第Ⅱ部分
C++标准库

内容

第 8 章　IO 库 277
第 9 章　顺序容器 291
第 10 章　泛型算法 335
第 11 章　关联容器 373
第 12 章　动态内存 399

随着 C++版本的一次次修订，标准库也在不断成长。确实，新的 C++标准中有三分之二的文本都用来描述标准库。虽然我们不能深入讨论所有标准库设施，但有些核心库设施是每个 C++程序员都应该熟练掌握的，第二部分将介绍这些内容。

我们首先在第 8 章中介绍基本的 IO 库设施。除了使用标准库读写与控制台窗口相关联的流之外，我们还将学习其他一些库类型，可以帮助我们读写命名文件以及完成到 `string` 对象的内存 IO 操作。

标准库的核心是很多容器类和一族泛型算法，这些设施能帮助我们编写简洁高效的程序。标准库会去关注那些簿记操作的细节，特别是内存管理，这样我们的程序就可以将全部注意力投入到需要求解的问题上。

我们在第 3 章中已经介绍了容器类型 `vector`，在第 9 章中将介绍更多 `vector` 相关的内容，这一章也会涉及其他顺序容器。我们还会介绍更多 `string` 类型所支持的操作，可以将 `string` 看作一种只包含字符元素的特殊容器。`string` 支持很多容器操作，但并不是全部。

第 10 章介绍泛型算法。这类算法通常在顺序容器一定范围内的元素上或其他类型的序列上进行操作。算法库为各种经典算法提供了高效的实现，如排序和搜索算法，还提供了其他一些常用操作。例如，标准库提供了 `copy` 算法，完成一个序列到另一个序列的元素拷贝；还提供了 `find` 算法，实现给定元素的查找，等等。泛型算法的通用性体现在两个层面：可应用于不同类型的序列；对序列中元素的类型限制小，大多数类型都是允许的。

标准库还提供了一些关联容器，第 11 章介绍这部分内容。关联容器中的元素是通过关键字来访问的。关联容器支持很多顺序容器的操作，也定义了一些自己特有的操作。

第 12 章是第二部分的最后一章，这一章介绍动态内存管理相关的一些语言特性和库设施。这一章介绍智能指针的一个标准版本，它是新标准库中最重要的类之一。通过使用智能指针，我们可以大幅度提高使用动态内存的代码的鲁棒性。这一章最后将给出一个较大的例子，使用了第II部分介绍的所有标准库设施。

第 8 章

IO 库

内容

8.1 IO 类 278
8.2 文件输入输出 283
8.3 string 流 287
小结 290
术语表 290

C++语言不直接处理输入输出，而是通过一族定义在标准库中的类型来处理 IO。这些类型支持从设备读取数据、向设备写入数据的 IO 操作，设备可以是文件、控制台窗口等。还有一些类型允许内存 IO，即，从 `string` 读取数据，向 `string` 写入数据。

IO 库定义了读写内置类型值的操作。此外，一些类，如 `string`，通常也会定义类似的 IO 操作，来读写自己的对象。

本章介绍 IO 库的基本内容。后续章节会介绍更多 IO 库的功能：第 14 章将会介绍如何编写自己的输入输出运算符，第 17 章将会介绍如何控制输出格式以及如何对文件进行随机访问。

310 我们的程序已经使用了很多 IO 库设施了。我们在 1.2 节（第 5 页）已经介绍了大部分 IO 库设施：

- istream（输入流）类型，提供输入操作。
- ostream（输出流）类型，提供输出操作。
- cin，一个 istream 对象，从标准输入读取数据。
- cout，一个 ostream 对象，向标准输出写入数据。
- cerr，一个 ostream 对象，通常用于输出程序错误消息，写入到标准错误。
- >>运算符，用来从一个 istream 对象读取输入数据。
- <<运算符，用来向一个 ostream 对象写入输出数据。
- getline 函数（参见 3.3.2 节，第 78 页），从一个给定的 istream 读取一行数据，存入一个给定的 string 对象中。



8.1 IO 类

到目前为止，我们已经使用过的 IO 类型和对象都是操纵 char 数据的。默认情况下，这些对象都是关联到用户的控制台窗口的。当然，我们不能限制实际应用程序仅从控制台窗口进行 IO 操作，应用程序常常需要读写命名文件。而且，使用 IO 操作处理 string 中的字符会很方便。此外，应用程序还可能读写需要宽字符支持的语言。

为了支持这些不同种类的 IO 处理操作，在 istream 和 ostream 之外，标准库还定义了其他一些 IO 类型，我们之前都已经使用过了。表 8.1 列出了这些类型，分别定义在三个独立的头文件中：iostream 定义了用于读写流的基本类型，fstream 定义了读写命名文件的类型，sstream 定义了读写内存 string 对象的类型。

表 8.1：IO 库类型和头文件

头文件	类型
iostream	istream，wistream 从流读取数据 ostream，wostream 向流写入数据 iostream，wiostream 读写流
fstream	ifstream，wifstream 从文件读取数据 ofstream，wofstream 向文件写入数据 fstream，wfstream 读写文件
sstream	istringstream，wistringstream 从 string 读取数据 ostringstream，wostringstream 向 string 写入数据 stringstream，wstringstream 读写 string

311 为了支持使用宽字符的语言，标准库定义了一组类型和对象来操纵 wchar_t 类型的数据（参见 2.1.1 节，第 30 页）。宽字符版本的类型和函数的名字以一个 w 开始。例如，wcin、wcout 和 wcerr 是分别对应 cin、cout 和 cerr 的宽字符版对象。宽字符版本的类型和对象与其对应的普通 char 版本的类型定义在同一个头文件中。例如，头文件 fstream 定义了 ifstream 和 wifstream 类型。

IO 类型间的关系

概念上，设备类型和字符大小都不会影响我们要执行的 IO 操作。例如，我们可以用>>读取数据，而不用管是从一个控制台窗口，一个磁盘文件，还是一个 string 读取。类似的，我们也不用管读取的字符能存入一个 char 对象内，还是需要一个 wchar_t 对象来存储。

标准库使我们能忽略这些不同类型的流之间的差异，这是通过**继承机制**（inheritance）实现的。利用模板（参见 3.3 节，第 87 页），我们可以使用具有继承关系的类，而不必了解继承机制如何工作的细节。我们将在第 15 章和 18.3 节（第 710 页）介绍 C++是如何支持继承机制的。

简单地说，继承机制使我们可以声明一个特定的类继承自另一个类。我们通常可以将一个派生类（继承类）对象当作其基类（所继承的类）对象来使用。

类型 ifstream 和 istringstream 都继承自 istream。因此，我们可以像使用 istream 对象一样来使用 ifstream 和 istringstream 对象。也就是说，我们是如何使用 cin 的，就可以同样地使用这些类型的对象。例如，可以对一个 ifstream 或 istringstream 对象调用 getline，也可以使用>>从一个 ifstream 或 istringstream 对象中读取数据。类似的，类型 ofstream 和 ostringstream 都继承自 ostream。因此，我们是如何使用 cout 的，就可以同样地使用这些类型的对象。



本节剩下部分所介绍的标准库流特性都可以无差别地应用于普通流、文件流和 string 流，以及 char 或宽字符流版本。

8.1.1 IO 对象无拷贝或赋值



如我们在 7.1.3 节（第 234 页）所见，我们不能拷贝或对 IO 对象赋值：

```
ofstream out1, out2;
out1 = out2;                          // 错误：不能对流对象赋值
ofstream print(ofstream);             // 错误：不能初始化 ofstream 参数
out2 = print(out2);                   // 错误：不能拷贝流对象
```

由于不能拷贝 IO 对象，因此我们也不能将形参或返回类型设置为流类型（参见 6.2.1 节，第 188 页）。进行 IO 操作的函数通常以引用方式传递和返回流。读写一个 IO 对象会改变其状态，因此传递和返回的引用不能是 const 的。

8.1.2 条件状态

312

IO 操作一个与生俱来的问题是可能发生错误。一些错误是可恢复的，而其他错误则发生在系统深处，已经超出了应用程序可以修正的范围。表 8.2 列出了 IO 类所定义的一些函数和标志，可以帮助我们访问和操纵流的**条件状态**（condition state）。

表 8.2：IO 库条件状态

strm::iostate	*strm* 是一种 IO 类型，在表 8.1（第 278 页）中已列出。iostate 是一种机器相关的类型，提供了表达条件状态的完整功能
strm::badbit	*strm*::badbit 用来指出流已崩溃
strm::failbit	*strm*::failbit 用来指出一个 IO 操作失败了

续表

strm::eofbit	*strm*::eofbit 用来指出流到达了文件结束
strm::goodbit	*strm*::goodbit 用来指出流未处于错误状态。此值保证为零
s.eof()	若流 s 的 eofbit 置位，则返回 true
s.fail()	若流 s 的 failbit 或 badbit 置位，则返回 true
s.bad()	若流 s 的 badbit 置位，则返回 true
s.good()	若流 s 处于有效状态，则返回 true
s.clear()	将流 s 中所有条件状态位复位，将流的状态设置为有效。返回 void
s.clear(flags)	根据给定的 flags 标志位，将流 s 中对应条件状态位复位。flags 的类型为 *strm*::iostate。返回 void
s.setstate(flags)	根据给定的 flags 标志位，将流 s 中对应条件状态位置位。flags 的类型为 *strm*::iostate。返回 void
s.rdstate()	返回流 s 的当前条件状态，返回值类型为 strm::iostate

下面是一个 IO 错误的例子：

```
int ival;
cin >> ival;
```

如果我们在标准输入上键入 Boo，读操作就会失败。代码中的输入运算符期待读取一个 int，但却得到了一个字符 B。这样，cin 会进入错误状态。类似的，如果我们输入一个文件结束标识，cin 也会进入错误状态。

一个流一旦发生错误，其上后续的 IO 操作都会失败。只有当一个流处于无错状态时，我们才可以从它读取数据，向它写入数据。由于流可能处于错误状态，因此代码通常应该在使用一个流之前检查它是否处于良好状态。确定一个流对象的状态的最简单的方法是将它当作一个条件来使用：

```
while (cin >> word)
    // ok: 读操作成功……
```

while 循环检查>>表达式返回的流的状态。如果输入操作成功，流保持有效状态，则条件为真。

查询流的状态

将流作为条件使用，只能告诉我们流是否有效，而无法告诉我们具体发生了什么。有时我们也需要知道流为什么失败。例如，在键入文件结束标识后我们的应对措施，可能与遇到一个 IO 设备错误的处理方式是不同的。

IO 库定义了一个与机器无关的 iostate 类型，它提供了表达流状态的完整功能。这个类型应作为一个位集合来使用，使用方式与我们在 4.8 节中（第 137 页）使用 quiz1 的方式一样。IO 库定义了 4 个 iostate 类型的 constexpr 值（参见 2.4.4 节，第 58 页），表示特定的位模式。这些值用来表示特定类型的 IO 条件，可以与位运算符（参见 4.8 节，第 137 页）一起使用来一次性检测或设置多个标志位。

badbit 表示系统级错误，如不可恢复的读写错误。通常情况下，一旦 badbit 被置位，流就无法再使用了。在发生可恢复错误后，failbit 被置位，如期望读取数值却读出一个字符等错误。这种问题通常是可以修正的，流还可以继续使用。如果到达文件结束位置，eofbit 和 failbit 都会被置位。goodbit 的值为 0，表示流未发生错误。如果 badbit、failbit 和 eofbit 任一个被置位，则检测流状态的条件会失败。

标准库还定义了一组函数来查询这些标志位的状态。操作 good 在所有错误位均未置位的情况下返回 true，而 bad、fail 和 eof 则在对应错误位被置位时返回 true。此外，在 badbit 被置位时，fail 也会返回 true。这意味着，使用 good 或 fail 是确定流的总体状态的正确方法。实际上，我们将流当作条件使用的代码就等价于!fail()。而 eof 和 bad 操作只能表示特定的错误。 313

管理条件状态

流对象的 rdstate 成员返回一个 iostate 值，对应流的当前状态。setstate 操作将给定条件位置位，表示发生了对应错误。clear 成员是一个重载的成员（参见 6.4 节，第 206 页）：它有一个不接受参数的版本，而另一个版本接受一个 iostate 类型的参数。

clear 不接受参数的版本清除（复位）所有错误标志位。执行 clear() 后，调用 good 会返回 true。我们可以这样使用这些成员：

```
// 记住 cin 的当前状态
auto old_state = cin.rdstate();  // 记住 cin 的当前状态
cin.clear();                     // 使 cin 有效
process_input(cin);              // 使用 cin
cin.setstate(old_state);         // 将 cin 置为原有状态
```

带参数的 clear 版本接受一个 iostate 值，表示流的新状态。为了复位单一的条件状态位，我们首先用 rdstate 读出当前条件状态，然后用位操作将所需位复位来生成新的状态。例如，下面的代码将 failbit 和 badbit 复位，但保持 eofbit 不变： 314

```
// 复位 failbit 和 badbit，保持其他标志位不变
cin.clear(cin.rdstate() & ~cin.failbit & ~cin.badbit);
```

8.1.2 节练习

练习 8.1：编写函数，接受一个 istream&参数，返回值类型也是 istream&。此函数须从给定流中读取数据，直至遇到文件结束标识时停止。它将读取的数据打印在标准输出上。完成这些操作后，在返回流之前，对流进行复位，使其处于有效状态。

练习 8.2：测试函数，调用参数为 cin。

练习 8.3：什么情况下，下面的 while 循环会终止？

```
while (cin >> i) /* ... */
```

8.1.3 管理输出缓冲

每个输出流都管理一个缓冲区，用来保存程序读写的数据。例如，如果执行下面的代码

```
os << "please enter a value: ";
```

文本串可能立即打印出来，但也有可能被操作系统保存在缓冲区中，随后再打印。有了缓冲机制，操作系统就可以将程序的多个输出操作组合成单一的系统级写操作。由于设备的写操作可能很耗时，允许操作系统将多个输出操作组合为单一的设备写操作可以带来很大的性能提升。

导致缓冲刷新（即，数据真正写到输出设备或文件）的原因有很多：

- 程序正常结束，作为 main 函数的 return 操作的一部分，缓冲刷新被执行。

- 缓冲区满时，需要刷新缓冲，而后新的数据才能继续写入缓冲区。
- 我们可以使用操纵符如 endl（参见 1.2 节，第 6 页）来显式刷新缓冲区。
- 在每个输出操作之后，我们可以用操纵符 unitbuf 设置流的内部状态，来清空缓冲区。默认情况下，对 cerr 是设置 unitbuf 的，因此写到 cerr 的内容都是立即刷新的。
- 一个输出流可能被关联到另一个流。在这种情况下，当读写被关联的流时，关联到的流的缓冲区会被刷新。例如，默认情况下，cin 和 cerr 都关联到 cout。因此，读 cin 或写 cerr 都会导致 cout 的缓冲区被刷新。

315 刷新输出缓冲区

我们已经使用过操纵符 endl，它完成换行并刷新缓冲区的工作。IO 库中还有两个类似的操纵符：flush 和 ends。flush 刷新缓冲区，但不输出任何额外的字符；ends 向缓冲区插入一个空字符，然后刷新缓冲区：

```
cout << "hi!" << endl;   // 输出 hi 和一个换行，然后刷新缓冲区
cout << "hi!" << flush;  // 输出 hi，然后刷新缓冲区，不附加任何额外字符
cout << "hi!" << ends;   // 输出 hi 和一个空字符，然后刷新缓冲区
```

unitbuf 操纵符

如果想在每次输出操作后都刷新缓冲区，我们可以使用 unitbuf 操纵符。它告诉流在接下来的每次写操作之后都进行一次 flush 操作。而 nounitbuf 操纵符则重置流，使其恢复使用正常的系统管理的缓冲区刷新机制：

```
cout << unitbuf;          // 所有输出操作后都会立即刷新缓冲区
// 任何输出都立即刷新，无缓冲
cout << nounitbuf;        // 回到正常的缓冲方式
```

警告：如果程序崩溃，输出缓冲区不会被刷新

如果程序异常终止，输出缓冲区是不会被刷新的。当一个程序崩溃后，它所输出的数据很可能停留在输出缓冲区中等待打印。

当调试一个已经崩溃的程序时，需要确认那些你认为已经输出的数据确实已经刷新了。否则，可能将大量时间浪费在追踪代码为什么没有执行上，而实际上代码已经执行了，只是程序崩溃后缓冲区没有被刷新，输出数据被挂起没有打印而已。

关联输入和输出流

当一个输入流被关联到一个输出流时，任何试图从输入流读取数据的操作都会先刷新关联的输出流。标准库将 cout 和 cin 关联在一起，因此下面语句

```
cin >> ival;
```

导致 cout 的缓冲区被刷新。



交互式系统通常应该关联输入流和输出流。这意味着所有输出，包括用户提示信息，都会在读操作之前被打印出来。

tie 有两个重载的版本（参见 6.4 节，第 206 页）：一个版本不带参数，返回指向输

出流的指针。如果本对象当前关联到一个输出流，则返回的就是指向这个流的指针，如果对象未关联到流，则返回空指针。tie 的第二个版本接受一个指向 ostream 的指针，将自己关联到此 ostream。即，x.tie(&o)将流 x 关联到输出流 o。 316

我们既可以将一个 istream 对象关联到另一个 ostream，也可以将一个 ostream 关联到另一个 ostream：

```
cin.tie(&cout);          // 仅仅是用来展示：标准库将 cin 和 cout 关联在一起
// old_tie 指向当前关联到 cin 的流（如果有的话）
ostream *old_tie = cin.tie(nullptr); // cin 不再与其他流关联
// 将 cin 与 cerr 关联；这不是一个好主意，因为 cin 应该关联到 cout
cin.tie(&cerr);          // 读取 cin 会刷新 cerr 而不是 cout
cin.tie(old_tie);        // 重建 cin 和 cout 间的正常关联
```

在这段代码中，为了将一个给定的流关联到一个新的输出流，我们将新流的指针传递给了 tie。为了彻底解开流的关联，我们传递了一个空指针。每个流同时最多关联到一个流，但多个流可以同时关联到同一个 ostream。

8.2 文件输入输出



头文件 fstream 定义了三个类型来支持文件 IO：**ifstream** 从一个给定文件读取数据，**ofstream** 向一个给定文件写入数据，以及 **fstream** 可以读写给定文件。在 17.5.3 节中（第 676 页）我们将介绍如何对同一个文件流既读又写。

这些类型提供的操作与我们之前已经使用过的对象 cin 和 cout 的操作一样。特别是，我们可以用 IO 运算符（<<和>>）来读写文件，可以用 getline（参见 3.2.2 节，第 79 页）从一个 ifstream 读取数据，包括 8.1 节中（第 278 页）介绍的内容也都适用于这些类型。

除了继承自 iostream 类型的行为之外，fstream 中定义的类型还增加了一些新的成员来管理与流关联的文件。在表 8.3 中列出了这些操作，我们可以对 fstream、ifstream 和 ofstream 对象调用这些操作，但不能对其他 IO 类型调用这些操作。

表 8.3：fstream 特有的操作

fstream fstrm;	创建一个未绑定的文件流。*fstream* 是头文件 fstream 中定义的一个类型
fstream fstrm(s);	创建一个 *fstream*，并打开名为 s 的文件。s 可以是 string 类型，或者是一个指向 C 风格字符串的指针（参见 3.5.4 节，第 109 页）。这些构造函数都是 explicit 的（参见 7.5.4 节，第 265 页）。默认的文件模式 mode 依赖于 *fstream* 的类型
fstream fstrm(s, mode);	与前一个构造函数类似，但按指定 mode 打开文件
fstrm.open(s)	打开名为 s 的文件，并将文件与 fstrm 绑定。s 可以是一个 string 或一个指向 C 风格字符串的指针。默认的文件 mode 依赖于 *fstream* 的类型。返回 void
fstrm.close()	关闭与 fstrm 绑定的文件。返回 void
fstrm.is_open()	返回一个 bool 值，指出与 fstrm 关联的文件是否成功打开且尚未关闭

317 8.2.1 使用文件流对象



当我们想要读写一个文件时，可以定义一个文件流对象，并将对象与文件关联起来。每个文件流类都定义了一个名为 open 的成员函数，它完成一些系统相关的操作，来定位给定的文件，并视情况打开为读或写模式。

创建文件流对象时，我们可以提供文件名（可选的）。如果提供了一个文件名，则 open 会自动被调用：

```
ifstream in(ifile);      // 构造一个 ifstream 并打开给定文件
ofstream out;            // 输出文件流未关联到任何文件
```

这段代码定义了一个输入流 in，它被初始化为从文件读取数据，文件名由 string 类型的参数 ifile 指定。第二条语句定义了一个输出流 out，未与任何文件关联。在新 C++ 标准中，文件名既可以是库类型 string 对象，也可以是 C 风格字符数组（参见 3.5.4 节，第 109 页）。旧版本的标准库只允许 C 风格字符数组。

C++ 11

用 fstream 代替 iostream&

我们在 8.1 节（第 279 页）已经提到过，在要求使用基类型对象的地方，我们可以用继承类型的对象来替代。这意味着，接受一个 iostream 类型引用（或指针）参数的函数，可以用一个对应的 fstream（或 sstream）类型来调用。也就是说，如果有一个函数接受一个 ostream&参数，我们在调用这个函数时，可以传递给它一个 ofstream 对象，对 istream&和 ifstream 也是类似的。

例如，我们可以用 7.1.3 节中的 read 和 print 函数来读写命名文件。在本例中，我们假定输入和输出文件的名字是通过传递给 main 函数的参数来指定的（参见 6.2.5 节，第 196 页）：

```
ifstream input(argv[1]);      // 打开销售记录文件
ofstream output(argv[2]);     // 打开输出文件
Sales_data total;             // 保存销售总额的变量
if (read(input, total)) {     // 读取第一条销售记录
    Sales_data trans;         // 保存下一条销售记录的变量
    while(read(input, trans)) {                // 读取剩余记录
        if (total.isbn() == trans.isbn())      // 检查 isbn
            total.combine(trans);              // 更新销售总额
        else {
            print(output, total) << endl;      // 打印结果
            total = trans;                     // 处理下一本书
        }
    }
    print(output, total) << endl;     // 打印最后一本书的销售额
} else                                // 文件中无输入数据
    cerr << "No data?!" << endl;
```

除了读写的是命名文件外，这段程序与 229 页的加法程序几乎是完全相同的。重要的部分是对 read 和 print 的调用。虽然两个函数定义时指定的形参分别是 istream&和 ostream&，但我们可以向它们传递 fstream 对象。

318 成员函数 open 和 close

如果我们定义了一个空文件流对象，可以随后调用 open 来将它与文件关联起来：

```
ifstream in(ifile);              // 构筑一个 ifstream 并打开给定文件
ofstream out;                    // 输出文件流未与任何文件相关联
out.open(ifile + ".copy");       // 打开指定文件
```

如果调用 open 失败，failbit 会被置位（参见 8.1.2 节，第 280 页）。因为调用 open 可能失败，进行 open 是否成功的检测通常是一个好习惯：

```
if (out)     // 检查 open 是否成功
             // open 成功，我们可以使用文件了
```

这个条件判断与我们之前将 cin 用作条件相似。如果 open 失败，条件会为假，我们就不会去使用 out 了。

一旦一个文件流已经打开，它就保持与对应文件的关联。实际上，对一个已经打开的文件流调用 open 会失败，并会导致 failbit 被置位。随后的试图使用文件流的操作都会失败。为了将文件流关联到另外一个文件，必须首先关闭已经关联的文件。一旦文件成功关闭，我们可以打开新的文件：

```
in.close();                  // 关闭文件
in.open(ifile + "2");        // 打开另一个文件
```

如果 open 成功，则 open 会设置流的状态，使得 good() 为 true。

自动构造和析构

考虑这样一个程序，它的 main 函数接受一个要处理的文件列表（参见 6.2.5 节，第 196 页）。这种程序可能会有如下的循环：

```
// 对每个传递给程序的文件执行循环操作
for (auto p = argv + 1; p != argv + argc; ++p) {
    ifstream input(*p);  // 创建输出流并打开文件
    if (input) {         // 如果文件打开成功，“处理” 此文件
        process(input);
    } else
        cerr << "couldn't open: " + string(*p);
} // 每个循环步 input 都会离开作用域，因此会被销毁
```

每个循环步构造一个新的名为 input 的 ifstream 对象，并打开它来读取给定的文件。像之前一样，我们检查 open 是否成功。如果成功，将文件传递给一个函数，该函数负责读取并处理输入数据。如果 open 失败，打印一条错误信息并继续处理下一个文件。

因为 input 是 while 循环的局部变量，它在每个循环步中都要创建和销毁一次（参见 5.4.1 节，第 165 页）。当一个 fstream 对象离开其作用域时，与之关联的文件会自动关闭。在下一步循环中，input 会再次被创建。



当一个 fstream 对象被销毁时，close 会自动被调用。

8.2.1 节练习

319

练习 8.4：编写函数，以读模式打开一个文件，将其内容读入到一个 string 的 vector 中，将每一行作为一个独立的元素存于 vector 中。

练习 8.5：重写上面的程序，将每个单词作为一个独立的元素进行存储。

练习 8.6：重写 7.1.1 节的书店程序（第 229 页），从一个文件中读取交易记录。将文件名作为一个参数传递给 main（参见 6.2.5 节，第 196 页）。

8.2.2 文件模式

每个流都有一个关联的**文件模式**（file mode），用来指出如何使用文件。表 8.4 列出了文件模式和它们的含义。

表 8.4：文件模式	
in	以读方式打开
out	以写方式打开
app	每次写操作前均定位到文件末尾
ate	打开文件后立即定位到文件末尾
trunc	截断文件
binary	以二进制方式进行 IO

无论用哪种方式打开文件，我们都可以指定文件模式，调用 open 打开文件时可以，用一个文件名初始化流来隐式打开文件时也可以。指定文件模式有如下限制：

- 只可以对 ofstream 或 fstream 对象设定 out 模式。
- 只可以对 ifstream 或 fstream 对象设定 in 模式。
- 只有当 out 也被设定时才可设定 trunc 模式。
- 只要 trunc 没被设定，就可以设定 app 模式。在 app 模式下，即使没有显式指定 out 模式，文件也总是以输出方式被打开。
- 默认情况下，即使我们没有指定 trunc，以 out 模式打开的文件也会被截断。为了保留以 out 模式打开的文件的内容，我们必须同时指定 app 模式，这样只会将数据追加写到文件末尾；或者同时指定 in 模式，即打开文件同时进行读写操作（参见 17.5.3 节，第 676 页，将介绍对同一个文件既进行输入又进行输出的方法）。
- ate 和 binary 模式可用于任何类型的文件流对象，且可以与其他任何文件模式组合使用。

每个文件流类型都定义了一个默认的文件模式，当我们未指定文件模式时，就使用此默认模式。与 ifstream 关联的文件默认以 in 模式打开；与 ofstream 关联的文件默认以 out 模式打开；与 fstream 关联的文件默认以 in 和 out 模式打开。

320 以 out 模式打开文件会丢弃已有数据

默认情况下，当我们打开一个 ofstream 时，文件的内容会被丢弃。阻止一个 ofstream 清空给定文件内容的方法是同时指定 app 模式：

```
// 在这几条语句中，file1 都被截断
ofstream out("file1"); // 隐含以输出模式打开文件并截断文件
ofstream out2("file1", ofstream::out); // 隐含地截断文件
ofstream out3("file1", ofstream::out | ofstream::trunc);
// 为了保留文件内容，我们必须显式指定 app 模式
ofstream app("file2", ofstream::app); // 隐含为输出模式
ofstream app2("file2", ofstream::out | ofstream::app);
```



保留被 ofstream 打开的文件中已有数据的唯一方法是显式指定 app 或 in 模式。

每次调用 open 时都会确定文件模式

对于一个给定流，每当打开文件时，都可以改变其文件模式。

```
ofstream out; // 未指定文件打开模式
out.open("scratchpad"); // 模式隐含设置为输出和截断
out.close(); // 关闭 out，以便我们将其用于其他文件
out.open("precious", ofstream::app); // 模式为输出和追加
out.close();
```

第一个 open 调用未显式指定输出模式，文件隐式地以 out 模式打开。通常情况下，out 模式意味着同时使用 trunc 模式。因此，当前目录下名为 scratchpad 的文件的内容将被清空。当打开名为 precious 的文件时，我们指定了 append 模式。文件中已有的数据都得以保留，所有写操作都在文件末尾进行。



在每次打开文件时，都要设置文件模式，可能是显式地设置，也可能是隐式地设置。当程序未指定模式时，就使用默认值。

8.2.2 节练习

练习 8.7：修改上一节的书店程序，将结果保存到一个文件中。将输出文件名作为第二个参数传递给 main 函数。

练习 8.8：修改上一题的程序，将结果追加到给定的文件末尾。对同一个输出文件，运行程序至少两次，检验数据是否得以保留。

8.3 string 流

321

sstream 头文件定义了三个类型来支持内存 IO，这些类型可以向 string 写入数据，从 string 读取数据，就像 string 是一个 IO 流一样。

istringstream 从 string 读取数据，**ostringstream** 向 string 写入数据，而头文件 **stringstream** 既可从 string 读数据也可向 string 写数据。与 fstream 类型类似，头文件 sstream 中定义的类型都继承自我们已经使用过的 iostream 头文件中定义的类型。除了继承得来的操作，sstream 中定义的类型还增加了一些成员来管理与流相关联的 string。表 8.5 列出了这些操作，可以对 stringstream 对象调用这些操作，但不能对其他 IO 类型调用这些操作。

表 8.5：stringstream 特有的操作

sstream strm;	strm 是一个未绑定的 stringstream 对象。*sstream* 是头文件 sstream 中定义的一个类型
sstream strm(s);	strm 是一个 *sstream* 对象，保存 string s 的一个拷贝。此构造函数是 explicit 的（参见 7.5.4 节，第 265 页）
strm.str()	返回 strm 所保存的 string 的拷贝
strm.str(s)	将 string s 拷贝到 strm 中。返回 void

8.3.1 使用 istringstream

当我们的某些工作是对整行文本进行处理，而其他一些工作是处理行内的单个单词

时，通常可以使用 istringstream。

考虑这样一个例子，假定有一个文件，列出了一些人和他们的电话号码。某些人只有一个号码，而另一些人则有多个——家庭电话、工作电话、移动电话等。我们的输入文件看起来可能是这样的：

```
morgan 2015552368 8625550123
drew 9735550130
lee 6095550132 2015550175 8005550000
```

文件中每条记录都以一个人名开始，后面跟随一个或多个电话号码。我们首先定义一个简单的类来描述输入数据：

```
// 成员默认为公有；参见 7.2 节（第 240 页）
struct PersonInfo {
    string name;
    vector<string> phones;
};
```

类型 PersonInfo 的对象会有一个成员来表示人名，还有一个 vector 来保存此人的所有电话号码。

322 我们的程序会读取数据文件，并创建一个 PersonInfo 的 vector。vector 中每个元素对应文件中的一条记录。我们在一个循环中处理输入数据，每个循环步读取一条记录，提取出一个人名和若干电话号码：

```
string line, word;                  // 分别保存来自输入的一行和单词
vector<PersonInfo> people;          // 保存来自输入的所有记录
// 逐行从输入读取数据，直至 cin 遇到文件尾（或其他错误）
while (getline(cin, line)) {
    PersonInfo info;                // 创建一个保存此记录数据的对象
    istringstream record(line);     // 将记录绑定到刚读入的行
    record >> info.name;            // 读取名字
    while (record >> word)          // 读取电话号码
        info.phones.push_back(word); // 保持它们
    people.push_back(info);         // 将此记录追加到 people 末尾
}
```

这里我们用 getline 从标准输入读取整条记录。如果 getline 调用成功，那么 line 中将保存着从输入文件而来的一条记录。在 while 中，我们定义了一个局部 PersonInfo 对象，来保存当前记录中的数据。

接下来我们将一个 istringstream 与刚刚读取的文本行进行绑定，这样就可以在此 istringstream 上使用输入运算符来读取当前记录中的每个元素。我们首先读取人名，随后用一个 while 循环读取此人的电话号码。

当读取完 line 中所有数据后，内层 while 循环就结束了。此循环的工作方式与前面章节中读取 cin 的循环很相似，不同之处是，此循环从一个 string 而不是标准输入读取数据。当 string 中的数据全部读出后，同样会触发“文件结束”信号，在 record 上的下一个输入操作会失败。

我们将刚刚处理好的 PersonInfo 追加到 vector 中，外层 while 循环的一个循环步就随之结束了。外层 while 循环会持续执行，直至遇到 cin 的文件结束标识。

8.3.1 节练习

练习 8.9：使用你为 8.1.2 节（第 281 页）第一个练习所编写的函数打印一个 istringstream 对象的内容。

练习 8.10：编写程序，将来自一个文件中的行保存在一个 vector<string>中。然后使用一个 istringstream 从 vector 读取数据元素，每次读取一个单词。

练习 8.11：本节的程序在外层 while 循环中定义了 istringstream 对象。如果 record 对象定义在循环之外，你需要对程序进行怎样的修改？重写程序，将 record 的定义移到 while 循环之外，验证你设想的修改方法是否正确。

练习 8.12：我们为什么没有在 PersonInfo 中使用类内初始化？

8.3.2 使用 ostringstream

323

当我们逐步构造输出，希望最后一起打印时，ostringstream 是很有用的。例如，对上一节的例子，我们可能想逐个验证电话号码并改变其格式。如果所有号码都是有效的，我们希望输出一个新的文件，包含改变格式后的号码。对于那些无效的号码，我们不会将它们输出到新文件中，而是打印一条包含人名和无效号码的错误信息。

由于我们不希望输出有无效电话号码的人，因此对每个人，直到验证完所有电话号码后才可以进行输出操作。但是，我们可以先将输出内容“写入”到一个内存 ostringstream 中：

```
for (const auto &entry : people) { // 对people中每一项
    ostringstream formatted, badNums; // 每个循环步创建的对象
    for (const auto &nums : entry.phones) { // 对每个数
        if (!valid(nums)) {
            badNums << " " << nums; // 将数的字符串形式存入badNums
        } else
            // 将格式化的字符串“写入”formatted
            formatted << " " << format(nums);
    }
    if (badNums.str().empty())          // 没有错误的数
        os << entry.name << " "         // 打印名字
           << formatted.str() << endl;  // 和格式化的数
    else // 否则，打印名字和错误的数
        cerr << "input error: " << entry.name
             << " invalid number(s) " << badNums.str() << endl;
}
```

在此程序中，我们假定已有两个函数，valid 和 format，分别完成电话号码验证和改变格式的功能。程序最有趣的部分是对字符串流 formatted 和 badNums 的使用。我们使用标准的输出运算符(<<)向这些对象写入数据，但这些“写入”操作实际上转换为 string 操作，分别向 formatted 和 badNums 中的 string 对象添加字符。

8.3.2 节练习

练习 8.13：重写本节的电话号码程序，从一个命名文件而非 cin 读取数据。

练习 8.14：我们为什么将 entry 和 nums 定义为 const auto&？

324 小结

C++使用标准库类来处理面向流的输入和输出：

- iostream 处理控制台 IO
- fstream 处理命名文件 IO
- stringstream 完成内存 string 的 IO

类 fstream 和 stringstream 都是继承自类 iostream 的。输入类都继承自 istream，输出类都继承自 ostream。因此，可以在 istream 对象上执行的操作，也可在 ifstream 或 istringstream 对象上执行。继承自 ostream 的输出类也有类似情况。

每个 IO 对象都维护一组条件状态，用来指出此对象上是否可以进行 IO 操作。如果遇到了错误——例如在输入流上遇到了文件末尾，则对象的状态变为失效，所有后续输入操作都不能执行，直至错误被纠正。标准库提供了一组函数，用来设置和检测这些状态。

术语表

条件状态（condition state）可被任何流类使用的一组标志和函数，用来指出给定流是否可用。

文件模式（file mode）类 fstream 定义的一组标志，在打开文件时指定，用来控制文件如何被使用。

文件流（file stream）用来读写命名文件的流对象。除了普通的 iostream 操作，文件流还定义了 open 和 close 成员。成员函数 open 接受一个 string 或一个 C 风格字符串参数，指定要打开的文件名，它还可以接受一个可选的参数，指明文件打开模式。成员函数 close 关闭流所关联的文件，调用 close 后才可以调用 open 打开另一个文件。

fstream 用于同时读写一个相同文件的文件流。默认情况下，fstream 以 in 和 out 模式打开文件。

ifstream 用于从输入文件读取数据的文件流。默认情况下，ifstream 以 in 模式打开文件。

继承（inheritance）程序设计功能，令一个类型可以从另一个类型继承接口。类 ifstream 和 istringstream 继承自 istream，ofstream 和 ostringstream 继承自 ostream。第 15 章将介绍继承。

istringstream 用来从给定 string 读取数据的字符串流。

ofstream 用来向输出文件写入数据的文件流。默认情况下，ofstream 以 out 模式打开文件。

字符串流（string stream）用于读写 string 的流对象。除了普通的 iostream 操作外，字符串流还定义了一个名为 str 的重载成员。调用 str 的无参版本会返回字符串流关联的 string。调用时传递给它一个 string 参数，则会将字符串流与该 string 的一个拷贝相关联。

stringstream 用于读写给定 string 的字符串流。

第 9 章
顺序容器

内容

9.1 顺序容器概述 292
9.2 容器库概览 294
9.3 顺序容器操作 305
9.4 vector 对象是如何增长的 317
9.5 额外的 string 操作 320
9.6 容器适配器 329
小结 332
术语表 332

本章是第 3 章内容的扩展，完成本章的学习后，对标准库顺序容器知识的掌握就完整了。元素在顺序容器中的顺序与其加入容器时的位置相对应。标准库还定义了几种关联容器，关联容器中元素的位置由元素相关联的关键字值决定。我们将在第 11 章中介绍关联容器特有的操作。

所有容器类都共享公共的接口，不同容器按不同方式对其进行扩展。这个公共接口使容器的学习更加容易——我们基于某种容器所学习的内容也都适用于其他容器。每种容器都提供了不同的性能和功能的权衡。

326 一个容器就是一些特定类型对象的集合。**顺序容器**（sequential container）为程序员提供了控制元素存储和访问顺序的能力。这种顺序不依赖于元素的值，而是与元素加入容器时的位置相对应。与之相对的，我们将在第 11 章介绍的有序和无序关联容器，则根据关键字的值来存储元素。

标准库还提供了三种容器适配器，分别为容器操作定义了不同的接口，来与容器类型适配。我们将在本章末尾介绍适配器。



本章的内容基于 3.2 节、3.3 节和 3.4 节中已经介绍的有关容器的知识，我们假定读者已经熟悉了这几节的内容。

9.1 顺序容器概述

表 9.1 列出了标准库中的顺序容器，所有顺序容器都提供了快速顺序访问元素的能力。但是，这些容器在以下方面都有不同的性能折中：

- 向容器添加或从容器中删除元素的代价
- 非顺序访问容器中元素的代价

表 9.1：顺序容器类型

`vector`	可变大小数组。支持快速随机访问。在尾部之外的位置插入或删除元素可能很慢
`deque`	双端队列。支持快速随机访问。在头尾位置插入/删除速度很快
`list`	双向链表。只支持双向顺序访问。在 `list` 中任何位置进行插入/删除操作速度都很快
`forward_list`	单向链表。只支持单向顺序访问。在链表任何位置进行插入/删除操作速度都很快
`array`	固定大小数组。支持快速随机访问。不能添加或删除元素
`string`	与 `vector` 相似的容器，但专门用于保存字符。随机访问快。在尾部插入/删除速度快

除了固定大小的 `array` 外，其他容器都提供高效、灵活的内存管理。我们可以添加和删除元素，扩张和收缩容器的大小。容器保存元素的策略对容器操作的效率有着固有的，有时是重大的影响。在某些情况下，存储策略还会影响特定容器是否支持特定操作。

327 例如，`string` 和 `vector` 将元素保存在连续的内存空间中。由于元素是连续存储的，由元素的下标来计算其地址是非常快速的。但是，在这两种容器的中间位置添加或删除元素就会非常耗时：在一次插入或删除操作后，需要移动插入/删除位置之后的所有元素，来保持连续存储。而且，添加一个元素有时可能还需要分配额外的存储空间。在这种情况下，每个元素都必须移动到新的存储空间中。

`list` 和 `forward_list` 两个容器的设计目的是令容器任何位置的添加和删除操作都很快速。作为代价，这两个容器不支持元素的随机访问：为了访问一个元素，我们只能遍历整个容器。而且，与 `vector`、`deque` 和 `array` 相比，这两个容器的额外内存开销也很大。

`deque` 是一个更为复杂的数据结构。与 `string` 和 `vector` 类似，`deque` 支持快速

的随机访问。与 string 和 vector 一样，在 deque 的中间位置添加或删除元素的代价（可能）很高。但是，在 deque 的两端添加或删除元素都是很快的，与 list 或 forward_list 添加删除元素的速度相当。

forward_list 和 array 是新 C++标准增加的类型。与内置数组相比，array 是一种更安全、更容易使用的数组类型。与内置数组类似，array 对象的大小是固定的。因此，array 不支持添加和删除元素以及改变容器大小的操作。forward_list 的设计目标是达到与最好的手写的单向链表数据结构相当的性能。因此，forward_list 没有 size 操作，因为保存或计算其大小就会比手写链表多出额外的开销。对其他容器而言，size 保证是一个快速的常量时间的操作。

C++ 11



新标准库的容器比旧版本快得多，原因我们将在 13.6 节（第 470 页）解释。新标准库容器的性能几乎肯定与最精心优化过的同类数据结构一样好（通常会更好）。现代 C++程序应该使用标准库容器，而不是更原始的数据结构，如内置数组。

确定使用哪种顺序容器



通常，使用 vector 是最好的选择，除非你有很好的理由选择其他容器。

以下是一些选择容器的基本原则：

- 除非你有很好的理由选择其他容器，否则应使用 vector。
- 如果你的程序有很多小的元素，且空间的额外开销很重要，则不要使用 list 或 forward_list。
- 如果程序要求随机访问元素，应使用 vector 或 deque。
- 如果程序要求在容器的中间插入或删除元素，应使用 list 或 forward_list。
- 如果程序需要在头尾位置插入或删除元素，但不会在中间位置进行插入或删除操作，则使用 deque。
- 如果程序只有在读取输入时才需要在容器中间位置插入元素，随后需要随机访问元素，则 328
  - 首先，确定是否真的需要在容器中间位置添加元素。当处理输入数据时，通常可以很容易地向 vector 追加数据，然后再调用标准库的 sort 函数（我们将在 10.2.3 节介绍 sort（第 343 页））来重排容器中的元素，从而避免在中间位置添加元素。
  - 如果必须在中间位置插入元素，考虑在输入阶段使用 list，一旦输入完成，将 list 中的内容拷贝到一个 vector 中。

如果程序既需要随机访问元素，又需要在容器中间位置插入元素，那该怎么办？答案取决于在 list 或 forward_list 中访问元素与 vector 或 deque 中插入/删除元素的相对性能。一般来说，应用中占主导地位的操作（执行的访问操作更多还是插入/删除更多）决定了容器类型的选择。在此情况下，对两种容器分别测试应用的性能可能就是必要的了。



如果你不确定应该使用哪种容器，那么可以在程序中只使用 vector 和 list 公共的操作：使用迭代器，不使用下标操作，避免随机访问。这样，在必要时选择使用 vector 或 list 都很方便。

9.1 节练习

练习 9.1：对于下面的程序任务，vector、deque 和 list 哪种容器最为适合？解释你的选择的理由。如果没有哪一种容器优于其他容器，也请解释理由。

(a) 读取固定数量的单词，将它们按字典序插入到容器中。我们将在下一章中看到，关联容器更适合这个问题。

(b) 读取未知数量的单词，总是将新单词插入到末尾。删除操作在头部进行。

(c) 从一个文件读取未知数量的整数。将这些数排序，然后将它们打印到标准输出。



9.2 容器库概览

容器类型上的操作形成了一种层次：

- 某些操作是所有容器类型都提供的（参见表 9.2，第 295 页）。
- 另外一些操作仅针对顺序容器（参见表 9.3，第 299 页）、关联容器（参见表 11.7，第 388 页）或无序容器（参见表 11.8，第 395 页）。
- 还有一些操作只适用于一小部分容器。

329 在本节中，我们将介绍对所有容器都适用的操作。本章剩余部分将聚焦于仅适用于顺序容器的操作。关联容器特有的操作将在第 11 章介绍。

一般来说，每个容器都定义在一个头文件中，文件名与类型名相同。即，deque 定义在头文件 deque 中，list 定义在头文件 list 中，以此类推。容器均定义为模板类（参见 3.3 节，第 86 页）。例如对 vector，我们必须提供额外信息来生成特定的容器类型。对大多数，但不是所有容器，我们还需要额外提供元素类型信息：

```
list<Sales_data>      // 保存 Sales_data 对象的 list
deque<double>         // 保存 double 的 deque
```

对容器可以保存的元素类型的限制

顺序容器几乎可以保存任意类型的元素。特别是，我们可以定义一个容器，其元素的类型是另一个容器。这种容器的定义与任何其他容器类型完全一样：在尖括号中指定元素类型（此种情况下，是另一种容器类型）：

```
vector<vector<string>> lines; // vector 的 vector
```

C++ 11 此处 lines 是一个 vector，其元素类型是 string 的 vector。



较旧的编译器可能需要在两个尖括号之间键入空格，例如，vector<vector<string> >。

虽然我们可以在容器中保存几乎任何类型，但某些容器操作对元素类型有其自己的特殊要求。我们可以为不支持特定操作需求的类型定义容器，但这种情况下就只能使用那些没有特殊要求的容器操作了。

例如，顺序容器构造函数的一个版本接受容器大小参数（参见 3.3.1 节，第 88 页），它使用了元素类型的默认构造函数。但某些类没有默认构造函数。我们可以定义一个保存这种类型对象的容器，但我们在构造这种容器时不能只传递给它一个元素数目参数：

```
// 假定 noDefault 是一个没有默认构造函数的类型
vector<noDefault> v1(10, init);      // 正确：提供了元素初始化器
vector<noDefault> v2(10);            // 错误：必须提供一个元素初始化器
```

当后面介绍容器操作时，我们还会注意到每个容器操作对元素类型的其他限制。

330

表 9.2：容器操作

类型别名	
iterator	此容器类型的迭代器类型
const_iterator	可以读取元素，但不能修改元素的迭代器类型
size_type	无符号整数类型，足够保存此种容器类型最大可能容器的大小
difference_type	带符号整数类型，足够保存两个迭代器之间的距离
value_type	元素类型
reference	元素的左值类型；与 value_type& 含义相同
const_reference	元素的 const 左值类型（即，const value_type&）
构造函数	
C c;	默认构造函数，构造空容器（array，参见第 301 页）
C c1(c2);	构造 c2 的拷贝 c1
C c(b, e);	构造 c，将迭代器 b 和 e 指定的范围内的元素拷贝到 c（array 不支持）
C c{a, b, c...};	列表初始化 c
赋值与 swap	
c1 = c2	将 c1 中的元素替换为 c2 中元素
c1 = {a, b, c...}	将 c1 中的元素替换为列表中元素（不适用于 array）
a.swap(b)	交换 a 和 b 的元素
swap(a, b)	与 a.swap(b) 等价
大小	
c.size()	c 中元素的数目（不支持 forward_list）
c.max_size()	c 可保存的最大元素数目
c.empty()	若 c 中存储了元素，返回 false，否则返回 true
添加/删除元素（不适用于 array）	
注：在不同容器中，这些操作的接口都不同	
c.insert(*args*)	将 *args* 中的元素拷贝进 c
c.emplace(*inits*)	使用 *inits* 构造 c 中的一个元素
c.erase(*args*)	删除 *args* 指定的元素
c.clear()	删除 c 中的所有元素，返回 void
关系运算符	
==, !=	所有容器都支持相等（不等）运算符
<, <=, >, >=	关系运算符（无序关联容器不支持）
获取迭代器	
c.begin(), c.end()	返回指向 c 的首元素和尾元素之后位置的迭代器
c.cbegin(), c.cend()	返回 const_iterator

续表

反向容器的额外成员（不支持 forward_list）	
reverse_iterator	按逆序寻址元素的迭代器
const_reverse_iterator	不能修改元素的逆序迭代器
c.rbegin(), c.rend()	返回指向 c 的尾元素和首元素之前位置的迭代器
c.crbegin(), c.crend()	返回 const_reverse_iterator

9.2 节练习

练习 9.2：定义一个 list 对象，其元素类型是 int 的 deque。

331

9.2.1 迭代器



与容器一样，迭代器有着公共的接口：如果一个迭代器提供某个操作，那么所有提供相同操作的迭代器对这个操作的实现方式都是相同的。例如，标准容器类型上的所有迭代器都允许我们访问容器中的元素，而所有迭代器都是通过解引用运算符来实现这个操作的。类似的，标准库容器的所有迭代器都定义了递增运算符，从当前元素移动到下一个元素。

表 3.6（第 96 页）列出了容器迭代器支持的所有操作，其中有一个例外不符合公共接口特点——forward_list 迭代器不支持递减运算符（--）。表 3.7（第 99 页）列出了迭代器支持的算术运算，这些运算只能应用于 string、vector、deque 和 array 的迭代器。我们不能将它们用于其他任何容器类型的迭代器。

迭代器范围



迭代器范围的概念是标准库的基础。

一个**迭代器范围**（iterator range）由一对迭代器表示，两个迭代器分别指向同一个容器中的元素或者是尾元素之后的位置（one past the last element）。这两个迭代器通常被称为 begin 和 end，或者是 first 和 last（可能有些误导），它们标记了容器中元素的一个范围。

虽然第二个迭代器常常被称为 last，但这种叫法有些误导，因为第二个迭代器从来都不会指向范围中的最后一个元素，而是指向尾元素之后的位置。迭代器范围中的元素包含 first 所表示的元素以及从 first 开始直至 last（但不包含 last）之间的所有元素。

这种元素范围被称为**左闭合区间**（left-inclusive interval），其标准数学描述为

```
[begin, end)
```

表示范围自 begin 开始，于 end 之前结束。迭代器 begin 和 end 必须指向相同的容器。end 可以与 begin 指向相同的位置，但不能指向 begin 之前的位置。

对构成范围的迭代器的要求

如果满足如下条件，两个迭代器 begin 和 end 构成一个迭代器范：

- 它们指向同一个容器中的元素，或者是容器最后一个元素之后的位置，且
- 我们可以通过反复递增 begin 来到达 end。换句话说，end 不在 begin 之前。



编译器不会强制这些要求。确保程序符合这些约定是程序员的责任。

使用左闭合范围蕴含的编程假定

标准库使用左闭合范围是因为这种范围有三种方便的性质。假定 begin 和 end 构成 332 一个合法的迭代器范围，则

- 如果 begin 与 end 相等，则范围为空
- 如果 begin 与 end 不等，则范围至少包含一个元素，且 begin 指向该范围中的第一个元素
- 我们可以对 begin 递增若干次，使得 begin==end

这些性质意味着我们可以像下面的代码一样用一个循环来处理一个元素范围，而这是安全的：

```
while (begin != end) {
    *begin = val;  // 正确：范围非空，因此 begin 指向一个元素
    ++begin;       // 移动迭代器，获取下一个元素
}
```

给定构成一个合法范围的迭代器 begin 和 end，若 begin==end，则范围为空。在此情况下，我们应该退出循环。如果范围不为空，begin 指向此非空范围的一个元素。因此，在 while 循环体中，可以安全地解引用 begin，因为 begin 必然指向一个元素。最后，由于每次循环对 begin 递增一次，我们确定循环最终会结束。

9.2.1 节练习

练习 9.3：构成迭代器范围的迭代器有何限制？

练习 9.4：编写函数，接受一对指向 vector<int>的迭代器和一个 int 值。在两个迭代器指定的范围中查找给定的值，返回一个布尔值来指出是否找到。

练习 9.5：重写上一题的函数，返回一个迭代器指向找到的元素。注意，程序必须处理未找到给定值的情况。

练习 9.6：下面程序有何错误？你应该如何修改它？

```
list<int> lst1;
list<int>::iterator iter1 = lst1.begin(),
                    iter2 = lst1.end();
while (iter1 < iter2) /* ... */
```

9.2.2 容器类型成员

每个容器都定义了多个类型，如表 9.2 所示（第 295 页）。我们已经使用过其中三种：size_type（参见 3.2.2 节，第 79 页）、iterator 和 const_iterator（参见 3.4.1 节，第 97 页）。

除了已经使用过的迭代器类型，大多数容器还提供反向迭代器。简单地说，反向迭代 333 器就是一种反向遍历容器的迭代器，与正向迭代器相比，各种操作的含义也都发生了颠倒。例如，对一个反向迭代器执行++操作，会得到上一个元素。我们将在 10.4.3 节（第 363 页）

介绍更多关于反向迭代器的内容。

剩下的就是类型别名了，通过类型别名，我们可以在不了解容器中元素类型的情况下使用它。如果需要元素类型，可以使用容器的 value_type。如果需要元素类型的一个引用，可以使用 reference 或 const_reference。这些元素相关的类型别名在泛型编程中非常有用，我们将在 16 章中介绍相关内容。

为了使用这些类型，我们必须显式使用其类名：

```
// iter 是通过 list<string>定义的一个迭代器类型
list<string>::iterator iter;
// count 是通过 vector<int>定义的一个 difference_type 类型
vector<int>::difference_type count;
```

这些声明语句使用了作用域运算符（参见 1.2 节，第 7 页）来说明我们希望使用 list<string>类的 iterator 成员及 vector<int>类定义的 difference_type。

9.2.2 节练习

练习 9.7：为了索引 int 的 vector 中的元素，应该使用什么类型？

练习 9.8：为了读取 string 的 list 中的元素，应该使用什么类型？如果写入 list，又该使用什么类型？



9.2.3 begin 和 end 成员

begin 和 end 操作（参见 3.4.1 节，第 95 页）生成指向容器中第一个元素和尾元素之后位置的迭代器。这两个迭代器最常见的用途是形成一个包含容器中所有元素的迭代器范围。

如表 9.2（第 295 页）所示，begin 和 end 有多个版本：带 r 的版本返回反向迭代器（我们将在 10.4.3 节（第 363 页）中介绍相关内容）；以 c 开头的版本则返回 const 迭代器：

```
list<string> a = {"Milton", "Shakespeare", "Austen"};
auto it1 = a.begin();  // list<string>::iterator
auto it2 = a.rbegin(); // list<string>::reverse_iterator
auto it3 = a.cbegin(); // list<string>::const_iterator
auto it4 = a.crbegin();// list<string>::const_reverse_iterator
```

不以 c 开头的函数都是被重载过的。也就是说，实际上有两个名为 begin 的成员。一个是 const 成员（参见 7.1.2 节，第 231 页），返回容器的 const_iterator 类型。另一个是非常量成员，返回容器的 iterator 类型。rbegin、end 和 rend 的情况类似。当我们对一个非常量对象调用这些成员时，得到的是返回 iterator 的版本。只有在对一个 const 对象调用这些函数时，才会得到一个 const 版本。与 const 指针和引用类似，可以将一个普通的 iterator 转换为对应的 const_iterator，但反之不行。

334



以 c 开头的版本是 C++新标准引入的，用以支持 auto（参见 2.5.2 节，第 61 页）与 begin 和 end 函数结合使用。过去，没有其他选择，只能显式声明希望使用哪种类型的迭代器：

```
// 显式指定类型
list<string>::iterator it5 = a.begin();
```

```
list<string>::const_iterator it6 = a.begin();
// 是 iterator 还是 const_iterator 依赖于 a 的类型
auto it7 = a.begin();  // 仅当 a 是 const 时，it7 是 const_iterator
auto it8 = a.cbegin(); // it8 是 const_iterator
```

当 auto 与 begin 或 end 结合使用时，获得的迭代器类型依赖于容器类型，与我们想要如何使用迭代器毫不相干。但以 c 开头的版本还是可以获得 const_iterator 的，而不管容器的类型是什么。



当不需要写访问时，应使用 cbegin 和 cend。

9.2.3 节练习

练习 9.9：begin 和 cbegin 两个函数有什么不同？

练习 9.10：下面 4 个对象分别是什么类型？

```
vector<int> v1;
const vector<int> v2;
auto it1 = v1.begin(), it2 = v2.begin();
auto it3 = v1.cbegin(), it4 = v2.cbegin();
```

9.2.4 容器定义和初始化



每个容器类型都定义了一个默认构造函数（参见 7.1.4 节，第 236 页）。除 array 之外，其他容器的默认构造函数都会创建一个指定类型的空容器，且都可以接受指定容器大小和元素初始值的参数。

表 9.3：容器定义和初始化

C c;	默认构造函数。如果 C 是一个 array，则 c 中元素按默认方式初始化；否则 c 为空
C c1(c2) C c1=c2	c1 初始化为 c2 的拷贝。c1 和 c2 必须是相同类型（即，它们必须是相同的容器类型，且保存的是相同的元素类型；对于 array 类型，两者还必须具有相同大小）
C c{a,b,c...} C c={a,b,c...}	c 初始化为初始化列表中元素的拷贝。列表中元素的类型必须与 C 的元素类型相容。对于 array 类型，列表中元素数目必须等于或小于 array 的大小，任何遗漏的元素都进行值初始化（参见 3.3.1 节，第 88 页）
C c(b,e)	c 初始化为迭代器 b 和 e 指定范围中的元素的拷贝。范围中元素的类型必须与 C 的元素类型相容（array 不适用）
只有顺序容器（不包括 array）的构造函数才能接受大小参数	
C seq(n)	seq 包含 n 个元素，这些元素进行了值初始化；此构造函数是 explicit 的（参见 7.5.4 节，第 265 页）。（string 不适用）
C seq(n,t)	seq 包含 n 个初始化为值 t 的元素

将一个容器初始化为另一个容器的拷贝

将一个新容器创建为另一个容器的拷贝的方法有两种：可以直接拷贝整个容器，或者

(array 除外) 拷贝由一个迭代器对指定的元素范围。

为了创建一个容器为另一个容器的拷贝，两个容器的类型及其元素类型必须匹配。不过，当传递迭代器参数来拷贝一个范围时，就不要求容器类型是相同的了。而且，新容器和原容器中的元素类型也可以不同，只要能将要拷贝的元素转换（参见 4.11 节，第 141 页）为要初始化的容器的元素类型即可。

335

```
// 每个容器有三个元素，用给定的初始化器进行初始化
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};

list<string> list2(authors);      // 正确：类型匹配
deque<string> authList(authors); // 错误：容器类型不匹配
vector<string> words(articles);  // 错误：容器类型必须匹配
// 正确：可以将 const char*元素转换为 string
forward_list<string> words(articles.begin(), articles.end());
```



当将一个容器初始化为另一个容器的拷贝时，两个容器的容器类型和元素类型都必须相同。

接受两个迭代器参数的构造函数用这两个迭代器表示我们想要拷贝的一个元素范围。与以往一样，两个迭代器分别标记想要拷贝的第一个元素和尾元素之后的位置。新容器的大小与范围中元素的数目相同。新容器中的每个元素都用范围中对应元素的值进行初始化。

由于两个迭代器表示一个范围，因此可以使用这种构造函数来拷贝一个容器中的子序列。例如，假定迭代器 it 表示 authors 中的一个元素，我们可以编写如下代码

```
// 拷贝元素，直到（但不包括）it 指向的元素
deque<string> authList(authors.begin(), it);
```

336

列表初始化

C++ 11

在新标准中，我们可以对一个容器进行列表初始化（参见 3.3.1 节，第 88 页）

```
// 每个容器有三个元素，用给定的初始化器进行初始化
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};
```

当这样做时，我们就显式地指定了容器中每个元素的值。对于除 array 之外的容器类型，初始化列表还隐含地指定了容器的大小：容器将包含与初始值一样多的元素。

与顺序容器大小相关的构造函数

除了与关联容器相同的构造函数外，顺序容器（array 除外）还提供另一个构造函数，它接受一个容器大小和一个（可选的）元素初始值。如果我们不提供元素初始值，则标准库会创建一个值初始化器（参见 3.3.1 节，第 88 页）：

```
vector<int> ivec(10, -1);          // 10 个 int 元素，每个都初始化为-1
list<string> svec(10, "hi!");      // 10 个 strings；每个都初始化为"hi!"
forward_list<int> ivec(10);        // 10 个元素，每个都初始化为 0
deque<string> svec(10);            // 10 个元素，每个都是空 string
```

如果元素类型是内置类型或者是具有默认构造函数（参见 9.2 节，第 294 页）的类类型，可以只为构造函数提供一个容器大小参数。如果元素类型没有默认构造函数，除了大小参数外，还必须指定一个显式的元素初始值。



只有顺序容器的构造函数才接受大小参数，关联容器并不支持。

标准库 array 具有固定大小

与内置数组一样，标准库 array 的大小也是类型的一部分。当定义一个 array 时，除了指定元素类型，还要指定容器大小：

```
array<int, 42>        // 类型为：保存 42 个 int 的数组
array<string, 10>     // 类型为：保存 10 个 string 的数组
```

为了使用 array 类型，我们必须同时指定元素类型和大小：

```
array<int, 10>::size_type i;      // 数组类型包括元素类型和大小
array<int>::size_type j;          // 错误：array<int>不是一个类型
```

由于大小是 array 类型的一部分，array 不支持普通的容器构造函数。这些构造函数都会确定容器的大小，要么隐式地，要么显式地。而允许用户向一个 array 构造函数传递大小参数，最好情况下也是多余的，而且容易出错。

array 大小固定的特性也影响了它所定义的构造函数的行为。与其他容器不同，一个默认构造的 array 是非空的：它包含了与其大小一样多的元素。这些元素都被默认初始化（参见 2.2.1 节，第 40 页），就像一个内置数组（参见 3.5.1 节，第 102 页）中的元素那样。如果我们对 array 进行列表初始化，初始值的数目必须等于或小于 array 的大小。如果初始值数目小于 array 的大小，则它们被用来初始化 array 中靠前的元素，所有剩余元素都会进行值初始化（参见 3.3.1 节，第 88 页）。在这两种情况下，如果元素类型是一个类类型，那么该类必须有一个默认构造函数，以使值初始化能够进行：

337

```
array<int, 10> ia1;           // 10 个默认初始化的 int
array<int, 10> ia2 = {0,1,2,3,4,5,6,7,8,9}; // 列表初始化
array<int, 10> ia3 = {42};    // ia3[0]为 42，剩余元素为 0
```

值得注意的是，虽然我们不能对内置数组类型进行拷贝或对象赋值操作（参见 3.5.1 节，第 102 页），但 array 并无此限制：

```
int digs[10] = {0,1,2,3,4,5,6,7,8,9};
int cpy[10] = digs;               // 错误：内置数组不支持拷贝或赋值
array<int, 10> digits = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> copy = digits; // 正确：只要数组类型匹配即合法
```

与其他容器一样，array 也要求初始值的类型必须与要创建的容器类型相同。此外，array 还要求元素类型和大小也都一样，因为大小是 array 类型的一部分。

9.2.4 节练习

练习 9.11：对 6 种创建和初始化 vector 对象的方法，每一种都给出一个实例。解释每个 vector 包含什么值。

练习 9.12：对于接受一个容器创建其拷贝的构造函数，和接受两个迭代器创建拷贝的构造函数，解释它们的不同。

练习 9.13： 如何从一个 list<int>初始化一个 vector<double>？从一个 vector<int>又该如何创建？编写代码验证你的答案。

9.2.5 赋值和 swap

表 9.4 中列出的与赋值相关的运算符可用于所有容器。赋值运算符将其左边容器中的全部元素替换为右边容器中元素的拷贝：

```
c1 = c2;        // 将 c1 的内容替换为 c2 中元素的拷贝
c1 = {a,b,c};   // 赋值后，c1 大小为 3
```

第一个赋值运算后，左边容器将与右边容器相等。如果两个容器原来大小不同，赋值运算后两者的大小都与右边容器的原大小相同。第二个赋值运算后，c1 的 size 变为 3，即花括号列表中值的数目。

338 与内置数组不同，标准库 array 类型允许赋值。赋值号左右两边的运算对象必须具有相同的类型：

```
array<int, 10> a1 = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> a2 = {0}; // 所有元素值均为 0
a1 = a2;  // 替换 a1 中的元素
a2 = {0}; // 错误：不能将一个花括号列表赋予数组
```

由于右边运算对象的大小可能与左边运算对象的大小不同，因此 array 类型不支持 assign，也不允许用花括号包围的值列表进行赋值。

表 9.4：容器赋值运算

c1=c2	将 c1 中的元素替换为 c2 中元素的拷贝。c1 和 c2 必须具有相同的类型
c={a,b,c...}	将 c1 中元素替换为初始化列表中元素的拷贝（array 不适用）
swap(c1,c2) c1.swap(c2)	交换 c1 和 c2 中的元素。c1 和 c2 必须具有相同的类型。swap 通常比从 c2 向 c1 拷贝元素快得多
assign 操作不适用于关联容器和 array	
seq.assign(b,e)	将 seq 中的元素替换为迭代器 b 和 e 所表示的范围中的元素。迭代器 b 和 e 不能指向 seq 中的元素
seq.assign(il)	将 seq 中的元素替换为初始化列表 il 中的元素
seq.assign(n,t)	将 seq 中的元素替换为 n 个值为 t 的元素



赋值相关运算会导致指向左边容器内部的迭代器、引用和指针失效。而 swap 操作将容器内容交换不会导致指向容器的迭代器、引用和指针失效（容器类型为 array 和 string 的情况除外）。

使用 assign（仅顺序容器）

赋值运算符要求左边和右边的运算对象具有相同的类型。它将右边运算对象中所有元素拷贝到左边运算对象中。顺序容器（array 除外）还定义了一个名为 assign 的成员，允许我们从一个不同但相容的类型赋值，或者从容器的一个子序列赋值。assign 操作用参数所指定的元素（的拷贝）替换左边容器中的所有元素。例如，我们可以用 assgin 实现将一个 vector 中的一段 char *值赋予一个 list 中的 string：

```
list<string> names;
vector<const char*> oldstyle;
names = oldstyle; // 错误：容器类型不匹配
// 正确：可以将 const char*转换为 string
names.assign(oldstyle.cbegin(), oldstyle.cend());
```

这段代码中对 assign 的调用将 names 中的元素替换为迭代器指定的范围中的元素的拷贝。assign 的参数决定了容器中将有多少个元素以及它们的值都是什么。 339



由于其旧元素被替换，因此传递给 assign 的迭代器不能指向调用 assign 的容器。

assign 的第二个版本接受一个整型值和一个元素值。它用指定数目且具有相同给定值的元素替换容器中原有的元素：

```
// 等价于 slist1.clear();
// 后跟 slist1.insert(slist1.begin(), 10, "Hiya!");
list<string> slist1(1);       // 1 个元素，为空 string
slist1.assign(10, "Hiya!");  // 10 个元素，每个都是“Hiya!”
```

使用 swap

swap 操作交换两个相同类型容器的内容。调用 swap 之后，两个容器中的元素将会交换：

```
vector<string> svec1(10); // 10 个元素的 vector
vector<string> svec2(24); // 24 个元素的 vector
swap(svec1, svec2);
```

调用 swap 后，svec1 将包含 24 个 string 元素，svec2 将包含 10 个 string。除 array 外，交换两个容器内容的操作保证会很快——元素本身并未交换，swap 只是交换了两个容器的内部数据结构。



除 array 外，swap 不对任何元素进行拷贝、删除或插入操作，因此可以保证在常数时间内完成。

元素不会被移动的事实意味着，除 string 外，指向容器的迭代器、引用和指针在 swap 操作之后都不会失效。它们仍指向 swap 操作之前所指向的那些元素。但是，在 swap 之后，这些元素已经属于不同的容器了。例如，假定 iter 在 swap 之前指向 svec1[3] 的 string，那么在 swap 之后它指向 svec2[3] 的元素。与其他容器不同，对一个 string 调用 swap 会导致迭代器、引用和指针失效。

与其他容器不同，swap 两个 array 会真正交换它们的元素。因此，交换两个 array 所需的时间与 array 中元素的数目成正比。

因此，对于 array，在 swap 操作之后，指针、引用和迭代器所绑定的元素保持不变，但元素值已经与另一个 array 中对应元素的值进行了交换。

在新标准库中，容器既提供成员函数版本的 swap，也提供非成员版本的 swap。而早期标准库版本只提供成员函数版本的 swap。非成员版本的 swap 在泛型编程中是非常重要的。统一使用非成员版本的 swap 是一个好习惯。 C++ 11

340

9.2.5 节练习

练习 9.14：编写程序，将一个 list 中的 char *指针（指向 C 风格字符串）元素赋值给一个 vector 中的 string。



9.2.6 容器大小操作

除了一个例外，每个容器类型都有三个与大小相关的操作。成员函数 size（参见 3.2.2 节，第 78 页）返回容器中元素的数目；empty 当 size 为 0 时返回布尔值 true，否则返回 false；max_size 返回一个大于或等于该类型容器所能容纳的最大元素数的值。forward_list 支持 max_size 和 empty，但不支持 size，原因我们将在下一节解释。

9.2.7 关系运算符

每个容器类型都支持相等运算符（==和!=）；除了无序关联容器外的所有容器都支持关系运算符（>、>=、<、<=）。关系运算符左右两边的运算对象必须是相同类型的容器，且必须保存相同类型的元素。即，我们只能将一个 vector<int>与另一个 vector<int>进行比较，而不能将一个 vector<int>与一个 list<int>或一个 vector<double>进行比较。

比较两个容器实际上是进行元素的逐对比较。这些运算符的工作方式与 string 的关系运算（参见 3.2.2 节，第 79 页）类似：

- 如果两个容器具有相同大小且所有元素都两两对应相等，则这两个容器相等；否则两个容器不等。
- 如果两个容器大小不同，但较小容器中每个元素都等于较大容器中的对应元素，则较小容器小于较大容器。
- 如果两个容器都不是另一个容器的前缀子序列，则它们的比较结果取决于第一个不相等的元素的比较结果。

下面的例子展示了这些关系运算符是如何工作的：

```
vector<int> v1 = { 1, 3, 5, 7, 9, 12 };
vector<int> v2 = { 1, 3, 9 };
vector<int> v3 = { 1, 3, 5, 7 };
vector<int> v4 = { 1, 3, 5, 7, 9, 12 };
v1 < v2  // true; v1 和 v2 在元素[2]处不同：v1[2]小于等于 v2[2]
v1 < v3  // false; 所有元素都相等，但 v3 中元素数目更少
v1 == v4 // true; 每个元素都相等，且 v1 和 v4 大小相同
v1 == v2 // false; v2 元素数目比 v1 少
```

341

容器的关系运算符使用元素的关系运算符完成比较



只有当其元素类型也定义了相应的比较运算符时，我们才可以使用关系运算符来比较两个容器。

容器的相等运算符实际上是使用元素的==运算符实现比较的，而其他关系运算符是使用元素的<运算符。如果元素类型不支持所需运算符，那么保存这种元素的容器就不能使用相应的关系运算。例如，我们在第 7 章中定义的 Sales_data 类型并未定义==和<运算。因此，就不能比较两个保存 Sales_data 元素的容器：

```
vector<Sales_data> storeA, storeB;
if (storeA < storeB) // 错误：Sales_data 没有<运算符
```

9.2.7 节练习

练习 9.15：编写程序，判定两个 vector<int>是否相等。

练习 9.16：重写上一题的程序，比较一个 list<int>中的元素和一个 vector<int>中的元素。

练习 9.17：假定 c1 和 c2 是两个容器，下面的比较操作有何限制（如果有的话）？

```
if (c1 < c2)
```

9.3 顺序容器操作

顺序容器和关联容器的不同之处在于两者组织元素的方式。这些不同之处直接关系到了元素如何存储、访问、添加以及删除。上一节介绍了所有容器都支持的操作（罗列于表 9.2（第 295 页））。本章剩余部分将介绍顺序容器所特有的操作。

9.3.1 向顺序容器添加元素



除 array 外，所有标准库容器都提供灵活的内存管理。在运行时可以动态添加或删除元素来改变容器大小。表 9.5 列出了向顺序容器（非 array）添加元素的操作。

表 9.5：向顺序容器添加元素的操作

这些操作会改变容器的大小；array 不支持这些操作。
forward_list 有自己专有版本的 insert 和 emplace；参见 9.3.4 节（第 312 页）。
forward_list 不支持 push_back 和 emplace_back。
vector 和 string 不支持 push_front 和 emplace_front。

c.push_back(t) c.emplace_back(*args*)	在 c 的尾部创建一个值为 t 或由 *args* 创建的元素。返回 void
c.push_front(t) c.emplace_front(*args*)	在 c 的头部创建一个值为 t 或由 *args* 创建的元素。返回 void
c.insert(p,t) c.emplace(p,*args*)	在迭代器 p 指向的元素之前创建一个值为 t 或由 *args* 创建的元素。返回指向新添加的元素的迭代器
c.insert(p,n,t)	在迭代器 p 指向的元素之前插入 n 个值为 t 的元素。返回指向新添加的第一个元素的迭代器；若 n 为 0，则返回 p
c.insert(p,b,e)	将迭代器 b 和 e 指定的范围内的元素插入到迭代器 p 指向的元素之前。b 和 e 不能指向 c 中的元素。返回指向新添加的第一个元素的迭代器；若范围为空，则返回 p
c.insert(p,il)	il 是一个花括号包围的元素值列表。将这些给定值插入到迭代器 p 指向的元素之前。返回指向新添加的第一个元素的迭代器；若列表为空，则返回 p



向一个 vector、string 或 deque 插入元素会使所有指向容器的迭代器、引用和指针失效。

当我们使用这些操作时，必须记得不同容器使用不同的策略来分配元素空间，而这些策略直接影响性能。在一个 vector 或 string 的尾部之外的任何位置，或是一个 deque 的首尾之外的任何位置添加元素，都需要移动元素。而且，向一个 vector 或 string 添
342 加元素可能引起整个对象存储空间的重新分配。重新分配一个对象的存储空间需要分配新的内存，并将元素从旧的空间移动到新的空间中。

使用 push_back

在 3.3.2 节（第 90 页）中，我们看到 push_back 将一个元素追加到一个 vector 的尾部。除 array 和 forward_list 之外，每个顺序容器（包括 string 类型）都支持 push_back。

例如，下面的循环每次读取一个 string 到 word 中，然后追加到容器尾部：

```
// 从标准输入读取数据，将每个单词放到容器末尾
string word;
while (cin >> word)
    container.push_back(word);
```

对 push_back 的调用在 container 尾部创建了一个新的元素，将 container 的 size 增大了 1。该元素的值为 word 的一个拷贝。container 的类型可以是 list、vector 或 deque。

由于 string 是一个字符容器，我们也可以用 push_back 在 string 末尾添加字符：

```
void pluralize(size_t cnt, string &word)
{
    if (cnt > 1)
        word.push_back('s'); // 等价于 word += 's'
}
```

关键概念：容器元素是拷贝

当我们用一个对象来初始化容器时，或将一个对象插入到容器中时，实际上放入到容器中的是对象值的一个拷贝，而不是对象本身。就像我们将一个对象传递给非引用参数（参见 3.2.2 节，第 79 页）一样，容器中的元素与提供值的对象之间没有任何关联。随后对容器中元素的任何改变都不会影响到原始对象，反之亦然。

使用 push_front

除了 push_back，list、forward_list 和 deque 容器还支持名为 push_front 的类似操作。此操作将元素插入到容器头部：

```
list<int> ilist;
// 将元素添加到 ilist 开头
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_front(ix);
```

此循环将元素 0、1、2、3 添加到 ilist 头部。每个元素都插入到 list 的新的开始位
343 置（new beginning）。即，当我们插入 1 时，它会被放置在 0 之前，2 被放置在 1 之前，依此类推。因此，在循环中以这种方式将元素添加到容器中，最终会形成逆序。在循环执行完毕后，ilist 保存序列 3、2、1、0。

注意，deque 像 vector 一样提供了随机访问元素的能力，但它提供了 vector 所

不支持的 push_front。deque 保证在容器首尾进行插入和删除元素的操作都只花费常数时间。与 vector 一样，在 deque 首尾之外的位置插入元素会很耗时。

在容器中的特定位置添加元素

push_back 和 push_front 操作提供了一种方便地在顺序容器尾部或头部插入单个元素的方法。insert 成员提供了更一般的添加功能，它允许我们在容器中任意位置插入 0 个或多个元素。vector、deque、list 和 string 都支持 insert 成员。forward_list 提供了特殊版本的 insert 成员，我们将在 9.3.4 节（第 312 页）中介绍。

每个 insert 函数都接受一个迭代器作为其第一个参数。迭代器指出了在容器中什么位置放置新元素。它可以指向容器中任何位置，包括容器尾部之后的下一个位置。由于迭代器可能指向容器尾部之后不存在的元素的位置，而且在容器开始位置插入元素是很有用的功能，所以 insert 函数将元素插入到迭代器所指定的位置之前。例如，下面的语句

344

```
slist.insert(iter, "Hello!"); // 将"Hello!"添加到 iter 之前的位置
```

将一个值为"Hello"的 string 插入到 iter 指向的元素之前的位置。

虽然某些容器不支持 push_front 操作，但它们对于 insert 操作并无类似的限制（插入开始位置）。因此我们可以将元素插入到容器的开始位置，而不必担心容器是否支持 push_front：

```
vector<string> svec;
list<string> slist;

// 等价于调用 slist.push_front("Hello!");
slist.insert(slist.begin(), "Hello!");

// vector 不支持 push_front，但我们可以插入到 begin()之前
// 警告：插入到 vector 末尾之外的任何位置都可能很慢
svec.insert(svec.begin(), "Hello!");
```



将元素插入到 vector、deque 和 string 中的任何位置都是合法的。然而，这样做可能很耗时。

插入范围内元素

除了第一个迭代器参数之外，insert 函数还可以接受更多的参数，这与容器构造函数类似。其中一个版本接受一个元素数目和一个值，它将指定数量的元素添加到指定位置之前，这些元素都按给定值初始化：

```
svec.insert(svec.end(), 10, "Anna");
```

这行代码将 10 个元素插入到 svec 的末尾，并将所有元素都初始化为 string "Anna"。

接受一对迭代器或一个初始化列表的 insert 版本将给定范围中的元素插入到指定位置之前：

```
vector<string> v = {"quasi", "simba", "frollo", "scar"};
// 将 v 的最后两个元素添加到 slist 的开始位置
slist.insert(slist.begin(), v.end() - 2, v.end());
slist.insert(slist.end(), {"these", "words", "will",
                           "go", "at", "the", "end"});
```

```
// 运行时错误：迭代器表示要拷贝的范围，不能指向与目的位置相同的容器
slist.insert(slist.begin(), slist.begin(), slist.end());
```

如果我们传递给 insert 一对迭代器，它们不能指向添加元素的目标容器。

在新标准下，接受元素个数或范围的 insert 版本返回指向第一个新加入元素的迭代器。(在旧版本的标准库中，这些操作返回 void。)如果范围为空，不插入任何元素，insert 操作会将第一个参数返回。

C++ 11

345 使用 insert 的返回值

通过使用 insert 的返回值，可以在容器中一个特定位置反复插入元素：

```
list<string> lst;
auto iter = lst.begin();
while (cin >> word)
    iter = lst.insert(iter, word); // 等价于调用 push_front
```



理解这个循环是如何工作的非常重要，特别是理解这个循环为什么等价于调用 push_front 尤为重要。

在循环之前，我们将 iter 初始化为 lst.begin()。第一次调用 insert 会将我们刚刚读入的 string 插入到 iter 所指向的元素之前的位置。insert 返回的迭代器恰好指向这个新元素。我们将此迭代器赋予 iter 并重复循环，读取下一个单词。只要继续有单词读入，每步 while 循环就会将一个新元素插入到 iter 之前，并将 iter 改变为新加入元素的位置。此元素为（新的）首元素。因此，每步循环将一个新元素插入到 list 首元素之前的位置。

使用 emplace 操作

C++ 11

新标准引入了三个新成员——emplace_front、emplace 和 emplace_back，这些操作构造而不是拷贝元素。这些操作分别对应 push_front、insert 和 push_back，允许我们将元素放置在容器头部、一个指定位置之前或容器尾部。

当调用 push 或 insert 成员函数时，我们将元素类型的对象传递给它们，这些对象被拷贝到容器中。而当我们调用一个 emplace 成员函数时，则是将参数传递给元素类型的构造函数。emplace 成员使用这些参数在容器管理的内存空间中直接构造元素。例如，假定 c 保存 Sales_data（参见 7.1.4 节，第 237 页）元素：

```
// 在 c 的末尾构造一个 Sales_data 对象
// 使用三个参数的 Sales_data 构造函数
c.emplace_back("978-0590353403", 25, 15.99);
// 错误：没有接受三个参数的 push_back 版本
c.push_back("978-0590353403", 25, 15.99);
// 正确：创建一个临时的 Sales_data 对象传递给 push_back
c.push_back(Sales_data("978-0590353403", 25, 15.99));
```

其中对 emplace_back 的调用和第二个 push_back 调用都会创建新的 Sales_data 对象。在调用 emplace_back 时，会在容器管理的内存空间中直接创建对象。而调用 push_back 则会创建一个局部临时对象，并将其压入容器中。

emplace 函数的参数根据元素类型而变化，参数必须与元素类型的构造函数相匹配：

346

```
// iter 指向 c 中一个元素，其中保存了 Sales_data 元素
```

```
c.emplace_back(); // 使用 Sales_data 的默认构造函数
c.emplace(iter, "999-999999999"); // 使用 Sales_data(string)
// 使用 Sales_data 的接受一个 ISBN、一个 count 和一个 price 的构造函数
c.emplace_front("978-0590353403", 25, 15.99);
```



emplace 函数在容器中直接构造元素。传递给 emplace 函数的参数必须与元素类型的构造函数相匹配。

9.3.1 节练习

练习 9.18：编写程序，从标准输入读取 string 序列，存入一个 deque 中。编写一个循环，用迭代器打印 deque 中的元素。

练习 9.19：重写上题的程序，用 list 替代 deque。列出程序要做出哪些改变。

练习 9.20：编写程序，从一个 list<int>拷贝元素到两个 deque 中。值为偶数的所有元素都拷贝到一个 deque 中，而奇数值元素都拷贝到另一个 deque 中。

练习 9.21：如果我们将第 308 页中使用 insert 返回值将元素添加到 list 中的循环程序改写为将元素插入到 vector 中，分析循环将如何工作。

练习 9.22：假定 iv 是一个 int 的 vector，下面的程序存在什么错误？你将如何修改？

```
vector<int>::iterator iter = iv.begin(),
                      mid = iv.begin() + iv.size()/2;
while (iter != mid)
     if (*iter == some_val)
          iv.insert(iter, 2 * some_val);
```

9.3.2 访问元素



表 9.6 列出了我们可以用来在顺序容器中访问元素的操作。如果容器中没有元素，访问操作的结果是未定义的。

包括 array 在内的每个顺序容器都有一个 front 成员函数，而除 forward_list 之外的所有顺序容器都有一个 back 成员函数。这两个操作分别返回首元素和尾元素的引用：

```
// 在解引用一个迭代器或调用 front 或 back 之前检查是否有元素
if (!c.empty()) {
     // val 和 val2 是 c 中第一个元素值的拷贝
     auto val = *c.begin(), val2 = c.front();
     // val3 和 val4 是 c 中最后一个元素值的拷贝
     auto last = c.end();
     auto val3 = *(--last); // 不能递减 forward_list 迭代器
     auto val4 = c.back();  // forward_list 不支持
}
```

此程序用两种不同方式来获取 c 中的首元素和尾元素的引用。直接的方法是调用 front 和 back。而间接的方法是通过解引用 begin 返回的迭代器来获得首元素的引用，以及通过递减然后解引用 end 返回的迭代器来获得尾元素的引用。 347

这个程序有两点值得注意：迭代器 end 指向的是容器尾元素之后的（不存在的）元

素。为了获取尾元素，必须首先递减此迭代器。另一个重要之处是，在调用 front 和 back（或解引用 begin 和 end 返回的迭代器）之前，要确保 c 非空。如果容器为空，if 中操作的行为将是未定义的。

表 9.6：在顺序容器中访问元素的操作

at 和下标操作只适用于 string、vector、deque 和 array。
back 不适用于 forward_list。

c.back()	返回 c 中尾元素的引用。若 c 为空，函数行为未定义
c.front()	返回 c 中首元素的引用。若 c 为空，函数行为未定义
c[n]	返回 c 中下标为 n 的元素的引用，n 是一个无符号整数。若 n>=c.size()，则函数行为未定义
c.at(n)	返回下标为 n 的元素的引用。如果下标越界，则抛出一 out_of_range 异常



对一个空容器调用 front 和 back，就像使用一个越界的下标一样，是一种严重的程序设计错误。

访问成员函数返回的是引用

在容器中访问元素的成员函数（即，front、back、下标和 at）返回的都是引用。如果容器是一个 const 对象，则返回值是 const 的引用。如果容器不是 const 的，则返回值是普通引用，我们可以用来改变元素的值：

```
if (!c.empty()) {
    c.front() = 42;          // 将 42 赋予 c 中的第一个元素
    auto &v = c.back();      // 获得指向最后一个元素的引用
    v = 1024;                // 改变 c 中的元素
    auto v2 = c.back();      // v2 不是一个引用，它是 c.back()的一个拷贝
    v2 = 0;                  // 未改变 c 中的元素
}
```

与往常一样，如果我们使用 auto 变量来保存这些函数的返回值，并且希望使用此变量来改变元素的值，必须记得将变量定义为引用类型。

下标操作和安全的随机访问

提供快速随机访问的容器（string、vector、deque 和 array）也都提供下标运算符（参见 3.3.3 节，第 91 页）。就像我们已经看到的那样，下标运算符接受一个下标参数，返回容器中该位置的元素的引用。给定下标必须“在范围内”（即，大于等于 0，且小于容器的大小）。保证下标有效是程序员的责任，下标运算符并不检查下标是否在合法范围内。使用越界的下标是一种严重的程序设计错误，而且编译器并不检查这种错误。

348

如果我们希望确保下标是合法的，可以使用 at 成员函数。at 成员函数类似下标运算符，但如果下标越界，at 会抛出一个 out_of_range 异常（参见 5.6 节，第 173 页）：

```
vector<string> svec;      // 空 vector
cout << svec[0];          // 运行时错误：svec 中没有元素！
cout << svec.at(0);       // 抛出一个 out_of_range 异常
```

9.3.2 节练习

练习 9.23：在本节第一个程序（第 309 页）中，若 c.size() 为 1，则 val、val2、val3 和 val4 的值会是什么？

练习 9.24：编写程序，分别使用 at、下标运算符、front 和 begin 提取一个 vector 中的第一个元素。在一个空 vector 上测试你的程序。

9.3.3 删除元素



与添加元素的多种方式类似，（非 array）容器也有多种删除元素的方式。表 9.7 列出了这些成员函数。

表 9.7：顺序容器的删除操作

这些操作会改变容器的大小，所以不适用于 array。
forward_list 有特殊版本的 erase，参见 9.3.4 节（第 312 页）。
forward_list 不支持 pop_back；vector 和 string 不支持 pop_front。

c.pop_back()	删除 c 中尾元素。若 c 为空，则函数行为未定义。函数返回 void
c.pop_front()	删除 c 中首元素。若 c 为空，则函数行为未定义。函数返回 void
c.erase(p)	删除迭代器 p 所指定的元素，返回一个指向被删元素之后元素的迭代器，若 p 指向尾元素，则返回尾后（off-the-end）迭代器。若 p 是尾后迭代器，则函数行为未定义
c.erase(b,e)	删除迭代器 b 和 e 所指定范围内的元素。返回一个指向最后一个被删元素之后元素的迭代器，若 e 本身就是尾后迭代器，则函数也返回尾后迭代器
c.clear()	删除 c 中的所有元素。返回 void



删除 deque 中除首尾位置之外的任何元素都会使所有迭代器、引用和指针失效。指向 vector 或 string 中删除点之后位置的迭代器、引用和指针都会失效。



删除元素的成员函数并不检查其参数。在删除元素之前，程序员必须确保它（们）是存在的。

pop_front 和 pop_back 成员函数

pop_front 和 pop_back 成员函数分别删除首元素和尾元素。与 vector 和 string 不支持 push_front 一样，这些类型也不支持 pop_front。类似的，forward_list 不支持 pop_back。与元素访问成员函数类似，不能对一个空容器执行弹出操作。

这些操作返回 void。如果你需要弹出的元素的值，就必须在执行弹出操作之前保存它：

```
while (!ilist.empty()) {
    process(ilist.front()); // 对 ilist 的首元素进行一些处理
    ilist.pop_front();      // 完成处理后删除首元素
}
```

349 从容器内部删除一个元素

成员函数 erase 从容器中指定位置删除元素。我们可以删除由一个迭代器指定的单个元素，也可以删除由一对迭代器指定的范围内的所有元素。两种形式的 erase 都返回指向删除的(最后一个)元素之后位置的迭代器。即，若 j 是 i 之后的元素，那么 erase(i) 将返回指向 j 的迭代器。

例如，下面的循环删除一个 list 中的所有奇数元素：

```
list<int> lst = {0,1,2,3,4,5,6,7,8,9};
auto it = lst.begin();
while (it != lst.end())
    if (*it % 2)                // 若元素为奇数
        it = lst.erase(it);     // 删除此元素
    else
        ++it;
```

每个循环步中，首先检查当前元素是否是奇数。如果是，就删除该元素，并将 it 设置为我们所删除的元素之后的元素。如果*it 为偶数，我们将 it 递增，从而在下一步循环检查下一个元素。

删除多个元素

接受一对迭代器的 erase 版本允许我们删除一个范围内的元素：

```
// 删除两个迭代器表示的范围内的元素
// 返回指向最后一个被删元素之后位置的迭代器
elem1 = slist.erase(elem1, elem2); // 调用后，elem1 == elem2
```

迭代器 elem1 指向我们要删除的第一个元素，elem2 指向我们要删除的最后一个元素之后的位置。

350 为了删除一个容器中的所有元素，我们既可以调用 clear，也可以用 begin 和 end 获得的迭代器作为参数调用 erase：

```
slist.clear(); // 删除容器中所有元素
slist.erase(slist.begin(), slist.end()); // 等价调用
```

9.3.3 节练习

练习 9.25：对于第 312 页中删除一个范围内的元素的程序，如果 elem1 与 elem2 相等会发生什么？如果 elem2 是尾后迭代器，或者 elem1 和 elem2 皆为尾后迭代器，又会发生什么？

练习 9.26：使用下面代码定义的 ia，将 ia 拷贝到一个 vector 和一个 list 中。使用单迭代器版本的 erase 从 list 中删除奇数元素，从 vector 中删除偶数元素。

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
```



9.3.4 特殊的 forward_list 操作

为了理解 forward_list 为什么有特殊版本的添加和删除操作，考虑当我们从一个单向链表中删除一个元素时会发生什么。如图 9.1 所示，删除一个元素会改变序列中的链接。在此情况下，删除 $elem_3$ 会改变 $elem_2$，$elem_2$ 原来指向 $elem_3$，但删除 $elem_3$ 后，$elem_2$ 指向了 $elem_4$。

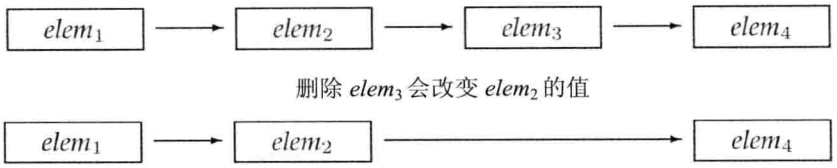


图 9.1：forward_list 的特殊操作

当添加或删除一个元素时，删除或添加的元素之前的那个元素的后继会发生改变。为了添加或删除一个元素，我们需要访问其前驱，以便改变前驱的链接。但是，forward_list 是单向链表。在一个单向链表中，没有简单的方法来获取一个元素的前驱。出于这个原因，在一个 forward_list 中添加或删除元素的操作是通过改变给定元素之后的元素来完成的。这样，我们总是可以访问到被添加或删除操作所影响的元素。

由于这些操作与其他容器上的操作的实现方式不同，forward_list 并未定义 insert、emplace 和 erase，而是定义了名为 insert_after、emplace_after 和 erase_after 的操作（参见表 9.8）。例如，在我们的例子中，为了删除 $elem_3$，应该用指向 $elem_2$ 的迭代器调用 erase_after。为了支持这些操作，forward_list 也定义了 before_begin，它返回一个**首前**（off-the-beginning）迭代器。这个迭代器允许我们在链表首元素之前并不存在的元素“之后”添加或删除元素（亦即在链表首元素之前添加删除元素）。 351

表 9.8：在 forward_list 中插入或删除元素的操作

lst.before_begin() lst.cbefore_begin()	返回指向链表首元素之前不存在的元素的迭代器。此迭代器不能解引用。cbefore_begin()返回一个 const_iterator
lst.insert_after(p,t) lst.insert_after(p,n,t) lst.insert_after(p,b,e) lst.insert_after(p,il)	在迭代器 p 之后的位置插入元素。t 是一个对象，n 是数量，b 和 e 是表示范围的一对迭代器（b 和 e 不能指向 lst 内），il 是一个花括号列表。返回一个指向最后一个插入元素的迭代器。如果范围为空，则返回 p。若 p 为尾后迭代器，则函数行为未定义
emplace_after(p,*args*)	使用 *args* 在 p 指定的位置之后创建一个元素。返回一个指向这个新元素的迭代器。若 p 为尾后迭代器，则函数行为未定义
lst.erase_after(p) lst.erase_after(b,e)	删除 p 指向的位置之后的元素，或删除从 b 之后直到（但不包含）e 之间的元素。返回一个指向被删元素之后元素的迭代器，若不存在这样的元素，则返回尾后迭代器。如果 p 指向 lst 的尾元素或者是一个尾后迭代器，则函数行为未定义

当在 forward_list 中添加或删除元素时，我们必须关注两个迭代器——一个指向我们要处理的元素，另一个指向其前驱。例如，可以改写第 312 页中从 list 中删除奇数元素的循环程序，将其改为从 forward_list 中删除元素：

```
forward_list<int> flst = {0,1,2,3,4,5,6,7,8,9};
auto prev = flst.before_begin();          // 表示 flst 的“首前元素”
auto curr = flst.begin();                 // 表示 flst 中的第一个元素
while (curr != flst.end()) {              // 仍有元素要处理
    if (*curr % 2)                        // 若元素为奇数
        curr = flst.erase_after(prev);    // 删除它并移动 curr
    else {
        prev = curr;      // 移动迭代器 curr，指向下一个元素，prev 指向
```

```
            ++curr;        // curr之前的元素
        }
    }
```

此例中，curr 表示我们要处理的元素，prev 表示 curr 的前驱。调用 begin 来初始化 curr，这样第一步循环就会检查第一个元素是否是奇数。我们用 before_begin 来初始化 prev，它返回指向 curr 之前不存在的元素的迭代器。

当找到奇数元素后，我们将 prev 传递给 erase_after。此调用将 prev 之后的元素删除，即，删除 curr 指向的元素。然后我们将 curr 重置为 erase_after 的返回值，使得 curr 指向序列中下一个元素，prev 保持不变，仍指向（新）curr 之前的元素。如果 curr 指向的元素不是奇数，在 else 中我们将两个迭代器都向前移动。

352

9.3.4 节练习

练习 9.27：编写程序，查找并删除 forward_list<int>中的奇数元素。

练习 9.28：编写函数，接受一个 forward_list<string>和两个 string 共三个参数。函数应在链表中查找第一个 string，并将第二个 string 插入到紧接着第一个 string 之后的位置。若第一个 string 未在链表中，则将第二个 string 插入到链表末尾。

9.3.5 改变容器大小

如表 9.9 所描述，我们可以用 resize 来增大或缩小容器，与往常一样，array 不支持 resize。如果当前大小大于所要求的大小，容器后部的元素会被删除；如果当前大小小于新大小，会将新元素添加到容器后部：

```
list<int> ilist(10, 42);      // 10个int：每个的值都是42
ilist.resize(15);             // 将5个值为0的元素添加到ilist的末尾
ilist.resize(25, -1);         // 将10个值为-1的元素添加到ilist的末尾
ilist.resize(5);              // 从ilist末尾删除20个元素
```

resize 操作接受一个可选的元素值参数，用来初始化添加到容器中的元素。如果调用者未提供此参数，新元素进行值初始化（参见 3.3.1 节，第 88 页）。如果容器保存的是类类型元素，且 resize 向容器添加新元素，则我们必须提供初始值，或者元素类型必须提供一个默认构造函数。

表 9.9：顺序容器大小操作

resize 不适用于 array	
c.resize(n)	调整 c 的大小为 n 个元素。若 n<c.size()，则多出的元素被丢弃。若必须添加新元素，对新元素进行值初始化
c. resize(n,t)	调整 c 的大小为 n 个元素。任何新添加的元素都初始化为值 t



如果 resize 缩小容器，则指向被删除元素的迭代器、引用和指针都会失效；对 vector、string 或 deque 进行 resize 可能导致迭代器、指针和引用失效。

9.3.5 节练习

353

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

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



9.3.6 容器操作可能使迭代器失效

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

在向容器添加元素后：

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

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

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



使用失效的迭代器、指针或引用是严重的运行时错误。

建议：管理迭代器

354

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

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

编写改变容器的循环程序

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

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

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

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

不要保存 end 返回的迭代器

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

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

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

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

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



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

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

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

9.3.6 节练习

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

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

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

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

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

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

9.4 vector 对象是如何增长的



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

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

为了避免这种代价，标准库实现者采用了可以减少容器空间重新分配次数的策略。当

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

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

管理容量的成员函数

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

表 9.10：容器大小管理操作

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



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

357

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

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

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

C++11

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

capacity 和 size

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

下面的代码展示了 size 和 capacity 之间的相互作用：

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

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

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

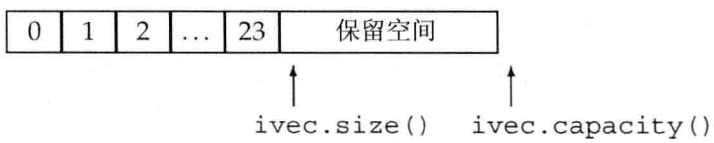
当在我们的系统上运行时，这段程序得到如下输出：

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

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

358

可以想象 ivec 的当前状态如下图所示：



现在可以预分配一些额外空间：

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

程序的输出表明 reserve 严格按照我们需求的大小分配了新的空间：

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

接下来可以用光这些预留空间：

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

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

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

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

如果我们现在再添加一个新元素，vector 就不得不重新分配空间：

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

这段程序的输出为

359

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

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

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

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

调用 shrink_to_fit 只是一个请求，标准库并不保证退还内存。



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

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

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

9.4 节练习

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

练习 9.36：一个容器的 capacity 可能小于它的 size 吗？

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

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

练习 9.39：解释下面程序片段做了什么：

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

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

360

9.5 额外的 string 操作

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

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

9.5.1 构造 string 的其他方法



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

表 9.11：构造 string 的其他方法

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

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

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

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

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

substr 操作

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

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

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

表 9.12：子字符串操作	
s.substr(pos,n)	返回一个 string，包含 s 中从 pos 开始的 n 个字符的拷贝。pos 的默认值为 0。n 的默认值为 s.size()-pos，即拷贝从 pos 开始的所有字符

9.5.1 节练习

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

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



9.5.2 改变 string 的其他方法

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

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

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

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

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

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

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

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

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

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

append 和 replace 函数

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

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

replace 操作是调用 erase 和 insert 的一种简写形式：

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

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

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

在此调用中，删除了 3 个字符，但在其位置插入了 5 个新字符。

表 9.13：修改 string 的操作 363

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

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

str 不能与 s 相同，迭代器 b 和 e 不能指向 s。

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

续表

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

改变 string 的多种重载函数

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

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

replace 函数提供了两种指定删除元素范围的方式。可以通过一个位置和一个长度
364
来指定范围，也可以通过一个迭代器范围来指定。insert 函数允许我们用两种方式指定插入点：用一个下标或一个迭代器。在两种情况下，新元素都会插入到给定下标（或迭代器）之前的位置。

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

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

9.5.2 节练习

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

练习 9.44：重写上一题的函数，这次使用一个下标和 replace。

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

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

9.5.3 string 搜索操作

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



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

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

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

365

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

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

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

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

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

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

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

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

表 9.14：string 搜索操作

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

续表

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

指定在哪里开始搜索

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

366

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

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

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

逆向搜索

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

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

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

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

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

每个操作都接受一个可选的第二参数，可用来指出从什么位置开始搜索。

9.5.3 节练习

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

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

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

9.5.4 compare 函数



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

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

367

表 9.15：s.compare 的几种参数形式

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

9.5.5 数值转换



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

新标准引入了多个函数，可以实现数值数据与标准库 string 之间的转换：

C++ 11

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

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

要转换为数值的 string 中第一个非空白符必须是数值中可能出现的字符：

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

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

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



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

表 9.16：string 和数值之间的转换

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

9.5.5 节练习

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

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

9.6 容器适配器



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

369

表 9.17：所有容器适配器都支持的操作和类型

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

定义一个适配器

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

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

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

370

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

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

栈适配器

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

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

其中，声明语句

```
stack<int> intStack; // 空栈
```

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

表 9.18：表 9.17 未列出的栈操作

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

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

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

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

队列适配器

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

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

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

续表

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

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

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

9.6 节练习

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

372 小结

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

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

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

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

术语表

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

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

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

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

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

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

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

373

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

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

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

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

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

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

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

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

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

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

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

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

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

第 10 章

泛型算法

内容

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

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

泛型算法和关于迭代器的更多细节，构成了本章的主要内容。

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

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

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

10.1 概述

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

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

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

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

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

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

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

377

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

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

还可以在序列的子范围中查找，只需将指向子范围首元素和尾元素之后位置的迭代器

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

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

算法如何工作

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

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

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

迭代器令算法不依赖于容器，……

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

……，但算法依赖于元素类型的操作

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

10.1 节练习

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

练习 10.2：重做上一题，但读取 string 序列存入 list 中。

关键概念：算法永远不会执行容器的操作

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

的值，也可能在容器内移动元素，但永远不会直接添加或删除元素。

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



10.2 初识泛型算法

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

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

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



10.2.1 只读算法

379

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

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

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

这条语句将 sum 设置为 vec 中元素的和，和的初值被设置为 0。



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

算法和元素类型

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

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

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

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

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

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



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

操作两个序列的算法

380

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

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

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

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



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

10.2.1 节练习

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

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

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

10.2.2 写容器元素的算法



一些算法将新值赋予序列中的元素。当我们使用这类算法时，必须注意确保序列原大

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

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

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

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

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

381

关键概念：迭代器参数

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

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

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



算法不检查写操作

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

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

函数 fill_n 假定写入指定个元素是安全的。即，如下形式的调用

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

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

382

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

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

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



向目的位置迭代器写入数据的算法假定目的位置足够大，能容纳要写入的元素。

介绍 back_inserter

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

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

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

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

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

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

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

拷贝算法

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

我们可以用 copy 实现内置数组的拷贝，如下面代码所示：

383

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

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

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

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

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

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

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

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

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

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

10.2.2 节练习

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

练习 10.7：下面程序是否有错误？如果有，请改正。

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

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

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



10.2.3 重排容器元素的算法

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

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

为了便于说明问题，我们将使用下面简单的故事作为输入：

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

给定此输入，我们的程序应该生成如下 vector：

fox	jumps	over	quick	red	slow	the	turtle

消除重复单词

384

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

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

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

fox	jumps	over	quick	red	red	slow	the	the	turtle

注意，单词 red 和 the 各出现了两次。

使用 unique



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

385

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

end_unique
（最后一个不重复元素之后的位置）

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



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

使用容器操作删除元素



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

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

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

10.2.3 节练习

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

练习 10.10：你认为算法不改变容器大小的原因是什么？

10.3 定制操作

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

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

10.3.1 向算法传递函数

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

谓词

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

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

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

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

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

排序算法

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

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

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

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

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

10.3.1 节练习

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

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

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

10.3.2 lambda 表达式

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

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

388 我们将此函数命名为 biggies，其框架如下所示：

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

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

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

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

介绍 lambda

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

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

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

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

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

我们可以忽略参数列表和返回类型，但必须永远包含捕获列表和函数体

389

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

此例中，我们定义了一个可调用对象 f，它不接受参数，返回 42。

lambda 的调用方式与普通函数的调用方式相同，都是使用调用运算符：

```
cout << f() << endl; // 打印 42
```

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



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

向 lambda 传递参数

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

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

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

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

如下所示，可以使用此 lambda 来调用 stable_sort：

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

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

使用捕获列表

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

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

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

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

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

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

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



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

调用 find_if

使用此 lambda，我们就可以查找第一个长度大于等于 sz 的元素：

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

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

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

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

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

391

for_each 算法

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

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

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

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



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

完整的 biggies

到目前为止，我们已经解决了程序的所有细节，下面就是完整的程序：

```
void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    elimDups(words); // 将 words 按字典序排序，删除重复单词
    // 按长度排序，长度相同的单词维持字典序
    stable_sort(words.begin(), words.end(),
                [](const string &a, const string &b)
                  { return a.size() < b.size();});
    // 获取一个迭代器，指向第一个满足 size()>= sz 的元素
    auto wc = find_if(words.begin(), words.end(),
                [sz](const string &a)
                   { return a.size() >= sz; });
    // 计算满足 size >= sz 的元素的数目
    auto count = words.end() - wc;
    cout << count << " " << make_plural(count, "word", "s")
         << " of length " << sz << " or longer" << endl;
    // 打印长度大于等于给定值的单词，每个单词后面接一个空格
    for_each(wc, words.end(),
             [](const string &s){cout << s << " ";});
    cout << endl;
}
```

10.3.2 节练习

392

练习 10.14：编写一个 lambda，接受两个 int，返回它们的和。

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

练习 10.16：使用 lambda 编写你自己版本的 biggies。

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

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

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

10.3.3 lambda 捕获和返回

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

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

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

值捕获

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

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

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

393 ### 引用捕获

我们定义 lambda 时可以采用引用方式捕获变量。例如：

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

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

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

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

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

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

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

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

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



当以引用方式捕获一个变量时，必须保证在 lambda 执行时变量是存在的。

建议：尽量保持 lambda 的变量捕获简单化

394

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

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

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

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

隐式捕获

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

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

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

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

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

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

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

表 10.1：lambda 捕获列表

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

可变 lambda

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

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

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

396

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

指定 lambda 返回类型

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

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

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

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

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

但是，如果我们将程序改写为看起来是等价的 if 语句，就会产生编译错误：

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

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

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




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

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

397

10.3.3 节练习

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

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

10.3.4 参数绑定

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

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

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

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

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

标准库 bind 函数

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

C++ 11

398

调用 bind 的一般形式为：

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

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

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

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

绑定 check_size 的 sz 参数

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

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

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

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

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

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

替换为如下使用 check_size 的版本：

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

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

使用 placeholders 名字

399

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

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

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

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

```
using namespace namespace_name;
```

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

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

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

bind 的参数

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

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

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

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

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

映射为

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

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

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

400 ### 用 bind 重排参数顺序

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

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

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

绑定引用参数

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

例如，为了替换一个引用方式捕获 ostream 的 lambda：

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

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

可以很容易地编写一个函数，完成相同的工作：

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

但是，不能直接用 bind 来代替对 os 的捕获：

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

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

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

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

向后兼容：参数绑定

401

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

10.3.4 节练习

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

练习 10.23：bind 接受几个参数？

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

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

10.4 再探迭代器

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

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

关联的 IO 流。

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



10.4.1 插入迭代器

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

表 10.2：插入迭代器操作

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

402 插入器有三种类型，差异在于元素插入的位置：

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



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

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

```
*it = val;
```

其效果与下面代码一样

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

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

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

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

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

10.4.1 节练习

403

练习 10.26：解释三种插入迭代器的不同之处。

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

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

10.4.2 iostream 迭代器



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

istream_iterator 操作

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

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

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

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

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

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

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

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

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

表 10.3：istream_iterator 操作

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

使用算法操作流迭代器

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

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

此调用会计算出从标准输入读取的值的和。如果输入为：

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

则输出为 664。

405 istream_iterator 允许使用懒惰求值

当我们将一个 istream_iterator 绑定到一个流时，标准库并不保证迭代器立即从流读取数据。具体实现可以推迟从流中读取数据，直到我们使用迭代器时才真正读取。标

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

ostream_iterator 操作

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

表 10.4：ostream_iterator 操作

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

我们可以用 ostream_iterator 来输出值的序列：

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

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

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

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

运算符*和++实际上对 ostream_iterator 对象不做任何事情，因此忽略它们对我们的程序没有任何影响。但是，推荐第一种形式。在这种写法中，流迭代器的使用与其他迭代器的使用保持一致。如果想将此循环改为操作其他迭代器类型，修改起来非常容易。而且，对于读者来说，此循环的行为也更为清晰。 406

可以通过调用 copy 来打印 vec 中的元素，这比编写循环更为简单：

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

使用流迭代器处理类类型

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

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

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

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

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

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

407

10.4.2 节练习

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

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

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

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

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

10.4.3 反向迭代器

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

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

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

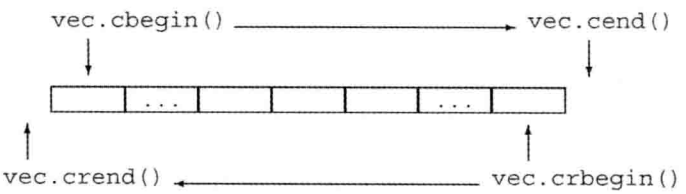


图 10.1：比较 cbegin/cend 和 crbegin/crend

下面的循环是一个使用反向迭代器的例子，它按逆序打印 vec 中的元素：

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

408

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

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

反向迭代器需要递减运算符

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

反向迭代器和其他迭代器间的关系



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

line 中的第一个单词。使用 find 可以很容易地完成这一任务：

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

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

如果希望打印最后一个单词，可以改用反向迭代器：

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

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

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

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

409 但它会生成错误的输出结果。例如，如果我们的输入是

```
FIRST,MIDDLE,LAST
```

则这条语句会打印 TSAL!

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

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

给定和之前一样的输入，这条语句会如我们的预期打印出 LAST。

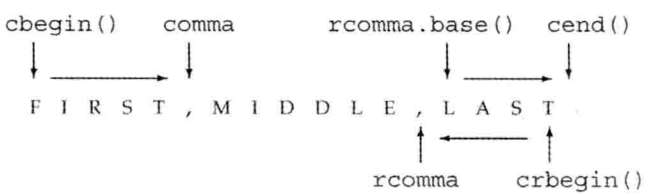


图 10.2：反向迭代器和普通迭代器间的关系

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

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



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

10.4.3 节练习

410

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

练习 10.35：使用普通迭代器逆序打印一个 vector。

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

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

10.5 泛型算法结构



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

表 10.5：迭代器类别

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

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

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

10.5.1 5 类迭代器



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

迭代器是按它们所提供的操作来分类的，而这种分类形成了一种层次。除了输出迭代

器之外，一个高层类别的迭代器支持低层类别迭代器的所有操作。

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



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

迭代器类别

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

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

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

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

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

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

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

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

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

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

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

10.5.1 节练习

练习 10.38：列出 5 个迭代器类别，以及每类迭代器所支持的操作。

练习 10.39：list 上的迭代器属于哪类？vector 呢？

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

10.5.2 算法形参模式



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

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

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

413

接受单个目标迭代器的算法

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



向输出迭代器写入数据的算法都假定目标空间足够容纳写入的数据。

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

接受第二个输入序列的算法

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

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

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



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

10.5.3 算法命名规范

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

一些算法使用重载形式传递一个谓词

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

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

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

_if 版本的算法

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

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

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

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

区分拷贝元素的版本和不拷贝的版本

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

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

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

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

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

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

10.5.3 节练习

415

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

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

10.6 特定容器算法

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

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

Best Practices

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

表 10.6：list 和 forward_list 成员函数版本的算法

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

续表

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

splice 成员

416

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

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

`lst.splice(`*args*`)`或 `flst.splice_after(`*args*`)`

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

链表特有的操作会改变容器

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

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

10.6 节练习

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

小结

417

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

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

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

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

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

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

术语表

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

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

二元谓词（binary predicate） 接受两个参数的谓词。

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

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

捕获列表（capture list） lambda 表达式的一部分，指出 lambda 表达式可以访问所在上下文中哪些变量。

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

前向迭代器（forward iterator） 可以读写元素，但不必支持--的迭代器。 418

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

泛型算法（generic algorithm） 类型无关的算法。

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

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

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

istream_iterator 读取输入流的流迭代器。

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

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

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

ostream_iterator 写输出流的迭代器。

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

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

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

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

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

流迭代器（stream iterator） 可以绑定到一个流的迭代器。

一元谓词（unary predicate） 接受一个参数的谓词。

第 11 章
关联容器

内容

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

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

虽然关联容器的很多行为与顺序容器相同，但其不同之处反映了关键字的作用。

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

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

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

表 11.1：关联容器类型

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

11.1 使用关联容器

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

map 是关键字-值对的集合。例如，可以将一个人的名字作为关键字，将其电话号码作为值。我们称这样的数据结构为“将名字映射到电话号码”。map 类型通常被称为**关联数组**（associative array）。关联数组与“正常”数组类似，不同之处在于其下标不必是整数。
421 我们通过一个关键字而不是位置来查找值。给定一个名字到电话号码的 map，我们可以使用一个人的名字作为下标来获取此人的电话号码。

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