习题 11.3 用 accumulate 统计 vector<int> 容器对象中的元素之和。

习题 11.4 假定 v 是 vector<double> 类型的对象，则调用 accumulate(v.begin(), v.end(), 0) 是否有错？如果有的话，错在哪里？

习题 11.5 对于本节调用 find_first_of 的例程，如果不给 it 加 1，将会如何？

11.2.2 写容器元素的算法

一些算法写入元素值。在使用这些算法写元素时要当心，必须确保算法所写的序列至少足以存储要写入的元素。

有些算法直接将数据写入到输入序列，另外一些则带有一个额外的迭代器参数指定写入目标。这类算法将目标迭代器用作输出的位置。还有第三种算法将指定数目的元素写入某个序列。

1. 写入输入序列的元素

写入到输入序列的算法本质上是安全的——只会写入与指定输入范围数量相同的元素。

写入到输入序列的一个简单算法是 fill 函数，考虑如下例子：

```
fill(vec.begin(), vec.end(), 0); // reset each element to 0
// set subsequence of the range to 10
fill(vec.begin(), vec.begin() + vec.size()/2, 10);
```

fill 带有一对迭代器形参，用于指定要写入的范围，而所写的值是它的第三个形参的副本。执行时，将该范围内的每个元素都设为给定的值。如果输入范围有效，则可安全写入。这个算法只会对输入范围内已存在的元素进行写入操作。

398

2. 不检查写入操作的算法

fill_n 函数带有的参数包括：一个迭代器、一个计数器以及一个值。该函数从迭代器指向的元素开始，将指定数量的元素设置为给定的值。fill_n 函数假定对指定数量的元素做写操作是安全的。初学者常犯的错误是：在没有元素的空容器上调用 fill_n 函数（或者类似的写元素算法）。

```
vector<int> vec; // empty vector
// disaster: attempts to write to 10 (nonexistent) elements in vec
fill_n(vec.begin(), 10, 0);
```

这个 fill_n 函数的调用将带来灾难性的后果。我们指定要写入 10 个元素，但这些元素却不存

在——vec 是空的。其结果未定义，很可能导致严重的运行时错误。



对指定数目的元素做写入运算，或者写到目标迭代器的算法，都不检查目标的大小是否足以存储要写入的元素。

3. 引入 **back_inserter**

确保算法有足够的元素存储输出数据的一种方法是使用**插入迭代器** (insert iterator)。插入迭代器是可以给基础容器添加元素的迭代器。通常，用迭代器给容器元素赋值时，被赋值的是迭代器所指向的元素。而使用插入迭代器赋值时，则会在容器中添加一个新元素，其值等于赋值运算的右操作数的值。

11.3.1 节将会讨论更多关于插入迭代器的内容。然而，为了说明如何安全使用写容器的算法，下面将使用 **back_inserter**。使用 back_inserter 的程序必须包含 iterator 头文件。

back_inserter 函数是迭代器适配器。与容器适配器 (9.7 节) 一样，迭代器适配器使用一个对象作为实参，并生成一个适应其实参行为的新对象。在本例中，传递给 back_inserter 的实参是一个容器的引用。back_inserter 生成一个绑定在该容器上的插入迭代器。在试图通过这个迭代器给元素赋值时，赋值运算将调用 push_back 在容器中添加一个具有指定值的元素。使用 back_inserter 可以生成一个指向 fill_n 写入目标的迭代器：

```
vector<int> vec; // empty vector
// ok: back_inserter creates an insert iterator that adds elements to vec
fill_n (back_inserter(vec), 10, 0); // appends 10 elements to vec
```

现在，fill_n 函数每写入一个值，都会通过 back_inserter 生成的插入迭代器实现。效果相当于在 vec 上调用 push_back，在 vec 末尾添加 10 个元素，每个元素的值都是 0。

4. 写入到目标迭代器的算法

第三类算法向目标迭代器写入未知个数的元素。正如 fill_n 函数一样，目标迭代器指向存放输出数据的序列中第一个元素。这类算法中最简单的是 copy 函数。copy 带有三个迭代器参数：头两个指定输入范围，第三个则指向目标序列的一个元素。传递给 copy 的目标序列必须至少要与输入范围一样大。假设 ilst 是一个存放 int 型数据的 list 对象，可如下将它 copy 给一个 vector 对象：

```
vector<int> ivec; // empty vector
// copy elements from ilst into ivec
copy (ilst.begin(), ilst.end(), back_inserter(ivec));
```

399 copy 从输入范围中读取元素，然后将它们复制给目标 ivec。

当然，这个例子的效率比较差：通常，如果要以一个已存在的容器为副本创建新容器，更好的方法是直接用输入范围作为新构造容器的初始化式：

```
// better way to copy elements from ilst
vector<int> ivec(ilst.begin(), ilst.end());
```

5. 算法的 **_copy** 版本

有些算法提供所谓的“复制(copying)”版本。这些算法对输入序列的元素做出处理，但不修

改原来的元素，而是创建一个新序列存储元素的处理结果。

`replace` 算法就是一个很好的例子。该算法对输入序列做读写操作，将序列中特定的值替换为新的值。该算法带有四个形参：一对指定输入范围的迭代器和两个值。每一个等于第一个值的元素替换成第二个值。

```
// replace any element with value of 0 by 42
replace(ilst.begin(), ilst.end(), 0, 42);
```

这个调用将所有值为 0 的实例替换成 42。如果不想改变原来的序列，则调用 `replace_copy`。这个算法接受第三个迭代器实参，指定保存调整后序列的目标位置。

```
// create empty vector to hold the replacement
vector<int> ivec;
// use back_inserter to grow destination as needed
replace_copy (ilst.begin(), ilst.end(),
              back_inserter(ivec), 0, 42);
```

调用该函数后，`ilst` 没有改变，`ivec` 存储 `ilst` 的一份副本，而 `ilst` 内所有的 0 在 `ivec` 中都变成了 42。

习题

习题 11.6 使用 `fill_n` 编写程序，将一个 `int` 型序列的值设为 0。

习题 11.7 判断下面的程序是否有错，如果有，请改正之：

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

```
(b) vector<int> vec;
    vec.reserve(10);
    fill_n(vec.begin(), 10, 0);
```

习题 11.8 前面说过，算法不改变它所操纵的容器的大小，为什么使用 `back_inserter` 也不能突破这个限制？

11.2.3 对容器元素重新排序的算法

假设我们要分析一组儿童故事中所使用的单词。例如，可能想知道它们使用了多少个由六个或以上字母组成的单词。每个单词只统计一次，不考虑它出现的次数，也不考虑它是否在多个故事中出现。要求以长度的大小输出这些单词，对于同样长的单词，则以字典顺序输出。

假定每本书的文本已经读入并保存在一个 `string` 类型的 `vector` 对象中，它的名字是 `words`。现在，应该怎么解决包括统计单词出现次数这个问题呢？为了解决此问题，要做下面几项操作：

- (1) 去掉所有重复的单词。
- (2) 按单词的长度排序。
- (3) 统计长度等于或超过 6 个字符的单词个数。

上述每一步都可使用泛型算法实现。

为了说清楚，使用下面这个简单的故事作为我们的输入：

the quick red fox jumps over the slow red turtle

对于这个输入，我们的程序应该产生如下输出：

1 word 6 characters or longer

1. 去除重复

假设我们的输入存储在一个名为 words 的 vector 对象中，第一个子问题是将 words 中重复出现的单词去除掉：

```
// sort words alphabetically so we can find the duplicates
sort(words.begin(), words.end());
/* eliminate duplicate words:
 * unique reorders words so that each word appears once in the
 * front portion of words and returns an iterator one past the unique range;
 * erase uses a vector operation to remove the nonunique elements
 */
vector<string>::iterator end_unique =
    unique(words.begin(), words.end());
words.erase(end_unique, words.end());
```

vector 对象包含每个故事中使用的所有单词。首先对此 vector 对象排序。sort 算法带有两个迭代器实参，指出要排序的元素范围。这个算法使用小于 (<) 操作符比较元素。在本次调用中，要求对整个 vector 对象排序。

调用 sort 后，此 vector 对象的元素按次序排列：

fox jumps over quick red red slow the the turtle

401 注意，单词 red 和 the 重复出现了。

2. unique 的使用

单词按次序排列后，现在的问题是：让故事中所用到的每个单词都只保留一个副本。unique 算法很适合用于解决这个问题，它带有两个指定元素范围的迭代器参数。该算法删除相邻的重复元素，然后重新排列输入范围内的元素，并且返回一个迭代器，表示无重复的值范围的结束。

调用 unique 后，vector 中存储的内容是：

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

↑
 last_word
 (One past last unique element)

注意，words 的大小并没有改变，依然保存着 10 个元素；只是这些元素的顺序改变了。调用 unique “删除”了相邻的重复值。给“删除”加上引号是因为 unique 实际上并没有删除任何元素，而是将无重复的元素复制到序列的前端，从而覆盖相邻的重复元素。unique 返回的迭代器指向超出无重复的元素范围末端的下一位置。

3. 使用容器操作删除元素

如果要删除重复的项，必须使用容器操作，在本例中调用 erase 实现该功能。这个函数调

用从 `end_unique` 指向的元素开始删除，直到 `words` 的最后一个元素也删除掉为止。调用之后，`words` 存储输入的 8 个不相同的元素。



算法不直接修改容器的大小。如果需要添加或删除元素，则必须使用容器操作。

值得注意的是，对没有重复元素的 `vector` 对象，调用 `erase` 也是安全的。如果不存在重复的元素，`unique` 就会返回 `words.end()`，此时，调用 `erase` 的两个实参值相同，都是 `words.end()`。两个迭代器相等这个事实意味着 `erase` 函数要删除的范围是空的。删除一段空的范围没有任何作用，所以即使输入中没有重复的元素，我们的程序仍然正确。

4. 定义需要的实用函数

下一个子问题是统计长度不小于 6 的单词个数。为了解决这个问题，需要用到另外两个泛型算法：`stable_sort` 和 `count_if`。使用这些算法，还需要一个配套的实用函数，称为谓词 (predicate)。谓词是做某些检测的函数，返回用于条件判断的类型，指出条件是否成立。

我们需要的第一个谓词将用在基于大小的元素排序中。为了实现排序，必须定义一个谓词函数来实现两个 `string` 对象的比较，并返回一个 `bool` 值，指出第一个字符串是否比第二个短：

402

```
// comparison function to be used to sort by word length
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

另一个所需的谓词函数将判断给出的 `string` 对象的长度是否不小于 6：

```
// determine whether a length of a given word is 6 or more
bool GT6(const string &s)
{
    return s.size() >= 6;
}
```

尽管这个函数能解决问题，但存在不必要的限制——函数内部硬性规定了对长度大小的要求。如果要统计其他长度的单词个数，则必须编写另一个函数。其实很容易写出更通用的比较函数，使它带有两个形参，分别是 `string` 对象和一个长度大小值即可。但是，传递给 `count_if` 算法的函数只能带有一个实参，因此本程序不能使用上述更通用的方法。14.8.1 节将为这个问题提供更好的解决方案。

5. 排序算法

标准库定义了四种不同的排序算法，上面只使用了最简单的 `sort` 算法，使 `words` 按字典次序排列。除了 `sort` 之外，标准库还定义了 `stable_sort` 算法，`stable_sort` 保留相等元素的原始相对位置。通常，对于已排序的序列，我们并不关心其相等元素的相对位置，毕竟，这些元素是相等的。但是，在这个应用中，我们将“相等”定义为“相同的长度”，有着相同长度的元素还能以字典次序的不同而区分。调用 `stable_sort` 后，对于长度相同的元素，将保留其字典顺序。

`sort` 和 `stable_sort` 都是重载函数。其中一个版本使用元素类型提供的小于 (<) 操作符

实现比较。在查找重复元素之前，我们就是用这个 `sort` 版本对元素排序。第二个重载版本带有第三个形参：比较元素所使用的谓词函数的名字。这个谓词函数必须接受两个实参，实参的类型必须与元素类型相同，并返回一个可用作条件检测的值。下面将比较元素的 `isShorter` 函数作为实参，调用第二个版本的排序函数：

```
// sort words by size, but maintain alphabetic order for words of the same size
stable_sort(words.begin(), words.end(), isShorter);
```

调用后，`words` 中的元素按长度大小排序，而长度相同的单词则仍然保持字典顺序：

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

403

6. 统计长度不小于 6 的单词

现在此 `vector` 对象已经按单词长度排序，剩下的问题就是统计长度不小于 6 的单词个数。使用 `count_if` 算法处理这个问题：

```
vector<string>::size_type wc =
    count_if(words.begin(), words.end(), GT6);
```

执行 `count_if` 时，首先读取它的头两个实参所标记的范围内的元素。每读出一个元素，就将它传递给第三个实参表示的谓词函数。此谓词函数需要单个元素类型的实参，并返回一个可用作条件检测的值。`count_if` 算法返回使谓词函数返回条件成立的元素个数。在这个程序中，`count_if` 将每个单词传递给 `GT6`，而 `GT6` 返回一个 `bool` 值，如果单词长度不小于 6，则该 `bool` 值为 `true`。

7. 将全部程序段放在一起

了解程序的细节之后，下面是完整的程序：

```
// comparison function to be used to sort by word length
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
// determine whether a length of a given word is 6 or more
bool GT6(const string &s)
{
    return s.size() >= 6;
}
int main()
{
    vector<string> words;
    // copy contents of each book into a single vector
    string next_word;
    while (cin >> next_word) {
        // insert next book's contents at end of words
        words.push_back(next_word);
    }
    // sort words alphabetically so we can find the duplicates
    sort(words.begin(), words.end());
    /* eliminate duplicate words:
       * unique reorders words so that each word appears once in the
```

```

    * front portion of words and returns an iterator one past the unique range;
    * erase uses a vector operation to remove the nonunique elements
    */
vector<string>::iterator end_unique =
    unique(words.begin(), words.end());
words.erase(end_unique, words.end());
// sort words by size, but maintain alphabetic order for words of the same size
stable_sort(words.begin(), words.end(), isShorter);
vector<string>::size_type wc =
    count_if(words.begin(), words.end(), GT6);
cout << wc << " " << make_plural(wc, "word", "s")
    << " 6 characters or longer" << endl;
return 0;
}

```

404

最后，我们留下按长度顺序输出单词这个问题作为习题。

习题

习题 11.9 编写程序统计长度不小于 4 的单词，并输出输入序列中不重复的单词。在程序源文件上运行和测试你自己编写的程序。

习题 11.10 标准库定义了一个 `find_if` 函数。与 `find` 一样，`find_if` 函数带有一对迭代器形参，指定其操作的范围。与 `count_if` 一样，该函数还带有第三个形参，表明用于检查范围内每个元素的谓词函数。`find_if` 返回一个迭代器，指向第一个使谓词函数返回非零值的元素。如果这样的元素不存在，则返回第二个迭代器实参。使用 `find_if` 函数重写上述例题中统计长度大于 6 的单词个数的程序部分。

习题 11.11 你认为为什么算法不改变容器的大小？

习题 11.12 为什么必须使用 `erase`，而不是定义一个泛型算法来删除容器中的元素？

11.3 再谈迭代器

11.2.2 节已强调标准库所定义的迭代器不依赖于特定的容器。事实上，C++ 语言还提供了另外三种迭代器：

(1) **插入迭代器** (insert iterator)：这类迭代器与容器绑定在一起，实现在容器中插入元素的功能。

(2) **iostream 迭代器** (iostream iterator)：这类迭代器可与输入或输出流绑定在一起，用于迭代遍历所关联的 IO 流。

(3) **反向迭代器** (reverse iterator)：这类迭代器实现向后遍历，而不是向前遍历。所有容器类型都定义了自己的 `reverse_iterator` 类型，由 `rbegin` 和 `rend` 成员函数返回。

上述迭代器类型都在 `iterator` 头文件中定义。

本节将详细分析上述每种迭代器，并介绍在泛型算法中如何使用这些迭代器，还会了解什么时候应该使用和如何使用 `const_iterator` 容器。

405

11.3.1 插入迭代器

11.2.2 节使用 `back_inserter` 创建一个迭代器，用来给容器添加元素。`back_inserter` 函数是一种插入器。插入器是一种迭代器适配器（9.7 节），带有一个容器参数，并生成一个迭代器，用于在指定容器中插入元素。通过插入迭代器赋值时，迭代器将会插入一个新的元素。C++ 语言提供了三种插入器，其差别在于插入元素的位置不同。

- (1) `back_inserter`，创建使用 `push_back` 实现插入的迭代器。
- (2) `front_inserter`，使用 `push_front` 实现插入。
- (3) `inserter`，使用 `insert` 实现插入操作。除了所关联的容器外，`inserter` 还带有第二个实参：指向插入起始位置的迭代器。

1. `front_inserter` 需要使用 `push_front`

`front_inserter` 的操作类似于 `back_inserter`：该函数将创建一个迭代器，调用它所关联的基础容器的 `push_front` 成员函数代替赋值操作。



只有当容器提供 `push_front` 操作时，才能使用 `front_inserter`。在 `vector` 或其他没有 `push_front` 运算的容器上使用 `front_inserter`，将产生错误。

2. `inserter` 将产生在指定位置实现插入的迭代器

`inserter` 适配器提供更普通的插入形式。这种适配器带有两个实参：所关联的容器和指示起始插入位置的迭代器。

```
// position an iterator into ilst
list<int>::iterator it =
    find (ilst.begin(), ilst.end(), 42);
// insert replaced copies of ivec at that point in ilst
replace_copy (ivec.begin(), ivec.end(),
    inserter (ilst, it), 100, 0);
```

首先用 `find` 定位 `ilst` 中的某个元素。使用 `inserter` 作为实参调用 `replace_copy`，`inserter` 将会在 `ilst` 中由 `find` 返回的迭代器所指向的元素前面插入新元素。而调用 `replace_copy` 的效果是从 `ivec` 中复制元素，并将其中值为 100 的元素替换为 0 值。`ilst` 的新元素在 `it` 所标明的元素前面插入。

在创建 `inserter` 时，应指明新元素在何处插入。`inserter` 函数总是在它的迭代器实参所标明的位置前面插入新元素。

也许我们会认为可使用 `inserter` 和容器的 `begin` 迭代器来模拟 `front_inserter` 的效果。

然而，`inserter` 的行为与 `front_inserter` 的有很大差别。在使用 `front_inserter` 时，元素始终在容器的第一个元素前面插入。而使用 `inserter` 时，元素则在指定位置前面插入。即使此指定位置初始化为容器中的第一个元素，但是，一旦在该位置前插入一个新元素后，插入位置就不再是容器的首元素了：

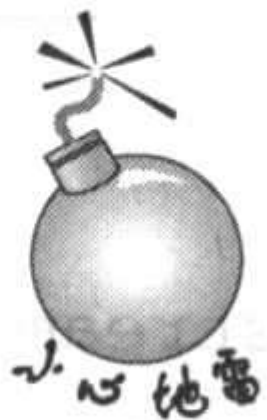
```
list<int> ilst, ilst2, ilst3;    // empty lists
// after this loop ilst contains: 3 2 1 0
for (list<int>::size_type i = 0; i != 4; ++i)
```

```

    ilst.push_front(i);
    // after copy ilst2 contains: 0 1 2 3
    copy (ilst.begin(), ilst.end(), front_inserter(ilst2));
    // after copy, ilst3 contains: 3 2 1 0
    copy (ilst.begin(), ilst.end(),
          inserter (ilst3, ilst3.begin()));

```

在复制并创建 `ilst2` 的过程中，元素总是在这个 `list` 对象的所有元素之前插入。而在复制创建 `ilst3` 的过程中，元素则在 `ilst3` 中的固定位置插入。刚开始时，这个插入位置是此 `list` 对象的头部，但在插入一个元素后，就不再是首元素了。



回顾 9.3.3 节的讨论，应该清楚理解 `front_inserter` 的使用将导致元素以相反的次序出现在目标对象中，这点非常重要。

习题

习题 11.13 解释三种插入迭代器的区别。

习题 11.14 编写程序使用 `replace_copy` 将一个容器中的序列复制给另一个容器，并将前一个序列中给定的值替换为指定的新值。分别使用 `inserter`、`back_inserter` 和 `front_inserter` 实现这个程序。讨论在不同情况下输出序列如何变化。

习题 11.15 算法标准库定义了一个名为 `unique_copy` 的函数，其操作与 `unique` 类似，唯一的区别在于：前者接受第三个迭代器实参，用于指定复制不重复元素的目标序列。编写程序使用 `unique_copy` 将一个 `list` 对象中不重复的元素复制到一个空的 `vector` 对象中。

11.3.2 iostream 迭代器

虽然 `iostream` 类型不是容器，但标准库同样提供了在 `iostream` 对象上使用的迭代器：`istream_iterator` 用于读取输入流，而 `ostream_iterator` 则用于写输出流（表 11-1）。这些迭代器将它们所对应的流视为特定类型的元素序列。使用流迭代器时，可以用泛型算法从流对象中读数据（或将数据写到流对象中）。

407

表 11-1 `iostream` 迭代器的构造函数

<code>istream_iterator<T> in(strm);</code>	创建从输入流 <code>strm</code> 中读取 <code>T</code> 类型对象的 <code>istream_iterator</code> 对象
<code>istream_iterator<T> in;</code>	<code>istream_iterator</code> 对象的超出末端迭代器
<code>ostream_iterator<T> in(strm);</code>	创建将 <code>T</code> 类型的对象写到输出流 <code>strm</code> 的 <code>ostream_iterator</code> 对象
<code>ostream_iterator<T> in(strm, delim);</code>	创建将 <code>T</code> 类型的对象写到输出流 <code>strm</code> 的 <code>ostream_iterator</code> 对象，在写入过程中使用 <code>delim</code> 作为元素的分隔符。 <code>delim</code> 是以空字符结束的字符数组

流迭代器只定义了最基本的迭代器操作：自增、解引用和赋值。此外，可比较两个 `istream` 迭代器是否相等（或不等）。而 `ostream` 迭代器则不提供比较运算（表 11-2）。

表 11-2 istream_iterator 的操作

<code>it1 == it2</code>	比较两个 <code>istream_iterator</code> 对象是否相等（不等）。迭代器读取的必须是相同的类型。
<code>it1 != it2</code>	如果两个迭代器都是 <code>end</code> 值，则它们相等。对于两个都不指向流结束位置的迭代器，如果它们使用同一个输入流构造，则它们也相等
<code>*it</code>	返回从流中读取的值
<code>it->mem</code>	是 <code>(*it).mem</code> 的同义词。返回从流中读取的对象的 <code>mem</code> 成员
<code>++it</code> <code>it++</code>	通过使用元素类型提供的 <code>>></code> 操作符从输入流中读取下一个元素值，使迭代器向前移动。通常，前缀版本使迭代器在流中向前移动，并返回对加 1 后的迭代器的引用。而后缀版本使迭代器在流中向前移动后，返回原值

1. 流迭代器的定义

流迭代器都是类模板：任何已定义输入操作符（`>>`操作符）的类型都可以定义 `istream_iterator`。类似地，任何已定义输出操作符（`<<`操作符）的类型也可定义 `ostream_iterator`。在创建流迭代器时，必须指定迭代器所读写的对象类型：

```
istream_iterator<int> cin_it(cin);      // reads ints from cin
istream_iterator<int> end_of_stream;    // end iterator value
// writes Sales_items from the ostream named outfile
// each element is followed by a space
ofstream outfile;
ostream_iterator<Sales_item> output(outfile, " ");
```

`ostream_iterator` 对象必须与特定的流绑定在一起。在创建 `istream_iterator` 时，可直接将它绑定到一个流上。另一种方法是在创建时不提供实参，则该迭代器指向超出末端位置。`ostream_iterator` 不提供超出末端迭代器。

在创建 `ostream_iterator` 对象时，可提供第二个（可选的）实参，指定将元素写入输出流时使用的分隔符。分隔符必须是 C 风格字符串。因为它是 C 风格字符串，所以必须以空字符结束；否则，其行为将是未定义的。

2. istream_iterator 对象上的操作

构造与流绑定在一起的 `istream_iterator` 对象时，将对迭代器定位，以便第一次对该迭代器进行解引用时即可从流中读取第一个值。

考虑下面例子，可使用 `istream_iterator` 对象将标准输入读到 `vector` 对象中。

```
istream_iterator<int> in_iter(cin);    // read ints from cin
istream_iterator<int> eof;             // istream "end" iterator
// read until end of file, storing what was read in vec
while (in_iter != eof)
    // increment advances the stream to the next value
    // dereference reads next value from the istream
    vec.push_back(*in_iter++);
```

这个循环从 `cin` 中读取 `int` 型数据，并将读入的内容保存在 `vec` 中。每次循环都检查 `in_iter` 是否为 `eof`。其中 `eof` 迭代器定义为空的 `istream_iterator` 对象，用作结束迭代器。绑在流上的迭代器在遇到文件结束或某个错误时，将等于结束迭代器的值。

本程序最难理解的部分是传递给 `push_back` 的实参，该实参使用解引用和后自增操作符。

根据优先级规则 (5.5 节), 自增运算的结果将是解引用运算的操作数。对 `istream_iterator` 对象做自增运算使该迭代器在流中向前移动。然而, 使用后自增运算的表达式, 其结果是迭代器原来的值。自增的效果是使迭代器在流中移动到下一个值, 但返回指向前一个值的迭代器。对该迭代器进行解引用获取该值。

更有趣的是可以这样重写程序:

```
istream_iterator<int> in_iter(cin);    // read ints from cin
istream_iterator<int> eof;             // istream "end" iterator
vector<int> vec(in_iter, eof);         // construct vec from an iterator range
```

这里, 用一对标记元素范围的迭代器构造 `vec` 对象。这些迭代器是 `istream_iterator` 对象, 这就意味着这段范围的元素是通过读取所关联的流来获得的。这个构造函数的效果是读 `cin`, 直到到达文件结束或输入的不是 `int` 型数值为止。读取的元素将用于构造 `vec` 对象。

409

3. `ostream_iterator` 对象和 `ostream_iterator` 对象的使用

可使用 `ostream_iterator` 对象将一个值序列写入流中, 其操作的过程与使用迭代器将一组值逐个赋给容器中的元素相同:

```
// write one string per line to the standard output
ostream_iterator<string> out_iter(cout, "\n");
// read strings from standard input and the end iterator
istream_iterator<string> in_iter(cin), eof;
// read until eof and write what was read to the standard output
while (in_iter != eof)
    // write value of in_iter to standard output
    // and then increment the iterator to get the next value from cin
    *out_iter++ = *in_iter++;
```

这个程序读 `cin`, 并将每个读入的值依次写到 `cout` 中不同的行中。

首先, 定义一个 `ostream_iterator` 对象, 用于将 `string` 类型的数据写到 `cout` 中, 每个 `string` 对象后跟一个换行符。定义两个 `istream_iterator` 对象, 用于从 `cin` 中读取 `string` 对象。`while` 循环类似前一个例子。但是这一次不是将读取的数据存储在 `vector` 对象中, 而是将读取的数据赋给 `out_iter`, 从而输出到 `cout` 上。

这个赋值类似于 6.7 节将一个数组复制给另一个数组的程序。对这两个迭代器进行解引用, 将右边的值赋给左边的元素, 然后两个迭代器都自增 1。其效果就是: 将读取的数据输出到 `cout` 上, 然后两个迭代器都加 1, 再从 `cin` 中读取下一个值。

4. 在类类型上使用 `istream_iterator`

提供了输入操作符 (`>>`) 的任何类型都可以创建 `istream_iterator` 对象。例如, 可如下使用 `istream_iterator` 对象读取一系列的 `Sales_item` 对象, 并求和:

```
istream_iterator<Sales_item> item_iter(cin), eof;
Sales_item sum;           // initially empty Sales_item
sum = *item_iter++;       // read first transaction into sum and get next record
while (item_iter != eof) {
    if (item_iter->same_isbn(sum))
        sum = sum + *item_iter;
    else {
        cout << sum << endl;
```

```

        sum = *item_iter;
    }
    ++item_iter; // read next transaction
}
cout << sum << endl; // remember to print last set of records

```

该程序将迭代器 `item_iter` 与 `cin` 绑在一起，意味着迭代器将读取 `Sales_item` 类型的对象。然后读入第一个记录，赋给 `sum`：

410 `sum = *item_iter++; // read first transaction into sum and get next record`

这个语句使用解引用操作符获取标准输入的第一个记录，并将这个值赋给 `sum`。然后给迭代器加 1，使流从标准输入中读取下一记录。

`while` 循环反复执行直到到达 `cin` 的结束位置为止。在 `while` 循环中，将刚读入记录的 `isbn` 与 `sum` 的 `isbn` 比较。`while` 中的第一个语句使用了箭头操作符对 `istream` 迭代器进行解引用，获得最近读入的对象。然后在该对象和 `sum` 对象上调用 `same_isbn` 成员。

如果 `isbn` 值相同，则增加总和 `sum`。否则，输出 `sum` 的当前值，并将它重设为最近读取对象的副本。循环的最后一步是给迭代器加 1，在本例中，将导致从标准输入中读入下一个 `Sales_item` 对象。循环持续直到遇到错误或结束位置为止。在结束程序之前，记住输出从输入中读入的最后一个 ISBN 所关联的值。

5. 流迭代器的限制

流迭代器有下面几个重要的限制：

- 不可能从 `ostream_iterator` 对象读入，也不可能写到 `istream_iterator` 对象中。
- 一旦给 `ostream_iterator` 对象赋了一个值，写入就提交了。赋值后，没有办法再改变这个值。此外，`ostream_iterator` 对象中每个不同的值都只能正好输出一次。
- `ostream_iterator` 没有 `->` 操作符。

6. 与算法一起使用流迭代器

正如大家所知，算法是基于迭代器操作实现的。如同前面所述，流迭代器至少定义了一些迭代器操作。由于流迭代器支持迭代器操作，因此，至少可在一些泛型算法上使用这类迭代器。考虑下面的例子，从标准输入读取一些数，再将读取的不重复的数写到标准输出：

```

istream_iterator<int> cin_it(cin); // reads ints from cin
istream_iterator<int> end_of_stream; // end iterator value
// initialize vec from the standard input:
vector<int> vec(cin_it, end_of_stream);
sort(vec.begin(), vec.end());
// writes ints to cout using " " as the delimiter
ostream_iterator<int> output(cout, " ");
// write only the unique elements in vec to the standard output
unique_copy(vec.begin(), vec.end(), output);

```

如果程序的输入是：

411 `23 109 45 89 6 34 12 90 34 23 56 23 8 89 23`

输出则是：

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

程序用一对迭代器 `input` 和 `end_of_stream` 创建了 `vec` 对象。这个初始化的效果是读取 `cin` 直到文件结束或者出现错误为止。读取的值保存在 `vec` 里。

读取输入和初始化 `vec` 后，调用 `sort` 对输入的数排序。`sort` 调用完成后，重复输入的数就会相邻存储。

程序再使用 `unique_copy` 算法，这是 `unique` 的“复制”版本。该算法将输入范围中不重复的值复制到目标迭代器。该调用将输出迭代器用作目标。其效果是将 `vec` 中不重复的值复制给 `cout`，每个复制的值后面输出一个空格。

习题

习题 11.16 重写 (11.3.2 节第 3 小节) 的程序，使用 `copy` 算法将一个文件的内容写到标准输出中。

习题 11.17 使用一对 `istream_iterator` 对象初始化一个 `int` 型的 `vector` 对象。

习题 11.18 编写程序使用 `istream_iterator` 对象从标准输入读入一系列整数。使用 `ostream_iterator` 对象将其中的奇数写到一个文件中，并在每个写入的值后面加一个空格。同样使用 `ostream_iterator` 对象将偶数写到第二个文件，每个写入的值都存放在单独的行中。

11.3.3 反向迭代器

反向迭代器是一种反向遍历容器的迭代器。也就是，从最后一个元素到第一个元素遍历容器。反向迭代器将自增（和自减）的含义反过来了：对于反向迭代器，`++` 运算将访问前一个元素，而 `--` 运算则访问下一个元素。

回想一下，所有容器都定义了 `begin` 和 `end` 成员，分别返回指向容器首元素和尾元素下一位置的迭代器。容器还定义了 `rbegin` 和 `rend` 成员，分别返回指向容器尾元素和首元素前位置的反向迭代器。与普通迭代器一样，反向迭代器也有常量 (`const`) 和非常量 (`nonconst`) 类型。图 11-1 使用一个假设名为 `vec` 的 `vector` 类型对象阐明了这四种迭代器之间的关系。

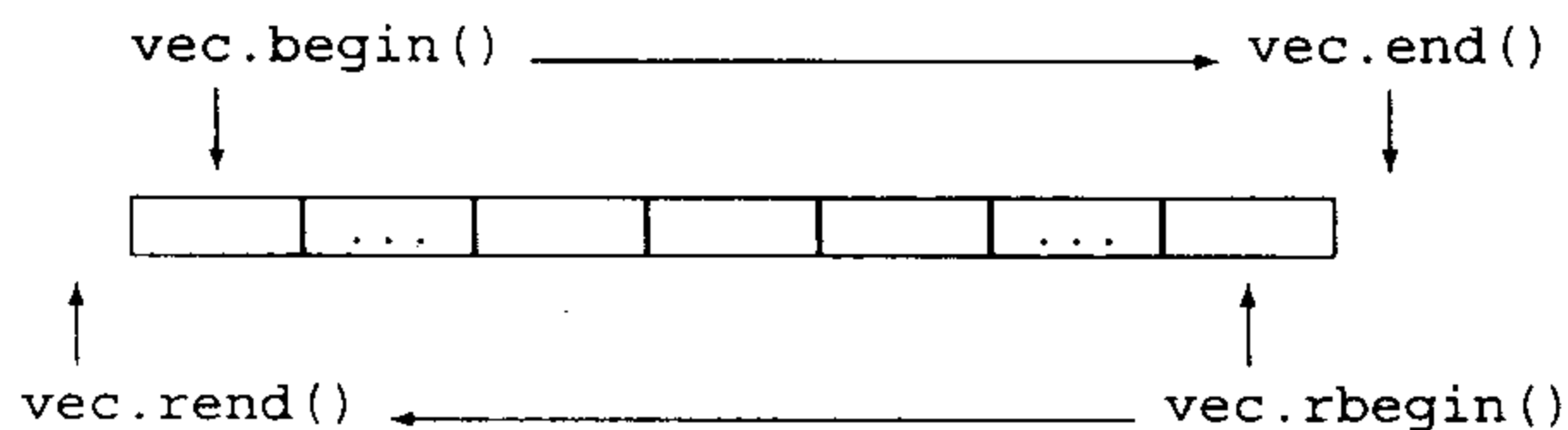


图 11-1 比较 `begin/end` 和 `rbegin/rend` 迭代器

假设有一个 `vector` 容器对象，存储了 0~9 这 10 个以升序排列的数字：

```
vector<int> vec;
for (vector<int>::size_type i = 0; i != 10; ++i)
    vec.push_back(i); // elements are 0,1,2,...,9
```

下面的 `for` 循环将以逆序输出这些元素：

```
// reverse iterator of vector from back to front
```

```
vector<int>::reverse_iterator r_iter;
for (r_iter = vec.rbegin(); // binds r_iter to last element
     r_iter != vec.rend(); // rend refers 1 before 1st element
     ++r_iter) // decrements iterator one element
    cout << *r_iter << endl; // prints 9,8,7,...0
```

虽然颠倒自增和自减这两个操作符的意义似乎容易使人迷惑,但是它让程序员可以透明地向前或向后处理容器。例如,为了以降序排列 vector,只需向 sort 传递一对反向迭代器:

```
// sorts vec in "normal" order
sort(vec.begin(), vec.end());
// sorts in reverse: puts smallest element at the end of vec
sort(vec.rbegin(), vec.rend());
```

1. 反向迭代器需要使用自减操作符

从一个既支持--也支持++的迭代器就可以定义反向迭代器,这不用感到吃惊。毕竟,反向迭代器的目的是移动迭代器反向遍历序列。标准容器上的迭代器既支持自增运算,也支持自减运算。但是,流迭代器却不然,由于不能反向遍历流,因此流迭代器不能创建反向迭代器。

2. 反向迭代器与其他迭代器之间的关系

假设有一个名为 line 的 string 对象,存储以逗号分隔的单词列表。我们希望输出 line 中的第一个单词。使用 find 可很简单地实现这个任务:

```
// find first element in a comma-separated list
string::iterator comma = find(line.begin(), line.end(), ',');
cout << string(line.begin(), comma) << endl;
```

如果在 line 中有一个逗号,则 comma 指向这个逗号;否则,comma 的值为 line.end()。在输出 string 对象中从 line.begin()到 comma 的内容时,从头开始输出字符直到遇到逗号为止。

413 如果该 string 对象中没有逗号,则输出整个 string 字符串。

如果要输出列表中最后一个单词,可使用反向迭代器:

```
// find last element in a comma-separated list
string::reverse_iterator rcomma =
    find(line.rbegin(), line.rend(), ',');
```

因为此时传递的是 rbegin()和 rend(),这个函数调用从 line 的最后一个字符开始往回搜索。当 find 完成时,如果列表中有逗号,那么 rcomma 指向其最后一个逗号,即指向反向搜索找到的第一个逗号。如果没有逗号,则 rcomma 的值为 line.rend()。

在尝试输出所找到的单词时,有趣的事情发生了。直接尝试:

```
// wrong: will generate the word in reverse order
cout << string(line.rbegin(), rcomma) << endl;
```

会产生假的输出。例如,如果输入是:

FIRST,MIDDLE,LAST

则将输出 TSAL!

图 11-2 阐明了这个问题:使用反向迭代器时,以逆序从后向前处理 string 对象。为了得到正确的输出,必须将反向迭代器 line.rbegin()和 rcomma 转换为从前向后移动的普通迭代器。

其实没必要转换 `line.rbegin()`，因为我们知道转换的结果必定是 `line.end()`。只需调用所有反向迭代器类型都提供的成员函数 `base` 转换 `rcomma` 即可：

```
// ok: get a forward iterator and read to end of line
cout << string(rcomma.base(), line.end()) << endl;
```

假设还是前面给出的输入，该语句将如愿输出 LAST。

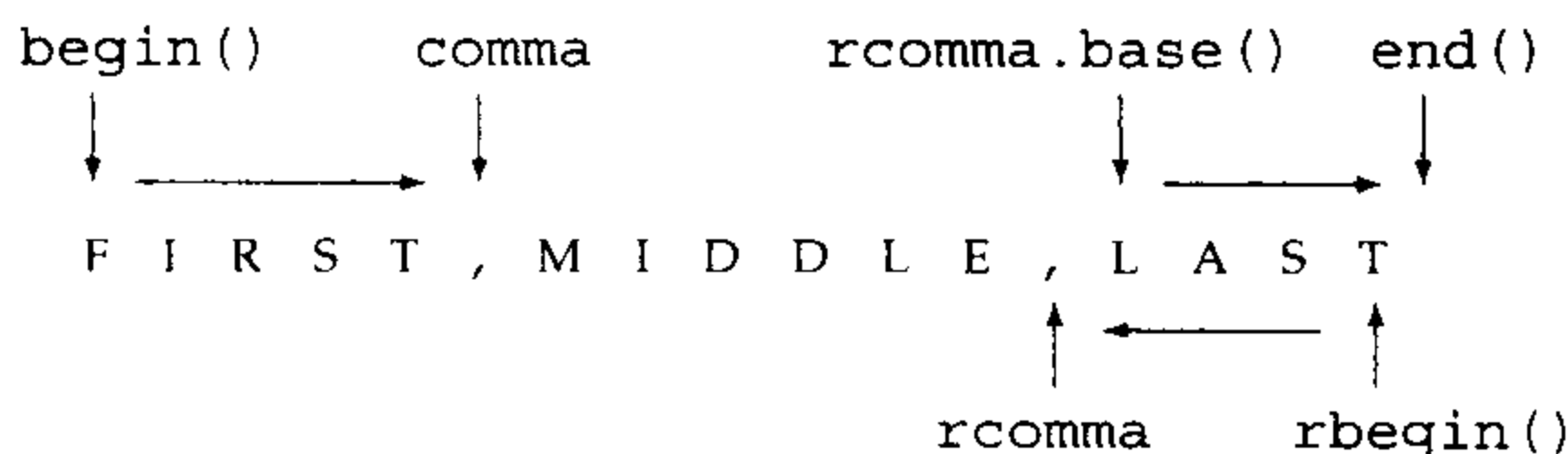


图 11-2 反向迭代器与普通迭代器之间的关系

图 11-2 显示的对象直观地解释了普通迭代器与反向迭代器之间的关系。例如，正如 `line_rbegin()` 和 `line.end()` 一样，`rcomma` 和 `rcomma.base()` 也指向不同的元素。为了确保正向和反向处理元素的范围相同，这些区别是必要的。从技术上来说，设计普通迭代器与反向迭代器之间的关系是为了适应左闭合范围（9.2.1 节）这个性质的，所以，`[ line_rbegin(), rcomma )` 和 `[ rcomma.base(), line.end() )` 标记的是 `line` 中的相同元素。

414



反向迭代器用于表示范围，而所表示的范围是不对称的，这个事实可推导出一个重要的结论：使用普通的迭代器对反向迭代器进行初始化或赋值时，所得到的迭代器并不是指向原迭代器所指向的元素。

习题

习题 11.19 编写程序使用 `reverse_iterator` 对象以逆序输出 `vector` 容器对象的内容。

习题 11.20 现在，使用普通的迭代器逆序输出上题中对象的元素。

习题 11.21 使用 `find` 在一个 `int` 型的 `list` 中寻找值为 0 的最后一个元素。

习题 11.22 假设有一个存储了 10 个元素的 `vector` 对象，将其中第 3~第 7 个位置上的元素以逆序复制给 `list` 对象。

11.3.4 const 迭代器

细心的读者可能已经注意到，在 11.1 节使用 `find` 的程序中，我们将 `result` 定义为 `const_iterator` 类型。这样做是因为我们不希望使用这个迭代器来修改容器中的元素。

另一方面，虽然 11.2.1 节的程序也不打算改变容器内的任何元素，但是它却使用了普通的非 `const` 迭代器来保存 `find_first_of` 的返回值。这两种处理存在细微的差别，值得解释一下。

原因是，在第二个例子中，程序将迭代器用作 `find_first_of` 的实参：

```
find_first_of(it, roster1.end(),
              roster2.begin(), roster2.end())
```

该函数调用的输入范围由 `it` 和调用 `roster1.end()` 返回的迭代器指定。算法要求用于指定范围的两个迭代器必须具有完全一样的类型。`roster1.end()` 返回的迭代器依赖于 `roster1` 的类型。如果该容器是 `const` 对象,则返回的迭代器是 `const_iterator` 类型;否则,就是普通的 `iterator` 类型。在这个程序中, `roster1` 不是 `const` 对象,因而 `end` 返回的只是一个普通的迭代器。

如果我们将 `it` 定义为 `const_iterator`,那么 `find_first_of` 的调用将无法编译。用来指定范围的两个迭代器的类型不相同。`it` 是 `const_iterator` 类型的对象,而 `rotser1.end()` 返回的则是一个 `iterator` 对象。

415

11.3.5 五种迭代器

迭代器定义了常用的操作集,但有些迭代器具有比其他迭代器更强大的功能。例如, `ostream_iterator` 只支持自增、解引用和赋值运算,而 `vector` 容器提供的迭代器除了这些运算,还支持自减、关系和算术运算。因此,迭代器可根据所提供的操作集进行分类。

类似地,还可根据算法要求它的迭代器提供什么类型的操作,对算法分类。有一些算法,例如 `find`,只要求迭代器提供读取所指向内容和自增的功能。另一些算法,比如 `sort`,则要求其迭代器有读、写和随机访问元素的能力。算法要求的迭代器操作分为五个类别,分别对应表 11-3 列出的五种迭代器。

表 11-3 迭代器种类

输入迭代器	读,不能写;只支持自增运算
输出迭代器	写,不能读;只支持自增运算
前向迭代器	读和写;只支持自增运算
双向迭代器	读和写;支持自增和自减运算
随机访问迭代器	读和写;支持完整的迭代器算术运算

(1) **输入迭代器** (input iterator) 可用于读取容器中的元素,但是不保证能支持容器的写入操作。输入迭代器必须至少提供下列支持。

- 相等和不等操作符 (`==`, `!=`), 比较两个迭代器。
- 前置和后置的自增运算 (`++`), 使迭代器向前递进指向下一个元素。
- 用于读取元素的解引用操作符 (`*`), 此操作符只能出现在赋值运算的右操作数上。
- 箭头操作符 (`->`), 这是 `(*it).member` 的同义语,也就是说,对迭代器进行解引用来获取其所关联的对象的成员。

输入迭代器只能顺序使用;一旦输入迭代器自增了,就无法再用它检查之前的元素。要求在这个层次上提供支持的泛型算法包括 `find` 和 `accumulate`。标准库 `istream_iterator` 类型是输入迭代器。

(2) **输出迭代器** (output iterator) 可视为与输入迭代器功能互补的迭代器;输出迭代器可用于向容器写入元素,但是不保证能支持读取容器内容。输出迭代器要求:

- 前置和后置的自增运算 (`++`), 使迭代器向前递进指向下一个元素。
- 解引用操作符 (`*`), 此操作符只能出现在赋值运算的左操作数上。给解引用的输出迭代

器赋值，将对该迭代器所指向的元素做写入操作。

416

输出迭代器可以要求每个迭代器的值必须正好写入一次。使用输出迭代器时，对于指定的迭代器值应该使用一次*运算，而且只能用一次。输出迭代器一般用作算法的第三个实参，标记起始写入的位置。例如，copy 算法使用一个输出迭代器作为它的第三个实参，将输入范围内的元素复制到输出迭代器指定的目标位置。标准库 ostream_iterator 类型是输出迭代器。

(3) 前向迭代器 (forward iterator) 用于读写指定的容器。这类迭代器只会以一个方向遍历序列。前向迭代器支持输入迭代器和输出迭代器提供的所有操作，除此之外，还支持对同一个元素的多次读写。可复制前向迭代器来记录序列中的一个位置，以便将来返回此处。需要前向迭代器的泛型算法包括 replace。

(4) 双向迭代器 (bidirectional iterator) 从两个方向读写容器。除了提供前向迭代器的全部操作之外，双向迭代器还提供前置和后置的自减运算 (--)。需要使用双向迭代器的泛型算法包括 reverse。所有标准库容器提供的迭代器都至少达到双向迭代器的要求。

(5) 随机访问迭代器 (random-access iterator) 提供在常量时间内访问容器任意位置的功能。这种迭代器除了支持双向迭代器的所有功能之外，还支持下面的操作：

- 关系操作符 <、<=、> 和 >=，比较两个迭代器的相对位置。
- 迭代器与整型数值 n 之间的加法和减法操作符 +、+=、- 和 -=，结果是迭代器在容器中向前（或退回）n 个元素。
- 两个迭代器之间的减法操作符 (-)，得到两个迭代器间的距离。
- 下标操作符 iter[n]，这是 *(iter + n) 的同义词。

需要随机访问迭代器的泛型算法包括 sort 算法。vector、deque 和 string 迭代器是随机访问迭代器，用作访问内置数组元素的指针也是随机访问迭代器。

除了输出迭代器，其他类别的迭代器形成了一个层次结构：需要低级类别迭代器的地方，可使用任意一种更高级的迭代器。对于需要输入迭代器的算法，可传递前向、双向或随机访问迭代器调用该算法。调用需要随机访问迭代器的算法时，必须传递随机访问迭代器。

map、set 和 list 类型提供双向迭代器，而 string、vector 和 deque 容器上定义的迭代器都是随机访问迭代器，用作访问内置数组元素的指针也是随机访问迭代器。istream_iterator 是输入迭代器，而 ostream_iterator 则是输出迭代器。

417

关键概念：关联容器与算法

尽管 map 和 set 类型提供双向迭代器，但关联容器只能使用算法的一个子集。问题在于：关联容器的键是 const 对象。因此，关联容器不能使用任何写序列元素的算法。只能使用与关联容器绑在一起的迭代器来提供用于读操作的实参。



提示 在处理算法时，最好将关联容器上的迭代器视为支持自减运算的输入迭代器，而不是完整的双向迭代器。

C++ 标准为所有泛型和算术算法的每一个迭代器形参指定了范围最小的迭代器种类。例如，

find (以只读方式单步遍历容器) 至少需要一个输入迭代器。replace 函数至少需要一对前向迭代器。replace_copy 函数的头两个迭代器必须至少是前向迭代器, 第三个参数代表输出目标, 必须至少是输出迭代器。

对于每一个形参, 迭代器必须保证最低功能。将支持更少功能的迭代器传递给函数是错误的; 而传递更强功能的迭代器则没问题。



向算法传递无效的迭代器类别所引起的错误, 无法保证会在编译时被捕获到。

习题

习题 11.23 列出五种迭代器类型及其各自支持的操作。

习题 11.24 list 容器拥有什么类型的迭代器? 而 vector 呢?

习题 11.25 你认为 copy 算法需要使用哪种迭代器? 而 reverse 和 unique 呢?

习题 11.26 解释下列代码错误的原因, 指出哪些错误可以在编译时捕获。

```
(a) string sa[10];
    const vector<string> file_names(sa, sa+6);
    vector<string>::iterator it = file_names.begin()+2;

(b) const vector<int> ivec;
    fill(ivec.begin(), ivec.end(), ival);

(c) sort(ivec.begin(), ivec.rend());

(d) sort(ivec1.begin(), ivec2.end());
```

418

11.4 泛型算法的结构

正如所有的容器都建立在一致的设计模式上一样, 算法也具有共同的设计基础。理解标准算法库的设计基础有利于学习和使用算法。C++提供了超过一百个算法, 了解它们的结构显然要比死记所有的算法更好。

算法最基本的性质是需要使用的迭代器种类。所有算法都指定了它的每个迭代器形参可使用的迭代器类型。如果形参必须为随机访问迭代器, 则可提供 vector 或 deque 类型的迭代器, 或者提供指向数组的指针。而其他容器的迭代器不能用在这类算法上。

另一种为算法分类的方法, 则如本章开头介绍的一样, 根据对元素的操作将算法分为下面几种:

- 只读算法, 不改变元素的值和顺序。
- 给指定元素赋新值的算法。
- 将一个元素的值移给另一个元素的算法。

正如本节后续部分所介绍的, C++还提供了另外两种算法模式: 一种模式由算法所带的形参定义; 另一种模式则通过两种函数命名和重载的规范定义。

11.4.1 算法的形参模式

任何其他的算法分类都含有一组形参规范。理解这些形参规范有利于学习新的算法——只要知道形参的含义，就可专注于了解算法实现的操作。大多数算法采用下面四种形式之一：

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

其中，alg 是算法的名字，beg 和 end 指定算法操作的元素范围。我们通常将该范围称为算法的“输入范围”。尽管几乎所有算法都有输入范围，但算法是否使用其他形参取决于它所执行的操作。这里列出了比较常用的其他形参：dest、beg2 和 end2，它们都是迭代器。这些迭代器在使用时，充当类似的角色。除了这些迭代器形参之外，有些算法还带有其他的非迭代器形参，它们是这些算法特有的。

1. 带有单个目标迭代器的算法

dest 形参是一个迭代器，用于指定存储输出数据的目标对象。算法假定无论需要写入多少个元素都是安全的。

419



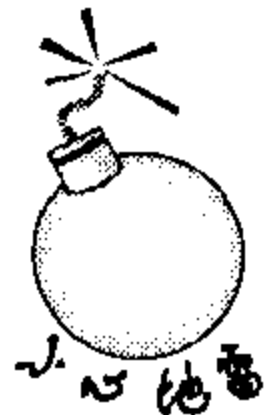
调用这些算法时，必须确保输出容器有足够大的容量存储输出数据，这正是通常要使用插入迭代器或者 ostream_iterator 来调用这些算法的原因。如果使用容器迭代器调用这些算法，算法将假定容器里有足够多个需要的元素。

如果 dest 是容器上的迭代器，则算法将输出内容写到容器中已存在的元素上。更普遍的使用法是，将 dest 与某个插入迭代器（11.3.1 节）或者 ostream_iterator 绑定在一起。插入迭代器在容器中添加元素，以确保容器有足够的空间存储输出。ostream_iterator 则实现写输出流的功能，无需考虑所写的元素个数。

2. 带第二个输入序列的算法

有一些算法带有一个 beg2 迭代器形参，或者同时带有 beg2 和 end2 迭代器形参，来指定它的第二个输入范围。这类算法通常将联合两个输入范围的元素来完成计算功能。算法同时使用 beg2 和 end2 时，这些迭代器用于标记完整的第二个范围。也就是说，此时，算法完整地指定了两个范围：beg 和 end 标记第一个输入范围，而 beg2 和 end2 则标记第二个输入范围。

带有 beg2 而不带 end2 的算法将 beg2 视为第二个输入范围的首元素，但没有指定该范围的最后一个元素。这些算法假定以 beg2 开始的范围至少与 beg 和 end 指定的范围一样大。



与写入 dest 的算法一样，只带有 beg2 的算法也假定以 beg2 开始的序列与 beg 和 end 标记的序列一样大。

11.4.2 算法的命名规范

标准库使用一组相同的命名和重载规范，了解这些规范有助于更容易地学习标准库。它们包括两种重要模式：第一种模式包括测试输入范围内元素的算法，第二种模式则应用于对输入范围

内元素重新排序的算法。

1. 区别带有一个值或一个谓词函数参数的算法版本

很多算法通过检查其输入范围内的元素实现其功能。这些算法通常要用到标准关系操作符：`==` 或 `<`。其中的大部分算法会提供第二个版本的函数，允许程序员提供比较或测试函数取代操作符的使用。

重新对容器元素排序的算法要使用`<`操作符。这些算法的第二个重载版本带有一个额外的形参，表示用于元素排序的不同运算：

```
sort (beg, end);           // use < operator to sort the elements
sort (beg, end, comp);     // use function named comp to sort the elements
```

检查指定值的算法默认使用`==`操作符。系统为这类算法提供另外命名的（而非重载的）版本，带有谓词函数（11.2.3 节）形参。带有谓词函数形参的算法，其名字带有后缀`_if`：

```
find(beg, end, val);       // find first instance of val in the input range
find_if(beg, end, pred);   // find first instance for which pred is true
```

上述两个算法都在输入范围内寻找指定元素的第一个实例。其中，`find` 算法查找一个指定的值，而 `find_if` 算法则用于查找一个使谓词函数 `pred` 返回非零值的元素。

标准库为这些算法提供另外命名的版本，而非重载版本，其原因在于这两种版本的算法带有相同数目的形参。对于排序算法，只要根据参数的个数就很容易消除函数调用的歧义。而对于查找指定元素的算法，不管检查的是一个值还是谓词函数，函数调用都需要相同个数的参数。此时，如果使用重载版本，则可能导致二义性（7.8.2 节），尽管这个可能出现的几率很低。因此，标准库为这些算法提供两种不同名字的版本，而没有使用重载。

2. 区别是否实现复制的算法版本

无论算法是否检查它的元素值，都可能重新排列输入范围内的元素。在默认情况下，这些算法将重新排列的元素写回其输入范围。标准库也为这些算法提供另外命名的版本，将元素写到指定的输出目标。此版本的算法在名字中添加了`_copy`后缀：

```
reverse(beg, end);
reverse_copy(beg, end, dest);
```

`reverse` 函数的功能就如它的名字所意味的：将输入序列中的元素反向重新排列。其中，第一个函数版本将自己的输入序列中的元素反向重排。而第二个版本，`reverse_copy`，则复制输入序列的元素，并将它们以逆序存储到 `dest` 开始的序列中。

习题

习题 11.27 标准库定义了下面的算法：

```
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);
```

只根据这些函数的名字和形参，描述这些算法的功能。

习题 11.28 假设 `lst` 是存储了 100 个元素的容器。请解释下面的程序段，并修正你认为的错误。

```
vector<int> vec1;
reverse_copy(lst.begin(), lst.end(), vec1.begin());
```

11.5 容器特有的算法

`list` 容器上的迭代器是双向的，而不是随机访问类型。由于 `list` 容器不支持随机访问，因此，在此容器上不能使用需要随机访问迭代器的算法。这些算法包括 `sort` 及其相关的算法。还有一些其他的泛型算法，如 `merge`、`remove`、`reverse` 和 `unique`，虽然可以用在 `list` 上，但却付出了性能上的代价。如果这些算法利用 `list` 容器实现的特点，则可以更高效地执行。

421

如果可以结合利用 `list` 容器的内部结构，则可能编写出更快的算法。与其他顺序容器所支持的操作相比，标准库为 `list` 容器定义了更精细的操作集合，使它不必只依赖于泛型操作。表 11-4 列出了 `list` 容器特有的操作，其中不包括要求支持双向或更弱的迭代器类型的泛型算法，这类泛型算法无论是用在 `list` 容器上，还是用在其他容器上，都具有相同的效果。

表 11-4 `list` 容器特有的操作

```
lst.merge(lst2)
```

```
lst.merge(lst2, comp)
```

将 `lst2` 的元素合并到 `lst` 中。这两个 `list` 容器对象都必须排序。`lst2` 中的元素将被删除。合并后，`lst2` 为空。返回 `void` 类型。第一个版本使用 `<` 操作符，而第二个版本则使用 `comp` 指定的比较运算

```
lst.remove(val)
```

```
lst.remove_if(unaryPred)
```

调用 `lst.erase` 删除所有等于指定值或使指定的谓词函数返回非零值的元素。返回 `void` 类型

```
lst.reverse()
```

反向排列 `lst` 中的元素

```
lst.sort()
```

对 `lst` 中的元素排序

```
lst.splice(iter, lst2)
```

```
lst.splice(iter, lst2, iter2)
```

```
lst.splice(iter, beg, end)
```

将 `lst2` 的元素移到 `lst` 中迭代器 `iter` 指向的元素前面。在 `lst2` 中删除移出的元素。第一个版本将 `lst2` 的所有元素移到 `lst` 中；合并后，`lst2` 为空。`lst` 和 `lst2` 不能是同一个 `list` 对象。第二个版本只移动 `iter2` 所指向的元素，这个元素必须是 `lst2` 中的元素。在这种情况下，`lst` 和 `lst2` 可以是同一个 `list` 对象。也就是说，可在一个 `list` 对象中使用 `splice` 运算移动一个元素。第三个版本移动迭代器 `beg` 和 `end` 标记的范围内的元素。`beg` 和 `end` 照例必须指定一个有效的范围。这两个迭代器可标记任意 `list` 对象内的范围，包括 `lst`。当它们指定 `lst` 的一段范围时，如果 `iter` 也指向这个范围内的一个元素，则该运算未定义

```
lst.unique()
```

```
lst.unique(binaryPred)
```

调用 `erase` 删除同一个值的连续副本。第一个版本使用 `==` 操作符判断元素是否相等；第二个版本则使用指定的谓词函数实现判断



对于 list 对象，应该优先使用 list 容器特有的成员版本，而不是泛型算法。

大多数 list 容器特有的算法类似于其泛型形式中已经见过的相应的算法，但并不相同：

```
l.remove(val);           // removes all instances of val from l
l.remove_if(pred);       // removes all instances for which pred is true from l
l.reverse();             // reverses the order of elements in l
l.sort();                // use element type < operator to compare elements
l.sort(comp);            // use comp to compare elements
l.unique();              // uses element == to remove adjacent duplicates
l.unique(comp);          // uses comp to remove duplicate adjacent copies
```

list 容器特有的算法与其泛型算法版本之间有两个至关重要的差别。其中一个差别是 remove 和 unique 的 list 版本修改了其关联的基础容器：真正删除了指定的元素。例如，list::unique 将 list 中第二个和后续重复的元素删除出该容器。



与对应的泛型算法不同，list 容器特有的操作能添加和删除元素。

另一个差别是 list 容器提供的 merge 和 splice 运算会破坏它们的实参。使用 merge 的泛型算法版本时，合并的序列将写入目标迭代器指向的对象，而它的两个输入序列保持不变。但是，使用 list 容器的 merge 成员函数时，则会破坏它的实参 list 对象——当实参对象的元素合并到调用 merge 函数的 list 对象时，实参对象的元素被移出并删除。

习题

习题 11.29 用 list 容器取代 vector 重新实现 11.2.3 节编写的排除重复单词的程序。

小结

C++ 标准化过程做出的更重要的贡献之一是：创建和扩展了标准库。容器和算法库是标准库的基础。标准库定义了超过一百个算法。幸运的是，这些算法具有相同的结构，使它们更易于学习和使用。

算法与类型无关：它们通常在一个元素序列上操作，这些元素可以存储在标准库容器类型、内置数组甚至是生成的序列（例如读写流所生成的序列）上。算法基于迭代器操作，从而实现类型无关性。大多数算法使用一对指定元素范围的迭代器作为其头两个实参。其他的迭代器实参包括指定输出目标的输出迭代器，或者用于指定第二个输入序列的另一个或一对迭代器。

迭代器可通过其所支持的操作来分类。标准库定义了五种迭代器类别：输入、输出、前向、双向和随机访问迭代器。如果一个迭代器支持某种迭代器类别要求的运算，则该迭代器属于这个迭代器类别。

正如迭代器根据操作来分类一样，算法的迭代器形参也通过其所要求的迭代器操作来分类。只需要读取其序列的算法通常只要求输入迭代器的操作。而写目标迭代器的算法则通常只要求输出迭代器的操作，依此类推。

查找某个值的算法通常提供第二个版本，用于查找使谓词函数返回非零值的元素。对于这种算法，第二个版本的函数名字以 `_if` 后缀标识。类似地，很多算法提供所谓的复制版本，将（修改过的）元素写到输出序列，而不是写回输入范围。这种版本的名字以 `_copy` 结束。

第三种模式是考虑算法是否对元素读、写或者重新排序。算法从不直接改变它所操纵的序列的大小。（如果算法的实参是插入迭代器，则该迭代器会添加新元素，但算法并不直接这么做。）算法可以从一个位置将元素复制到另一个位置，但不能直接添加或删除元素。

术语

back_inserter 形参为指向容器的引用的迭代器适配器，生成使用 `push_back` 为指定容器添加元素的插入迭代器。

bidirectional iterator（双向迭代器）除了提供前向迭代器相同的操作之外，还支持使用 `--` 操作符向后遍历序列。

forward iterator（前向迭代器）可读写元素的迭代器，但不支持 `--` 操作符。

front_inserter 一种迭代器适配器，生成使用 `push_front` 在指定容器的开始位置添加新元素的插入迭代器。

generic algorithm（泛型算法）与类型无关的算法。

input iterator（输入迭代器）只能读不能写元素的迭代器。

insert iterator（插入迭代器）使用容器操作插入元素而不是覆写元素的迭代器。给插入迭代器赋值，等效于将具有所赋值的新元素插入到序列中。

inserter（插入器）一种迭代器适配器，形参为一个迭代器和一个指向容器的引用，生成使用 `insert` 为容器添加元素的插入迭代器，新

元素插入在该适配器的迭代器形参所指向的元素前面。

istream_iterator 读输入流的流迭代器。

iterator category（迭代器种类）基于迭代器所支持的操作，在概念上对迭代器进行分类。迭代器种类形成了一个层次结构，功能较强的迭代器种类提供比它弱的迭代器的所有操作。算法使用迭代器种类来指定它的迭代器实参必须支持什么操作。只要迭代器至少提供这个层次的操作，就可以用于该算法。例如，一些算法只要求输入迭代器，则可以使用除了输出迭代器之外的任意迭代器调用这样的算法。而要求使用随机访问迭代器的算法只能用在支持随机访问运算的迭代器上。

off-the-end iterator（超出末端迭代器）一种迭代器，用于标记序列中一个元素范围的结束位置。超出末端迭代器用作结束遍历的“哨兵”，指向范围内最后一个元素的下一位置。超出末端迭代器可能指向不存在的元素，因此永远不能做解引用运算。

ostream_iterator 写输出流的迭代器。

output iterator（输出迭代器）只能写不能读元素的迭代器。

predicate (谓词) 其返回类型可转换为 `bool` 值的函数。通常被泛型算法用于检查元素。标准库所使用的谓词函数不是一元 (需要一个实参) 的就是二元的 (需要两个实参)。

random-access iterator (随机访问迭代器)

除了支持双向迭代器相同的操作之外, 还提供了使用关系运算比较迭代器值的能力, 以及在

迭代器上做算术运算的能力。因此, 这类迭代器支持随机访问元素。

reverse iterator (反向迭代器) 向后遍历序列的迭代器。这些迭代器颠倒了 `++` 和 `--` 的含义。

stream iterator (流迭代器) 可与流绑定在一起的迭代器。

424
425

类和数据抽象

目录

第 12 章 类	367
第 13 章 复制控制	405
第 14 章 重载操作符与转换	429

在大多数 C++ 程序中，类都是至关重要的：我们能够使用类来定义为要解决的问题定制的数据类型，从而得到更加易于编写和理解的应用程序。设计良好的类类型可以像内置类型一样容易使用。

类定义了数据成员和函数成员：数据成员用于存储与该类类型的对象相关联的状态，而函数成员则负责执行赋予数据意义的操作。通过类我们能够将实现和接口分离，用接口指定类所支持的操作，而实现的细节只需类的实现者了解或关心。这种分离可以减少使编程冗长乏味和容易出错的那些繁琐工作。

类类型常被称为抽象数据类型。抽象数据类型将数据（即状态）和作用于状态的操作视为一个单元。我们可以抽象地考虑类该做些什么，而无须知道类如何去完成这些操作。抽象数据类型是面向对象编程和泛型编程的基础。

第 12 章开始详细地介绍如何定义类，包括类的使用中非常基本的主题：类作用域、数据隐藏和构造函数。此外，还介绍了类的一些新特征：友元、使用隐含的 `this` 指针，以及静态（`static`）和可变（`mutable`）成员的作用。

C++中的类能够控制在初始化、复制、赋值和销毁对象时发生的操作。在此方面，C++不同于许多其他语言，它们大多没有赋予类设计者控制这些操作的能力。第 13 章讨论了这些主题。

第 14 章考察了操作符重载，允许将类类型的操作数与内置操作符一起使用。利用操作符重载，在 C++中创建新的类型，就像创建内置类型一样。此外，还介绍了另一种特殊的类成员函数——转换函数，这种函数定义了类类型对象之间的隐式转换。编译器应用这些转换就像它们是在内置类型之间发生的转换一样。

第 12 章

类

目录

12.1 类的定义和声明.....	368
12.2 隐含的 this 指针	376
12.3 类作用域.....	380
12.4 构造函数.....	385
12.5 友元.....	396
12.6 static 类成员	398
小结.....	403
术语.....	403

在 C++ 中，用类来定义自己的抽象数据类型（abstract data type）。通过定义类型来对应所要解决的问题中的各种概念，可以使我们更容易编写、调试和修改程序。

本章进一步讨论类，并将更详细地阐述数据抽象的重要性。数据抽象能够隐藏对象的内部表示，同时仍然允许执行对象的公有（public）操作。

我们也将进一步解释类作用域、构造函数以及 this 指针。此外，还要介绍与类有关的三个新特征：友元、可变成员和静态成员。

类是 C++ 中最重要的特征。C++ 语言的早期版本被命名为“带类的 C (C with Classes)”，以强调类机制的中心作用。随着语言的演变，创建类的配套支持也在不断增加。语言设计的主要目标已变成提供这样一些特性：允许程序员定义自己的类型，它们用起来与内置类型一样容易和直观。本章将介绍类的许多基本特征。

12.1 类的定义和声明

从第 1 章开始，程序中就已经使用了类。已经用过的标准库类型，比如 `vector`，`istream` 和 `string`，都是类类型。还定义了一些简单的类，如 `Sales_item` 和 `TextQuery` 类。为了扼要重述，再来看看 `Sales_item` 类：

```
class Sales_item {
public:
    // operations on Sales_item objects
    double avg_price() const;
    bool same_isbn(const Sales_item &rhs) const
        { return isbn == rhs.isbn; }
    // default constructor needed to initialize members of built-in type
    Sales_item(): units_sold(0), revenue(0.0) { }
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};

double Sales_item::avg_price() const
{
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

12.1.1 类定义：扼要重述

在 2.8 节和 7.7 节中编写这个类时，已经学习了有关类的一些知识。



最简单地说来，类就是定义了一个新的类型和一个新的作用域。

1. 类成员

430

每个类可以没有成员，也可以定义多个成员，成员可以是数据、函数或类型别名。

一个类可以包含若干公有的、私有的和受保护的部分。我们已经使用过 `public` 和 `private` 访问标号：在 `public` 部分定义的成员可被使用该类型的所有代码访问；在 `private` 部分定义的成员可被其他类成员访问。在第 15 章讨论继承时将进一步探讨 `protected`。

所有成员必须在类的内部声明，一旦类定义完成后，就没有任何方式可以增加成员了。

2. 构造函数

创建一个类类型的对象时，编译器会自动使用一个构造函数（2.3.3 节）来初始化该对象。构造函数是一个特殊的、与类同名的成员函数，用于给每个数据成员设置适当的初始值。

构造函数一般应使用一个构造函数初始化列表（7.7.3 节），来初始化对象的数据成员：

```
// default constructor needed to initialize members of built-in type
Sales_item(): units_sold(0), revenue(0.0) { }
```

构造函数初始化列表由成员名和带括号的初始值组成，跟在构造函数的形参表之后，并以冒号开头。

3. 成员函数

在类内部，声明成员函数是必需的，而定义成员函数则是可选的。在类内部定义的函数默认为 inline（7.6 节）。

在类外部定义的成员函数必须指明它们是在类的作用域中。Sales_item::avg_price 的定义使用作用域操作符（1.2.2 节）来指明这是 Sales_item 类中 avg_price 函数的定义。

成员函数有一个附加的隐含实参，将函数绑定到调用函数的对象——当我们编写下面的函数时：

```
trans.avg_price()
```

就是在调用名为 trans 的对象的 avg_price 函数。如果 trans 是一个 Sales_item 对象，则在 avg_price 函数内部对 Sales_item 类成员的引用就是对 trans 成员的引用。

将关键字 const 加在形参表之后，就可以将成员函数声明为常量：

```
double avg_price() const;
```

const 成员不能改变其所操作的对象的数据成员。const 必须同时出现在声明和定义中，若只出现在其中一处，就会出现一个编译时错误。

431

习题

习题 12.1 编写一个名为 Person 的类，表示人的名字和地址。使用 string 来保存每个元素。

习题 12.2 为 Person 提供一个接受两个 string 参数的构造函数。

习题 12.3 提供返回名字和地址的操作。这些函数应为 const 吗？解释你的选择。

习题 12.4 指明 Person 的哪个成员应声明为 public，哪个成员应声明为 private。解释你的选择。

12.1.2 数据抽象和封装

类背后蕴涵的基本思想是数据抽象和封装。

数据抽象是一种依赖于接口和实现分离的编程（和设计）技术。类设计者必须关心类是如何实现的，但使用该类的程序员不必了解这些细节。相反，使用一个类型的程序员仅需了解类型的接口，他们可以抽象地考虑该类型做什么，而不必具体地考虑该类型如何工作。

封装是一项将低层次的元素组合起来形成新的、高层次实体的技术。函数是封装的一种形式：函数所执行的细节行为被封装在函数本身这个更大的实体中。被封装的元素隐藏了它们的实现细

节——可以调用一个函数但不能访问它所执行的语句。同样地，类也是一个封装的实体：它代表若干成员的聚集，大多数（良好设计的）类类型隐藏了实现该类型的成员。

标准库类型 `vector` 同时具备数据抽象和封装的特性。在使用方面它是抽象的，只需考虑它的接口，即它能执行的操作。它又是封装的，因为我们既无法了解该类型如何表示的细节，也无法访问其任意的实现制品。另一方面，数组在概念上类似于 `vector`，但既不是抽象的，也不是封装的。可以通过访问存放数组的内存来直接操纵数组。

1. 访问标号实施抽象和封装

在 C++ 中，使用访问标号（2.8 节）来定义类的抽象接口和实施封装。一个类可以没有访问标号，也可以包含多个访问标号：

- 程序的所有部分都可以访问带有 `public` 标号的成员。类型的数据抽象视图由其 `public` 成员定义。
- 使用类的代码不可以访问带有 `private` 标号的成员。`private` 封装了类型的实现细节。

一个访问标号可以出现的次数通常是没有限制的。每个访问标号指定了随后的成员定义的访问级别。这个指定的访问级别持续有效，直至遇到下一个访问标号或看到类定义体的右花括号为止。

可以在任意的访问标号出现之前定义类成员。在类的左花括号之后、第一个访问标号之前定义成员的访问级别，其值依赖于类是如何定义的。如果类是用 `struct` 关键字定义的，则在第一个访问标号之前的成员是公有的；如果类是用 `class` 关键字定义的，则这些成员是私有的。

建议：具体类型和抽象类型

并非所有类型都必须是抽象的。标准库中的 `pair` 类就是一个实用的、设计良好的具体类而不是抽象类。具体类会暴露而非隐藏其实现细节。

一些类，例如 `pair`，确实没有抽象接口。`pair` 类型只是将两个数据成员捆绑成单个对象。在这种情况下，隐藏数据成员没有必要也没有明显的好处。在像 `pair` 这样的类中隐藏数据成员只会造成类型使用的复杂化。

尽管如此，这样的类型通常还是有成员函数。特别地，如果类具有内置类型或复合类型数据成员，那么定义构造函数来初始化这些成员就是一个好主意。类的使用者也可以初始化或赋值数据成员，但由类来做更不易出错。

2. 编程角色的不同类别

程序员经常会将运行应用程序的人看作“用户”。应用程序为最终“使用”它的用户而设计，并响应用户的反馈而完善。类也类似：类的设计者为类的“用户”设计并实现类。在这种情况下，“用户”是程序员，而不是应用程序的最终用户。

成功的应用程序的创建者会很好地理解和实现用户的需求。同样地，良好设计的、实用的类，其设计也要贴近类用户的需求。

另一方面，类的设计者与实现者之间的区别，也反映了应用程序的用户与设计和实现者之间的区分。用户只关心应用程序能否以合理的费用满足他们的需求。同样地，类的使用者只关心它

432

的接口。好的类设计者会定义直观和易用的类接口，而使用者只关心类中影响他们使用的那部分实现。如果类的实现速度太慢或给类的使用者加上负担，则必然引起使用者的关注。在良好设计的类中，只有类的设计者会关心实现。

在简单的应用程序中，类的使用者和设计者也许是同一个人。即使在这种情况下，保持角色区分也是有益的。设计类的接口时，设计者应该考虑的是如何方便类的使用；使用类的时候，设计者就不应该考虑类如何工作。

433



注意 C++程序员经常会将应用程序的用户和类的使用者都称为“用户”。

提到“用户”时，应该由上下文清楚地标明所指的是哪类用户。如果提到“用户代码”或 `Sales_item` 类的“用户”，指的就是使用类编写应用程序的程序员。如果提到书店应用程序的“用户”，那么指的是运行应用程序的书店管理人员。

关键概念：数据抽象和封装的好处

数据抽象和封装提供了两个重要优点：

- 避免类内部出现无意的、可能破坏对象状态的用户级错误。
- 随时间推移可以根据需求改变或缺陷（bug）报告来完善类实现，而无需改变用户级代码。

仅在类的私有部分定义数据成员，类的设计者就可以自由地修改数据。如果实现改变了，那么只需检查类代码来了解此变化可能造成的影响。如果数据为公有的，则任何直接访问原有数据成员的函数都可能遭到破坏。在程序可重新使用之前，有必要定位和重写依赖原有表示的那部分代码。

同样地，如果类的内部状态是私有的，则数据成员的改变只可能在有限的地方发生。避免数据中出现用户可能引入的错误。如果有缺陷会破坏对象的状态，就在局部位置搜寻缺陷：如果数据是私有的，那么只有成员函数可能对该错误负责。对错误的搜寻是有限的，从而大大方便了程序的维护和修正。

如果数据是私有的并且没有改变成员函数的接口，则操纵类对象的用户函数无须改变。



改变头文件中的类定义可有效地改变包含该头文件的每个源文件的程序文本，所以，当类发生改变时，使用该类的代码必须重新编译。

习题

习题 12.5 C++类支持哪些访问标号？在每个访问标号之后应定义哪种成员？如果有的话，在类的定义中，一个访问标号可以出现在何处以及可出现多少次？约束条件是什么？

习题 12.6 用 `class` 关键字定义的和用 `struct` 定义的类型有什么不同？

习题 12.7 什么是封装？为什么封装是有用的？

12.1.3 关于类定义的更多内容

434 迄今为止，所定义类都是简单的，然而通过这些类我们已经了解到 C++ 语言为类所提供的相当多的支持。本节的其余部分将阐述编写类的更多基础知识。

1. 同一类型的多个数据成员

正如我们所见，类的数据成员的声明类似于普通变量的声明。如果一个类具有多个同一类型的数据成员，则这些成员可以在一个成员声明中指定，这种情况下，成员声明和普通变量声明是相同的。

例如，可以定义一个名为 `Screen` 的类型来表示计算机上的窗口。每个 `Screen` 可以有一个保存窗口内容的 `string` 成员，以及三个 `string::size_type` 成员：一个指定光标当前停留的字符，另外两个指定窗口的高度和宽度。可以用如下方式定义这个类的成员：

```
class Screen {
public:
    // interface member functions
private:
    std::string contents;
    std::string::size_type cursor;
    std::string::size_type height, width;
};
```

2. 使用类型别名来简化类

除了定义数据和函数成员之外，类还可以定义自己的局部类型名字。如果为 `std::string::size_type` 提供一个类型别名，那么 `Screen` 类将是一个更好的抽象：

```
class Screen {
public:
    // interface member functions
    typedef std::string::size_type index;
private:
    std::string contents;
    index cursor;
    index height, width;
};
```

435 类所定义的类型名遵循任何其他成员的标准访问控制。将 `index` 的定义放在类的 `public` 部分，是因为希望用户使用这个名字。`Screen` 类的使用者不必了解用 `string` 实现的底层细节。定义 `index` 来隐藏 `Screen` 的实现细节。将这个类型设为 `public`，就允许用户使用这个名字。

3. 成员函数可被重载

这些类之所以简单，另一个方面也是因为它们只定义了几个成员函数。特别地，这些类都不需要定义其任意成员函数的重载版本。然而，像非成员函数一样，成员函数也可以被重载（7.8 节）。

重载操作符（14.9.5 节）有特殊规则，是个例外，成员函数只能重载本类的其他成员函数。类的成员函数与普通的非成员函数以及其他类中声明的函数不相关，也不能重载它们。重载的成员函数和普通函数应用相同的规则：两个重载成员的形参数量和类型不能完全相同。调用非成员重载函数所用到的函数匹配（7.8.2 节）过程也应用于重载成员函数的调用。

4. 定义重载成员函数

为了举例说明重载，可以给出 Screen 类的两个重载成员，用于从窗口返回一个特定字符。两个重载成员中，一个版本返回由当前光标指示的字符，另一个返回指定行列处的字符：

```
class Screen {
public:
    typedef std::string::size_type index;
    // return character at the cursor or at a given position
    char get() const { return contents[cursor]; }
    char get(index ht, index wd) const;
    // remaining members
private:
    std::string contents;
    index cursor;
    index height, width;
};
```

与任意的重载函数一样，给指定的函数调用提供适当数目和/或类型的实参来选择运行哪个版本：

```
Screen myscreen;
char ch = myscreen.get(); // calls Screen::get()
ch = myscreen.get(0,0); // calls Screen::get(index, index)
```

5. 显式指定 **inline** 成员函数

在类内部定义的成员函数，例如不接受实参的 get 成员，将自动作为 inline 处理。也就是说，当它们被调用时，编译器将试图在同一行内扩展该函数（7.6 节）。也可以显式地将成员函数声明为 inline：

436

```
class Screen {
public:
    typedef std::string::size_type index;
    // implicitly inline when defined inside the class declaration
    char get() const { return contents[cursor]; }
    // explicitly declared as inline; will be defined outside the class declaration
    inline char get(index ht, index wd) const;
    // inline not specified in class declaration, but can be defined inline later
    index get_cursor() const;
    // ...
};
// inline declared in the class declaration; no need to repeat on the definition
char Screen::get(index r, index c) const
{
    index row = r * width; // compute the row location
    return contents[row + c]; // offset by c to fetch specified character
}
// not declared as inline in the class declaration, but ok to make inline in definition
inline Screen::index Screen::get_cursor() const
{
    return cursor;
}
```

可以在类定义体内部指定一个成员为 inline，作为其声明的一部分。或者，也可以在类定义体外部的函数定义上指定 inline。在声明和定义处指定 inline 都是合法的。在类的外部定义

inline 的一个好处是可以使得类比较容易阅读。



像其他 inline 一样, inline 成员函数的定义必须在调用该函数的每个源文件中是可见的。不在类定义体内定义的 inline 成员函数, 其定义通常应放在有类定义的同头文件中。

习题

习题 12.8 将 Sales_item::avg_price 定义为内联函数。

习题 12.9 修改本节中给出的 Screen 类, 给出一个构造函数, 根据屏幕的高度、宽度和内容的值来创建 Screen。

习题 12.10 解释下述类中的每个成员:

```
class Record {
    typedef std::size_t size;
    Record(): byte_count(0) { }
    Record(size s): byte_count(s) { }
    Record(std::string s): name(s), byte_count(0) { }
    size byte_count;
    std::string name;
public:
    size get_count() const { return byte_count; }
    std::string get_name() const { return name; }
};
```

12.1.4 类声明与类定义

一旦遇到右花括号, 类的定义就结束了。并且一旦定义了类, 那以我们就知道了所有的类成员, 以及存储该类的对象所需的存储空间。在一个给定的源文件中, 一个类只能被定义一次。如果在多个文件中定义一个类, 那么每个文件中的定义必须是完全相同的。

将类定义放在头文件中, 可以保证在每个使用类的文件中以同样的方式定义类。使用头文件保护符 (header guard) (2.9.2 节), 来保证即使头文件在同一文件中被包含多次, 类定义也只出现一次。

可以声明一个类而不定义它:

```
class Screen; // declaration of the Screen class
```

这个声明, 有时称为前向声明 (forward declaration), 在程序中引入了类类型的 Screen。在声明之后、定义之前, 类 Screen 是一个不完全类型 (incomplete type), 即已知 Screen 是一个类型, 但不知道包含哪些成员。



不完全类型只能以有限方式使用。不能定义该类型的对象。不完全类型只能用于定义指向该类型的指针及引用, 或者用于声明 (而不是定义) 使用该类型作为形参类型或返回类型的函数。

在创建类的对象之前, 必须完整地定义该类。必须定义类, 而不只是声明类, 这样, 编译器

就会给类的对象预定相应的存储空间。同样地，在使用引用或指针访问类的成员之前，必须已经定义类。

为类的成员使用类声明

只有当类定义已经在前面出现过，数据成员才能被指定为该类类型。如果该类型是不完全类型，那么数据成员只能是指向该类类型的指针或引用。

438

因为只有当类定义体完成后才能定义类，因此类不能具有自身类型的数据成员。然而，只要类名一出现就可以认为该类已声明。因此，类的数据成员可以是指向自身类型的指针或引用：

```
class LinkScreen {
    Screen window;
    LinkScreen *next;
    LinkScreen *prev;
};
```



类的前向声明一般用来编写相互依赖的类。在 13.4 节中，我们将看到这种用法的一个例子。

习题

习题 12.11 定义两个类 x 和 y ， x 中有一个指向 y 的指针， y 中有一个 x 类型的对象。

习题 12.12 解释类声明与类定义之间的差异。何时使用类声明？何时使用类定义？

12.1.5 类对象

定义一个类时，也就是定义了一个类型。一旦定义了类，就可以定义该类型的对象。定义对象时，将为其分配存储空间，但（一般而言）定义类型时不进行存储分配：

```
class Sales_item {
public:
    // operations on Sales_item objects
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

定义了一个新的类型，但没有进行存储分配。当我们定义一个对象

```
Sales_item item;
```

时，编译器分配了足以容纳一个 `Sales_item` 对象的存储空间。`item` 指的就是那个存储空间。每个对象具有自己的类数据成员的副本。修改 `item` 的数据成员不会改变任何其他 `Sales_item` 对象的数据成员。

439

1. 定义类类型的对象

定义了一个类类型之后，可以按以下两种方式使用。

- 将类的名字直接用作类型名。
- 指定关键字 `class` 或 `struct`，后面跟着类的名字：

```
Sales_item item1;           // default initialized object of type Sales_item
class Sales_item item1;     // equivalent definition of item1
```

两种引用类类型的方法是等价的。第二种方法是从 C 继承而来的，在 C++ 中仍然有效。第一种更为简练，由 C++ 语言引入，使得类类型更容易使用。

2. 为什么类的定义以分号结束

我们在 2.8 节中指出，类的定义以分号结束。分号是必需的，因为在类定义之后可以接一个对象定义列表。定义必须以分号结束：

```
class Sales_item { /* ... */ };
class Sales_item { /* ... */ } accum, trans;
```



最佳
实践

通常，将对象定义成类定义的一部分是个坏主意。这样做，会使所发生的操作难以理解。对读者而言，将两个不同的实体（类和变量）组合在一个语句中，也会令人迷惑不解。

12.2 隐含的 this 指针

在 7.7.1 节中已经提到，成员函数具有一个附加的隐含形参，即指向该类对象的一个指针。这个隐含形参命名为 `this`，与调用成员函数的对象绑定在一起。成员函数不能定义 `this` 形参，而是由编译器隐含地定义。成员函数的函数体可以显式使用 `this` 指针，但不是必须这么做。如果对类成员的引用没有限定，编译器会将这种引用处理成通过 `this` 指针的引用。

1. 何时使用 `this` 指针

尽管在成员函数内部显式引用 `this` 通常是不必要的，但有一种情况下必须这样做：当我们需要将一个对象作为整体引用而不是引用对象的一个成员时。最常见的情况是在这样的函数中使用 `this`：该函数返回对调用该函数的对象的引用。

440

某种类可能具有某些操作，这些操作应该返回引用，`Screen` 类就是这样的。迄今为止，我们的类只有一对 `get` 操作。逻辑上，我们可以添加下面的操作。

- 一对 `set` 操作，将特定字符或光标指向的字符设置为给定值。
- 一个 `move` 操作，给定两个 `index` 值，将光标移至新位置。

理想情况下，希望用户能够将这些操作的序列连接成一个单独的表达式：

```
// move cursor to given position, and set that character
myScreen.move(4,0).set('#');
```

这个语句等价于：

```
myScreen.move(4,0);
myScreen.set('#');
```

2. 返回 `*this`

在单个表达式中调用 `move` 和 `set` 操作时，每个操作必须返回一个引用，该引用指向执行操作的那个对象：

```
class Screen {
```

```

public:
    // interface member functions
    Screen& move(index r, index c);
    Screen& set(char);
    Screen& set(index, index, char);
    // other members as before
};

```

注意, 这些函数的返回类型是 `Screen&`, 指明该成员函数返回对其自身类类型的对象的引用。每个函数都返回调用自己的那个对象。使用 `this` 指针来访问该对象。下面是对两个新成员的实现:

```

Screen& Screen::set(char c)
{
    contents[cursor] = c;
    return *this;
}
Screen& Screen::move(index r, index c)
{
    index row = r * width; // row location
    cursor = row + c;
    return *this;
}

```

函数中唯一需要关注的部分是 `return` 语句。在这两个操作中, 每个函数都返回 `*this`。在这些函数中, `this` 是一个指向非常量 `Screen` 的指针。如同任意的指针一样, 可以通过对 `this` 指针解引用来访问 `this` 指向的对象。

441

3. 从 `const` 成员函数返回 `*this`

在普通的非 `const` 成员函数中, `this` 的类型是一个指向类类型的 `const` 指针 (4.2.5 节)。可以改变 `this` 所指向的值, 但不能改变 `this` 所保存的地址。在 `const` 成员函数中, `this` 的类型是一个指向 `const` 类类型对象的 `const` 指针。既不能改变 `this` 所指向的对象, 也不能改变 `this` 所保存的地址。



不能从 `const` 成员函数返回指向类对象的普通引用。`const` 成员函数只能返回 `*this` 作为一个 `const` 引用。

例如, 我们可以给 `Screen` 类增加一个 `display` 操作。这个函数应该在给定的 `ostream` 上打印 `contents`。逻辑上, 这个操作应该是一个 `const` 成员。打印 `contents` 不会改变对象。如果将 `display` 作为 `Screen` 的 `const` 成员, 则 `display` 内部的 `this` 指针将是一个 `const Screen*` 类型的 `const`。

然而, 与 `move` 和 `set` 操作一样, 我们希望能够在一个操作序列中使用 `display`:

```

// move cursor to given position, set that character and display the screen
myScreen.move(4, 0).set('#').display(cout);

```

这个用法暗示了 `display` 应该返回一个 `Screen` 引用, 并接受一个 `ostream` 引用。如果 `display` 是一个 `const` 成员, 则它的返回类型必须是 `const Screen&`。

不幸的是，这个设计存在一个问题。如果将 `display` 定义为 `const` 成员，就可以在非 `const` 对象上调用 `display`，但不能将对 `display` 的调用嵌入到一个长表达式中。下面的代码将是非法的：

```
Screen myScreen;
// this code fails if display is a const member function
// display return a const reference; we cannot call set on a const
myScreen.display().set('*');
```

问题在于这个表达式是在由 `display` 返回的对象上运行 `set`。该对象是 `const`，因为 `display` 将其对象作为 `const` 返回。我们不能在 `const` 对象上调用 `set`。

4. 基于 `const` 的重载

为了解决这个问题，我们必须定义两个 `display` 操作：一个是 `const`，另一个不是 `const`。基于成员函数是否为 `const`，可以重载一个成员函数；同样地，基于一个指针形参是否指向 `const` (7.8.4 节)，可以重载一个函数。`const` 对象只能使用 `const` 成员。非 `const` 对象可以使用任一成员，但非 `const` 版本是一个更好的匹配。

在此，我们将定义一个名为 `do_display` 的 `private` 成员来打印 `Screen`。每个 `display` 操作都将调用此函数，然后返回调用自己的那个对象：

442

```
class Screen {
public:
    // interface member functions
    // display overloaded on whether the object is const or not
    Screen& display(std::ostream &os)
    { do_display(os); return *this; }
    const Screen& display(std::ostream &os) const
    { do_display(os); return *this; }
private:
    // single function to do the work of displaying a Screen,
    // will be called by the display operations
    void do_display(std::ostream &os) const
    { os << contents; }

    // as before
};
```

现在，当我们将 `display` 嵌入到一个长表达式中时，将调用非 `const` 版本。当我们 `display` 一个 `const` 对象时，就调用 `const` 版本：

```
Screen myScreen(5,3);
const Screen blank(5, 3);
myScreen.set('#').display(cout); // calls nonconst version
blank.display(cout);             // calls const version
```

5. 可变数据成员

有时（但不是很经常），我们希望类的数据成员（甚至在 `const` 成员函数内）可以修改。这可以通过将它们声明为 `mutable` 来实现。

可变数据成员永远都不能为 `const`，甚至当它是 `const` 对象的成员时也如此。因此，`const` 成员函数可以改变 `mutable` 成员。要将数据成员声明为可变的，必须将关键字 `mutable` 放在成

员声明之前:

```
class Screen {
public:
    // interface member functions
private:
    mutable size_t access_ctr; // may change in a const members
    // other data members as before
};
```

我们给 Screen 添加了一个新的可变数据成员 access_ctr。使用 access_ctr 来跟踪调用 Screen 成员函数的频繁程度:

```
void Screen::do_display(std::ostream& os) const
{
    ++access_ctr; // keep count of calls to any member function
    os << contents;
}
```

443

尽管 do_display 是 const，它也可以增加 access_ctr。该成员是可变成员，所以，任意成员函数，包括 const 函数，都可以改变 access_ctr 的值。

建议：用于公共代码的私有实用函数

有些读者可能会奇怪为什么要费力地单独定义一个 do_display 操作。毕竟，对 do_display 的调用并不比在 do_display 内部所做的操作更简单。为什么还要如此麻烦？我们这样做有下面几个原因。

(1) 一般愿望是避免在多个地方编写同样的代码。

(2) display 操作预期会随着类的演变而变得更复杂。当所涉及的动作变得更复杂时，只在一处而不是两处编写这些动作有更显著的意义。

(3) 很可能我们会希望在开发时给 do_display 增加调试信息，这些调试信息将会在代码的最终成品版本中去掉。如果只需要改变一个 do_display 的定义来增加或删除调试代码，这样做将更容易。

(4) 这个额外的函数调用不需要涉及任何开销。我们使 do_display 成为内联的，所以调用 do_display 与将代码直接放入 display 操作的运行时性能应该是相同的。

实际上，设计良好的 C++ 程序经常具有许多像 do_display 这样的小函数，它们被调用来完成一些其他函数的“实际”工作。

习题

习题 12.13 扩展 Screen 类以包含 move、set 和 display 操作。通过执行如下表达式来测试类:

```
// move cursor to given position, set that character and display the screen
myScreen.move(4,0).set('#').display(cout);
```

习题 12.14 通过 this 指针引用成员虽然合法，但却是多余的。讨论显式使用 this 指针访问成员的优缺点。

12.3 类作用域

每个类都定义了自己的新作用域和唯一的类型。在类的定义体内声明类成员，将成员名引入类的作用域。两个不同的类具有两个不同的类作用域。



即使两个类具有完全相同的成员列表，它们也是不同的类型。每个类的成员不同于任何其他类（或任何其他作用域）的成员。

444

例如：

```
class First {
public:
    int memi;
    double memd;
};

class Second {
public:
    int memi;
    double memd;
};

First obj1;
Second obj2 = obj1; // error: obj1 and obj2 have different types
```

1. 使用类的成员

在类作用域之外，成员只能通过对象或指针分别使用成员访问操作符 `.` 或 `->` 来访问。这些操作符左边的操作数分别是一个类对象或指向类对象的指针。跟在操作符后面的成员名字必须在相关联的类的作用域中声明：

```
Class obj;           // Class is some class type
Class *ptr = &obj;
// member is a data member of that class
ptr->member;          // fetches member from the object to which ptr points
obj.member;           // fetches member from the object named obj
// memfcn is a function member of that class
ptr->memfcn();         // runs memfcn on the object to which ptr points
obj.memfcn();          // runs memfcn on the object named obj
```

一些成员使用成员访问操作符来访问，另一些直接通过类使用作用域操作符 `::` 来访问。一般的数据或函数成员必须通过对象来访问。定义类型的成员，如 `Screen::index`，使用作用域操作符来访问。

2. 作用域与成员定义

尽管成员是在类的定义体之外定义的，但成员定义就好像它们是在类的作用域中一样。回忆一下，出现在类的定义体之外的成员定义必须指明成员出现在哪个类中：

```
double Sales_item::avg_price() const
{
    if (units_sold)
        return revenue/units_sold;
    else
```

```

        return 0;
    }

```

445

在这里，我们用完全限定名 `Sales_item::avg_price` 来指出这是类 `Sales_item` 作用域中的 `avg_price` 成员的定义。一旦看到成员的完全限定名，就知道该定义是在类作用域中。因为该定义是在类作用域中，所以我们可以引用 `revenue` 或 `units_sold`，而不必写 `this->revenue` 或 `this->units_sold`。

3. 形参表和函数体处于类作用域中

在定义于类外部的成员函数中，形参表和成员函数体都出现在成员名之后。这些都是在类作用域中定义，所以可以不用限定而引用其他成员。例如，类 `Screen` 中 `get` 的二形参版本的定义：

```

char Screen::get(index r, index c) const
{
    index row = r * width;          // compute the row location
    return contents[row + c];       // offset by c to fetch specified character
}

```

该函数用 `Screen` 内定义的 `index` 类型来指定其形参类型。因为形参表是在 `Screen` 类的作用域内，所以不必指明我们想要的是 `Screen::index`。我们想要的是定义在当前类作用域中的，这是隐含的。同样，使用 `index`、`width` 和 `contents` 时指的都是 `Screen` 类中声明的名字。

4. 函数返回类型不一定在类作用域中

与形参类型相比，返回类型出现在成员名字前面。如果函数在类定义体之外定义，则用于返回类型的名字在类作用域之外。如果返回类型使用由类定义的类型，则必须使用完全限定名。例如，考虑 `get_cursor` 函数：

```

class Screen {
public:
    typedef std::string::size_type index;
    index get_cursor() const;
};
inline Screen::index Screen::get_cursor() const
{
    return cursor;
}

```

该函数的返回类型是 `index`，这是在 `Screen` 类内部定义的一个类型名。如果在类定义体之外定义 `get_cursor`，则在函数名被处理之前，代码不在类作用域内。当看到返回类型时，其名字是在类作用域之外使用。必须用完全限定的类型名 `Screen::index` 来指定所需要的 `index` 是在类 `Screen` 中定义的名字。

446

习题

习题 12.15 列出在类作用域中的程序文本部分。

习题 12.16 如果如下定义 `get_cursor`，将会发生什么：

```

index Screen::get_cursor() const
{
    return cursor;
}

```

类作用域中的名字查找

迄今为止，在我们所编写的程序中，**名字查找**（寻找与给定的名字使用相匹配的声明的过程）是相对直接的。

- (1) 首先，在使用该名字的块中查找名字的声明。只考虑在该项使用之前声明的名字。
- (2) 如果找不到该名字，则在包围的作用域中查找。

如果找不到任何声明，则程序出错。在 C++ 程序中，所有名字必须在使用之前声明。

类作用域也许表现得有点不同，但实际上遵循同一规则。可能引起混淆的是函数中名字确定的方式，而该函数是在类定义体内定义的。



类定义实际上是在两个阶段中处理：

- (1) 首先，编译成员声明；
- (2) 只有在所有成员出现之后，才编译它们的定义本身。

当然，类作用域中使用的名字并非必须是类成员名。类作用域中的名字查找也会发现在其他作用域中声明的名字。在名字查找期间，如果类作用域中使用的名字不能确定为类成员名，则在包含该类或成员定义的作用域中查找，以便找到该名字的声明。

1. 类成员声明的名字查找

按以下方式确定在类成员的声明中用到的名字。

- 检查出现在名字使用之前的类成员的声明。
- 如果第 1 步查找不成功，则检查包含类定义的作用域中出现的声明以及出现在类定义之前的声明。

例如：

```
typedef double Money;
class Account {
public:
    Money balance() { return bal; }
private:
    Money bal;
    // ...
};
```

在处理 `balance` 函数的声明时，编译器首先在类 `Account` 的作用域中查找 `Money` 的声明。编译器只考虑出现在 `Money` 使用之前的声明。因为找不到任何成员声明，编译器随后在全局作用域中查找 `Money` 的声明。只考虑出现在类 `Account` 的定义之前的声明。找到全局的类型别名 `Money` 的声明，并将它用作函数 `balance` 的返回类型和数据成员 `bal` 的类型。



必须在类中先定义类型名字，才能将它们用作数据成员的类型，或者成员函数的返回类型或形参类型。

编译器按照成员声明在类中出现的次序来处理它们。通常，名字必须在使用之前进行定义。而且，一旦一个名字被用作类型名，该名字就不能被重复定义：

```

typedef double Money;
class Account {
public:
    Money balance() { return bal; } // uses global definition of Money
private:
    // error: cannot change meaning of Money
    typedef long double Money;
    Money bal;
    // ...
};

```

2. 类成员定义中的名字查找

按以下方式确定在成员函数的函数体中用到的名字。

- 首先检查成员函数局部作用域中的声明。
- 如果在成员函数中找不到该名字的声明，则检查对所有类成员的声明。
- 如果在类中找不到该名字的声明，则检查在此成员函数定义之前的作用域中出现的声明。

448

3. 类成员遵循常规的块作用域名字查找



例示名字查找的程序经常不得不依赖一些坏习惯。下面的几个程序故意包含了坏的风格。

下面的函数使用了相同的名字来表示形参和成员，这是通常应该避免的。这样做的目的是展示如何确定名字：

```

// Note: This code is for illustration purposes only and reflects bad practice
// It is a bad idea to use the same name for a parameter and a member
int height;
class Screen {
public:
    void dummy_fcn(index height) {
        cursor = width * height; // which height? The parameter
    }
private:
    index cursor;
    index height, width;
};

```

查找 `dummy_fcn` 的定义中使用的名字 `height` 的声明时，编译器首先在该函数的局部作用域中查找。函数的局部作用域中声明了一个函数形参。`dummy_fcn` 的函数体中使用的名字 `height` 指的就是这个形参声明。

在本例中，`height` 形参屏蔽名为 `height` 的成员。



尽管类的成员被屏蔽了，但仍然可以通过用类名来限定成员名或显式使用 `this` 指针来使用它。

如果我们想覆盖常规的查找规则，应该这样做：

```
// bad practice: Names local to member functions shouldn't hide member names
void dummy_fcn(index height) {
    cursor = width * this->height;    // member height
    // alternative way to indicate the member
    cursor = width * Screen::height;  // member height
}
```

4. 函数作用域之后，在类作用域中查找

如果想要使用 height 成员，更好的方式也许是为形参取一个不同的名字：

```
// good practice: Don't use member name for a parameter or other local variable
void dummy_fcn(index ht) {
    cursor = width * height; // member height
}
```

449

现在当编译器查找名字 height 时，它将不会在函数内查找该名字。编译器接着会在 Screen 类中查找。因为 height 是在成员函数内部使用，所以编译器在所有成员声明中查找。尽管 height 是先在 dummy_fcn 中使用，然后再声明，编译器还是确定这里用的是名为 height 的数据成员。

5. 类作用域之后，在外围作用域中查找

如果编译器不能在函数或类作用域中找到，就在外围作用域中查找。在本例子中，出现在 Screen 定义之前的全局作用域中声明了一个名为 height 的全局对象。然而，该对象被屏蔽了。



尽管全局对象被屏蔽了，但通过用全局作用域确定操作符来限定名字，仍然可以使用它。

```
// bad practice: Don't hide names that are needed from surrounding scopes
void dummy_fcn(index height) {
    cursor = width * ::height; // which height? The global one
}
```

6. 在文件中名字的出现处确定名字

当成员定义在类定义的外部时，名字查找的第 3 步不仅要考虑在 Screen 类定义之前的全局作用域中的声明，而且要考虑在成员函数定义之前出现的全局作用域声明。例如：

```
class Screen {
public:
    // ...
    void setHeight(index);
private:
    index height;
};

Screen::index verify(Screen::index);

void Screen::setHeight(index var) {
    // var: refers to the parameter
    // height: refers to the class member
    // verify: refers to the global function
    height = verify(var);
}
```