

第 II 部分

C++ 标准库

内容

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

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

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

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

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

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，标准库还定义了一些其他 IO 类型，我们之前都已经使用过了。表 8.1 列出了这些类型，分别定义在三个独立的头文件中：`iostream` 定义了用于读写流的基本类型，`fstream` 定义了读写命名文件的类型，`sstream` 定义了读写内存 `string` 对象的类型。

表 8.1: IO 库类型和头文件	
头文件	类型
iostream	<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
fstream	<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
sstream	<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(outstream); // 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

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

表 8.2: IO 库条件状态	
strm::iostate	strm 是一种 IO 类型，在表 8.1（第 278 页）中已列出。iostate 是一种机器相关的类型，提供了表达条件状态的完整功能
strm::badbit	strm::badbit 用来指出流已崩溃
strm::failbit	strm::failbit 用来指出一个 IO 操作失败了

续表

<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 <code>s</code> 的 <code>eofbit</code> 置位, 则返回 <code>true</code>
<code>s.fail()</code>	若流 <code>s</code> 的 <code>failbit</code> 或 <code>badbit</code> 置位, 则返回 <code>true</code>
<code>s.bad()</code>	若流 <code>s</code> 的 <code>badbit</code> 置位, 则返回 <code>true</code>
<code>s.good()</code>	若流 <code>s</code> 处于有效状态, 则返回 <code>true</code>
<code>s.clear()</code>	将流 <code>s</code> 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 <code>s</code> 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 <code>s</code> 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.rdstate()</code>	返回流 <code>s</code> 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：

◀ 314

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

8.1.2 节练习

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

练习 8.2：测试函数，调用参数为 `cin`。

练习 8.3：什么情况下，下面的 `while` 循环会终止？

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

8.1.3 管理输出缓冲

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

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

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

导致缓冲刷新（即，数据真正写到输出设备或文件）的原因有很多：

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

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

315 刷新输出缓冲区

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

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

unitbuf 操纵符

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

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

警告：如果程序崩溃，输出缓冲区不会被刷新

如果程序异常终止，输出缓冲区是会被刷新的。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: <code>fstream</code> 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。默认的文件模式依赖于 <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;                // 处理下一本书
        }
    }
    print(output, total) << endl; // 打印最后一本书的销售总额
} 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，ostreamstream 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>stringstream strm;</code>	<code>strm</code> 是一个未绑定的 <code>stringstream</code> 对象。 <code>stringstream</code> 是头文件 <code>sstream</code> 中定义的一个类型
<code>stringstream strm(s);</code>	<code>strm</code> 是一个 <code>stringstream</code> 对象，保存 <code>string s</code> 的一个拷贝。此构造函数是 <code>explicit</code> 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。我们可以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 `istreamstream` 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 `istreamstream` 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 `istreamstream` 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 `ostreamstream`

323

当我们逐步构造输出, 希望最后一起打印时, `ostreamstream` 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 `ostreamstream` 中:

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

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

8.3.2 节练习

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

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

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。