# 格式化I/O

**流操纵符仅是一种用于控制格式化输出的一种特定类型值的有趣名称。**C++类库提供了各种各样的流操纵符，可以使用它们来指定输出值的格式，它们中最常见的都显示在表4-1中。

当在程序中包含了`<iostream>`库头文件时，这些流操纵符的绝大部分都是自动可用的。唯一例外是读取参数的流操纵符，例如`set(n)`，`setprecision(digits)`和`setfill(d)`。为了使用这些流操纵符，你还需要包含`<iomanip>`库头文件。

流操纵符典型的作用是通过设置输出流的属性值来改变输出序列的格式。正如表4-1中逐一列出的各条目所阐明的那样，某些流操纵符的作用是短暂的（transient），这意味着它们只影响下一个输出的数据值。然而，大部分流操作符的作用是持久的（persistent），这意味着其作用一直有效，直到它们被明确地改变为止。

表 4-1　输出流操纵符

| | |
|---|---|
| **endl** | 将行结束序列插入到输出流，并确保输出的字符能被写到目的地流中 |
| **setw(*n*)** | 将下一个输出字段的宽度设置为 *n* 个字符。如果输出值所需域宽小于 *n*，则额外的空间用空格填充。这种性质是短暂的，这意味着它只影响下一个插入到流中的数据值的输出宽度 |
| **setprecision(*digits*)** | 将输出流的精度设置为 *digits*。精度说明的解释依赖于其他流的设置。如果你已经将模式设置为 **fixed** 或 **scientific**，*digits* 会指定小数点后数字的位数。如果没有设置以上两种模式，*digits* 表示有效数字的位数，并且不考虑这些数字出现在什么地方。这种性质是持久的，这意味着它一直保持有效，直到它被明确地改变为止 |
| **setfill(*ch*)** | 为流设置填充字符 *ch*。默认地，如果需要额外的字符填充到 **setw** 设置的字段宽度中，则空格作为填充字符输出。调用 **setfill** 使输出流可以改变填充字符。例如，调用 **setfill('0')** 意味着字段将用 0 填充。这种性质是持久的 |
| **left** | 指定输出字段为左对齐，这意味着任何填充字符都在输出值之后插入。这种性质是持久的 |

（续）

| | |
|---|---|
| **right** | 指定输出字段为右对齐，这意味着任何填充字符都在输出值之前插入。这种性质是持久的 |
| **fixed** | 指定之后的浮点数输出应该完整地呈现，并且不使用科学计数法。默认地，浮点数应该以最简洁的形式呈现。这种性质是持久的 |
| **scientific** | 指定之后的浮点数输出应该以科学计数法的形式呈现。这种性质是持久的 |
| **showpoint**<br>**noshowpoint** | 这两个流操纵符控制浮点数中是否能出现小数点，这种控制同样适用于整数的情况。可以用 **showpoint** 强制要求出现小数点，然后通过 **noshowpoint** 来恢复默认设置，这种性质是持久的 |
| **showpos**<br>**noshowpos** | 这两个流操纵符控制在一个正数前是否应有一个正号。默认地，正数前没有正号。这种性质是持久的 |
| **uppercase**<br>**nouppercase** | 这两个流操纵符控制作为数据转换的一部分所产生的任意字母的大小写，例如科学计数法中的大写字符 E。默认地，字符以小写字母呈现。这种性质是持久的 |
| **boolalpha**<br>**noboolalpha** | 这两个流操纵符控制布尔值的格式，它一般使用它们内在的数值表示呈现。使用 **boolalpha** 流操纵符导致它们以 **true** 或 **false** 的形式出现。这种性质是持久的 |



# 格式化输入

C++的格式化输入已嵌入了流操作符>>，你已经在各种各样的程序中用到过它。这个操作符称为提取操作符（extraction operator），因为它用于从一个输入流中提取格式化数据。到目前为止，你已经使用>>操作符从控制台请求输入数据，例如第1章中PowersOfTwo程序的语句行：

```
int limit;
cout << "Enter exponent limit:";
cin >> limit;
```

默认地，>>操作符在尝试读取输入数据之前忽略所有空白字符。如果有必要，可以使用skipws和noskipws流操纵符来改变这种行为，它们显示在表4-2的输入流操纵符的列表中。

表 4-2　输入流操纵符

| | |
|---|---|
| skipws<br>noskipws | 这两个流操纵符控制提取操作符 >> 在读取一个值之前是否忽略空白字符。如果指定 noskipws，提取操作符将所有的字符（包括空白字符）看作是输入字段的一部分。之后可以使用 skipws 恢复默认的行为。这个性质是持久的 |
| ws | 从输入流中读取字符，直到它不属于空白字符。因此，这个流操纵符的作用是跳过输入中的任何空白字符、制表符和换行符。不像 skipws 和 noskipws 改变的是流关于之后的输入操作行为，ws 流操纵符是立即起作用的 |

例如，当你执行下述语句时：

```
char ch;
cout << "Enter a single character:”
cin >> noskipws >>ch;
```

用户可以输入一个空格字符或制表符来响应屏幕提示。一旦你省略了noskipws流操纵符，程序将会在存储ch的下一个输入字符之前跳过空白字符。

尽管提取操作符使得编写一个简单地从控制台读取输入数据的测试程序变得简单，**但在实际中它并没有广泛采用。**

`>>`**操作符的主要问题是它几乎不提供任何支持检测用户输入是否有效的功能。**众所周知，用户在向计算机中输入数据时是很草率的。他们会造成一些“笔误”，或者更糟糕的是，他们根本没有理解程序真正想要什么输入。设计良好的程序会检测用户的输入以确保它形式正确，并且在程序中是有意义的。

> 为此，我们可以实现getInteget()
>
> **习题中会有完整的解析。**

# 文件流

在C++中，读或者写一个文件要求遵循以下步骤：

1. 声明一个指向某个文件的流变量。处理文件的程序通常为每一个活动文件声明一个流变量。

   因此，如果你正在编写一个读取输入文件的程序，然后再处理其中的数据以产生另一个输出文件，那么你需要声明以下两个变量：

   ```
   ifstream infile;
   ofstream outfile;
   ```

2. 打开文件。**在可以使用一个流变量之前，需要在所声明的变量和一个实际的文件间建立关联。**该操作称为打开（opening）文件，它是通过调用流方法open实现的。例如，如果你想读取包含在Jabberwocky.txt文件中的文本，通过执行以下方法调用可以打开这个文件：`infile.open("Jabberwocky.txt");`

~~由于流库先于string类的介绍，因此open方法将C风格的字符串看作是文件名。于是，一个字符串字面值的文件名是可接受的。然而，如果文件名存储在名为filename的string变量中，你可以通过以下方法调用打开文件：~~ `~~infile.open(filename.c_str());~~`

> 测试发现并不影响，后续的CPP版本中可以接受string或者Filename类型。

如果请求的文件丢失，流会记录那个错误，并且可调用判定方法fail去检测它。从这些故障中恢复是你作为一个程序员的责任，在本章的后面，你将学到各种进行故障恢复的策略。

3. 传输数据。一旦你打开了数据文件，你之后会使用合适的流操作去实现实际的I/O操作。根据应用，可以选择任意的传输文件数据策略。最简单的是逐个字符地读或写文件。然而，在某些情况下，逐行处理文件会更方便。在更高的层面上，可以选择读或写格式化数据，它允许你将数值数据与字符串和其他的数据类型混合在一起。这些策略的细节将在后续章节中阐述。
4. 关闭文件。当你结束了所有的文件数据传输后，通过调用流方法close以告知文件系统关闭打开的文件，如下所示：`infile.close();`此操作称为关闭（closing）文件，它切断了流与所关联文件之间的关系。

表 4-3　流类中的有用方法

| 所有的流都支持的方法 | |
|---|---|
| *stream*.**fail()** | 如果流处于失效状态，则返回 **true**。这个条件通常发生在你尝试超出文件的结尾去读取数据的时候，但这也表示数据中出现了一个完整性错误 |
| *stream*.**eof()** | 如果流位于文件的结尾，则返回 **true**。鉴于 C++ 流库的语义，**eof** 方法只用在一个 **fail** 调用之后。此时，**eof** 调用允许你判断故障提示是否是由于到达文件的结尾引起的 |
| *stream*.**clear()** | 重置与流相关的状态位。当一个故障发生后，无论何时需要重新使用一个流，都必须调用这个函数 |
| **if**(*stream*)… | 判断流是否有效。就大部分情况而言，这个测试和调用 **if**(!*stream*.**fail()**) 的效果相同 |

（续）

| 所有文件流都支持的方法 | |
|---|---|
| *stream*.**open**(*filename*) | 尝试打开文件 *filemane* 并将其附加到流中。流的方向由流的类型所决定：输入流对于输入打开，输出流对于输出打开。*filename* 参数是一个 C 风格的字符串，这意味着你将需要在任何 C++ 字符串上调用 **c_str**。通过调用 **fail**，可以检测 **open** 方法是否失败 |
| *stream*.**close**() | 关闭依附于流的文件 |
| **所有的输入流都支持的方法** | |
| *stream*>>*variable* | 将格式化数据读入到一个变量中。数据的格式是由变量类型控制的，并且无论输入流操纵符是什么，它都是有效的 |
| *stream*.**get**(*var*) | 将下一个字符读入到字符变量 *var* 中，*var* 是引用参数。返回值是流本身，如果没有更多的字符，设置 **fail** 标志 |
| *stream*.**get**() | 返回流的下一个字符。返回值是一个整数，它可识别以常量 **EOF** 表示文件结尾的字符 |
| *stream*.**unget**() | 复制流的内部指针以便最后读取的一个字符能再次被下一个 **get** 调用读取 |
| ***getline***(*stream*, *str*) | 将流 *stream* 中的下一行读入到字符串变量 *str* 中。**getline** 函数返回流，它简化了文件结尾的测试 |
| **所有的输出流都支持的方法** | |
| *stream*<<*expression* | 将格式化数据写入到一个输出流。数据格式由表达式的类型所控制，并且对于任何输出流操纵符都有效 |
| *stream*.**put**(*ch*) | 将字符 *ch* 写入到输出流 |

# 字符串流

```
int stringToInteger(string str) {
    istringstream stream(str);
    int value;
    stream >> value >> ws;
    if (stream.fail() || !stream.eof()) {
        error("stringToIntegar: Illegal integer format");
    }
    return value;
}
```

`stream >> value >> ws;`这代码从流中读取一个整数值并将其存储在变量value中。在这个实现中，空白字符允许出现在值前面或后面。第一个>>操作符会自动地跳过任何出现在值前面的空白字符。**位于行结尾的ws流操纵符会读取任何出现在值后面的空白字符**，因此，它确保了：如果输入能正确地初始化，流的位置也是正确的。

```
string integerToString(int n) {
    ostringstream stream;
    stream << n;
    return stream.str();
}
```

# getInteger 代替 >>

```
/*
 * Function: getInteger
 * Usage: int n = getInteger(prompt);
 * ----------------------------------
 * Reads a complete line from <code>cin</code> and scans it as an
 * integer. If the scan succeeds, the integer value is returned. If
 * the argument is not a legal integer or if extraneous characters
 * (other than whitespace) appear in the string, the user is given
 * a chance to reenter the value. If supplied, the optional
 * <code>prompt</code> string is printed before reading the value.
 */

int getInteger(std::string prompt = "") {
    int value;
    string line;
    while (true) {
        cout << prompppt;
        getline(cin, line);
        istringstream stream(line);
        stream >> value >> ws;
        if (!stream.fail() && stream.eof()) break;
        cout << "getIntegar: Illegal integer format. Try again." << endl;
    }
    return value;
}
```

# copyStream:  尽可能使用最广泛的类层次

```
void copyStream(istream &is, ostream &os) {
    char ch;
    while (is.get(ch)) {
        os.put(ch);
    }
}
```

如此，可以实现`copyStream(infile, cout)` 将文件内容复制到cout中。

# simpleio.h 和 filelib.h

第3章介绍了几种新的函数，将它们打包成类似strlib.h库的形式是很有用的。

在这一章中，你已经见过几种有用的处理流的工具，在Stanford类库中，流已封装为两种接口：第2章介绍的simpio.h接口和用于文件密切相关的方法的filelib.h接口。

虽然表列描述非常适合打印在纸张上，但基于网络版的在线文档更适合在线浏览，但你还有另一种选择来学习一个接口中什么资源是可用的：你可以阅读.h文件。**如果一个接口能有效地被设计并记录，阅读.h文件可以提供你所需的所有信息。在任何情况下，如果你想精通C++，阅读.h文件是你需要培养的一项编程技能。**

> 为了使你在阅读接口方面进行一些实践，你应该阅读由Stanford类库提供的filelib.h接口。这个接口包括了本章给出的promptUserForFile函数，以及许多其他的函数，当你处理文件时，它们迟早会有用。

simpio.h (stanford.edu)

filelib.h (stanford.edu)

The Stanford Library documentation