

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

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

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

续表

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

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：

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

< 314

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`。

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

<code>fstream fstrm;</code>	创建一个未绑定的文件流。 <code>fstream</code> 是头文件 <code>fstream</code> 中定义的一个类型
<code>fstream fstrm(s);</code>	创建一个 <code>fstream</code> ，并打开名为 <code>s</code> 的文件。 <code>s</code> 可以是 <code>string</code> 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 <code>explicit</code> 的（参见 7.5.4 节，第 265 页）。默认的文件模式 <code>mode</code> 依赖于 <code>fstream</code> 的类型
<code>fstream fstrm(s, mode);</code>	与前一个构造函数类似，但按指定 <code>mode</code> 打开文件
<code>fstrm.open(s)</code>	打开名为 <code>s</code> 的文件，并将文件与 <code>fstrm</code> 绑定。 <code>s</code> 可以是一个 <code>string</code> 或一个指向 C 风格字符串的指针。默认的文件 <code>mode</code> 依赖于 <code>fstream</code> 的类型。返回 <code>void</code>
<code>fstrm.close()</code>	关闭与 <code>fstrm</code> 绑定的文件。返回 <code>void</code>
<code>fstrm.is_open()</code>	返回一个 <code>bool</code> 值，指出与 <code>fstrm</code> 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
        }
    }
    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

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

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 << " "
        << formatted.str() << endl; // 打印名字 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

<code>vector</code>	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
<code>deque</code>	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
<code>list</code>	双向链表。只支持双向顺序访问。在 <code>list</code> 中任何位置进行插入/删除操作速度都很快
<code>forward_list</code>	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
<code>array</code>	固定大小数组。支持快速随机访问。不能添加或删除元素
<code>string</code>	与 <code>vector</code> 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。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
 - 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。

328

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。

Best Practices

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。

表 9.2: 容器操作

330

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(init)	使用 init 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

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）

[\[333\]](#)

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。

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();
```

334

C++
11

```
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)	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 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。范围内元素的类型必须与 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



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：

```
array<int, 10> ial;           // 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

<code>c1=c2</code>	将 <code>c1</code> 中的元素替换为 <code>c2</code> 中元素的拷贝。 <code>c1</code> 和 <code>c2</code> 必须具有相同的类型
<code>c={a,b,c...}</code>	将 <code>c1</code> 中元素替换为初始化列表中元素的拷贝 (<code>array</code> 不适用)
<code>swap(c1,c2)</code>	交换 <code>c1</code> 和 <code>c2</code> 中的元素。 <code>c1</code> 和 <code>c2</code> 必须具有相同的类型。 <code>swap</code> 通常比从 <code>c2</code> 向 <code>c1</code> 拷贝元素快得多
<code>assign</code> 操作不适用于关联容器和 <code>array</code>	
<code>seq.assign(b,e)</code>	将 <code>seq</code> 中的元素替换为迭代器 <code>b</code> 和 <code>e</code> 所表示的范围中的元素。迭代器 <code>b</code> 和 <code>e</code> 不能指向 <code>seq</code> 中的元素
<code>seq.assign(il)</code>	将 <code>seq</code> 中的元素替换为初始化列表 <code>il</code> 中的元素
<code>seq.assign(n,t)</code>	将 <code>seq</code> 中的元素替换为 <code>n</code> 个值为 <code>t</code> 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。339 `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。



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。

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)` 在 `c` 的尾部创建一个值为 `t` 或由 `args` 创建的元素。返回 `void`
`c.emplace_back(args)`

`c.push_front(t)` 在 `c` 的头部创建一个值为 `t` 或由 `args` 创建的元素。返回 `void`
`c.emplace_front(args)`

`c.insert(p, t)` 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.emplace(p, args)`

`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。

342

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。

343

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

```
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。

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：

```
// 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, "999-99999999"); // 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。

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 <code>string</code> 、 <code>vector</code> 、 <code>deque</code> 和 <code>array</code> 。 <code>back</code> 不适用于 <code>forward_list</code> 。
<code>c.back()</code> 返回 <code>c</code> 中尾元素的引用。若 <code>c</code> 为空，函数行为未定义
<code>c.front()</code> 返回 <code>c</code> 中首元素的引用。若 <code>c</code> 为空，函数行为未定义
<code>c[n]</code> 返回 <code>c</code> 中下标为 <code>n</code> 的元素的引用， <code>n</code> 是一个无符号整数。若 <code>n >= c.size()</code> ，则函数行为未定义
<code>c.at(n)</code> 返回下标为 <code>n</code> 的元素的引用。如果下标越界，则抛出一 <code>out_of_range</code> 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。

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`



WARNING 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。



WARNING 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？

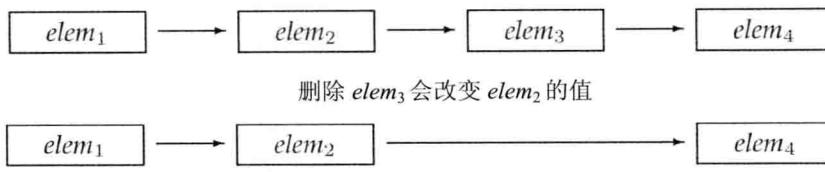
练习 9.26：使用下面代码定义的 `ia`，将 `ia` 拷贝到一个 `vector` 和一个 `list` 中。使用单迭代器版本的 `erase` 从 `list` 中删除奇数元素，从 `vector` 中删除偶数元素。

```
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 `elem3` 会改变 `elem2`，`elem2` 原来指向 `elem3`，但删除 `elem3` 后，`elem2` 指向了 `elem4`。

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 `elem3`，应该用指向 `elem2` 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

<code>lst.before_begin()</code>	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。 <code>cbefore_begin()</code> 返回一个 <code>const_iterator</code>
<code>lst.insert_after(p, t)</code>	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 <code>lst</code> 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
<code>lst.insert_after(p, b, e)</code>	
<code>lst.insert_after(p, il)</code>	
<code>emplace_after(p, args)</code>	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
<code>lst.erase_after(p)</code>	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 <code>lst</code> 的尾元素或者是一个尾后迭代器，则函数行为未定义
<code>lst.erase_after(b, e)</code>	

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()) {
    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。

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。



WARNING

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

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

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

<code>s.substr(pos, n)</code>	返回一个 <code>string</code> ，包含 <code>s</code> 中从 <code>pos</code> 开始的 <code>n</code> 个字符的拷贝。 <code>pos</code> 的默认值为 0。 <code>n</code> 的默认值为 <code>s.size() - pos</code> ，即拷贝从 <code>pos</code> 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，我们可以将由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

<code>s.insert(pos,args)</code>	在 pos 之前插入 args 指定的字符。pos 可以是一个下标或一个迭代器。接受下标的版本返回一个指向 s 的引用; 接受迭代器的版本返回指向第一个插入字符的迭代器
<code>s.erase(pos,len)</code>	删除从位置 pos 开始的 len 个字符。如果 len 被省略, 则删除从 pos 开始直至 s 末尾的所有字符。返回一个指向 s 的引用
<code>s.assign(args)</code>	将 s 中的字符替换为 args 指定的字符。返回一个指向 s 的引用
<code>s.append(args)</code>	将 args 追加到 s。返回一个指向 s 的引用
<code>s.replace(range,args)</code>	删除 s 中范围 range 内的字符, 替换为 args 指定的字符。range 或者是一个下标和一个长度, 或者是一对指向 s 的迭代器。返回一个指向 s 的引用
<i>args</i> 可以是下列形式之一; append 和 assign 可以使用所有形式。	
str 不能与 s 相同, 迭代器 b 和 e 不能指向 s。	
<code>str</code>	字符串 str
<code>str, pos, len</code>	str 中从 pos 开始最多 len 个字
<code>cp, len</code>	从 cp 指向的字符数组的前(最多) len 个字符
<code>cp</code>	cp 指向的以空字符结尾的字符数组
<code>n, c</code>	n 个字符 c
<code>b, e</code>	迭代器 b 和 e 指定的范围内的字符
初始化列表	花括号包围的, 以逗号分隔的字符列表

续表

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

改变 string 的多种重载函数

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

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

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

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

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

9.5.2 节练习

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

练习 9.44：重写上一题的函数，这次使用一个下标和 replace。

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

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



9.5.3 string 搜索操作

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



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

< 365

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

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

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

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

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

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

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

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

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

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

表 9.14: string 搜索操作

搜索操作返回指定字符出现的下标，如果未找到则返回 `npos`。

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

续表

args 必须是以下形式之一

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

指定在哪里开始搜索

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

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

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

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

逆向搜索

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

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

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

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

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

每个操作都接受一个可选的第二参数，可用来指出从什么位置开始搜索。

9.5.3 节练习

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

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

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

9.5.4 compare 函数



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

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

< 367

表 9.15: `s.compare` 的几种参数形式

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

9.5.5 数值转换



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

新标准引入了多个函数，可以实现数值数据与标准库 `string` 之间的转换：

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

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

9.5.5 节练习

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

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

9.6 容器适配器



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

<369

表 9.17：所有容器适配器都支持的操作和类型

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)	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
a.swap(b)	

定义一个适配器

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

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

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

<370

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

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

栈适配器

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

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

其中，声明语句

```
stack<int> intStack; // 空栈
```

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

表 9.18: 表 9.17 未列出的栈操作

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

371 >

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

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

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

队列适配器

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

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

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

续表

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

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

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

9.6 节练习

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

372 小结

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

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

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

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

术语表

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

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

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

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

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

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

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

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

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

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

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

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

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

首前迭代器 (off-the-beginning iterator) 表示一个 `forward_list` 开始位置之前

（不存在的）元素的迭代器。是 `forward_list` 的成员函数 `before_begin` 的返回值。与 `end()` 迭代器类似，不能被解引用。

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

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

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

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

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

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

第 10 章

泛型算法

内容

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

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

泛型算法和关于迭代器的更多细节，构成了本章的主要内容。

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

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

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

10.1 概述

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

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

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

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

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

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

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

377

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

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

还可以在序列的子范围中查找，只需将指向子范围首元素和尾元素之后位置的迭代器

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

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

算法如何工作

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

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

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

迭代器令算法不依赖于容器, ……

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

……, 但算法依赖于元素类型的操作

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

< 378

10.1 节练习

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

练习 10.2: 重做上一题, 但读取 `string` 序列存入 `list` 中。

关键概念: 算法永远不会执行容器的操作

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

的值，也可能在容器内移动元素，但永远不会直接添加或删除元素。

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



10.2 初识泛型算法

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

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

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



10.2.1 只读算法

379

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

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

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

这条语句将 `sum` 设置为 `vec` 中元素的和，和的初值被设置为 0。



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

算法和元素类型

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

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

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

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

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

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



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

操作两个序列的算法

< 380

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

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

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

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



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

10.2.1 节练习

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

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

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

10.2.2 写容器元素的算法



一些算法将新值赋予序列中的元素。当我们使用这类算法时，必须注意确保序列原大

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

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

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

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

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

381

关键概念：迭代器参数

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

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

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



算法不检查写操作

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

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

函数 `fill_n` 假定写入指定个元素是安全的。即，如下形式的调用

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

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

382

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

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

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



向目的位置迭代器写入数据的算法假定目的位置足够大，能容纳要写入的元素。

介绍 `back_inserter`

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

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

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

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

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

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

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

拷贝算法

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

我们可以用 `copy` 实现内置数组的拷贝，如下面代码所示：

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

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

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

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

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

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

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

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

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

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

10.2.2 节练习

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

练习 10.7： 下面程序是否有错误？如果有，请改正。

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



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

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



10.2.3 重排容器元素的算法

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

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

为了便于说明问题，我们将使用下面简单的故事作为输入：

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

给定此输入，我们的程序应该生成如下 `vector`:

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

消除重复单词

< 384

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

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

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

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

注意，单词 `red` 和 `the` 各出现了两次。

使用 `unique`

< 385

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

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

↑
end_unique
(最后一个不重复元素之后的位置)

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



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

使用容器操作删除元素

< 386

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

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

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

10.2.3 节练习

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

练习 10.10: 你认为算法不改变容器大小的原因是什么?

10.3 定制操作

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

386

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



10.3.1 向算法传递函数

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

谓词

谓词是一个可调用的表达式, 其返回结果是一个能用作条件的值。标准库算法所使用的谓词分为两类: 一元谓词(`unary predicate`, 意味着它们只接受单一参数)和二元谓词(`binary predicate`, 意味着它们有两个参数)。接受谓词参数的算法对输入序列中的元素调用谓词。因此, 元素类型必须能转换为谓词的参数类型。

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

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

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

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

排序算法

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

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

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

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

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

10.3.1 节练习

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

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

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

10.3.2 lambda 表达式

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

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

388

我们将此函数命名为 `biggies`, 其框架如下所示:

```
void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    elimDups(words); // 将 words 按字典序排序, 删除重复单词
    // 按长度排序, 长度相同的单词维持字典序
    stable_sort(words.begin(), words.end(), isShorter);
    // 获得一个迭代器, 指向第一个满足 size()>= sz 的元素
    // 计算满足 size >= sz 的元素的数目
    // 打印长度大于等于给定值的单词, 每个单词后面接一个空格
}
```

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

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

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

介绍 lambda

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

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

C++ 11

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

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

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

我们可以忽略参数列表和返回类型, 但必须永远包含捕获列表和函数体

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

389

此例中，我们定义了一个可调用对象 `f`，它不接受参数，返回 42。

`lambda` 的调用方式与普通函数的调用方式相同，都是使用调用运算符：

```
cout << f() << endl; // 打印 42
```

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



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

向 `lambda` 传递参数

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

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

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

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

如下所示，可以使用此 `lambda` 来调用 `stable_sort`：

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

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

使用捕获列表

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

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

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

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

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

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

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



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

调用 `find_if`

使用此 `lambda`，我们就可以查找第一个长度大于等于 `sz` 的元素：

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

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

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

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

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

391> `for_each` 算法

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

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

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

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



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

完整的 biggies

到目前为止，我们已经解决了程序的所有细节，下面就是完整的程序：

```
void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    elimDups(words); // 将 words 按字典序排序，删除重复单词
    // 按长度排序，长度相同的单词维持字典序
    stable_sort(words.begin(), words.end(),
                 [] (const string &a, const string &b)
                     { return a.size() < b.size(); });
    // 获得一个迭代器，指向第一个满足 size()>= sz 的元素
    auto wc = find_if(words.begin(), words.end(),
                       [sz] (const string &a)
                           { return a.size() >= sz; });
    // 计算满足 size >= sz 的元素的数目
    auto count = words.end() - wc;
    cout << count << " " << make_plural(count, "word", "s")
        << " of length " << sz << " or longer" << endl;
    // 打印长度大于等于给定值的单词，每个单词后面接一个空格
    for_each(wc, words.end(),
             [] (const string &s){cout << s << " "});
    cout << endl;
}
```

10.3.2 节练习

392

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

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

练习 10.16: 使用 lambda 编写你自己版本的 biggies。

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

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

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

10.3.3 lambda 捕获和返回

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

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

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

值捕获

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

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

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

393 引用捕获

我们定义 `lambda` 时可以采用引用方式捕获变量。例如：

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

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

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

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

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

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

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

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

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



WARNING

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

< 394

建议：尽量保持 `lambda` 的变量捕获简单化

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

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

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

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

隐式捕获

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

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

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

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

```

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

```

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

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

表 10.1: lambda 捕获列表

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

可变 lambda

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

```

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

```

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

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

< 396

指定 lambda 返回类型

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

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

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

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

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

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

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

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

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

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

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

C++
11

397

10.3.3 节练习

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

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



10.3.4 参数绑定

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

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

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

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

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

标准库 `bind` 函数

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

398

调用 `bind` 的一般形式为：

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

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

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

C++ 11

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

绑定 *check_size* 的 *sz* 参数

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

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

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

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

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

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

替换为如下使用 *check_size* 的版本：

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

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

使用 *placeholders* 名字

399

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

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

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

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

```
using namespace namespace_name;
```

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

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

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

bind 的参数

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

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

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

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

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

映射为

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

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

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

400> 用 bind 重排参数顺序

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

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

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

绑定引用参数

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

例如，为了替换一个引用方式捕获 ostream 的 lambda：

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

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

可以很容易地编写一个函数，完成相同的工作：

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

但是，不能直接用 bind 来代替对 os 的捕获：

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

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

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

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

向后兼容：参数绑定

401

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

10.3.4 节练习

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

练习 10.23：bind 接受几个参数？

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

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

10.4 再探迭代器

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

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

关联的 IO 流。

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



10.4.1 插入迭代器

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

表 10.2: 插入迭代器操作

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

402

插入器有三种类型，差异在于元素插入的位置：

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



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

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

```
*it = val;
```

其效果与下面代码一样

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

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

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

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

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

10.4.1 节练习

403

练习 10.26：解释三种插入迭代器的不同之处。

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

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

10.4.2 iostream 迭代器



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

istream_iterator 操作

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

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

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

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

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

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

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

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

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

表 10.3: `istream_iterator` 操作

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

使用算法操作流迭代器

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

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

此调用会计算出从标准输入读取的值的和。如果输入为：

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

则输出为 664。

405 > `istream_iterator` 允许使用懒惰求值

当我们将一个 `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。对于大多数组程序来说，立即读取还是推迟读取没什么差别。但是，如果我们创建了一个 `istream_iterator`，没有使用就销毁了，或者我们正在从两个不同的对象同步读取同一个流，那么何时读取可能就很重要了。

`ostream_iterator` 操作

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

表 10.4: `ostream_iterator` 操作

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

我们可以用 `ostream_iterator` 来输出值的序列：

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

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

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

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

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

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

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

406

使用流迭代器处理类类型

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

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

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

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

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

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

407

10.4.2 节练习

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

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

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

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

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

10.4.3 反向迭代器

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

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

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

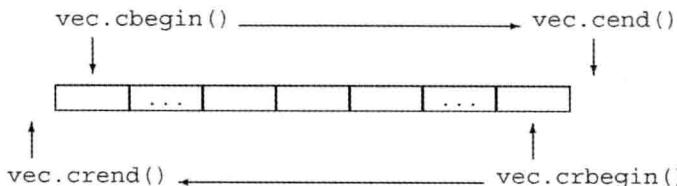


图 10.1：比较 `cbegin/cend` 和 `crbegin/crend`

下面的循环是一个使用反向迭代器的例子，它按逆序打印 `vec` 中的元素：

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

408

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

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

反向迭代器需要递减运算符

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

反向迭代器和其他迭代器间的关系

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



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

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

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

如果希望打印最后一个单词，可以改用反向迭代器：

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

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

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

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

409> 但它会生成错误的输出结果。例如，如果我们的输入是

FIRST,MIDDLE,LAST

则这条语句会打印 `TSAL!`

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

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

给定和之前一样的输入，这条语句会如我们的预期打印出 `LAST`。

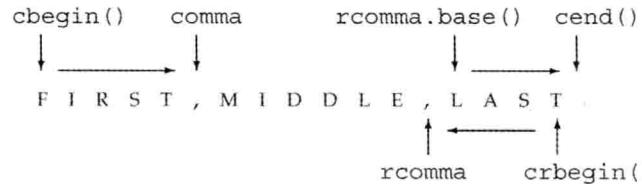


图 10.2：反向迭代器和普通迭代器间的关系

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

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



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

10.4.3 节练习

410

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

练习 10.35： 使用普通迭代器逆序打印一个 vector。

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

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

10.5 泛型算法结构



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

表 10.5：迭代器类别

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

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

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

10.5.1 5 类迭代器



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

迭代器是按它们所提供的操作来分类的，而这种分类形成了一种层次。除了输出迭代

器之外，一个高层类别的迭代器支持低层类别迭代器的所有操作。

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



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

迭代器类别

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

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

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

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

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

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

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

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

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

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

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

10.5.1 节练习

练习 10.38：列出 5 个迭代器类别，以及每类迭代器所支持的操作。

练习 10.39：`list` 上的迭代器属于哪类？`vector` 呢？

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

10.5.2 算法形参模式



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

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

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

413

接受单个目标迭代器的算法

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



WARNING

向输出迭代器写入数据的算法都假定目标空间足够容纳写入的数据。

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

接受第二个输入序列的算法

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

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

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



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



10.5.3 算法命名规范

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

一些算法使用重载形式传递一个谓词

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

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

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

_if 版本的算法

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

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

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

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

区分拷贝元素的版本和不拷贝的版本

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

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

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

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

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

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

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

10.5.3 节练习

415

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

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

10.6 特定容器算法

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

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



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

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

这些操作都返回 `void`

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

续表

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

📚 splice 成员

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

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

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

链表特有的操作会改变容器

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

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

10.6 节练习

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

小结

< 417

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

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

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

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

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

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

术语表

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

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

二元谓词（binary predicate） 接受两个参数的谓词。

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

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

捕获列表（capture list） `lambda` 表达式的

一部分，指出 `lambda` 表达式可以访问所在上下文中哪些变量。

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

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

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

泛型算法（generic algorithm） 类型无关的算法。

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

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

< 418

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

istream_iterator 读取输入流的流迭代器。

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

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

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

ostream_iterator 写输出流的迭代器。

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

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

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

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

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

流迭代器 (stream iterator) 可以绑定到一个流的迭代器。

一元谓词 (unary predicate) 接受一个参数的谓词。

第 11 章

关联容器

内容

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

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

虽然关联容器的很多行为与顺序容器相同，但其不同之处反映了关键字的作用。

420

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

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

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

表 11.1：关联容器类型

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



11.1 使用关联容器

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

map 是关键字-值对的集合。例如，可以将一个人的名字作为关键字，将其电话号码作为值。我们称这样的数据结构为“将名字映射到电话号码”。**map** 类型通常被称为**关联数组**（associative array）。关联数组与“正常”数组类似，不同之处在于其下标不必是整数。421 我们通过一个关键字而不是位置来查找值。给定一个名字到电话号码的 **map**，我们可以使用一个人的名字作为下标来获取此人的电话号码。

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

使用 map

一个经典的使用关联数组的例子是单词计数程序：

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

此程序读取输入，报告每个单词出现多少次。

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

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

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

如果我们对本节第一段中的文本（指英文版中的文本）运行这个程序，输出将会是：

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

使用 set

422

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

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

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

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

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

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

如果用此程序处理与之前相同的输入，输出将会是：

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

11.1 节练习

练习 11.1：描述 `map` 和 `vector` 的不同。

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

练习 11.3：编写你自己的单词计数程序。

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

423 >

11.2 关联容器概述

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

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

关联容器的迭代器都是双向的（参见 10.5.1 节，第 365 页）。



11.2.1 定义关联容器

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

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

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

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

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

```
{key, value}
```

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

初始化 `multimap` 或 `multiset`

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

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

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

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

11.2.1 节练习

练习 11.5：解释 map 和 set 的区别。你如何选择使用哪个？

练习 11.6：解释 set 和 list 的区别。你如何选择使用哪个？

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

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



11.2.2 关键字类型的要求

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



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

有序容器的关键字类型

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

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

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



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

使用关键字类型的比较函数

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

义哪种类型的容器，自定义的操作类型必须在尖括号中紧跟着元素类型给出。

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

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

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

426

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

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

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

11.2.2 节练习

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

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

练习 11.11： 不使用 decltype 重新定义 bookstore。

11.2.3 pair 类型

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

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

```

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

```

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

我们也可以为每个成员提供初始化器：

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

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

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

```

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

```

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

表 11.2: pair 上的操作

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

创建 pair 对象的函数

C++ 11 想象有一个函数需要返回一个 pair。在新标准下，我们可以对返回值进行列表初始化（参见 6.3.2 节，第 203 页）

428

```

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

```

```

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

```

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

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

```

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

```

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

```

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

```

11.2.3 节练习

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

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

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

11.3 关联容器操作

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

表 11.3：关联容器额外的类型别名

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

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

```

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

```

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

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

11.3.1 关联容器迭代器

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

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



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

set 的迭代器是 `const` 的

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

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

430 遍历关联容器

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

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

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

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



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

关联容器和算法

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

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

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

11.3.1 节练习

< 431

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

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

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

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

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

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

11.3.2 添加元素

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

存在的元素对容器没有任何影响：

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

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

向 map 添加元素

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

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

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

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

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

表 11.4：关联容器 `insert` 操作

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

检测 `insert` 的返回值

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

素被插入容器中，且 bool 值为 true。

作为一个例子，我们用 insert 重写单词计数程序：

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

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

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

展开递增语句

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

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

下面我们一步一步来解释此表达式：

ret 保存 insert 返回的值，是一个 pair。

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

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

ret.first->second map 中元素的值部分。

++ret.first->second 递增此值。

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

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

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

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

向 multiset 或 multimap 添加元素

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

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

434 >

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

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

11.3.2 节练习

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

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

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

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

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

11.3.3 删除元素

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

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

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

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

435 > 对允许重复关键字的容器，删除元素的数量可能大于 1：

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

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

表 11.5：从关联容器删除元素

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

11.3.4 map 的下标操作



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

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

例如，如果我们编写如下代码

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

将会执行如下操作：

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

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

436



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

表 11.6：`map` 和 `unordered_map` 的下标操作

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

使用下标操作的返回值

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

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

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

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



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

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

11.3.4 节练习

练习 11.24：下面的程序完成什么功能？

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

练习 11.25：对比下面程序与上一题程序

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

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

11.3.5 访问元素

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

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

表 11.7：在一个关联容器中查找元素的操作

lower_bound 和 upper_bound 不适用于无序容器。

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

续表

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

对 map 使用 find 代替下标操作

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

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

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

在 multimap 或 multiset 中查找元素

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

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

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

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



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

一种不同的，面向迭代器的解决方法

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

439 >

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



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

使用这两个操作，我们可以重写前面的程序：

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

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

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

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

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



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

equal_range 函数

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

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

可以用 equal_range 来再次修改我们的程序：

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

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

11.3.5 节练习

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

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

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

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

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

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

11.3.6 一个单词转换的 map

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

如果单词转换文件的内容如下所示：

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

```

u you
pic picture
thk thanks!
18r later

```

我们希望转换的文本为

```

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

```

则程序应该生成这样的输出：

```

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

```

单词转换程序

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

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

```

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

```

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

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

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

建立转换映射

函数 `buildMap` 读入给定文件, 建立起转换映射。

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

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

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

生成转换文本

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

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

< 443

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

用来替代 s 的内容。

11.3.6 节练习

练习 11.33: 实现你自己版本的单词转换程序。

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

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

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

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



11.4 无序容器

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

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

444



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

使用无序容器

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

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

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

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

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

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

对于每个单词，我们将得到相同的计数结果。但单词不太可能按字典序输出。

管理桶

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

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

<445

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

表 11.8：无序容器管理操作

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

无序容器对关键字类型的要求

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

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

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

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

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

我们使用这些函数来定义一个`unordered_multiset`

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

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

如果我们的类定义了`==`运算符，则可以只重载哈希函数：

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

11.4 节练习

练习 11.37：一个无序容器与其有序版本相比有何优势？有序版本有何优势？

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

小结

447

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

标准库定义了 8 个关联容器，每个容器

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

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

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

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

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

术语表

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

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

hash 特殊的标准库模板，无序容器用它来管理元素的位置。

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

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

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

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

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

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

448

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

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

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

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

unordered_map 保存关键字-值对的容器，不允许重复关键字。

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

unordered_multiset 保存关键字的容器，

允许重复关键字。

unordered_set 保存关键字的容器，不允许重复关键字。

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

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

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

第 12 章

动态内存

内容

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

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

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

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

450

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

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



虽然使用动态内存有时是必要的，但众所周知，正确地管理动态内存是非常棘手的。

12.1 动态内存与智能指针

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

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

C++ 11

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



12.1.1 shared_ptr 类

C++ 11

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

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

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

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

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

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

操作列于表 12.2 中。

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

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

表 12.2: `shared_ptr` 独有的操作

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

make_shared 函数

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

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

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

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

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

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

shared_ptr 的拷贝和赋值

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

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

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

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

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

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



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

shared_ptr 自动销毁所管理的对象……

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

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

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

……shared_ptr 还会自动释放相关联的内存

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

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

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

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

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

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

但如果其他 `shared_ptr` 也指向这块内存，它就不会被释放掉：

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

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

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



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

使用了动态生存期的资源的类

程序使用动态内存出于以下三种原因之一：

1. 程序不知道自己需要使用多少对象
2. 程序不知道所需对象的准确类型

3. 程序需要在多个对象间共享数据

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

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

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

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

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

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

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

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



使用动态内存的一个常见原因是允许多个对象共享相同的状态。

定义 `StrBlob` 类

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

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

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

我们将 `vector` 保存在动态内存中。

为了实现我们所希望的数据共享，我们为每个 `StrBlob` 设置一个 `shared_ptr` 来管理动态分配的 `vector`。此 `shared_ptr` 的成员将记录有多少个 `StrBlob` 共享相同的 `vector`，并在 `vector` 的最后一个使用者被销毁时释放 `vector`。456

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

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

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

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

StrBlob 构造函数

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

```
StrBlob::StrBlob(): data(make_shared<vector<string>>()) {}
StrBlob::StrBlob(initializer_list<string> il):
    data(make_shared<vector<string>>(il)) {}
```

接受一个 `initializer_list` 的构造函数将其参数传递给对应的 `vector` 构造函数（参见 2.2.1 节，第 39 页）。此构造函数通过拷贝列表中的值来初始化 `vector` 的元素。

元素访问成员函数

`pop_back`、`front` 和 `back` 操作访问 `vector` 中的元素。这些操作在试图访问元素之前必须检查元素是否存在。由于这些成员函数需要做相同的检查操作，我们为 `StrBlob` 定义了一个名为 `check` 的 `private` 工具函数，它检查一个给定索引是否在合法范围内。457

除了索引，`check` 还接受一个 `string` 参数，它会将此参数传递给异常处理程序，这个 `string` 描述了错误内容：

```
void StrBlob::check(size_type i, const string &msg) const
{
    if (i >= data->size())
        throw out_of_range(msg);
}
```

`pop_back` 和元素访问成员函数首先调用 `check`。如果 `check` 成功，这些成员函数继续利用底层 `vector` 的操作来完成自己的工作：

```
string& StrBlob::front()
{
    // 如果 vector 为空，check 会抛出一个异常
    check(0, "front on empty StrBlob");
    return data->front();
}

string& StrBlob::back()
{
    check(0, "back on empty StrBlob");
    return data->back();
}

void StrBlob::pop_back()
{
    check(0, "pop_back on empty StrBlob");
    data->pop_back();
}
```

`front` 和 `back` 应该对 `const` 进行重载（参见 7.3.2 节，第 247 页），这些版本的定义留作练习。

StrBlob 的拷贝、赋值和销毁

类似 `Sales_data` 类，`StrBlob` 使用默认版本的拷贝、赋值和销毁成员函数来对此类型的对象进行这些操作（参见 7.1.5 节，第 239 页）。默认情况下，这些操作拷贝、赋值和销毁类的数据成员。我们的 `StrBlob` 类只有一个数据成员，它是 `shared_ptr` 类型。因此，当我们拷贝、赋值或销毁一个 `StrBlob` 对象时，它的 `shared_ptr` 成员会被拷贝、赋值或销毁。

如前所见，拷贝一个 `shared_ptr` 会递增其引用计数；将一个 `shared_ptr` 赋予另一个 `shared_ptr` 会递增赋值号右侧 `shared_ptr` 的引用计数，而递减左侧 `shared_ptr` 的引用计数。如果一个 `shared_ptr` 的引用计数变为 0，它所指向的对象会被自动销毁。因此，对于由 `StrBlob` 构造函数分配的 `vector`，当最后一个指向它的 `StrBlob` 对象被销毁时，它会随之被自动销毁。

458

12.1.1 节练习

练习 12.1：在此代码的结尾，`b1` 和 `b2` 各包含多少个元素？

```
StrBlob b1;
{
    StrBlob b2 = {"a", "an", "the"};
```

```

    b1 = b2;
    b2.push_back("about");
}

```

练习 12.2: 编写你自己的 StrBlob 类，包含 const 版本的 front 和 back。

练习 12.3: StrBlob 需要 const 版本的 push_back 和 pop_back 吗？如果需要，添加进去。否则，解释为什么不需要。

练习 12.4: 在我们的 check 函数中，没有检查 i 是否大于 0。为什么可以忽略这个检查？

练习 12.5: 我们未编写接受一个 initializer_list explicit（参见 7.5.4 节，第 264 页）参数的构造函数。讨论这个设计策略的优点和缺点。

12.1.2 直接管理内存

C++语言定义了两个运算符来分配和释放动态内存。运算符 new 分配内存，delete 释放 new 分配的内存。

相对于智能指针，使用这两个运算符管理内存非常容易出错，随着我们逐步详细介绍这两个运算符，这一点会更为清楚。而且，自己直接管理内存的类与使用智能指针的类不同，它们不能依赖类对象拷贝、赋值和销毁操作的任何默认定义（参见 7.1.4 节，第 237 页）。因此，使用智能指针的程序更容易编写和调试。



在学习第 13 章之前，除非使用智能指针来管理内存，否则不要分配动态内存。

使用 new 动态分配和初始化对象

在自由空间分配的内存是无名的，因此 new 无法为其分配的对象命名，而是返回一个指向该对象的指针：

```
int *pi = new int; // pi 指向一个动态分配的、未初始化的无名对象
```

此 new 表达式在自由空间构造一个 int 型对象，并返回指向该对象的指针。

默认情况下，动态分配的对象是默认初始化的（参见 2.2.1 节，第 40 页），这意味着内置类型或组合类型的对象的值将是未定义的，而类类型对象将用默认构造函数进行初始化：

```
string *ps = new string; // 初始化为空 string
int *pi = new int; // pi 指向一个未初始化的 int
```

459

我们可以使用直接初始化方式（参见 3.2.1 节，第 76 页）来初始化一个动态分配的对象。我们可以使用传统的构造方式（使用圆括号），在新标准下，也可以使用列表初始化（使用花括号）：

```
int *pi = new int(1024); // pi 指向的对象的值为 1024
string *ps = new string(10, '9'); // *ps 为"9999999999"
// vector 有 10 个元素，值依次从 0 到 9

vector<int> *pv = new vector<int>{0,1,2,3,4,5,6,7,8,9};
```

C++
11

也可以对动态分配的对象进行值初始化（参见 3.3.1 节，第 88 页），只需在类型名之

后跟一对空括号即可：

```
string *ps1 = new string;      // 默认初始化为空 string
string *ps = new string();    // 值初始化为空 string
int *pi1 = new int;           // 默认初始化；*pi1 的值未定义
int *pi2 = new int();         // 值初始化为 0；*pi2 为 0
```

对于定义了自己的构造函数（参见 7.1.4 节，第 235 页）的类类型（例如 `string`）来说，要求值初始化是没有意义的；不管采用什么形式，对象都会通过默认构造函数来初始化。但对于内置类型，两种形式的差别就很大了：值初始化的内置类型对象有着良好定义的值，而默认初始化的对象的值则是未定义的。类似的，对于类中那些依赖于编译器合成的默认构造函数的内置类型成员，如果它们未在类内被初始化，那么它们的值也是未定义的（参见 7.1.4 节，第 236 页）。

Best Practices

出于与变量初始化相同的原因，对动态分配的对象进行初始化通常是个好主意。

C++ 11 如果我们提供了一个括号包围的初始化器，就可以使用 `auto`（参见 2.5.2 节，第 61 页）从此初始化器来推断我们想要分配的对象的类型。但是，由于编译器要用初始化器的类型来推断要分配的类型，只有当括号中仅有单一初始化器时才可以使用 `auto`：

```
auto p1 = new auto(obj);      // p 指向一个与 obj 类型相同的对象
                               // 该对象用 obj 进行初始化
auto p2 = new auto{a,b,c};    // 错误：括号中只能有单个初始化器
```

`p1` 的类型是一个指针，指向从 `obj` 自动推断出的类型。若 `obj` 是一个 `int`，那么 `p1` 就是 `int*`；若 `obj` 是一个 `string`，那么 `p1` 是一个 `string*`；依此类推。新分配的对象用 `obj` 的值进行初始化。

动态分配的 `const` 对象

用 `new` 分配 `const` 对象是合法的：

```
// 分配并初始化一个 const int
const int *pci = new const int(1024);
// 分配并默认初始化一个 const 的空 string
const string *pcs = new const string;
```

460 类似其他任何 `const` 对象，一个动态分配的 `const` 对象必须进行初始化。对于一个定义了默认构造函数（参见 7.1.4 节，第 236 页）的类类型，其 `const` 动态对象可以隐式初始化，而其他类型的对象就必须显式初始化。由于分配的对象是 `const` 的，`new` 返回的指针是一个指向 `const` 的指针（参见 2.4.2 节，第 56 页）。

内存耗尽

虽然现代计算机通常都配备大容量内存，但是自由空间被耗尽的情况还是有可能发生。一旦一个程序用光了它所有可用的内存，`new` 表达式就会失败。默认情况下，如果 `new` 不能分配所要求的内存空间，它会抛出一个类型为 `bad_alloc`（参见 5.6 节，第 173 页）的异常。我们可以改变使用 `new` 的方式来阻止它抛出异常：

```
// 如果分配失败，new 返回一个空指针
int *p1 = new int; // 如果分配失败，new 抛出 std::bad_alloc
int *p2 = new (nothrow) int; // 如果分配失败，new 返回一个空指针
```

我们称这种形式的 new 为定位 new (placement new)，其原因我们将在 19.1.2 节（第 729 页）中解释。定位 new 表达式允许我们向 new 传递额外的参数。在此例中，我们传递给它一个由标准库定义的名为 noexcept 的对象。如果将 noexcept 传递给 new，我们的意图是告诉它不能抛出异常。如果这种形式的 new 不能分配所需内存，它会返回一个空指针。bad_alloc 和 noexcept 都定义在头文件 new 中。

释放动态内存

为了防止内存耗尽，在动态内存使用完毕后，必须将其归还给系统。我们通过 delete 表达式（delete expression）来将动态内存归还给系统。delete 表达式接受一个指针，指向我们想要释放的对象：

```
delete p; // p 必须指向一个动态分配的对象或是一个空指针
```

与 new 类型类似，delete 表达式也执行两个动作：销毁给定的指针指向的对象；释放对应的内存。

指针值和 delete

我们传递给 delete 的指针必须指向动态分配的内存，或者是一个空指针（参见 2.3.2 节，第 48 页）。释放一块并非 new 分配的内存，或者将相同的指针值释放多次，其行为是未定义的：

```
int i, *pi1 = &i, *pi2 = nullptr;
double *pd = new double(33), *pd2 = pd;
delete i;      // 错误: i 不是一个指针
delete pi1;    // 未定义: pi1 指向一个局部变量
delete pd;     // 正确
delete pd2;    // 未定义: pd2 指向的内存已经被释放了
delete pi2;    // 正确: 释放一个空指针总是没有错误的
```

对于 delete i 的请求，编译器会生成一个错误信息，因为它知道 i 不是一个指针。执行 delete pi1 和 pd2 所产生的错误则更具潜在危害：通常情况下，编译器不能分辨一个指针指向的是静态还是动态分配的对象。类似的，编译器也不能分辨一个指针所指向的内存是否已经被释放了。对于这些 delete 表达式，大多数编译器会编译通过，尽管它们是错误的。◀ 461

虽然一个 const 对象的值不能被改变，但它本身是可以被销毁的。如同任何其他动态对象一样，想要释放一个 const 动态对象，只要 delete 指向它的指针即可：

```
const int *pci = new const int(1024);
delete pci; // 正确: 释放一个 const 对象
```

动态对象的生存期直到被释放时为止

如 12.1.1 节（第 402 页）所述，由 shared_ptr 管理的内存在线程最后一个 shared_ptr 销毁时会被自动释放。但对于通过内置指针类型来管理的内存，就不是这样了。对于一个由内置指针管理的动态对象，直到被显式释放之前它都是存在的。

返回指向动态内存的指针（而不是智能指针）的函数给其调用者增加了一个额外负担——调用者必须记得释放内存：

```
// factory 返回一个指针，指向一个动态分配的对象
Foo* factory(T arg)
```

```

{
    // 视情况处理 arg
    return new Foo(arg); // 调用者负责释放此内存
}

```

类似我们之前定义的 `factory` 函数（参见 12.1.1 节，第 403 页），这个版本的 `factory` 分配一个对象，但并不 `delete` 它。`factory` 的调用者负责在不需要此对象时释放它。不幸的是，调用者经常忘记释放对象：

```

void use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p 但不 delete 它
} // p 离开了它的作用域，但它所指向的内存没有被释放！

```

此处，`use_factory` 函数调用 `factory`，后者分配一个类型为 `Foo` 的新对象。当 `use_factory` 返回时，局部变量 `p` 被销毁。此变量是一个内置指针，而不是一个智能指针。

与类类型不同，内置类型的对象被销毁时什么也不会发生。特别是，当一个指针离开其作用域时，它所指向的对象什么也不会发生。如果这个指针指向的是动态内存，那么内存将不会被自动释放。



WARNING 由内置指针(而不是智能指针)管理的动态内存存在被显式释放前一直都会存在。

462>

在本例中，`p` 是指向 `factory` 分配的内存的唯一指针。一旦 `use_factory` 返回，程序就没有办法释放这块内存了。根据整个程序的逻辑，修正这个错误的正确方法是在 `use_factory` 中记得释放内存：

```

void use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p
    delete p; // 现在记得释放内存，我们已经不需要它了
}

```

还有一种可能，我们的系统中的其他代码要使用 `use_factory` 所分配的对象，我们就应该修改此函数，让它返回一个指针，指向它分配的内存：

```

Foo* use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p
    return p; // 调用者必须释放内存
}

```

小心：动态内存的管理非常容易出错

使用 `new` 和 `delete` 管理动态内存存在三个常见问题：

1. 忘记 `delete` 内存。忘记释放动态内存会导致人们常说的“内存泄漏”问题，因为这种内存永远不可能被归还给自由空间了。查找内存泄露错误是非常困难的，因为通常应用程序运行很长一段时间后，真正耗尽内存时，才能检测到这种错误。
2. 使用已经释放掉的对象。通过在释放内存后将指针置为空，有时可以检测出这

种错误。

3. 同一块内存释放两次。当有两个指针指向相同的动态分配对象时，可能发生这种错误。如果对其中一个指针进行了 `delete` 操作，对象的内存就被归还给自由空间了。如果我们随后又 `delete` 第二个指针，自由空间就可能被破坏。

相对于查找和修正这些错误来说，制造出这些错误要简单得多。

Best Practices

坚持只使用智能指针，就可以避免所有这些问题。对于一块内存，只有在没有任何智能指针指向它的情况下，智能指针才会自动释放它。

delete 之后重置指针值……

当我们 `delete` 一个指针后，指针值就变为无效了。虽然指针已经无效，但在很多机器上指针仍然保存着（已经释放了的）动态内存的地址。在 `delete` 之后，指针就变成了人们所说的空悬指针（dangling pointer），即，指向一块曾经保存数据对象但现在已经无效的内存的指针。◀ 463

未初始化指针（参见 2.3.2 节，第 49 页）的所有缺点空悬指针也都有。有一种方法可以避免空悬指针的问题：在指针即将要离开其作用域之前释放掉它所关联的内存。这样，在指针关联的内存被释放掉之后，就没有机会继续使用指针了。如果我们需要保留指针，可以在 `delete` 之后将 `nullptr` 赋予指针，这样就清楚地指出指针不指向任何对象。

……这只是提供了有限的保护

动态内存的一个基本问题是可能有多个指针指向相同的内存。在 `delete` 内存之后重置指针的方法只对这个指针有效，对其他任何仍指向（已释放的）内存的指针是没有作用的。例如：

```
int *p(new int(42)); // p 指向动态内存
auto q = p;          // p 和 q 指向相同的内存
delete p;            // p 和 q 均变为无效
p = nullptr;         // 指出 p 不再绑定到任何对象
```

本例中 `p` 和 `q` 指向相同的动态分配的对象。我们 `delete` 此内存，然后将 `p` 置为 `nullptr`，指出它不再指向任何对象。但是，重置 `p` 对 `q` 没有任何作用，在我们释放 `p` 所指向的（同时也是 `q` 所指向的！）内存时，`q` 也变为无效了。在实际系统中，查找指向相同内存的所有指针是异常困难的。

12.1.2 节练习

练习 12.6：编写函数，返回一个动态分配的 `int` 的 `vector`。将此 `vector` 传递给另一个函数，这个函数读取标准输入，将读入的值保存在 `vector` 元素中。再将 `vector` 传递给另一个函数，打印读入的值。记得在恰当的时刻 `delete vector`。

练习 12.7：重做上一题，这次使用 `shared_ptr` 而不是内置指针。

练习 12.8：下面的函数是否有错误？如果有，解释错误原因。

```
bool b() {
    int* p = new int;
    // ...
```

```

    return p;
}

```

练习 12.9：解释下面代码执行的结果：

```

int *q = new int(42), *r = new int(100);
r = q;
auto q2 = make_shared<int>(42), r2 = make_shared<int>(100);
r2 = q2;

```

464 12.1.3 shared_ptr 和 new 结合使用

如前所述，如果我们不初始化一个智能指针，它就会被初始化为一个空指针。如表 12.3 所示，我们还可以用 new 返回的指针来初始化智能指针：

```

shared_ptr<double> p1; // shared_ptr 可以指向一个 double
shared_ptr<int> p2(new int(42)); // p2 指向一个值为 42 的 int

```

接受指针参数的智能指针构造函数是 *explicit* 的（参见 7.5.4 节，第 265 页）。因此，我们不能将一个内置指针隐式转换为一个智能指针，必须使用直接初始化形式（参见 3.2.1 节，第 76 页）来初始化一个智能指针：

```

shared_ptr<int> p1 = new int(1024); // 错误：必须使用直接初始化形式
shared_ptr<int> p2(new int(1024)); // 正确：使用了直接初始化形式

```

p1 的初始化隐式地要求编译器用一个 new 返回的 int* 来创建一个 shared_ptr。由于我们不能进行内置指针到智能指针间的隐式转换，因此这条初始化语句是错误的。出于相同的原因，一个返回 shared_ptr 的函数不能在其返回语句中隐式转换一个普通指针：

```

shared_ptr<int> clone(int p) {
    return new int(p); // 错误：隐式转换为 shared_ptr<int>
}

```

我们必须将 shared_ptr 显式绑定到一个想要返回的指针上：

```

shared_ptr<int> clone(int p) {
    // 正确：显式地用 int* 创建 shared_ptr<int>
    return shared_ptr<int>(new int(p));
}

```

默认情况下，一个用来初始化智能指针的普通指针必须指向动态内存，因为智能指针默认使用 delete 释放它所关联的对象。我们可以将智能指针绑定到一个指向其他类型的资源的指针上，但是为了这样做，必须提供自己的操作来替代 delete。我们将在 12.1.4 节（第 415 页）介绍如何定义自己的释放操作。

表 12.3：定义和改变 shared_ptr 的其他方法

<code>shared_ptr<T> p(q)</code>	<code>p</code> 管理内置指针 <code>q</code> 所指向的对象； <code>q</code> 必须指向 new 分配的内存，且能够转换为 <code>T*</code> 类型
<code>shared_ptr<T> p(u)</code>	<code>p</code> 从 <code>unique_ptr u</code> 那里接管了对象的所有权；将 <code>u</code> 置为空
<code>shared_ptr<T> p(q, d)</code>	<code>p</code> 接管了内置指针 <code>q</code> 所指向的对象的所有权。 <code>q</code> 必须能转换为 <code>T*</code> 类型（参见 4.11.2 节，第 143 页）。 <code>p</code> 将使用可调用对象 <code>d</code> （参见 10.3.2 节，第 346 页）来代替 <code>delete</code>

续表

<code>shared_ptr<T> p(p2, d)</code>	如表 12.2 所示, p 是 shared_ptr p2 的拷贝, 唯一的区别是 p 将用可调用对象 d 来代替 delete
<code>p.reset()</code>	若 p 是唯一指向其对象的 shared_ptr, reset 会释放此对象。若传递了可选的参数内置指针 q, 会令 p 指向 q, 否则会将 p 置为空。若还传递了参数 d, 将会调用 d 而不是 delete 来释放 q
<code>p.reset(q)</code>	
<code>p.reset(q, d)</code>	

不要混合使用普通指针和智能指针……



`shared_ptr` 可以协调对象的析构, 但这仅限于其自身的拷贝 (也是 `shared_ptr`) 之间。这也是为什么我们推荐使用 `make_shared` 而不是 `new` 的原因。这样, 我们就能在分配对象的同时就将 `shared_ptr` 与之绑定, 从而避免了无意中将同一块内存绑定到多个独立创建的 `shared_ptr` 上。

考虑下面对 `shared_ptr` 进行操作的函数:

```
// 在函数被调用时 ptr 被创建并初始化
void process(shared_ptr<int> ptr)
{
    // 使用 ptr
} // ptr 离开作用域, 被销毁
```

`process` 的参数是传值方式传递的, 因此实参会 ◀465 被拷贝到 `ptr` 中。拷贝一个 `shared_ptr` 会递增其引用计数, 因此, 在 `process` 运行过程中, 引用计数值至少为 2。当 `process` 结束时, `ptr` 的引用计数会递减, 但不会变为 0。因此, 当局部变量 `ptr` 被销毁时, `ptr` 指向的内存不会被释放。

使用此函数的正确方法是传递给它一个 `shared_ptr`:

```
shared_ptr<int> p(new int(42)); // 引用计数为 1
process(p); // 拷贝 p 会递增它的引用计数; 在 process 中引用计数值为 2
int i = *p; // 正确: 引用计数值为 1
```

虽然不能传递给 `process` 一个内置指针, 但可以传递给它一个 (临时的) `shared_ptr`, 这个 `shared_ptr` 是用一个内置指针显式构造的。但是, 这样做很可能会导致错误:

```
int *x(new int(1024));           // 危险: x 是一个普通指针, 不是一个智能指针
process(x); // 错误: 不能将 int* 转换为一个 shared_ptr<int>
process(shared_ptr<int>(x)); // 合法的, 但内存会被释放!
int j = *x; // 未定义的: x 是一个空悬指针!
```

在上面的调用中, 我们将一个临时 `shared_ptr` 传递给 `process`。当这个调用所在的表达式结束时, 这个临时对象就被销毁了。销毁这个临时变量会递减引用计数, 此时引用计数就变为 0 了。因此, 当临时对象被销毁时, 它所指向的内存会被释放。

但 `x` 继续指向 (已经释放的) 内存, 从而变成一个空悬指针。如果试图使用 `x` 的值, 其行为是未定义的。

当将一个 `shared_ptr` 绑定到一个普通指针时, 我们就将内存的管理责任交给了这个 `shared_ptr`。一旦这样做了, 我们就不应该再使用内置指针来访问 `shared_ptr` 所指向的内存了。

466



WARNING

使用一个内置指针来访问一个智能指针所负责的对象是很危险的，因为我们无法知道对象何时会被销毁。

467

……也不要使用 get 初始化另一个智能指针或为智能指针赋值

智能指针类型定义了一个名为 `get` 的函数（参见表 12.1），它返回一个内置指针，指向智能指针管理的对象。此函数是为了这样一种情况而设计的：我们需要向不能使用智能指针的代码传递一个内置指针。使用 `get` 返回的指针的代码不能 `delete` 此指针。

虽然编译器不会给出错误信息，但将另一个智能指针也绑定到 `get` 返回的指针上是错误的：

```
shared_ptr<int> p(new int(42)); // 引用计数为 1
int *q = p.get(); // 正确：但使用 q 时要注意，不要让它管理的指针被释放
{ // 新程序块
    // 未定义：两个独立的 shared_ptr 指向相同的内存
    shared_ptr<int>(q);
} // 程序块结束，q 被销毁，它指向的内存被释放
int foo = *p; // 未定义：p 指向的内存已经被释放了
```

在本例中，`p` 和 `q` 指向相同的内存。由于它们是相互独立创建的，因此各自的引用计数都是 1。当 `q` 所在的程序块结束时，`q` 被销毁，这会导致 `q` 指向的内存被释放。从而 `p` 变成一个空悬指针，意味着当我们试图使用 `p` 时，将发生未定义的行为。而且，当 `p` 被销毁时，这块内存会被第二次 `delete`。



WARNING

`get` 用来将指针的访问权限传递给代码，你只有在确定代码不会 `delete` 指针的情况下，才能使用 `get`。特别是，永远不要用 `get` 初始化另一个智能指针或者为另一个智能指针赋值。

其他 `shared_ptr` 操作

`shared_ptr` 还定义了其他一些操作，参见表 12.2 和表 12.3 所示。我们可以用 `reset` 来将一个新的指针赋予一个 `shared_ptr`：

```
p = new int(1024);           // 错误：不能将一个指针赋予 shared_ptr
p.reset(new int(1024));     // 正确：p 指向一个新对象
```

与赋值类似，`reset` 会更新引用计数，如果需要的话，会释放 `p` 指向的对象。`reset` 成员经常与 `unique` 一起使用，来控制多个 `shared_ptr` 共享的对象。在改变底层对象之前，我们检查自己是否是当前对象仅有的用户。如果不是，在改变之前要制作一份新的拷贝：

```
if (!p.unique())
    p.reset(new string(*p)); // 我们不是唯一用户；分配新的拷贝
*p += newVal; // 现在我们知道我们是唯一的用户，可以改变对象的值
```

467

12.1.3 节练习

练习 12.10：下面的代码调用了第 413 页中定义的 `process` 函数，解释此调用是否正确。如果不正确，应如何修改？

```
shared_ptr<int> p(new int(42));
```

```
process(shared_ptr<int>(p));
```

练习 12.11: 如果我们像下面这样调用 process，会发生什么？

```
process(shared_ptr<int>(p.get()));
```

练习 12.12: p 和 q 的定义如下，对于接下来的对 process 的每个调用，如果合法，解释它做了什么，如果不合法，解释错误原因：

```
auto p = new int();
auto sp = make_shared<int>();
(a) process(sp);
(b) process(new int());
(c) process(p);
(d) process(shared_ptr<int>(p));
```

练习 12.13: 如果执行下面的代码，会发生什么？

```
auto sp = make_shared<int>();
auto p = sp.get();
delete p;
```

12.1.4 智能指针和异常



5.6.2 节（第 175 页）中介绍了使用异常处理的程序能在异常发生后令程序流程继续，我们注意到，这种程序需要确保在异常发生后资源能被正确地释放。一个简单的确保资源被释放的方法是使用智能指针。

如果使用智能指针，即使程序块过早结束，智能指针类也能确保在内存不再需要时将其释放，：

```
void f()
{
    shared_ptr<int> sp(new int(42)); // 分配一个新对象
    // 这段代码抛出一个异常，且在 f 中未被捕获
} // 在函数结束时 shared_ptr 自动释放内存
```

函数的退出有两种可能，正常处理结束或者发生了异常，无论哪种情况，局部对象都会被销毁。在上面的程序中，sp 是一个 shared_ptr，因此 sp 销毁时会检查引用计数。在此例中，sp 是指向这块内存的唯一指针，因此内存会被释放掉。

与之相对的，当发生异常时，我们直接管理的内存是不会自动释放的。如果使用内置指针管理内存，且在 new 之后在对应的 delete 之前发生了异常，则内存不会被释放：

```
void f()
{
    int *ip = new int(42); // 动态分配一个新对象
    // 这段代码抛出一个异常，且在 f 中未被捕获
    delete ip;           // 在退出之前释放内存
}
```

468

如果在 new 和 delete 之间发生异常，且异常未在 f 中被捕获，则内存就永远不会被释放了。在函数 f 之外没有指针指向这块内存，因此就无法释放它了。



智能指针和哑类

包括所有标准库类在内的很多 C++ 类都定义了析构函数（参见 12.1.1 节，第 402 页），负责清理对象使用的资源。但是，不是所有的类都是这样良好定义的。特别是那些为 C 和 C++ 两种语言设计的类，通常都要求用户显式地释放所使用的任何资源。

那些分配了资源，而又没有定义析构函数来释放这些资源的类，可能会遇到与使用动态内存相同的错误——程序员非常容易忘记释放资源。类似的，如果在资源分配和释放之间发生了异常，程序也会发生资源泄漏。

与管理动态内存类似，我们通常可以使用类似的技术来管理不具有良好定义的析构函数的类。例如，假定我们正在使用一个 C 和 C++ 都使用的网络库，使用这个库的代码可能是这样的：

```
struct destination;           // 表示我们正在连接什么
struct connection;            // 使用连接所需的信息
connection connect(destination*); // 打开连接
void disconnect(connection);   // 关闭给定的连接
void f(destination &d /* 其他参数 */)
{
    // 获得一个连接；记住使用完后要关闭它
    connection c = connect(&d);
    // 使用连接
    // 如果我们在 f 退出前忘记调用 disconnect，就无法关闭 c 了
}
```

如果 `connection` 有一个析构函数，就可以在 `f` 结束时由析构函数自动关闭连接。但是，`connection` 没有析构函数。这个问题与我们上一个程序中使用 `shared_ptr` 避免内存泄漏几乎是等价的。使用 `shared_ptr` 来保证 `connection` 被正确关闭，已被证明是一种有效的方法。



使用我们自己的释放操作

469 默认情况下，`shared_ptr` 假定它们指向的是动态内存。因此，当一个 `shared_ptr` 被销毁时，它默认地对它管理的指针进行 `delete` 操作。为了用 `shared_ptr` 来管理一个 `connection`，我们必须首先定义一个函数来代替 `delete`。这个删除器（deleter）函数必须能够完成对 `shared_ptr` 中保存的指针进行释放的操作。在本例中，我们的删除器必须接受单个类型为 `connection*` 的参数：

```
void end_connection(connection *p) { disconnect(*p); }
```

当我们创建一个 `shared_ptr` 时，可以传递一个（可选的）指向删除器函数的参数（参见 6.7 节，第 221 页）：

```
void f(destination &d /* 其他参数 */)
{
    connection c = connect(&d);
    shared_ptr<connection> p(&c, end_connection);
    // 使用连接
    // 当 f 退出时（即使是由异常而退出），connection 会被正确关闭
}
```

当 p 被销毁时，它不会对自己保存的指针执行 delete，而是调用 end_connection。接下来，end_connection 会调用 disconnect，从而确保连接被关闭。如果 f 正常退出，那么 p 的销毁会作为结束处理的一部分。如果发生了异常，p 同样会被销毁，从而连接被关闭。

注意：智能指针陷阱

智能指针可以提供对动态分配的内存安全而又方便的管理，但这建立在正确使用的前提下。为了正确使用智能指针，我们必须坚持一些基本规范：

- 不使用相同的内置指针值初始化（或 reset）多个智能指针。
- 不 delete get() 返回的指针。
- 不使用 get() 初始化或 reset 另一个智能指针。
- 如果你使用 get() 返回的指针，记住当最后一个对应的智能指针销毁后，你的指针就变为无效了。
- 如果你使用智能指针管理的资源不是 new 分配的内存，记住传递给它一个删除器（参见 12.1.4 节，第 415 页和 12.1.5 节，第 419 页）。

12.1.4 节练习

练习 12.14：编写你自己版本的用 shared_ptr 管理 connection 的函数。

练习 12.15：重写第一题的程序，用 lambda（参见 10.3.2 节，第 346 页）代替 end_connection 函数。

12.1.5 unique_ptr

< 470

一个 unique_ptr “拥有” 它所指向的对象。与 shared_ptr 不同，某个时刻只能有一个 unique_ptr 指向一个给定对象。当 unique_ptr 被销毁时，它所指向的对象也被销毁。表 12.4 列出了 unique_ptr 特有的操作。与 shared_ptr 相同的操作列在表 12.1（第 401 页）中。

C++
11

与 shared_ptr 不同，没有类似 make_shared 的标准库函数返回一个 unique_ptr。当我们定义一个 unique_ptr 时，需要将其绑定到一个 new 返回的指针上。类似 shared_ptr，初始化 unique_ptr 必须采用直接初始化形式：

```
unique_ptr<double> p1; // 可以指向一个 double 的 unique_ptr  
unique_ptr<int> p2(new int(42)); // p2 指向一个值为 42 的 int
```

由于一个 unique_ptr 拥有它指向的对象，因此 unique_ptr 不支持普通的拷贝或赋值操作：

```
unique_ptr<string> p1(new string("Stegosaurus"));  
unique_ptr<string> p2(p1); // 错误：unique_ptr 不支持拷贝  
unique_ptr<string> p3;  
p3 = p2; // 错误：unique_ptr 不支持赋值
```

表 12.4: unique_ptr 操作 (另参见表 12.1, 第 401 页)	
unique_ptr<T> u1	空 unique_ptr, 可以指向类型为 T 的对象。u1 会使用 delete 来释放它的指针;
unique_ptr<T, D> u2	u2 会使用一个类型为 D 的可调用对象来释放它的指针
unique_ptr<T, D> u(d)	空 unique_ptr, 指向类型为 T 的对象, 用类型为 D 的对象 d 替代 delete
u = nullptr	释放 u 指向的对象, 将 u 置为空
u.release()	u 放弃对指针的控制权, 返回指针, 并将 u 置为空
u.reset()	释放 u 指向的对象
u.reset(q)	如果提供了内置指针 q, 令 u 指向这个对象; 否则将 u 置为空
u.reset(nullptr)	

虽然我们不能拷贝或赋值 unique_ptr, 但可以通过调用 release 或 reset 将指针的所有权从一个 (非 const) unique_ptr 转移给另一个 unique:

```
// 将所有权从 p1 (指向 string Stegosaurus) 转移给 p2
unique_ptr<string> p2(p1.release()); // release 将 p1 置为空
unique_ptr<string> p3(new string("Trex"));
// 将所有权从 p3 转移给 p2
p2.reset(p3.release()); // reset 释放了 p2 原来指向的内存
```

release 成员返回 unique_ptr 当前保存的指针并将其置为空。因此, p2 被初始化为 p1 原来保存的指针, 而 p1 被置为空。

471 > reset 成员接受一个可选的指针参数, 令 unique_ptr 重新指向给定的指针。如果 unique_ptr 不为空, 它原来指向的对象被释放。因此, 对 p2 调用 reset 释放了用 "Stegosaurus" 初始化的 string 所使用的内存, 将 p3 对指针的所有权转移给 p2, 并将 p3 置为空。

调用 release 会切断 unique_ptr 和它原来管理的对象间的联系。release 返回的指针通常被用来初始化另一个智能指针或给另一个智能指针赋值。在本例中, 管理内存的责任简单地从一个智能指针转移给另一个。但是, 如果我们不用另一个智能指针来保存 release 返回的指针, 我们的程序就要负责资源的释放:

```
p2.release(); // 错误: p2 不会释放内存, 而且我们丢失了指针
auto p = p2.release(); // 正确, 但我们必须记得 delete(p)
```

传递 unique_ptr 参数和返回 unique_ptr

不能拷贝 unique_ptr 的规则有一个例外: 我们可以拷贝或赋值一个将要被销毁的 unique_ptr。最常见的例子是从函数返回一个 unique_ptr:

```
unique_ptr<int> clone(int p) {
    // 正确: 从 int* 创建一个 unique_ptr<int>
    return unique_ptr<int>(new int(p));
}
```

还可以返回一个局部对象的拷贝:

```
unique_ptr<int> clone(int p) {
    unique_ptr<int> ret(new int (p));
    // ...
    return ret;
}
```

对于两段代码，编译器都知道要返回的对象将要被销毁。在此情况下，编译器执行一种特殊的“拷贝”，我们将在 13.6.2 节（第 473 页）中介绍它。

向后兼容：auto_ptr

标准库的较早版本包含了一个名为 `auto_ptr` 的类，它具有 `unique_ptr` 的部分特性，但不是全部。特别是，我们不能在容器中保存 `auto_ptr`，也不能从函数中返回 `auto_ptr`。

虽然 `auto_ptr` 仍是标准库的一部分，但编写程序时应该使用 `unique_ptr`。

向 unique_ptr 传递删除器

类似 `shared_ptr`, `unique_ptr` 默认情况下用 `delete` 释放它指向的对象。与 `shared_ptr` 一样，我们可以重载一个 `unique_ptr` 中默认的删除器（参见 12.1.4 节，第 415 页）。但是，`unique_ptr` 管理删除器的方式与 `shared_ptr` 不同，其原因我们将在 16.1.6 节（第 599 页）中介绍。

< 472

重载一个 `unique_ptr` 中的删除器会影响到 `unique_ptr` 类型以及如何构造（或 `reset`）该类型的对象。与重载关联容器的比较操作（参见 11.2.2 节，第 378 页）类似，我们必须在尖括号中 `unique_ptr` 指向类型之后提供删除器类型。在创建或 `reset` 一个这种 `unique_ptr` 类型的对象时，必须提供一个指定类型的可调用对象（删除器）：

```
// p 指向一个类型为 objT 的对象，并使用一个类型为 delT 的对象释放 objT 对象
// 它会调用一个名为 fcn 的 delT 类型对象
unique_ptr<objT, delT> p (new objT, fcn);
```

作为一个更具体的例子，我们将重写连接程序，用 `unique_ptr` 来代替 `shared_ptr`，如下所示：

```
void f(destination &d /* 其他需要的参数 */)
{
    connection c = connect(&d); // 打开连接
    // 当 p 被销毁时，连接将会关闭
    unique_ptr<connection, decltype(end_connection)*>
        p(&c, end_connection);
    // 使用连接
    // 当 f 退出时（即使是由于异常而退出），connection 会被正确关闭
}
```

在本例中我们使用了 `decltype`（参见 2.5.3 节，第 62 页）来指明函数指针类型。由于 `decltype(end_connection)` 返回一个函数类型，所以我们必须添加一个*来指出我们正在使用该类型的一个指针（参见 6.7 节，第 223 页）。

12.1.5 节练习

练习 12.16：如果你试图拷贝或赋值 `unique_ptr`，编译器并不总是能给出易于理解的错误信息。编写包含这种错误的程序，观察编译器如何诊断这种错误。

练习 12.17：下面的 `unique_ptr` 声明中，哪些是合法的，哪些可能导致后续的程序错误？解释每个错误的问题在哪里。

```

int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef unique_ptr<int> IntP;
(a) IntP p0(ix);           (b) IntP p1(pi);
(c) IntP p2(pi2);         (d) IntP p3(&ix);
(e) IntP p4(new int(2048)); (f) IntP p5(p2.get());

```

练习 12.18: `shared_ptr` 为什么没有 `release` 成员?



12.1.6 weak_ptr

473

C++
11

`weak_ptr` (见表 12.5) 是一种不控制所指向对象生存期的智能指针, 它指向由一个 `shared_ptr` 管理的对象。将一个 `weak_ptr` 绑定到一个 `shared_ptr` 不会改变 `shared_ptr` 的引用计数。一旦最后一个指向对象的 `shared_ptr` 被销毁, 对象就会被释放。即使有 `weak_ptr` 指向对象, 对象也还是会被释放, 因此, `weak_ptr` 的名字抓住了这种智能指针“弱”共享对象的特点。

表 12.5: `weak_ptr`

<code>weak_ptr<T> w</code>	空 <code>weak_ptr</code> 可以指向类型为 <code>T</code> 的对象
<code>weak_ptr<T> w(sp)</code>	与 <code>shared_ptr</code> <code>sp</code> 指向相同对象的 <code>weak_ptr</code> 。 <code>T</code> 必须能转换为 <code>sp</code> 指向的类型
<code>w = p</code>	<code>p</code> 可以是一个 <code>shared_ptr</code> 或一个 <code>weak_ptr</code> 。赋值后 <code>w</code> 与 <code>p</code> 共享对象
<code>w.reset()</code>	将 <code>w</code> 置为空
<code>w.use_count()</code>	与 <code>w</code> 共享对象的 <code>shared_ptr</code> 的数量
<code>w.expired()</code>	若 <code>w.use_count()</code> 为 0, 返回 <code>true</code> , 否则返回 <code>false</code>
<code>w.lock()</code>	如果 <code>expired</code> 为 <code>true</code> , 返回一个空 <code>shared_ptr</code> ; 否则返回一个指向 <code>w</code> 的对象的 <code>shared_ptr</code>

当我们创建一个 `weak_ptr` 时, 要用一个 `shared_ptr` 来初始化它:

```

auto p = make_shared<int>(42);
weak_ptr<int> wp(p); // wp 弱共享 p; p 的引用计数未改变

```

本例中 `wp` 和 `p` 指向相同的对象。由于是弱共享, 创建 `wp` 不会改变 `p` 的引用计数; `wp` 指向的对象可能被释放掉。

由于对象可能不存在, 我们不能使用 `weak_ptr` 直接访问对象, 而必须调用 `lock`。此函数检查 `weak_ptr` 指向的对象是否仍存在。如果存在, `lock` 返回一个指向共享对象的 `shared_ptr`。与任何其他 `shared_ptr` 类似, 只要此 `shared_ptr` 存在, 它所指向的底层对象也就会一直存在。例如:

```

if (shared_ptr<int> np = wp.lock()) { // 如果 np 不为空则条件成立
    // 在 if 中, np 与 p 共享对象
}

```

在这段代码中, 只有当 `lock` 调用返回 `true` 时我们才会进入 `if` 语句体。在 `if` 中, 使用 `np` 访问共享对象是安全的。

核查指针类

作为 `weak_ptr` 用途的一个展示, 我们将为 `StrBlob` 类定义一个伴随指针类。我们

的指针类将命名为 StrBlobPtr，会保存一个 `weak_ptr`，指向 `StrBlob` 的 `data` 成员，这是初始化时提供给它的。通过使用 `weak_ptr`，不会影响一个给定的 `StrBlob` 所指向的 `vector` 的生存期。但是，可以阻止用户访问一个不再存在的 `vector` 的企图。 ◀ 474

`StrBlobPtr` 会有两个数据成员：`wptr`，或者为空，或者指向一个 `StrBlob` 中的 `vector`；`curr`，保存当前对象所表示的元素的下标。类似它的伴随类 `StrBlob`，我们的指针类也有一个 `check` 成员来检查解引用 `StrBlobPtr` 是否安全：

```
// 对于访问一个不存在元素的尝试，StrBlobPtr 抛出一个异常
class StrBlobPtr {
public:
    StrBlobPtr(): curr(0) { }
    StrBlobPtr(StrBlob &a, size_t sz = 0):
        wptr(a.data), curr(sz) { }
    std::string& deref() const;
    StrBlobPtr& incr(); // 前缀递增
private:
    // 若检查成功，check 返回一个指向 vector 的 shared_ptr
    std::shared_ptr<std::vector<std::string>>
        check(std::size_t, const std::string&) const;
    // 保存一个 weak_ptr，意味着底层 vector 可能会被销毁
    std::weak_ptr<std::vector<std::string>> wptr;
    std::size_t curr; // 在数组中的当前位置
};
```

默认构造函数生成一个空的 `StrBlobPtr`。其构造函数初始化列表（参见 7.1.4 节，第 237 页）将 `curr` 显式初始化为 0，并将 `wptr` 隐式初始化为一个空 `weak_ptr`。第二个构造函数接受一个 `StrBlob` 引用和一个可选的索引值。此构造函数初始化 `wptr`，令其指向给定 `StrBlob` 对象的 `shared_ptr` 中的 `vector`，并将 `curr` 初始化为 `sz` 的值。我们使用了默认参数（参见 6.5.1 节，第 211 页），表示默认情况下将 `curr` 初始化为第一个元素的下标。我们将会看到，`StrBlob` 的 `end` 成员将会用到参数 `sz`。

值得注意的是，我们不能将 `StrBlobPtr` 绑定到一个 `const StrBlob` 对象。这个限制是由于构造函数接受一个非 `const StrBlob` 对象的引用而导致的。

`StrBlobPtr` 的 `check` 成员与 `StrBlob` 中的同名成员不同，它还要检查指针指向的 `vector` 是否还存在：

```
std::shared_ptr<std::vector<std::string>>
StrBlobPtr::check(std::size_t i, const std::string &msg) const
{
    auto ret = wptr.lock(); // vector 还存在吗？
    if (!ret)
        throw std::runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw std::out_of_range(msg);
    return ret; // 否则，返回指向 vector 的 shared_ptr
}
```

由于一个 `weak_ptr` 不参与其对应的 `shared_ptr` 的引用计数，`StrBlobPtr` 指向的 `vector` 可能已经被释放了。如果 `vector` 已销毁，`lock` 将返回一个空指针。在本例中，任何 `vector` 的引用都会失败，于是抛出一个异常。否则，`check` 会检查给定索引，如果索引合法，`check` 返回从 `lock` 获得的 `shared_ptr`。 ◀ 475

指针操作

我们将在第 14 章学习如何定义自己的运算符。现在，我们将定义名为 deref 和 incr 的函数，分别用来解引用和递增 StrBlobPtr。

deref 成员调用 check，检查使用 vector 是否安全以及 curr 是否在合法范围内：

```
std::string& StrBlobPtr::deref() const
{
    auto p = check(curr, "dereference past end");
    return (*p)[curr]; // (*p) 是对象所指向的 vector
}
```

如果 check 成功，p 就是一个 shared_ptr，指向 StrBlobPtr 所指向的 vector。表达式 (*p)[curr] 解引用 shared_ptr 来获得 vector，然后使用下标运算符提取并返回 curr 位置上的元素。

incr 成员也调用 check：

```
// 前缀递增：返回递增后的对象的引用
StrBlobPtr& StrBlobPtr::incr()
{
    // 如果 curr 已经指向容器的尾后位置，就不能递增它
    check(curr, "increment past end of StrBlobPtr");
    ++curr; // 推进当前位置
    return *this;
}
```

当然，为了访问 data 成员，我们的指针类必须声明为 StrBlob 的 friend（参见 7.3.4 节，第 250 页）。我们还要为 StrBlob 类定义 begin 和 end 操作，返回一个指向它自身的 StrBlobPtr：

```
// 对于 StrBlob 中的友元声明来说，此前置声明是必要的
class StrBlobPtr;
class StrBlob {
    friend class StrBlobPtr;
    // 其他成员与 12.1.1 节（第 405 页）中声明相同
    // 返回指向首元素和尾后元素的 StrBlobPtr
    StrBlobPtr begin() { return StrBlobPtr(*this); }
    StrBlobPtr end()
    { auto ret = StrBlobPtr(*this, data->size());
      return ret; }
};
```

12.1.6 节练习

练习 12.19： 定义你自己版本的 StrBlobPtr，更新 StrBlob 类，加入恰当的 friend 声明及 begin 和 end 成员。

练习 12.20： 编写程序，逐行读入一个输入文件，将内容存入一个 StrBlob 中，用一个 StrBlobPtr 打印出 StrBlob 中的每个元素。

练习 12.21： 也可以这样编写 StrBlobPtr 的 deref 成员：

```
std::string& deref() const
```

```
{ return (*check(curr, "dereference past end))[curr]; }
```

你认为哪个版本更好？为什么？

练习 12.22: 为了能让 `StrBlobPtr` 使用 `const StrBlob`, 你觉得应该如何修改？定义一个名为 `ConstStrBlobPtr` 的类，使其能够指向 `const StrBlob`。



12.2 动态数组

`new` 和 `delete` 运算符一次分配/释放一个对象，但某些应用需要一次为很多对象分配内存的功能。例如，`vector` 和 `string` 都是在连续内存中保存它们的元素，因此，当容器需要重新分配内存时（参见 9.4 节，第 317 页），必须一次性为很多元素分配内存。

为了支持这种需求，C++语言和标准库提供了两种一次分配一个对象数组的方法。C++语言定义了另一种 `new` 表达式语法，可以分配并初始化一个对象数组。标准库中包含一个名为 `allocator` 的类，允许我们将分配和初始化分离。使用 `allocator` 通常会提供更好的性能和更灵活的内存管理能力，原因我们将在 12.2.2 节（第 427 页）中解释。

很多（可能是大多数）应用都没有直接访问动态数组的需求。当一个应用需要可变数量的对象时，我们在 `StrBlob` 中所采用的方法几乎总是更简单、更快速并且更安全的——即，使用 `vector`（或其他标准库容器）。如我们将在 13.6 节（第 470 页）中看到的，使用标准库容器的优势在新标准下更为显著。在支持新标准的标准库中，容器操作比之前的版本要快速得多。

Best Practices

大多数应用应该使用标准库容器而不是动态分配的数组。使用容器更为简单、更不容易出现内存管理错误并且可能有更好的性能。

如前所述，使用容器的类可以使用默认版本的拷贝、赋值和析构操作（参见 7.1.5 节，第 239 页）。分配动态数组的类则必须定义自己版本的操作，在拷贝、复制以及销毁对象时管理所关联的内存。



直到学习完第 13 章，不要在类内的代码中分配动态内存。



12.2.1 new 和数组

477

为了让 `new` 分配一个对象数组，我们要在类型名之后跟一对方括号，在其中指明要分配的对象的数目。在下例中，`new` 分配要求数量的对象并（假定分配成功后）返回指向第一个对象的指针：

```
// 调用 get_size 确定分配多少个 int
int *pia = new int[get_size()]; // pia 指向第一个 int
```

方括号中的大小必须是整型，但不必是常量。

也可以用一个表示数组类型的类型别名（参见 2.5.1 节，第 60 页）来分配一个数组，这样，`new` 表达式中就不需要方括号了：

```
typedef int arrT[42];      // arrT 表示 42 个 int 的数组类型
int *p = new arrT;         // 分配一个 42 个 int 的数组；p 指向第一个 int
```

在本例中，`new` 分配一个 `int` 数组，并返回指向第一个 `int` 的指针。即使这段代码中没

有方括号，编译器执行这个表达式时还是会用 new[]。即，编译器执行如下形式：

```
int *p = new int[42];
```

分配一个数组会得到一个元素类型的指针

虽然我们通常称 new T[] 分配的内存为“动态数组”，但这种叫法某种程度上有些误导。当用 new 分配一个数组时，我们并未得到一个数组类型的对象，而是得到一个数组元素类型的指针。即使我们使用类型别名定义了一个数组类型，new 也不会分配一个数组类型的对象。在上例中，我们正在分配一个数组的事实甚至都是不可见的——连 [num] 都没有。new 返回的是一个元素类型的指针。

由于分配的内存并不是一个数组类型，因此不能对动态数组调用 begin 或 end（参见 3.5.3 节，第 106 页）。这些函数使用数组维度（回忆一下，维度是数组类型的一部分）来返回指向首元素和尾后元素的指针。出于相同的原因，也不能用范围 for 语句来处理（所谓的）动态数组中的元素。



WARNING

要记住我们所说的动态数组并不是数组类型，这是很重要的。

初始化动态分配对象的数组

默认情况下，new 分配的对象，不管是单个分配的还是数组中的，都是默认初始化的。可以对数组中的元素进行值初始化（参见 3.3.1 节，第 88 页），方法是在大小之后跟一对空括号。

```
int *pia = new int[10];           // 10 个未初始化的 int
int *pia2 = new int[10]();        // 10 个值初始化为 0 的 int
string *psa = new string[10];    // 10 个空 string
string *psa2 = new string[10](); // 10 个空 string
```

478 在新标准中，我们还可以提供一个元素初始化器的花括号列表：

```
// 10 个 int 分别用列表中对应的初始化器初始化
int *pia3 = new int[10]{0,1,2,3,4,5,6,7,8,9};
// 10 个 string，前 4 个用给定的初始化器初始化，剩余的进行值初始化
string *psa3 = new string[10]{"a", "an", "the", string(3,'x')};
```

与内置数组对象的列表初始化（参见 3.5.1 节，第 102 页）一样，初始化器会用来初始化动态数组中开始部分的元素。如果初始化器数目小于元素数目，剩余元素将进行值初始化。如果初始化器数目大于元素数目，则 new 表达式失败，不会分配任何内存。在本例中，new 会抛出一个类型为 bad_array_new_length 的异常。类似 bad_alloc，此类型定义在头文件 new 中。

C++ 11 虽然我们用空括号对数组中元素进行值初始化，但不能在括号中给出初始化器，这意味着不能用 auto 分配数组（参见 12.1.2 节，第 407 页）。

动态分配一个空数组是合法的

可以用任意表达式来确定要分配的对象的数目：

```
size_t n = get_size(); // get_size 返回需要的元素的数目
int* p = new int[n]; // 分配数组保存元素
for (int* q = p; q != p + n; ++q)
/* 处理数组 */;
```

这产生了一个有意思的问题：如果 `get_size` 返回 0，会发生什么？答案是代码仍能正常工作。虽然我们不能创建一个大小为 0 的静态数组对象，但当 `n` 等于 0 时，调用 `new[n]` 是合法的：

```
char arr[0];           // 错误：不能定义长度为 0 的数组
char *cp = new char[0]; // 正确：但 cp 不能解引用
```

当我们用 `new` 分配一个大小为 0 的数组时，`new` 返回一个合法的非空指针。此指针保证与 `new` 返回的其他任何指针都不相同。对于零长度的数组来说，此指针就像尾后指针一样（参见 3.5.3 节，第 106 页），我们可以像使用尾后迭代器一样使用这个指针。可以用此指针进行比较操作，就像上面循环代码中那样。可以向此指针加上（或从此指针减去）0，也可以从此指针减去自身从而得到 0。但此指针不能解引用——毕竟它不指向任何元素。

在我们假想的循环中，若 `get_size` 返回 0，则 `n` 也是 0，`new` 会分配 0 个对象。`for` 循环中的条件会失败（`p` 等于 `q+n`，因为 `n` 为 0）。因此，循环体不会被执行。

释放动态数组

为了释放动态数组，我们使用一种特殊形式的 `delete`——在指针前加上一个空方括号对：

```
delete p;           // p 必须指向一个动态分配的对象或为空
delete [] pa;      // pa 必须指向一个动态分配的数组或为空
```

< 479

第二条语句销毁 `pa` 指向的数组中的元素，并释放对应的内存。数组中的元素按逆序销毁，即，最后一个元素首先被销毁，然后是倒数第二个，依此类推。

当我们释放一个指向数组的指针时，空方括号对是必需的：它指示编译器此指针指向一个对象数组的第一个元素。如果我们在 `delete` 一个指向数组的指针时忽略了方括号（或者在 `delete` 一个指向单一对象的指针时使用了方括号），其行为是未定义的。

回忆一下，当我们使用一个类型别名来定义一个数组类型时，在 `new` 表达式中不使用 `[]`。即使这样，在释放一个数组指针时也必须使用方括号：

```
typedef int arrT[42];    // arrT 是 42 个 int 的数组的类型别名
int *p = new arrT;       // 分配一个 42 个 int 的数组；p 指向第一个元素
delete [] p;             // 方括号是必需的，因为我们当初分配的是一个数组
```

不管外表如何，`p` 指向一个对象数组的首元素，而不是一个类型为 `arrT` 的单一对象。因此，在释放 `p` 时我们必须使用 `[]`。



如果我们在 `delete` 一个数组指针时忘记了方括号，或者在 `delete` 一个单一对象的指针时使用了方括号，编译器很可能不会给出警告。我们的程序可能在执行过程中在没有任何警告的情况下行为异常。

智能指针和动态数组

标准库提供了一个可以管理 `new` 分配的数组的 `unique_ptr` 版本。为了用一个 `unique_ptr` 管理动态数组，我们必须在对象类型后面跟一对空方括号：

```
// up 指向一个包含 10 个未初始化 int 的数组
unique_ptr<int[]> up(new int[10]);
up.release(); // 自动用 delete[] 销毁其指针
```

类型说明符中的方括号 (`<int[]>`) 指出 `up` 指向一个 `int` 数组而不是一个 `int`。由于 `up` 指向一个数组，当 `up` 销毁它管理的指针时，会自动使用 `delete[]`。

指向数组的 `unique_ptr` 提供的操作与我们在 12.1.5 节（第 417 页）中使用的那些操作有一些不同，我们在表 12.6 中描述了这些操作。当一个 `unique_ptr` 指向一个数组时，我们不能使用点和箭头成员运算符。毕竟 `unique_ptr` 指向的是一个数组而不是单个对象，因此这些运算符是无意义的。另一方面，当一个 `unique_ptr` 指向一个数组时，我们可以使用下标运算符来访问数组中的元素：

```
for (size_t i = 0; i != 10; ++i)
    up[i] = i; // 为每个元素赋予一个新值
```

480 >

表 12.6：指向数组的 `unique_ptr`

指向数组的 `unique_ptr` 不支持成员访问运算符（点和箭头运算符）。

其他 `unique_ptr` 操作不变。

`unique_ptr<T[]> u` `u` 可以指向一个动态分配的数组，数组元素类型为 `T`

`unique_ptr<T[]> u(p)` `u` 指向内置指针 `p` 所指向的动态分配的数组。`p` 必须能转换为类型 `T*`（参见 4.11.2 节，第 143 页）

`u[i]` 返回 `u` 拥有的数组中位置 `i` 处的对象

`u` 必须指向一个数组

与 `unique_ptr` 不同，`shared_ptr` 不直接支持管理动态数组。如果希望使用 `shared_ptr` 管理一个动态数组，必须提供自己定义的删除器：

```
// 为了使用 shared_ptr，必须提供一个删除器
shared_ptr<int> sp(new int[10], [](int *p) { delete[] p; });
sp.reset(); // 使用我们提供的 lambda 释放数组，它使用 delete[]
```

本例中我们传递给 `shared_ptr` 一个 `lambda`（参见 10.3.2 节，第 346 页）作为删除器，它使用 `delete[]` 释放数组。

如果未提供删除器，这段代码将是未定义的。默认情况下，`shared_ptr` 使用 `delete` 销毁它指向的对象。如果此对象是一个动态数组，对其使用 `delete` 所产生的问题与释放一个动态数组指针时忘记 `[]` 产生的问题一样（参见 12.2.1 节，第 425 页）。

`shared_ptr` 不直接支持动态数组管理这一特性会影响我们如何访问数组中的元素：

```
// shared_ptr 未定义下标运算符，并且不支持指针的算术运算
for (size_t i = 0; i != 10; ++i)
    *(sp.get() + i) = i; // 使用 get 获取一个内置指针
```

`shared_ptr` 未定义下标运算符，而且智能指针类型不支持指针算术运算。因此，为了访问数组中的元素，必须用 `get` 获取一个内置指针，然后用它来访问数组元素。

12.2.1 节练习

练习 12.23： 编写一个程序，连接两个字符串字面常量，将结果保存在一个动态分配的 `char` 数组中。重写这个程序，连接两个标准库 `string` 对象。

练习 12.24： 编写一个程序，从标准输入读取一个字符串，存入一个动态分配的字符数组中。描述你的程序如何处理变长输入。测试你的程序，输入一个超出你分配的数组长度的字符串。

练习 12.25：给定下面的 new 表达式，你应该如何释放 pa？

```
int *pa = new int[10];
```



12.2.2 allocator 类

<481

new 有一些灵活性上的局限，其中一方面表现在它将内存分配和对象构造组合在了一起。类似的，delete 将对象析构和内存释放组合在了一起。我们分配单个对象时，通常希望将内存分配和对象初始化组合在一起。因为在这种情况下，我们几乎肯定知道对象应有什么值。

当分配一大块内存时，我们通常计划在这块内存上按需构造对象。在此情况下，我们希望将内存分配和对象构造分离。这意味着我们可以分配大块内存，但只在真正需要时才真正执行对象创建操作（同时付出一定开销）。

一般情况下，将内存分配和对象构造组合在一起可能会导致不必要的浪费。例如：

```
string *const p = new string[n]; // 构造 n 个空 string
string s;
string *q = p; // q 指向第一个 string
while (cin >> s && q != p + n)
    *q++ = s; // 赋予*q 一个新值
const size_t size = q - p; // 记住我们读取了多少个 string
// 使用数组
delete[] p; // p 指向一个数组；记得用 delete[] 来释放
```

new 表达式分配并初始化了 n 个 string。但是，我们可能不需要 n 个 string，少量 string 可能就足够了。这样，我们就可能创建了一些永远也用不到的对象。而且，对于那些确实要使用的对象，我们也在初始化之后立即赋予了它们新值。每个使用到的元素都被赋值了两次：第一次是在默认初始化时，随后是在赋值时。

更重要的是，那些没有默认构造函数的类就不能动态分配数组了。

allocator 类

标准库 **allocator** 类定义在头文件 `memory` 中，它帮助我们将内存分配和对象构造分离开来。它提供一种类型感知的内存分配方法，它分配的内存是原始的、未构造的。表 12.7 概述了 **allocator** 支持的操作。在本节中，我们将介绍这些 **allocator** 操作。在 13.5 节（第 464 页），我们将看到如何使用这个类的典型例子。

类似 `vector`，`allocator` 是一个模板（参见 3.3 节，第 86 页）。为了定义一个 `allocator` 对象，我们必须指明这个 `allocator` 可以分配的对象类型。当一个 `allocator` 对象分配内存时，它会根据给定的对象类型来确定恰当的内存大小和对齐位置：

```
allocator<string> alloc; // 可以分配 string 的 allocator 对象
auto const p = alloc.allocate(n); // 分配 n 个未初始化的 string
```

这个 `allocate` 调用为 n 个 string 分配了内存。

482

表 12.7：标准库 allocator 类及其算法

allocator<T> a	定义了一个名为 a 的 allocator 对象，它可以为类型为 T 的对象分配内存
a.allocate(n)	分配一段原始的、未构造的内存，保存 n 个类型为 T 的对象
a.deallocate(p, n)	释放从 T* 指针 p 中地址开始的内存，这块内存保存了 n 个类型为 T 的对象；p 必须是一个先前由 allocate 返回的指针，且 n 必须是 p 创建时所要求的大小。在调用 deallocate 之前，用户必须对每个在这块内存中创建的对象调用 destroy
a.construct(p, args)	p 必须是一个类型为 T* 的指针，指向一块原始内存；arg 被传递给类型为 T 的构造函数，用来在 p 指向的内存中构造一个对象
a.destroy(p)	p 为 T* 类型的指针，此算法对 p 指向的对象执行析构函数（参见 12.1.1 节，第 402 页）

allocator 分配未构造的内存

allocator 分配的内存是未构造的（unconstructed）。我们按需要在此内存中构造对象。在新标准库中，construct 成员函数接受一个指针和零个或多个额外参数，在给定位置构造一个元素。额外参数用来初始化构造的对象。类似 make_shared 的参数（参见 12.1.1 节，第 401 页），这些额外参数必须是与构造的对象的类型相匹配的合法的初始化器：

```
auto q = p; // q 指向最后构造的元素之后的位置
alloc.construct(q++); // *q 为空字符串
alloc.construct(q++, 10, 'c'); // *q 为 cccccccccc
alloc.construct(q++, "hi"); // *q 为 hi!
```

在早期版本的标准库中，construct 只接受两个参数：指向创建对象位置的指针和一个元素类型的值。因此，我们只能将一个元素拷贝到未构造空间中，而不能用元素类型的任何其他构造函数来构造一个元素。

还未构造对象的情况下就使用原始内存是错误的：

```
cout << *p << endl; // 正确：使用 string 的输出运算符
cout << *q << endl; // 灾难：q 指向未构造的内存！
```



为了使用 allocate 返回的内存，我们必须用 construct 构造对象。使用未构造的内存，其行为是未定义的。

当我们用完对象后，必须对每个构造的元素调用 destroy 来销毁它们。函数 destroy 接受一个指针，对指向的对象执行析构函数（参见 12.1.1 节，第 402 页）：

```
483 while (q != p)
        alloc.destroy(--q); // 释放我们真正构造的 string
```

在循环开始处，q 指向最后构造的元素之后的位置。我们在调用 destroy 之前对 q 进行了递减操作。因此，第一次调用 destroy 时，q 指向最后一个构造的元素。最后一步循环中我们 destroy 了第一个构造的元素，随后 q 将与 p 相等，循环结束。



我们只能对真正构造了的元素进行 destroy 操作。

一旦元素被销毁后，就可以重新使用这部分内存来保存其他 string，也可以将其归

还给系统。释放内存通过调用 `deallocate` 来完成：

```
alloc.deallocate(p, n);
```

我们传递给 `deallocate` 的指针不能为空，它必须指向由 `allocate` 分配的内存。而且，传递给 `deallocate` 的大小参数必须与调用 `allocate` 分配内存时提供的大小参数具有一样的值。

拷贝和填充未初始化内存的算法

标准库还为 `allocator` 类定义了两个伴随算法，可以在未初始化内存中创建对象。表 12.8 描述了这些函数，它们都定义在头文件 `memory` 中。

表 12.8: `allocator` 算法

这些函数在给定目的位置创建元素，而不是由系统分配内存给它们。

<code>uninitialized_copy(b, e, b2)</code>	从迭代器 <code>b</code> 和 <code>e</code> 指出的输入范围内拷贝元素到迭代器 <code>b2</code> 指定的未构造的原始内存中。 <code>b2</code> 指向的内存必须足够大，能容纳输入序列中元素的拷贝
<code>uninitialized_copy_n(b, n, b2)</code>	从迭代器 <code>b</code> 指向的元素开始，拷贝 <code>n</code> 个元素到 <code>b2</code> 开始的内存中
<code>uninitialized_fill(b, e, t)</code>	在迭代器 <code>b</code> 和 <code>e</code> 指定的原始内存范围内创建对象，对象的值均为 <code>t</code> 的拷贝
<code>uninitialized_fill_n(b, n, t)</code>	从迭代器 <code>b</code> 指向的内存地址开始创建 <code>n</code> 个对象。 <code>b</code> 必须指向足够大的未构造的原始内存，能够容纳给定数量的对象

作为一个例子，假定有一个 `int` 的 `vector`，希望将其内容拷贝到动态内存中。我们将分配一块比 `vector` 中元素所占用空间大一倍的动态内存，然后将原 `vector` 中的元素拷贝到前一半空间，对后一半空间用一个给定值进行填充：

```
// 分配比 vi 中元素所占用空间大一倍的动态内存
auto p = alloc.allocate(vi.size() * 2);
// 通过拷贝 vi 中的元素来构造从 p 开始的元素
auto q = uninitialized_copy(vi.begin(), vi.end(), p);
// 将剩余元素初始化为 42
uninitialized_fill_n(q, vi.size(), 42);
```

484

类似拷贝算法（参见 10.2.2 节，第 341 页），`uninitialized_copy` 接受三个迭代器参数。前两个表示输入序列，第三个表示这些元素将要拷贝到的目的空间。传递给 `uninitialized_copy` 的目的位置迭代器必须指向未构造的内存。与 `copy` 不同，`uninitialized_copy` 在给定目的位置构造元素。

类似 `copy`，`uninitialized_copy` 返回（递增后的）目的位置迭代器。因此，一次 `uninitialized_copy` 调用会返回一个指针，指向最后一个构造的元素之后的位置。在本例中，我们将此指针保存在 `q` 中，然后将 `q` 传递给 `uninitialized_fill_n`。此函数类似 `fill_n`（参见 10.2.2 节，第 340 页），接受一个指向目的位置的指针、一个计数和一个值。它会在目的位置指针指向的内存中创建给定数目个对象，用给定值对它们进行初始化。

12.2.2 节练习

练习 12.26: 用 allocator 重写第 427 页中的程序。



12.3 使用标准库: 文本查询程序

我们将实现一个简单的文本查询程序, 作为标准库相关内容学习的总结。我们的程序允许用户在一个给定文件中查询单词。查询结果是单词在文件中出现的次数及其所在行的列表。如果一个单词在一行中出现多次, 此行只列出一次。行会按照升序输出——即, 第 7 行会在第 9 行之前显示, 依此类推。

例如, 我们可能读入一个包含本章内容 (指英文版中的文本) 的文件, 在其中寻找单词 element。输出结果的前几行应该是这样的:

```
element occurs 112 times
  (line 36) A set element contains only a key;
  (line 158) operator creates a new element
  (line 160) Regardless of whether the element
  (line 168) When we fetch an element from a map, we
  (line 214) If the element is not found, find returns
```

接下来还有大约 100 行, 都是单词 element 出现的位置。



12.3.1 文本查询程序设计

485

开始一个程序的设计的一种好方法是列出程序的操作。了解需要哪些操作会帮助我们分析出需要什么样的数据结构。从需求入手, 我们的文本查询程序需要完成如下任务:

- 当程序读取输入文件时, 它必须记住单词出现的每一行。因此, 程序需要逐行读取输入文件, 并将每一行分解为独立的单词
- 当程序生成输出时,
 - 它必须能提取每个单词所关联的行号
 - 行号必须按升序出现且无重复
 - 它必须能打印给定行号中的文本。

利用多种标准库设施, 我们可以很漂亮地实现这些要求:

- 我们将使用一个 `vector<string>` 来保存整个输入文件的一份拷贝。输入文件中的每行保存为 `vector` 中的一个元素。当需要打印一行时, 可以用行号作为下标来提取行文本。
- 我们使用一个 `istringstream` (参见 8.3 节, 第 287 页) 来将每行分解为单词。
- 我们使用一个 `set` 来保存每个单词在输入文本中出现的行号。这保证了每行只出现一次且行号按升序保存。
- 我们使用一个 `map` 来将每个单词与它出现的行号 `set` 关联起来。这样我们就可以方便地提取任意单词的 `set`。

我们的解决方案还使用了 `shared_ptr`, 原因稍后进行解释。

数据结构

虽然我们可以用 `vector`、`set` 和 `map` 来直接编写文本查询程序, 但如果定义一个更

为抽象的解决方案，会更为有效。我们将从定义一个保存输入文件的类开始，这会令文件查询更为容易。我们将这个类命名为 `TextQuery`，它包含一个 `vector` 和一个 `map`。`vector` 用来保存输入文件的文本，`map` 用来关联每个单词和它出现的行号的 `set`。这个类将会有个用来读取给定输入文件的构造函数和一个执行查询的操作。

查询操作要完成的任务非常简单：查找 `map` 成员，检查给定单词是否出现。设计这个函数的难点是确定应该返回什么内容。一旦找到了一个单词，我们需要知道它出现了多少次、它出现的行号以及每行的文本。

返回所有这些内容的最简单的方法是定义另一个类，可以命名为 `QueryResult`，来保存查询结果。这个类会有一个 `print` 函数，完成结果打印工作。

在类之间共享数据

< 486

我们的 `QueryResult` 类要表达查询的结果。这些结果包括与给定单词关联的行号的 `set` 和这些行对应的文本。这些数据都保存在 `TextQuery` 类型的对象中。

由于 `QueryResult` 所需要的数据都保存在一个 `TextQuery` 对象中，我们就必须确定如何访问它们。我们可以拷贝行号的 `set`，但这样做可能很耗时。而且，我们当然不希望拷贝 `vector`，因为这可能会引起整个文件的拷贝，而目标只不过是为了打印文件的一小部分而已（通常会是这样）。

通过返回指向 `TextQuery` 对象内部的迭代器（或指针），我们可以避免拷贝操作。但是，这种方法开启了一个陷阱：如果 `TextQuery` 对象在对应的 `QueryResult` 对象之前被销毁，会发生什么？在此情况下，`QueryResult` 就将引用一个不再存在的对象中的数据。

对于 `QueryResult` 对象和对应的 `TextQuery` 对象的生存期应该同步这一观察结果，其实已经暗示了问题的解决方案。考虑到这两个类概念上“共享”了数据，可以使用 `shared_ptr`（参见 12.1.1 节，第 400 页）来反映数据结构中的这种共享关系。

使用 `TextQuery` 类

当我们设计一个类时，在真正实现成员之前先编写程序使用这个类，是一种非常有用的方法。通过这种方法，可以看到类是否具有我们所需要的操作。例如，下面的程序使用了 `TextQuery` 和 `QueryResult` 类。这个函数接受一个指向要处理的文件的 `ifstream`，并与用户交互，打印给定单词的查询结果

```
void runQueries(ifstream &infile)
{
    // infile 是一个 ifstream，指向我们要处理的文件
    TextQuery tq(infile); // 保存文件并建立查询 map
    // 与用户交互：提示用户输入要查询的单词，完成查询并打印结果
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        // 若遇到文件尾或用户输入了'q'时循环终止
        if (!(cin >> s) || s == "q") break;
        // 指向查询并打印结果
        print(cout, tq.query(s)) << endl;
    }
}
```

我们首先用给定的 `ifstream` 初始化一个名为 `tq` 的 `TextQuery` 对象。`TextQuery` 的构造函数读取输入文件，保存在 `vector` 中，并建立单词到所在行号的 `map`。

487 while (无限) 循环提示用户输入一个要查询的单词，并打印出查询结果，如此往复。循环条件检测字面常量 `true` (参见 2.1.3 节，第 37 页)，因此永远成功。循环的退出是通过 `if` 语句中的 `break` (参见 5.5.1 节，第 170 页) 实现的。此 `if` 语句检查输入是否成功。如果成功，它再检查用户是否输入了 `q`。输入失败或用户输入了 `q` 都会使循环终止。一旦用户输入了要查询的单词，我们要求 `tq` 查找这个单词，然后调用 `print` 打印搜索结果。

12.3.1 节练习

练习 12.27: `TextQuery` 和 `QueryResult` 类只使用了我们已经介绍过的语言和标准库特性。不要提前看后续章节内容，只用已经学到的知识对这两个类编写你自己的版本。

练习 12.28: 编写程序实现文本查询，不要定义类来管理数据。你的程序应该接受一个文件，并与用户交互来查询单词。使用 `vector`、`map` 和 `set` 容器来保存来自文件的数据并生成查询结果。

练习 12.29: 我们曾经用 `do while` 循环来编写管理用户交互的循环 (参见 5.4.4 节，第 169 页)。用 `do while` 重写本节程序，解释你倾向于哪个版本，为什么。



12.3.2 文本查询程序类的定义

我们以 `TextQuery` 类的定义开始。用户创建此类的对象时会提供一个 `istream`，用来读取输入文件。这个类还提供一个 `query` 操作，接受一个 `string`，返回一个 `QueryResult` 表示 `string` 出现的那些行。

设计类的数据成员时，需要考虑与 `QueryResult` 对象共享数据的需求。`QueryResult` 类需要共享保存输入文件的 `vector` 和保存单词关联的行号的 `set`。因此，这个类应该有两个数据成员：一个指向动态分配的 `vector` (保存输入文件) 的 `shared_ptr` 和一个 `string` 到 `shared_ptr<set>` 的 `map`。`map` 将文件中每个单词关联到一个动态分配的 `set` 上，而此 `set` 保存了该单词所出现的行号。

为了使代码更易读，我们还会定义一个类型成员 (参见 7.3.1 节，第 243 页) 来引用行号，即 `string` 的 `vector` 中的下标：

```
class QueryResult; // 为了定义函数 query 的返回类型，这个定义是必需的
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
private:
    std::shared_ptr<std::vector<std::string>> file; // 输入文件
    // 每个单词到它所在的行号的集合的映射
    std::map<std::string,
            std::shared_ptr<std::set<line_no>>> wm;
};
```

488 这个类定义最困难的部分是解开类名。与往常一样，对于可能置于头文件中的代码，在使用标准库名字时要加上 `std::` (参见 3.1 节，第 74 页)。在本例中，我们反复使用了 `std::`，

使得代码开始可能有些难读。例如，

```
std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
```

如果写成下面的形式可能就更好理解一些

```
map<string, shared_ptr<set<line_no>>> wm;
```

TextQuery 构造函数

TextQuery 的构造函数接受一个 ifstream，逐行读取输入文件：

```
// 读取输入文件并建立单词到行号的映射
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
    string text;
    while (getline(is, text)) {           // 对文件中每一行
        file->push_back(text);          // 保存此行文本
        int n = file->size() - 1;        // 当前行号
        istringstream line(text);         // 将行文本分解为单词
        string word;
        while (line >> word) {          // 对行中每个单词
            // 如果单词不在 wm 中，以之为下标在 wm 中添加一项
            auto &lines = wm[word];      // lines 是一个 shared_ptr
            if (!lines) // 在我们第一次遇到这个单词时，此指针为空
                lines.reset(new set<line_no>); // 分配一个新的 set
            lines->insert(n);           // 将此行号插入 set 中
        }
    }
}
```

构造函数的初始化器分配一个新的 vector 来保存输入文件中的文本。我们用 getline 逐行读取输入文件，并存入 vector 中。由于 file 是一个 shared_ptr，我们用-> 运算符解引用 file 来提取 file 指向的 vector 对象的 push_back 成员。

接下来我们用一个 istringstream (参见 8.3 节，第 287 页) 来处理刚刚读入的一行中的每个单词。内层 while 循环用 istringstream 的输入运算符来从当前行读取每个单词，存入 word 中。在 while 循环内，我们用 map 下标运算符提取与 word 相关联的 shared_ptr<set>，并将 lines 绑定到此指针。注意，lines 是一个引用，因此改变 lines 也会改变 wm 中的元素。

若 word 不在 map 中，下标运算符会将 word 添加到 wm 中 (参见 11.3.4 节，第 387 页)，与 word 关联的值进行值初始化。这意味着，如果下标运算符将 word 添加到 wm 中，lines 将是一个空指针。如果 lines 为空，我们分配一个新的 set，并调用 reset 更新 lines 引用的 shared_ptr，使其指向这个新分配的 set。

不管是否创建了一个新的 set，我们都调用 insert 将当前行号添加到 set 中。由于 lines 是一个引用，对 insert 的调用会将新元素添加到 wm 中的 set 中。如果一个 [489] 给定单词在同一行中出现多次，对 insert 的调用什么都不会做。

QueryResult 类

QueryResult 类有三个数据成员：一个 string，保存查询单词；一个 shared_ptr，指向保存输入文件的 vector；一个 shared_ptr，指向保存单词出现行号的 set。它唯

一的一个成员函数是一个构造函数，初始化这三个数据成员：

```
class QueryResult {
    friend std::ostream& print(std::ostream&, const QueryResult&);
public:
    QueryResult(std::string s,
                std::shared_ptr<std::set<line_no>> p,
                std::shared_ptr<std::vector<std::string>> f):
        sought(s), lines(p), file(f) { }
private:
    std::string sought; // 查询单词
    std::shared_ptr<std::set<line_no>> lines;           // 出现的行号
    std::shared_ptr<std::vector<std::string>> file;       // 输入文件
};
```

构造函数的唯一工作是将参数保存在对应的数据成员中，这是在其初始化器列表中完成的（参见 7.1.4 节，第 237 页）。

query 函数

query 函数接受一个 string 参数，即查询单词，query 用它来在 map 中定位对应的行号 set。如果找到了这个 string，query 函数构造一个 QueryResult，保存给定 string、TextQuery 的 file 成员以及从 wm 中提取的 set。

唯一的问题是：如果给定 string 未找到，我们应该返回什么？在这种情况下，没有可返回的 set。为了解决此问题，我们定义了一个局部 static 对象，它是一个指向空的行号 set 的 shared_ptr。当未找到给定单词时，我们返回此对象的一个拷贝：

```
QueryResult
TextQuery::query(const string &sought) const
{
    // 如果未找到 sought，我们将返回一个指向此 set 的指针
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // 使用 find 而不是下标运算符来查找单词，避免将单词添加到 wm 中！
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // 未找到
    else
        return QueryResult(sought, loc->second, file);
}
```

490 打印结果

print 函数在给定的流上打印出给定的 QueryResult 对象：

```
ostream &print(ostream &os, const QueryResult &qr)
{
    // 如果找到了单词，打印出现次数和所有出现的位置
    os << qr.sought << " occurs " << qr.lines->size() << " "
        << make_plural(qr.lines->size(), "time", "s") << endl;
    // 打印单词出现的每一行
    for (auto num : *qr.lines) // 对 set 中每个单词
        // 避免行号从 0 开始给用户带来的困惑
        os << "\t(line " << num + 1 << ") "
```

```
    << * (qr.file->begin() + num) << endl;
return os;
}
```

我们调用 qr.lines 指向的 set 的 size 成员来报告单词出现了多少次。由于 set 是一个 shared_ptr，必须解引用 lines。调用 make_plural（参见 6.3.2 节，第 201 页）来根据大小是否等于 1 打印 time 或 times。

在 for 循环中，我们遍历 lines 所指向的 set。for 循环体打印行号，并按人们习惯的方式调整计数值。set 中的数值就是 vector 中元素的下标，从 0 开始编号。但大多数用户认为第一行的行号应该是 1，因此我们对每个行号都加上 1，转换为人们更习惯的形式。

我们用行号从 file 指向的 vector 中提取一行文本。回忆一下，当给一个迭代器加上一个数时，会得到 vector 中相应偏移之后位置的元素（参见 3.4.2 节，第 99 页）。因此，file->begin() + num 即为 file 指向的 vector 中第 num 个位置的元素。

注意此函数能正确处理未找到单词的情况。在此情况下，set 为空。第一条输出语句会注意到单词出现了 0 次。由于 *res.lines 为空，for 循环一次也不会执行。

12.3.2 节练习

练习 12.30: 定义你自己版本的 TextQuery 和 QueryResult 类，并执行 12.3.1 节（第 431 页）中的 runQueries 函数。

练习 12.31: 如果用 vector 代替 set 保存行号，会有什么差别？哪种方法更好？为什么？

练习 12.32: 重写 TextQuery 和 QueryResult 类，用 StrBlob 代替 vector<string> 保存输入文件。

练习 12.33: 在第 15 章中我们将扩展查询系统，在 QueryResult 类中将会需要一些额外的成员。添加名为 begin 和 end 的成员，返回一个迭代器，指向一个给定查询返回的行号的 set 中的位置。再添加一个名为 get_file 的成员，返回一个 shared_ptr，指向 QueryResult 对象中的文件。

491

小结

在 C++ 中，内存是通过 `new` 表达式分配，通过 `delete` 表达式释放的。标准库还定义了一个 `allocator` 类来分配动态内存块。

分配动态内存的程序应负责释放它所分配的内存。内存的正确释放是非常容易出错的地方：要么内存永远不会被释放，要么在仍有指针引用它时就被释放了。新的标准库定义了智能指针类型——`shared_ptr`、`unique_ptr` 和 `weak_ptr`，可令动态内存管理更为安全。对于一块内存，当没有任何用户使用它时，智能指针会自动释放它。现代 C++ 程序应尽可能使用智能指针。

术语表

allocator 标准库类，用来分配未构造的内存。

空悬指针（dangling pointer） 一个指针，指向曾经保存一个对象但现在已释放的内存。众所周知，空悬指针引起的程序错误非常难以调试。

delete 释放 `new` 分配的内存。`delete p` 释放对象，`delete []p` 释放 `p` 指向的数组。`p` 可以为空，或者指向 `new` 分配的内存。

释放器（deleter） 传递给智能指针的函数，用来代替 `delete` 释放指针绑定的对象。

析构函数（destructor） 特殊的成员函数，负责在对象离开作用域或被释放时完成清理工作。

动态分配的（dynamically allocated） 在自由空间中分配的对象。在自由空间中分配的对象直到被显式释放或程序结束才会销毁。

自由空间（free store） 程序可用的内存池，保存动态分配的对象。

堆（heap） 自由空间的同义词。

new 从自由空间分配内存。`new T` 分配并构造一个类型为 `T` 的对象，并返回一个指向该对象的指针。如果 `T` 是一个数组类型，`new` 返回一个指向数组首元素的指针。类似的，`new [n] T` 分配 `n` 个类型为 `T` 的对象，并返回指向数组首元素的指针。

默认情况下，分配的对象进行默认初始化。我们也可以提供可选的初始化器。

定位 new（placement new） 一种 `new` 表达式形式，接受一些额外的参数，在 `new` 关键字后面的括号中给出。例如，`new (nothrow) int` 告诉 `new` 不要抛出异常。

引用计数（reference count） 一个计数器，记录有多少用户共享一个对象。智能指针用它来判断什么时候释放所指向的对象是安全的。

shared_ptr 提供所有权共享的智能指针：对共享对象来说，当最后一个指向它的 `shared_ptr` 被销毁时会被释放。

智能指针（smart pointer） 标准库类型，行为类似指针，但可以检查什么时候使用指针是安全的。智能指针类型负责在恰当的时候释放内存。

unique_ptr 提供独享所有权的智能指针：当 `unique_ptr` 被销毁时，它指向的对象被释放。`unique_ptr` 不能直接拷贝或赋值。

weak_ptr 一种智能指针，指向由 `shared_ptr` 管理的对象。在确定是否应释放对象时，`shared_ptr` 并不把 `weak_ptr` 统计在内。

第III部分

类设计者的工具

内容

第 13 章 拷贝控制.....	439
第 14 章 操作重载与类型转换.....	489
第 15 章 面向对象程序设计.....	525
第 16 章 模板与泛型编程.....	577

类是 C++ 的核心概念。我们已经从第 7 章开始详细介绍了如何定义类。第 7 章涵盖了使用类的所有基本知识：类作用域、数据隐藏以及构造函数，还介绍了类的一些重要特性：成员函数、隐式 `this` 指针、友元以及 `const`、`static` 和 `mutable` 成员。在第 III 部分中，我们将延伸类的有关话题的讨论，将介绍拷贝控制、重载运算符、继承和模板。

如前所述，在 C++ 中，我们通过定义构造函数来控制在类类型的对象初始化时做什么。类还可以控制在对象拷贝、赋值、移动和销毁时做什么。在这方面，C++ 与其他语言是不同的，其他很多语言都没有给予类设计者控制这些操作的能力。第 13 章将介绍这些内容。本章还会介绍新标准引入的两个重要概念：右值引用和移动操作。

第 14 章介绍运算符重载，这种机制允许内置运算符作用于类类型的运算对象。这样，我们创建的类型直观上就可以像内置类型一样使用，运算符重载是 C++ 借以实现这一目的方法之一。

类可以重载的运算符中有一种特殊的运算符——函数调用运算符。对于重载了这种运算符的类，我们可以“调用”其对象，就好像它们是函数一样。新标准库中提供了一些设施，使得不同类型的可调用对象可以以一种一致的方式来使用，我们也将介绍这部分内容。

第 14 章最后将介绍另一种特殊类型的类成员函数——转换运算符。这些运算符定义了类类型对象的隐式转换机制。编译器应用这种转换机制的场合与原因都与内置类型转换是一样的。

第 III 部分的最后两章将介绍 C++ 如何支持面向对象编程和泛型编程。

第 15 章介绍继承和动态绑定。继承和动态绑定与数据抽象一起构成了面向对象编程的基础。继承令关联类型的定义更为简单，而动态绑定可以帮助我们编写类型无关的代码，可以忽略具有继承关系的类型之间的差异。

第 16 章介绍函数模板和类模板。模板可以让我们写出类型无关的通用类和函数。新标准引入了一些模板相关的新特性：可变参数模板、模板类型别名以及控制实例化的新方法。

编写我们自己的面向对象的或是泛型的类型需要对 C++ 有深刻的理解。幸运的是，我们无须掌握如何构建面向对象和泛型类型的细节也可以使用它们。例如，标准库中广泛使用了我们将在第 15 章和第 16 章中学习的技术，虽然我们已经使用过标准库类型和算法，但实际上我们并不了解它们是如何实现的。

因此，读者应该明白第 III 部分涉及的是相当深入的内容。编写模板或面向对象的类要求对 C++ 的基本知识和基本类的定义有着深刻的理解。

第 13 章

拷贝控制

内容

13.1 拷贝、赋值与销毁	440
13.2 拷贝控制和资源管理	452
13.3 交换操作	457
13.4 拷贝控制示例	460
13.5 动态内存管理类	464
13.6 对象移动	470
小结	486
术语表	486

如我们在第 7 章所见，每个类都定义了一个新类型和在此类型对象上可执行的操作。在本章中，我们还将学到，类可以定义构造函数，用来控制在创建此类型对象时做什么。

在本章中，我们还将学习类如何控制该类型对象拷贝、赋值、移动或销毁时做什么。类通过一些特殊的成员函数控制这些操作，包括：拷贝构造函数、移动构造函数、拷贝赋值运算符、移动赋值运算符以及析构函数。

496

当定义一个类时，我们显式地或隐式地指定在此类型的对象拷贝、移动、赋值和销毁时做什么。一个类通过定义五种特殊的成员函数来控制这些操作，包括：**拷贝构造函数**（copy constructor）、**拷贝赋值运算符**（copy-assignment operator）、**移动构造函数**（move constructor）、**移动赋值运算符**（move-assignment operator）和**析构函数**（destructor）。拷贝和移动构造函数定义了当用同类型的另一个对象初始化本对象时做什么。拷贝和移动赋值运算符定义了将一个对象赋予同类型的另一个对象时做什么。析构函数定义了当此类型对象销毁时做什么。我们称这些操作为**拷贝控制操作**（copy control）。

如果一个类没有定义所有这些拷贝控制成员，编译器会自动为它定义缺失的操作。因此，很多类会忽略这些拷贝控制操作（参见 7.1.5 节，第 239 页）。但是，对一些类来说，依赖这些操作的默认定义会导致灾难。通常，实现拷贝控制操作最困难的地方是首先认识到什么时候需要定义这些操作。



WARNING

在定义任何 C++ 类时，拷贝控制操作都是必要部分。对初学 C++ 的程序员来说，必须定义对象拷贝、移动、赋值或销毁时做什么，这常常令他们感到困惑。这种困扰很复杂，因为如果我们不显式定义这些操作，编译器也会为我们定义，但编译器定义的版本的行为可能并非我们所想。

13.1 拷贝、赋值与销毁

我们将以最基本的操作——拷贝构造函数、拷贝赋值运算符和析构函数作为开始。我们在 13.6 节（第 470 页）中将介绍移动操作（新标准所引入的操作）。



13.1.1 拷贝构造函数

如果一个构造函数的第一个参数是自身类类型的引用，且任何额外参数都有默认值，则此构造函数是拷贝构造函数。

```
class Foo {
public:
    Foo();           // 默认构造函数
    Foo(const Foo&); // 拷贝构造函数
    // ...
};
```

拷贝构造函数的第一个参数必须是一个引用类型，原因我们稍后解释。虽然我们可以定义一个接受非 const 引用的拷贝构造函数，但此参数几乎总是一个 const 的引用。拷贝构造函数在几种情况下都会被隐式地使用。因此，拷贝构造函数通常不应该是 explicit 的（参见 7.5.4 节，第 265 页）。

497

合成拷贝构造函数

如果我们没有为一个类定义拷贝构造函数，编译器会为我们定义一个。与合成默认构造函数（参见 7.1.4 节，第 235 页）不同，即使我们定义了其他构造函数，编译器也会为我们合成一个拷贝构造函数。

如我们将在 13.1.6 节（第 450 页）中所见，对某些类来说，**合成拷贝构造函数**（synthesized copy constructor）用来阻止我们拷贝该类类型的对象。而一般情况，合成的拷贝构造函数会将其参数的成员逐个拷贝到正在创建的对象中（参见 7.1.5 节，第 239 页）。编译器从给

定对象中依次将每个非 `static` 成员拷贝到正在创建的对象中。

每个成员的类型决定了它如何拷贝：对类类型的成员，会使用其拷贝构造函数来拷贝；内置类型的成员则直接拷贝。虽然我们不能直接拷贝一个数组（参见 3.5.1 节，第 102 页），但合成拷贝构造函数会逐元素地拷贝一个数组类型的成员。如果数组元素是类类型，则使用元素的拷贝构造函数来进行拷贝。

作为一个例子，我们的 `Sales_data` 类的合成拷贝构造函数等价于：

```
class Sales_data {
public:
    // 其他成员和构造函数的定义，如前
    // 与合成的拷贝构造函数等价的拷贝构造函数的声明
    Sales_data(const Sales_data&);

private:
    std::string bookNo;
    int units_sold = 0;
    double revenue = 0.0;
};

// 与 Sales_data 的合成的拷贝构造函数等价
Sales_data::Sales_data(const Sales_data &orig):
    bookNo(orig.bookNo),           // 使用 string 的拷贝构造函数
    units_sold(orig.units_sold),   // 拷贝 orig.units_sold
    revenue(orig.revenue)         // 拷贝 orig.revenue
{                                // 空函数体}
```

拷贝初始化

现在，我们可以完全理解直接初始化和拷贝初始化之间的差异了（参见 3.2.1 节，第 76 页）：

```
string dots(10, '.');           // 直接初始化
string s(dots);                // 直接初始化
string s2 = dots;               // 拷贝初始化
string null_book = "9-999-99999-9"; // 拷贝初始化
string nines = string(100, '9'); // 拷贝初始化
```

当使用直接初始化时，我们实际上是要求编译器使用普通的函数匹配（参见 6.4 节，第 209 页）来选择与我们提供的参数最匹配的构造函数。当我们使用 **拷贝初始化**（copy initialization）时，我们要求编译器将右侧运算对象拷贝到正在创建的对象中，如果需要的话还要进行类型转换（参见 7.5.4 节，第 263 页）。

拷贝初始化通常使用拷贝构造函数来完成。但是，如我们将在 13.6.2 节（第 473 页）◀ 498 所见，如果一个类有一个移动构造函数，则拷贝初始化有时会使用移动构造函数而非拷贝构造函数来完成。但现在，我们只需了解拷贝初始化何时发生，以及拷贝初始化是依靠拷贝构造函数或移动构造函数来完成的就可以了。

拷贝初始化不仅在我们用`=`定义变量时会发生，在下列情况下也会发生

- 将一个对象作为实参传递给一个非引用类型的形参
- 从一个返回类型为非引用类型的函数返回一个对象
- 用花括号列表初始化一个数组中的元素或一个聚合类中的成员（参见 7.5.5 节，第 266 页）

某些类类型还会对它们所分配的对象使用拷贝初始化。例如，当我们初始化标准库容器或是调用其 `insert` 或 `push` 成员（参见 9.3.1 节，第 306 页）时，容器会对其元素进行拷贝初始化。与之相对，用 `emplace` 成员创建的元素都进行直接初始化（参见 9.3.1 节，第 308 页）。

参数和返回值

在函数调用过程中，具有非引用类型的参数要进行拷贝初始化（参见 6.2.1 节，第 188 页）。类似的，当一个函数具有非引用的返回类型时，返回值会被用来初始化调用方的结果（参见 6.3.2 节，第 201 页）。

拷贝构造函数被用来初始化非引用类类型参数，这一特性解释了为什么拷贝构造函数自己的参数必须是引用类型。如果其参数不是引用类型，则调用永远也不会成功——为了调用拷贝构造函数，我们必须拷贝它的实参，但为了拷贝实参，我们又需要调用拷贝构造函数，如此无限循环。

拷贝初始化的限制

如前所述，如果我们使用的初始化值要求通过一个 `explicit` 的构造函数来进行类型转换（参见 7.5.4 节，第 265 页），那么使用拷贝初始化还是直接初始化就不是无关紧要的了：

```
vector<int> v1(10); // 正确：直接初始化
vector<int> v2 = 10; // 错误：接受大小参数的构造函数是 explicit 的
void f(vector<int>); // f 的参数进行拷贝初始化
f(10); // 错误：不能用一个 explicit 的构造函数拷贝一个实参
f(vector<int>(10)); // 正确：从一个 int 直接构造一个临时 vector
```

直接初始化 `v1` 是合法的，但看起来与之等价的拷贝初始化 `v2` 则是错误的，因为 `vector` 的接受单一大小参数的构造函数是 `explicit` 的。出于同样的原因，当传递一个实参或从函数返回一个值时，我们不能隐式使用一个 `explicit` 构造函数。如果我们希望使用一个 `explicit` 构造函数，就必须显式地使用，像此代码中最后一行那样。

499 > 编译器可以绕过拷贝构造函数

在拷贝初始化过程中，编译器可以（但不是必须）跳过拷贝/移动构造函数，直接创建对象。即，编译器被允许将下面的代码

```
string null_book = "9-999-99999-9"; // 拷贝初始化
```

改写为

```
string null_book("9-999-99999-9"); // 编译器略过了拷贝构造函数
```

但是，即使编译器略过了拷贝/移动构造函数，但在这个程序点上，拷贝/移动构造函数必须是存在且可访问的（例如，不能是 `private` 的）。

13.1.1 节练习

练习 13.1：拷贝构造函数是什么？什么时候使用它？

练习 13.2：解释为什么下面的声明是非法的：

```
Sales_data::Sales_data(Sales_data rhs);
```

练习 13.3: 当我们拷贝一个 StrBlob 时，会发生什么？拷贝一个 StrBlobPtr 呢？

练习 13.4: 假定 Point 是一个类类型，它有一个 public 的拷贝构造函数，指出下面程序片段中哪些地方使用了拷贝构造函数：

```
Point global;
Point foo_bar(Point arg)
{
    Point local = arg, *heap = new Point(global);
    *heap = local;
    Point pa[ 4 ] = { local, *heap };
    return *heap;
}
```

练习 13.5: 给定下面的类框架，编写一个拷贝构造函数，拷贝所有成员。你的构造函数应该动态分配一个新的 string（参见 12.1.2 节，第 407 页），并将对象拷贝到 ps 指向的位置，而不是 ps 本身的位置。

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
private:
    std::string *ps;
    int i;
};
```

13.1.2 拷贝赋值运算符

与类控制其对象如何初始化一样，类也可以控制其对象如何赋值：

```
Sales_data trans, accum;
trans = accum; // 使用 Sales_data 的拷贝赋值运算符
```

与拷贝构造函数一样，如果类未定义自己的拷贝赋值运算符，编译器会为它合成一个。

重载赋值运算符

在介绍合成赋值运算符之前，我们需要了解一点儿有关重载运算符（overloaded operator）的知识，详细内容将在第 14 章中进行介绍。

重载运算符本质上是函数，其名字由 operator 关键字后接表示要定义的运算符的符号组成。因此，赋值运算符就是一个名为 operator= 的函数。类似于任何其他函数，运算符函数也有一个返回类型和一个参数列表。

重载运算符的参数表示运算符的运算对象。某些运算符，包括赋值运算符，必须定义为成员函数。如果一个运算符是一个成员函数，其左侧运算对象就绑定到隐式的 this 参数（参见 7.1.2 节，第 231 页）。对于一个二元运算符，例如赋值运算符，其右侧运算对象作为显式参数传递。

拷贝赋值运算符接受一个与其所在类相同类型的参数：

```
class Foo {
public:
```

```
Foo& operator=(const Foo&); // 赋值运算符
// ...
};
```

为了与内置类型的赋值（参见 4.4 节，第 129 页）保持一致，赋值运算符通常返回一个指向其左侧运算对象的引用。另外值得注意的是，标准库通常要求保存在容器中的类型要具有赋值运算符，且其返回值是左侧运算对象的引用。

Best Practices

赋值运算符通常应该返回一个指向其左侧运算对象的引用。

合成拷贝赋值运算符

与处理拷贝构造函数一样，如果一个类未定义自己的拷贝赋值运算符，编译器会为它生成一个合成拷贝赋值运算符（synthesized copy-assignment operator）。类似拷贝构造函数，对于某些类，合成拷贝赋值运算符用来禁止该类型对象的赋值（参见 13.1.6 节，第 450 页）。如果拷贝赋值运算符并非出于此目的，它会将右侧运算对象的每个非 static 成员赋予左侧运算对象的对应成员，这一工作是通过成员类型的拷贝赋值运算符来完成的。对于数组类型的成员，逐个赋值数组元素。合成拷贝赋值运算符返回一个指向其左侧运算对象的引用。

501

作为一个例子，下面的代码等价于 Sales_data 的合成拷贝赋值运算符：

```
// 等价于合成拷贝赋值运算符
Sales_data&
Sales_data::operator=(const Sales_data &rhs)
{
    bookNo = rhs.bookNo;           // 调用 string::operator=
    units_sold = rhs.units_sold;  // 使用内置的 int 赋值
    revenue = rhs.revenue;        // 使用内置的 double 赋值
    return *this;                 // 返回一个此对象的引用
}
```

13.1.2 节练习

练习 13.6：拷贝赋值运算符是什么？什么时候使用它？合成拷贝赋值运算符完成什么工作？什么时候会生成合成拷贝赋值运算符？

练习 13.7：当我们将一个 StrBlob 赋值给另一个 StrBlob 时，会发生什么？赋值 StrBlobPtr 呢？

练习 13.8：为 13.1.1 节（第 443 页）练习 13.5 中的 HasPtr 类编写赋值运算符。类似拷贝构造函数，你的赋值运算符应该将对象拷贝到 ps 指向的位置。



13.1.3 析构函数

析构函数执行与构造函数相反的操作：构造函数初始化对象的非 static 数据成员，还可能做一些其他工作；析构函数释放对象使用的资源，并销毁对象的非 static 数据成员。

析构函数是类的一个成员函数，名字由波浪号接类名构成。它没有返回值，也不接受参数：

```
class Foo {
public:
    ~Foo(); // 析构函数
```

```
//...  
};
```

由于析构函数不接受参数，因此它不能被重载。对一个给定类，只会有唯一一个析构函数。

析构函数完成什么工作

如同构造函数有一个初始化部分和一个函数体（参见 7.5.1 节，第 257 页），析构函数也有一个函数体和一个析构部分。在一个构造函数中，成员的初始化是在函数体执行之前完成的，且按照它们在类中出现的顺序进行初始化。在一个析构函数中，首先执行函数体，502然后销毁成员。成员按初始化顺序的逆序销毁。

在对象最后一次使用之后，析构函数的函数体可执行类设计者希望执行的任何收尾工作。通常，析构函数释放对象在生存期分配的所有资源。

在一个析构函数中，不存在类似构造函数中初始化列表的东西来控制成员如何销毁，析构部分是隐式的。成员销毁时发生什么完全依赖于成员的类型。销毁类类型的成员需要执行成员自己的析构函数。内置类型没有析构函数，因此销毁内置类型成员什么也不需要做。



隐式销毁一个内置指针类型的成员不会 `delete` 它所指向的对象。

与普通指针不同，智能指针（参见 12.1.1 节，第 402 页）是类类型，所以具有析构函数。因此，与普通指针不同，智能指针成员在析构阶段会被自动销毁。

什么时候会调用析构函数

无论何时一个对象被销毁，就会自动调用其析构函数：

- 变量在离开其作用域时被销毁。
- 当一个对象被销毁时，其成员被销毁。
- 容器（无论是标准库容器还是数组）被销毁时，其元素被销毁。
- 对于动态分配的对象，当对指向它的指针应用 `delete` 运算符时被销毁（参见 12.1.2 节，第 409 页）。
- 对于临时对象，当创建它的完整表达式结束时被销毁。

由于析构函数自动运行，我们的程序可以按需要分配资源，而（通常）无须担心何时释放这些资源。

例如，下面代码片段定义了四个 `Sales_data` 对象：

```
{ // 新作用域  
    // p 和 p2 指向动态分配的对象  
    Sales_data *p = new Sales_data;           // p 是一个内置指针  
    auto p2 = make_shared<Sales_data>();      // p2 是一个 shared_ptr  
    Sales_data item(*p);                      // 拷贝构造函数将*p 拷贝到 item 中  
    vector<Sales_data> vec;                   // 局部对象  
    vec.push_back(*p2);                       // 拷贝 p2 指向的对象  
    delete p;                                // 对 p 指向的对象执行析构函数  
} // 退出局部作用域；对 item、p2 和 vec 调用析构函数  
// 销毁 p2 会递减其引用计数；如果引用计数变为 0，对象被释放  
// 销毁 vec 会销毁它的元素
```

503 每个 Sales_data 对象都包含一个 string 成员，它分配动态内存来保存 bookNo 成员中的字符。但是，我们的代码唯一需要直接管理的内存就是我们直接分配的 Sales_data 对象。我们的代码只需直接释放绑定到 p 的动态分配对象。

其他 Sales_data 对象会在离开作用域时被自动销毁。当程序块结束时，vec、p2 和 item 都离开了作用域，意味着在这些对象上分别会执行 vector、shared_ptr 和 Sales_data 的析构函数。vector 的析构函数会销毁我们添加到 vec 的元素。shared_ptr 的析构函数会递减 p2 指向的对象的引用计数。在本例中，引用计数会变为 0，因此 shared_ptr 的析构函数会 delete p2 分配的 Sales_data 对象。

在所有情况下，Sales_data 的析构函数都会隐式地销毁 bookNo 成员。销毁 bookNo 会调用 string 的析构函数，它会释放用来保存 ISBN 的内存。



当指向一个对象的引用或指针离开作用域时，析构函数不会执行。

合成析构函数

当一个类未定义自己的析构函数时，编译器会为它定义一个合成析构函数（synthesized destructor）。类似拷贝构造函数和拷贝赋值运算符，对于某些类，合成析构函数被用来阻止该类型的对象被销毁（参见 13.1.6 节，第 450 页）。如果不是这种情况，合成析构函数的函数体就为空。

例如，下面的代码片段等价于 Sales_data 的合成析构函数：

```
class Sales_data {
public:
    // 成员会被自动销毁，除此之外不需要做其他事情
    ~Sales_data() { }
    // 其他成员的定义，如前
};
```

在（空）析构函数体执行完毕后，成员会被自动销毁。特别的，string 的析构函数会被调用，它将释放 bookNo 成员所用的内存。

认识到析构函数体自身并不直接销毁成员是非常重要的。成员是在析构函数体之后隐含的析构阶段中被销毁的。在整个对象销毁过程中，析构函数体是作为成员销毁步骤之外的另一部分而进行的。

13.1.3 节练习

练习 13.9：析构函数是什么？合成析构函数完成什么工作？什么时候会生成合成析构函数？

练习 13.10：当一个 StrBlob 对象销毁时会发生什么？一个 StrBlobPtr 对象销毁时呢？

练习 13.11：为前面练习中的 HasPtr 类添加一个析构函数。

练习 13.12：在下面的代码片段中会发生几次析构函数调用？

```
bool fcn(const Sales_data *trans, Sales_data accum)
{
    Sales_data item1(*trans), item2(accum);
```

```

        return item1.isbn() != item2.isbn();
    }
}

```

练习 13.13：理解拷贝控制成员和构造函数的一个好方法是定义一个简单的类，为该类定义这些成员，每个成员都打印出自己的名字：

```

struct X {
    X() {std::cout << "X()" << std::endl;}
    X(const X&) {std::cout << "X(const X&)" << std::endl;}
};

```

给 `X` 添加拷贝赋值运算符和析构函数，并编写一个程序以不同方式使用 `X` 的对象：将它们作为非引用和引用参数传递；动态分配它们；将它们存放于容器中；诸如此类。观察程序的输出，直到你确认理解了什么时候会使用拷贝控制成员，以及为什么会使用它们。当你观察程序输出时，记住编译器可以略过对拷贝构造函数的调用。

13.1.4 三/五法则



如前所述，有三个基本操作可以控制类的拷贝操作：拷贝构造函数、拷贝赋值运算符和析构函数。而且，在新标准下，一个类还可以定义一个移动构造函数和一个移动赋值运算符，我们将在 13.6 节（第 470 页）中介绍这些内容。

C++语言并不要求我们定义所有这些操作：可以只定义其中一个或两个，而不必定义所有。但是，这些操作通常应该被看作一个整体。通常，只需要其中一个操作，而不需要定义所有操作的情况是很少见的。

< 504

需要析构函数的类也需要拷贝和赋值操作

当我们决定一个类是否要定义它自己版本的拷贝控制成员时，一个基本原则是首先确定这个类是否需要一个析构函数。通常，对析构函数的需求要比对拷贝构造函数或赋值运算符的需求更为明显。如果这个类需要一个析构函数，我们几乎可以肯定它也需要一个拷贝构造函数和一个拷贝赋值运算符。

我们在练习中用过的 `HasPtr` 类是一个好例子（参见 13.1.1 节，第 443 页）。这个类在构造函数中分配动态内存。合成析构函数不会 `delete` 一个指针数据成员。因此，此类需要定义一个析构函数来释放构造函数分配的内存。

应该怎么做可能还有点儿不清晰，但基本原则告诉我们，`HasPtr` 也需要一个拷贝构造函数和一个拷贝赋值运算符。

< 505

如果我们为 `HasPtr` 定义一个析构函数，但使用合成版本的拷贝构造函数和拷贝赋值运算符，考虑会发生什么：

```

class HasPtr {
public:
    HasPtr(const std::string &s = std::string()): ps(new std::string(s)), i(0) { }
    ~HasPtr() { delete ps; }
    // 错误：HasPtr 需要一个拷贝构造函数和一个拷贝赋值运算符
    // 其他成员的定义，如前
};

```

在这个版本的类定义中，构造函数中分配的内存将在 `HasPtr` 对象销毁时被释放。但不幸的是，我们引入了一个严重的错误！这个版本的类使用了合成的拷贝构造函数和拷贝

赋值运算符。这些函数简单拷贝指针成员，这意味着多个 HasPtr 对象可能指向相同的内存：

```
HasPtr f(HasPtr hp)           // HasPtr 是传值参数，所以将被拷贝
{
    HasPtr ret = hp;          // 拷贝给定的 HasPtr
    // 处理 ret
    return ret;               // ret 和 hp 被销毁
}
```

当 `f` 返回时，`hp` 和 `ret` 都被销毁，在两个对象上都会调用 `HasPtr` 的析构函数。此析构函数会 `delete` `ret` 和 `hp` 中的指针成员。但这两个对象包含相同的指针值。此代码会导致此指针被 `delete` 两次，这显然是一个错误（参见 12.1.2 节，第 411 页）。将要发生什么是未定义的。

此外，`f` 的调用者还会使用传递给 `f` 的对象：

```
HasPtr p("some values");
f(p);                      // 当 f 结束时，p.ps 指向的内存被释放
HasPtr q(p);                // 现在 p 和 q 都指向无效内存！
```

`p`（以及 `q`）指向的内存不再有效，在 `hp`（或 `ret`）销毁时它就被归还给系统了。



如果一个类需要自定义析构函数，几乎可以肯定它也需要自定义拷贝赋值运算符和拷贝构造函数。

需要拷贝操作的类也需要赋值操作，反之亦然

虽然很多类需要定义所有（或是不需要定义任何）拷贝控制成员，但某些类所要完成的工作，只需要拷贝或赋值操作，不需要析构函数。

作为一个例子，考虑一个类为每个对象分配一个独有的、唯一的序号。这个类需要一个拷贝构造函数为每个新创建的对象生成一个新的、独一无二的序号。除此之外，这个拷贝构造函数从给定对象拷贝所有其他数据成员。这个类还需要自定义拷贝赋值运算符来避免将序号赋予目的对象。但是，这个类不需要自定义析构函数。

这个例子引出了第二个基本原则：如果一个类需要一个拷贝构造函数，几乎可以肯定它也需要一个拷贝赋值运算符。反之亦然——如果一个类需要一个拷贝赋值运算符，几乎可以肯定它也需要一个拷贝构造函数。然而，无论是需要拷贝构造函数还是需要拷贝赋值运算符都不必然意味着也需要析构函数。

13.1.4 节练习

练习 13.14：假定 `numbered` 是一个类，它有一个默认构造函数，能为每个对象生成一个唯一的序号，保存在名为 `mysn` 的数据成员中。假定 `numbered` 使用合成的拷贝控制成员，并给定如下函数：

```
void f (numbered s) { cout << s.mysn << endl; }
```

则下面代码输出什么内容？

```
numbered a, b = a, c = b;
f(a); f(b); f(c);
```

练习 13.15：假定 `numbered` 定义了一个拷贝构造函数，能生成一个新的序号。这会改变上一题中调用的输出结果吗？如果会改变，为什么？新的输出结果是什么？

练习 13.16: 如果 `f` 中的参数是 `const numbered&`, 将会怎样? 这会改变输出结果吗? 如果会改变, 为什么? 新的输出结果是什么?

练习 13.17: 分别编写前三题中所描述的 `numbered` 和 `f`, 验证你是否正确预测了输出结果。

13.1.5 使用`=default`

我们可以通过将拷贝控制成员定义为`=default` 来显式地要求编译器生成合成的版本 (参见 7.1.4 节, 第 237 页):

```
class Sales_data {
public:
    // 拷贝控制成员; 使用 default
    Sales_data() = default;
    Sales_data(const Sales_data&) = default;
    Sales_data& operator=(const Sales_data &);
    ~Sales_data() = default;
    // 其他成员的定义, 如前
};

Sales_data& Sales_data::operator=(const Sales_data&) = default;
```

当我们在类内用`=default` 修饰成员的声明时, 合成的函数将隐式地声明为内联的 (就像任何其他类内声明的成员函数一样)。如果我们不希望合成的成员是内联函数, 应该只对成员的类外定义使用`=default`, 就像对拷贝赋值运算符所做的那样。



我们只能对具有合成版本的成员函数使用`=default` (即, 默认构造函数或拷贝控制成员)。

13.1.6 阻止拷贝



大多数类应该定义默认构造函数、拷贝构造函数和拷贝赋值运算符, 无论是隐式地还是显式地。

虽然大多数类应该定义 (而且通常也的确定义了) 拷贝构造函数和拷贝赋值运算符, 但对某些类来说, 这些操作没有合理的意义。在此情况下, 定义类时必须采用某种机制阻止拷贝或赋值。例如, `iostream` 类阻止了拷贝, 以避免多个对象写入或读取相同的 IO 缓冲。为了阻止拷贝, 看起来可能应该不定义拷贝控制成员。但是, 这种策略是无效的: 如果我们的类未定义这些操作, 编译器为它生成合成的版本。

定义删除的函数

在新标准下, 我们可以通过将拷贝构造函数和拷贝赋值运算符定义为删除的函数 (deleted function) 来阻止拷贝。删除的函数是这样一种函数: 我们虽然声明了它们, 但不能以任何方式使用它们。在函数的参数列表后面加上`=delete` 来指出我们希望将它定义为删除的:

```
struct NoCopy {
    NoCopy() = default;           // 使用合成的默认构造函数
    NoCopy(const NoCopy&) = delete;        // 阻止拷贝
    NoCopy &operator=(const NoCopy&) = delete;    // 阻止赋值
```

```

~NoCopy() = default;      // 使用合成的析构函数
// 其他成员
};

=delete 通知编译器（以及我们代码的读者），我们不希望定义这些成员。

```

与`=default`不同，`=delete`必须出现在函数第一次声明的时候，这个差异与这些声明的含义在逻辑上是吻合的。一个默认的成员只影响为这个成员而生成的代码，因此`=default`直到编译器生成代码时才需要。而另一方面，编译器需要知道一个函数是删除的，以便禁止试图使用它的操作。

与`=default`的另一个不同之处是，我们可以对任何函数指定`=delete`（我们只能对编译器可以合成的默认构造函数或拷贝控制成员使用`=default`）。虽然删除函数的主要用途是禁止拷贝控制成员，但当我们希望引导函数匹配过程时，删除函数有时也是有用的。

析构函数不能是删除的成员

值得注意的是，我们不能删除析构函数。如果析构函数被删除，就无法销毁此类型的对象了。对于一个删除了析构函数的类型，编译器将不允许定义该类型的变量或创建该类的临时对象。而且，如果一个类有某个成员的类型删除了析构函数，我们也不能定义该类的变量或临时对象。因为如果一个成员的析构函数是删除的，则该成员无法被销毁。而如果一个成员无法被销毁，则对象整体也就无法被销毁了。

对于删除了析构函数的类型，虽然我们不能定义这种类型的变量或成员，但可以动态分配这种类型的对象。但是，不能释放这些对象：

```

struct NoDtor {
    NoDtor() = default; // 使用合成默认构造函数
    ~NoDtor() = delete; // 我们不能销毁 NoDtor 类型的对象
};

NoDtor nd; // 错误：NoDtor 的析构函数是删除的
NoDtor *p = new NoDtor(); // 正确：但我们不能 delete p
delete p; // 错误：NoDtor 的析构函数是删除的

```



对于析构函数已删除的类型，不能定义该类型的变量或释放指向该类型动态分配对象的指针。

合成的拷贝控制成员可能是删除的

如前所述，如果我们未定义拷贝控制成员，编译器会为我们定义合成的版本。类似的，如果一个类未定义构造函数，编译器会为其合成一个默认构造函数（参见 7.1.4 节，第 235 页）。对某些类来说，编译器将这些合成的成员定义为删除的函数：

- 如果类的某个成员的析构函数是删除的或不可访问的（例如，是 `private` 的），则类的合成析构函数被定义为删除的。
- 如果类的某个成员的拷贝构造函数是删除的或不可访问的，则类的合成拷贝构造函数被定义为删除的。如果类的某个成员的析构函数是删除的或不可访问的，则类合成的拷贝构造函数也被定义为删除的。
- 如果类的某个成员的拷贝赋值运算符是删除的或不可访问的，或是类有一个 `const` 的或引用成员，则类的合成拷贝赋值运算符被定义为删除的。
- 如果类的某个成员的析构函数是删除的或不可访问的，或是类有一个引用成员，它没有类内初始化器（参见 2.6.1 节，第 65 页），或是类有一个 `const` 成员，它没有

类内初始化器且其类型未显式定义默认构造函数，则该类的默认构造函数被定义为删除的。

本质上，这些规则的含义是：如果一个类有数据成员不能默认构造、拷贝、复制或销毁，509则对应的成员函数将被定义为删除的。

一个成员有删除的或不可访问的析构函数会导致合成的默认和拷贝构造函数被定义为删除的，这看起来可能有些奇怪。其原因是，如果没有这条规则，我们可能会创建出无法销毁的对象。

对于具有引用成员或无法默认构造的 `const` 成员的类，编译器不会为其合成默认构造函数，这应该不奇怪。同样不出人意料的规则是：如果一个类有 `const` 成员，则它不能使用合成的拷贝赋值运算符。毕竟，此运算符试图赋值所有成员，而将一个新值赋予一个 `const` 对象是不可能的。

虽然我们可以将一个新值赋予一个引用成员，但这样做改变的是引用指向的对象的值，而不是引用本身。如果为这样的类合成拷贝赋值运算符，则赋值后，左侧运算对象仍然指向与赋值前一样的对象，而不会与右侧运算对象指向相同的对象。由于这种行为看起来并不是我们所期望的，因此对于有引用成员的类，合成拷贝赋值运算符被定义为删除的。

我们将在 13.6.2 节（第 476 页）、15.7.2 节（第 553 页）及 19.6 节（第 751 页）中介绍导致类的拷贝控制成员被定义为删除函数的其他原因。



本质上，当不可能拷贝、赋值或销毁类的成员时，类的合成拷贝控制成员就被定义为删除的。

private 拷贝控制

在新标准发布之前，类是通过将其拷贝构造函数和拷贝赋值运算符声明为 `private` 的来阻止拷贝：

```
class PrivateCopy {  
    // 无访问说明符；接下来的成员默认为 private 的；参见 7.2 节（第 240 页）  
    // 拷贝控制成员是 private 的，因此普通用户代码无法访问  
    PrivateCopy(const PrivateCopy&);  
    PrivateCopy &operator=(const PrivateCopy&);  
    // 其他成员  
  
public:  
    PrivateCopy() = default; // 使用合成的默认构造函数  
    ~PrivateCopy(); // 用户可以定义此类型的对象，但无法拷贝它们  
};
```

由于析构函数是 `public` 的，用户可以定义 `PrivateCopy` 类型的对象。但是，由于拷贝构造函数和拷贝赋值运算符是 `private` 的，用户代码将不能拷贝这个类型的对象。但是，友元和成员函数仍旧可以拷贝对象。为了阻止友元和成员函数进行拷贝，我们将这些拷贝控制成员声明为 `private` 的，但并不定义它们。

声明但不定义一个成员函数是合法的（参见 6.1.2 节，第 186 页），对此只有一个例外，我们将在 15.2.1 节（第 528 页）中介绍。试图访问一个未定义的成员将导致一个链接时错误。通过声明（但不定义）`private` 的拷贝构造函数，我们可以预先阻止任何拷贝该类型对象的企图：试图拷贝对象的用户代码将在编译阶段被标记为错误；成员函数或友元函数中的拷贝操作将会导致链接时错误。510



希望阻止拷贝的类应该使用`=delete` 来定义它们自己的拷贝构造函数和拷贝赋值运算符，而不应该将它们声明为 `private` 的。

13.1.6 节练习

练习 13.18: 定义一个 `Employee` 类，它包含雇员的姓名和唯一的雇员证号。为这个类定义默认构造函数，以及接受一个表示雇员姓名的 `string` 的构造函数。每个构造函数应该通过递增一个 `static` 数据成员来生成一个唯一的证号。

练习 13.19: 你的 `Employee` 类需要定义它自己的拷贝控制成员吗？如果需要，为什么？如果不呢，为什么？实现你认为 `Employee` 需要的拷贝控制成员。

练习 13.20: 解释当我们拷贝、赋值或销毁 `TextQuery` 和 `QueryResult` 类（参见 12.3 节，第 430 页）对象时会发生什么。

练习 13.21: 你认为 `TextQuery` 和 `QueryResult` 类需要定义它们自己版本的拷贝控制成员吗？如果需要，为什么？如果不呢，为什么？实现你认为这两个类需要的拷贝控制操作。



13.2 拷贝控制和资源管理

通常，管理类外资源的类必须定义拷贝控制成员。如我们在 13.1.4 节（第 447 页）中所见，这种类需要通过析构函数来释放对象所分配的资源。一旦一个类需要析构函数，那么它几乎肯定也需要一个拷贝构造函数和一个拷贝赋值运算符。

为了定义这些成员，我们首先必须确定此类型对象的拷贝语义。一般来说，有两种选择：可以定义拷贝操作，使类的行为看起来像一个值或者像一个指针。

类的行为像一个值，意味着它应该也有自己的状态。当我们拷贝一个像值的对象时，副本和原对象是完全独立的。改变副本不会对原对象有任何影响，反之亦然。

行为像指针的类则共享状态。当我们拷贝一个这种类的对象时，副本和原对象使用相同的底层数据。改变副本也会改变原对象，反之亦然。

在我们使用过的标准库类中，标准库容器和 `string` 类的行为像一个值。而不出意外的，`shared_ptr` 类提供类似指针的行为，就像我们的 `StrBlob` 类（参见 12.1.1 节，第 405 页）一样，`IO` 类型和 `unique_ptr` 不允许拷贝或赋值，因此它们的行为既不像值也不像指针。

为了说明这两种方式，我们会为练习中的 `HasPtr` 类定义拷贝控制成员。首先，我们将令类的行为像一个值；然后重新实现类，使它的行为像一个指针。

我们的 `HasPtr` 类有两个成员，一个 `int` 和一个 `string` 指针。通常，类直接拷贝内置类型（不包括指针）成员；这些成员本身就是值，因此通常应该让它们的行为像值一样。我们如何拷贝指针成员决定了像 `HasPtr` 这样的类是具有类值行为还是类指针行为。

13.2 节练习

练习 13.22: 假定我们希望 `HasPtr` 的行为像一个值。即，对于对象所指向的 `string`

成员，每个对象都有一份自己的拷贝。我们将在下一节介绍拷贝控制成员的定义。但是，你已经学习了定义这些成员所需的所有知识。在继续学习下一节之前，为 HasPtr 编写拷贝构造函数和拷贝赋值运算符。

13.2.1 行为像值的类



为了提供类值的行为，对于类管理的资源，每个对象都应该拥有一份自己的拷贝。这意味着对于 ps 指向的 string，每个 HasPtr 对象都必须有自己的拷贝。为了实现类值行为，HasPtr 需要

- 定义一个拷贝构造函数，完成 string 的拷贝，而不是拷贝指针
- 定义一个析构函数来释放 string
- 定义一个拷贝赋值运算符来释放对象当前的 string，并从右侧运算对象拷贝 string

类值版本的 HasPtr 如下所示

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
    // 对 ps 指向的 string，每个 HasPtr 对象都有自己的拷贝
    HasPtr(const HasPtr &p):
        ps(new std::string(*p.ps)), i(p.i) { }
    HasPtr& operator=(const HasPtr &);

    ~HasPtr() { delete ps; }

private:
    std::string *ps;
    int      i;
};
```

我们的类足够简单，在类内就已定义了除赋值运算符之外的所有成员函数。第一个构造函数接受一个（可选的）string 参数。这个构造函数动态分配它自己的 string 副本，并将指向 string 的指针保存在 ps 中。拷贝构造函数也分配它自己的 string 副本。析构函数对指针成员 ps 执行 delete，释放构造函数中分配的内存。

<512

类值拷贝赋值运算符

赋值运算符通常组合了析构函数和构造函数的操作。类似析构函数，赋值操作会销毁左侧运算对象的资源。类似拷贝构造函数，赋值操作会从右侧运算对象拷贝数据。但是，非常重要的一点是，这些操作是以正确的顺序执行的，即使将一个对象赋予它自身，也保证正确。而且，如果可能，我们编写的赋值运算符还应该是异常安全的——当异常发生时能将左侧运算对象置于一个有意义的状态（参见 5.6.2 节，第 175 页）。

在本例中，通过先拷贝右侧运算对象，我们可以处理自赋值情况，并能保证在异常发生时代码也是安全的。在完成拷贝后，我们释放左侧运算对象的资源，并更新指针指向新分配的 string：

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    auto newp = new string(*rhs.ps); // 拷贝底层 string
    delete ps; // 释放旧内存
```

```

    ps = newp;           // 从右侧运算对象拷贝数据到本对象
    i = rhs.i;
    return *this;        // 返回本对象
}

```

在这个赋值运算符中，非常清楚，我们首先进行了构造函数的工作：newp 的初始化器等价于 HasPtr 的拷贝构造函数中 ps 的初始化器。接下来与析构函数一样，我们 delete 当前 ps 指向的 string。然后就只剩下拷贝指向新分配的 string 的指针，以及从 rhs 拷贝 int 值到本对象了。

关键概念：赋值运算符

当你编写赋值运算符时，有两点需要记住：

- 如果将一个对象赋予它自身，赋值运算符必须能正确工作。
- 大多数赋值运算符组合了析构函数和拷贝构造函数的工作。

当你编写一个赋值运算符时，一个好的模式是先将右侧运算对象拷贝到一个局部临时对象中。当拷贝完成后，销毁左侧运算对象的现有成员就是安全的了。一旦左侧运算对象的资源被销毁，就只剩下将数据从临时对象拷贝到左侧运算对象的成员中了。

513 >

为了说明防范自赋值操作的重要性，考虑如果赋值运算符如下编写将会发生什么

```

// 这样编写赋值运算符是错误的!
HasPtr&
HasPtr::operator=(const HasPtr &rhs)
{
    delete ps; // 释放对象指向的 string
    // 如果 rhs 和*this 是同一个对象，我们就将从已释放的内存中拷贝数据!
    ps = new string(*(rhs.ps));
    i = rhs.i;
    return *this;
}

```

如果 rhs 和本对象是同一个对象，delete ps 会释放*this 和 rhs 指向的 string。接下来，当我们在 new 表达式中试图拷贝*(rhs.ps) 时，就会访问一个指向无效内存的指针，其行为和结果是未定义的。



对于一个赋值运算符来说，正确工作是非常重要的，即使是将一个对象赋予它自身，也要能正确工作。一个好的方法是在销毁左侧运算对象资源之前拷贝右侧运算对象。

13.2.1 节练习

练习 13.23：比较上一节练习中你编写的拷贝控制成员和这一节中的代码。确定你理解了你的代码和我们的代码之间的差异（如果有的话）。

练习 13.24：如果本节中的 HasPtr 版本未定义析构函数，将会发生什么？如果未定义拷贝构造函数，将会发生什么？

练习 13.25：假定希望定义 StrBlob 的类值版本，而且希望继续使用 shared_ptr，

这样我们的 `StrBlobPtr` 类就仍能使用指向 `vector` 的 `weak_ptr` 了。你修改后的类将需要一个拷贝构造函数和一个拷贝赋值运算符，但不需要析构函数。解释拷贝构造函数和拷贝赋值运算符必须要做什么。解释为什么不需要析构函数。

练习 13.26：对上一题中描述的 `StrBlob` 类，编写你自己的版本。

13.2.2 定义行为像指针的类



对于行为类似指针的类，我们需要为其定义拷贝构造函数和拷贝赋值运算符，来拷贝指针成员本身而不是它指向的 `string`。我们的类仍然需要自己的析构函数来释放接受 `string` 参数的构造函数分配的内存（参见 13.1.4 节，第 447 页）。但是，在本例中，析构函数不能单方面地释放关联的 `string`。只有当最后一个指向 `string` 的 `HasPtr` 销毁时，它才可以释放 `string`。

令一个类展现类似指针的行为的最好方法是使用 `shared_ptr` 来管理类中的资源。拷贝（或赋值）一个 `shared_ptr` 会拷贝（赋值）`shared_ptr` 所指向的指针。<514>
`shared_ptr` 类自己记录有多少用户共享它所指向的对象。当没有用户使用对象时，`shared_ptr` 类负责释放资源。

但是，有时我们希望直接管理资源。在这种情况下，使用引用计数（reference count）（参见 12.1.1 节，第 402 页）就很有用了。为了说明引用计数如何工作，我们将重新定义 `HasPtr`，令其行为像指针一样，但我们不使用 `shared_ptr`，而是设计自己的引用计数。

引用计数

引用计数的工作方式如下：

- 除了初始化对象外，每个构造函数（拷贝构造函数除外）还要创建一个引用计数，用来记录有多少对象与正在创建的对象共享状态。当我们创建一个对象时，只有一个对象共享状态，因此将计数器初始化为 1。
- 拷贝构造函数不分配新的计数器，而是拷贝给定对象的数据成员，包括计数器。拷贝构造函数递增共享的计数器，指出给定对象的状态又被一个新用户所共享。
- 析构函数递减计数器，指出共享状态的用户少了一个。如果计数器变为 0，则析构函数释放状态。
- 拷贝赋值运算符递增右侧运算对象的计数器，递减左侧运算对象的计数器。如果左侧运算对象的计数器变为 0，意味着它的共享状态没有用户了，拷贝赋值运算符就必须销毁状态。

唯一的难题是确定在哪里存放引用计数。计数器不能直接作为 `HasPtr` 对象的成员。下面的例子说明了原因：

```
HasPtr p1("Hiya!");
HasPtr p2(p1);    // p1 和 p2 指向相同的 string
HasPtr p3(p1);    // p1、p2 和 p3 都指向相同的 string
```

如果引用计数保存在每个对象中，当创建 `p3` 时我们应该如何正确更新它呢？可以递增 `p1` 中的计数器并将其拷贝到 `p3` 中，但如何更新 `p2` 中的计数器呢？

解决此问题的一种方法是将计数器保存在动态内存中。当创建一个对象时，我们也分配一个新的计数器。当拷贝或赋值对象时，我们拷贝指向计数器的指针。使用这种方法，副本和原对象都会指向相同的计数器。

定义一个使用引用计数的类

通过使用引用计数，我们就可以编写类指针的 HasPtr 版本了：

```
515> class HasPtr {
public:
    // 构造函数分配新的 string 和新的计数器，将计数器置为 1
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0), use(new std::size_t(1)) {}
    // 拷贝构造函数拷贝所有三个数据成员，并递增计数器
    HasPtr(const HasPtr &p):
        ps(p.ps), i(p.i), use(p.use) { ++*use; }
    HasPtr& operator=(const HasPtr &);

    ~HasPtr();

private:
    std::string *ps;
    int i;
    std::size_t *use; // 用来记录有多少个对象共享*ps 的成员
};
```

在此，我们添加了一个名为 `use` 的数据成员，它记录有多少对象共享相同的 `string`。接受 `string` 参数的构造函数分配新的计数器，并将其初始化为 1，指出当前有一个用户使用本对象的 `string` 成员。

类指针的拷贝成员“篡改”引用计数

当拷贝或赋值一个 `HasPtr` 对象时，我们希望副本和原对象都指向相同的 `string`。即，当拷贝一个 `HasPtr` 时，我们将拷贝 `ps` 本身，而不是 `ps` 指向的 `string`。当我们进行拷贝时，还会递增该 `string` 关联的计数器。

(我们在类内定义的) 拷贝构造函数拷贝给定 `HasPtr` 的所有三个数据成员。这个构造函数还递增 `use` 成员，指出 `ps` 和 `p.ps` 指向的 `string` 又有了一个新的用户。

析构函数不能无条件地 `delete ps`——可能还有其他对象指向这块内存。析构函数应该递减引用计数，指出共享 `string` 的对象少了一个。如果计数器变为 0，则析构函数释放 `ps` 和 `use` 指向的内存：

```
HasPtr::~HasPtr()
{
    if (--*use == 0) { // 如果引用计数变为 0
        delete ps; // 释放 string 内存
        delete use; // 释放计数器内存
    }
}
```

拷贝赋值运算符与往常一样执行类似拷贝构造函数和析构函数的工作。即，它必须递增右侧运算对象的引用计数（即，拷贝构造函数的工作），并递减左侧运算对象的引用计数，在必要时释放使用的内存（即，析构函数的工作）。

而且与往常一样，赋值运算符必须处理自赋值。我们通过先递增 `rhs` 中的计数然后递减左侧运算对象中的计数来实现这一点。通过这种方法，当两个对象相同时，在我们检查 `ps`（及 `use`）是否应该释放之前，计数器就已经被递增过了：

```
516> HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    if (this != &rhs) {
```

```

++*rhs.use; // 递增右侧运算对象的引用计数
if (--*use == 0) { // 然后递减本对象的引用计数
    delete ps; // 如果没有其他用户
    delete use; // 释放本对象分配的成员
}
ps = rhs.ps; // 将数据从 rhs 拷贝到本对象
i = rhs.i;
use = rhs.use;
return *this; // 返回本对象
}

```

13.2.2 节练习

练习 13.27: 定义你自己的使用引用计数版本的 HasPtr。

练习 13.28: 给定下面的类，为其实现一个默认构造函数和必要的拷贝控制成员。

<p>(a) class TreeNode { private: std::string value; int count; TreeNode *left; TreeNode *right; };</p>	<p>(b) class BinStrTree { private: TreeNode *root; };</p>
--	--

13.3 交换操作

除了定义拷贝控制成员，管理资源的类通常还定义一个名为 swap 的函数（参见 9.2.5 节，第 303 页）。对于那些与重排元素顺序的算法（参见 10.2.3 节，第 342 页）一起使用的类，定义 swap 是非常重要的。这类算法在需要交换两个元素时会调用 swap。

如果一个类定义了自己的 swap，那么算法将使用类自定义版本。否则，算法将使用标准库定义的 swap。虽然与往常一样我们不知道 swap 是如何实现的，但理论上很容易理解，为了交换两个对象我们需要进行一次拷贝和两次赋值。例如，交换两个类值 HasPtr 对象（参见 13.2.1 节，第 453 页）的代码可能像下面这样：

```

HasPtr temp = v1; // 创建 v1 的值的一个临时副本
v1 = v2; // 将 v2 的值赋予 v1
v2 = temp; // 将保存的 v1 的值赋予 v2

```

这段代码将原来 v1 中的 string 拷贝了两次——第一次是 HasPtr 的拷贝构造函数将 v1 拷贝给 temp，第二次是赋值运算符将 temp 赋予 v2。将 v2 赋予 v1 的语句还拷贝了原来 v2 中的 string。如我们所见，拷贝一个类值的 HasPtr 会分配一个新 string 并将其拷贝到 HasPtr 指向的位置。517

理论上，这些内存分配都是不必要的。我们更希望 swap 交换指针，而不是分配 string 的新副本。即，我们希望这样交换两个 HasPtr：

```

string *temp = v1.ps; // 为 v1.ps 中的指针创建一个副本
v1.ps = v2.ps; // 将 v2.ps 中的指针赋予 v1.ps
v2.ps = temp; // 将保存的 v1.ps 中原来的指针赋予 v2.ps

```

编写我们自己的 swap 函数

可以在我们的类上定义一个自己版本的 swap 来重载 swap 的默认行为。swap 的典型实现如下：

```
class HasPtr {
    friend void swap(HasPtr&, HasPtr&);
    // 其他成员定义，与 13.2.1 节（第 453 页）中一样
};

inline
void swap(HasPtr &lhs, HasPtr &rhs)
{
    using std::swap;
    swap(lhs.ps, rhs.ps);      // 交换指针，而不是 string 数据
    swap(lhs.i, rhs.i);        // 交换 int 成员
}
```

我们首先将 swap 定义为 friend，以便能访问 HasPtr 的（private 的）数据成员。由于 swap 的存在就是为了优化代码，我们将其声明为 inline 函数（参见 6.5.2 节，第 213 页）。swap 的函数体对给定对象的每个数据成员调用 swap。我们首先 swap 绑定到 rhs 和 lhs 的对象的指针成员，然后是 int 成员。



与拷贝控制成员不同，swap 并不是必要的。但是，对于分配了资源的类，定义 swap 可能是一种很重要的优化手段。



swap 函数应该调用 swap，而不是 std::swap

此代码中有一个很重要的微妙之处：虽然这一点在这个特殊的例子中并不重要，但在一般情况下它非常重要——swap 函数中调用的 swap 不是 std::swap。在本例中，数据成员是内置类型的，而内置类型是没有特定版本的 swap 的，所以在本例中，对 swap 的调用会调用标准库 std::swap。

但是，如果一个类的成员有自己类型特定的 swap 函数，调用 std::swap 就是错误的了。例如，假定我们有另一个命名为 Foo 的类，它有一个类型为 HasPtr 的成员 h。如果我们未定义 Foo 版本的 swap，那么就会使用标准库版本的 swap。如我们所见，标准库 swap 对 HasPtr 管理的 string 进行了不必要的拷贝。

518 我们可以为 Foo 编写一个 swap 函数，来避免这些拷贝。但是，如果这样编写 Foo 版本的 swap：

```
void swap(Foo &lhs, Foo &rhs)
{
    // 错误：这个函数使用了标准库版本的 swap，而不是 HasPtr 版本
    std::swap(lhs.h, rhs.h);
    // 交换类型 Foo 的其他成员
}
```

此编码会编译通过，且正常运行。但是，使用此版本与简单使用默认版本的 swap 并没有任何性能差异。问题在于我们显式地调用了标准库版本的 swap。但是，我们不希望使用 std 中的版本，我们希望调用为 HasPtr 对象定义的版本。

正确的 swap 函数如下所示：

```
void swap(Foo &lhs, Foo &rhs)
```

```

{
    using std::swap;
    swap(lhs.h, rhs.h); // 使用 HasPtr 版本的 swap
    // 交换类型 Foo 的其他成员
}

```

每个 `swap` 调用应该都是未加限定的。即，每个调用都应该是 `swap`，而不是 `std::swap`。如果存在类型特定的 `swap` 版本，其匹配程度会优于 `std` 中定义的版本，原因我们将在 16.3 节（第 616 页）中进行解释。因此，如果存在类型特定的 `swap` 版本，`swap` 调用会与之匹配。如果不存在类型特定的版本，则会使用 `std` 中的版本（假定作用域中有 `using` 声明）。

非常仔细的读者可能会奇怪为什么 `swap` 函数中的 `using` 声明没有隐藏 `HasPtr` 版本 `swap` 的声明（参见 6.4.1 节，第 210 页）。我们将在 18.2.3 节（第 706 页）中解释为什么这段代码能正常工作。

在赋值运算符中使用 `swap`

定义 `swap` 的类通常用 `swap` 来定义它们的赋值运算符。这些运算符使用了一种名为拷贝并交换（copy and swap）的技术。这种技术将左侧运算对象与右侧运算对象的一个副本进行交换：

```

// 注意 rhs 是按值传递的，意味着 HasPtr 的拷贝构造函数
// 将右侧运算对象中的 string 拷贝到 rhs
HasPtr& HasPtr::operator=(HasPtr rhs)
{
    // 交换左侧运算对象和局部变量 rhs 的内容
    swap(*this, rhs);      // rhs 现在指向本对象曾经使用的内存
    return *this;           // rhs 被销毁，从而 delete 了 rhs 中的指针
}

```

在这个版本的赋值运算符中，参数并不是一个引用，我们将右侧运算对象以传值方式传递给了赋值运算符。因此，`rhs` 是右侧运算对象的一个副本。参数传递时拷贝 `HasPtr` 的操作会分配该对象的 `string` 的一个新副本。519

在赋值运算符的函数体中，我们调用 `swap` 来交换 `rhs` 和 `*this` 中的数据成员。这个调用将左侧运算对象中原来保存的指针存入 `rhs` 中，并将 `rhs` 中原来的指针存入 `*this` 中。因此，在 `swap` 调用之后，`*this` 中的指针成员将指向新分配的 `string`——右侧运算对象中 `string` 的一个副本。

当赋值运算符结束时，`rhs` 被销毁，`HasPtr` 的析构函数将执行。此析构函数 `delete rhs` 现在指向的内存，即，释放掉左侧运算对象中原来的内存。

这个技术的有趣之处是它自动处理了自赋值情况且天然就是异常安全的。它通过在改变左侧运算对象之前拷贝右侧运算对象保证了自赋值的正确，这与我们在原来的赋值运算符中使用的方法是一致的（参见 13.2.1 节，第 453 页）。它保证异常安全的方法也与原来的赋值运算符实现一样。代码中唯一可能抛出异常的是拷贝构造函数中的 `new` 表达式。如果真发生了异常，它也会在我们改变左侧运算对象之前发生。



使用拷贝和交换的赋值运算符自动就是异常安全的，且能正确处理自赋值。

13.3 节练习

练习 13.29: 解释 swap(HasPtr&, HasPtr&) 中对 swap 的调用不会导致递归循环。

练习 13.30: 为你的类值版本的 HasPtr 编写 swap 函数, 并测试它。为你的 swap 函数添加一个打印语句, 指出函数什么时候执行。

练习 13.31: 为你的 HasPtr 类定义一个<运算符, 并定义一个 HasPtr 的 vector。为这个 vector 添加一些元素, 并对它执行 sort。注意何时会调用 swap。

练习 13.32: 类指针的 HasPtr 版本会从 swap 函数受益吗? 如果会, 得到了什么益处? 如果不是, 为什么?

13.4 拷贝控制示例

虽然通常来说分配资源的类更需要拷贝控制, 但资源管理并不是一个类需要定义自己的拷贝控制成员的唯一原因。一些类也需要拷贝控制成员的帮助来进行簿记工作或其他操作。

作为类需要拷贝控制来进行簿记操作的例子, 我们将概述两个类的设计, 这两个类可能用于邮件处理应用中。两个类命名为 Message 和 Folder, 分别表示电子邮件 (或者其他类型的) 消息和消息目录。每个 Message 对象可以出现在多个 Folder 中。但是, 任意给定的 Message 的内容只有一个副本。这样, 如果一条 Message 的内容被改变, 则我们从它所在的任何 Folder 来浏览此 Message 时, 都会看到改变后的内容。

为了记录 Message 位于哪些 Folder 中, 每个 Message 都会保存一个它所在 Folder 的指针的 set, 同样的, 每个 Folder 都保存一个它包含的 Message 的指针的 set。图 13.1 说明了这种设计思路。

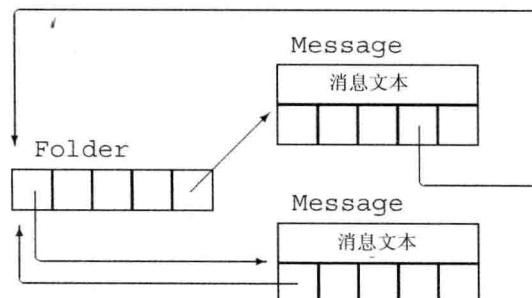


图 13.1: Message 和 Folder 类设计

我们的 Message 类会提供 save 和 remove 操作, 来向一个给定 Folder 添加一条 Message 或是从中删除一条 Message。为了创建一个新的 Message, 我们会指明消息内容, 但不会指出 Folder。为了将一条 Message 放到一个特定 Folder 中, 我们必须调用 save。

当我们拷贝一个 Message 时, 副本和原对象将是不同的 Message 对象, 但两个 Message 都出现在相同的 Folder 中。因此, 拷贝 Message 的操作包括消息内容和 Folder 指针 set 的拷贝。而且, 我们必须在每个包含此消息的 Folder 中都添加一个指向新创建的 Message 的指针。

当我们销毁一个 Message 时, 它将不复存在。因此, 我们必须从包含此消息的所有

Folder 中删除指向此 Message 的指针。

当我们把一个 Message 对象赋予另一个 Message 对象时，左侧 Message 的内容会被右侧 Message 的内容所替代。我们还必须更新 Folder 集合，从原来包含左侧 Message 的 Folder 中将它删除，并将它添加到包含右侧 Message 的 Folder 中。

观察这些操作，我们可以看到，析构函数和拷贝赋值运算符都必须从包含一条 Message 的所有 Folder 中删除它。类似的，拷贝构造函数和拷贝赋值运算符都要将一个 Message 添加到给定的一组 Folder 中。我们将定义两个 private 的工具函数来完成这些工作。



拷贝赋值运算符通常执行拷贝构造函数和析构函数中也要做的工作。这种情况下，公共的工作应该放在 private 的工具函数中完成。

Folder 类也需要类似的拷贝控制成员，来添加或删除它保存的 Message。

521

我们将 Folder 类的设计和实现留作练习。但是，我们将假定 Folder 类包含名为 addMsg 和 remMsg 的成员，分别完成在给定 Folder 对象的消息集合中添加和删除 Message 的工作。

Message 类

根据上述设计，我们可以编写 Message 类，如下所示：

```
class Message {
    friend class Folder;
public:
    // folders 被隐式初始化为空集合
    explicit Message(const std::string &str = "") :
        contents(str) { }
    // 拷贝控制成员，用来管理指向本 Message 的指针
    Message(const Message&);           // 拷贝构造函数
    Message& operator=(const Message&); // 拷贝赋值运算符
    ~Message();                         // 析构函数
    // 从给定 Folder 集合中添加/删除本 Message
    void save(Folder&);
    void remove(Folder&);
private:
    std::string contents;           // 实际消息文本
    std::set<Folder*> folders;    // 包含本 Message 的 Folder
    // 拷贝构造函数、拷贝赋值运算符和析构函数所使用的工具函数
    // 将本 Message 添加到指向参数的 Folder 中
    void add_to_Folders(const Message&);
    // 从 folders 中的每个 Folder 中删除本 Message
    void remove_from_Folders();
};
```

这个类定义了两个数据成员：contents，保存消息文本；folders，保存指向本 Message 所在 Folder 的指针。接受一个 string 参数的构造函数将给定 string 拷贝给 contents，并将 folders（隐式）初始化为空集。由于此构造函数有一个默认参数，因此它也被当作 Message 的默认构造函数（参见 7.5.1 节，第 260 页）。

save 和 remove 成员

除拷贝控制成员外，Message 类只有两个公共成员：save，将本 Message 存放在给定 Folder 中；remove，删除本 Message：

```
void Message::save(Folder &f)
{
    folders.insert(&f); // 将给定 Folder 添加到我们的 Folder 列表中
    f.addMsg(this); // 将本 Message 添加到 f 的 Message 集合中
}
522 void Message::remove(Folder &f)
{
    folders.erase(&f); // 将给定 Folder 从我们的 Folder 列表中删除
    f.remMsg(this); // 将本 Message 从 f 的 Message 集合中删除
}
```

为了保存（或删除）一个 Message，需要更新本 Message 的 folders 成员。当 save 一个 Message 时，我们应保存一个指向给定 Folder 的指针；当 remove 一个 Message 时，我们要删除此指针。

这些操作还必须更新给定的 Folder。更新一个 Folder 的任务是由 Folder 类的 addMsg 和 remMsg 成员来完成的，分别添加和删除给定 Message 的指针。

Message 类的拷贝控制成员

当我们拷贝一个 Message 时，得到的副本应该与原 Message 出现在相同的 Folder 中。因此，我们必须遍历 Folder 指针的 set，对每个指向原 Message 的 Folder 添加一个指向新 Message 的指针。拷贝构造函数和拷贝赋值运算符都需要做这个工作，因此我们定义一个函数来完成这个公共操作：

```
// 将本 Message 添加到指向 m 的 Folder 中
void Message::add_to_Folders(const Message &m)
{
    for (auto f : m.folders) // 对每个包含 m 的 Folder
        f->addMsg(this); // 向该 Folder 添加一个指向本 Message 的指针
}
```

此例中我们对 m.folders 中每个 Folder 调用 addMsg。函数 addMsg 会将本 Message 的指针添加到每个 Folder 中。

Message 的拷贝构造函数拷贝给定对象的数据成员：

```
Message::Message(const Message &m):
    contents(m.contents), folders(m.folders)
{
    add_to_Folders(m); // 将本消息添加到指向 m 的 Folder 中
}
```

并调用 add_to_Folders 将新创建的 Message 的指针添加到每个包含原 Message 的 Folder 中。

Message 的析构函数

当一个 Message 被销毁时，我们必须从指向此 Message 的 Folder 中删除它。拷贝赋值运算符也要执行此操作，因此我们会定义一个公共函数来完成此工作：

```
// 从对应的 Folder 中删除本 Message
void Message::remove_from_Folders()
{
    for (auto f : folders) // 对 folders 中每个指针
        f->remMsg(this); // 从该 Folder 中删除本 Message
}
```

函数 `remove_from_Folders` 的实现类似 `add_to_Folders`, 不同之处是它调用 `remMsg` 来删除当前 `Message` 而不是调用 `addMsg` 来添加 `Message`。 ◀523

有了 `remove_from_Folders` 函数, 编写析构函数就很简单了:

```
Message::~Message()
{
    remove_from_Folders();
}
```

调用 `remove_from_Folders` 确保没有任何 `Folder` 保存正在销毁的 `Message` 的指针。编译器自动调用 `string` 的析构函数来释放 `contents`, 并自动调用 `set` 的析构函数来清理集合成员使用的内存。

Message 的拷贝赋值运算符

与大多数赋值运算符相同, 我们的 `Message` 类的拷贝赋值运算符必须执行拷贝构造函数和析构函数的工作。与往常一样, 最重要的是我们要组织好代码结构, 使得即使左侧和右侧运算对象是同一个 `Message`, 拷贝赋值运算符也能正确执行。

在本例中, 我们先从左侧运算对象的 `folders` 中删除此 `Message` 的指针, 然后再将指针添加到右侧运算对象的 `folders` 中, 从而实现了自赋值的正确处理:

```
Message& Message::operator=(const Message &rhs)
{
    // 通过先删除指针再插入它们来处理自赋值情况
    remove_from_Folders(); // 更新已有 Folder
    contents = rhs.contents; // 从 rhs 拷贝消息内容
    folders = rhs.folders; // 从 rhs 拷贝 Folder 指针
    add_to_Folders(rhs); // 将本 Message 添加到那些 Folder 中
    return *this;
}
```

如果左侧和右侧运算对象是相同的 `Message`, 则它们具有相同的地址。如果我们在 `add_to_Folders` 之后调用 `remove_from_Folders`, 就会将此 `Message` 从它所在的所有 `Folder` 中删除。

Message 的 swap 函数

标准库中定义了 `string` 和 `set` 的 `swap` 版本 (参见 9.2.5 节, 第 303 页)。因此, 如果为我们的 `Message` 类定义它自己的 `swap` 版本, 它将从中受益。通过定义一个 `Message` 特定版本的 `swap`, 我们可以避免对 `contents` 和 `folders` 成员进行不必要的拷贝。

但是, 我们的 `swap` 函数必须管理指向被交换 `Message` 的 `Folder` 指针。在调用 `swap(m1, m2)` 之后, 原来指向 `m1` 的 `Folder` 现在必须指向 `m2`, 反之亦然。

我们通过两遍扫描 `folders` 中每个成员来正确处理 `Folder` 指针。第一遍扫描将 `Message` 从它们所在的 `Folder` 中删除。接下来我们调用 `swap` 来交换数据成员。最后

对 `folders` 进行第二遍扫描来添加交换过的 `Message`:

```
524> void swap(Message &lhs, Message &rhs)
{
    using std::swap; // 在本例中严格来说并不需要，但这是一个好习惯
    // 将每个消息的指针从它（原来）所在 Folder 中删除
    for (auto f: lhs.folders)
        f->remMsg(&lhs);
    for (auto f: rhs.folders)
        f->remMsg(&rhs);
    // 交换 contents 和 Folder 指针 set
    swap(lhs.folders, rhs.folders);           // 使用 swap(set&, set&)
    swap(lhs.contents, rhs.contents);         // swap(string&, string&)
    // 将每个 Message 的指针添加到它的（新）Folder 中
    for (auto f: lhs.folders)
        f->addMsg(&lhs);
    for (auto f: rhs.folders)
        f->addMsg(&rhs);
}
```

13.4 节练习

练习 13.33: 为什么 `Message` 的成员 `save` 和 `remove` 的参数是一个 `Folder&`? 为什么我们不将参数定义为 `Folder` 或是 `const Folder&`?

练习 13.34: 编写本节所描述的 `Message`。

练习 13.35: 如果 `Message` 使用合成的拷贝控制成员，将会发生什么?

练习 13.36: 设计并实现对应的 `Folder` 类。此类应该保存一个指向 `Folder` 中包含的 `Message` 的 `set`。

练习 13.37: 为 `Message` 类添加成员，实现向 `folders` 添加或删除一个给定的 `Folder*`。这两个成员类似 `Folder` 类的 `addMsg` 和 `remMsg` 操作。

练习 13.38: 我们并未使用拷贝和交换方式来设计 `Message` 的赋值运算符。你认为其原因是什么?



13.5 动态内存管理类

某些类需要在运行时分配可变大小的内存空间。这种类通常可以（并且如果它们确实可以说的话，一般应该）使用标准库容器来保存它们的数据。例如，我们的 `StrBlob` 类使用一个 `vector` 来管理其元素的底层内存。

但是，这一策略并不是对每个类都适用；某些类需要自己进行内存分配。这些类一般来说必须定义自己的拷贝控制成员来管理所分配的内存。



例如，我们将实现标准库 `vector` 类的一个简化版本。我们所做的一个简化是不使用模板，我们的类只用于 `string`。因此，它被命名为 `StrVec`。

StrVec 类的设计

回忆一下，`vector` 类将其元素保存在连续内存中。为了获得可接受的性能，`vector`

预先分配足够的内存来保存可能需要的更多元素（参见 9.4 节，第 317 页）。`vector` 的每个添加元素的成员函数会检查是否有空间容纳更多的元素。如果有，成员函数会在下一个可用位置构造一个对象。如果没有可用空间，`vector` 就会重新分配空间：它获得新的空间，将已有元素移动到新空间中，释放旧空间，并添加新元素。

我们在 `StrVec` 类中使用类似的策略。我们将使用一个 `allocator` 来获得原始内存（参见 12.2.2 节，第 427 页）。由于 `allocator` 分配的内存是未构造的，我们将在需要添加新元素时用 `allocator` 的 `construct` 成员在原始内存中创建对象。类似的，当我们需要删除一个元素时，我们将使用 `destroy` 成员来销毁元素。

每个 `StrVec` 有三个指针成员指向其元素所使用的内存：

- `elements`, 指向分配的内存中的首元素
- `first_free`, 指向最后一个实际元素之后的位置
- `cap`, 指向分配的内存末尾之后的位置

图 13.2 说明了这些指针的含义。

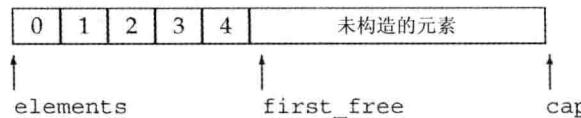


图 13.2: StrVec 内存分配策略

除了这些指针之外，`StrVec` 还有一个名为 `alloc` 的静态成员，其类型为 `allocator<string>`。`alloc` 成员会分配 `StrVec` 使用的内存。我们的类还有 4 个工具函数：

- `alloc_n_copy` 会分配内存，并拷贝一个给定范围中的元素。
- `free` 会销毁构造的元素并释放内存。
- `chk_n_alloc` 保证 `StrVec` 至少有容纳一个新元素的空间。如果没有空间添加新元素，`chk_n_alloc` 会调用 `reallocate` 来分配更多内存。
- `reallocate` 在内存用完时为 `StrVec` 分配新内存。

虽然我们关注的是类的实现，但我们将定义 `vector` 接口中的一些成员。

StrVec 类定义

526

有了上述实现概要，我们现在可以定义 `StrVec` 类，如下所示：

```
// 类 vector 类内存分配策略的简化实现
class StrVec {
public:
    StrVec(): // allocator 成员进行默认初始化
        elements(nullptr), first_free(nullptr), cap(nullptr) { }
    StrVec(const StrVec&); // 拷贝构造函数
    StrVec &operator=(const StrVec&); // 拷贝赋值运算符
    ~StrVec(); // 析构函数
    void push_back(const std::string&); // 拷贝元素
    size_t size() const { return first_free - elements; }
    size_t capacity() const { return cap - elements; }
    std::string *begin() const { return elements; }
    std::string *end() const { return first_free; }
```

```

    // ...
private:
    Static std::allocator<std::string> alloc; // 分配元素
    // 被添加元素的函数所使用
    void chk_n_alloc()
    {
        if (size() == capacity()) reallocate();
    }
    // 工具函数，被拷贝构造函数、赋值运算符和析构函数所使用
    std::pair<std::string*, std::string*> alloc_n_copy
        (const std::string*, const std::string*);
    void free(); // 销毁元素并释放内存
    void reallocate(); // 获得更多内存并拷贝已有元素
    std::string *elements; // 指向数组首元素的指针
    std::string *first_free; // 指向数组第一个空闲元素的指针
    std::string *cap; // 指向数组尾后位置的指针
};

};

类体定义了多个成员：
```

- 默认构造函数(隐式地)默认初始化 alloc 并(显式地)将指针初始化为 nullptr，表明没有元素。
- size 成员返回当前真正在使用的元素的数目，等于 first_free-elements。
- capacity 成员返回 StrVec 可以保存的元素的数量，等价于 cap-elements。
- 当没有空间容纳新元素，即 cap==first_free 时，chk_n_alloc 会为 StrVec 重新分配内存。
- begin 和 end 成员分别返回指向首元素(即 elements)和最后一个构造的元素之后位置(即 first_free)的指针。

使用 construct

函数 push_back 调用 chk_n_alloc 确保有空间容纳新元素。如果需要，
527 chk_n_alloc 会调用 reallocate。当 chk_n_alloc 返回时，push_back 知道必有空间容纳新元素。它要求其 allocator 成员来 construct 新的尾元素：

```

void StrVec::push_back(const string& s)
{
    chk_n_alloc(); // 确保有空间容纳新元素
    // 在 first_free 指向的元素中构造 s 的副本
    alloc.construct(first_free++, s);
}

```

当我们用 allocator 分配内存时，必须记住内存是未构造的(参见 12.2.2 节，第 428 页)。为了使用此原始内存，我们必须调用 construct，在此内存中构造一个对象。传递给 construct 的第一个参数必须是一个指针，指向调用 allocate 所分配的未构造的内存空间。剩余参数确定用哪个构造函数来构造对象。在本例中，只有一个额外参数，类型为 string，因此会使用 string 的拷贝构造函数。

值得注意的是，对 construct 的调用也会递增 first_free，表示已经构造了一个新元素。它使用前置递增(参见 4.5 节，第 131 页)，因此这个调用会在 first_free 当前值指定的地址构造一个对象，并递增 first_free 指向下一个未构造的元素。

alloc_n_copy 成员

我们在拷贝或赋值 StrVec 时，可能会调用 alloc_n_copy 成员。类似 vector，我们的 StrVec 类有类值的行为（参见 13.2.1 节，第 453 页）。当我们拷贝或赋值 StrVec 时，必须分配独立的内存，并从原 StrVec 对象拷贝元素至新对象。

alloc_n_copy 成员会分配足够的内存来保存给定范围的元素，并将这些元素拷贝到新分配的内存中。此函数返回一个指针的 pair（参见 11.2.3 节，第 379 页），两个指针分别指向新空间的开始位置和拷贝的尾后的位置：

```
pair<string*, string*>
StrVec::alloc_n_copy(const string *b, const string *e)
{
    // 分配空间保存给定范围中的元素
    auto data = alloc.allocate(e - b);
    // 初始化并返回一个 pair，该 pair 由 data 和 uninitialized_copy 的返回值构成
    return {data, uninitialized_copy(b, e, data)};
}
```

alloc_n_copy 用尾后指针减去首元素指针，来计算需要多少空间。在分配内存之后，它必须在此空间中构造给定元素的副本。

它是在返回语句中完成拷贝工作的，返回语句中对返回值进行了列表初始化（参见 6.3.2 节，第 203 页）。返回的 pair 的 first 成员指向分配的内存的开始位置；second 成员则是 uninitialized_copy（参见 12.2.2 节，第 429 页）的返回值，此值是一个指针，指向最后一个构造元素之后的位置。528

free 成员

free 成员有两个责任：首先 destroy 元素，然后释放 StrVec 自己分配的内存空间。for 循环调用 allocator 的 destroy 成员，从构造的尾元素开始，到首元素为止，逆序销毁所有元素：

```
void StrVec::free()
{
    // 不能传递给 deallocate 一个空指针，如果 elements 为 0，函数什么也不做
    if (elements) {
        // 逆序销毁旧元素
        for (auto p = first_free; p != elements; /* 空 */)
            alloc.destroy(--p);
        alloc.deallocate(elements, cap - elements);
    }
}
```

destroy 函数会运行 string 的析构函数。string 的析构函数会释放 string 自己分配的内存空间。

一旦元素被销毁，我们就调用 deallocate 来释放本 StrVec 对象分配的内存空间。我们传递给 deallocate 的指针必须是之前某次 allocate 调用所返回的指针。因此，在调用 deallocate 之前我们首先检查 elements 是否为空。

拷贝控制成员

实现了 alloc_n_copy 和 free 成员后，为我们的类实现拷贝控制成员就很简单了。

拷贝构造函数调用 alloc_n_copy:

```
StrVec::StrVec(const StrVec &s)
{
    // 调用 alloc_n_copy 分配空间以容纳与 s 中一样多的元素
    auto newdata = alloc_n_copy(s.begin(), s.end());
    elements = newdata.first;
    first_free = cap = newdata.second;
}
```

并将返回结果赋予数据成员。alloc_n_copy 的返回值是一个指针的 pair。其 first 成员指向第一个构造的元素，second 成员指向最后一个构造的元素之后的位置。由于 alloc_n_copy 分配的空间恰好容纳给定的元素，cap 也指向最后一个构造的元素之后的位置。

析构函数调用 free:

```
StrVec::~StrVec() { free(); }
```

拷贝赋值运算符在释放已有元素之前调用 alloc_n_copy，这样就可以正确处理自赋值了：

529

```
StrVec &StrVec::operator=(const StrVec &rhs)
{
    // 调用 alloc_n_copy 分配内存，大小与 rhs 中元素占用空间一样多
    auto data = alloc_n_copy(rhs.begin(), rhs.end());
    free();
    elements = data.first;
    first_free = cap = data.second;
    return *this;
}
```

类似拷贝构造函数，拷贝赋值运算符使用 alloc_n_copy 的返回值来初始化它的指针。

在重新分配内存的过程中移动而不是拷贝元素

在编写 reallocate 成员函数之前，我们稍微思考一下此函数应该做什么。它应该

- 为一个新的、更大的 string 数组分配内存
- 在内存空间的前一部分构造对象，保存现有元素
- 销毁原内存空间中的元素，并释放这块内存

观察这个操作步骤，我们可以看出，为一个 StrVec 重新分配内存空间会引起从旧内存空间到新内存空间逐个拷贝 string。虽然我们不知道 string 的实现细节，但我们知道 string 具有类值行为。当拷贝一个 string 时，新 string 和原 string 是相互独立的。改变原 string 不会影响到副本，反之亦然。

由于 string 的行为类似值，我们可以得出结论，每个 string 对构成它的所有字符都会保存自己的一份副本。拷贝一个 string 必须为这些字符分配内存空间，而销毁一个 string 必须释放所占用的内存。

拷贝一个 string 就必须真的拷贝数据，因为通常情况下，在我们拷贝了一个 string 之后，它就会有两个用户。但是，如果是 reallocate 拷贝 StrVec 中的 string，则在拷贝之后，每个 string 只有唯一的用户。一旦将元素从旧空间拷贝到了新空间，我们就会立即销毁原 string。

因此，拷贝这些 `string` 中的数据是多余的。在重新分配内存空间时，如果我们能避免分配和释放 `string` 的额外开销，`StrVec` 的性能会好得多。

移动构造函数和 `std::move`

通过使用新标准库引入的两种机制，我们就可以避免 `string` 的拷贝。首先，有一些标准库类，包括 `string`，都定义了所谓的“移动构造函数”。关于 `string` 的移动构造函数如何工作的细节，以及有关实现的任何其他细节，目前都尚未公开。但是，我们知道，移动构造函数通常是将资源从给定对象“移动”而不是拷贝到正在创建的对象。而且我们知道标准库保证“移后源”（moved-from）`string` 仍然保持一个有效的、可析构的状态。对于 `string`，我们可以想象每个 `string` 都有一个指向 `char` 数组的指针。可以假定 `string` 的移动构造函数进行了指针的拷贝，而不是为字符分配内存空间然后拷贝字符。

C++ 11

< 530

我们使用的第二个机制是一个名为 `move` 的标准库函数，它定义在 `utility` 头文件中。目前，关于 `move` 我们需要了解两个关键点。首先，当 `reallocate` 在新内存中构造 `string` 时，它必须调用 `move` 来表示希望使用 `string` 的移动构造函数，原因我们将在 13.6.1 节（第 470 页）中解释。如果它漏掉了 `move` 调用，将会使用 `string` 的拷贝构造函数。其次，我们通常不为 `move` 提供一个 `using` 声明（参见 3.1 节，第 74 页），原因我们将在 18.2.3 节（第 706 页）中解释。当我们使用 `move` 时，直接调用 `std::move` 而不是 `move`。

`reallocate` 成员

了解了这些知识，现在就可以编写 `reallocate` 成员了。首先调用 `allocate` 分配新内存空间。我们每次重新分配内存时都会将 `StrVec` 的容量加倍。如果 `StrVec` 为空，我们将分配容纳一个元素的空间：

```
void StrVec::reallocate()
{
    // 我们将分配当前大小两倍的内存空间
    auto newcapacity = size() ? 2 * size() : 1;
    // 分配新内存
    auto newdata = alloc.allocate(newcapacity);
    // 将数据从旧内存移动到新内存
    auto dest = newdata;        // 指向新数组中下一个空闲位置
    auto elem = elements;      // 指向旧数组中下一个元素
    for (size_t i = 0; i != size(); ++i)
        alloc.construct(dest++, std::move(*elem++));
    free(); // 一旦我们移动完元素就释放旧内存空间
    // 更新我们的数据结构，执行新元素
    elements = newdata;
    first_free = dest;
    cap = elements + newcapacity;
}
```

`for` 循环遍历每个已有元素，并在新内存空间中 `construct` 一个对应元素。我们使用 `dest` 指向构造新 `string` 的内存，使用 `elem` 指向原数组中的元素。我们每次用后置递增运算将 `dest`（和 `elem`）推进到各自数组中的下一个元素。

`construct` 的第二个参数（即，确定使用哪个构造函数的参数（参见 12.2.2 节，第 428 页））是 `move` 返回的值。调用 `move` 返回的结果会令 `construct` 使用 `string` 的移

动构造函数。由于我们使用了移动构造函数，这些 `string` 管理的内存将不会被拷贝。相反，我们构造的每个 `string` 都会从 `elem` 指向的 `string` 那里接管内存的所有权。

531 在元素移动完毕后，我们调用 `free` 销毁旧元素并释放 `StrVec` 原来使用的内存。`string` 成员不再管理它们曾经指向的内存；其数据的管理职责已经转移给新 `StrVec` 内存中的元素了。我们不知道旧 `StrVec` 内存中的 `string` 包含什么值，但我们保证对它们执行 `string` 的析构函数是安全的。

剩下的就是更新指针，指向新分配并已初始化过的数组了。`first_free` 和 `cap` 指针分别被设置为指向最后一个构造的元素之后的位置及指向新分配空间的尾后位置。

13.5 节练习

练习 13.39：编写你自己版本的 `StrVec`，包括自己版本的 `reserve`、`capacity`（参见 9.4 节，第 318 页）和 `resize`（参见 9.3.5 节，第 314 页）。

练习 13.40：为你的 `StrVec` 类添加一个构造函数，它接受一个 `initializer_list<string>` 参数。

练习 13.41：在 `push_back` 中，我们为什么在 `construct` 调用中使用前置递增运算？如果使用后置递增运算的话，会发生什么？

练习 13.42：在你的 `TextQuery` 和 `QueryResult` 类（参见 12.3 节，第 431 页）中用你的 `StrVec` 类代替 `vector<string>`，以此来测试你的 `StrVec` 类。

练习 13.43：重写 `free` 成员，用 `for_each` 和 `lambda`（参见 10.3.2 节，第 346 页）来代替 `for` 循环 `destroy` 元素。你更倾向于哪种实现，为什么？

练习 13.44：编写标准库 `string` 类的简化版本，命名为 `String`。你的类应该至少有一个默认构造函数和一个接受 C 风格字符串指针参数的构造函数。使用 `allocator` 为你的 `String` 类分配所需内存。



13.6 对象移动

新标准的一个最主要的特性是可以移动而非拷贝对象的能力。如我们在 13.1.1 节（第 440 页）中所见，很多情况下都会发生对象拷贝。在其中某些情况下，对象拷贝后就立即被销毁了。在这些情况下，移动而非拷贝对象会大幅度提升性能。

如我们已经看到的，我们的 `StrVec` 类是这种不必要的拷贝的一个很好的例子。在重新分配内存的过程中，从旧内存将元素拷贝到新内存是不必要的，更好的方式是移动元素。使用移动而不是拷贝的另一个原因源于 `IO` 类或 `unique_ptr` 这样的类。这些类都包含不能被共享的资源（如指针或 `IO` 缓冲）。因此，这些类型的对象不能拷贝但可以移动。



532 在旧 C++ 标准中，没有直接的方法移动对象。因此，即使不必拷贝对象的情况下，我们也不得不拷贝。如果对象较大，或者是对象本身要求分配内存空间（如 `string`），进行不必要的拷贝代价非常高。类似的，在旧版本的标准库中，容器中所保存的类必须是可拷贝的。但在新标准中，我们可以用容器保存不可拷贝的类型，只要它们能被移动即可。



标准库容器、`string` 和 `shared_ptr` 类既支持移动也支持拷贝。IO 类和 `unique_ptr` 类可以移动但不能拷贝。

13.6.1 右值引用



C++ 11

为了支持移动操作，新标准引入了一种新的引用类型——右值引用（rvalue reference）。所谓右值引用就是必须绑定到右值的引用。我们通过`&&`而不是`&`来获得右值引用。如我们将要看到的，右值引用有一个重要的性质——只能绑定到一个将要销毁的对象。因此，我们可以自由地将一个右值引用的资源“移动”到另一个对象中。

回忆一下，左值和右值是表达式的属性（参见 4.1.1 节，第 121 页）。一些表达式生成或要求左值，而另外一些则生成或要求右值。一般而言，一个左值表达式表示的是一个对象的身份，而一个右值表达式表示的是对象的值。

类似任何引用，一个右值引用也不过是某个对象的另一个名字而已。如我们所知，对于常规引用（为了与右值引用区分开来，我们可以称之为左值引用（lvalue reference）），我们不能将其绑定到要求转换的表达式、字面常量或是返回右值的表达式（参见 2.3.1 节，第 46 页）。右值引用有着完全相反的绑定特性：我们可以将一个右值引用绑定到这类表达式上，但不能将一个右值引用直接绑定到一个左值上：

```
int i = 42;
int &r = i;           // 正确: r 引用 i
int &&rr = i;         // 错误: 不能将一个右值引用绑定到一个左值上
int &r2 = i * 42;    // 错误: i*42 是一个右值
const int &r3 = i * 42; // 正确: 我们可以将一个 const 的引用绑定到一个右值上
int &&rr2 = i * 42;  // 正确: 将 rr2 绑定到乘法结果上
```

返回左值引用的函数，连同赋值、下标、解引用和前置递增/递减运算符，都是返回左值的表达式的例子。我们可以将一个左值引用绑定到这类表达式的结果上。

返回非引用类型的函数，连同算术、关系、位以及后置递增/递减运算符，都生成右值。我们不能将一个左值引用绑定到这类表达式上，但我们可以将一个 `const` 的左值引用或者一个右值引用绑定到这类表达式上。

左值持久：右值短暂

533

考察左值和右值表达式的列表，两者相互区别之处就很明显了：左值有持久的状态，而右值要么是字面常量，要么是在表达式求值过程中创建的临时对象。

由于右值引用只能绑定到临时对象，我们得知

- 所引用的对象将要被销毁
- 该对象没有其他用户

这两个特性意味着：使用右值引用的代码可以自由地接管所引用的对象的资源。



右值引用指向将要被销毁的对象。因此，我们可以从绑定到右值引用的对象“窃取”状态。

变量是左值

变量可以看作只有一个运算对象而没有运算符的表达式，虽然我们很少这样看待变

量。类似其他任何表达式，变量表达式也有左值/右值属性。变量表达式都是左值。带来的结果就是，我们不能将一个右值引用绑定到一个右值引用类型的变量上，这有些令人惊讶：

```
int &&rr1 = 42; // 正确：字面常量是右值
int &&rr2 = rr1; // 错误：表达式 rr1 是左值！
```

其实有了右值表示临时对象这一观察结果，变量是左值这一特性并不令人惊讶。毕竟，变量是持久的，直至离开作用域时才被销毁。



变量是左值，因此我们不能将一个右值引用直接绑定到一个变量上，即使这个变量是右值引用类型也不行。

标准库 move 函数

C++ 11

虽然不能将一个右值引用直接绑定到一个左值上，但我们可以显式地将一个左值转换为对应的右值引用类型。我们还可以通过调用一个名为 **move** 的新标准库函数来获得绑定到左值上的右值引用，此函数定义在头文件 utility 中。**move** 函数使用了我们将在 16.2.6 节（第 610 页）中描述的机制来返回给定对象的右值引用。

```
int &&rr3 = std::move(rr1); // ok
```

move 调用告诉编译器：我们有一个左值，但我们希望像一个右值一样处理它。我们必须认识到，调用 **move** 就意味着承诺：除了对 **rr1** 赋值或销毁它外，我们将不再使用它。在调用 **move** 之后，我们不能对移后源对象的值做任何假设。

534



我们可以销毁一个移后源对象，也可以赋予它新值，但不能使用一个移后源对象的值。

如前所述，与大多数标准库名字的使用不同，对 **move**（参见 13.5 节，第 469 页）我们不提供 **using** 声明（参见 3.1 节，第 74 页）。我们直接调用 **std::move** 而不是 **move**，其原因将在 18.2.3 节（第 707 页）中解释。



使用 **move** 的代码应该使用 **std::move** 而不是 **move**。这样做可以避免潜在的名字冲突。

13.6.1 节练习

练习 13.45：解释右值引用和左值引用的区别。

练习 13.46：什么类型的引用可以绑定到下面的初始化器上？

```
int f();
vector<int> vi(100);
int? r1 = f();
int? r2 = vi[0];
int? r3 = r1;
int? r4 = vi[0] * f();
```

练习 13.47：对你在练习 13.44（13.5 节，第 470 页）中定义的 **String** 类，为它的拷贝构造函数和拷贝赋值运算符添加一条语句，在每次函数执行时打印一条信息。

练习 13.48: 定义一个 `vector<String>` 并在其上多次调用 `push_back`。运行你的程序，并观察 `String` 被拷贝了多少次。

13.6.2 移动构造函数和移动赋值运算符



类似 `string` 类（及其他标准库类），如果我们自己的类也同时支持移动和拷贝，那么也能从中受益。为了让我们自己的类型支持移动操作，需要为其定义移动构造函数和移动赋值运算符。这两个成员类似对应的拷贝操作，但它们从给定对象“窃取”资源而不是拷贝资源。

类似拷贝构造函数，移动构造函数的第一个参数是该类类型的一个引用。不同于拷贝构造函数的是，这个引用参数在移动构造函数中是一个右值引用。与拷贝构造函数一样，任何额外的参数都必须有默认实参。

除了完成资源移动，移动构造函数还必须确保移后源对象处于这样一个状态——销毁它是无害的。特别是，一旦资源完成移动，源对象必须不再指向被移动的资源——这些资源的所有权已经归属新创建的对象。

作为一个例子，我们为 `StrVec` 类定义移动构造函数，实现从一个 `StrVec` 到另一个 `StrVec` 的元素移动而非拷贝：

```
StrVec::StrVec(StrVec &&s) noexcept // 移动操作不应抛出任何异常
    // 成员初始化器接管 s 中的资源
    : elements(s.elements), first_free(s.first_free), cap(s.cap)
{
    // 令 s 进入这样的状态——对其运行析构函数是安全的
    s.elements = s.first_free = s.cap = nullptr;
}
```

我们将简短解释 `noexcept`（它通知标准库我们的构造函数不抛出任何异常），但让我们先分析一下此构造函数完成什么工作。

与拷贝构造函数不同，移动构造函数不分配任何新内存；它接管给定的 `StrVec` 中的内存。在接管内存之后，它将给定对象中的指针都置为 `nullptr`。这样就完成了从给定对象的移动操作，此对象将继续存在。最终，移后源对象会被销毁，意味着将在其上运行析构函数。`StrVec` 的析构函数在 `first_free` 上调用 `deallocate`。如果我们忘记了改变 `s.first_free`，则销毁移后源对象就会释放掉我们刚刚移动的内存。

移动操作、标准库容器和异常



由于移动操作“窃取”资源，它通常不分配任何资源。因此，移动操作通常不会抛出任何异常。当编写一个不抛出异常的移动操作时，我们应该将此事通知标准库。我们将看到，除非标准库知道我们的移动构造函数不会抛出异常，否则它会认为移动我们的类对象时可能会抛出异常，并且为了处理这种可能性而做一些额外的工作。

一种通知标准库的方法是在我们的构造函数中指明 `noexcept`。`noexcept` 是新标准引入的，我们将在 18.1.4 节（第 690 页）中讨论更多细节。目前重要的是要知道，`noexcept` 是我们承诺一个函数不抛出异常的一种方法。我们在一个函数的参数列表后指定 `noexcept`。在一个构造函数中，`noexcept` 出现在参数列表和初始化列表开始的冒号之间：

```
class StrVec {
```

C++
11

535

```

public:
    StrVec(StrVec&&) noexcept; // 移动构造函数
    // 其他成员的定义，如前
};

StrVec::StrVec(StrVec &&s) noexcept : /* 成员初始化器 */
{ /* 构造函数体 */ }

```

我们必须在类头文件的声明中和定义中（如果定义在类外的话）都指定 `noexcept`。



不抛出异常的移动构造函数和移动赋值运算符必须标记为 `noexcept`。

536

搞清楚为什么需要 `noexcept` 能帮助我们深入理解标准库是如何与我们自定义的类型交互的。我们需要指出一个移动操作不抛出异常，这是因为两个相互关联的事实：首先，虽然移动操作通常不抛出异常，但抛出异常也是允许的；其次，标准库容器能对异常发生时其自身的行为提供保障。例如，`vector` 保证，如果我们调用 `push_back` 时发生异常，`vector` 自身不会发生改变。

现在让我们思考 `push_back` 内部发生了什么。类似对应的 `StrVec` 操作（参见 13.5 节，第 466 页），对一个 `vector` 调用 `push_back` 可能要求为 `vector` 重新分配内存空间。当重新分配 `vector` 的内存时，`vector` 将元素从旧空间移动到新内存中，就像我们在 `reallocate` 中所做的那样（参见 13.5 节，第 469 页）。

如我们刚刚看到的那样，移动一个对象通常会改变它的值。如果重新分配过程使用了移动构造函数，且在移动了部分而不是全部元素后抛出了一个异常，就会产生问题。旧空间中的移动源元素已经被改变了，而新空间中未构造的元素可能尚不存在。在此情况下，`vector` 将不能满足自身保持不变的要求。

另一方面，如果 `vector` 使用了拷贝构造函数且发生了异常，它可以很容易地满足要求。在此情况下，当在新内存中构造元素时，旧元素保持不变。如果此时发生了异常，`vector` 可以释放新分配的（但还未成功构造的）内存并返回。`vector` 原有的元素仍然存在。

为了避免这种潜在问题，除非 `vector` 知道元素类型的移动构造函数不会抛出异常，否则在重新分配内存的过程中，它就必须使用拷贝构造函数而不是移动构造函数。如果希望在 `vector` 重新分配内存这类情况下对我们自定义类型的对象进行移动而不是拷贝，就必须显式地告诉标准库我们的移动构造函数可以安全使用。我们通过将移动构造函数（及移动赋值运算符）标记为 `noexcept` 来做到这一点。

移动赋值运算符

移动赋值运算符执行与析构函数和移动构造函数相同的工作。与移动构造函数一样，如果我们的移动赋值运算符不抛出任何异常，我们就应该将它标记为 `noexcept`。类似拷贝赋值运算符，移动赋值运算符必须正确处理自赋值：

```

StrVec &StrVec::operator=(StrVec &&rhs) noexcept
{
    // 直接检测自赋值
    if (this != &rhs) {
        free(); // 释放已有元素
        elements = rhs.elements; // 从 rhs 接管资源
        first_free = rhs.first_free;
    }
}

```

```
    cap = rhs.cap;
    // 将 rhs 置于可析构状态
    rhs.elements = rhs.first_free = rhs.cap = nullptr;
}
return *this;
}
```

在此例中，我们直接检查 `this` 指针与 `rhs` 的地址是否相同。如果相同，右侧和左侧运算对象指向相同的对象，我们不需要做任何事情。否则，我们释放左侧运算对象所使用的内存，并接管给定对象的内存。与移动构造函数一样，我们将 `rhs` 中的指针置为 `nullptr`。537

我们费心地去检查自赋值情况看起来有些奇怪。毕竟，移动赋值运算符需要右侧运算对象的一个右值。我们进行检查的原因是此右值可能是 `move` 调用的返回结果。与其他任何赋值运算符一样，关键点是我们不能在使用右侧运算对象的资源之前就释放左侧运算对象的资源（可能是相同的资源）。

移后源对象必须可析构



从一个对象移动数据并不会销毁此对象，但有时在移动操作完成后，源对象会被销毁。因此，当我们编写一个移动操作时，必须确保移后源对象进入一个可析构的状态。我们的 `StrVec` 的移动操作满足这一要求，这是通过将移后源对象的指针成员置为 `nullptr` 来实现的。

除了将移后源对象置为析构安全的状态之外，移动操作还必须保证对象仍然是有效的。一般来说，对象有效就是指可以安全地为其赋予新值或者可以安全地使用而不依赖其当前值。另一方面，移动操作对移后源对象中留下的值没有任何要求。因此，我们的程序不应该依赖于移后源对象中的数据。

例如，当我们从一个标准库 `string` 或容器对象移动数据时，我们知道移后源对象仍然保持有效。因此，我们可以对它执行诸如 `empty` 或 `size` 这些操作。但是，我们不知道将会得到什么结果。我们可能期望一个移后源对象是空的，但这并没有保证。

我们的 `StrVec` 类的移动操作将移后源对象置于与默认初始化的对象相同的状态。因此，我们可以继续对移后源对象执行所有的 `StrVec` 操作，与任何其他默认初始化的对象一样。而其他内部结构更为复杂的类，可能表现出完全不同的行为。



在移动操作之后，移后源对象必须保持有效的、可析构的状态，但是用户不能对其值进行任何假设。

合成的移动操作

与处理拷贝构造函数和拷贝赋值运算符一样，编译器也会合成移动构造函数和移动赋值运算符。但是，合成移动操作的条件与合成拷贝操作的条件大不相同。

回忆一下，如果我们不声明自己的拷贝构造函数或拷贝赋值运算符，编译器总会为我们合成这些操作（参见 13.1.1 节，第 440 页和 13.1.2 节，第 444 页）。拷贝操作要么被定义为逐成员拷贝，要么被定义为对象赋值，要么被定义为删除的函数。

与拷贝操作不同，编译器根本不会为某些类合成移动操作。特别是，如果一个类定义了自己的拷贝构造函数、拷贝赋值运算符或者析构函数，编译器就不会为它合成移动构造函数和移动赋值运算符了。因此，某些类就没有移动构造函数或移动赋值运算符。如我们将在第 477 页所见，如果一个类没有移动操作，通过正常的函数匹配，类会使用对应的拷

538

贝操作来代替移动操作。

只有当一个类没有定义任何自己版本的拷贝控制成员，且类的每个非 static 数据成员都可以移动时，编译器才会为它合成移动构造函数或移动赋值运算符。编译器可以移动内置类型的成员。如果一个成员是类类型，且该类有对应的移动操作，编译器也能移动这个成员：

```
// 编译器会为 X 和 hasX 合成移动操作
struct X {
    int i;           // 内置类型可以移动
    std::string s;  // string 定义了自己的移动操作
};

struct hasX {
    X mem; // X 有合成的移动操作
};

X x, x2 = std::move(x);           // 使用合成的移动构造函数
hasX hx, hx2 = std::move(hx);    // 使用合成的移动构造函数
```



只有当一个类没有定义任何自己版本的拷贝控制成员，且它的所有数据成员都能移动构造或移动赋值时，编译器才会为它合成移动构造函数或移动赋值运算符。

与拷贝操作不同，移动操作永远不会隐式定义为删除的函数。但是，如果我们显式地要求编译器生成`=default` 的（参见 7.1.4 节，第 237 页）移动操作，且编译器不能移动所有成员，则编译器会将移动操作定义为删除的函数。除了一个重要例外，什么时候将合成的移动操作定义为删除的函数遵循与定义删除的合成拷贝操作类似的原则（参见 13.1.6 节，第 449 页）：

- 与拷贝构造函数不同，移动构造函数被定义为删除的函数的条件是：有类成员定义了自己的拷贝构造函数且未定义移动构造函数，或者是有类成员未定义自己的拷贝构造函数且编译器不能为其合成移动构造函数。移动赋值运算符的情况类似。
- 如果有类成员的移动构造函数或移动赋值运算符被定义为删除的或是不可访问的，则类的移动构造函数或移动赋值运算符被定义为删除的。
- 类似拷贝构造函数，如果类的析构函数被定义为删除的或不可访问的，则类的移动构造函数被定义为删除的。
- 类似拷贝赋值运算符，如果有类成员是 `const` 的或是引用，则类的移动赋值运算符被定义为删除的。

539 例如，假定 Y 是一个类，它定义了自己的拷贝构造函数但未定义自己的移动构造函数：

```
// 假定 Y 是一个类，它定义了自己的拷贝构造函数但未定义自己的移动构造函数
struct hasY {
    hasY() = default;
    hasY(hasY&&) = default;
    Y mem; // hasY 将有一个删除的移动构造函数
};
hasY hy, hy2 = std::move(hy); // 错误：移动构造函数是删除的
```

编译器可以拷贝类型为 Y 的对象，但不能移动它们。类 `hasY` 显式地要求一个移动构造函数，但编译器无法为其生成。因此，`hasY` 会有一个删除的移动构造函数。如果 `hasY` 忽略了移动构造函数的声明，则编译器根本不能为它合成一个。如果移动操作可能被定义为

删除的函数，编译器就不会合成它们。

移动操作和合成的拷贝控制成员间还有最后一个相互作用关系：一个类是否定义了自己的移动操作对拷贝操作如何合成有影响。如果类定义了一个移动构造函数和/或一个移动赋值运算符，则该类的合成拷贝构造函数和拷贝赋值运算符会被定义为删除的。



定义了一个移动构造函数或移动赋值运算符的类必须也定义自己的拷贝操作。
否则，这些成员默认地被定义为删除的。

移动右值，拷贝左值……

如果一个类既有移动构造函数，也有拷贝构造函数，编译器使用普通的函数匹配规则来确定使用哪个构造函数（参见 6.4 节，第 208 页）。赋值操作的情况类似。例如，在我们的 StrVec 类中，拷贝构造函数接受一个 `const StrVec` 的引用。因此，它可用于任何可以转换为 `StrVec` 的类型。而移动构造函数接受一个 `StrVec&&`，因此只能用于实参是（非 `static`）右值的情形：

```
StrVec v1, v2;
v1 = v2;                                // v2 是左值；使用拷贝赋值
StrVec getVec(istream &);      // getVec 返回一个右值
v2 = getVec(cin);           // getVec(cin) 是一个右值；使用移动赋值
```

在第一个赋值中，我们将 `v2` 传递给赋值运算符。`v2` 的类型是 `StrVec`，表达式 `v2` 是一个左值。因此移动版本的赋值运算符是不可行的（参见 6.6 节，第 217 页），因为我们不能隐式地将一个右值引用绑定到一个左值。因此，这个赋值语句使用拷贝赋值运算符。

在第二个赋值中，我们赋予 `v2` 的是 `getVec` 调用的结果。此表达式是一个右值。在此情况下，两个赋值运算符都是可行的——将 `getVec` 的结果绑定到两个运算符的参数都是允许的。调用拷贝赋值运算符需要进行一次到 `const` 的转换，而 `StrVec&&` 则是精确匹配。因此，第二个赋值会使用移动赋值运算符。

……但如果没有移动构造函数，右值也被拷贝

540

如果一个类有一个拷贝构造函数但未定义移动构造函数，会发生什么呢？在此情况下，编译器不会合成移动构造函数，这意味着此类将有拷贝构造函数但不会有移动构造函数。如果一个类没有移动构造函数，函数匹配规则保证该类型的对象会被拷贝，即使我们试图通过调用 `move` 来移动它们时也是如此：

```
class Foo {
public:
    Foo() = default;
    Foo(const Foo&); // 拷贝构造函数
    // 其他成员定义，但 Foo 未定义移动构造函数
};

Foo x;
Foo y(x);           // 拷贝构造函数；x 是一个左值
Foo z(std::move(x)); // 拷贝构造函数，因为未定义移动构造函数
```

在对 `z` 进行初始化时，我们调用了 `move(x)`，它返回一个绑定到 `x` 的 `Foo&&`。`Foo` 的拷贝构造函数是可行的，因为我们可以将一个 `Foo&&` 转换为一个 `const Foo&`。因此，`z` 的初始化将使用 `Foo` 的拷贝构造函数。

值得注意的是，用拷贝构造函数代替移动构造函数几乎肯定是安全的（赋值运算符的

情况类似)。一般情况下，拷贝构造函数满足对应的移动构造函数的要求：它会拷贝给定对象，并将原对象置于有效状态。实际上，拷贝构造函数甚至都不会改变原对象的值。



如果一个类有一个可用的拷贝构造函数而没有移动构造函数，则其对象是通过拷贝构造函数来“移动”的。拷贝赋值运算符和移动赋值运算符的情况类似。

拷贝并交换赋值运算符和移动操作

我们的 `HasPtr` 版本定义了一个拷贝并交换赋值运算符（参见 13.3 节，第 459 页），它是函数匹配和移动操作间相互关系的一个很好的示例。如果我们为此类添加一个移动构造函数，它实际上也会获得一个移动赋值运算符：

```
class HasPtr {
public:
    // 添加的移动构造函数
    HasPtr(HasPtr &p) noexcept : ps(p.ps), i(p.i) { p.ps = 0; }
    // 赋值运算符既是移动赋值运算符，也是拷贝赋值运算符
    HasPtr& operator=(HasPtr rhs)
        { swap(*this, rhs); return *this; }
    // 其他成员的定义，同 13.2.1 节（第 453 页）
};
```

在这个版本中，我们为类添加了一个移动构造函数，它接管了给定实参的值。构造函数体将给定的 `HasPtr` 的指针置为 0，从而确保销毁移后源对象是安全的。此函数不会抛出异常，因此我们将其标记为 `noexcept`（参见 13.6.2 节，第 473 页）。

现在让我们观察赋值运算符。此运算符有一个非引用参数，这意味着此参数要进行拷贝初始化（参见 13.1.1 节，第 441 页）。依赖于实参的类型，拷贝初始化要么使用拷贝构造函数，要么使用移动构造函数——左值被拷贝，右值被移动。因此，单一的赋值运算符就实现了拷贝赋值运算符和移动赋值运算符两种功能。

例如，假定 `hp` 和 `hp2` 都是 `HasPtr` 对象：

```
hp = hp2; // hp2 是一个左值；hp2 通过拷贝构造函数来拷贝
hp = std::move(hp2); // 移动构造函数移动 hp2
```

在第一个赋值中，右侧运算对象是一个左值，因此移动构造函数是不可行的。`rhs` 将使用拷贝构造函数来初始化。拷贝构造函数将分配一个新 `string`，并拷贝 `hp2` 指向的 `string`。

在第二个赋值中，我们调用 `std::move` 将一个右值引用绑定到 `hp2` 上。在此情况下，拷贝构造函数和移动构造函数都是可行的。但是，由于实参是一个右值引用，移动构造函数是精确匹配的。移动构造函数从 `hp2` 拷贝指针，而不会分配任何内存。

不管使用的是拷贝构造函数还是移动构造函数，赋值运算符的函数体都 `swap` 两个运算对象的状态。交换 `HasPtr` 会交换两个对象的指针（及 `int`）成员。在 `swap` 之后，`rhs` 中的指针将指向原来左侧运算对象所拥有的 `string`。当 `rhs` 离开其作用域时，这个 `string` 将被销毁。

建议：更新三/五法则

所有五个拷贝控制成员应该看作一个整体：一般来说，如果一个类定义了任何一个

拷贝操作，它就应该定义所有五个操作。如前所述，某些类必须定义拷贝构造函数、拷贝赋值运算符和析构函数才能正确工作（参见 13.1.4 节，第 447 页）。这些类通常拥有一个资源，而拷贝成员必须拷贝此资源。一般来说，拷贝一个资源会导致一些额外开销。在这种拷贝并非必要的情况下，定义了移动构造函数和移动赋值运算符的类就可以避免此问题。

Message 类的移动操作

定义了自己的拷贝构造函数和拷贝赋值运算符的类通常也会从移动操作受益。例如，我们的 Message 和 Folder 类（参见 13.4 节，第 460 页）就应该定义移动操作。通过定义移动操作，Message 类可以使用 string 和 set 的移动操作来避免拷贝 contents 和 folders 成员的额外开销。

但是，除了移动 folders 成员，我们还必须更新每个指向原 Message 的 Folder。我们必须删除指向旧 Message 的指针，并添加一个指向新 Message 的指针。

移动构造函数和移动赋值运算符都需要更新 Folder 指针，因此我们首先定义一个操作来完成这一共同的工作：

```
// 从本 Message 移动 Folder 指针
void Message::move_Folders(Message *m)
{
    folders = std::move(m->folders); // 使用 set 的移动赋值运算符
    for (auto f : folders) { // 对每个 Folder
        f->remMsg(m); // 从 Folder 中删除旧 Message
        f->addMsg(this); // 将本 Message 添加到 Folder 中
    }
    m->folders.clear(); // 确保销毁 m 是无害的
}
```

此函数首先移动 folders 集合。通过调用 move，我们使用了 set 的移动赋值运算符而不是它的拷贝赋值运算符。如果我们忽略了 move 调用，代码仍能正常工作，但带来了不必要的拷贝。函数然后遍历所有 Folder，从其中删除指向原 Message 的指针并添加指向新 Message 的指针。

值得注意的是，向 set 插入一个元素可能会抛出一个异常——向容器添加元素的操作要求分配内存，意味着可能会抛出一个 bad_alloc 异常（参见 12.1.2 节，第 409 页）。因此，与我们的 HasPtr 和 StrVec 类的移动操作不同，Message 的移动构造函数和移动赋值运算符可能会抛出异常。因此我们未将它们标记为 noexcept（参见 13.6.2 节，第 473 页）。

函数最后对 m.folders 调用 clear。在执行了 move 之后，我们知道 m.folders 是有效的，但不知道它包含什么内容。由于 Message 的析构函数遍历 folders，我们希望能确定 set 是空的。

Message 的移动构造函数调用 move 来移动 contents，并默认初始化自己的 folders 成员：

```
Message::Message(Message &m) : contents(std::move(m.contents))
{
    move_Folders(&m); // 移动 folders 并更新 Folder 指针
}
```

在构造函数体中，我们调用了 `move_Folders` 来删除指向 `m` 的指针并插入指向本 `Message` 的指针。

移动赋值运算符直接检查自赋值情况：

```
Message& Message::operator=(Message &&rhs)
{
    if (this != &rhs) {           // 直接检查自赋值情况
        remove_from_Folders();
        contents = std::move(rhs.contents); // 移动赋值运算符
        move_Folders(&rhs);   // 重置 Folders 指向本 Message
    }
    return *this;
}
```

543 与任何赋值运算符一样，移动赋值运算符必须销毁左侧运算对象的旧状态。在本例中，销毁左侧运算对象要求我们从现有 `folders` 中删除指向本 `Message` 的指针，我们调用 `remove_from_Folders` 来完成这一工作。完成删除工作后，我们调用 `move` 从 `rhs` 将 `contents` 移动到 `this` 对象。剩下的就是调用 `move_Messages` 来更新 `Folder` 指针了。

移动迭代器

`StrVec` 的 `reallocate` 成员（参见 13.5 节，第 469 页）使用了一个 `for` 循环来调用 `construct` 从旧内存将元素拷贝到新内存中。作为一种替换方法，如果我们能调用 `uninitialized_copy` 来构造新分配的内存，将比循环更为简单。但是，`uninitialized_copy` 恰如其名：它对元素进行拷贝操作。标准库中并没有类似的函数将对象“移动”到未构造的内存中。

C++ 11 新标准库中定义了一种**移动迭代器**（move iterator）适配器（参见 10.4 节，第 358 页）。一个移动迭代器通过改变给定迭代器的解引用运算符的行为来适配此迭代器。一般来说，一个迭代器的解引用运算符返回一个指向元素的左值。与其他迭代器不同，移动迭代器的解引用运算符生成一个右值引用。

我们通过调用标准库的 `make_move_iterator` 函数将一个普通迭代器转换为一个移动迭代器。此函数接受一个迭代器参数，返回一个移动迭代器。

原迭代器的所有其他操作在移动迭代器中都照常工作。由于移动迭代器支持正常的迭代器操作，我们可以将一对移动迭代器传递给算法。特别是，可以将移动迭代器传递给 `uninitialized_copy`：

```
void StrVec::reallocate()
{
    // 分配大小两倍于当前规模的内存空间
    auto newcapacity = size() ? 2 * size() : 1;
    auto first = alloc.allocate(newcapacity);
    // 移动元素
    auto last = uninitialized_copy(make_move_iterator(begin()),
                                    make_move_iterator(end()),
                                    first);
    free();           // 释放旧空间
    elements = first; // 更新指针
    first_free = last;
```

```
    cap = elements + newcapacity;
}
```

uninitialized_copy 对输入序列中的每个元素调用 construct 来将元素“拷贝”到目的位置。此算法使用迭代器的解引用运算符从输入序列中提取元素。由于我们传递给它的是移动迭代器，因此解引用运算符生成的是一个右值引用，这意味着 construct 将使用移动构造函数来构造元素。

值得注意的是，标准库不保证哪些算法适用移动迭代器，哪些不适用。由于移动一个对象可能销毁掉原对象，因此你只有在确信算法在为一个元素赋值或将其传递给一个用户定义的函数后不再访问它时，才能将移动迭代器传递给算法。544

建议：不要随意使用移动操作

由于一个移后源对象具有不确定的状态，对其调用 std::move 是危险的。当我们调用 move 时，必须绝对确认移后源对象没有其他用户。

通过在类代码中小心地使用 move，可以大幅度提升性能。而如果随意在普通用户代码（与类实现代码相对）中使用移动操作，很可能导致莫名其妙的、难以查找的错误，而难以提升应用程序性能。

Best Practices 在移动构造函数和移动赋值运算符这些类实现代码之外的地方，只有当你确信需要进行移动操作且移动操作是安全的，才可以使用 std::move。

13.6.2 节练习

练习 13.49：为你的 StrVec、String 和 Message 类添加一个移动构造函数和一个移动赋值运算符。

练习 13.50：在你的 String 类的移动操作中添加打印语句，并重新运行 13.6.1 节（第 473 页）的练习 13.48 中的程序，它使用了一个 vector<String>，观察什么时候会避免拷贝。

练习 13.51：虽然 unique_ptr 不能拷贝，但我们在 12.1.5 节（第 418 页）中编写了一个 clone 函数，它以值方式返回一个 unique_ptr。解释为什么函数是合法的，以及为什么它能正确工作。

练习 13.52：详细解释第 478 页中的 HasPtr 对象的赋值发生了什么？特别是，一步一步描述 hp、hp2 以及 HasPtr 的赋值运算符中的参数 rhs 的值发生了什么变化。

练习 13.53：从底层效率的角度看，HasPtr 的赋值运算符并不理想，解释为什么。为 HasPtr 实现一个拷贝赋值运算符和一个移动赋值运算符，并比较你的新的移动赋值运算符中执行的操作和拷贝并交换版本中执行的操作。

练习 13.54：如果我们为 HasPtr 定义了移动赋值运算符，但未改变拷贝并交换运算符，会发生什么？编写代码验证你的答案。

13.6.3 右值引用和成员函数

除了构造函数和赋值运算符之外，如果一个成员函数同时提供拷贝和移动版本，它也能从中受益。这种允许移动的成员函数通常使用与拷贝/移动构造函数和赋值运算符相同的参数模式——一个版本接受一个指向 const 的左值引用，第二个版本接受一个指向非

`const` 的右值引用。

例如，定义了 `push_back` 的标准库容器提供两个版本：一个版本有一个右值引用参数，而另一个版本有一个 `const` 左值引用。假定 `X` 是元素类型，那么这些容器就会定义以下两个 `push_back` 版本：

```
void push_back(const X&);      // 拷贝：绑定到任意类型的 X
void push_back(X&&);         // 移动：只能绑定到类型 X 的可修改的右值
```

我们可以将能转换为类型 `X` 的任何对象传递给第一个版本的 `push_back`。此版本从其参数拷贝数据。对于第二个版本，我们只可以传递给它非 `const` 的右值。此版本对于非 `const` 的右值是精确匹配（也是更好的匹配）的，因此当我们传递一个可修改的右值（参见 13.6.2 节，第 477 页）时，编译器会选择运行这个版本。此版本会从其参数窃取数据。

一般来说，我们不需要为函数操作定义接受一个 `const X&&` 或是一个（普通的）`X&` 参数的版本。当我们希望从实参“窃取”数据时，通常传递一个右值引用。为了达到这一目的，实参不能是 `const` 的。类似的，从一个对象进行拷贝的操作不应该改变该对象。因此，通常不需要定义一个接受一个（普通的）`X&` 参数的版本。



区分移动和拷贝的重载函数通常有一个版本接受一个 `const T&`，而另一个版本接受一个 `T&&`。

作为一个更具体的例子，我们将为 `StrVec` 类定义另一个版本的 `push_back`：

```
class StrVec {
public:
    void push_back(const std::string&); // 拷贝元素
    void push_back(std::string&&);     // 移动元素
    // 其他成员的定义，如前
};

// 与 13.5 节（第 466 页）中的原版本相同
void StrVec::push_back(const string &s)
{
    chk_n_alloc(); // 确保有空间容纳新元素
    // 在 first_free 指向的元素中构造 s 的一个副本
    alloc.construct(first_free++, s);
}
void StrVec::push_back(string &&s)
{
    chk_n_alloc(); // 如果需要的话为 StrVec 重新分配内存
    alloc.construct(first_free++, std::move(s));
}
```

这两个成员几乎是相同的。差别在于右值引用版本调用 `move` 来将其参数传递给 `construct`。如前所述，`construct` 函数使用其第二个和随后的实参的类型来确定使用哪个构造函数。由于 `move` 返回一个右值引用，传递给 `construct` 的实参类型是 `string&&`。因此，会使用 `string` 的移动构造函数来构造新元素。

当我们调用 `push_back` 时，实参类型决定了新元素是拷贝还是移动到容器中：

```
StrVec vec; // 空 StrVec
string s = "some string or another";
vec.push_back(s); // 调用 push_back(const string&)
```

```
vec.push_back("done"); // 调用 push_back(string&&)
```

这些调用的差别在于实参是一个左值还是一个右值（从"done"创建的临时 string），具体调用哪个版本据此来决定。

右值和左值引用成员函数

通常，我们在一个对象上调用成员函数，而不管该对象是一个左值还是一个右值。例如：

```
string s1 = "a value", s2 = "another";
auto n = (s1 + s2).find('a');
```

此例中，我们在一个 string 右值上调用 find 成员（参见 9.5.3 节，第 325 页），该 string 右值是通过连接两个 string 而得到的。有时，右值的使用方式可能令人惊讶：

```
s1 + s2 = "wow!";
```

此处我们对两个 string 的连接结果——一个右值，进行了赋值。

在旧标准中，我们没有办法阻止这种使用方式。为了维持向后兼容性，新标准库类仍然允许向右值赋值。但是，我们可能希望在自己的类中阻止这种用法。在此情况下，我们希望强制左侧运算对象（即，this 指向的对象）是一个左值。

我们指出 this 的左值/右值属性的方式与定义 const 成员函数相同（参见 7.1.2 节，C++ 11 第 231 页），即，在参数列表后放置一个引用限定符（reference qualifier）：

```
class Foo {
public:
    Foo &operator=(const Foo&) &; // 只能向可修改的左值赋值
    // Foo 的其他参数
};

Foo &Foo::operator=(const Foo &rhs) &
{
    // 执行将 rhs 赋予本对象所需的工作
    return *this;
}
```

引用限定符可以是&或&&，分别指出 this 可以指向一个左值或右值。类似 const 限定符，引用限定符只能用于（非 static）成员函数，且必须同时出现在函数的声明和定义中。

对于&限定的函数，我们只能将它用于左值；对于&&限定的函数，只能用于右值：

```
Foo &retFoo(); // 返回一个引用；retFoo 调用是一个左值
Foo retVal(); // 返回一个值；retVal 调用是一个右值
Foo i, j; // i 和 j 是左值
i = j; // 正确：i 是左值
retFoo() = j; // 正确：retFoo() 返回一个左值
retVal() = j; // 错误：retVal() 返回一个右值
i = retVal(); // 正确：我们可以将一个右值作为赋值操作的右侧运算对象
```

一个函数可以同时用 const 和引用限定。在此情况下，引用限定符必须跟随在 const 限定符之后：

```
class Foo {
public:
    Foo someMem() & const; // 错误：const 限定符必须在前
    Foo anotherMem() const &; // 正确：const 限定符在前
};
```

重载和引用函数

就像一个成员函数可以根据是否有 `const` 来区分其重载版本一样（参见 7.3.2 节，第 247 页），引用限定符也可以区分重载版本。而且，我们可以综合引用限定符和 `const` 来区分一个成员函数的重载版本。例如，我们将为 `Foo` 定义一个名为 `data` 的 `vector` 成员和一个名为 `sorted` 的成员函数，`sorted` 返回一个 `Foo` 对象的副本，其中 `vector` 已被排序：

```
class Foo {
public:
    Foo sorted() &&;           // 可用于可改变的右值
    Foo sorted() const &;      // 可用于任何类型的 Foo
    // Foo 的其他成员的定义
private:
    vector<int> data;
};

// 本对象为右值，因此可以原址排序
Foo Foo::sorted() &&
{
    sort(data.begin(), data.end());
    return *this;
}
// 本对象是 const 或是一个左值，哪种情况我们都不能对其进行原址排序
Foo Foo::sorted() const & {
    Foo ret(*this);           // 拷贝一个副本
    sort(ret.data.begin(), ret.data.end()); // 排序副本
    return ret;               // 返回副本
}
```

当我们对一个右值执行 `sorted` 时，它可以安全地直接对 `data` 成员进行排序。对象是一个右值，意味着没有其他用户，因此我们可以改变对象。当对一个 `const` 右值或一个左值执行 `sorted` 时，我们不能改变对象，因此就需要在排序前拷贝 `data`。

编译器会根据调用 `sorted` 的对象的左值/右值属性来确定使用哪个 `sorted` 版本：

548 `retVal().sorted();` // `retVal()` 是一个右值，调用 `Foo::sorted() &&`
`retFoo().sorted();` // `retFoo()` 是一个左值，调用 `Foo::sorted() const &`

当我们定义 `const` 成员函数时，可以定义两个版本，唯一的差别是一个版本有 `const` 限定而另一个没有。引用限定的函数则不一样。如果我们定义两个或两个以上具有相同名字和相同参数列表的成员函数，就必须对所有函数都加上引用限定符，或者所有都不加：

```
class Foo {
public:
    Foo sorted() &&;
    Foo sorted() const; // 错误：必须加上引用限定符
    // Comp 是函数类型的类型别名（参见 6.7 节，第 222 页）
    // 此函数类型可以用来比较 int 值
    using Comp = bool(const int&, const int&);
    Foo sorted(Comp*);           // 正确：不同的参数列表
    Foo sorted(Comp*) const;     // 正确：两个版本都没有引用限定符
};
```

本例中声明了一个没有参数的 `const` 版本的 `sorted`, 此声明是错误的。因为 `Foo` 类中还有一个无参的 `sorted` 版本, 它有一个引用限定符, 因此 `const` 版本也必须有引用限定符。另一方面, 接受一个比较操作指针的 `sorted` 版本是没问题的, 因为两个函数都没有引用限定符。



如果一个成员函数有引用限定符, 则具有相同参数列表的所有版本都必须有引用限定符。

13.6.3 节练习

练习 13.55: 为你的 `StrBlob` 添加一个右值引用版本的 `push_back`。

练习 13.56: 如果 `sorted` 定义如下, 会发生什么:

```
Foo Foo::sorted() const & {
    Foo ret(*this);
    return ret.sorted();
}
```

练习 13.57: 如果 `sorted` 定义如下, 会发生什么:

```
Foo Foo::sorted() const & { return Foo(*this).sorted(); }
```

练习 13.58: 编写新版本的 `Foo` 类, 其 `sorted` 函数中有打印语句, 测试这个类, 来验证你对前两题的答案是否正确。

549

小结

每个类都会控制该类型对象拷贝、移动、赋值以及销毁时发生什么。特殊的成员函数——拷贝构造函数、移动构造函数、拷贝赋值运算符、移动赋值运算符和析构函数定义了这些操作。移动构造函数和移动赋值运算符接受一个（通常是非 `const` 的）右值引用；而拷贝版本则接受一个（通常是 `const` 的）普通左值引用。

如果一个类未声明这些操作，编译器会自动为其生成。如果这些操作未定义成删除的，它们会逐成员初始化、移动、赋值或销毁对象：合成的操作依次处理每个非 `static` 数据成员，根据成员类型确定如何移动、拷贝、赋值或销毁它。

分配了内存或其他资源的类几乎总是需要定义拷贝控制成员来管理分配的资源。如果一个类需要析构函数，则它几乎肯定也需要定义移动和拷贝构造函数及移动和拷贝赋值运算符。

术语表

拷贝并交换（copy and swap） 涉及赋值运算符的技术，首先拷贝右侧运算对象，然后调用 `swap` 来交换副本和左侧运算对象。

拷贝赋值运算符（copy-assignment operator） 接受一个本类型对象的赋值运算符版本。通常，拷贝赋值运算符的参数是一个 `const` 的引用，并返回指向本对象的引用。如果类未显式定义拷贝赋值运算符，编译器会为它合成一个。

拷贝构造函数（copy constructor） 一种构造函数，将新对象初始化为同类型另一个对象的副本。当向函数传递对象，或以传值方式从函数返回对象时，会隐式使用拷贝构造函数。如果我们未提供拷贝构造函数，编译器会为我们合成一个。

拷贝控制（copy control） 特殊的成员函数，控制拷贝、移动、赋值及销毁本类类型对象时发生什么。如果类未定义这些操作，编译器会为它合成恰当的定义。

拷贝初始化（copy initialization） 一种初始化形式，当我们使用`=`为一个新创建的对象提供初始化器时，会使用拷贝初始化。如果我们向函数传递对象或以传值方式从函数返回对象，以及初始化一个数组或一个聚合类时，也会使用拷贝初始化。

删除的函数（deleted function） 不能使用的函数。我们在一个函数的声明上指定`=delete` 来删除它。删除的函数的一个常见用途是告诉编译器不要为类合成拷贝和/或移动操作。

析构函数（destructor） 特殊的成员函数，当对象离开作用域或被释放时进行清理工作。编译器会自动销毁每个数据成员。类类型的成员通过调用其析构函数来销毁；而内置类型或复合类型的成员的销毁则不需要做任何工作。特别是，析构函数不会释放指针成员指向的对象。

左值引用（lvalue reference） 可以绑定到左值的引用。

逐成员拷贝 / 赋值（memberwise copy/assign） 合成的拷贝与移动构造函数及拷贝与移动赋值运算符的工作方式。合成的拷贝或移动构造函数依次处理每个非 `static` 数据成员，通过从给定对象拷贝或移动对应成员来初始化本对象成员；拷贝或移动赋值运算符从右侧运算对象中将每个成员拷贝赋值或移动赋值到左侧运算对象中。内置类型或复合类型的成员直接进行初始化或赋值。类类型的成员通过成员对应的拷贝/移动构造函数或拷贝/移动赋值运算符进行初始化或赋值。

move 用来将一个右值引用绑定到一个左值的标准库函数。调用 `move` 隐含地承诺我们将不会再使用移后源对象，除了销毁它或赋予它一个新值之外。

移动赋值运算符（move-assignment operator） 接受一个本类型右值引用参数的赋值运算符版本。通常，移动赋值运算符将数据从右侧运算对象移动到左侧运算对象。赋值之后，对右侧运算对象执行析构函数必须是安全的。

移动构造函数（move constructor） 一种构造函数，接受一个本类型的右值引用。通常，移动构造函数将数据从其参数移动到新创建的对象中。移动之后，对给定的实参执行析构函数必须是安全的。

移动迭代器（move iterator） 迭代器适配器，它生成的迭代器在解引用时会得到一个右值引用。

重载运算符（overloaded operator） 一种函数，重定义了运算符应用于类类型的对象时的含义。本章介绍了如何定义赋值运算符；第 14 章中将介绍重载运算符的更多细节内容。

引用计数（reference count） 一种程序设计技术，通常用于拷贝控制成员的设计。引用计数记录了有多少对象共享状态。构造函数（不是拷贝/移动构造函数）将引用计数置为 1。每当创建一个新副本时，计数

值递增。当一个对象被销毁时，计数值递减。赋值运算符和析构函数检查递减的引用计数是否为 0，如果是，它们会销毁对象。

引用限定符（reference qualifier） 用来指出一个非 `static` 成员函数可以用于左值或右值的符号。限定符`&`和`&&`应该放在参数列表之后或 `const` 限定符之后（如果有的话）。被`&`限定的函数只能用于左值；被`&&`限定的函数只能用于右值。

右值引用（rvalue reference） 指向一个将要销毁的对象的引用。

合成赋值运算符（synthesized assignment operator） 编译器为未显式定义赋值运算符的类创建的（合成的）拷贝或移动赋值运算符版本。除非定义为删除的，合成赋值运算符会逐成员地将右侧运算对象赋予（移动到）左侧运算对象。

合成拷贝/移动构造函数（synthesized copy/move constructor） 编译器为未显式定义对应的构造函数的类生成的拷贝或移动构造函数版本。除非定义为删除的，合成拷贝或移动构造函数分别通过从给定对象拷贝或移动成员来逐成员地初始化新对象。

合成析构函数（synthesized destructor） 编译器为未显式定义析构函数的类创建的（合成的）版本。合成析构函数的函数体为空。

第 14 章

重载运算与类型转换

内容

14.1 基本概念	490
14.2 输入和输出运算符	494
14.3 算术和关系运算符	497
14.4 赋值运算符	499
14.5 下标运算符	501
14.6 递增和递减运算符	502
14.7 成员访问运算符	504
14.8 函数调用运算符	506
14.9 重载、类型转换与运算符	514
小结	523
术语表	523

在第 4 章中我们看到，C++语言定义了大量运算符以及内置类型的自动转换规则。这些特性使得程序员能编写出形式丰富、含有多种混合类型的表达式。

当运算符被用于类类型的对象时，C++语言允许我们为其指定新的含义；同时，我们也能自定义类类型之间的转换规则。和内置类型的转换一样，类类型转换隐式地将一种类型的对象转换成另一种我们所需类型的对象。

552 当运算符作用于类类型的运算对象时，可以通过运算符重载重新定义该运算符的含义。明智地使用运算符重载能令我们的程序更易于编写和阅读。举个例子，因为在 Sales_item 类（参见 1.5.1 节，第 17 页）中定义了输入、输出和加法运算符，所以可以通过下述形式输出两个 Sales_item 的和：

```
cout << item1 + item2; // 输出两个 Sales_item 的和
```

相反的，由于我们的 Sales_data 类（参见 7.1 节，第 228 页）还没有重载这些运算符，因此它的加法代码显得比较冗长而不清晰：

```
print(cout, add(data1, data2)); // 输出两个 Sales_data 的和
```



14.1 基本概念

重载的运算符是具有特殊名字的函数：它们的名字由关键字 operator 和其后要定义的运算符号共同组成。和其他函数一样，重载的运算符也包含返回类型、参数列表以及函数体。

重载运算符函数的参数数量与该运算符作用的运算对象数量一样多。一元运算符有一个参数，二元运算符有两个。对于二元运算符来说，左侧运算对象传递给第一个参数，而右侧运算对象传递给第二个参数。除了重载的函数调用运算符 operator() 之外，其他重载运算符不能含有默认实参（参见 6.5.1 节，第 211 页）。

如果一个运算符函数是成员函数，则它的第一个（左侧）运算对象绑定到隐式的 this 指针上（参见 7.1.2 节，第 231 页），因此，成员运算符函数的（显式）参数数量比运算符的运算对象总数少一个。



当一个重载的运算符是成员函数时，this 绑定到左侧运算对象。成员运算符函数的（显式）参数数量比运算对象的数量少一个。

对于一个运算符函数来说，它或者是类的成员，或者至少含有一个类类型的参数：

```
// 错误：不能为 int 重定义内置的运算符  
int operator+(int, int);
```

这一约定意味着当运算符作用于内置类型的运算对象时，我们无法改变该运算符的含义。

我们可以重载大多数（但不是全部）运算符。表 14.1 指明了哪些运算符可以被重载，哪些不行。我们将在 19.1.1 节（第 726 页）介绍重载 new 和 delete 的方法。

我们只能重载已有的运算符，而无权发明新的运算符号。例如，我们不能提供 operator** 来执行幂操作。

有四个符号（+、-、*、&）既是一元运算符也是二元运算符，所有这些运算符都能被重载，从参数的数量我们可以推断到底定义的是哪种运算符。

553 对于一个重载的运算符来说，其优先级和结合律（参见 4.1.2 节，第 121 页）与对应的内置运算符保持一致。不考虑运算对象类型的话，

```
x == y + z;
```

永远等价于 $x == (y + z)$ 。

表 14.1: 运算符

可以被重载的运算符						
+	-	*	/	%	^	
&		~	!	,	=	
<	>	<=	>=	++	--	
<<	>>	==	!=	&&		
+=	-=	/=	%=	^=	&=	
=	*=	<<=	>>=	[]	()	
->	->*	new	new[]	delete	delete[]	
不能被重载的运算符						
:	.*	.	.	?:		

直接调用一个重载的运算符函数

通常情况下，我们将运算符作用于类型正确的实参，从而以这种间接方式“调用”重载的运算符函数。然而，我们也能像调用普通函数一样直接调用运算符函数，先指定函数名字，然后传入数量正确、类型适当的实参：

```
// 一个非成员运算符函数的等价调用
data1 + data2;                                // 普通的表达式
operator+(data1, data2);                      // 等价的函数调用
```

这两次调用是等价的，它们都调用了非成员函数 `operator+`，传入 `data1` 作为第一个实参、传入 `data2` 作为第二个实参。

我们像调用其他成员函数一样显式地调用成员运算符函数。具体做法是，首先指定运行函数的对象（或指针）的名字，然后使用点运算符（或箭头运算符）访问希望调用的函数：

```
data1 += data2;                                // 基于“调用”的表达式
data1.operator+=(data2);                      // 对成员运算符函数的等价调用
```

这两条语句都调用了成员函数 `operator+=`，将 `this` 绑定到 `data1` 的地址、将 `data2` 作为实参传入了函数。

某些运算符不应该被重载

回忆之前介绍过的，某些运算符指定了运算对象求值的顺序。因为使用重载的运算符本质上是一次函数调用，所以这些关于运算对象求值顺序的规则无法应用到重载的运算符上。特别是，逻辑与运算符、逻辑或运算符（参见 4.3 节，第 126 页）和逗号运算符（参见 4.10 节，第 140 页）的运算对象求值顺序规则无法保留下来。除此之外，`&&` 和 `||` 运算符的重载版本也无法保留内置运算符的短路求值属性，两个运算对象总是会被求值。554

因为上述运算符的重载版本无法保留求值顺序和/或短路求值属性，因此不建议重载它们。当代码使用了这些运算符的重载版本时，用户可能会突然发现他们一直习惯的求值规则不再适用了。

还有一个原因使得我们一般不重载逗号运算符和取地址运算符：C++语言已经定义了这两种运算符用于类类型对象时的特殊含义，这一点与大多数运算符都不相同。因为这两种运算符已经有了内置的含义，所以一般来说它们不应该被重载，否则它们的行为将异于常态，从而导致类的用户无法适应。

Best Practices

通常情况下，不应该重载逗号、取地址、逻辑与和逻辑或运算符。

使用与内置类型一致的含义

当你开始设计一个类时，首先应该考虑的是这个类将提供哪些操作。在确定类需要哪些操作之后，才能思考到底应该把每个类操作设成普通函数还是重载的运算符。如果某些操作在逻辑上与运算符相关，则它们适合于定义成重载的运算符：

- 如果类执行 IO 操作，则定义移位运算符使其与内置类型的 IO 保持一致。
- 如果类的某个操作是检查相等性，则定义 `operator==`；如果类有了 `operator==`，意味着它通常也应该有 `operator!=`。
- 如果类包含一个内在的单序比较操作，则定义 `operator<`；如果类有了 `operator<`，则它也应该含有其他关系操作。
- 重载运算符的返回类型通常情况下应该与其内置版本的返回类型兼容：逻辑运算符和关系运算符应该返回 `bool`，算术运算符应该返回一个类类型的值，赋值运算符和复合赋值运算符则应该返回左侧运算对象的一个引用。

提示：尽量明智地使用运算符重载

每个运算符在用于内置类型时都有比较明确的含义。以二元`+`运算符为例，它明显执行的是加法操作。因此，把二元`+`运算符映射到类类型的一个类似操作上可以极大地简化记忆。例如对于标准库类型 `string` 来说，我们就会使用`+`把一个 `string` 对象连接到另一个后面，很多编程语言都有类似的用法。

当在内置的运算符和我们自己的操作之间存在逻辑映射关系时，运算符重载的效果最好。此时，使用重载的运算符显然比另起一个名字更自然也更直观。不过，过分滥用运算符重载也会使我们的类变得难以理解。

在实际编程过程中，一般没有特别明显的滥用运算符重载的情况。例如，一般来说没有哪个程序员会定义 `operator+` 并让它执行减法操作。然而经常发生的一种情况是，程序员可能会强行扭曲了运算符的“常规”含义使得其适应某种给定的类型，这显然是我们不希望发生的。因此我们的建议是：只有当操作的含义对于用户来说清晰明了时才使用运算符。如果用户对运算符可能有几种不同的理解，则使用这样的运算符将产生二义性。

赋值和复合赋值运算符

赋值运算符的行为与复合版本的类似：赋值之后，左侧运算对象和右侧运算对象的值相等，并且运算符应该返回它左侧运算对象的一个引用。重载的赋值运算应该继承而非违背其内置版本的含义。

如果类含有算术运算符（参见 4.2 节，第 124 页）或者位运算符（参见 4.8 节，第 136 页），则最好也提供对应的复合赋值运算符。无须赘言，`+=` 运算符的行为显然应该与其内置版本一致，即先执行`+`，再执行`=`。

选择作为成员或者非成员

当我们定义重载的运算符时，必须首先决定是将其声明为类的成员函数还是声明为一个普通的非成员函数。在某些时候我们别无选择，因为有的运算符必须作为成员；另一些

情况下，运算符作为普通函数比作为成员更好。

下面的准则有助于我们在将运算符定义为成员函数还是普通的非成员函数做出抉择：

- 赋值（=）、下标（[]）、调用（()）和成员访问箭头（->）运算符必须是成员。
- 复合赋值运算符一般来说应该是成员，但并非必须，这一点与赋值运算符略有不同。
- 改变对象状态的运算符或者与给定类型密切相关的运算符，如递增、递减和解引用运算符，通常应该是成员。
- 具有对称性的运算符可能转换任意一端的运算对象，例如算术、相等性、关系和位运算符等，因此它们通常应该是普通的非成员函数。

程序员希望能在含有混合类型的表达式中使用对称性运算符。例如，我们能求一个 int 和一个 double 的和，因为它们中的任意一个都可以是左侧运算对象或右侧运算对象，所以加法是对称的。如果我们想提供含有类对象的混合类型表达式，则运算符必须定义成非成员函数。556

当我们把运算符定义成成员函数时，它的左侧运算对象必须是运算符所属类的一个对象。例如：

```
string s = "world";
string t = s + "!"; // 正确：我们能把一个 const char* 加到一个 string 对象中
string u = "hi" + s; // 如果+是 string 的成员，则产生错误
```

如果 operator+ 是 string 类的成员，则上面的第一个加法等价于 s.operator+("!"）。同样的，"hi"+s 等价于 "hi".operator+(s)。显然 "hi" 的类型是 const char*，这是一种内置类型，根本就没有成员函数。

因为 string 将 + 定义成了普通的非成员函数，所以 "hi"+s 等价于 operator+("hi", s)。和任何其他函数调用一样，每个实参都能被转换成形参类型。唯一的要求是至少有一个运算对象是类类型，并且两个运算对象都能准确无误地转换成 string。

14.1 节练习

练习 14.1：在什么情况下重载的运算符与内置运算符有所区别？在什么情况下重载的运算符又与内置运算符一样？

练习 14.2：为 Sales_data 编写重载的输入、输出、加法和复合赋值运算符。

练习 14.3： string 和 vector 都定义了重载的 == 以比较各自的对象，假设 svec1 和 svec2 是存放 string 的 vector，确定在下面的表达式中分别使用了哪个版本的 == ？

- | | |
|-------------------------|--------------------------|
| (a) "cobble" == "stone" | (b) svec1[0] == svec2[0] |
| (c) svec1 == svec2 | (d) "svec1[0] == "stone" |

练习 14.4：如何确定下列运算符是否应该是类的成员？

- (a) % (b) %= (c) ++ (d) -> (e) << (f) && (g) == (h) ()

练习 14.5：在 7.5.1 节的练习 7.40（第 261 页）中，编写了下列类中某一个的框架，请问在这个类中应该定义重载的运算符吗？如果是，请写出来。

- | | | |
|-------------|------------|--------------|
| (a) Book | (b) Date | (c) Employee |
| (d) Vehicle | (e) Object | (f) Tree |

14.2 输入和输出运算符

如我们所知，IO 标准库分别使用`>>`和`<<`执行输入和输出操作。对于这两个运算符来说，IO 库定义了用其读写内置类型的版本，而类则需要自定义适合其对象的新版本以支持 IO 操作。

14.2.1 重载输出运算符`<<`

通常情况下，输出运算符的第一个形参是一个非常量 `ostream` 对象的引用。之所以 `ostream` 是非常量是因为向流写入内容会改变其状态；而该形参是引用是因为我们无法直接复制一个 `ostream` 对象。

第二个形参一般来说是一个常量的引用，该常量是我们想要打印的类类型。第二个形参是引用的原因是我们希望避免复制实参；而之所以该形参可以是常量是因为（通常情况下）打印对象不会改变对象的内容。

为了与其他输出运算符保持一致，`operator<<`一般要返回它的 `ostream` 形参。

Sales_data 的输出运算符

举个例子，我们按照如下形式编写 `Sales_data` 的输出运算符：

```
ostream &operator<<(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
        << item.revenue << " " << item.avg_price();
    return os;
}
```

除了名字之外，这个函数与之前的 `print` 函数（参见 7.1.3 节，第 234 页）完全一样。打印一个 `Sales_data` 对象意味着要分别打印它的三个数据成员以及通过计算得到的平均销售价格，每个元素以空格隔开。完成输出后，运算符返回刚刚使用的 `ostream` 的引用。

输出运算符尽量减少格式化操作

用于内置类型的输出运算符不太考虑格式化操作，尤其不会打印换行符，用户希望类的输出运算符也像如此行事。如果运算符打印了换行符，则用户就无法在对象的同一行内接着打印一些描述性的文本了。相反，令输出运算符尽量减少格式化操作可以使用户有权控制输出的细节。

Best Practices

通常，输出运算符应该主要负责打印对象的内容而非控制格式，输出运算符不应该打印换行符。

输入输出运算符必须是非成员函数

与 `iostream` 标准库兼容的输入输出运算符必须是普通的非成员函数，而不能是类的成员函数。否则，它们的左侧运算对象将是我们的类的一个对象：

```
Sales_data data;
data << cout;           // 如果 operator<< 是 Sales_data 的成员
```

假设输入输出运算符是某个类的成员，则它们也必须是 `istream` 或 `ostream` 的成员。然而，这两个类属于标准库，并且我们无法给标准库中的类添加任何成员。

因此，如果我们希望为类自定义 IO 运算符，则必须将其定义成非成员函数。当然，IO 运算符通常需要读写类的非公有数据成员，所以 IO 运算符一般被声明为友元（参见 7.2.1 节，第 241 页）。 558

14.2.1 节练习

练习 14.6：为你的 Sales_data 类定义输出运算符。

练习 14.7：你在 13.5 节的练习（第 470 页）中曾经编写了一个 String 类，为它定义一个输出运算符。

练习 14.8：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，为它定义一个输出运算符。

14.2.2 重载输入运算符>>



通常情况下，输入运算符的第一个形参是运算符将要读取的流的引用，第二个形参是将要读入到的（非常量）对象的引用。该运算符通常会返回某个给定流的引用。第二个形参之所以必须是个非常量是因为输入运算符本身的目的就是将数据读入到这个对象中。

Sales_data 的输入运算符

举个例子，我们将按照如下形式编写 Sales_data 的输入运算符：

```
istream &operator>>(istream &is, Sales_data &item)
{
    double price; // 不需要初始化，因为我们将先读入数据到 price，之后才使用它
    is >> item.bookNo >> item.units_sold >> price;
    if (is) // 检查输入是否成功
        item.revenue = item.units_sold * price;
    else
        item = Sales_data(); // 输入失败：对象被赋予默认的状态
    return is;
}
```

除了 if 语句之外，这个定义与之前的 read 函数（参见 7.1.3 节，第 234 页）完全一样。if 语句检查读取操作是否成功，如果发生了 IO 错误，则运算符将给定的对象重置为空 Sales_data，这样可以确保对象处于正确的状态。



输入运算符必须处理输入可能失败的情况，而输出运算符不需要。

输入时的错误

559

在执行输入运算符时可能发生下列错误：

- 当流含有错误类型的数据时读取操作可能失败。例如在读取完 bookNo 后，输入运算符假定接下来读入的是两个数字数据，一旦输入的不是数字数据，则读取操作及后续对流的其他使用都将失败。
- 当读取操作到达文件末尾或者遇到输入流的其他错误时也会失败。

在程序中我们没有逐个检查每个读取操作，而是等读取了所有数据后赶在使用这些数据前一次性检查：

```

if (is)                                // 检查输入是否成功
    item.revenue = item.units_sold * price;
else
    item = Sales_data();      // 输入失败：对象被赋予默认的状态

```

如果读取操作失败，则 `price` 的值将是未定义的。因此，在使用 `price` 前我们需要首先检查输入流的合法性，然后才能执行计算并将结果存入 `revenue`。如果发生了错误，我们无须在意到底是哪部分输入失败，只要将一个新的默认初始化的 `Sales_data` 对象赋予 `item` 从而将其重置为空 `Sales_data` 就可以了。执行这样的赋值后，`item` 的 `bookNo` 成员将是一个空 `string`，`revenue` 和 `units_sold` 成员将等于 0。

如果在发生错误前对象已经有一部分被改变，则适时地将对象置为合法状态显得异常重要。例如在这个输入运算符中，我们可能在成功读取新的 `bookNo` 后遇到错误，这意味着对象的 `units_sold` 和 `revenue` 成员并没有改变，因此有可能会将这两个数据与一条完全不匹配的 `bookNo` 组合在一起。

通过将对象置为合法的状态，我们能（略微）保护使用者免于受到输入错误的影响。此时的对象处于可用状态，即它的成员都是被正确定义的。而且该对象也不会产生误导性的结果，因为它的数据在本质上确实是一体的。



当读取操作发生错误时，输入运算符应该负责从错误中恢复。

标示错误

一些输入运算符需要做更多数据验证的工作。例如，我们的输入运算符可能需要检查 `bookNo` 是否符合规范的格式。在这样的例子中，即使从技术上来看 IO 是成功的，输入运算符也应该设置流的条件状态以标示出失败信息（参见 8.1.2 节，第 279 页）。通常情况下，输入运算符只设置 `failbit`。除此之外，设置 `eofbit` 表示文件耗尽，而设置 `badbit` 表示流被破坏。最好的方式是由 IO 标准库自己来标示这些错误。

560 >

14.2.2 节练习

练习 14.9: 为你的 `Sales_data` 类定义输入运算符。

练习 14.10: 对于 `Sales_data` 的输入运算符来说如果给定了下面的输入将发生什么情况？

- (a) 0-201-99999-9 10 24.95 (b) 10 24.95 0-210-99999-9

练习 14.11: 下面的 `Sales_data` 输入运算符存在错误吗？如果有，请指出来。对于这个输入运算符如果仍然给定上个练习的输入将发生什么情况？

```

istream& operator>>(istream& in, Sales_data& s)
{
    double price;
    in >> s.bookNo >> s.units_sold >> price;
    s.revenue = s.units_sold * price;
    return in;
}

```

练习 14.12: 你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，为它定义一个输入运算符并确保该运算符可以处理输入错误。

14.3 算术和关系运算符

通常情况下，我们把算术和关系运算符定义成非成员函数以允许对左侧或右侧的运算对象进行转换（参见 14.1 节，第 492 页）。因为这些运算符一般不需要改变运算对象的状态，所以形参都是常量的引用。

算术运算符通常会计算它的两个运算对象并得到一个新值，这个值有别于任意一个运算对象，常常位于一个局部变量之内，操作完成后返回该局部变量的副本作为其结果。如果类定义了算术运算符，则它一般也会定义一个对应的复合赋值运算符。此时，最有效的方式是使用复合赋值来定义算术运算符：

```
// 假设两个对象指向同一本书
Sales_data
operator+(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs;           // 把 lhs 的数据成员拷贝给 sum
    sum += rhs;                   // 将 rhs 加到 sum 中
    return sum;
}
```

这个定义与原来的 add 函数（参见 7.1.3 节，第 234 页）是完全等价的。我们把 lhs 拷贝给局部变量 sum，然后使用 Sales_data 的复合赋值运算符（将在第 500 页定义）将 rhs 的值加到 sum 中，最后函数返回 sum 的副本。



如果类同时定义了算术运算符和相关的复合赋值运算符，则通常情况下应该使用复合赋值来实现算术运算符。

561

14.3 节练习

练习 14.13：你认为 Sales_data 类还应该支持哪些其他算术运算符（参见表 4.1，第 124 页）？如果有的话，请给出它们的定义。

练习 14.14：你觉得为什么调用 operator+= 来定义 operator+ 比其他方法更有效？

练习 14.15：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有其他算术运算符吗？如果是，请实现它们；如果不是，解释原因。

14.3.1 相等运算符



通常情况下，C++ 中的类通过定义相等运算符来检验两个对象是否相等。也就是说，它们会比较对象的每一个数据成员，只有当所有对应的成员都相等时才认为两个对象相等。依据这一思想，我们的 Sales_data 类的相等运算符不但应该比较 bookNo，还应该比较具体的销售数据：

```
bool operator==(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn() &&
           lhs.units_sold == rhs.units_sold &&
           lhs.revenue == rhs.revenue;
}
bool operator!=(const Sales_data &lhs, const Sales_data &rhs)
```

```

{
    return !(lhs == rhs);
}

```

就上面这些函数的定义本身而言，它们似乎比较简单，也没什么价值，对于我们来说重要的是从这些函数中体现出来的设计准则：

- 如果一个类含有判断两个对象是否相等的操作，则它显然应该把函数定义成 `operator==` 而非一个普通的命名函数：因为用户肯定希望能使用 `==` 比较对象，所以提供了 `==` 就意味着用户无须再费时费力地学习并记忆一个全新的函数名字。此外，类定义了 `==` 运算符之后也更容易使用标准库容器和算法。
- 如果类定义了 `operator==`，则该运算符应该能判断一组给定的对象中是否含有重复数据。
- 通常情况下，相等运算符应该具有传递性，换句话说，如果 `a==b` 和 `b==c` 都为真，则 `a==c` 也应该为真。
- 如果类定义了 `operator==`，则这个类也应该定义 `operator!=`。对于用户来说，当他们能使用 `==` 时肯定也希望使用 `!=`，反之亦然。
- 相等运算符和不相等运算符中的一个应该把工作委托给另外一个，这意味着其中一个运算符应该负责实际比较对象的工作，而另一个运算符则只是调用那个真正工作的运算符。



如果某个类在逻辑上有相等性的含义，则该类应该定义 `operator==`，这样做可以使得用户更容易使用标准库算法来处理这个类。

14.3.1 节练习

练习 14.16：为你的 `StrBlob` 类（参见 12.1.1 节，第 405 页）、`StrBlobPtr` 类（参见 12.1.6 节，第 421 页）、`StrVec` 类（参见 13.5 节，第 465 页）和 `String` 类（参见 13.5 节，第 470 页）分别定义相等运算符和不相等运算符。

练习 14.17：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有相等运算符吗？如果是，请实现它；如果不是，解释原因。



14.3.2 关系运算符

定义了相等运算符的类也常常（但不总是）包含关系运算符。特别是，因为关联容器和一些算法要用到小于运算符，所以定义 `operator<` 会比较有用。

通常情况下关系运算符应该

1. 定义顺序关系，令其与关联容器中对关键字的要求一致（参见 11.2.2 节，第 378 页）；并且
2. 如果类同时也含有 `==` 运算符的话，则定义一种关系令其与 `==` 保持一致。特别是，如果两个对象是 `!=` 的，那么一个对象应该 `<` 另外一个。



尽管我们可能会认为 `Sales_data` 类应该支持关系运算符，但事实证明并非如此，其中的缘由比较微妙，值得读者深思。

一开始我们可能会认为应该像 `compareIsbn`（参见 11.2.2 节，第 379 页）那样定义 `<`，该函数通过比较 `ISBN` 来实现对两个对象的比较。然而，尽管 `compareIsbn` 提供的

顺序关系符合要求 1，但是函数得到的结果显然与我们定义的`==`不一致，因此它不满足要求 2。

对于 `Sales_data` 的`==`运算符来说，如果两笔交易的 `revenue` 和 `units_sold` 成员不同，那么即使它们的 `ISBN` 相同也无济于事，它们仍然是不相等的。如果我们定义的`<`运算符仅仅比较 `ISBN` 成员，那么将发生这样的情况：两个 `ISBN` 相同但 `revenue` 和 `units_sold` 不同的对象经比较是不相等的，但是其中的任何一个都不比另一个小。然而实际情况是，如果我们有两个对象并且哪个都不比另一个小，则从道理上来讲这两个对象应该是相等的。563

基于上述分析我们也许会认为，只要让 `operator<` 依次比较每个数据元素就能解决问题了，比方说让 `operator<` 先比较 `isbn`，相等的话继续比较 `units_sold`，还相等再继续比较 `revenue`。

然而，这样的排序没有任何必要。根据将来使用 `Sales_data` 类的实际需要，我们可能会希望先比较 `units_sold`，也可能希望先比较 `revenue`。有的时候，我们希望 `units_sold` 少的对象“小于”`units_sold` 多的对象；另一些时候，则可能希望 `revenue` 少的对象“小于”`revenue` 多的对象。

因此对于 `Sales_data` 类来说，不存在一种逻辑可靠的`<` 定义，这个类不定义`<` 运算符也许更好。



如果存在唯一一种逻辑可靠的`<` 定义，则应该考虑为这个类定义`<` 运算符。如果类同时还包含`==`，则当且仅当`<` 的定义和`==` 产生的结果一致时才定义`<` 运算符。

14.3.2 节练习

练习 14.18：为你的 `StrBlob` 类、`StrBlobPtr` 类、`StrVec` 类和 `String` 类定义关系运算符。

练习 14.19：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有关系运算符吗？如果是，请实现它；如果不是，解释原因。

14.4 赋值运算符

之前已经介绍过拷贝赋值和移动赋值运算符（参见 13.1.2 节，第 443 页和 13.6.2 节，第 474 页），它们可以把类的一个对象赋值给该类的另一个对象。此外，类还可以定义其他赋值运算符以使用别的类型作为右侧运算对象。

举个例子，在拷贝赋值和移动赋值运算符之外，标准库 `vector` 类还定义了第三种赋值运算符，该运算符接受花括号内的元素列表作为参数（参见 9.2.5 节，第 302 页）。我们能以如下的形式使用该运算符：

```
vector<string> v;
v = {"a", "an", "the"};
```

同样，也可以把这个运算符添加到 `StrVec` 类中（参见 13.5 节，第 465 页）：

```
class StrVec {
public:
    StrVec &operator=(std::initializer_list<std::string>);
    // 其他成员与 13.5 节（第 465 页）一致
};
```

564> 为了与内置类型的赋值运算符保持一致（也与我们已经定义的拷贝赋值和移动赋值运算一致），这个新的赋值运算符将返回其左侧运算对象的引用：

```
StrVec &StrVec::operator=(initializer_list<string> il)
{
    // alloc_n_copy 分配内存空间并从给定范围内拷贝元素
    auto data = alloc_n_copy(il.begin(), il.end());
    free();           // 销毁对象中的元素并释放内存空间
    elements = data.first; // 更新数据成员使其指向新空间
    first_free = cap = data.second;
    return *this;
}
```

和拷贝赋值及移动赋值运算符一样，其他重载的赋值运算符也必须先释放当前内存空间，再创建一片新空间。不同之处是，这个运算符无须检查对象向自身的赋值，这是因为它的形参 `initializer_list<string>`（参见 6.2.6 节，第 198 页）确保 `il` 与 `this` 所指的不是同一个对象。



我们可以重载赋值运算符。不论形参的类型是什么，赋值运算符都必须定义为成员函数。

复合赋值运算符

复合赋值运算符非得是类的成员，不过我们还是倾向于把包括复合赋值在内的所有赋值运算都定义在类的内部。为了与内置类型的复合赋值保持一致，类中的复合赋值运算符也要返回其左侧运算对象的引用。例如，下面是 `Sales_data` 类中复合赋值运算符的定义：

```
// 作为成员的二元运算符：左侧运算对象绑定到隐式的 this 指针
// 假定两个对象表示的是同一本书
Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```



赋值运算符必须定义成类的成员，复合赋值运算符通常情况下也应该这样做。这两类运算符都应该返回左侧运算对象的引用。

14.4 节练习

练习 14.20: 为你的 `Sales_data` 类定义加法和复合赋值运算符。

练习 14.21: 编写 `Sales_data` 类的`+和+=`运算符，使得`+`执行实际的加法操作而`+=`调用`+`。相比于 14.3 节（第 497 页）和 14.4 节（第 500 页）对这两个运算符的定义，本题的定义有何缺点？试讨论之。

练习 14.22: 定义赋值运算符的一个新版本，使得我们能把一个表示 ISBN 的 `string` 赋给一个 `Sales_data` 对象。

练习 14.23: 为你的 `StrVec` 类定义一个 `initializer_list` 赋值运算符。

练习 14.24: 你在 7.5.1 节的练习 7.40 (第 261 页) 中曾经选择并编写了一个类, 你认为它应该含有拷贝赋值和移动赋值运算符吗? 如果是, 请实现它们。

练习 14.25: 上题的这个类还需要定义其他赋值运算符吗? 如果是, 请实现它们; 同时说明运算对象应该是什么类型并解释原因。

14.5 下标运算符

表示容器的类通常可以通过元素在容器中的位置访问元素, 这些类一般会定义下标运算符 `operator[]`。



下标运算符必须是成员函数。

565

为了与下标的原始定义兼容, 下标运算符通常以所访问元素的引用作为返回值, 这样做的好处是下标可以出现在赋值运算符的任意一端。进一步, 我们最好同时定义下标运算符的常量版本和非常量版本, 当作用于一个常量对象时, 下标运算符返回常量引用以确保我们不会给返回的对象赋值。



如果一个类包含下标运算符, 则它通常会定义两个版本: 一个返回普通引用, 另一个是类的常量成员并且返回常量引用。

举个例子, 我们按照如下形式定义 `StrVec` (参见 13.5 节, 第 465 页) 的下标运算符:

```
class StrVec {
public:
    std::string& operator[](std::size_t n)
    { return elements[n]; }
    const std::string& operator[](std::size_t n) const
    { return elements[n]; }
    // 其他成员与 13.5 (第 465 页) 一致
private:
    std::string *elements;           // 指向数组首元素的指针
};
```

上面这两个下标运算符的用法类似于 `vector` 或者数组中的下标。因为下标运算符返回的是元素的引用, 所以当 `StrVec` 是非常量时, 我们可以给元素赋值; 而当我们对常量对象取下标时, 不能为其赋值:

```
// 假设 svec 是一个 StrVec 对象
const StrVec cvec = svec;           // 把 svec 的元素拷贝到 cvec 中
// 如果 svec 中含有元素, 对第一个元素运行 string 的 empty 函数
if (svec.size() && svec[0].empty()) {
    svec[0] = "zero";               // 正确: 下标运算符返回 string 的引用
    cvec[0] = "Zip";                // 错误: 对 cvec 取下标返回的是常量引用
}
```

566

14.5 节练习

练习 14.26: 为你的 StrBlob 类、StrBlobPtr 类、StrVec 类和 String 类定义下标运算符。

14.6 递增和递减运算符

在迭代器类中通常会实现递增运算符（`++`）和递减运算符（`--`），这两种运算符使得类可以在元素的序列中前后移动。C++语言并不要求递增和递减运算符必须是类的成员，但是因为它们改变的正好是所操作对象的状态，所以建议将其设定为成员函数。

对于内置类型来说，递增和递减运算符既有前置版本也有后置版本。同样，我们也应该为类定义两个版本的递增和递减运算符。接下来我们首先介绍前置版本，然后实现后置版本。



定义递增和递减运算符的类应该同时定义前置版本和后置版本。这些运算符通常应该被定义成类的成员。

定义前置递增/递减运算符

为了说明递增和递减运算符，我们不妨在 StrBlobPtr 类（参见 12.1.6 节，第 421 页）中定义它们：

```
class StrBlobPtr {
public:
    // 递增和递减运算符
    StrBlobPtr& operator++(); // 前置运算符
    StrBlobPtr& operator--();
    // 其他成员和之前的版本一致
};
```



为了与内置版本保持一致，前置运算符应该返回递增或递减后对象的引用。

567

递增和递减运算符的工作机理非常相似：它们首先调用 `check` 函数检验 `StrBlobPtr` 是否有效，如果是，接着检查给定的索引值是否有效。如果 `check` 函数没有抛出异常，则运算符返回对象的引用。

在递增运算符的例子中，我们把 `curr` 的当前值传递给 `check` 函数。如果这个值小于 `vector` 的大小，则 `check` 正常返回；否则，如果 `curr` 已经到达了 `vector` 的末尾，`check` 将抛出异常：

```
// 前置版本：返回递增/递减对象的引用
StrBlobPtr& StrBlobPtr::operator++()
{
    // 如果 curr 已经指向了容器的尾后位置，则无法递增它
    check(curr, "increment past end of StrBlobPtr");
    ++curr; // 将 curr 在当前状态下向前移动一个元素
    return *this;
}
```

```

StrBlobPtr& StrBlobPtr::operator--()
{
    // 如果 curr 是 0，则继续递减它将产生一个无效下标
    --curr;                                // 将 curr 在当前状态下向后移动一个元素
    check(curr, "decrement past begin of StrBlobPtr");
    return *this;
}

```

递减运算符先递减 curr，然后调用 check 函数。此时，如果 curr（一个无符号数）已经是 0 了，那么我们传递给 check 的值将是一个表示无效下标的非常大的正数值（参见 2.1.2 节，第 33 页）。

区分前置和后置运算符

要想同时定义前置和后置运算符，必须首先解决一个问题，即普通的重载形式无法区分这两种情况。前置和后置版本使用的是同一个符号，意味着其重载版本所用的名字将是相同的，并且运算对象的数量和类型也相同。

为了解决这个问题，后置版本接受一个额外的（不被使用）int 类型的形参。当我们使用后置运算符时，编译器为这个形参提供一个值为 0 的实参。尽管从语法上来说后置函数可以使用这个额外的形参，但是在实际过程中通常不会这么做。这个形参的唯一作用就是区分前置版本和后置版本的函数，而不是真的要在实现后置版本时参与运算。

接下来我们为 StrBlobPtr 添加后置运算符：

```

class StrBlobPtr {
public:
    // 递增和递减运算符
    StrBlobPtr operator++(int);           // 后置运算符
    StrBlobPtr operator--(int);
    // 其他成员和之前的版本一致
};

```



为了与内置版本保持一致，后置运算符应该返回对象的原值（递增或递减之前 的值），返回的形式是一个值而非引用。

568

对于后置版本来说，在递增对象之前需要首先记录对象的状态：

```

// 后置版本：递增/递减对象的值但是返回原值
StrBlobPtr StrBlobPtr::operator++(int)
{
    // 此处无须检查有效性，调用前置递增运算时才需要检查
    StrBlobPtr ret = *this; // 记录当前的值
    ++*this;              // 向前移动一个元素，前置++需要检查递增的有效性
    return ret;            // 返回之前记录的状态
}
StrBlobPtr StrBlobPtr::operator--(int)
{
    // 此处无须检查有效性，调用前置递减运算时才需要检查
    StrBlobPtr ret = *this; // 记录当前的值
    --*this;              // 向后移动一个元素，前置--需要检查递减的有效性
    return ret;            // 返回之前记录的状态
}

```

由上可知，我们的后置运算符调用各自的前置版本来完成实际的工作。例如后置递增运算符执行

```
++*this
```

该表达式调用前置递增运算符，前置递增运算符首先检查递增操作是否安全，根据检查的结果抛出一个异常或者执行递增 curr 的操作。假定通过了检查，则后置函数返回事先存好的 ret 的副本。因此最终的效果是，对象本身向前移动了一个元素，而返回的结果仍然反映对象在未递增之前原始的值。



因为我们不会用到 int 形参，所以无须为其命名。

显式地调用后置运算符

如在第 491 页介绍的，可以显式地调用一个重载的运算符，其效果与在表达式中以运算符号的形式使用它完全一样。如果我们想通过函数调用的方式调用后置版本，则必须为它的整型参数传递一个值：

```
StrBlobPtr p(a1);           // p 指向 a1 中的 vector
p.operator++(0);            // 调用后置版本的 operator++
p.operator++();             // 调用前置版本的 operator++
```

尽管传入的值通常会被运算符函数忽略，但却必不可少，因为编译器只有通过它才能知道应该使用后置版本。

569

14.6 节练习

练习 14.27：为你的 StrBlobPtr 类添加递增和递减运算符。

练习 14.28：为你的 StrBlobPtr 类添加加法和减法运算符，使其可以实现指针的算术运算（参见 3.5.3 节，第 106 页）。

练习 14.29：为什么不定义 const 版本的递增和递减运算符？

14.7 成员访问运算符

在迭代器类及智能指针类（参见 12.1 节，第 400 页）中常常用到解引用运算符 (*) 和箭头运算符 (->)。我们以如下形式向 StrBlobPtr 类添加这两种运算符：

```
class StrBlobPtr {
public:
    std::string& operator*() const
    { auto p = check(curr, "dereference past end");
      return (*p)[curr];           // (*p) 是对象所指的 vector
    }
    std::string* operator->() const
    { // 将实际工作委托给解引用运算符
      return & this->operator*();
    }
    // 其他成员与之前的版本一致
}
```

解引用运算符首先检查 curr 是否仍在作用范围内，如果是，则返回 curr 所指元素的一个引用。箭头运算符不执行任何自己的操作，而是调用解引用运算符并返回解引用结果元素的地址。



箭头运算符必须是类的成员。解引用运算符通常也是类的成员，尽管并非必须如此。

值得注意的是，我们将这两个运算符定义成了 const 成员，这是因为与递增和递减运算符不一样，获取一个元素并不会改变 StrBlobPtr 对象的状态。同时，它们的返回值分别是非常量 string 的引用或指针，因为一个 StrBlobPtr 只能绑定到非常量的 StrBlob 对象（参见 12.1.6 节，第 421 页）。

这两个运算符的用法与指针或者 vector 迭代器的对应操作完全一致：

```
StrBlob a1 = {"hi", "bye", "now"};
StrBlobPtr p(a1);                                // p 指向 a1 中的 vector
*p = "okay";                                     // 给 a1 的首元素赋值
cout << p->size() << endl;                      // 打印 4，这是 a1 首元素的大小
cout << (*p).size() << endl;                     // 等价于 p->size()
```

对箭头运算符返回值的限定

570

和大多数其他运算符一样（尽管这么做不太好），我们能令 operator* 完成任何我们指定的操作。换句话说，我们可以让 operator* 返回一个固定值 42，或者打印对象的内容，或者其他。箭头运算符则不是这样，它永远不能丢掉成员访问这个最基本的含义。当我们重载箭头时，可以改变的是箭头从哪个对象当中获取成员，而箭头获取成员这一事实则永远不变。

对于形如 point->mem 的表达式来说，point 必须是指向类对象的指针或者是一个重载了 operator-> 的类的对象。根据 point 类型的不同，point->mem 分别等价于

```
(*point).mem;                                // point 是一个内置的指针类型
point.operator()->mem;                         // point 是类的一个对象
```

除此之外，代码都将发生错误。point->mem 的执行过程如下所示：

1. 如果 point 是指针，则我们应用内置的箭头运算符，表达式等价于 (*point).mem。首先解引用该指针，然后从所得的对象中获取指定的成员。如果 point 所指的类型没有名为 mem 的成员，程序会发生错误。
2. 如果 point 是定义了 operator-> 的类的一个对象，则我们使用 point.operator->() 的结果来获取 mem。其中，如果该结果是一个指针，则执行第 1 步；如果该结果本身含有重载的 operator->()，则重复调用当前步骤。最终，当这一过程结束时程序或者返回了所需的内容，或者返回一些表示程序错误的信息。



重载的箭头运算符必须返回类的指针或者自定义了箭头运算符的某个类的对象。

14.7 节练习

练习 14.30: 为你的 StrBlobPtr 类和在 12.1.6 节练习 12.22（第 423 页）中定义的 ConstStrBlobPtr 类分别添加解引用运算符和箭头运算符。注意：因为 ConstStrBlobPtr 的数据成员指向 const vector，所以 ConstStrBlobPtr 中的运算符必须返回常量引用。

练习 14.31: 我们的 StrBlobPtr 类没有定义拷贝构造函数、赋值运算符及析构函数，为什么？

练习 14.32: 定义一个类令其含有指向 StrBlobPtr 对象的指针，为这个类定义重载的箭头运算符。



14.8 函数调用运算符

571

如果类重载了函数调用运算符，则我们可以像使用函数一样使用该类的对象。因为这样的类同时也能存储状态，所以与普通函数相比它们更加灵活。

举个简单的例子，下面这个名为 absInt 的 struct 含有一个调用运算符，该运算符负责返回其参数的绝对值：

```
struct absInt {
    int operator()(int val) const {
        return val < 0 ? -val : val;
    }
};
```

这个类只定义了一种操作：函数调用运算符，它负责接受一个 int 类型的实参，然后返回该实参的绝对值。

我们使用调用运算符的方式是令一个 absInt 对象作用于一个实参列表，这一过程看起来非常像调用函数的过程：

```
int i = -42;
absInt absObj;           // 含有函数调用运算符的对象
int ui = absObj(i);      // 将 i 传递给 absObj.operator()
```

即使 absObj 只是一个对象而非函数，我们也能“调用”该对象。调用对象实际上是在运行重载的调用运算符。在此例中，该运算符接受一个 int 值并返回其绝对值。



函数调用运算符必须是成员函数。一个类可以定义多个不同版本的调用运算符，相互之间应该在参数数量或类型上有所区别。

如果类定义了调用运算符，则该类的对象称作 **函数对象**（function object）。因为可以调用这种对象，所以我们说这些对象的“行为像函数一样”。

含有状态的函数对象类

和其他类一样，函数对象类除了 operator() 之外也可以包含其他成员。函数对象类通常含有一些数据成员，这些成员被用于定制调用运算符中的操作。

举个例子，我们将定义一个打印 string 实参内容的类。默认情况下，我们的类会将

内容写入到 cout 中，每个 string 之间以空格隔开。同时也允许类的用户提供其他可写入的流及其他分隔符。我们将该类定义如下：

```
class PrintString {
public:
    PrintString(ostream &o = cout, char c = ' '):
        os(o), sep(c) {}
    void operator()(const string &s) const { os << s << sep; }
private:
    ostream &os;           // 用于写入的目的流
    char sep;              // 用于将不同输出隔开的字符
};
```

我们的类有一个构造函数，它接受一个输出流的引用以及一个用于分隔的字符，这两个形参的默认实参（参见 6.5.1 节，第 211 页）分别是 cout 和空格。572之后的函数调用运算符使用这些成员协助其打印给定的 string。

当定义 PrintString 的对象时，对于分隔符及输出流既可以使用默认值也可以提供我们自己的值：

```
PrintString printer;          // 使用默认值，打印到 cout
printer(s);                  // 在 cout 中打印 s，后面跟一个空格
PrintString errors(cerr, '\n');
errors(s);                   // 在 cerr 中打印 s，后面跟一个换行符
```

函数对象常常作为泛型算法的实参。例如，可以使用标准库 for_each 算法（参见 10.3.2 节，第 348 页）和我们自己的 PrintString 类来打印容器的内容：

```
for_each(vs.begin(), vs.end(), PrintString(cerr, '\n'));
```

for_each 的第三个实参是类型 PrintString 的一个临时对象，其中我们用 cerr 和换行符初始化了该对象。当程序调用 for_each 时，将会把 vs 中的每个元素依次打印到 cerr 中，元素之间以换行符分隔。

14.8 节练习

练习 14.33: 一个重载的函数调用运算符应该接受几个运算对象？

练习 14.34: 定义一个函数对象类，令其执行 if-then-else 的操作：该类的调用运算符接受三个形参，它首先检查第一个形参，如果成功返回第二个形参的值；如果不成功返回第三个形参的值。

练习 14.35: 编写一个类似于 PrintString 的类，令其从 istream 中读取一行输入，然后返回一个表示我们所读内容的 string。如果读取失败，返回空 string。

练习 14.36: 使用前一个练习定义的类读取标准输入，将每一行保存为 vector 的一个元素。

练习 14.37: 编写一个类令其检查两个值是否相等。使用该对象及标准库算法编写程序，令其替换某个序列中具有给定值的所有实例。

14.8.1 lambda 是函数对象

在前一节中，我们使用一个 PrintString 对象作为调用 for_each 的实参，这一

用法类似于我们在 10.3.2 节（第 346 页）中编写的使用 lambda 表达式的程序。当我们编写了一个 lambda 后，编译器将该表达式翻译成一个未命名类的未命名对象（参见 10.3.3 节，第 349 页）。在 lambda 表达式产生的类中含有一个重载的函数调用运算符，例如，对于我们传递给 stable_sort 作为其最后一个实参的 lambda 表达式来说：

```
// 根据单词的长度对其进行排序，对于长度相同的单词按照字母表顺序排序
stable_sort(words.begin(), words.end(),
[](const string &a, const string &b)
{ return a.size() < b.size();});
```

其行为类似于下面这个类的一个未命名对象

```
class ShorterString {
public:
    bool operator()(const string &s1, const string &s2) const
    { return s1.size() < s2.size(); }
};
```

产生的类只有一个函数调用运算符成员，它负责接受两个 string 并比较它们的长度，它的形参列表和函数体与 lambda 表达式完全一样。如我们在 10.3.3 节（第 352 页）所见，默认情况下 lambda 不能改变它捕获的变量。因此在默认情况下，由 lambda 产生的类当中的函数调用运算符是一个 const 成员函数。如果 lambda 被声明为可变的，则调用运算符就不是 const 的了。

用这个类替代 lambda 表达式后，我们可以重写并重新调用 stable_sort：

```
stable_sort(words.begin(), words.end(), ShorterString());
```

第三个实参是新构建的 ShorterString 对象，当 stable_sort 内部的代码每次比较两个 string 时就会“调用”这一对象，此时该对象将调用运算符的函数体，判断第一个 string 的大小小于第二个时返回 true。

表示 lambda 及相应捕获行为的类

如我们所知，当一个 lambda 表达式通过引用捕获变量时，将由程序负责确保 lambda 执行时引用的对象确实存在（参见 10.3.3 节，第 350 页）。因此，编译器可以直接使用该引用而无须在 lambda 产生的类中将其存储为数据成员。

相反，通过值捕获的变量被拷贝到 lambda 中（参见 10.3.3 节，第 350 页）。因此，这种 lambda 产生的类必须为每个值捕获的变量建立对应的数据成员，同时创建构造函数，令其使用捕获的变量的值来初始化数据成员。举个例子，在 10.3.2 节（第 347 页）中有一个 lambda，它的作用是找到第一个长度不小于给定值的 string 对象：

```
// 获得第一个指向满足条件元素的迭代器，该元素满足 size() is >= sz
auto wc = find_if(words.begin(), words.end(),
[sz](const string &a)
{ return a.size() >= sz;});
```

该 lambda 表达式产生的类将形如：

```
574> class SizeComp {
    SizeComp(size_t n): sz(n) {} // 该形参对应捕获的变量
    // 该调用运算符的返回类型、形参和函数体都与 lambda 一致
    bool operator()(const string &s) const
    { return s.size() >= sz; }
```

```

private:
    size_t sz; // 该数据成员对应通过值捕获的变量
};

```

和我们的 `ShorterString` 类不同，上面这个类含有一个数据成员以及一个用于初始化该成员的构造函数。这个合成的类不含有默认构造函数，因此要想使用这个类必须提供一个实参：

```

// 获得第一个指向满足条件元素的迭代器，该元素满足 size() is >= sz
auto wc = find_if(words.begin(), words.end(), SizeComp(sz));

```

`lambda` 表达式产生的类不含默认构造函数、赋值运算符及默认析构函数；它是否含有默认的拷贝/移动构造函数则通常要视捕获的数据成员类型而定（参见 13.1.6 节，第 450 页和 13.6.2 节，第 475 页）。

14.8.1 节练习

练习 14.38：编写一个类令其检查某个给定的 `string` 对象的长度是否与一个阈值相等。使用该对象编写程序，统计并报告在输入的文件中长度为 1 的单词有多少个、长度为 2 的单词又有多少个、……、长度为 10 的单词又有多少个。

练习 14.39：修改上一题的程序令其报告长度在 1 至 9 之间的单词有多少个、长度在 10 以上的单词又有多少个。

练习 14.40：重新编写 10.3.2 节（第 349 页）的 `biggies` 函数，使用函数对象类替换其中的 `lambda` 表达式。

练习 14.41：你认为 C++11 新标准为什么要增加 `lambda`？对于你自己来说，什么情况下会使用 `lambda`，什么情况下会使用类？

14.8.2 标准库定义的函数对象

标准库定义了一组表示算术运算符、关系运算符和逻辑运算符的类，每个类分别定义了一个执行命名操作的调用运算符。例如，`plus` 类定义了一个函数调用运算符用于对一对运算对象执行 + 的操作；`modulus` 类定义了一个调用运算符执行二元的 % 操作；`equal_to` 类执行 ==，等等。

这些类都被定义成模板的形式，我们可以为其指定具体的应用类型，这里的类型即调用运算符的形参类型。例如，`plus<string>` 令 `string` 加法运算符作用于 `string` 对象；`plus<int>` 的运算对象是 `int`；`plus<Sales_data>` 对 `Sales_data` 对象执行加法运算，以此类推：

```

plus<int> intAdd; // 可执行 int 加法的函数对象
negate<int> intNegate; // 可对 int 值取反的函数对象
// 使用 intAdd::operator(int, int) 求 10 和 20 的和
int sum = intAdd(10, 20); // 等价于 sum = 30
sum = intNegate(intAdd(10, 20)); // 等价于 sum = 30
// 使用 intNegate::operator(int) 生成 -10
// 然后将 -10 作为 intAdd::operator(int, int) 的第二个参数
sum = intAdd(10, intNegate(10)); // sum = 0

```

< 575

表 14.2 所列的类型定义在 `functional` 头文件中。

表 14.2: 标准库函数对象

算术	关系	逻辑
<code>plus<Type></code>	<code>equal_to<Type></code>	<code>logical_and<Type></code>
<code>minus<Type></code>	<code>not_equal_to<Type></code>	<code>logical_or<Type></code>
<code>multiplies<Type></code>	<code>greater<Type></code>	<code>logical_not<Type></code>
<code>divides<Type></code>	<code>greater_equal<Type></code>	
<code>modulus<Type></code>	<code>less<Type></code>	
<code>negate<Type></code>	<code>less_equal<Type></code>	

在算法中使用标准库函数对象

表示运算符的函数对象类常用来替换算法中的默认运算符。如我们所知，在默认情况下排序算法使用 `operator<` 将序列按照升序排列。如果要执行降序排列的话，我们可以传入一个 `greater` 类型的对象。该类将产生一个调用运算符并负责执行待排序类型的大于运算。例如，如果 `svec` 是一个 `vector<string>`，

```
// 传入一个临时的函数对象用于执行两个 string 对象的>比较运算
sort(svec.begin(), svec.end(), greater<string>());
```

则上面的语句将按照降序对 `svec` 进行排序。第三个实参是 `greater<string>` 类型的一个未命名的对象，因此当 `sort` 比较元素时，不再是使用默认的`<`运算符，而是调用给定的 `greater` 函数对象。该对象负责在 `string` 元素之间执行`>`比较运算。

需要特别注意的是，标准库规定其函数对象对于指针同样适用。我们之前曾经介绍过比较两个无关指针将产生未定义的行为（参见 3.5.3 节，第 107 页），然而我们可能会希望通过比较指针的内存地址来 `sort` 指针的 `vector`。直接这么做将产生未定义的行为，因此我们可以使用一个标准库函数对象来实现该目的：

```
vector<string *> nameTable; // 指针的 vector
// 错误：nameTable 中的指针彼此之间没有关系，所以<将产生未定义的行为
sort(nameTable.begin(), nameTable.end(),
    [](string *a, string *b) { return a < b; });
// 正确：标准库规定指针的 less 是定义良好的
sort(nameTable.begin(), nameTable.end(), less<string*>());
```

576 关联容器使用 `less<key_type>` 对元素排序，因此我们可以定义一个指针的 `set` 或者在 `map` 中使用指针作为关键值而无须直接声明 `less`。

14.8.2 节练习

练习 14.42：使用标准库函数对象及适配器定义一条表达式，令其

- (a) 统计大于 1024 的值有多少个。
- (b) 找到第一个不等于 pooh 的字符串。
- (c) 将所有的值乘以 2。

练习 14.43：使用标准库函数对象判断一个给定的 `int` 值是否能被 `int` 容器中的所有元素整除。

14.8.3 可调用对象与 function

C++语言中有几种可调用的对象：函数、函数指针、lambda 表达式（参见 10.3.2 节，第 346 页）、bind 创建的对象（参见 10.3.4 节，第 354 页）以及重载了函数调用运算符的类。

和其他对象一样，可调用的对象也有类型。例如，每个 lambda 有它自己唯一的（未命名）类类型；函数及函数指针的类型则由其返回值类型和实参类型决定，等等。

然而，两个不同类型的可调用对象却可能共享同一种调用形式（call signature）。调用形式指明了调用返回的类型以及传递给调用的实参类型。一种调用形式对应一个函数类型，例如：

```
int(int, int)
```

是一个函数类型，它接受两个 int、返回一个 int。

不同类型可能具有相同的调用形式

对于几个可调用对象共享同一种调用形式的情况，有时我们会希望把它们看成具有相同的类型。例如，考虑下列不同类型的可调用对象：

```
// 普通函数
int add(int i, int j) { return i + j; }
// lambda，其产生一个未命名的函数对象类
auto mod = [] (int i, int j) { return i % j; };
// 函数对象类
struct divide {
    int operator()(int denominator, int divisor) {
        return denominator / divisor;
    }
};
```

上面这些可调用对象分别对其参数执行了不同的算术运算，尽管它们的类型各不相同，但 577 是共享同一种调用形式：

```
int(int, int)
```

我们可能希望使用这些可调用对象构建一个简单的桌面计算器。为了实现这一目的，需要定义一个函数表（function table）用于存储指向这些可调用对象的“指针”。当程序需要执行某个特定的操作时，从表中查找该调用的函数。

在 C++语言中，函数表很容易通过 map 来实现。对于此例来说，我们使用一个表示运算符符号的 string 对象作为关键字；使用实现运算符的函数作为值。当我们需要求给定运算符的值时，先通过运算符索引 map，然后调用找到的那个元素。

假定我们的所有函数都相互独立，并且只处理关于 int 的二元运算，则 map 可以定义成如下的形式：

```
// 构建从运算符到函数指针的映射关系，其中函数接受两个 int、返回一个 int
map<string, int(*)(int,int)> binops;
```

我们可以按照下面的形式将 add 的指针添加到 binops 中：

```
// 正确：add 是一个指向正确类型函数的指针
binops.insert({"+", add}); // {"+", add} 是一个 pair (参见 11.2.3 节，379 页)
```

但是我们不能将 mod 或者 divide 存入 binops：

```
binops.insert({"%", mod});           // 错误: mod 不是一个函数指针
```

问题在于 `mod` 是个 `lambda` 表达式，而每个 `lambda` 有它自己的类类型，该类型与存储在 `binops` 中的值的类型不匹配。

标准库 function 类型

C++ 11

我们可以使用一个名为 `function` 的新的标准库类型解决上述问题，`function` 定义在 `functional` 头文件中，表 14.3 列举出了 `function` 定义的操作。

表 14.3: `function` 的操作

<code>function<T> f;</code>	<code>f</code> 是一个用来存储可调用对象的空 <code>function</code> ，这些可调用对象的调用形式应该与函数类型 <code>T</code> 相同（即 <code>T</code> 是 <code>retType(args)</code> ）
<code>function<T> f(nullptr);</code>	显式地构造一个空 <code>function</code>
<code>function<T> f(obj);</code>	在 <code>f</code> 中存储可调用对象 <code>obj</code> 的副本
<code>f</code>	将 <code>f</code> 作为条件：当 <code>f</code> 含有一个可调用对象时为真；否则为假
<code>f(args)</code>	调用 <code>f</code> 中的对象，参数是 <code>args</code>
定义为 <code>function<T></code> 的成员的类型	
<code>result_type</code>	该 <code>function</code> 类型的可调用对象返回的类型
<code>argument_type</code>	当 <code>T</code> 有一个或两个实参时定义的类型。如果 <code>T</code> 只有一个实参，则 <code>argument_type</code> 是该类型的同义词；如果 <code>T</code> 有两个实参，则 <code>first_argument_type</code> 和 <code>second_argument_type</code> 分别代表两个实参的类型
<code>first_argument_type</code>	
<code>second_argument_type</code>	

`function` 是一个模板，和我们使用过的其他模板一样，当创建一个具体的 `function` 类型时我们必须提供额外的信息。在此例中，所谓额外的信息是指该 `function` 类型能够表示的对象的调用形式。参考其他模板，我们在一对尖括号内指定类型：

```
function<int(int, int)>
```

在这里我们声明了一个 `function` 类型，它可以表示接受两个 `int`、返回一个 `int` 的可调用对象。因此，我们可以用这个新声明的类型表示任意一种桌面计算器用到的类型；

```
function<int(int, int)> f1 = add;           // 函数指针
function<int(int, int)> f2 = divide();       // 函数对象类的对象
function<int(int, int)> f3 = [](int i, int j) // lambda
    { return i * j; };
cout << f1(4,2) << endl;                  // 打印 6
cout << f2(4,2) << endl;                  // 打印 2
cout << f3(4,2) << endl;                  // 打印 8
```

578 使用这个 `function` 类型我们可以重新定义 `map`：

```
// 列举了可调用对象与二元运算符对应关系的表格
// 所有可调用对象都必须接受两个 int、返回一个 int
// 其中的元素可以是函数指针、函数对象或者 lambda
map<string, function<int(int, int)>> binops;
```

我们能把所有可调用对象，包括函数指针、`lambda` 或者函数对象在内，都添加到这个 `map` 中：

```
map<string, function<int(int, int)>> binops = {
    {"+", add},                                // 函数指针
    {"-", std::minus<int>()},                  // 标准库函数对象
    {"/", divide()},                           // 用户定义的函数对象
    {"*", [](int i, int j) { return i * j; }}, // 未命名的 lambda
    {"%", mod} };                            // 命名了的 lambda 对象
```

我们的 map 中包含 5 个元素，尽管其中的可调用对象的类型各不相同，我们仍然能够把所有这些类型都存储在同一个 `function<int (int, int)>` 类型中。

一如往常，当我们索引 map 时将得到关联值的一个引用。如果我们索引 `binops`，将得到 `function` 对象的引用。`function` 类型重载了调用运算符，该运算符接受它自己的实参然后将其传递给存好的可调用对象：

```
binops["+"](10, 5); // 调用 add(10, 5)
binops["-"](10, 5); // 使用 minus<int>对象的调用运算符
binops["/"](10, 5); // 使用 divide 对象的调用运算符
binops["*"](10, 5); // 调用 lambda 函数对象
binops["%"](10, 5); // 调用 lambda 函数对象
```

我们依次调用了 `binops` 中存储的每个操作。在第一个调用中，我们获得的元素存放着一个指向 `add` 函数的指针，因此调用 `binops["+"] (10, 5)` 实际上是使用该指针调用 `add`，并传入 10 和 5。在接下来的调用中，`binops["-"]` 返回一个存放着 `std::minus<int>` 类型对象的 `function`，我们将执行该对象的调用运算符。

重载的函数与 `function`

我们不能（直接）将重载函数的名字存入 `function` 类型的对象中：

```
int add(int i, int j) { return i + j; }
Sales_data add(const Sales_data&, const Sales_data&);
map<string, function<int(int, int)>> binops;
binops.insert( {"+", add}); // 错误：哪个 add?
```

解决上述二义性问题的一条途径是存储函数指针（参见 6.7 节，第 221 页）而非函数的名字：

```
int (*fp)(int, int) = add; // 指针所指的 add 是接受两个 int 的版本
binops.insert( {"+", fp}); // 正确：fp 指向一个正确的 add 版本
```

同样，我们也能使用 `lambda` 来消除二义性：

```
// 正确：使用 lambda 来指定我们希望使用的 add 版本
binops.insert( {"+", [](int a, int b) {return add(a, b);}} );
```

`lambda` 内部的函数调用传入了两个 `int`，因此该调用只能匹配接受两个 `int` 的 `add` 版本，而这也正是执行 `lambda` 时真正调用的函数。



新版本标准库中的 `function` 类与旧版本中的 `unary_function` 和 `binary_function` 没有关系，后两个类已经被更通用的 `bind` 函数替代了（参见 10.3.4 节，第 357 页）。

14.8.3 节练习

练习 14.44：编写一个简单的桌面计算器使其能处理二元运算。

14.9 重载、类型转换与运算符

在 7.5.4 节（第 263 页）中我们看到由一个实参调用的非显式构造函数定义了一种隐式的类型转换，这种构造函数将实参类型的对象转换成类类型。我们同样能定义对于类类型的类型转换，通过定义类型转换运算符可以做到这一点。转换构造函数和类型转换运算符共同定义了类类型转换（class-type conversions），这样的转换有时也被称作用户定义的类型转换（user-defined conversions）。

14.9.1 类型转换运算符

类型转换运算符（conversion operator）是类的一种特殊成员函数，它负责将一个类类型的值转换成其他类型。类型转换函数的一般形式如下所示：

```
operator type() const;
```

其中 *type* 表示某种类型。类型转换运算符可以面向任意类型（除了 `void` 之外）进行定义，只要该类型能作为函数的返回类型（参见 6.1 节，第 184 页）。因此，我们不允许转换成数组或者函数类型，但允许转换成指针（包括数组指针及函数指针）或者引用类型。

类型转换运算符既没有显式的返回类型，也没有形参，而且必须定义成类的成员函数。类型转换运算符通常不应该改变待转换对象的内容，因此，类型转换运算符一般被定义成 `const` 成员。



一个类型转换函数必须是类的成员函数；它不能声明返回类型，形参列表也必须为空。类型转换函数通常应该是 `const`。

定义含有类型转换运算符的类

举个例子，我们定义一个比较简单的类，令其表示 0 到 255 之间的一个整数：

```
class SmallInt {
public:
    SmallInt(int i = 0) : val(i)
    {
        if (i < 0 || i > 255)
            throw std::out_of_range("Bad SmallInt value");
    }
    operator int() const { return val; }
private:
    std::size_t val;
};
```

我们的 `SmallInt` 类既定义了向类类型的转换，也定义了从类类型向其他类型的转换。其中，构造函数将算术类型的值转换成 `SmallInt` 对象，而类型转换运算符将 `SmallInt` 对象转换成 `int`：

```
SmallInt si;
```

```
si = 4;           // 首先将 4 隐式地转换成 SmallInt，然后调用 SmallInt::operator=
si + 3;          // 首先将 si 隐式地转换成 int，然后执行整数的加法
```

尽管编译器一次只能执行一个用户定义的类型转换（参见 4.11.2 节，第 144 页），但是隐式的用户定义类型转换可以置于一个标准（内置）类型转换之前或之后（参见 4.11.1 节，第 141 页），并与其一起使用。因此，我们可以将任何算术类型传递给 SmallInt 的构造函数。类似的，我们也能使用类型转换运算符将一个 SmallInt 对象转换成 int，然后再将所得的 int 转换成任何其他算术类型：

```
// 内置类型转换将 double 实参转换成 int
SmallInt si = 3.14;           // 调用 SmallInt(int) 构造函数
// SmallInt 的类型转换运算符将 si 转换成 int
si + 3.14;                   // 内置类型转换将所得的 int 继续转换成 double
```

因为类型转换运算符是隐式执行的，所以无法给这些函数传递实参，当然也就不能在类型转换运算符的定义中使用任何形参。同时，尽管类型转换函数不负责指定返回类型，但实际上每个类型转换函数都会返回一个对应类型的值：

```
class SmallInt;
operator int(SmallInt&);                                // 错误：不是成员函数
class SmallInt {
public:
    int operator int() const;                            // 错误：指定了返回类型
    operator int(int = 0) const;                          // 错误：参数列表不为空
    operator int*() const { return 42; } // 错误：42 不是一个指针
};
```

提示：避免过度使用类型转换函数

和使用重载运算符的经验一样，明智地使用类型转换运算符也能极大地简化类设计者的工作，同时使得使用类更加容易。然而，如果在类类型和转换类型之间不存在明显的映射关系，则这样的类型转换可能具有误导性。

例如，假设某个类表示 Date，我们也许会为它添加一个从 Date 到 int 的转换。然而，类型转换函数的返回值应该是什么？一种可能的解释是，函数返回一个十进制数，依次表示年、月、日，例如，July 30, 1989 可能转换为 int 值 19890730。同时还存在另外一种合理的解释，即类型转换运算符返回的 int 表示的是从某个时间节点（比如 January 1, 1970）开始经过的天数。显然这两种理解都合情合理，毕竟从形式上看它们产生的效果都是越靠后的日期对应的整数值越大，而且两种转换都有实际的用处。

问题在于 Date 类型的对象和 int 类型的值之间不存在明确的一对一映射关系。因此在此例中，不定义该类型转换运算符也许会更好。作为替代的手段，类可以定义一个或多个普通的成员函数以从各种不同形式中提取所需的信息。

类型转换运算符可能产生意外结果

在实践中，类很少提供类型转换运算符。在大多数情况下，如果类型转换自动发生，用户可能会感觉比较意外，而不是感觉受到了帮助。然而这条经验法则存在一种例外情况：对于类来说，定义向 bool 的类型转换还是比较普遍的现象。

在 C++ 标准的早期版本中，如果类想定义一个向 bool 的类型转换，则它常常遇到一个问题：因为 bool 是一种算术类型，所以类类型的对象转换成 bool 后就能被用在任何

需要算术类型的上下文中。这样的类型转换可能引发意想不到的结果，特别是当 `istream` 含有向 `bool` 的类型转换时，下面的代码仍将编译通过：

```
int i = 42;
cin << i; // 如果向 bool 的类型转换不是显式的，则该代码在编译器看来将是合法的！
```

这段程序试图将输出运算符作用于输入流。因为 `istream` 本身并没有定义 `<<`，所以本来代码应该产生错误。然而，该代码能使用 `istream` 的 `bool` 类型转换运算符将 `cin` 转换成 `bool`，而这个 `bool` 值接着会被提升成 `int` 并用作内置的左移运算符的左侧运算对象。这样一来，提升后的 `bool` 值（1 或 0）最终会被左移 42 个位置。这一结果显然与我们的预期大相径庭。

显式的类型转换运算符

C++ 11 为了防止这样的异常情况发生，C++11 新标准引入了显式的类型转换运算符（`explicit conversion operator`）：

```
class SmallInt {
public:
    // 编译器不会自动执行这一类型转换
    explicit operator int() const { return val; }
    // 其他成员与之前的版本一致
};
```

和显式的构造函数（参见 7.5.4 节，第 265 页）一样，编译器（通常）也不会将一个显式的类型转换运算符用于隐式类型转换：

```
SmallInt si = 3;      // 正确：SmallInt 的构造函数不是显式的
si + 3;              // 错误：此处需要隐式的类型转换，但类的运算符是显式的
static_cast<int>(si) + 3; // 正确：显式地请求类型转换
```

当类型转换运算符是显式的时，我们也能执行类型转换，不过必须通过显式的强制类型转换才可以。

该规定存在一个例外，即如果表达式被用作条件，则编译器会将显式的类型转换自动应用于它。换句话说，当表达式出现在下列位置时，显式的类型转换将被隐式地执行：

- `if`、`while` 及 `do` 语句的条件部分
- `for` 语句头的条件表达式
- 逻辑非运算符 `(!)`、逻辑或运算符 `(||)`、逻辑与运算符 `(&&)` 的运算对象
- 条件运算符 `(?:)` 的条件表达式。

583 转换为 `bool`

在标准库的早期版本中，IO 类型定义了向 `void*` 的转换规则，以求避免上面提到的问题。在 C++11 新标准下，IO 标准库通过定义一个向 `bool` 的显式类型转换实现同样的目的。

无论我们什么时候在条件中使用流对象，都会使用为 IO 类型定义的 `operator bool`。例如：

```
while (std::cin >> value)
```

`while` 语句的条件执行输入运算符，它负责将数据读入到 `value` 并返回 `cin`。为了对条件求值，`cin` 被 `istream` `operator bool` 类型转换函数隐式地执行了转换。如果 `cin` 的条件状态是 `good`（参见 8.1.2 节，第 280 页），则该函数返回为真；否则该函数返回为假。



向 `bool` 的类型转换通常用在条件部分，因此 `operator bool` 一般定义成 `explicit` 的。

14.9.1 节练习

练习 14.45: 编写类型转换运算符将一个 `Sales_data` 对象分别转换成 `string` 和 `double`，你认为这些运算符的返回值应该是什么？

练习 14.46: 你认为应该为 `Sales_data` 类定义上面两种类型转换运算符吗？应该把它们声明成 `explicit` 的吗？为什么？

练习 14.47: 说明下面这两个类型转换运算符的区别。

```
struct Integral {
    operator const int();
    operator int() const;
};
```

练习 14.48: 你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有向 `bool` 的类型转换运算符吗？如果是，解释原因并说明该运算符是否应该是 `explicit` 的；如果不是，也请解释原因。

练习 14.49: 为上一题提到的类定义一个转换目标是 `bool` 的类型转换运算符，先不用在意这么做是否应该。

14.9.2 避免有二义性的类型转换



如果类中包含一个或多个类型转换，则必须确保在类类型和目标类型之间只存在唯一一种转换方式。否则的话，我们编写的代码将很可能会具有二义性。

在两种情况下可能产生多重转换路径。第一种情况是两个类提供相同的类型转换：例如，当 A 类定义了一个接受 B 类对象的转换构造函数，同时 B 类定义了一个转换目标是 A 类的类型转换运算符时，我们就说它们提供了相同的类型转换。

第二种情况是类定义了多个转换规则，而这些转换涉及的类型本身可以通过其他类型转换联系在一起。最典型的例子是算术运算符，对某个给定的类来说，最好只定义最多一个与算术类型有关的转换规则。



通常情况下，不要为类定义相同的类型转换，也不要在类中定义两个及以上以
上转换源或转换目标是算术类型的转换。

584

实参匹配和相同的类型转换

在下面的例子中，我们定义了两种将 B 转换成 A 的方法：一种使用 B 的类型转换运
算符、另一种使用 A 的以 B 为参数的构造函数：

```
// 最好不要在两个类之间构建相同的类型转换
struct B;
struct A {
    A() = default;
    A(const B&);           // 把一个 B 转换成 A
    // 其他数据成员
```

```

};

struct B {
    operator A() const; // 也是把一个 B 转换成 A
    // 其他数据成员
};
A f(const A&);

B b;
A a = f(b); // 二义性错误：含义是 f(B::operator A())
              // 还是 f(A::A(const B&))？

```

因为同时存在两种由 B 获得 A 的方法，所以造成编译器无法判断应该运行哪个类型转换，也就是说，对 f 的调用存在二义性。该调用可以使用以 B 为参数的 A 的构造函数，也可以使用 B 当中把 B 转换成 A 的类型转换运算符。因为这两个函数效果相当、难分伯仲，所以该调用将产生错误。

如果我们确实想执行上述的调用，就不得不显式地调用类型转换运算符或者转换构造函数：

```

A a1 = f(b.operator A()); // 正确：使用 B 的类型转换运算符
A a2 = f(A(b));          // 正确：使用 A 的构造函数

```

值得注意的是，我们无法使用强制类型转换来解决二义性问题，因为强制类型转换本身也面临二义性。

二义性与转换目标为内置类型的多重类型转换

另外如果类定义了一组类型转换，它们的转换源（或者转换目标）类型本身可以通过其他类型转换联系在一起，则同样会产生二义性的问题。最简单也是最困扰我们的例子就是类当中定义了多个参数都是算术类型的构造函数，或者转换目标都是算术类型的类型转换运算符。

例如，在下面的类中包含两个转换构造函数，它们的参数是两种不同的算术类型；同时还包含两个类型转换运算符，它们的转换目标也恰好是两种不同的算术类型：

585

```

struct A {
    A(int = 0);           // 最好不要创建两个转换源都是算术类型的类型转换
    A(double);
    operator int() const; // 最好不要创建两个转换对象都是算术类型的类型转换
    operator double() const;
    // 其他成员
};

void f2(long double);
A a;
f2(a); // 二义性错误：含义是 f(A::operator int())
        // 还是 f(A::operator double())？

long lg;
A a2(lg); // 二义性错误：含义是 A::A(int) 还是 A::A(double)?

```

在对 f2 的调用中，哪个类型转换都无法精确匹配 long double。然而这两个类型转换都可以使用，只要后面再执行一次生成 long double 的标准类型转换即可。因此，在上面的两个类型转换中哪个都不比另一个更好，调用将产生二义性。

当我们试图用 long 初始化 a2 时也遇到了同样问题，哪个构造函数都无法精确匹配 long 类型。它们在使用构造函数前都要求先将实参进行类型转换：

- 先执行 long 到 double 的标准类型转换，再执行 A(double)
- 先执行 long 到 int 的标准类型转换，再执行 A(int)

编译器没办法区分这两种转换序列的好坏，因此该调用将产生二义性。

调用 f2 及初始化 a2 的过程之所以会产生二义性，根本原因是它们所需的标准类型转换级别一致（参见 6.6.1 节，第 219 页）。当我们使用用户定义的类型转换时，如果转换过程包含标准类型转换，则标准类型转换的级别将决定编译器选择最佳匹配的过程：

```
short s = 42;
// 把 short 提升成 int 优于把 short 转换成 double
A a3(s);           // 使用 A::A(int)
```

在此例中，把 short 提升成 int 的操作要优于把 short 转换成 double 的操作，因此编译器将使用 A::A(int) 构造函数构造 a3，其中实参是 s（提升后）的值。



当我们使用两个用户定义的类型转换时，如果转换函数之前或之后存在标准类型转换，则标准类型转换将决定最佳匹配到底是哪个。

提示：类型转换与运算符

< 586

要想正确地设计类的重载运算符、转换构造函数及类型转换函数，必须加倍小心。尤其是当类同时定义了类型转换运算符及重载运算符时特别容易产生二义性。以下的经验规则可能对你有所帮助：

- 不要令两个类执行相同的类型转换：如果 Foo 类有一个接受 Bar 类对象的构造函数，则不要在 Bar 类中再定义转换目标是 Foo 类的类型转换运算符。
- 避免转换目标是内置算术类型的类型转换。特别是当你已经定义了一个转换成算术类型的类型转换时，接下来
 - 不要再定义接受算术类型的重载运算符。如果用户需要使用这样的运算符，则类型转换操作将转换你的类型的对象，然后使用内置的运算符。
 - 不要定义转换到多种算术类型的类型转换。让标准类型转换完成向其他算术类型转换的工作。

一言以蔽之：除了显式地向 bool 类型的转换之外，我们应该尽量避免定义类型转换函数并尽可能地限制那些“显然正确”的非显式构造函数。

重载函数与转换构造函数

当我们调用重载的函数时，从多个类型转换中进行选择将变得更加复杂。如果两个或多个类型转换都提供了同一种可行匹配，则这些类型转换一样好。

举个例子，当几个重载函数的参数分属不同的类类型时，如果这些类恰好定义了同样的转换构造函数，则二义性问题将进一步提升：

```
struct C {
    C(int);
    // 其他成员
```

```

};

struct D {
    D(int);
    // 其他成员
};

void manip(const C&);

void manip(const D&);

manip(10);           // 二义性错误：含义是 manip(C(10)) 还是 manip(D(10))

```

其中 C 和 D 都包含接受 int 的构造函数，两个构造函数各自匹配 manip 的一个版本。因此调用将具有二义性：它的含义可能是把 int 转换成 C，然后调用 manip 的第一个版本；也可能是把 int 转换成 D，然后调用 manip 的第二个版本。

调用者可以显式地构造正确的类型从而消除二义性：

```
manip(C(10));      // 正确：调用 manip(const C&)
```



如果在调用重载函数时我们需要使用构造函数或者强制类型转换来改变实参的类型，则这通常意味着程序的设计存在不足。

重载函数与用户定义的类型转换

当调用重载函数时，如果两个（或多个）用户定义的类型转换都提供了可行匹配，则我们认为这些类型转换一样好。在这个过程中，我们不会考虑任何可能出现的标准类型转换的级别。只有当重载函数能通过同一个类型转换函数得到匹配时，我们才会考虑其中出现的标准类型转换。

例如当我们调用 manip 时，即使其中一个类定义了需要对实参进行标准类型转换的构造函数，这次调用仍然会具有二义性：

```

struct E {
    E(double);
    // 其他成员
};

void manip2(const C&);

void manip2(const E&);

// 二义性错误：两个不同的用户定义的类型转换都能用在此处
manip2(10);      // 含义是 manip2(C(10)) 还是 manip2(E(double(10)))

```

在此例中，C 有一个转换源为 int 的类型转换，E 有一个转换源为 double 的类型转换。对于 manip2(10) 来说，两个 manip2 函数都是可行的：

- manip2(const C&) 是可行的，因为 C 有一个接受 int 的转换构造函数，该构造函数与实参精确匹配。
- manip2(const E&) 是可行的，因为 E 有一个接受 double 的转换构造函数，而且为了使用该函数我们可以利用标准类型转换把 int 转换成所需的类型。

因为调用重载函数所请求的用户定义的类型转换不止一个且彼此不同，所以该调用具有二义性。即使其中一个调用需要额外的标准类型转换而另一个调用能精确匹配，编译器也会将该调用标示为错误。



在调用重载函数时，如果需要额外的标准类型转换，则该转换的级别只有当所有可行函数都请求同一个用户定义的类型转换时才有用。如果所需的用户定义的类型转换不止一个，则该调用具有二义性。

14.9.2 节练习

练习 14.50：在初始化 ex1 和 ex2 的过程中，可能用到哪些类类型的转换序列呢？说明初始化是否正确并解释原因。

```
struct LongDouble {
    LongDouble(double = 0.0);
    operator double();
    operator float();
};

LongDouble ldObj;
int ex1 = ldObj;
float ex2 = ldObj;
```

练习 14.51：在调用 calc 的过程中，可能用到哪些类型转换序列呢？说明最佳可行函数是如何被选出来的。

```
void calc(int);
void calc(LongDouble);
double dval;
calc(dval); // 哪个 calc?
```

14.9.3 函数匹配与重载运算符



重载的运算符也是重载的函数。因此，通用的函数匹配规则（参见 6.4 节，第 208 页）同样适用于判断在给定的表达式中到底应该使用内置运算符还是重载的运算符。不过当运算符函数出现在表达式中时，候选函数集的规模要比我们使用调用运算符调用函数时更大。如果 a 是一种类类型，则表达式 a sym b 可能是

```
a.operatorsym(b); // a 有一个 operatorsym 成员函数
operatorsym(a, b); // operatorsym 是一个普通函数
```

和普通函数调用不同，我们不能通过调用的形式来区分当前调用的是成员函数还是非成员函数。

当我们使用重载运算符作用于类类型的运算对象时，候选函数中包含该运算符的普通非成员版本和内置版本。除此之外，如果左侧运算对象是类类型，则定义在该类中的运算符的重载版本也包含在候选函数内。

588

当我们调用一个命名的函数时，具有该名字的成员函数和非成员函数不会彼此重载，这是因为我们用来调用命名函数的语法形式对于成员函数和非成员函数来说是不相同的。当我们通过类类型的对象（或者该对象的指针及引用）进行函数调用时，只考虑该类的成员函数。而当我们在表达式中使用重载的运算符时，无法判断正在使用的是成员函数还是非成员函数，因此二者都应该在考虑的范围内。



表达式中运算符的候选函数集既应该包括成员函数，也应该包括非成员函数。

举个例子，我们为 SmallInt 类定义一个加法运算符：

```
class SmallInt {
    friend
    SmallInt operator+(const SmallInt&, const SmallInt&);

public:
    SmallInt(int = 0); // 转换源为 int 的类型转换
    operator int() const { return val; } // 转换目标为 int 的类型转换

private:
    std::size_t val;
};
```

589 可以使用这个类将两个 SmallInt 对象相加，但如果我们试图执行混合模式的算术运算，就将遇到二义性的问题：

```
SmallInt s1, s2;
SmallInt s3 = s1 + s2; // 使用重载的 operator+
int i = s3 + 0; // 二义性错误
```

第一条加法语句接受两个 SmallInt 值并执行+运算符的重载版本。第二条加法语句具有二义性：因为我们可以把 0 转换成 SmallInt，然后使用 SmallInt 的+；或者把 s3 转换成 int，然后对于两个 int 执行内置的加法运算。



如果我们将同一个类既提供了转换目标是算术类型的类型转换，也提供了重载的运算符，则将会遇到重载运算符与内置运算符的二义性问题。

14.9.3 节练习

练习 14.52：在下面的加法表达式中分别选用了哪个 operator+？列出候选函数、可行函数及为每个可行函数的实参执行的类型转换：

```
struct LongDouble {
    // 用于演示的成员 operator+；在通常情况下+是个非成员
    LongDouble operator+(const SmallInt&);
    // 其他成员与 14.9.2 节（第 521 页）一致
};

LongDouble operator+(LongDouble&, double);
SmallInt si;
LongDouble ld;
ld = si + ld;
ld = ld + si;
```

练习 14.53：假设我们已经定义了如第 522 页所示的 SmallInt，判断下面的加法表达式是否合法。如果合法，使用了哪个加法运算符？如果不合法，应该怎样修改代码才能使其合法？

```
SmallInt s1;
double d = s1 + 3.14;
```

小结

590

一个重载的运算符必须是某个类的成员或者至少拥有一个类类型的运算对象。重载运算符的运算对象数量、结合律、优先级与对应的用于内置类型的运算符完全一致。当运算符被定义为类的成员时，类对象的隐式 `this` 指针绑定到第一个运算对象。赋值、下标、函数调用和箭头运算符必须作为类的成员。

如果类重载了函数调用运算符 `operator()`，则该类的对象被称作“函数对象”。这样的对象常用在标准函数中。`lambda` 表达式是一种简便的定义函数对象类的方式。

在类中可以定义转换源或转换目的是该类型本身的类型转换，这样的类型转换将自动执行。只接受单独一个实参的非显式构造函数定义了从实参类型到类类型的类型转换；而非显式的类型转换运算符则定义了从类类型到其他类型的转换。

术语表

调用形式 (call signature) 表示一个可调用对象的接口。在调用形式中包括返回类型以及一个实参类型列表，该列表在一对圆括号内，实参类型之间以逗号分隔。

类类型转换 (class-type conversion) 包括由构造函数定义的从其他类型到类类型的转换以及由类型转换运算符定义的从类类型到其他类型的转换。只接受单独一个实参的非显式构造函数定义了从实参类型到类类型的转换；而类型转换运算符则定义了从类类型到某个指定类型的转换。

类型转换运算符 (conversion operator) 是类的成员函数，定义了从类类型到其他类型的转换。类型转换运算符必须是它要转换的类的成员，并且通常被定义为常量成员。这类运算符既没有返回类型，也不接受参数。它们返回一个可变为转换运算符类型的值，也就是说，`operator int` 返回一个 `int`，`operator string` 返回一个 `string`，依此类推。

显式的类型转换运算符 (explicit conversion operator) 由关键字 `explicit` 限定的类

型转换运算符。这样的运算符用于条件中的隐式类型转换。

函数对象 (function object) 定义了重载调用运算符的对象。在需要使用函数的地方都能使用函数对象。

函数表 (function table) 形如 `map` 或 `vector` 的容器，容器中所存的值可以被调用。

函数模板 (function template) 能够表示任意可调用类型的标准库模板。

重载的运算符 (overloaded operator) 重定义了某种内置运算符的含义的函数。重载的运算符函数含有关键字 `operator`，之后是要定义的符号。重载的运算符必须含有至少一个类类型的运算对象。重载运算符的优先级、结合律、运算对象数量都与其内置版本一致。

用户定义的类型转换 (user-defined conversion) 类类型转换的同义词。

第 15 章

面向对象程序设计

内容

15.1 OOP: 概述	526
15.2 定义基类和派生类	527
15.3 虚函数	536
15.4 抽象基类	540
15.5 访问控制与继承	542
15.6 继承中的类作用域	547
15.7 构造函数与拷贝控制	551
15.8 容器与继承	558
15.9 文本查询程序再探	562
小结	575
术语表	575

面向对象程序设计基于三个基本概念：数据抽象、继承和动态绑定。第 7 章已经介绍了数据抽象的知识，本章将介绍继承和动态绑定。

继承和动态绑定对程序的编写有两方面的影响：一是我们可以更容易地定义与其他类相似但不完全相同的新类；二是在使用这些彼此相似的类编写程序时，我们可以在一定程度上忽略掉它们的区别。

592

在很多程序中都存在着一些相互关联但是有细微差别的概念。例如，书店中不同书籍的定价策略可能不同：有的书籍按原价销售，有的则打折销售。有时，我们给那些购买书籍超过一定数量的顾客打折；另一些时候，则只对前多少本销售的书籍打折，之后就调回原价，等等。面向对象的程序设计（OOP）适用于这类应用。



15.1 OOP：概述

面向对象程序设计（object-oriented programming）的核心思想是数据抽象、继承和动态绑定。通过使用数据抽象，我们可以将类的接口与实现分离（见第 7 章）；使用继承，可以定义相似的类型并对其相似关系建模；使用动态绑定，可以在一定程度上忽略相似类型的区别，而以统一的方式使用它们的对象。

继承

通过继承（inheritance）联系在一起的类构成一种层次关系。通常在层次关系的根部有一个基类（base class），其他类则直接或间接地从基类继承而来，这些继承得到的类称为派生类（derived class）。基类负责定义在层次关系中所有类共同拥有的成员，而每个派生类定义各自特有的成员。

为了对之前提到的不同定价策略建模，我们首先定义一个名为 `Quote` 的类，并将它作为层次关系中的基类。`Quote` 的对象表示按原价销售的书籍。`Quote` 派生出另一个名为 `Bulk_quote` 的类，它表示可以打折销售的书籍。

这些类将包含下面的两个成员函数：

- `isbn()`，返回书籍的 ISBN 编号。该操作不涉及派生类的特殊性，因此只定义在 `Quote` 类中。
- `net_price(size_t)`，返回书籍的实际销售价格，前提是用户购买该书的数量达到一定标准。这个操作显然是类型相关的，`Quote` 和 `Bulk_quote` 都应该包含该函数。

在 C++ 语言中，基类将类型相关的函数与派生类不做改变直接继承的函数区分对待。对于某些函数，基类希望它的派生类各自定义适合自身的版本，此时基类就将这些函数声明成虚函数（virtual function）。因此，我们可以将 `Quote` 类编写成：

```
class Quote {
public:
    std::string isbn() const;
    virtual double net_price(std::size_t n) const;
};
```

593

派生类必须通过使用类派生列表（class derivation list）明确指出它是从哪个（哪些）基类继承而来的。类派生列表的形式是：首先是一个冒号，后面紧跟以逗号分隔的基类列表，其中每个基类前面可以有访问说明符：

```
class Bulk_quote : public Quote {           // Bulk_quote 继承了 Quote
public:
    double net_price(std::size_t) const override;
};
```

因为 `Bulk_quote` 在它的派生列表中使用了 `public` 关键字，因此我们完全可以把

`Bulk_quote` 的对象当成 `Quote` 的对象来使用。

派生类必须在其内部对所有重新定义的虚函数进行声明。派生类可以在这样的函数之前加上 `virtual` 关键字，但是并不是非得这么做。出于 15.3 节（第 538 页）将要解释的原因，C++11 新标准允许派生类显式地注明它将使用哪个成员函数改写基类的虚函数，具体措施是在该函数的形参列表之后增加一个 `override` 关键字。

动态绑定

通过使用 **动态绑定**（dynamic binding），我们能用同一段代码分别处理 `Quote` 和 `Bulk_quote` 的对象。例如，当要购买的书籍和购买的数量都已知时，下面的函数负责打印总的费用：

```
// 计算并打印销售给定数量的某种书籍所得的费用
double print_total(ostream &os,
                    const Quote &item, size_t n)
{
    // 根据传入 item 形参的对象类型调用 Quote::net_price
    // 或者 Bulk_quote::net_price
    double ret = item.net_price(n);
    os << "ISBN: " << item.isbn()      // 调用 Quote::isbn
       << " # sold: " << n << " total due: " << ret << endl;
    return ret;
}
```

该函数非常简单：它返回调用 `net_price()` 的结果，并将该结果连同调用 `isbn()` 的结果一起打印出来。

关于上面的函数有两个有意思的结论：因为函数 `print_total` 的 `item` 形参是基类 `Quote` 的一个引用，所以出于 15.2.3 节（第 534 页）将要解释的原因，我们既能使用基类 `Quote` 的对象调用该函数，也能使用派生类 `Bulk_quote` 的对象调用它；又因为 `print_total` 是使用引用类型调用 `net_price` 函数的，所以出于 15.2.1 节（第 528 页）将要解释的原因，实际传入 `print_total` 的对象类型将决定到底执行 `net_price` 的哪个版本：

```
// basic 的类型是 Quote; bulk 的类型是 Bulk_quote
print_total(cout, basic, 20);           // 调用 Quote 的 net_price
print_total(cout, bulk, 20);            // 调用 Bulk_quote 的 net_price
```

第一条调用句将 `Quote` 对象传入 `print_total`，因此当 `print_total` 调用 `net_price` 时，执行的是 `Quote` 的版本；在第二条调用语句中，实参的类型是 `Bulk_quote`，因此执行的是 `Bulk_quote` 的版本（计算打折信息）。因为在上述过程中函数的运行版本由实参决定，即在运行时选择函数的版本，所以动态绑定有时又被称为运行时绑定（run-time binding）。

< 594



在 C++ 语言中，当我们使用基类的引用（或指针）调用一个虚函数时将发生动态绑定。

15.2 定义基类和派生类

定义基类和派生类的方式在很多方面都与我们已知的定义其他类的方式类似，但是也有一些不同之处。本节将介绍在定义有继承关系的类时可能用到的基本特性。



15.2.1 定义基类

我们首先完成 `Quote` 类的定义：

```
class Quote {
public:
    Quote() = default;           // 关于=default 请参见 7.1.4 节（第 237 页）
    Quote(const std::string &book, double sales_price):
        bookNo(book), price(sales_price) { }
    std::string isbn() const { return bookNo; }
    // 返回给定数量的书籍的销售总额
    // 派生类负责改写并使用不同的折扣计算算法
    virtual double net_price(std::size_t n) const
    { return n * price; }
    virtual ~Quote() = default;   // 对析构函数进行动态绑定
private:
    std::string bookNo;          // 书籍的 ISBN 编号
protected:
    double price = 0.0;          // 代表普通状态下不打折的价格
};
```

对于上面这个类来说，新增的部分是在 `net_price` 函数和析构函数之前增加的 `virtual` 关键字以及最后的 `protected` 访问说明符。我们将在 15.7.1 节（第 552 页）详细介绍虚析构函数的知识，现在只需记住作为继承关系中根节点的类通常都会定义一个虚析构函数。



基类通常都应该定义一个虚析构函数，即使该函数不执行任何实际操作也是如此。

成员函数与继承

派生类可以继承其基类的成员，然而当遇到如 `net_price` 这样与类型相关的操作时，

595 派生类必须对其重新定义。换句话说，派生类需要对这些操作提供自己的新定义以覆盖（`override`）从基类继承而来的旧定义。

在 C++ 语言中，基类必须将它的两种成员函数区分开来：一种是基类希望其派生类进行覆盖的函数；另一种是基类希望派生类直接继承而不要改变的函数。对于前者，基类通常将其定义为虚函数（`virtual`）。当我们使用指针或引用调用虚函数时，该调用将被动态绑定。根据引用或指针所绑定的对象类型不同，该调用可能执行基类的版本，也可能执行某个派生类的版本。

基类通过在其成员函数的声明语句之前加上关键字 `virtual` 使得该函数执行动态绑定。任何构造函数之外的非静态函数（参见 7.6 节，第 268 页）都可以是虚函数。关键字 `virtual` 只能出现在类内部的声明语句之前而不能用于类外部的函数定义。如果基类把一个函数声明成虚函数，则该函数在派生类中隐式地也是虚函数。我们将在 15.3 节（第 536 页）介绍更多关于虚函数的知识。

成员函数如果没被声明为虚函数，则其解析过程发生在编译时而非运行时。对于 `isbn` 成员来说这正是我们希望看到的结果。`isbn` 函数的执行与派生类的细节无关，不管作用于 `Quote` 对象还是 `Bulk_quote` 对象，`isbn` 函数的行为都一样。在我们的继承层次关系中只有一个 `isbn` 函数，因此也就不存在调用 `isbn()` 时到底执行哪个版本的疑问。

访问控制与继承

派生类可以继承定义在基类中的成员，但是派生类的成员函数不一定有权访问从基类继承而来的成员。和其他使用基类的代码一样，派生类能访问公有成员，而不能访问私有成员。不过在某些时候基类中还有这样一种成员，基类希望它的派生类有权访问该成员，同时禁止其他用户访问。我们用受保护的（protected）访问运算符说明这样的成员。

我们的 Quote 类希望它的派生类定义各自的 net_price 函数，因此派生类需要访问 Quote 的 price 成员。此时我们将 price 定义成受保护的。与之相反，派生类访问 bookNo 成员的方式与其他用户是一样的，都是通过调用 isbn 函数，因此 bookNo 被定义成私有的，即使是 Quote 派生出来的类也不能直接访问它。我们将在 15.5 节（第 542 页）介绍更多关于受保护成员的知识。

15.2.1 节练习

练习 15.1：什么是虚成员？

练习 15.2：protected 访问说明符与 private 有何区别？

练习 15.3：定义你自己的 Quote 类和 print_total 函数。

15.2.2 定义派生类

596

派生类必须通过使用类派生列表（class derivation list）明确指出它是从哪个（哪些）基类继承而来的。类派生列表的形式是：首先是一个冒号，后面紧跟以逗号分隔的基类列表，其中每个基类前面可以有以下三种访问说明符中的一个：public、protected 或者 private。



派生类必须将其继承而来的成员函数中需要覆盖的那些重新声明，因此，我们的 Bulk_quote 类必须包含一个 net_price 成员：

```
class Bulk_quote : public Quote {           // Bulk_quote 继承自 Quote
public:
    Bulk_quote() = default;
    Bulk_quote(const std::string&, double, std::size_t, double);
    // 覆盖基类的函数版本以实现基于大量购买的折扣政策
    double net_price(std::size_t) const override;
private:
    std::size_t min_qty = 0;                  // 适用折扣政策的最低购买量
    double discount = 0.0;                   // 以小数表示的折扣额
};
```

我们的 Bulk_quote 类从它的基类 Quote 那里继承了 isbn 函数和 bookNo、price 等数据成员。此外，它还定义了 net_price 的新版本，同时拥有两个新增加的数据成员 min_qty 和 discount。这两个成员分别用于说明享受折扣所需购买的最低数量以及一旦该数量达到之后具体的折扣信息。

我们将在 15.5 节（第 543 页）详细介绍派生列表中用到的访问说明符。现在，我们只需知道访问说明符的作用是控制派生类从基类继承而来的成员是否对派生类的用户可见。

如果一个派生是公有的，则基类的公有成员也是派生类接口的组成部分。此外，我们能将公有派生类型的对象绑定到基类的引用或指针上。因为我们在派生列表中使用了

`public`, 所以 `Bulk_quote` 的接口隐式地包含 `isbn` 函数, 同时在任何需要 `Quote` 的引用或指针的地方我们都能使用 `Bulk_quote` 的对象。

大多数类都只继承自一个类, 这种形式的继承被称作“单继承”, 它构成了本章的主题。关于派生列表中含有多个基类的情况将在 18.3 节(第 710 页)中介绍。

派生类中的虚函数

派生类经常(但不总是)覆盖它继承的虚函数。如果派生类没有覆盖其基类中的某个虚函数, 则该虚函数的行为类似于其他的普通成员, 派生类会直接继承其在基类中的版本。

C++ 11 派生类可以在它覆盖的函数前使用 `virtual` 关键字, 但不是非得这么做。我们将在 15.3 节(第 538 页)介绍其原因, C++11 新标准允许派生类显式地注明它使用某个成员函数覆盖了它继承的虚函数。具体做法是在形参列表后面、或者在 `const` 成员函数(参见 7.1.2 节, 第 231 页)的 `const` 关键字后面、或者在引用成员函数(参见 13.6.3 节, 第 483 页)的引用限定符后面添加一个关键字 `override`。

597 派生类对象及派生类向基类的类型转换

一个派生类对象包含多个组成部分: 一个含有派生类自己定义的(非静态)成员的子对象, 以及一个与该派生类继承的基类对应的子对象, 如果有多个基类, 那么这样的子对象也有多个。因此, 一个 `Bulk_quote` 对象将包含四个数据元素: 它从 `Quote` 继承而来的 `bookNo` 和 `price` 数据成员, 以及 `Bulk_quote` 自己定义的 `min_qty` 和 `discount` 成员。

C++ 标准并没有明确规定派生类的对象在内存中如何分布, 但是我们可以认为 `Bulk_quote` 的对象包含如图 15.1 所示的两部分。



在一个对象中, 继承自基类的部分和派生类自定义的部分不一定是连续存储的。图 15.1 只是表示类工作机理的概念模型, 而非物理模型。

图 15.1: Bulk_quote 对象的概念结构

因为在派生类对象中含有与其基类对应的组成部分, 所以我们能把派生类的对象当成基类对象来使用, 而且我们也能将基类的指针或引用绑定到派生类对象中的基类部分上。

```

Quote item;           // 基类对象
Bulk_quote bulk;     // 派生类对象
Quote *p = &item;      // p 指向 Quote 对象
p = &bulk;            // p 指向 bulk 的 Quote 部分
Quote &r = bulk;      // r 绑定到 bulk 的 Quote 部分

```

这种转换通常称为派生类到基类的(derived-to-base)类型转换。和其他类型转换一样, 编译器会隐式地执行派生类到基类的转换(参见 4.11 节, 第 141 页)。

这种隐式特性意味着我们可以把派生类对象或者派生类对象的引用用在需要基类引

用的地方；同样的，我们也可以把派生类对象的指针用在需要基类指针的地方。



在派生类对象中含有与其基类对应的组成部分，这一事实是继承的关键所在。

派生类构造函数

<598

尽管在派生类对象中含有从基类继承而来的成员，但是派生类并不能直接初始化这些成员。和其他创建了基类对象的代码一样，派生类也必须使用基类的构造函数来初始化它的基类部分。



每个类控制它自己的成员初始化过程。

派生类对象的基类部分与派生类对象自己的数据成员都是在构造函数的初始化阶段（参见 7.5.1 节，第 258 页）执行初始化操作的。类似于我们初始化成员的过程，派生类构造函数同样是通过构造函数初始化列表来将实参传递给基类构造函数的。例如，接受四个参数的 Bulk_quote 构造函数如下所示：

```
Bulk_quote(const std::string& book, double p,
           std::size_t qty, double disc) :
    Quote(book, p), min_qty(qty), discount(disc) {}
// 与之前一致
};
```

该函数将它的前两个参数（分别表示 ISBN 和价格）传递给 Quote 的构造函数，由 Quote 的构造函数负责初始化 Bulk_quote 的基类部分（即 bookNo 成员和 price 成员）。当（空的）Quote 构造函数体结束后，我们构建的对象的基类部分也就完成初始化了。接下来初始化由派生类直接定义的 min_qty 成员和 discount 成员。最后运行 Bulk_quote 构造函数的（空的）函数体。

除非我们特别指出，否则派生类对象的基类部分会像数据成员一样执行默认初始化。如果想使用其他的基类构造函数，我们需要以类名加圆括号内的实参列表的形式为构造函数提供初始值。这些实参将帮助编译器决定到底应该选用哪个构造函数来初始化派生类对象的基类部分。



首先初始化基类的部分，然后按照声明的顺序依次初始化派生类的成员。

派生类使用基类的成员

派生类可以访问基类的公有成员和受保护成员：

```
// 如果达到了购买书籍的某个最低限量值，就可以享受折扣价格了
double Bulk_quote::net_price(size_t cnt) const
{
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

该函数产生一个打折后的价格：如果给定的数量超过了 min_qty，则将 discount (一 <599

个小于 1 大于 0 的数) 作用于 price。

我们将在 15.6 节 (第 547 页) 进一步讨论作用域, 目前只需要了解派生类的作用域嵌套在基类的作用域之内。因此, 对于派生类的一个成员来说, 它使用派生类成员 (例如 min_qty 和 discount) 的方式与使用基类成员 (例如 price) 的方式没什么不同。

关键概念: 遵循基类的接口

必须明确一点: 每个类负责定义各自的接口。要想与类的对象交互必须使用该类的接口, 即使这个对象是派生类的基类部分也是如此。

因此, 派生类对象不能直接初始化基类的成员。尽管从语法上来说我们可以在派生类构造函数体内给它的公有或受保护的基类成员赋值, 但是最好不要这么做。和使用基类的其他场合一样, 派生类应该遵循基类的接口, 并且通过调用基类的构造函数来初始化那些从基类中继承而来的成员。

继承与静态成员

如果基类定义了一个静态成员 (参见 7.6 节, 第 268 页), 则在整个继承体系中只存在该成员的唯一定义。不论从基类中派生出来多少个派生类, 对于每个静态成员来说都只存在唯一的实例。

```
class Base {
public:
    static void statmem();
};

class Derived : public Base {
    void f(const Derived&);
};
```

静态成员遵循通用的访问控制规则, 如果基类中的成员是 private 的, 则派生类无权访问它。假设某静态成员是可访问的, 则我们既能通过基类使用它也能通过派生类使用它:

```
void Derived::f(const Derived &derived_obj)
{
    Base::statmem();           // 正确: Base 定义了 statmem
    Derived::statmem();        // 正确: Derived 继承了 statmem
    // 正确: 派生类的对象能访问基类的静态成员
    derived_obj.statmem();     // 通过 Derived 对象访问
    statmem();                 // 通过 this 对象访问
}
```

600> 派生类的声明

派生类的声明与其他类差别不大 (参见 7.3.3 节, 第 250 页), 声明中包含类名但是不包含它的派生列表:

```
class Bulk_quote : public Quote; // 错误: 派生列表不能出现在这里
class Bulk_quote;             // 正确: 声明派生类的正确方式
```

一条声明语句的目的是令程序知晓某个名字的存在以及该名字表示一个什么样的实体, 如一个类、一个函数或一个变量等。派生列表以及与定义有关的其他细节必须与类的主体一起出现。

被用作基类的类

如果我们想将某个类用作基类，则该类必须已经定义而非仅仅声明：

```
class Quote; // 声明但未定义
// 错误: Quote 必须被定义
class Bulk_quote : public Quote { ... };
```

这一规定的原因显而易见：派生类中包含并且可以使用它从基类继承而来的成员，为了使用这些成员，派生类当然要知道它们是什么。因此该规定还有一层隐含的意思，即一个类不能派生它本身。

一个类是基类，同时它也可以是一个派生类：

```
class Base { /* ... */ };
class D1: public Base { /* ... */ };
class D2: public D1 { /* ... */ };
```

在这个继承关系中，Base 是 D1 的直接基类 (direct base)，同时是 D2 的间接基类 (indirect base)。直接基类出现在派生列表中，而间接基类由派生类通过其直接基类继承而来。

每个类都会继承直接基类的所有成员。对于一个最终的派生类来说，它会继承其直接基类的成员；该直接基类的成员又含有其基类的成员；依此类推直至继承链的顶端。因此，最终的派生类将包含它的直接基类的子对象以及每个间接基类的子对象。

防止继承的发生

有时我们会定义这样一种类，我们不希望其他类继承它，或者不想考虑它是否适合作为一个基类。为了实现这一目的，C++11 新标准提供了一种防止继承发生的方法，即在类名后跟一个关键字 final：

```
class NoDerived final { /* */ }; // NoDerived 不能作为基类
class Base { /* */ };
// Last 是 final 的；我们不能继承 Last
class Last final : Base { /* */ }; // Last 不能作为基类
class Bad : NoDerived { /* */ }; // 错误: NoDerived 是 final 的
class Bad2 : Last { /* */ }; // 错误: Last 是 final 的
```

15.2.2 节练习

C++
11

601

练习 15.4：下面哪条声明语句是不正确的？请解释原因。

- class Base { ... };
- (a) class Derived : public Derived { ... };
- (b) class Derived : private Base { ... };
- (c) class Derived : public Base;

练习 15.5：定义你自己的 Bulk_quote 类。

练习 15.6：将 Quote 和 Bulk_quote 的对象传给 15.2.1 节（第 529 页）练习中的 print_total 函数，检查该函数是否正确。

练习 15.7：定义一个类使其实现一种数量受限的折扣策略，具体策略是：当购买书籍的数量不超过一个给定的限量时享受折扣，如果购买量一旦超过了限量，则超出的部分将以原价销售。



15.2.3 类型转换与继承



理解基类和派生类之间的类型转换是理解 C++ 语言面向对象编程的关键所在。

通常情况下，如果我们想把引用或指针绑定到一个对象上，则引用或指针的类型应与对象的类型一致（参见 2.3.1 节，第 46 页和 2.3.2 节，第 47 页），或者对象的类型含有一个可接受的 `const` 类型转换规则（参见 4.11.2 节，第 144 页）。存在继承关系的类是一个重要的例外：我们可以将基类的指针或引用绑定到派生类对象上。例如，我们可以用 `Quote&` 指向一个 `Bulk_quote` 对象，也可以把一个 `Bulk_quote` 对象的地址赋给一个 `Quote*`。

可以将基类的指针或引用绑定到派生类对象上有一层极为重要的含义：当使用基类的引用（或指针）时，实际上我们并不清楚该引用（或指针）所绑定对象的真实类型。该对象可能是基类的对象，也可能是派生类的对象。



和内置指针一样，智能指针类（参见 12.1 节，第 400 页）也支持派生类向基类的类型转换，这意味着我们可以将一个派生类对象的指针存储在一个基类的智能指针内。



静态类型与动态类型

当我们使用存在继承关系的类型时，必须将一个变量或其他表达式的 **静态类型**（static type）与该表达式表示对象的 **动态类型**（dynamic type）区分开来。表达式的静态类型在编译时总是已知的，它是变量声明时的类型或表达式生成的类型；动态类型则是变量或表达式表示的内存中的对象的类型。动态类型直到运行时才可知。

602 >

例如，当 `print_total` 调用 `net_price` 时（参见 15.1 节，第 527 页）：

```
double ret = item.net_price(n);
```

我们知道 `item` 的静态类型是 `Quote&`，它的动态类型则依赖于 `item` 绑定的实参，动态类型直到在运行时调用该函数时才会知道。如果我们传递一个 `Bulk_quote` 对象给 `print_total`，则 `item` 的静态类型将与它的动态类型不一致。如前所述，`item` 的静态类型是 `Quote&`，而在此例中它的动态类型则是 `Bulk_quote`。

如果表达式既不是引用也不是指针，则它的动态类型永远与静态类型一致。例如，`Quote` 类型的变量永远是一个 `Quote` 对象，我们无论如何都不能改变该变量对应的对象的类型。



基类的指针或引用的静态类型可能与其动态类型不一致，读者一定要理解其中的原因。

不存在从基类向派生类的隐式类型转换……

之所以存在派生类向基类的类型转换是因为每个派生类对象都包含一个基类部分，而基类的引用或指针可以绑定到该基类部分上。一个基类的对象既可以以独立的形式存在，也可以作为派生类对象的一部分存在。如果基类对象不是派生类对象的一部分，则它只含有基类定义的成员，而不含有派生类定义的成员。

因为一个基类的对象可能是派生类对象的一部分，也可能不是，所以不存在从基类向派生类的自动类型转换：

```
Quote base;
Bulk_quote* bulkP = &base;           // 错误：不能将基类转换成派生类
Bulk_quote& bulkRef = base;         // 错误：不能将基类转换成派生类
```

如果上述赋值是合法的，则我们有可能会使用 bulkP 或 bulkRef 访问 base 中本不存在的成员。

除此之外还有一种情况显得有点特别，即使一个基类指针或引用绑定在一个派生类对象上，我们也不能执行从基类向派生类的转换：

```
Bulk_quote bulk;
Quote *itemP = &bulk;                // 正确：动态类型是 Bulk_quote
Bulk_quote *bulkP = itemP;           // 错误：不能将基类转换成派生类
```

编译器在编译时无法确定某个特定的转换在运行时是否安全，这是因为编译器只能通过检查指针或引用的静态类型来推断该转换是否合法。如果在基类中含有一个或多个虚函数，我们可以使用 `dynamic_cast`（参见 19.2.1 节，第 730 页）请求一个类型转换，该转换的安全检查将在运行时执行。同样，如果我们已知某个基类向派生类的转换是安全的，则我们可以使用 `static_cast`（参见 4.11.3 节，第 144 页）来强制覆盖掉编译器的检查工作。

……在对象之间不存在类型转换



603 派生类向基类的自动类型转换只对指针或引用类型有效，在派生类类型和基类类型之间不存在这样的转换。很多时候，我们确实希望将派生类对象转换成它的基类类型，但是这种转换的实际发生过程往往与我们期望的有所差别。

请注意，当我们初始化或赋值一个类类型的对象时，实际上是在调用某个函数。当执行初始化时，我们调用构造函数（参见 13.1.1 节，第 440 页和 13.6.2 节，第 473 页）；而当执行赋值操作时，我们调用赋值运算符（参见 13.1.2 节，第 443 页和 13.6.2 节，第 474 页）。这些成员通常都包含一个参数，该参数的类型是类类型的 `const` 版本的引用。

因为这些成员接受引用作为参数，所以派生类向基类的转换允许我们给基类的拷贝/移动操作传递一个派生类的对象。这些操作不是虚函数。当我们给基类的构造函数传递一个派生类对象时，实际运行的构造函数是基类中定义的那个，显然该构造函数只能处理基类自己的成员。类似的，如果我们将一个派生类对象赋值给一个基类对象，则实际运行的赋值运算符也是基类中定义的那个，该运算符同样只能处理基类自己的成员。

例如，我们的书店类使用了合成版本的拷贝和赋值操作（参见 13.1.1 节，第 440 页和 13.1.2 节，第 444 页）。关于拷贝控制与继承的知识将在 15.7.2 节（第 552 页）做更详细的介绍，现在我们只需要知道合成版本会像其他类一样逐成员地执行拷贝或赋值操作：

```
Bulk_quote bulk;                  // 派生类对象
Quote item(bulk);                // 使用 Quote::Quote(const Quote&) 构造函数
item = bulk;                     // 调用 Quote::operator=(const Quote&)
```

当构造 `item` 时，运行 `Quote` 的拷贝构造函数。该函数只能处理 `bookNo` 和 `price` 两个成员，它负责拷贝 `bulk` 中 `Quote` 部分的成员，同时忽略掉 `bulk` 中 `Bulk_quote` 部分的成员。类似的，对于将 `bulk` 赋值给 `item` 的操作来说，只有 `bulk` 中 `Quote` 部分的成员被赋值给 `item`。

因为在上述过程中会忽略 `Bulk_quote` 部分，所以我们可以说 `bulk` 的 `Bulk_quote` 部分被切掉（sliced down）了。



当我们用一个派生类对象为一个基类对象初始化或赋值时，只有该派生类对象中的基类部分会被拷贝、移动或赋值，它的派生类部分将被忽略掉。

15.2.3 节练习

练习 15.8：给出静态类型和动态类型的定义。

练习 15.9：在什么情况下表达式的静态类型可能与动态类型不同？请给出三个静态类型与动态类型不同的例子。

练习 15.10：回忆我们在 8.1 节（第 279 页）进行的讨论，解释第 284 页中将 ifstream 传递给 Sales_data 的 read 函数的程序是如何工作的。

关键概念：存在继承关系的类型之间的转换规则

要想理解在具有继承关系的类之间发生的类型转换，有三点非常重要：

- 从派生类向基类的类型转换只对指针或引用类型有效。
- 基类向派生类不存在隐式类型转换。
- 和任何其他成员一样，派生类向基类的类型转换也可能会由于访问受限而变得不可行。我们将在 15.5 节（第 544 页）详细介绍可访问性的问题。

尽管自动类型转换只对指针或引用类型有效，但是继承体系中的大多数类仍然（显式或隐式地）定义了拷贝控制成员（参见第 13 章）。因此，我们通常能够将一个派生类对象拷贝、移动或赋值给一个基类对象。不过需要注意的是，这种操作只处理派生类对象的基类部分。



15.3 虚函数

如前所述，在 C++ 语言中，当我们使用基类的引用或指针调用一个虚成员函数时会执行动态绑定（参见 15.1 节，第 527 页）。因为我们直到运行时才能知道到底调用了哪个版本的虚函数，所以所有虚函数都必须有定义。通常情况下，如果我们不使用某个函数，则无须为该函数提供定义（参见 6.1.2 节，第 186 页）。但是我们必须为每一个虚函数都提供定义，而不管它是否被用到了，这是因为连编译器也无法确定到底会使用哪个虚函数。

对虚函数的调用可能在运行时才被解析

当某个虚函数通过指针或引用调用时，编译器产生的代码直到运行时才能确定应该调用哪个版本的函数。被调用的函数是与绑定到指针或引用上的对象的动态类型相匹配的那个。

举个例子，考虑 15.1 节（第 527 页）的 print_total 函数，该函数通过其名为 item 的参数来进一步调用 net_price，其中 item 的类型是 Quote&。因为 item 是引用而且 net_price 是虚函数，所以我们到底调用 net_price 的哪个版本完全依赖于运行时绑定到 item 的实参的实际（动态）类型：

```
Quote base("0-201-82470-1", 50);
print_total(cout, base, 10);           // 调用 Quote::net_price
Bulk_quote derived("0-201-82470-1", 50, 5, .19);
```

```
print_total(cout, derived, 10);           // 调用 Bulk_quote::net_price
```

在第一条调用语句中，item 绑定到 Quote 类型的对象上，因此当 print_total 调用 net_price 时，运行在 Quote 中定义的版本。在第二条调用语句中，item 绑定到 Bulk_quote 类型的对象上，因此 print_total 调用 Bulk_quote 定义的 net_price。605

必须要搞清楚的一点是，动态绑定只有当我们通过指针或引用调用虚函数时才会发生。

```
base = derived;                         // 把 derived 的 Quote 部分拷贝给 base  
base.net_price(20);                   // 调用 Quote::net_price
```

当我们通过一个具有普通类型（非引用非指针）的表达式调用虚函数时，在编译时就会将调用的版本确定下来。例如，如果我们使用 base 调用 net_price，则应该运行 net_price 的哪个版本是显而易见的。我们可以改变 base 表示的对象的值（即内容），但是不会改变该对象的类型。因此，在编译时该调用就会被解析成 Quote 的 net_price。

关键概念：C++的多态性

OOP 的核心思想是多态性（polymorphism）。多态性这个词源自希腊语，其含义是“多种形式”。我们把具有继承关系的多个类型称为多态类型，因为我们能使用这些类型的“多种形式”而无须在意它们的差异。引用或指针的静态类型与动态类型不同这一事实正是 C++ 语言支持多态性的根本所在。

当我们使用基类的引用或指针调用基类中定义的一个函数时，我们并不知道该函数真正作用的对象是什么类型，因为它可能是一个基类的对象也可能是一个派生类的对象。如果该函数是虚函数，则直到运行时才会决定到底执行哪个版本，判断的依据是引用或指针所绑定的对象的真实类型。

另一方面，对非虚函数的调用在编译时进行绑定。类似的，通过对象进行的函数（虚函数或非虚函数）调用也在编译时绑定。对象的类型是确定不变的，我们无论如何都不可能令对象的动态类型与静态类型不一致。因此，通过对象进行的函数调用将在编译时绑定到该对象所属类中的函数版本上。



当且仅当对通过指针或引用调用虚函数时，才会在运行时解析该调用，也只有在这种情况下对象的动态类型才有可能与静态类型不同。

派生类中的虚函数

当我们在派生类中覆盖了某个虚函数时，可以再一次使用 virtual 关键字指出该函数的性质。然而这么做并非必须，因为一旦某个函数被声明成虚函数，则在所有派生类中它都是虚函数。

一个派生类的函数如果覆盖了某个继承而来的虚函数，则它的形参类型必须与被它覆盖的基类函数完全一致。

同样，派生类中虚函数的返回类型也必须与基类函数匹配。该规则存在一个例外，当类的虚函数返回类型是类本身的指针或引用时，上述规则无效。也就是说，如果 D 由 B 派生得到，则基类的虚函数可以返回 B* 而派生类的对应函数可以返回 D*，只不过这样的返回类型要求从 D 到 B 的类型转换是可访问的。15.5 节（第 544 页）将介绍如何确定一个基类的可访问性，在 15.8.1 节（第 561 页）中我们将看到这种虚函数的一个实际例子。606



基类中的虚函数在派生类中隐含地也是一个虚函数。当派生类覆盖了某个虚函数时，该函数在基类中的形参必须与派生类中的形参严格匹配。

final 和 override 说明符

如我们将要在 15.6 节（第 550 页）介绍的，派生类如果定义了一个函数与基类中虚函数的名字相同但是形参列表不同，这仍然是合法的行为。编译器将认为新定义的这个函数与基类中原有的函数是相互独立的。这时，派生类的函数并没有覆盖掉基类中的版本。就实际的编程习惯而言，这种声明往往意味着发生了错误，因为我们可能原本希望派生类能覆盖掉基类中的虚函数，但是一不小心把形参列表弄错了。

C++
11

要想调试并发现这样的错误显然非常困难。在 C++11 新标准中我们可以使用 `override` 关键字来说明派生类中的虚函数。这么做的好处是在使得程序员的意图更加清晰的同时让编译器可以为我们发现一些错误，后者在编程实践中显得更加重要。如果我们使用 `override` 标记了某个函数，但该函数并没有覆盖已存在的虚函数，此时编译器将报错：

```
struct B {
    virtual void f1(int) const;
    virtual void f2();
    void f3();
};

struct D1 : B {
    void f1(int) const override;           // 正确: f1 与基类中的 f1 匹配
    void f2(int) override;                 // 错误: B 没有形如 f2(int) 的函数
    void f3() override;                   // 错误: f3 不是虚函数
    void f4() override;                   // 错误: B 没有名为 f4 的函数
};
```

在 D1 中，`f1` 的 `override` 说明符是正确的，因为基类和派生类中的 `f1` 都是 `const` 成员，并且它们都接受一个 `int` 返回 `void`，所以 D1 中的 `f1` 正确地覆盖了它从 B 中继承而来的虚函数。

D1 中 `f2` 的声明与 B 中 `f2` 的声明不匹配，显然 B 中定义的 `f2` 不接受任何参数而 D1 的 `f2` 接受一个 `int`。因为这两个声明不匹配，所以 D1 的 `f2` 不能覆盖 B 的 `f2`，它是一个新函数，仅仅是名字恰好与原来的函数一样而已。因为我们使用 `override` 所表达的意思是我们希望能覆盖基类中的虚函数而实际上并未做到，所以编译器会报错。

607

因为只有虚函数才能被覆盖，所以编译器会拒绝 D1 的 `f3`。该函数不是 B 中的虚函数，因此它不能被覆盖。类似的，`f4` 的声明也会发生错误，因为 B 中根本就没有名为 `f4` 的函数。

我们还能把某个函数指定为 `final`，如果我们已经把函数定义成 `final` 了，则之后任何尝试覆盖该函数的操作都将引发错误：

```
struct D2 : B {
    // 从 B 继承 f2() 和 f3()，覆盖 f1(int)
    void f1(int) const final;      // 不允许后续的其他类覆盖 f1(int)
};

struct D3 : D2 {
    void f2();                     // 正确：覆盖从间接基类 B 继承而来的 f2
    void f1(int) const;            // 错误：D2 已经将 f2 声明成 final
};
```

`final` 和 `override` 说明符出现在形参列表（包括任何 `const` 或引用修饰符）以及尾置返回类型（参见 6.3.3 节，第 206 页）之后。

虚函数与默认实参

和其他函数一样，虚函数也可以拥有默认实参（参见 6.5.1 节，第 211 页）。如果某次函数调用使用了默认实参，则该实参值由本次调用的静态类型决定。

换句话说，如果我们通过基类的引用或指针调用函数，则使用基类中定义的默认实参，即使实际运行的是派生类中的函数版本也是如此。此时，传入派生类函数的将是基类函数定义的默认实参。如果派生类函数依赖不同的实参，则程序结果将与我们的预期不符。



如果虚函数使用默认实参，则基类和派生类中定义的默认实参最好一致。

回避虚函数的机制

在某些情况下，我们希望对虚函数的调用不要进行动态绑定，而是强迫其执行虚函数的某个特定版本。使用作用域运算符可以实现这一目的，例如下面的代码：

```
// 强行调用基类中定义的函数版本而不管 baseP 的动态类型到底是什么  
double undiscounted = baseP->Quote::net_price(42);
```

该代码强行调用 `Quote` 的 `net_price` 函数，而不管 `baseP` 实际指向的对象类型到底是什么。该调用将在编译时完成解析。



通常情况下，只有成员函数（或友元）中的代码才需要使用作用域运算符来回避虚函数的机制。

什么时候我们需要回避虚函数的默认机制呢？通常是当一个派生类的虚函数调用它覆盖的基类的虚函数版本时。在此情况下，基类的版本通常完成继承层次中所有类型都要做的共同任务，而派生类中定义的版本需要执行一些与派生类本身密切相关的操作。



如果一个派生类虚函数需要调用它的基类版本，但是没有使用作用域运算符，则在运行时该调用将被解析为对派生类版本自身的调用，从而导致无限递归。

608

15.3 节练习

练习 15.11：为你的 `Quote` 类体系添加一个名为 `debug` 的虚函数，令其分别显示每个类的数据成员。

练习 15.12：有必要将一个成员函数同时声明成 `override` 和 `final` 吗？为什么？

练习 15.13：给定下面的类，解释每个 `print` 函数的机理：

```
class base {  
public:  
    string name() { return basename; }  
    virtual void print(ostream &os) { os << basename; }  
private:
```

```

        string basename;
    };
    class derived : public base {
public:
    void print(ostream &os) { print(os); os << " " << i; }
private:
    int i;
};

```

在上述代码中存在问题吗？如果有，你该如何修改它？

练习 15.14：给定上一题中的类以及下面这些对象，说明在运行时调用哪个函数：

base bobj;	base *bp1 = &bobj;	base &br1 = bobj;
derived dobj;	base *bp2 = &dobj;	base &br2 = dobj;
(a) bobj.print();	(b) dobj.print();	(c) bp1->name();
(d) bp2->name();	(e) br1.print();	(f) br2.print();

15.4 抽象基类

假设我们希望扩展书店程序并令其支持几种不同的折扣策略。除了购买量超过一定数量享受折扣外，我们也可能提供另外一种策略，即购买量不超过某个限额时可以享受折扣，但是一旦超过限额就要按原价支付。或者折扣策略还可能是购买量超过一定数量后购买的全部书籍都享受折扣，否则全都不打折。

上面的每个策略都要求一个购买量的值和一个折扣值。我们可以定义一个新的名为 Disc_quote 的类来支持不同的折扣策略，其中 Disc_quote 负责保存购买量的值和折扣值。其他的表示某种特定策略的类（如 Bulk_quote）将分别继承自 Disc_quote，每个派生类通过定义自己的 net_price 函数来实现各自的折扣策略。

在定义 Disc_quote 类之前，首先要确定它的 net_price 函数完成什么工作。显然我们的 Disc_quote 类与任何特定的折扣策略都无关，因此 Disc_quote 类中的 net_price 函数是没有实际含义的。

我们可以在 Disc_quote 类中不定义新的 net_price，此时，Disc_quote 将继承 Quote 中的 net_price 函数。

然而，这样的设计可能导致用户编写出一些无意义的代码。用户可能会创建一个 Disc_quote 对象并为其提供购买量和折扣值，如果将该对象传给一个像 print_total 这样的函数，则程序将调用 Quote 版本的 net_price。显然，最终计算出的销售价格并没有考虑我们在创建对象时提供的折扣值，因此上述操作毫无意义。

纯虚函数

认真思考上面描述的情形我们可以发现，关键问题不仅仅是不知道应该如何定义 net_price，而是我们根本就不希望用户创建一个 Disc_quote 对象。Disc_quote 类表示的是一本打折书籍的通用概念，而非某种具体的折扣策略。

我们可以将 net_price 定义成纯虚（pure virtual）函数从而令程序实现我们的设计意图，这样做可以清晰明了地告诉用户当前这个 net_price 函数是没有实际意义的。和普通的虚函数不一样，一个纯虚函数无须定义。我们通过在函数体的位置（即在声明语句

的分号之前) 书写=0 就可以将一个虚函数说明为纯虚函数。其中, =0 只能出现在类内部的虚函数声明语句处:

```
// 用于保存折扣值和购买量的类, 派生类使用这些数据可以实现不同的价格策略
class Disc_quote : public Quote {
public:
    Disc_quote() = default;
    Disc_quote(const std::string& book, double price,
               std::size_t qty, double disc):
        Quote(book, price),
        quantity(qty), discount(disc) { }
    double net_price(std::size_t) const = 0;
protected:
    std::size_t quantity = 0;           // 折扣适用的购买量
    double discount = 0.0;            // 表示折扣的小数值
};
```

和我们之前定义的 Bulk_quote 类一样, Disc_quote 也分别定义了一个默认构造函数和一个接受四个参数的构造函数。尽管我们不能直接定义这个类的对象, 但是 Disc_quote 的派生类构造函数将会使用 Disc_quote 的构造函数来构建各个派生类对象的 Disc_quote 部分。其中, 接受四个参数的构造函数将前两个参数传递给 Quote 的构造函数, 然后直接初始化自己的成员 discount 和 quantity。默认构造函数则对这些成员进行默认初始化。

值得注意的是, 我们也可以为纯虚函数提供定义, 不过函数体必须定义在类的外部。◀ 610也就是说, 我们不能在类的内部为一个=0 的函数提供函数体。

含有纯虚函数的类是抽象基类

含有(或者未经覆盖直接继承)纯虚函数的类是**抽象基类**(abstract base class)。抽象基类负责定义接口, 而后续的其他类可以覆盖该接口。我们不能(直接)创建一个抽象基类的对象。因为 Disc_quote 将 net_price 定义成了纯虚函数, 所以我们不能定义 Disc_quote 的对象。我们可以定义 Disc_quote 的派生类的对象, 前提是这些类覆盖了 net_price 函数:

```
// Disc_quote 声明了纯虚函数, 而 Bulk_quote 将覆盖该函数
Disc_quote discounted;           // 错误: 不能定义 Disc_quote 的对象
Bulk_quote bulk;                 // 正确: Bulk_quote 中没有纯虚函数
```

Disc_quote 的派生类必须给出自己的 net_price 定义, 否则它们仍将是抽象基类。



我们不能创建抽象基类的对象。

派生类构造函数只初始化它的直接基类

接下来可以重新实现 Bulk_quote 了, 这一次我们让它继承 Disc_quote 而非直接继承 Quote:

```
// 当同一书籍的销售量超过某个值时启用折扣
// 折扣的值是一个小于 1 的正的小数值, 以此来降低正常销售价格
class Bulk_quote : public Disc_quote {
public:
    Bulk_quote() = default;
```

```

Bulk_quote(const std::string& book, double price,
           std::size_t qty, double disc):
    Disc_quote(book, price, qty, disc) { }
// 覆盖基类中的函数版本以实现一种新的折扣策略
double net_price(std::size_t) const override;
};

}

```

这个版本的 `Bulk_quote` 的直接基类是 `Disc_quote`, 间接基类是 `Quote`。每个 `Bulk_quote` 对象包含三个子对象: 一个(空的)`Bulk_quote`部分、一个`Disc_quote`子对象和一个`Quote`子对象。

如前所述, 每个类各自控制其对象的初始化过程。因此, 即使 `Bulk_quote` 没有自己的数据成员, 它也仍然需要像原来一样提供一个接受四个参数的构造函数。该构造函数将它的实参传递给 `Disc_quote` 的构造函数, 随后 `Disc_quote` 的构造函数继续调用 `Quote` 的构造函数。`Quote` 的构造函数首先初始化 `bulk` 的 `bookNo` 和 `price` 成员, 当 `Quote` 的构造函数结束后, 开始运行 `Disc_quote` 的构造函数并初始化 `quantity` 和 `discount` 成员, 最后运行 `Bulk_quote` 的构造函数, 该函数无须执行实际的初始化或其他工作。

611 >

关键概念: 重构

在 `Quote` 的继承体系中增加 `Disc_quote` 类是重构 (refactoring) 的一个典型示例。重构负责重新设计类的体系以便将操作和/或数据从一个类移动到另一个类中。对于面向对象的应用程序来说, 重构是一种很普遍的现象。

值得注意的是, 即使我们改变了整个继承体系, 那些使用了 `Bulk_quote` 或 `Quote` 的代码也无须进行任何改动。不过一旦类被重构 (或以其他方式被改变), 就意味着我们必须重新编译含有这些类的代码了。

15.4 节练习

练习 15.15: 定义你自己的 `Disc_quote` 和 `Bulk_quote`。

练习 15.16: 改写你在 15.2.2 节 (第 533 页) 练习中编写的数量受限的折扣策略, 令其继承 `Disc_quote`。

练习 15.17: 尝试定义一个 `Disc_quote` 的对象, 看看编译器给出的错误信息是什么?



15.5 访问控制与继承

每个类分别控制自己的成员初始化过程 (参见 15.2.2 节, 第 531 页), 与之类似, 每个类还分别控制着其成员对于派生类来说是否可访问 (accessible)。

受保护的成员

如前所述, 一个类使用 `protected` 关键字来声明那些它希望与派生类分享但是不想被其他公共访问使用的成员。`protected` 说明符可以看做是 `public` 和 `private` 中和后的产物:

- 和私有成员类似, 受保护的成员对于类的用户来说是不可访问的。

- 和公有成员类似，受保护的成员对于派生类的成员和友元来说是可访问的。

此外，`protected` 还有另外一条重要的性质。

- 派生类的成员或友元只能通过派生类对象来访问基类的受保护成员。派生类对于一个基类对象中的受保护成员没有任何访问特权。

为了理解最后一条规则，请考虑如下的例子：

612

```
class Base {
protected:
    int prot_mem; // protected 成员
};

class Sneaky : public Base {
    friend void clobber(Sneaky&); // 能访问 Sneaky::prot_mem
    friend void clobber(Base&); // 不能访问 Base::prot_mem
    int j; // j 默认是 private
};

// 正确：clobber 能访问 Sneaky 对象的 private 和 protected 成员
void clobber(Sneaky &s) { s.j = s.prot_mem = 0; }
// 错误：clobber 不能访问 Base 的 protected 成员
void clobber(Base &b) { b.prot_mem = 0; }
```

如果派生类（及其友元）能访问基类对象的受保护成员，则上面的第二个 `clobber`（接受一个 `Base&`）将是合法的。该函数不是 `Base` 的友元，但是它仍然能够改变一个 `Base` 对象的内容。如果按照这样的思路，则我们只要定义一个形如 `Sneaky` 的新类就能非常简单地规避掉 `protected` 提供的访问保护了。

要想阻止以上的用法，我们就要做出如下规定，即派生类的成员和友元只能访问派生类对象中的基类部分的受保护成员；对于普通的基类对象中的成员不具有特殊的访问权限。

公有、私有和受保护继承

某个类对其继承而来的成员的访问权限受到两个因素影响：一是在基类中该成员的访问说明符，二是在派生类的派生列表中的访问说明符。举个例子，考虑如下的继承关系：

```
class Base {
public:
    void pub_mem(); // public 成员
protected:
    int prot_mem; // protected 成员
private:
    char priv_mem; // private 成员
};

struct Pub_Derv : public Base {
    // 正确：派生类能访问 protected 成员
    int f() { return prot_mem; }
    // 错误：private 成员对于派生类来说是不可访问的
    char g() { return priv_mem; }
};

struct Priv_Derv : private Base {
    // private 不影响派生类的访问权限
    int f1() const { return prot_mem; }
};
```

派生访问说明符对于派生类的成员（及友元）能否访问其直接基类的成员没什么影响。对 613 基类成员的访问权限只与基类中的访问说明符有关。Pub_Derv 和 Priv_Derv 都能访问受保护的成员 prot_mem，同时它们都不能访问私有成员 priv_mem。

派生访问说明符的目的是控制派生类用户（包括派生类的派生类在内）对于基类成员的访问权限：

```
Pub_Derv d1;           // 继承自 Base 的成员是 public 的
Priv_Derv d2;          // 继承自 Base 的成员是 private 的
d1.pub_mem();          // 正确：pub_mem 在派生类中是 public 的
d2.pub_mem();          // 错误：pub_mem 在派生类中是 private 的
```

Pub_Derv 和 Priv_Derv 都继承了 pub_mem 函数。如果继承是公有的，则成员将遵循其原有的访问说明符，此时 d1 可以调用 pub_mem。在 Priv_Derv 中，Base 的成员是私有的，因此类的用户不能调用 pub_mem。

派生访问说明符还可以控制继承自派生类的新类的访问权限：

```
struct Derived_from_Public : public Pub_Derv {
    // 正确：Base::prot_mem 在 Pub_Derv 中仍然是 protected 的
    int use_base() { return prot_mem; }
};

struct Derived_from_Private : public Priv_Derv {
    // 错误：Base::prot_mem 在 Priv_Derv 中是 private 的
    int use_base() { return prot_mem; }
};
```

Pub_Derv 的派生类之所以能访问 Base 的 prot_mem 成员是因为该成员在 Pub_Derv 中仍然是受保护的。相反，Priv_Derv 的派生类无法执行类的访问，对于它们来说，Priv_Derv 继承自 Base 的所有成员都是私有的。

假设我们之前还定义了一个名为 Prot_Derv 的类，它采用受保护继承，则 Base 的所有公有成员在新定义的类中都是受保护的。Prot_Derv 的用户不能访问 pub_mem，但是 Prot_Derv 的成员和友元可以访问那些继承而来的成员。



派生类向基类转换的可访问性

派生类向基类的转换（参见 15.2.2 节，第 530 页）是否可访问由使用该转换的代码决定，同时派生类的派生访问说明符也会有影响。假定 D 继承自 B：

- 只有当 D 公有地继承 B 时，用户代码才能使用派生类向基类的转换；如果 D 继承 B 的方式是受保护的或者私有的，则用户代码不能使用该转换。
- 不论 D 以什么方式继承 B，D 的成员函数和友元都能使用派生类向基类的转换；派生类向其直接基类的类型转换对于派生类的成员和友元来说永远是可访问的。
- 如果 D 继承 B 的方式是公有的或者受保护的，则 D 的派生类的成员和友元可以使用 D 向 B 的类型转换；反之，如果 D 继承 B 的方式是私有的，则不能使用。



对于代码中的某个给定节点来说，如果基类的公有成员是可访问的，则派生类向基类的类型转换也是可访问的；反之则不行。

关键概念：类的设计与受保护的成员

不考虑继承的话，我们可以认为一个类有两种不同的用户：普通用户和类的实现者。

其中，普通用户编写的代码使用类的对象，这部分代码只能访问类的公有（接口）成员；实现者则负责编写类的成员和友元的代码，成员和友元既能访问类的公有部分，也能访问类的私有（实现）部分。

如果进一步考虑继承的话就会出现第三种用户，即派生类。基类把它希望派生类能够使用的部分声明成受保护的。普通用户不能访问受保护的成员，而派生类及其友元仍旧不能访问私有成员。

和其他类一样，基类应该将其接口成员声明为公有的；同时将属于其实现的部分分成两组：一组可供派生类访问，另一组只能由基类及基类的友元访问。对于前者应该声明为受保护的，这样派生类就能在实现自己的功能时使用基类的这些操作和数据；对于后者应该声明为私有的。

友元与继承

就像友元关系不能传递一样（参见 7.3.4 节，第 250 页），友元关系同样也不能继承。基类的友元在访问派生类成员时不具有特殊性，类似的，派生类的友元也不能随意访问基类的成员：

```
class Base {
    // 添加 friend 声明，其他成员与之前的版本一致
    friend class Pal;           // Pal 在访问 Base 的派生类时不具有特殊性
};

class Pal {
public:
    int f(Base b) { return b.prot_mem; } // 正确：Pal 是 Base 的友元
    int f2(Sneaky s) { return s.j; }     // 错误：Pal 不是 Sneaky 的友元
    // 对基类的访问权限由基类本身控制，即使对于派生类的基类部分也是如此
    int f3(Sneaky s) { return s.prot_mem; } // 正确：Pal 是 Base 的友元
};
```

如前所述，每个类负责控制自己的成员的访问权限，因此尽管看起来有点儿奇怪，但 f3 确实是正确的。Pal 是 Base 的友元，所以 Pal 能够访问 Base 对象的成员，这种可访问性包括了 Base 对象内嵌在其派生类对象中的情况。◀615

当一个类将另一个类声明为友元时，这种友元关系只对做出声明的类有效。对于原来那个类来说，其友元的基类或者派生类不具有特殊的访问能力：

```
// D2 对 Base 的 protected 和 private 成员不具有特殊的访问能力
class D2 : public Pal {
public:
    int mem(Base b)
        { return b.prot_mem; }           // 错误：友元关系不能继承
};
```



不能继承友元关系；每个类负责控制各自成员的访问权限。

改变个别成员的可访问性

有时我们需要改变派生类继承的某个名字的访问级别，通过使用 using 声明（参见 3.1 节，第 74 页）可以达到这一目的：

```
class Base {
public:
```

```

        std::size_t size() const { return n; }
protected:
    std::size_t n;
};
class Derived : private Base { // 注意: private 继承
public:
    // 保持对象尺寸相关的成员的访问级别
    using Base::size;
protected:
    using Base::n;
};

```

因为 `Derived` 使用了私有继承，所以继承而来的成员 `size` 和 `n`（在默认情况下）是 `Derived` 的私有成员。然而，我们使用 `using` 声明语句改变了这些成员的可访问性。改变之后，`Derived` 的用户将可以使用 `size` 成员，而 `Derived` 的派生类将能使用 `n`。

通过在类的内部使用 `using` 声明语句，我们可以将该类的直接或间接基类中的任何可访问成员（例如，非私有成员）标记出来。`using` 声明语句中名字的访问权限由该 `using` 声明语句之前的访问说明符来决定。也就是说，如果一条 `using` 声明语句出现在类的 `private` 部分，则该名字只能被类的成员和友元访问；如果 `using` 声明语句位于 `public` 部分，则类的所有用户都能访问它；如果 `using` 声明语句位于 `protected` 部分，则该名字对于成员、友元和派生类是可访问的。



派生类只能为那些它可以访问的名字提供 `using` 声明。

616 >

默认的继承保护级别

在 7.2 节（第 240 页）中我们曾经介绍过使用 `struct` 和 `class` 关键字定义的类具有不同的默认访问说明符。类似的，默认派生运算符也由定义派生类所用的关键字来决定。默认情况下，使用 `class` 关键字定义的派生类是私有继承的；而使用 `struct` 关键字定义的派生类是公有继承的：

```

class Base { /* ... */ };
struct D1 : Base { /* ... */ }; // 默认 public 继承
class D2 : Base { /* ... */ }; // 默认 private 继承

```

人们常常有一种错觉，认为在使用 `struct` 关键字和 `class` 关键字定义的类之间还有更深层次的差别。事实上，唯一的差别就是默认成员访问说明符及默认派生访问说明符；除此之外，再无其他不同之处。



一个私有派生的类最好显式地将 `private` 声明出来，而不要仅仅依赖于默认的设置。显式声明的好处是可以令私有继承关系清晰明了，不至于产生误会。

15.5 节练习

练习 15.18：假设给定了第 543 页和第 544 页的类，同时已知每个对象的类型如注释所示，判断下面的哪些赋值语句是合法的。解释那些不合法的语句为什么不允许：

<code>Base *p = &d1;</code>	<code>// d1 的类型是 Pub_Derv</code>
<code>p = &d2;</code>	<code>// d2 的类型是 Priv_Derv</code>

```

p = &d3;           // d3 的类型是 Prot_Derv
p = &dd1;          // dd1 的类型是 Derived_from_Public
p = &dd2;          // dd2 的类型是 Derived_from_Private
p = &dd3;          // dd3 的类型是 Derived_from_Protected

```

练习 15.19: 假设 543 页和 544 页的每个类都有如下形式的成员函数：

```
void memfcn(Base &b) { b = *this; }
```

对于每个类，分别判断上面的函数是否合法。

练习 15.20: 编写代码检验你对前面两题的回答是否正确。

练习 15.21: 从下面这些一般性抽象概念中任选一个（或者选一个你自己的），将其对应的一组类型组织成一个继承体系：

- (a) 图形文件格式（如 gif、tiff、jpeg、bmp）
- (b) 图形基元（如方格、圆、球、圆锥）
- (c) C++语言中的类型（如类、函数、成员函数）

练习 15.22: 对于你在上一题中选择的类，为其添加合适的虚函数及公有成员和受保护的成员。

15.6 继承中的类作用域



每个类定义自己的作用域（参见 7.4 节，第 253 页），在这个作用域内我们定义类的成员。当存在继承关系时，派生类的作用域嵌套（参见 2.2.4 节，第 43 页）在其基类的作用域之内。如果一个名字在派生类的作用域内无法正确解析，则编译器将继续在外层的基类作用域中寻找该名字的定义。

< 617

派生类的作用域位于基类作用域之内这一事实可能有点儿出人意料，毕竟在我们的程序文本中派生类和基类的定义是相互分离开来的。不过也恰恰因为类作用域有这种继承嵌套的关系，所以派生类才能像使用自己的成员一样使用基类的成员。例如，当我们编写下面的代码时：

```
Bulk_quote bulk;
cout << bulk.isbn();
```

名字 isbn 的解析将按照下述过程所示：

- 因为我们是通过 Bulk_quote 的对象调用 isbn 的，所以首先在 Bulk_quote 中查找，这一步没有找到名字 isbn。
- 因为 Bulk_quote 是 Disc_quote 的派生类，所以接下来在 Disc_quote 中查找，仍然找不到。
- 因为 Disc_quote 是 Quote 的派生类，所以接着查找 Quote；此时找到了名字 isbn，所以我们使用的 isbn 最终被解析为 Quote 中的 isbn。

在编译时进行名字查找

一个对象、引用或指针的静态类型（参见 15.2.3 节，第 532 页）决定了该对象的哪些成员是可见的。即使静态类型与动态类型可能不一致（当使用基类的引用或指针时会发生

这种情况), 但是我们能使用哪些成员仍然是由静态类型决定的。举个例子, 我们可以给 Disc_quote 添加一个新成员, 该成员返回一个存有最小(或最大)数量及折扣价格的 pair (参见 11.2.3 节, 第 379 页):

```
class Disc_quote : public Quote {
public:
    std::pair<size_t, double> discount_policy() const
    { return {quantity, discount}; }
    // 其他成员与之前的版本一致
};
```

我们只能通过 Disc_quote 及其派生类的对象、引用或指针使用 discount_policy:

```
Bulk_quote bulk;
Bulk_quote *bulkP = &bulk;           // 静态类型与动态类型一致
Quote *itemP = &bulk;               // 静态类型与动态类型不一致
bulkP->discount_policy();         // 正确: bulkP 的类型是 Bulk_quote*
itemP->discount_policy();         // 错误: itemP 的类型是 Quote*
```

618 尽管在 bulk 中确实含有一个名为 discount_policy 的成员, 但是该成员对于 itemP 却是不可见的。itemP 的类型是 Quote 的指针, 意味着对 discount_policy 的搜索将从 Quote 开始。显然 Quote 不包含名为 discount_policy 的成员, 所以我们无法通过 Quote 的对象、引用或指针调用 discount_policy。

名字冲突与继承

和其他作用域一样, 派生类也能重用定义在其直接基类或间接基类中的名字, 此时定义在内层作用域(即派生类)的名字将隐藏定义在外层作用域(即基类)的名字(参见 2.2.4 节, 第 43 页):

```
struct Base {
    Base(): mem(0) { }
protected:
    int mem;
};

struct Derived : Base {
    Derived(int i): mem(i) { }           // 用 i 初始化 Derived::mem
                                         // Base::mem 进行默认初始化
    int get_mem() { return mem; }        // 返回 Derived::mem
protected:
    int mem;                           // 隐藏基类中的 mem
};
```

get_mem 中 mem 引用的解析结果是定义在 Derived 中的名字, 下面的代码

```
Derived d(42);
cout << d.get_mem() << endl;          // 打印 42
```

的输出结果将是 42。



派生类的成员将隐藏同名的基类成员。

通过作用域运算符来使用隐藏的成员

我们可以通过作用域运算符来使用一个被隐藏的基类成员:

```
struct Derived : Base {
    int get_base_mem() { return Base::mem; }
    // ...
};
```

作用域运算符将覆盖掉原有的查找规则，并指示编译器从 `Base` 类的作用域开始查找 `mem`。如果使用最新的 `Derived` 版本运行上面的代码，则 `d.get_mem()` 的输出结果将是 0。

Best Practices

除了覆盖继承而来的虚函数之外，派生类最好不要重用其他定义在基类中的名字。

关键概念：名字查找与继承

619

理解函数调用的解析过程对于理解 C++ 的继承至关重要，假定我们调用 `p->mem()`（或者 `obj.mem()`），则依次执行以下 4 个步骤：

- 首先确定 `p`（或 `obj`）的静态类型。因为我们调用的是一个成员，所以该类型必然是类类型。
- 在 `p`（或 `obj`）的静态类型对应的类中查找 `mem`。如果找不到，则依次在直接基类中不断查找直至到达继承链的顶端。如果找遍了该类及其基类仍然找不到，则编译器将报错。
- 一旦找到了 `mem`，就进行常规的类型检查（参见 6.1 节，第 183 页）以确认对于当前找到的 `mem`，本次调用是否合法。
- 假设调用合法，则编译器将根据调用的是否是虚函数而产生不同的代码：
 - 如果 `mem` 是虚函数且我们是通过引用或指针进行的调用，则编译器产生的代码将在运行时确定到底运行该虚函数的哪个版本，依据是对象的动态类型。
 - 反之，如果 `mem` 不是虚函数或者我们是通过对象（而非引用或指针）进行的调用，则编译器将产生一个常规函数调用。

一如既往，名字查找先于类型检查

如前所述，声明在内层作用域的函数并不会重载声明在外层作用域的函数（参见 6.4.1 节，第 210 页）。因此，定义派生类中的函数也不会重载其基类中的成员。和其他作用域一样，如果派生类（即内层作用域）的成员与基类（即外层作用域）的某个成员同名，则派生类将在其作用域内隐藏该基类成员。即使派生类成员和基类成员的形参列表不一致，基类成员也仍然会被隐藏掉：

```
struct Base {
    int memfcn();
};

struct Derived : Base {
    int memfcn(int);           // 隐藏基类的 memfcn
};

Derived d; Base b;
b.memfcn();                  // 调用 Base::memfcn
d.memfcn(10);                // 调用 Derived::memfcn
d.memfcn();                  // 错误：参数列表为空的 memfcn 被隐藏了
d.Base::memfcn();            // 正确：调用 Base::memfcn
```

Derived 中的 memfcn 声明隐藏了 Base 中的 memfcn 声明。在上面的代码中前两条调用语句容易理解，第一个通过 Base 对象 b 进行的调用执行基类的版本；类似的，第二个通过 d 进行的调用执行 Derived 的版本；第三条调用语句有点特殊，d.memfcn() 是非法的。

为了解析这条调用语句，编译器首先在 Derived 中查找名字 memfcn；因为 Derived 确实定义了一个名为 memfcn 的成员，所以查找过程终止。一旦名字找到，编译器就不再继续查找了。Derived 中的 memfcn 版本需要一个 int 实参，而当前的调用语句无法提供任何实参，所以该调用语句是错误的。

虚函数与作用域

我们现在可以理解为什么基类与派生类中的虚函数必须有相同的形参列表了（参见 15.3 节，第 537 页）。假如基类与派生类的虚函数接受的实参不同，则我们就无法通过基类的引用或指针调用派生类的虚函数了。例如：

```
class Base {
public:
    virtual int fcn();
};

class D1 : public Base {
public:
    // 隐藏基类的 fcn，这个 fcn 不是虚函数
    // D1 继承了 Base::fcn() 的定义
    int fcn(int);           // 形参列表与 Base 中的 fcn 不一致
    virtual void f2();       // 是一个新的虚函数，在 Base 中不存在
};

class D2 : public D1 {
public:
    int fcn(int);           // 是一个非虚函数，隐藏了 D1::fcn(int)
    int fcn();               // 覆盖了 Base 的虚函数 fcn
    void f2();               // 覆盖了 D1 的虚函数 f2
};
```

D1 的 fcn 函数并没有覆盖 Base 的虚函数 fcn，原因是它们的形参列表不同。实际上，D1 的 fcn 将隐藏 Base 的 fcn。此时拥有了两个名为 fcn 的函数：一个是 D1 从 Base 继承而来的虚函数 fcn；另一个是 D1 自己定义的接受一个 int 参数的非虚函数 fcn。

通过基类调用隐藏的虚函数

给定上面定义的这些类后，我们来看几种使用其函数的方法：

```
Base bobj; D1 d1obj; D2 d2obj;

Base *bp1 = &bobj, *bp2 = &d1obj, *bp3 = &d2obj;
bp1->fcn();           // 虚调用，将在运行时调用 Base::fcn
bp2->fcn();           // 虚调用，将在运行时调用 Base::fcn
bp3->fcn();           // 虚调用，将在运行时调用 D2::fcn

D1 *d1p = &d1obj; D2 *d2p = &d2obj;
bp2->f2();             // 错误：Base 没有名为 f2 的成员
d1p->f2();             // 虚调用，将在运行时调用 D1::f2()
d2p->f2();             // 虚调用，将在运行时调用 D2::f2()
```

前三条调用语句是通过基类的指针进行的，因为 `fcn` 是虚函数，所以编译器产生的代码将在运行时确定使用虚函数的那个版本。判断的依据是该指针所绑定对象的真实类型。在 `bp2` 的例子中，实际绑定的对象是 `D1` 类型，而 `D1` 并没有覆盖那个不接受实参的 `fcn`，所以通过 `bp2` 进行的调用将在运行时解析为 `Base` 定义的版本。

接下来的三条调用语句是通过不同类型的指针进行的，每个指针分别指向继承体系中的一个类型。因为 `Base` 类中没有 `fcn()`，所以第一条语句是非法的，即使当前的指针碰巧指向了一个派生类对象也无济于事。

为了完整地阐明上述问题，我们不妨再观察一些对于非虚函数 `fcn(int)` 的调用语句：

```
Base *p1 = &d2obj; D1 *p2 = &d2obj; D2 *p3 = &d2obj;
p1->fcn(42);           // 错误: Base 中没有接受一个 int 的 fcn
p2->fcn(42);           // 静态绑定, 调用 D1::fcn(int)
p3->fcn(42);           // 静态绑定, 调用 D2::fcn(int)
```

在上面的每条调用语句中，指针都指向了 `D2` 类型的对象，但是由于我们调用的是非虚函数，所以不会发生动态绑定。实际调用的函数版本由指针的静态类型决定。

覆盖重载的函数

和其他函数一样，成员函数无论是否是虚函数都能被重载。派生类可以覆盖重载函数的 0 个或多个实例。如果派生类希望所有的重载版本对于它来说都是可见的，那么它就需要覆盖所有的版本，或者一个也不覆盖。

有时一个类仅需覆盖重载集合中的一些而非全部函数，此时，如果我们不得不覆盖基类中的每一个版本的话，显然操作将极其烦琐。

一种好的解决方案是为重载的成员提供一条 `using` 声明语句（参见 15.5 节，第 546 页），这样我们就无须覆盖基类中的每一个重载版本了。`using` 声明语句指定一个名字而不指定形参列表，所以一条基类成员函数的 `using` 声明语句就可以把该函数的所有重载实例添加到派生类作用域中。此时，派生类只需要定义其特有的函数就可以了，而无须为继承而来的其他函数重新定义。

类内 `using` 声明的一般规则同样适用于重载函数的名字（参见 15.5 节，第 546 页）；基类函数的每个实例在派生类中都必须是可访问的。对派生类没有重新定义的重载版本的访问实际上是对 `using` 声明点的访问。

15.6 节练习

练习 15.23: 假设第 550 页的 `D1` 类需要覆盖它继承而来的 `fcn` 函数，你应该如何对其进行修改？如果你修改之后 `fcn` 匹配了 `Base` 中的定义，则该节的那些调用语句将如何解析？

15.7 构造函数与拷贝控制

和其他类一样，位于继承体系中的类也需要控制当其对象执行一系列操作时发生什么样的行为，这些操作包括创建、拷贝、移动、赋值和销毁。如果一个类（基类或派生类）没有定义拷贝控制操作，则编译器将为它合成一个版本。当然，这个合成的版本也可以定义成被删除的函数。



15.7.1 虚析构函数

继承关系对基类拷贝控制最直接的影响是基类通常应该定义一个虚析构函数（参见 15.2.1 节，第 528 页），这样我们就能动态分配继承体系中的对象了。

如前所述，当我们 `delete` 一个动态分配的对象的指针时将执行析构函数（参见 13.1.3 节，第 445 页）。如果该指针指向继承体系中的某个类型，则有可能出现指针的静态类型与被删除对象的动态类型不符的情况（参见 15.2.2 节，第 530 页）。例如，如果我们 `delete` 一个 `Quote*` 类型的指针，则该指针有可能实际指向了一个 `Bulk_quote` 类型的对象。如果这样的话，编译器就必须清楚它应该执行的是 `Bulk_quote` 的析构函数。和其他函数一样，我们通过在基类中将析构函数定义成虚函数以确保执行正确的析构函数版本：

```
class Quote {
public:
    // 如果我们删除的是一个指向派生类对象的基类指针，则需要虚析构函数
    virtual ~Quote() = default;           // 动态绑定析构函数
};
```

和其他虚函数一样，析构函数的虚属性也会被继承。因此，无论 `Quote` 的派生类使用合成的析构函数还是定义自己的析构函数，都将是虚析构函数。只要基类的析构函数是虚函数，就能确保当我们 `delete` 基类指针时将运行正确的析构函数版本：

```
Quote *itemP = new Quote;           // 静态类型与动态类型一致
delete itemP;                      // 调用 Quote 的析构函数
itemP = new Bulk_quote;             // 静态类型与动态类型不一致
delete itemP;                      // 调用 Bulk_quote 的析构函数
```



WARNING 如果基类的析构函数不是虚函数，则 `delete` 一个指向派生类对象的基类指针将产生未定义的行为。

之前我们曾介绍过一条经验准则，即如果一个类需要析构函数，那么它也同样需要拷贝和赋值操作（参见 13.1.4 节，第 447 页）。基类的析构函数并不遵循上述准则，它是一个重要的例外。一个基类总是需要析构函数，而且它能将析构函数设定为虚函数。此时，该析构函数为了成为虚函数而令内容为空，我们显然无法由此推断该基类还需要赋值运算符或拷贝构造函数。

623 虚析构函数将阻止合成移动操作

基类需要一个虚析构函数这一事实还会对基类和派生类的定义产生另外一个间接的影响：如果一个类定义了析构函数，即使它通过`=default` 的形式使用了合成的版本，编译器也不会为这个类合成移动操作（参见 13.6.2 节，第 475 页）。

15.7.1 节练习

练习 15.24：哪种类需要虚析构函数？虚析构函数必须执行什么样的操作？



15.7.2 合成拷贝控制与继承

基类或派生类的合成拷贝控制成员的行为与其他合成的构造函数、赋值运算符或析构函数类似：它们对类本身的成员依次进行初始化、赋值或销毁的操作。此外，这些合成的成员还负责使用直接基类中对应的操作对一个对象的直接基类部分进行初始化、赋值或销

毁的操作。例如，

- 合成的 Bulk_quote 默认构造函数运行 Disc_quote 的默认构造函数，后者又运行 Quote 的默认构造函数。
- Quote 的默认构造函数将 bookNo 成员默认初始化为空字符串，同时使用类内初始值将 price 初始化为 0。
- Quote 的构造函数完成后，继续执行 Disc_quote 的构造函数，它使用类内初始值初始化 qty 和 discount。
- Disc_quote 的构造函数完成后，继续执行 Bulk_quote 的构造函数，但是它什么具体工作也不做。

类似的，合成的 Bulk_quote 拷贝构造函数使用（合成的） Disc_quote 拷贝构造函数，后者又使用（合成的） Quote 拷贝构造函数。其中， Quote 拷贝构造函数拷贝 bookNo 和 price 成员； Disc_quote 拷贝构造函数拷贝 qty 和 discount 成员。

值得注意的是，无论基类成员是合成的版本（如 Quote 继承体系的例子）还是自定义的版本都没有太大影响。唯一的要求是相应的成员应该可访问（参见 15.5 节，第 542 页）并且不是一个被删除的函数。

在我们的 Quote 继承体系中，所有类都使用合成的析构函数。其中，派生类隐式地使用而基类通过将其虚析构函数定义成`=default`而显式地使用。一如既往，合成的析构函数体是空的，其隐式的析构部分负责销毁类的成员（参见 13.1.3 节，第 444 页）。对于派生类的析构函数来说，它除了销毁派生类自己的成员外，还负责销毁派生类的直接基类；该直接基类又销毁它自己的直接基类，以此类推直至继承链的顶端。

如前所述，Quote 因为定义了析构函数而不能拥有合成的移动操作，因此当我们移动 Quote 对象时实际使用的是合成的拷贝操作（参见 13.6.2 节，第 477 页）。如我们即将看到的那样，Quote 没有移动操作意味着它的派生类也没有。

派生类中删除的拷贝控制与基类的关系

就像其他任何类的情况一样，基类或派生类也能出于同样的原因将其合成的默认构造函数或者任何一个拷贝控制成员定义成被删除的函数（参见 13.1.6 节，第 450 页和 13.6.2 节，第 475 页）。此外，某些定义基类的方式也可能导致有的派生类成员成为被删除的函数：

- 如果基类中的默认构造函数、拷贝构造函数、拷贝赋值运算符或析构函数是被删除的函数或者不可访问（参见 15.5 节，第 543 页），则派生类中对应的成员将是被删除的，原因是编译器不能使用基类成员来执行派生类对象基类部分的构造、赋值或销毁操作。
- 如果在基类中有一个不可访问或删除掉的析构函数，则派生类中合成的默认和拷贝构造函数将是被删除的，因为编译器无法销毁派生类对象的基类部分。
- 和过去一样，编译器将不会合成一个删除掉的移动操作。当我们使用`=default`请求一个移动操作时，如果基类中的对应操作是删除的或不可访问的，那么派生类中该函数将是被删除的，原因是派生类对象的基类部分不可移动。同样，如果基类的析构函数是删除的或不可访问的，则派生类的移动构造函数也将是被删除的。

举个例子，对于下面的基类 B 来说：

```
class B {  
public:  
    B();
```

C++
11

624

C++
11

```

B(const B&) = delete;
// 其他成员，不含有移动构造函数
};

class D : public B {
    // 没有声明任何构造函数
};

D d;                      // 正确：D 的合成默认构造函数使用 B 的默认构造函数
D d2(d);                  // 错误：D 的合成拷贝构造函数是被删除的
D d3(std::move(d));       // 错误：隐式地使用 D 的被删除的拷贝构造函数

```

基类 B 含有一个可访问的默认构造函数和一个显式删除的拷贝构造函数。因为我们定义了拷贝构造函数，所以编译器将不会为 B 合成一个移动构造函数(参见 13.6.2 节，第 475 页)。因此，我们既不能移动也不能拷贝 B 的对象。如果 B 的派生类希望它自己的对象能被移动和拷贝，则派生类需要自定义相应版本的构造函数。当然，在这一过程中派生类还必须考虑如何移动或拷贝其基类部分的成员。在实际编程过程中，如果在基类中没有默认、拷贝或移动构造函数，则一般情况下派生类也不会定义相应的操作。

625 移动操作与继承

如前所述，大多数基类都会定义一个虚析构函数。因此在默认情况下，基类通常不含有合成的移动操作，而且在它的派生类中也没有合成的移动操作。

因为基类缺少移动操作会阻止派生类拥有自己的合成移动操作，所以当我们确实需要执行移动操作时应该首先在基类中进行定义。我们的 `Quote` 可以使用合成的版本，不过前提是 `Quote` 必须显式地定义这些成员。一旦 `Quote` 定义了自己的移动操作，那么它必须同时显式地定义拷贝操作(参见 13.6.2 节，第 476 页)：

```

class Quote {
public:
    Quote() = default;                                // 对成员依次进行默认初始化
    Quote(const Quote&) = default;                   // 对成员依次拷贝
    Quote(Quote&&) = default;                        // 对成员依次拷贝
    Quote& operator=(const Quote&) = default;        // 拷贝赋值
    Quote& operator=(Quote&&) = default;            // 移动赋值
    virtual ~Quote() = default;
    // 其他成员与之前的版本一致
};

```

通过上面的定义，我们就能对 `Quote` 的对象逐成员地分别进行拷贝、移动、赋值和销毁操作了。而且除非 `Quote` 的派生类中含有排斥移动的成员，否则它将自动获得合成的移动操作。

15.7.2 节练习

练习 15.25: 我们为什么为 `Disc_quote` 定义一个默认构造函数？如果去除掉该构造函数的话会对 `Bulk_quote` 的行为产生什么影响？



15.7.3 派生类的拷贝控制成员

如我们在 15.2.2 节(第 531 页)介绍过的，派生类构造函数在其初始化阶段中不但要初始化派生类自己的成员，还负责初始化派生类对象的基类部分。因此，派生类的拷贝和

移动构造函数在拷贝和移动自有成员的同时，也要拷贝和移动基类部分的成员。类似的，派生类赋值运算符也必须为其基类部分的成员赋值。

和构造函数及赋值运算符不同的是，析构函数只负责销毁派生类自己分配的资源。如前所述，对象的成员是被隐式销毁的（参见 13.1.3 节，第 445 页）；类似的，派生类对象的基类部分也是自动销毁的。



当派生类定义了拷贝或移动操作时，该操作负责拷贝或移动包括基类部分成员在内的整个对象。

<626

定义派生类的拷贝或移动构造函数



当为派生类定义拷贝或移动构造函数时（参见 13.1.1 节，第 440 页和 13.6.2 节，第 473 页），我们通常使用对应的基类构造函数初始化对象的基类部分：

```
class Base { /* ... */ };
class D: public Base {
public:
    // 默认情况下，基类的默认构造函数初始化对象的基类部分
    // 要想使用拷贝或移动构造函数，我们必须在构造函数初始值列表中
    // 显式地调用该构造函数
    D(const D& d): Base(d)           // 拷贝基类成员
        /* D 的成员的初始值 */ { /* ... */ }
    D(D&& d): Base(std::move(d))      // 移动基类成员
        /* D 的成员的初始值 */ { /* ... */ }
};
```

初始值 `Base(d)` 将一个 `D` 对象传递给基类构造函数。尽管从道理上来说，`Base` 可以包含一个参数类型为 `D` 的构造函数，但是在实际编程过程中通常不会这么做。相反，`Base(d)` 一般会匹配 `Base` 的拷贝构造函数。`D` 类型的对象 `d` 将被绑定到该构造函数的 `Base&` 形参上。`Base` 的拷贝构造函数负责将 `d` 的基类部分拷贝给要创建的对象。假如我们没有提供基类的初始值的话：

```
// D 的这个拷贝构造函数很可能是不正确的定义
// 基类部分被默认初始化，而非拷贝
D(const D& d) /* 成员初始值，但是没有提供基类初始值 */
{ /* ... */ }
```

在上面的例子中，`Base` 的默认构造函数将被用来初始化 `D` 对象的基类部分。假定 `D` 的构造函数从 `d` 中拷贝了派生类成员，则这个新构建的对象的配置将非常奇怪：它的 `Base` 成员被赋予了默认值，而 `D` 成员的值则是从其他对象拷贝得来的。



在默认情况下，基类默认构造函数初始化派生类对象的基类部分。如果我们想拷贝（或移动）基类部分，则必须在派生类的构造函数初始值列表中显式地使用基类的拷贝（或移动）构造函数。

<627

派生类赋值运算符

与拷贝和移动构造函数一样，派生类的赋值运算符（参见 13.1.2 节，第 443 页和 13.6.2 节，第 474 页）也必须显式地为其基类部分赋值：

```
// Base::operator=(const Base&) 不会被自动调用
```

<627

```
D &D::operator=(const D &rhs)
{
    Base::operator=(rhs); // 为基类部分赋值
    // 按照过去的方式为派生类的成员赋值
    // 酌情处理自赋值及释放已有资源等情况
    return *this;
}
```

上面的运算符首先显式地调用基类赋值运算符，令其为派生类对象的基类部分赋值。基类的运算符（应该可以）正确地处理自赋值的情况，如果赋值命令是正确的，则基类运算符将释放掉其左侧运算对象的基类部分的旧值，然后利用 `rhs` 为其赋一个新值。随后，我们继续进行其他为派生类成员赋值的工作。

值得注意的是，无论基类的构造函数或赋值运算符是自定义的版本还是合成的版本，派生类的对应操作都能使用它们。例如，对于 `Base::operator=` 的调用语句将执行 `Base` 的拷贝赋值运算符，至于该运算符是由 `Base` 显式定义的还是由编译器合成的无关紧要。

派生类析构函数

如前所述，在析构函数体执行完成后，对象的成员会被隐式销毁（参见 13.1.3 节，第 445 页）。类似的，对象的基类部分也是隐式销毁的。因此，和构造函数及赋值运算符不同的是，派生类析构函数只负责销毁由派生类自己分配的资源：

```
class D: public Base {
public:
    // Base::~Base 被自动调用执行
    ~D() { /* 该处由用户定义清除派生类成员的操作 */ }
};
```

对象销毁的顺序正好与其创建的顺序相反：派生类析构函数首先执行，然后是基类的析构函数，以此类推，沿着继承体系的反方向直至最后。

在构造函数和析构函数中调用虚函数

如我们所知，派生类对象的基类部分将首先被构建。当执行基类的构造函数时，该对象的派生类部分是未被初始化的状态。类似的，销毁派生类对象的次序正好相反，因此当执行基类的析构函数时，派生类部分已经被销毁掉了。由此可知，当我们执行上述基类成员的时候，该对象处于未完成的状态。

为了能够正确地处理这种未完成状态，编译器认为对象的类型在构造或析构的过程中仿佛发生了改变一样。也就是说，当我们构建一个对象时，需要把对象的类和构造函数的类看作是同一个；对虚函数的调用绑定正好符合这种把对象的类和构造函数的类看成同一个的要求；对于析构函数也是同样的道理。上述的绑定不但对直接调用虚函数有效，对间接调用也是有效的，这里的间接调用是指通过构造函数（或析构函数）调用另一个函数。
628>

为了理解上述行为，不妨考虑当基类构造函数调用虚函数的派生类版本时会发生什么情况。这个虚函数可能会访问派生类的成员，毕竟，如果它不需要访问派生类成员的话，则派生类直接使用基类的虚函数版本就可以了。然而，当执行基类构造函数时，它要用到的派生类成员尚未初始化，如果我们允许这样的访问，则程序很可能会崩溃。



如果构造函数或析构函数调用了某个虚函数，则我们应该执行与构造函数或析构函数所属类型相对应的虚函数版本。

15.7.3 节练习

练习 15.26: 定义 `Quote` 和 `Bulk_quote` 的拷贝控制成员，令其与合成的版本行为一致。为这些成员以及其他构造函数添加打印状态的语句，使得我们能够知道正在运行哪个程序。使用这些类编写程序，预测程序将创建和销毁哪些对象。重复实验，不断比较你的预测和实际输出结果是否相同，直到预测完全准确再结束。

15.7.4 继承的构造函数

在 C++11 新标准中，派生类能够重用其直接基类定义的构造函数。尽管如我们所知，这些构造函数并非以常规的方式继承而来，但是为了方便，我们不妨姑且称其为“继承”的。一个类只初始化它的直接基类，出于同样的原因，一个类也只继承其直接基类的构造函数。类不能继承默认、拷贝和移动构造函数。如果派生类没有直接定义这些构造函数，则编译器将为派生类合成它们。

派生类继承基类构造函数的方式是提供一条注明了（直接）基类名的 `using` 声明语句。举个例子，我们可以重新定义 `Bulk_quote` 类（参见 15.4 节，第 541 页），令其继承 `Disc_quote` 类的构造函数：

```
class Bulk_quote : public Disc_quote {
public:
    using Disc_quote::Disc_quote; // 继承 Disc_quote 的构造函数
    double net_price(std::size_t) const;
};
```

通常情况下，`using` 声明语句只是令某个名字在当前作用域内可见。而当作用于构造函数时，`using` 声明语句将令编译器产生代码。对于基类的每个构造函数，编译器都生成一个与之对应的派生类构造函数。换句话说，对于基类的每个构造函数，编译器都在派生类中生成一个形参列表完全相同的构造函数。

这些编译器生成的构造函数形如：

```
derived(parms) : base(args) { }
```

其中，`derived` 是派生类的名字，`base` 是基类的名字，`parms` 是构造函数的形参列表，`args` 将派生类构造函数的形参传递给基类的构造函数。在我们的 `Bulk_quote` 类中，继承的构造函数等价于：

```
Bulk_quote(const std::string& book, double price,
           std::size_t qty, double disc):
    Disc_quote(book, price, qty, disc) { }
```

如果派生类含有自己的数据成员，则这些成员将被默认初始化（参见 7.1.4 节，第 238 页）。

继承的构造函数的特点

和普通成员的 `using` 声明不一样，一个构造函数的 `using` 声明不会改变该构造函数的访问级别。例如，不管 `using` 声明出现在哪儿，基类的私有构造函数在派生类中还是一个私有构造函数；受保护的构造函数和公有构造函数也是同样的规则。

而且，一个 `using` 声明语句不能指定 `explicit` 或 `constexpr`。如果基类的构造函数是 `explicit`（参见 7.5.4 节，第 265 页）或者 `constexpr`（参见 7.5.6 节，第 267

C++
11

629

页), 则继承的构造函数也拥有相同的属性。

当一个基类构造函数含有默认实参(参见 6.5.1 节, 第 211 页)时, 这些实参并不会被继承。相反, 派生类将获得多个继承的构造函数, 其中每个构造函数分别省略掉一个含有默认实参的形参。例如, 如果基类有一个接受两个形参的构造函数, 其中第二个形参含有默认实参, 则派生类将获得两个构造函数: 一个构造函数接受两个形参(没有默认实参), 另一个构造函数只接受一个形参, 它对应于基类中最左侧的没有默认值的那个形参。

如果基类含有几个构造函数, 则除了两个例外情况, 大多数时候派生类会继承所有这些构造函数。第一个例外是派生类可以继承一部分构造函数, 而为其他构造函数定义自己的版本。如果派生类定义的构造函数与基类的构造函数具有相同的参数列表, 则该构造函数将不会被继承。定义在派生类中的构造函数将替换继承而来的构造函数。

第二个例外是默认、拷贝和移动构造函数不会被继承。这些构造函数按照正常规则被合成。继承的构造函数不会被作为用户定义的构造函数来使用, 因此, 如果一个类只含有继承的构造函数, 则它也将拥有一个合成的默认构造函数。

15.7.4 节练习

练习 15.27: 重新定义你的 Bulk_quote 类, 令其继承构造函数。



15.8 容器与继承

630 >

当我们使用容器存放继承体系中的对象时, 通常必须采取间接存储的方式。因为不允许在容器中保存不同类型的元素, 所以我们不能把具有继承关系的多种类型的对象直接存放在容器当中。

举个例子, 假定我们想定义一个 vector, 令其保存用户准备购买的几种书籍。显然我们不应该用 vector 保存 Bulk_quote 对象。因为我们不能将 Quote 对象转换成 Bulk_quote (参见 15.2.3 节, 第 534 页), 所以我们将无法把 Quote 对象放置在该 vector 中。

其实, 我们也不应该使用 vector 保存 Quote 对象。此时, 虽然我们可以把 Bulk_quote 对象放置在容器中, 但是这些对象再也不是 Bulk_quote 对象了:

```
vector<Quote> basket;
basket.push_back(Quote("0-201-82470-1", 50));
// 正确: 但是只能把对象的 Quote 部分拷贝给 basket
basket.push_back(Bulk_quote("0-201-54848-8", 50, 10, .25));
// 调用 Quote 定义的版本, 打印 750, 即 15 * $50
cout << basket.back().net_price(15) << endl;
```

basket 的元素是 Quote 对象, 因此当我们向该 vector 中添加一个 Bulk_quote 对象时, 它的派生类部分将被忽略掉 (参见 15.2.3 节, 第 535 页)。



当派生类对象被赋值给基类对象时, 其中的派生类部分将被“切掉”, 因此容器和存在继承关系的类型无法兼容。

在容器中放置（智能）指针而非对象

当我们希望在容器中存放具有继承关系的对象时，我们实际上存放的通常是基类的指针（更好的选择是智能指针（参见 12.1 节，第 400 页））。和往常一样，这些指针所指对象的动态类型可能是基类类型，也可能是派生类类型：

```
vector<shared_ptr<Quote>> basket;
basket.push_back(make_shared<Quote>("0-201-82470-1", 50));
basket.push_back(
    make_shared<Bulk_quote>("0-201-54848-8", 50, 10, .25));
// 调用 Quote 定义的版本；打印 562.5，即在 15*50 中扣除掉折扣金额
cout << basket.back()->net_price(15) << endl;
```

因为 `basket` 存放着 `shared_ptr`，所以我们必须解引用 `basket.back()` 的返回值以获得运行 `net_price` 的对象。我们通过在 `net_price` 的调用中使用 `->` 以达到这个目的。如我们所知，实际调用的 `net_price` 版本依赖于指针所指对象的动态类型。

值得注意的是，我们将 `basket` 定义成 `shared_ptr<Quote>`，但是在第二个 `push_back` 中传入的是一个 `Bulk_quote` 对象的 `shared_ptr`。正如我们可以将一个派生类的普通指针转换成基类指针一样（参见 15.2.2 节，第 530 页），我们也能把一个派生类的智能指针转换成基类的智能指针。在此例中，`make_shared<Bulk_quote>` 返回一个 `shared_ptr<Bulk_quote>` 对象，当我们调用 `push_back` 时该对象被转换成 `shared_ptr<Quote>`。因此尽管在形式上有所差别，但实际上 `basket` 的所有元素的类型都是相同的。

<631

15.8 节练习

练习 15.28： 定义一个存放 `Quote` 对象的 `vector`，将 `Bulk_quote` 对象传入其中。
计算 `vector` 中所有元素总的 `net_price`。

练习 15.29： 再运行一次你的程序，这次传入 `Quote` 对象的 `shared_ptr`。如果这次计算出的总额与之前的程序不一致，解释为什么；如果一致，也请说明原因。

15.8.1 编写 Basket 类



对于 C++ 面向对象的编程来说，一个悖论是我们无法直接使用对象进行面向对象编程。相反，我们必须使用指针和引用。因为指针会增加程序的复杂性，所以我们经常定义一些辅助的类来处理这种复杂情况。首先，我们定义一个表示购物篮的类：

```
class Basket {
public:
    // Basket 使用合成的默认构造函数和拷贝控制成员
    void add_item(const std::shared_ptr<Quote> &sale)
    { items.insert(sale); }
    // 打印每本书的总价和购物篮中所有书的总价
    double total_receipt(std::ostream&) const;
private:
    // 该函数用于比较 shared_ptr，multiset 成员会用到它
    static bool compare(const std::shared_ptr<Quote> &lhs,
                        const std::shared_ptr<Quote> &rhs)
    { return lhs->isbn() < rhs->isbn(); }
    // multiset 保存多个报价，按照 compare 成员排序
```

```
    std::multiset<std::shared_ptr<Quote>, decltype(compare)*>
        items{compare};
};
```

我们的类使用一个 `multiset` (参见 11.2.1 节, 第 377 页) 来存放交易信息, 这样我们就能保存同一本书的多条交易记录, 而且对于一本给定的书籍, 它的所有交易信息都保存在一起 (参见 11.2.2 节, 第 378 页)。

`multiset` 的元素是 `shared_ptr`。因为 `shared_ptr` 没有定义小于运算符, 所以为了对元素排序我们必须提供自己的比较运算符 (参见 11.2.2 节, 第 378 页)。在此例中, 我们定义了一个名为 `compare` 的私有静态成员, 该成员负责比较 `shared_ptr` 所指的对象的 `isbn`。我们初始化 `multiset`, 通过类内初始值调用比较函数 (参见 7.3.1 节, 第 246 页):

632 // multiset 保存多个报价, 按照 compare 成员排序

```
std::multiset<std::shared_ptr<Quote>, decltype(compare)*>
    items{compare};
```

这个声明看起来不太容易理解, 但是从左向右读的话, 我们就能明白它其实是定义了一个指向 `Quote` 对象的 `shared_ptr` 的 `multiset`。这个 `multiset` 将使用一个与 `compare` 成员类型相同的函数来对其中的元素进行排序。`multiset` 成员的名字是 `items`, 我们初始化 `items` 并令其使用我们的 `compare` 函数。

定义 Basket 的成员

`Basket` 类只定义两个操作。第一个成员是我们在类的内部定义的 `add_item` 成员, 该成员接受一个指向动态分配的 `Quote` 的 `shared_ptr`, 然后将这个 `shared_ptr` 放置在 `multiset` 中。第二个成员的名字是 `total_receipt`, 它负责将购物篮的内容逐项打印成清单, 然后返回购物篮中所有物品的总价格:

```
double Basket::total_receipt(ostream &os) const
{
    double sum = 0.0; // 保存实时计算出的总价格
    // iter 指向 ISBN 相同的一批元素中的第一个
    // upper_bound 返回一个迭代器, 该迭代器指向这批元素的尾后位置
    for (auto iter = items.cbegin();
        iter != items.cend();
        iter = items.upper_bound(*iter)) {
        // 我们知道在当前的 Basket 中至少有一个该关键字的元素
        // 打印该书籍对应的项目
        sum += print_total(os, **iter, items.count(*iter));
    }
    os << "Total Sale: " << sum << endl; // 打印最终的总价格
    return sum;
}
```

我们的 `for` 循环首先定义并初始化 `iter`, 令其指向 `multiset` 的第一个元素。条件部分检查 `iter` 是否等于 `items.cend()`: 如果相等, 表明我们已经处理完了所有购买记录, 接下来应该跳出 `for` 循环; 否则, 如果不相等, 则继续处理下一本书籍。

比较有趣的是, `for` 循环中的“递增”表达式。与通常的循环语句依次读取每个元素不同, 我们直接令 `iter` 指向下一个关键字, 调用 `upper_bound` 函数可以令我们跳过与当前关键字相同的所有元素 (参见 11.3.5 节, 第 390 页)。对于 `upper_bound` 函数来说, 它返回的是一个迭代器, 该迭代器指向所有与 `iter` 关键字相等的元素中最后一个元素的

下一位置。因此，我们得到的迭代器或者指向集合的末尾，或者指向下一本书籍。

在 `for` 循环内部，我们通过调用 `print_total`（参见 15.1 节，第 527 页）来打印购物篮中每本书籍的细节：

```
sum += print_total(os, **iter, items.count(*iter));
```

`print_total` 的实参包括一个用于写入数据的 `ostream`、一个待处理的 `Quote` 对象和一个计数值。当我们解引用 `iter` 后将得到一个指向准备打印的对象的 `shared_ptr`。为了得到这个对象，必须解引用该 `shared_ptr`。因此，`**iter` 是一个 `Quote` 对象（或者 `Quote` 的派生类的对象）。我们使用 `multiset` 的 `count` 成员（参见 11.3.5 节，第 388 页）来统计在 `multiset` 中有多少元素的键值相同（即 ISBN 相同）。 633

如我们所知，`print_total` 调用了虚函数 `net_price`，因此最终的计算结果依赖于 `**iter` 的动态类型。`print_total` 函数打印并返回给定书籍的总价格，我们把这个结果添加到 `sum` 当中，最后当循环结束后打印 `sum`。

隐藏指针

`Basket` 的用户仍然必须处理动态内存，原因是 `add_item` 需要接受一个 `shared_ptr` 参数。因此，用户不得不按照如下形式编写代码：

```
Basket bsk;
bsk.add_item(make_shared<Quote>("123", 45));
bsk.add_item(make_shared<Bulk_quote>("345", 45, 3, .15));
```

我们的下一步是重新定义 `add_item`，使得它接受一个 `Quote` 对象而非 `shared_ptr`。新版本的 `add_item` 将负责处理内存分配，这样它的用户就不必再受困于此了。我们将定义两个版本，一个拷贝它给定的对象，另一个则采取移动操作（参见 13.6.3 节，第 481 页）：

```
void add_item(const Quote& sale);           // 拷贝给定的对象
void add_item(Quote&& sale);                 // 移动给定的对象
```

唯一的问题是 `add_item` 不知道要分配的类型。当 `add_item` 进行内存分配时，它将拷贝（或移动）它的 `sale` 参数。在某处可能会有一条如下形式的 `new` 表达式：

```
new Quote(sale)
```

不幸的是，这条表达式所做的工作可能是不正确的：`new` 为我们请求的类型分配内存，因此这条表达式将分配一个 `Quote` 类型的对象并且拷贝 `sale` 的 `Quote` 部分。然而，`sale` 实际指向的可能是 `Bulk_quote` 对象，此时，该对象将被迫切掉一部分。

模拟虚拷贝



为了解决上述问题，我们给 `Quote` 类添加一个虚函数，该函数将申请一份当前对象的拷贝。

```
class Quote {
public:
    // 该虚函数返回当前对象的一份动态分配的拷贝
    // 这些成员使用的引用限定符参见 13.6.3 节（第 483 页）
    virtual Quote* clone() const & {return new Quote(*this);}
    virtual Quote* clone() &&
        {return new Quote(std::move(*this));}
    // 其他成员与之前的版本一致
};
```

634 >

```
class Bulk_quote : public Quote {
    Bulk_quote* clone() const & {return new Bulk_quote(*this);}
    Bulk_quote* clone() &&
        {return new Bulk_quote(std::move(*this));}
    // 其他成员与之前的版本一致
};
```

因为我们拥有 `add_item` 的拷贝和移动版本，所以我们分别定义 `clone` 的左值和右值版本(参见 13.6.3 节, 第 483 页)。每个 `clone` 函数分配当前类型的一个新对象，其中，`const` 左值引用成员将它自己拷贝给新分配的对象；右值引用成员则将自己移动到新数据中。

我们可以使用 `clone` 很容易地写出新版本的 `add_item`:

```
class Basket {
public:
    void add_item(const Quote& sale)      // 拷贝给定的对象
        { items.insert(std::shared_ptr<Quote>(sale.clone())); }
    void add_item(Quote&& sale)           // 移动给定的对象
        { items.insert(
            std::shared_ptr<Quote>(std::move(sale).clone())); }
    // 其他成员与之前的版本一致
};
```

和 `add_item` 本身一样，`clone` 函数也根据作用于左值还是右值而分为不同的重载版本。在此例中，第一个 `add_item` 函数调用 `clone` 的 `const` 左值版本，第二个函数调用 `clone` 的右值引用版本。在右值版本中，尽管 `sale` 的类型是右值引用类型，但实际上 `sale` 本身(和任何其他变量一样) 是个左值(参见 13.6.1 节, 第 471 页)。因此，我们调用 `move` 把一个右值引用绑定到 `sale` 上。

我们的 `clone` 函数也是一个虚函数。`sale` 的动态类型(通常)决定了到底运行 `Quote` 的函数还是 `Bulk_quote` 的函数。无论我们是拷贝还是移动数据，`clone` 都返回一个新分配对象的指针，该对象与 `clone` 所属的类型一致。我们把一个 `shared_ptr` 绑定到这个对象上，然后调用 `insert` 将这个新分配的对象添加到 `items` 中。注意，因为 `shared_ptr` 支持派生类向基类的类型转换(参见 15.2.2 节, 第 530 页)，所以我们将把 `shared_ptr<Quote>` 绑定到 `Bulk_quote*` 上。

15.8.1 节练习

练习 15.30: 编写你自己的 `Basket` 类，用它计算上一个练习中交易记录的总价格。

15.9 文本查询程序再探

接下来，我们扩展 12.3 节(第 430 页)的文本查询程序，用它作为说明继承的最后一个例子。在上一版的程序中，我们可以查询在文件中某个指定单词的出现情况。我们将在本节扩展该程序使其支持更多更复杂的查询操作。在后面的例子中，我们将针对下面这个小故事展开查询：

```
Alice Emma has long flowing red hair.
Her Daddy says when the wind blows
through her hair, it looks almost alive,
like a fiery bird in flight.
```

```

A beautiful fiery bird, he tells her,
magical but untamed.
"Daddy, shush, there is no such thing,"
she tells him, at the same time wanting
him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"

```

我们的系统将支持如下查询形式。

- 单词查询，用于得到匹配某个给定 string 的所有行：

```

Executing Query for: Daddy
Daddy occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 7) "Daddy, shush, there is no such thing,"
(line 10) Shyly, she asks, "I mean, Daddy, is there?"

```

- 逻辑非查询，使用~运算符得到不匹配查询条件的所有行：

```

Executing Query for: ~(Alice)
~(Alice) occurs 9 times
(line 2) Her Daddy says when the wind blows
(line 3) through her hair, it looks almost alive,
(line 4) like a fiery bird in flight.

...

```

- 逻辑或查询，使用 | 运算符返回匹配两个条件中任意一个的行：

```

Executing Query for: (hair | Alice)
(hair | Alice) occurs 2 times
(line 1) Alice Emma has long flowing red hair.
(line 3) through her hair, it looks almost alive,

```

- 逻辑与查询，使用 & 运算符返回匹配全部两个条件的行：

```

Executing query for: (hair & Alice)
(hair & Alice) occurs 1 time
(line 1) Alice Emma has long flowing red hair.

```

此外，我们还希望能够混合使用这些运算符，比如：

```
fiery & bird | wind
```

在类似这样的例子中，我们将使用 C++ 通用的优先级规则（参见 4.1.2 节，第 121 页）对复杂表达式求值。因此，这条查询语句所得行应该是如下二者之一：在该行中或者 `fiery` 和 `bird` 同时出现，或者出现了 `wind`：

```

Executing Query for: ((fiery & bird) | wind)
((fiery & bird) | wind) occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 4) like a fiery bird in flight.
(line 5) A beautiful fiery bird, he tells her,

```

636

在输出内容中首先是那条查询语句，我们使用圆括号来表示查询被解释和执行的次序。与之前实现的版本一样，接下来系统将按照查询结果中行号的升序显示结果并且每一行只显示一次。

15.9.1 面向对象的解决方案

我们可能会认为使用 12.3.2 节（第 432 页）的 `TextQuery` 类来表示单词查询，然后

从该类中派生出其他查询是一种可行的方案。

然而，这样的设计实际上存在缺陷。为了理解其中的原因，我们不妨考虑逻辑非查询。单词查询查找一个指定的单词，为了让逻辑非查询按照单词查询的方式执行，我们将不得不定义逻辑非查询所要查找的单词。但是在一般情况下，我们无法得到这样的单词。相反，一个逻辑非查询中含有一个结果值需要取反的查询语句（单词查询或任何其他查询）；类似的，一个逻辑与查询和一个逻辑或查询各包含两个结果值需要合并的查询语句。

由上述观察结果可知，我们应该将几种不同的查询建模成相互独立的类，这些类共享一个公共基类：

```
WordQuery      // Daddy
NotQuery       // ~Alice
OrQuery        // hair | Alice
AndQuery       // hair & Alice
```

这些类将只包含两个操作：

- eval，接受一个 `TextQuery` 对象并返回一个 `QueryResult`，`eval` 函数使用给定的 `TextQuery` 对象查找与之匹配的行。
- rep，返回基础查询的 `string` 表示形式，`eval` 函数使用 `rep` 创建一个表示匹配结果的 `QueryResult`，输出运算符使用 `rep` 打印查询表达式。

关键概念：继承与组合

继承体系的设计本身是一个非常复杂的问题，已经超出了本书的范围。然而，有一条设计准则非常重要也非常基础，每个程序员都应该熟悉它。

当我们令一个类公有地继承另一个类时，派生类应当反映与基类的“是一种 (Is A)”关系。在设计良好的类体系当中，公有派生类的对象应该可以用在任何需要基类对象的地方。

类型之间的另一种常见关系是“有一个 (Has A)”关系，具有这种关系的类暗含成员的意思。

在我们的书店示例中，基类表示的是按规定价格销售的书籍的报价。`Bulk_quote` “是一种” 报价结果，只不过它使用的价格策略不同。我们的书店类都“有一个” 价格成员和 `ISBN` 成员。

抽象基类

如我们所知，在这四种查询之间并不存在彼此的继承关系，从概念上来说它们互为兄弟。因为所有这些类都共享同一个接口，所以我们需要定义一个抽象基类（参见 15.4 节，第 541 页）来表示该接口。我们将所需的抽象基类命名为 `Query_base`，以此来表示它的角色是整个查询继承体系的根节点。

我们的 `Query_base` 类将把 `eval` 和 `rep` 定义成纯虚函数（参见 15.4 节，第 541 页），其他代表某种特定查询类型的类必须覆盖这两个函数。我们将从 `Query_base` 直接派生出 `WordQuery` 和 `NotQuery`。`AndQuery` 和 `OrQuery` 都具有系统中其他类所不具备的一个特殊属性：它们各自包含两个运算对象。为了对这种属性建模，我们定义另外一个名为 `BinaryQuery` 的抽象基类，该抽象基类用于表示含有两个运算对象的查询。`AndQuery` 和 `OrQuery` 继承自 `BinaryQuery`，而 `BinaryQuery` 继承自 `Query_base`。由这些分

析我们将得到如图 15.2 所示的类设计结果：

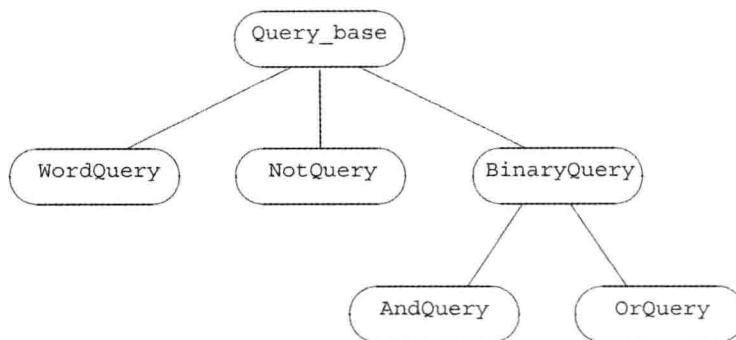


图 15.2: Query_base 继承体系

将层次关系隐藏于接口类中

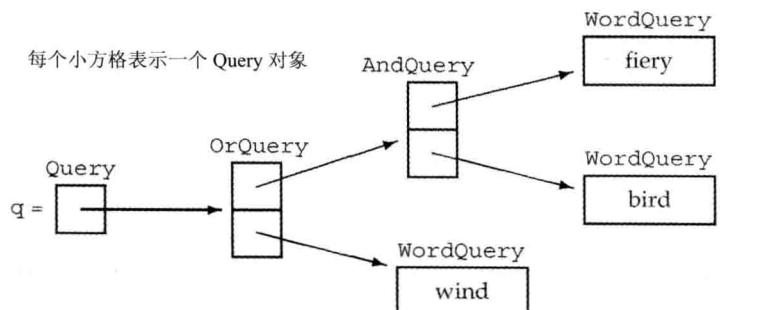
我们的程序将致力于计算查询结果，而非仅仅构建查询的体系。为了使程序能正常运行，我们必须首先创建查询命令，最简单的办法是编写 C++ 表达式。例如，可以编写下面的代码来生成之前描述的复合查询：

```
Query q = Query("fiery") & Query("bird") | Query("wind");
```

如上所述，其隐含的意思是用户层代码将不会直接使用这些继承的类；相反，我们将定义一个名为 `Query` 的接口类，由它负责隐藏整个继承体系。`Query` 类将保存一个 `Query_base` 指针，该指针绑定到 `Query_base` 的派生类对象上。`Query` 类与 `Query_base` 类提供的操作是相同的：`eval` 用于求查询的结果，`rep` 用于生成查询的 `string` 版本，同时 `Query` 也会定义一个重载的输出运算符用于显示查询。

用户将通过 `Query` 对象的操作间接地创建并处理 `Query_base` 对象。我们定义 `Query` 对象的三个重载运算符以及一个接受 `string` 参数的 `Query` 构造函数，这些函数动态分配一个新的 `Query_base` 派生类的对象：

- & 运算符生成一个绑定到新的 `AndQuery` 对象上的 `Query` 对象；
- | 运算符生成一个绑定到新的 `OrQuery` 对象上的 `Query` 对象；
- ~ 运算符生成一个绑定到新的 `NotQuery` 对象上的 `Query` 对象；
- 接受 `string` 参数的 `Query` 构造函数生成一个新的 `WordQuery` 对象。



```
Query q = Query("fiery") & Query("bird") | Query("wind");
```

图 15.3: 使用 Query 表达式创建的对象

理解这些类的工作机理

在这个应用程序中，很大一部分工作是构建代表用户查询的对象，对于读者来说认识到这一点非常重要。例如，像上面这样的表达式将生成如图 15.3 所示的一系列相关对象的集合。

一旦对象树构建完成后，对某一条查询语句的求值（或生成表示形式的）过程基本上就转换为沿着箭头方向依次对每个对象求值（或显示）的过程（由编译器为我们组织管理）。

639 例如，如果我们对 q（即树的根节点）调用 eval 函数，则该调用语句将令 q 所指的 OrQuery 对象 eval 它自己。对该 OrQuery 求值实际上是对它的两个运算对象执行 eval 操作：一个运算对象是 AndQuery，另一个是查找单词 wind 的 WordQuery。接下来，对 AndQuery 求值转化为对它的两个 WordQuery 求值，分别生成单词 fiery 和 bird 的查询结果。

对于面向对象编程的新手来说，要想理解一个程序，最困难的部分往往是理解程序的设计思路。一旦你掌握了程序的设计思路，接下来的实现也就水到渠成了。为了帮助读者理解程序设计的过程，我们在表 15.1 中整理了之前那个例子用到的类，并对其进行了简要的描述。

640

表 15.1：概述：Query 程序设计

Query 程序接口类和操作	
TextQuery	该类读入给定的文件并构建一个查找图。这个类包含一个 query 操作，它接受一个 string 实参，返回一个 QueryResult 对象；该 QueryResult 对象表示 string 出现的行（12.3.2 节，第 432 页）
QueryResult	该类保存一个 query 操作的结果（12.3.2 节，第 433 页）
Query	是一个接口类，指向 Query_base 派生类的对象
Query q(s)	将 Query 对象 q 绑定到一个存放着 string s 的新 WordQuery 对象上
q1 & q2	返回一个 Query 对象，该 Query 绑定到一个存放 q1 和 q2 的新 AndQuery 对象上
q1 q2	返回一个 Query 对象，该 Query 绑定到一个存放 q1 和 q2 的新 OrQuery 对象上
~q	返回一个 Query 对象，该 Query 绑定到一个存放 q 的新 NotQuery 对象上
Query 程序实现类	
Query_base	查询类的抽象基类
WordQuery	Query_base 的派生类，用于查找一个给定的单词
NotQuery	Query_base 的派生类，查询结果是 Query 运算对象没有出现的行的集合
BinaryQuery	Query_base 派生出来的另一个抽象基类，表示有两个运算对象的查询
OrQuery	BinaryQuery 的派生类，返回它的两个运算对象分别出现的行的并集
AndQuery	BinaryQuery 的派生类，返回它的两个运算对象分别出现的行的交集

15.9.1 节练习

练习 15.31：已知 s1、s2、s3 和 s4 都是 string，判断下面的表达式分别创建了什么样的对象：

- (a) Query(s1) | Query(s2) & ~ Query(s3);
- (b) Query(s1) | (Query(s2) & ~ Query(s3));
- (c) (Query(s1) & (Query(s2)) | (Query(s3) & Query(s4)));

15.9.2 Query_base 类和 Query 类

下面我们开始程序的实现过程，首先定义 `Query_base` 类：

```
// 这是一个抽象基类，具体的查询类型从中派生，所有成员都是 private 的
class Query_base {
    friend class Query;
protected:
    using line_no = TextQuery::line_no; // 用于 eval 函数
    virtual ~Query_base() = default;
private:
    // eval 返回与当前 Query 匹配的 QueryResult
    virtual QueryResult eval(const TextQuery&) const = 0;
    // rep 是表示查询的一个 string
    virtual std::string rep() const = 0;
};
```

`eval` 和 `rep` 都是纯虚函数，因此 `Query_base` 是一个抽象基类（参见 15.4 节，第 541 页）。因为我们不希望用户或者派生类直接使用 `Query_base`，所以它没有 `public` 成员。所有对 `Query_base` 的使用都需要通过 `Query` 对象，因为 `Query` 需要调用 `Query_base` 的虚函数，所以我们将 `Query` 声明成 `Query_base` 的友元。

受保护的成员 `line_no` 将在 `eval` 函数内部使用。类似的，析构函数也是受保护的，因为它将（隐式地）在派生类析构函数中使用。

Query 类

`Query` 类对外提供接口，同时隐藏了 `Query_base` 的继承体系。每个 `Query` 对象都含有一个指向 `Query_base` 对象的 `shared_ptr`。因为 `Query` 是 `Query_base` 的唯一接口，所以 `Query` 必须定义自己的 `eval` 和 `rep` 版本。

接受一个 `string` 参数的 `Query` 构造函数将创建一个新的 `WordQuery` 对象，然后将它的 `shared_ptr` 成员绑定到这个新创建的对象上。`&`、`|` 和 `~` 运算符分别创建 `AndQuery`、`OrQuery` 和 `NotQuery` 对象，这些运算符将返回一个绑定到新创建的对象上的 `Query` 对象。为了支持这些运算符，`Query` 还需要另外一个构造函数，它接受指向 `Query_base` 的 `shared_ptr` 并且存储给定的指针。我们将这个构造函数声明为私有的，原因是不希望一般的用户代码能随便定义 `Query_base` 对象。因为这个构造函数是私有的，所以我们需要将三个运算符声明为友元。

在形成了上述设计思路后，`Query` 类本身就比较简单了：

```
// 这是一个管理 Query_base 继承体系的接口类
class Query {
    // 这些运算符需要访问接受 shared_ptr 的构造函数，而该函数是私有的
    friend Query operator~(const Query &);
    friend Query operator|(const Query&, const Query&);
    friend Query operator&(const Query&, const Query&);

public:
    Query(const std::string&); // 构建一个新的 WordQuery
    // 接口函数：调用对应的 Query_base 操作
    QueryResult eval(const TextQuery &t) const
        { return q->eval(t); }
    std::string rep() const { return q->rep(); }
```

```

private:
    Query(std::shared_ptr<Query_base> query): q(query) { }
    std::shared_ptr<Query_base> q;
};

```

我们首先将创建 `Query` 对象的运算符声明为友元，之所以这么做是因为这些运算符需要访问那个私有构造函数。

在 `Query` 的公有接口部分，我们声明了接受 `string` 的构造函数，不过没有对其进行定义。因为这个构造函数将要创建一个 `WordQuery` 对象，所以我们应该首先定义 `WordQuery` 类，随后才能定义接受 `string` 的 `Query` 构造函数。

另外两个公有成员是 `Query_base` 的接口。其中，`Query` 操作使用它的 `Query_base` 指针来调用各自的 `Query_base` 虚函数。实际调用哪个函数版本将由 `q` 所指的对象类型决定，并且直到运行时才能最终确定下来。



Query 的输出运算符

输出运算符可以很好地解释我们的整个查询系统是如何工作的：

```

std::ostream &
operator<<(std::ostream &os, const Query &query)
{
    // Query::rep 通过它的 Query_base 指针对 rep() 进行了虚调用
    return os << query.rep();
}

```

当我们打印一个 `Query` 时，输出运算符调用 `Query` 类的公有 `rep` 成员。运算符函数通过指针成员虚调用当前 `Query` 所指对象的 `rep` 成员。也就是说，当我们编写如下代码时：

```

Query andq = Query(sought1) & Query(sought2);
cout << andq << endl;

```

输出运算符将调用 `andq` 的 `Query::rep`，而 `Query::rep` 通过它的 `Query_base` 指针虚调用 `Query_base` 版本的 `rep` 函数。因为 `andq` 指向的是一个 `AndQuery` 对象，所以本次的函数调用将运行 `AndQuery::rep`。

15.9.2 节练习

练习 15.32: 当一个 `Query` 类型的对象被拷贝、移动、赋值或销毁时，将分别发生什么？

练习 15.33: 当一个 `Query_base` 类型的对象被拷贝、移动、赋值或销毁时，将分别发生什么？

642 >

15.9.3 派生类

对于 `Query_base` 的派生类来说，最有趣的部分是这些派生类如何表示一个真实的查询。其中 `WordQuery` 类最直接，它的任务就是保存要查找的单词。

其他类分别操作一个或两个运算对象。`NotQuery` 有一个运算对象，`AndQuery` 和 `OrQuery` 有两个。在这些类当中，运算对象可以是 `Query_base` 的任意一个派生类的对象：一个 `NotQuery` 对象可以被用在 `WordQuery`、`AndQuery`、`OrQuery` 或另一个 `NotQuery` 中。为了支持这种灵活性，运算对象必须以 `Query_base` 指针的形式存储，

这样我们就能把该指针绑定到任何我们需要的具体类上。

然而，实际上我们的类并不存储 `Query_base` 指针，而是直接使用一个 `Query` 对象。就像用户代码可以通过接口类得到简化一样，我们也可以使用接口类来简化我们自己的类。

至此我们已经清楚了所有类的设计思路，接下来依次实现它们。

WordQuery 类

一个 `WordQuery` 查找一个给定的 `string`，它是在给定的 `TextQuery` 对象上实际执行查询的唯一一个操作：

```
class WordQuery: public Query_base {
    friend class Query; // Query 使用 WordQuery 构造函数
    WordQuery(const std::string &s): query_word(s) { }
    // 具体的类：WordQuery 将定义所有继承而来的纯虚函数
    QueryResult eval(const TextQuery &t) const
        { return t.query(query_word); }
    std::string rep() const { return query_word; }
    std::string query_word; // 要查找的单词
};
```

和 `Query_base` 一样，`WordQuery` 没有公有成员。同时，`Query` 必须作为 `WordQuery` 的友元，这样 `Query` 才能访问 `WordQuery` 的构造函数。

每个表示具体查询的类都必须定义继承而来的纯虚函数 `eval` 和 `rep`。我们在 `WordQuery` 类的内部定义这两个操作：`eval` 调用其 `TextQuery` 参数的 `query` 成员，由 `query` 成员在文件中实际进行查找；`rep` 返回这个 `WordQuery` 表示的 `string`（即 `query_word`）。

定义了 `WordQuery` 类之后，我们就能定义接受 `string` 的 `Query` 构造函数了：

```
inline
Query::Query(const std::string &s): q(new WordQuery(s)) { }
```

这个构造函数分配一个 `WordQuery`，然后令其指针成员指向新分配的对象。

NotQuery 类及~运算符

`~` 运算符生成一个 `NotQuery`，其中保存着一个需要对其取反的 `Query`：

```
class NotQuery: public Query_base {
    friend Query operator~(const Query &);
    NotQuery(const Query &q): query(q) { }
    // 具体的类：NotQuery 将定义所有继承而来的纯虚函数
    std::string rep() const { return "~(" + query.rep() + ")"; }
    QueryResult eval(const TextQuery&) const;
    Query query;
};
inline Query operator~(const Query &operand)
{
    return std::shared_ptr<Query_base>(new NotQuery(operand));
}
```

643

因为 `NotQuery` 的所有成员都是私有的，所以我们一开始就要把`~`运算符设定为友元。为

了 rep 一个 NotQuery，我们需要将~符号与基础的 Query 连接在一起。我们在输出的结果中加上适当的括号，这样读者就可以清楚地知道查询的优先级了。

值得注意的是，在 NotQuery 自己的 rep 成员中对 rep 的调用最终执行的是一个虚调用：query.rep() 是对 Query 类 rep 成员的非虚调用，接着 Query::rep 将调用 q->rep()，这是一个通过 Query_base 指针进行的虚调用。

~运算符动态分配一个新的 NotQuery 对象，其 return 语句隐式地使用接受一个 shared_ptr<Query_base> 的 Query 构造函数。也就是说，return 语句等价于：

```
// 分配一个新的 NotQuery 对象
// 将所得的 NotQuery 指针绑定到一个 shared_ptr<Query_base>
shared_ptr<Query_base> tmp(new NotQuery(expr));
return Query(tmp);           // 使用接受一个 shared_ptr 的 Query 构造函数
```

eval 成员比较复杂，因此我们将在类的外部实现它，15.9.4 节（第 573 页）将专门介绍如何定义 eval 函数。

BinaryQuery 类

BinaryQuery 类也是一个抽象基类，它保存操作两个运算对象的查询类型所需的数据：

```
class BinaryQuery: public Query_base {
protected:
    BinaryQuery(const Query &l, const Query &r, std::string s):
        lhs(l), rhs(r), opSym(s) { }
    // 抽象类: BinaryQuery 不定义 eval
    std::string rep() const { return "(" + lhs.rep() + " "
                                + opSym + " "
                                + rhs.rep() + ")"; }
    Query lhs, rhs;           // 左侧和右侧运算对象
    std::string opSym;        // 运算符的名字
};
```

644 BinaryQuery 中的数据是两个运算对象及相应的运算符符号，构造函数负责接受两个运算对象和一个运算符符号，然后将它们存储在对应的数据成员中。

要想 rep 一个 BinaryQuery，我们需要生成一个带括号的表达式。表达式的内容依次包括左侧运算对象、运算符以及右侧运算对象。就像我们显示 NotQuery 的方法一样，对 rep 的调用最终是对 lhs 和 rhs 所指 Query_base 对象的 rep 函数进行虚调用。



BinaryQuery 不定义 eval，而是继承了该纯虚函数。因此，BinaryQuery 也是一个抽象基类，我们不能创建 BinaryQuery 类型的对象。

AndQuery 类、OrQuery 类及相应的运算符

AndQuery 类和 OrQuery 类以及它们的运算符都非常相似：

```
class AndQuery: public BinaryQuery {
    friend Query operator&(const Query&, const Query&);
    AndQuery(const Query &left, const Query &right):
        BinaryQuery(left, right, "&") { }
    // 具体的类: AndQuery 继承了 rep 并且定义了其他纯虚函数
    QueryResult eval(const TextQuery&) const;
```

```

};

inline Query operator&(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new AndQuery(lhs, rhs));
}

class OrQuery: public BinaryQuery {
    friend Query operator|(const Query&, const Query&);
    OrQuery(const Query &left, const Query &right):
        BinaryQuery(left, right, "|") { }
    QueryResult eval(const TextQuery&) const;
};

inline Query operator|(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new OrQuery(lhs, rhs));
}

```

这两个类将各自的运算符定义成友元，并且各自定义了一个构造函数通过运算符创建 BinaryQuery 基类部分。它们继承 BinaryQuery 的 rep 函数，但是覆盖了 eval 函数。

和~运算符一样，&和|运算符也返回一个绑定到新分配对象上的 shared_ptr。在这些运算符中，return 语句负责将 shared_ptr 转换成 Query。

15.9.3 节练习

645

练习 15.34: 针对图 15.3（第 565 页）构建的表达式：

- (a) 列举出在处理表达式的过程中执行的所有构造函数。
- (b) 列举出 cout<<q 所调用的 rep。
- (c) 列举出 q.eval() 所调用的 eval。

练习 15.35: 实现 Query 类和 Query_base 类，其中需要定义 rep 而无须定义 eval。

练习 15.36: 在构造函数和 rep 成员中添加打印语句，运行你的代码以检验你对本节第一个练习中 (a)、(b) 两小题的回答是否正确。

练习 15.37: 如果在派生类中含有 shared_ptr<Query_base>类型的成员而非 Query 类型的成员，则你的类需要做出怎样的改变？

练习 15.38: 下面的声明合法吗？如果不合法，请解释原因；如果合法，请指出该声明的含义。

```

BinaryQuery a = Query("fiery") & Query("bird");
AndQuery b = Query("fiery") & Query("bird");
OrQuery c = Query("fiery") & Query("bird");

```

15.9.4 eval 函数

eval 函数是我们这个查询系统的核心。每个 eval 函数作用于各自的运算对象，同时遵循的内在逻辑也有所区别：OrQuery 的 eval 操作返回两个运算对象查询结果的并集，而 AndQuery 返回交集。与它们相比，NotQuery 的 eval 函数更加复杂一些：它需要返回运算对象没有出现的文本行。

为了支持上述 eval 函数的处理，我们需要使用 QueryResult，在它当中定义了 12.3.2 节练习（第 435 页）添加的成员。假设 QueryResult 包含 begin 和 end 成员，它们允许我们在 QueryResult 保存的行号 set 中进行迭代；另外假设 QueryResult 还包含一个名为 get_file 的成员，它返回一个指向待查询文件的 shared_ptr。



我们的 Query 类使用了 12.3.2 节练习(第 435 页)为 QueryResult 定义的成员。

OrQuery::eval

一个 OrQuery 表示的是它的两个运算对象结果的并集，对于每个运算对象来说，我们通过调用 eval 得到它的查询结果。因为这些运算对象的类型是 Query，所以调用 eval 也就是调用 Query::eval，而后者实际上是对潜在的 query_base 对象的 eval 进行虚调用。每次调用完成后，得到的结果是一个 QueryResult，它表示运算对象出现的行号。我们把这些行号组织在一个新 set 中：

```
646 // 返回运算对象查询结果 set 的并集
QueryResult
OrQuery::eval(const TextQuery& text) const
{
    // 通过 Query 成员 lhs 和 rhs 进行的虚调用
    // 调用 eval 返回每个运算对象的 QueryResult
    auto right = rhs.eval(text), left = lhs.eval(text);
    // 将左侧运算对象的行号拷贝到结果 set 中
    auto ret_lines =
        make_shared<set<line_no>>(left.begin(), left.end());
    // 插入右侧运算对象所得的行号
    ret_lines->insert(right.begin(), right.end());
    // 返回一个新的 QueryResult，它表示 lhs 和 rhs 的并集
    return QueryResult(rep(), ret_lines, left.get_file());
}
```

我们使用接受一对迭代器的 set 构造函数初始化 ret_lines。一个 QueryResult 的 begin 和 end 成员返回行号 set 的迭代器，因此，创建 ret_lines 的过程实际上是拷贝了 left 集合的元素。接下来对 ret_lines 调用 insert，并将 right 的元素插入进来。调用结束后，ret_lines 将包含在 left 或 right 中出现过的所有行号。

eval 函数在最后构建并返回一个表示混合查询匹配的 QueryResult。QueryResult 的构造函数（参见 12.3.2 节，第 434 页）接受三个实参：一个表示查询的 string、一个指向匹配行号 set 的 shared_ptr 和一个指向输入文件 vector 的 shared_ptr。我们调用 rep 生成所需的 string，调用 get_file 获取指向文件的 shared_ptr。因为 left 和 right 指向的是同一个文件，所以使用哪个执行 get_file 函数并不重要。

AndQuery::eval

AndQuery 的 eval 和 OrQuery 很类似，唯一的区别是它调用了一个标准库算法来求得两个查询结果中共有的行：

```
// 返回运算对象查询结果 set 的交集
QueryResult
AndQuery::eval(const TextQuery& text) const
{
```

```

// 通过 Query 运算对象进行的虚调用，以获得运算对象的查询结果 set
auto left = lhs.eval(text), right = rhs.eval(text);
// 保存 left 和 right 交集的 set
auto ret_lines = make_shared<set<line_no>>();
// 将两个范围的交集写入一个目的迭代器中
// 本次调用的目的迭代器向 ret 添加元素
set_intersection(left.begin(), left.end(),
                 right.begin(), right.end(),
                 inserter(*ret_lines, ret_lines->begin()));
return QueryResult(rep(), ret_lines, left.get_file());
}

```

其中我们使用标准库算法 `set_intersection` 来合并两个 `set`，关于 647
`set_intersection` 在附录 A.2.8 (第 779 页) 中有详细的描述。

`set_intersection` 算法接受五个迭代器。它使用前四个迭代器表示两个输入序列 (参见 10.5.2 节, 第 368 页), 最后一个实参表示目的位置。该算法将两个输入序列中共同出现的元素写入到目的位置中。

在上述调用中我们传入一个插入迭代器 (参见 10.4.1 节, 第 357 页) 作为目的位置。当 `set_intersection` 向这个迭代器写入内容时, 实际上是向 `ret_lines` 插入一个新元素。

和 `OrQuery` 的 `eval` 函数一样, `AndQuery` 的 `eval` 函数也在最后构建并返回一个表示混合查询匹配的 `QueryResult`。

NotQuery::eval

`NotQuery` 查找运算对象没有出现的文本行:

```

// 返回运算对象的结果 set 中不存在的行
QueryResult
NotQuery::eval(const TextQuery& text) const
{
    // 通过 Query 运算对象对 eval 进行虚调用
    auto result = query.eval(text);
    // 开始时结果 set 为空
    auto ret_lines = make_shared<set<line_no>>();
    // 我们必须在运算对象出现的所有行中进行迭代
    auto beg = result.begin(), end = result.end();
    // 对于输入文件的每一行, 如果该行不在 result 当中, 则将其添加到 ret_lines
    auto sz = result.get_file()->size();
    for (size_t n = 0; n != sz; ++n) {
        // 如果我们还没有处理完 result 的所有行
        // 检查当前行是否存在
        if (beg == end || *beg != n)
            ret_lines->insert(n);      // 如果不在 result 当中, 添加这一行
        else if (beg != end)
            ++beg;                  // 否则继续获取 result 的下一行 (如果有的话)
    }
    return QueryResult(rep(), ret_lines, result.get_file());
}

```

和其他 `eval` 函数一样, 我们首先对当前的运算对象调用 `eval`, 所得的结果

`QueryResult` 中包含的是运算对象出现的行号，但我们想要的是运算对象未出现的行号。也就是说，我们需要的是存在于文件中，但是不在 `result` 中的行。

要想得到最终的结果，我们需要遍历不超过输出文件大小的所有整数，并将所有不在 `result` 中的行号放入到 `ret_lines` 中。我们使用 `beg` 和 `end` 分别表示 `result` 的第一个元素和最后一个元素的下一位置。因为遍历的对象是一个 `set`，所以当遍历结束后获得的行号将按照升序排列。

648 循环体负责检查当前的编号是否在 `result` 当中。如果不在，将这个数字添加到 `ret_lines` 中；如果该数字属于 `result`，则我们递增 `result` 的迭代器 `beg`。

一旦处理完所有行号，就返回包含 `ret_lines` 的一个 `QueryResult` 对象；和之前版本的 `eval` 类似，该 `QueryResult` 对象还包含 `rep` 和 `get_file` 的运行结果。

15.9.4 节练习

练习 15.39：实现 `Query` 类和 `Query_base` 类，求图 15.3（第 565 页）中表达式的值并打印相关信息，验证你的程序是否正确。

练习 15.40：在 `OrQuery` 的 `eval` 函数中，如果 `rhs` 成员返回的是空集将发生什么？如果 `lhs` 是空集呢？如果 `lhs` 和 `rhs` 都是空集又将发生什么？

练习 15.41：重新实现你的类，这次使用指向 `Query_base` 的内置指针而非 `shared_ptr`。请注意，做出上述改动后你的类将不能再使用合成的拷贝控制成员。

练习 15.42：从下面的几种改进中选择一种，设计并实现它：

- (a) 按句子查询并打印单词，而不再是按行打印。
- (b) 引入一个历史系统，用户可以按编号查阅之前的某个查询，并可以在其中增加内容或者将其与其他查询组合。
- (c) 允许用户对结果做出限制，比如从给定范围的行中挑出匹配的进行显示。