China-pub.com
下载

第6章　输入输出流介绍

到目前为止，在这本书里，我们仍使用以前的可靠的C标准I/O库，这是一个可变成类的完美的例子。

事实上，处理一般的I/O问题，比仅仅用标准库并把它变为一个类，需要做更多的工作。如果能使得所有这样的“容器”——标准I/O、文件、甚至存储块——看上去都一样，只须记住一个接口，不是更好吗？这种思想是建立在输入输出流之上的。与标准C输入输出库的各种各样的函数相比，输入输出流更容易、安全、有效。

输入输出流通常是C++初学者学会使用的第一个类库。在这一章里，我们要考察输入输出流的用途，这样，输入输出流要代替这本书剩余部分的C I/O函数。下一章，我们会发现如何建立我们自己的与输入输出流相容的类。

6.1　为什么要用输入输出流

我们可能想知道以前的C库有什么不好。为什么不把C库封装成一个类，然后进行处理？其实，当我们想使C库用起来更安全、更容易一点时，在有些情况下，这样做很完美。例如，当我们想确保一标准输入输出文件总是被安全地打开，被正确地关闭，而不依赖用户是否记得调用close()函数：

```
//: FILECLAS.H -- Stdio files wrapped
#ifndef FILECLAS_H_
#define FILECLAS_H_
#include <stdio.h>

class file {
  FILE* f;
public:
  file(const char* fname, const char* mode="r");
  ~file();
  FILE* fp();
};
#endif // FILECLAS_H_
```

在C中执行文件I/O时，要用一个没有保护的指针指向文件结构。而这个类封装了这个指针，并用构造函数和析构函数保证它能被正确地初始化和清除。第二个构造函数参数是文件模式，其缺省值为“r”，代表“只读”。

在文件I/O函数中，为了取指针的值，可使用fp()访问函数。下面是成员函数的定义：

```
//: FILECLAS.CPP -- Stdio files wrapped
#include <stdlib.h>
#include "..\5\fileclas.h"

file::file(const char* fname, const char* mode){
```

China-pub.com 下载

```
  f = fopen(fname, mode);
  if(f == NULL) {
    printf("%s: file not found\n", fname);
    exit(1);
  }
}

file::~file() {
  fclose(f);
}

FILE* file::fp() {
  return f;
}
```

就像常常做的一样，构造函数调用 fopen( )，但它还检查结果，从而确保结果不是零，如果结果是零意味着打开文件时出错。如果出错，这个文件名就被打印，而且函数 exit( ) 被调用。

析构函数关闭这个文件，存取函数 fp( ) 返回值f。下面是使用类文件的一个简单的例子：

```
//: FCTEST.CPP -- Testing class file
#include <assert.h>
#include "..\5\fileclas.h"

main(int argc, char* argv[]) {
  assert(argc == 2);
  file f(argv[1]); // Opens and tests
  #define BSIZE 100
  char buf[BSIZE];
  while(fgets(buf, BSIZE, f.fp()))
    puts(buf);
} // File automatically closed by destructor
```

在该例子中创建了文件对象，在正常C文件I/O函数中通过调用 fp( ) 来使用它。当我们使用完毕时，就忘掉它，则这个文件在该范围的末端由析构函数关闭。

正式的真包装

即使文件指针是私有的，但由于 fp( ) 检索它，所以也不是特别安全的。仅有的保证作用是初始化和销毁。既然这样，为什么不使文件指针是公有的，或者用结构体来代替？注意使用 fp() 得到f的副本时，不能赋值给f —— 它完全处于类控制下。当然，一旦用户使用fp()返回的指针值，他仍能对结构元素赋值。所以安全性是保证一个有效的文件指针而不是结构里的正确内容。

如果想绝对安全，必须禁止用户直接访问文件指针。这意味着所有正常的文件 I/O函数的某些版本将必须作为类成员表现出来。这样，通过C途径所做的每件事，在C++类中均可做到：

```
//: FULLWRAP.H -- Completely hidden file IO
#ifndef FULLWRAP_H_
#define FULLWRAP_H_
```

China-pub.com
下载


```
#include <stdio.h>

class File {
  FILE* f;
  FILE* F(); // Produces checked pointer to f
public:
File(); // Create object but don't open file
File(const char* path,
     const char* mode = "r");
~File();
int open(const char* path,
         const char* mode = "r");
int reopen(const char* path,
           const char* mode);
int Getc();
int Ungetc(int c);
int Putc(int c);
int puts(const char* s);
char* gets(char* s, int n);
int printf(const char* format, ...);
size_t read(void* ptr, size_t size,
            size_t n);
size_t write(const void* ptr,
             size_t size, size_t n);
int eof();
int close();
int flush();
int seek(long offset, int whence);
int getpos(fpos_t* pos);
int setpos(const fpos_t* pos);
long tell();
void rewind();
void setbuf(char* buf);
int setvbuf(char* buf, int type, size_t sz);
int error();
void Clearerr();
};
#endif // FULLWRAP_H_
```

这个类包含几乎所有的来自STDIO.H文件的I/O函数。没有vfprintf()函数，它只是被用来实现printf()成员函数。

File有着与前面例子相同的构造函数，而且也有缺省的构造函数。如果要建立一个File对象数组或使用一个File对象作为另一个类的成员，在那个类里，构造函数不发生初始化（但有时在被包含的对象创建后初始化），那么缺省构造函数是重要的。

缺省构造函数设置私有File指针f为0，但是现在，在f的任何引用之前，它的值必须要被检

China-pub.com 下载

查以确保它不为0。这是由类的最后一个成员函数F（）来完成的。F（）函数仅由其他成员函数使用，因此是私有的。（我们不想让用户直接访问这个类的File结构）[1]

从任何意义上讲，这不是一个很坏的解决办法。它是相当有效的，可以设想为标准（控制台）I/O和内核格式化（读/写一个存储块而不是一个文件或控制台）构造类似的类。

大的障碍是运行期间用作参数表函数的解释程序。这是在运行期间对格式串做语法分析以及从参数表中索取并解释变量的代码。产生这个问题的四个原因是：

1) 即使仅使用解释程序的一部分功能，所有的东西将获得装载。假如说：

```
printf("%c",'x');
```

我们将得到整个包，包括打印出浮点数和串的那部分。没有办法可减少程序使用的空间。

2) 由于解释发生在运行期间，所以不能终止这个执行。令人灰心的是在编译时，格式串里的所有信息都在这儿，但是直到运行时，才能对其求值。但是，如果能在编译时分析格式串里的变量，就可以建立硬函数调用，它比运行期间解释程序快得多（虽然printf()族函数通常已被优化得很好）。

3) 由于直到运行期间才对格式串求值，一个更糟糕的问题出现了：可能没有编译时的错误检查。如果我们已经尝试找出由于在printf()说明里使用错误的数或变量类型而产生的错误，我们大概对这个问题很熟悉了。C++对编译期间错误检查作了许多工作，使我们及早发现错误，使工作更容易。特别是因为I/O库用得很多，如果弃之不用，那是很不明智的。

4) 对C++，最重要的问题是函数中的printf()族不是能扩展的。它们被设计是用来处理C中四类基本的数据类型（字符，整型，浮点数，双精度及它们的变形）。我们可能认为每增加一个新类时，都能增加一个重载的printf()和scanf()函数（以及它们对文件和串的变量）。但是要记住，重载函数在参数表里必须有不同的类型，printf()族把类型信息隐藏在可变参数表和格式串中。对一个像C++这样其目标是能容易地添加新的数据类型的语言，这是一个笨拙的限制。

6.2 解决输入输出流问题

所有这些问题都清楚地表明：C++中应该有一个最高级别标准类库，用以处理I/O。由于“hello,world”差不多是每个人使用一种新的语言所写的第一个程序，而且由于I/O通常是每个程序的一部分，因此C++中的I/O库必须特别容易使用。这是一个巨大的挑战：它不知道必须适应哪些类，但是它必须能适用于任何新的类。这样的约束要求这个最高级别的类是一个真正的有灵感的设计。

这一章看不到这个设计的细节以及如何向我们自己的类中加入输入输出流功能（在以后的章节里将会看到）。首先，我们必须学会使用输入输出流，其次，在处理I/O和格式时，我们除了能做大量的调节并提高清晰度外，还会看到，一个真正的、功能强大的C++库是如何工作的。

6.2.1 预先了解操作符重载

在使用输入输出流库前，必须明白这个语言的一个新性能，这一性能的详细情况在下一章介绍。要使用输入输出流，必须知道C++中所有操作符具有不同的含义。在这一章，我们特别讲一下“<<”和“>>”，我们说“操作符具有不同的含义”，这值得进一步探讨。

在第5章中，我们已经学到怎样用不同的参数表使用相同的函数名。编译器在编译一个变量后跟一个操作符再后跟一个变量组成的表达式时，它只调用一个函数。那就是说，一个操作

[1] FULLWRAP test file和其实现在此书的源程序中提供，详见前言。

China-pub.com
下载

符只不过是不同语法的函数调用。

当然，这就是C++在数据类型方面的特别之处。必须有一个事先说明过的函数来匹配操作符和那些特别变量类型，否则编译器不接受这个表达式。

大多数人发现，操作符重载的麻烦是由于这样的想法：即我们知道 C操作符的所有知识是错的。这是不对的。下面是C++的设计的两个主要目的：

1) 一个用C编译过的程序用C++也能编译。C++编译器仅有的编译错误和警告源于C语言的“漏洞”，修正这些需要局部编辑（其实， C++编译器的提示一般会使我们直接查找到 C程序中未被发现的错误）。

2) 用C++重新编译，C ++编译器不会暗地里改变C程序的行为。

记住这些目的有助于回答许多问题。知道从 C转向C++并不是无规律的变化，会使这种转变更容易。特别是，内部数据类型的操作符将不会以不同的方式工作 —— 不能改变它们的意思。重载的操作符仅能在包含新的数据类型的地方创建。所以能为一个新类建立一个新的重载操作符，但是表达式

```
1<<4；
```

不会突然间改变它的意思，而且非法代码

```
1.414<<1;
```

也不会突然能开始工作了。

6.2.2 插入符与提取符

在输入输出流库里，两个操作符被重载，目的是能容易地使用输入输出流。操作符“ <<”经常作为输入输出流的插入符，操作符“ >>”经常作为提取符。

一个流是一个格式化并保存字节的对象。可以有一个输入流（ istream）或一个输出流(ostream)。有不同类型的输入流和输出流：文件输入流(ifstreams)和文件输出流(ofstreams)、char*内存的（内核格式化）输入流（istrstreams）和输出流(ostrstreams)、以及与标准C++串（string）类接口的串输入流（istringstreams）和串输出流(ostringstreams)。不管在操作文件、标准I/O、存储块还是一串对象中所有这些流对象有相同的接口。单个接口被扩展用于支持新的类。

如果一个流能产生字节（一个输入流），可以用一个提取符从这个流中获取信息。这个提取符产生并格式化目的对象所期望的信息类型。可以使用 cin对象看一下这个例子。 cin对象相当于C中stdin的输入输出流，也就是可重定向的标准输入。不论何时包含了 IOSTREAM.H头文件，这个对象就被预定义了（这样，输入输出流库能自动与大部分编译器连接）。

```
int i;
cin >> i;

float f;
cin >> f;

char c;
cin >> c;

char buf[100];
cin >> buf;
```

China-pub.com 下载

每个数据类型都有重载操作符“ >> ”，这些数据类型在一个输入语句里作为“ >> ”右边参数使用（也可以重载我们自己的操作符，这一点将在下一章讲到）。

为了发现各种各样的变量里有什么，我们可以用带插入符“ << ”的cout对象(与标准输出相对应，还有一个与标准错误相对应的cerr对象)：

```
cout << "i = ";
cout << i;
cout << "\n";
cout << "f = ";
cout << f;
cout << "\n";
cout << "c = ";
cout << c;
cout << "\n";
cout << "buf = ";
cout << buf;
cout << "\n";
```

这是特别乏味的，而且对于 printf()函数，看来类型检查没有多大或根本没有改进。幸运的是，输入输出流的重载插入符和提取符，被设计在一起连接成一个更容易书写的复合表达式：

```
cout << "i = " << i << endl;
cout << "f = " << f << endl;
cout << "c = " << c << endl;
cout << "buf = " << buf << endl;
```

在下一章中，我们将明白这是怎样发生的，但是现在，以类用户的态度知道它如何工作也就足够了。

1. 操纵算子

这里已经添加了一个新的元素：一个称作 endl的操纵算子。一个操纵算子作用于流上，这种情况下，插入一新行并清空流（消除所有存储在内部流缓冲区里的还没有输出的字符）。也可以只清空流：

```
cout<<flush；
```

另外有一个基本的操纵算子把基数变为oct(八进制)，dec(十进制)或hex(十六进制)：

```
cout<<hex<<"0x"<<i<<endl；
```

有一个用于提取的操纵算子“ 跳过 ”空格：

```
cin>>ws；
```

还有一个叫 ends的操纵算子和 endl操纵算子一样，仅仅用于 strstreams（稍后介绍）。这些是IOSTREAM.H里所有的操纵算子， IOMANIP.H里会有更多的操纵算子，下一章介绍。

6.2.3 通常用法

虽然cin和提取符“ >> ”为cout和插入符“ << ”提供了一个良好的平衡，但在使用格式化的输入机制，尤其是标准输入时，会遇到与 scanf()中同样的问题。如果输入一个非期望值，进程则被偏离，而且它很难恢复。另外，格式化的输入缺省以空格为分隔符。这样，如果把上面

China-pub.com
下载

的代码块搜集成一个程序：

```
//: IOSEXAMP.CPP -- Iostream examples
#include <iostream.h>

main() {
  int i;
  cin >> i;

  float f;
  cin >> f;

  char c;
  cin >> c;

  char buf[100];
  cin >> buf;

  cout << "i = " << i << endl;
  cout << "f = " << f << endl;
  cout << "c = " << c << endl;
  cout << "buf = " << buf << endl;

  cout << flush;
  cout << hex << "0x" << i << endl;
}
```

给出以下输入：

```
12 1.4 c this is a test
```

我们应该得到与输入相同的输出

```
12
1.4
c
this is a test
```

但输出是：（有点不是所期望的）

```
i = 12
f = 1.4
c = c
buf = this
0xc
```

注意到buf只得到第一个字，这是由于输入机制是通过寻找空格来分隔输入的，而空格在“this”的后面。另外，如果连续的输入串长于为buf分配的存储空间，会发生buf溢出现象。

看来提供cin和提取符只是为了完整性，这可能是查看它的一个好方法。实际上，有时想得到列字符的一行一行排列的输入，然后扫描它们，一旦它们安全地处于缓冲区就执行转换。这样，我们就不必担心输入机制因非期望数据而阻塞了。

China-pub.com 下载

另一件要考虑的事是命令行接口的整体概念。当控制台还比不上一台玻璃打字机时人们就对这一点有所认识。世界发展迅速，现在图形用户接口（GUI）处于支配地位。这样的世界里，控制台I/O的意思是什么呢？除了很简单的例子或测试外，可以完全忽略 cin，并采取以下更有意义的步骤：

1) 如果程序需要输入，从一个文件中读取输入——会发现使用输入输出流文件非常容易。输入输出流在GUI（图形用户接口）下仍运行得很好。

2) 只读输入而不想去转换它。一旦在某处获得输入，它在转换时不会影响其他，那么我们可以安全地扫描它。

3) 输出的情况有所不同。如果正在使用图形用户接口，则必须将输出送到一个文件中（这与将输出送到cout是相同的）或者使用图形用户接口设备显示数据，cout不工作。然而，通常把输出送到cout是有意义的。在这两种情况下，输入输出流的输出格式化函数都是很有用的。

6.2.4 面向行的输入

要获取一行输入，有两种选择：成员函数 get（）或getline（）。两个函数都有三个参数：指向存储结果字符的缓冲区指针、缓冲区大小（不能超过其限度）和知道什么时候停止读输入的终止符。终止符有一个经常用到的缺省值“ \n ”。两个函数遇到输入终止符时，都把零储存在结果缓冲区里。

其不同点是什么呢？差别虽小但极其重要： get（）遇到输入流的分隔符时就停止，而不从输入流中提取分隔符。如果用同样的分隔符再调用一次 get（）函数，它会立即返回而不带任何输入。（要么在下一个get（）说明里用一个不同的分隔符，要么用一个不同的输入函数）。getline（）与其相反，它从输入流中提取分隔符，但仍没有把它储存在结果缓冲区里。

总之，当我们在处理文本文件时，无论什么时候读出一行，都会想到用 getline（）。

1. get（）的重载版本

get（）有三种其他的重载版本：一个没有参数表，返回下一个字符，用的是一个 int返回值；一个把字符填进字符参数表，用一个引用（想立即弄明白这个，要跳到第 10章看）；一个直接存储在另一个输入输出流对象的基本缓冲区结构里。这一点在本章的后面介绍。

2. 读原始字节

如果想确切知道正在处理什么，并把字节直接移进内存中的变量、数组或结构中，可以用read（）函数。第一个参数是一个指向内存目的地址的指针，第二个参数指明要读的字节数。预先将信息存储在一个文件里特别有用。例如，在二进制形式里，对一个输出流使用相反的write（）成员函数。以后我们会看到所有这些函数的例子。

3. 出错处理

除没有参数表的 get（）外。所有 get（）和getline（）的版本都返回字符来源的输入流，没有参数表的get（）返回下一个字符或EOF。如果取回输入流对象，要询问一下它是否正确。事实上，我们可用成员函数 good（）、eof（）、fail（）和bad（）询问任何输入输出流是否正确。这些返回状态信息基于 eofbit（指缓冲位于序列的末尾）、failbit（指由于格式化问题或不影响缓冲区的其他问题而使操作失败）和badbit（指缓冲区出错）。

然而，正如前面提到的，如果想输入特定类型，而且从输入中读出的类型与所期望的不一致时，输入流的状态一般要遭到莫名其妙的破坏。当然可通过处理输入流来改正这个问题。如果大家按照我的建议，一次读入一行或一个大的字符段（用 read（）函数），并且除简单情况外不使用输入格式函数，那么，所关心的只是读入位置是否处于输入的末尾。幸好这种测试很

China-pub.com
下载

简单，而且能在条件内部完成，如while(cin)和if(cin)等。要接受这样的事实，当我们在上下文环境中使用输入流对象时，正确值被安全、正确和魔术般地产生出来，以表明对象是否达到输入的末尾。我们也可以像if（！cin）一样，使用布尔非运算“！”，表明这个流不正确，即我们可能已经达到输入的末尾了，应该停止读这个流。

有时候，流处于非正确状态，我们知道这个情况，并且想继续使用它。例如，如果我们到达文件的末尾，eofbit和failbit被设置，这样，那个流对象的情况表明：这个流不再是正确的了。我们可能想通过寻找以前的位置并读出更多的数据而继续使用这个文件。要改变这个情况，只需简单地调用clear()成员函数即可[1]。

6.3 文件输入输出流

用输入输出流操作文件比在C中用STDIO.H要容易得多，安全得多。要打开文件，所做的就是建立一个对象。构造函数做打开文件的工作。不必明确地关闭一个文件（尽管我们用close()成员函数也能做到）。这是因为当对象超出范围时，析构函数将其关闭。要建立一缺省输出文件，只要建一个ofstream对象即可。下面这个例子说明到目前为止已讨论的很多特性。注意FSTREAM.H文件包含声明文件I/O类，这也包含IOSTREAM.H文件。

```
//: STRFILE.CPP -- Stream I/O with files
// The difference between get() & getline()
#include <fstream.h>  // Includes iostream.h
#include <assert.h>
#define SZ 100  // Buffer size

main() {
  char buf[SZ];
  {
    ifstream in("strfile.cpp"); // Read
    assert(in); // Ensure successful open
    ofstream out("strfile.out"); // Write
    assert(out);
    int i = 1; // Line counter

    // A less-convenient approach for line input:
    while(in.get(buf, SZ)) { // Leaves \n in input
      in.get(); // Throw away next character (\n)
      cout << buf << endl; // Must add \n
      // File output just like standard I/O:
      out << i++ << ": " << buf << endl;
    }
  } // Destructors close in & out

  ifstream in("strfile.out");
  assert(in);
```

[1] 更新的实现将支持输入输出流的这种处理错误的方式，但某些情况下也会弹出异常。

China-pub.com 下载

```
// More convenient line input:
while(in.getline(buf, SZ)) { // Removes \n
  char* cp = buf;
  while(*cp != ':')
    cp++;
  cp += 2; // Past ": "
  cout << cp << endl; // Must still add \n
 }
}
```

建立ifstream和ofstream，后跟一个assert()以保证文件能成功地被打开。还有一个对象，用在编译器期望一个整型结果的地方，产生一个表明成功或失败的值。 (这样做要调用一个自动类型转换成员函数,这一点将在第11章讨论)。

第一个while循环表明get()函数的两种形式的用法。一是不论读到第 SZ-1个字符还是遇到第三个参数（缺省值为“ \n ”），get（）函数把字符取进一个缓冲区内，并放入一个零终止符。get（）把终止符留在输入流内，这样，通过使用不带参数形式的 get（），这个终止符必然通过in.get()而被扔掉，这个 get（）函数取回一个字节并使它作为一个 int类型返回。二是可以用ignore（）成员函数，它有两个缺省的参数，第一个参数是扔掉字符的数目，缺省值是 1，第二参数表示ignore()函数退出处的那个字符（在提取后），缺省值是EOF。

下面将看到两个看起来很类似的输出说明： cout和out。注意这是很合适的，我们不必担心正在处理的是哪种对象，因为格式说明对所有的 ostream对象同样有效。第一类输入回显行至标准输出，第二类写行至新文件并包括一个行数目。

为说明getline（），打开我们刚刚建立的文件并除去行数是一件很有趣的事。在打开一个要读的文件之前，为确保文件是正当关闭的，有两种选择。可以用大括号包住程序的第一部分以迫使out对象脱离范围，这样，调用析构函数并在这里关闭这个文件。也可以为两个文件调用close（），如想这样做，可以调用open（）成员函数重用in对象（也可以像第12章讲的那样，在堆里动态地创建和消除对象）。

第二个while循环说明getline（）如何从其遇到的输入流中移走终止符（它的第三个参数缺省值是“ \n ”）。然而getline()像get()一样，把零放进缓冲区，而且它同样不能插入终止符。

打开方式

可以通过改变缺省变量来控制文件打开方式。下表列出控制文件打开方式的标志。

标　志	函　数
ios::in	打开一个输入文件，用这个标志作为 ifstream的打开方式，以防止截断一个现成的文件
ios::out	打开一个输出文件，当用于一个没有 ios::app、ios::ate或ios::in的ofstream时，ios::trunc被隐含
ios::app	以追加的方式打开一个输出文件
ios::ate	打开一现成文件（不论是输入还是输出）并寻找末尾
ios::nocreate	仅打开一个存在的文件（否则失败）
ios::noreplace	仅打开一个不存在的文件（否则失败）
ios::trunc	如果一个文件存在，打开它并删除旧的文件
ios::binary	打开一个二进制文件，缺省的是文本文件

China-pub.com
下载

这些标志可用一个“位或”（OR）运算来连接。

6.4 输入输出流缓冲

无论什么时候建立一个新类，都应当尽可能努力地对类的用户隐藏类的基本实现详情，仅显示他们需要知道的东西，把其余的变为私有以避免产生混淆。通常，当我们使用输入输出流的时候，我们不知道或不关心在哪个字节被产生或消耗。其实，无论正在处理的是标准 I/O、文件、内存还是某些新产生的类或设备，情况都有所不同。

这时，重要的是把消息发送到产生和消耗字节的输入输出流。为了提供通用接口给这些流并且仍然隐藏其基本的实现，它被抽像成自己的类，叫streambuf。每一个输入输出流都包含一个指针，指向某种streambuf（这依赖于它是否处理标准I/O、文件、内存等等）。我们可以直接访问streambuf。例如，可以向streambuf移进、移出原始字节，而不必通过输入输出流来格式化它们。当然，这是通过调用streambuf对象的成员函数来完成的。

当前，我们要知道的最重要的事是：每个输入输出流对象包含一个指向 streambuf的指针，而且，如果需要调用的话，streambuf有我们可以调用的成员函数。

为了允许我们访问streambuf，每个流对象有一个叫做rdbuf()的成员函数，这个函数返回指向对象的streambuf的指针。这样，我们可以为下层的 streambuf调用任何成员函数。然而，对streambuf指针所做的最有兴趣的事之一是：使用“ << ”操作符将其与另一个输入输出流联结。这使我们的对象中的所有字节流进“ << ”左边的对象中。这意味着，如果把一个输入输出流的所有字节移到另一个输入输出流，我们不必做读入它们的一个字节或一行这样单调的工作。这是一流的方法。

例如，下面是打开一个文件并将其内容发送到标准输出（类似前面的例子）的一个很简单的程序：

```
//: STYPE.CPP -- Type a file to standard output
#include <fstream.h>
#include <assert.h>

main(int argc, char* argv[]) {
  assert(argc == 2); // Must have a command line
  ifstream in(argv[1]);
  assert(in); // Exits if it doesn't exist
  cout << in.rdbuf(); // Outputs entire file
}
```

在确信命令行有一个参数后，通过使用这个变数建立一个文件输入流 ifstream。如果这个文件不存在，打开它时将会失败，这个失败被assert(in)捕获。

所有的工作实际上在这个说明里完成：

```
cout<<in.rdbuf()；
```

它把文件的整个内容送到cout。这不仅比代码更简明扼要，也比在每次移动字节更加有效。

使用带streambuf的get()函数

有一种get（）形式允许直接向另一对象的 streambuf写入。第一个参数是streambuf的目的

China-pub.com 下载

地址（它的地址神秘地由一个引用携带，第 10章讨论这个问题）。第二个参数是终止符，它终止get（）函数。所以，打印一个文件到标准输出的另一方法是：

```
//: SBUFGET.CPP -- Get directly into a streambuf
#include <fstream.h>

main() {
  ifstream in("sbufget.cpp");
  while(in.get(*cout.rdbuf()))
    in.ignore();
}
```

rdbuf()返回一个指针，它必须逆向引用，以满足这个函数看到对象的需要。 get（）函数不从输入流中拖出终止符，必须通过调用 ignore（）移走终止符。所以， get()永远不会跳到新行上去。

我们可能不必经常使用这样的技术，但知道它的存在是有用的。

6.5 在输入输出流中查找

每种输入输出流都有一个概念：“ 下一个 ”字符来自哪里（若是输入流）或去哪里（若是输出流）。在某些情况下,可能需要移动这个流的位置，可以用两种方式处理：第一种方式是在流里绝对定位，叫流定位（streampos）；第二种方式像标准C库函数fseek()那样做，从文件的开始、结尾或当前位置移动给定数目的字节。

流定位（streampos）方法要求先调用“ tell ”函数：对一个输出流用 tellp（）函数，对一个输入流用tellg（）函数。(“ p ”指“ 放指针 ”, “ g ”指“ 取指针 ”)。要返回到流中的那个位置时，这个函数返回一个streampos，我们以后可以在用于输出流的 seekp（）函数或用于输入流的seekg（）函数的单参数版本里使用这个streampos。

另一个方法是相对查找，使用 seekp（）和seekg（）的重载版本。第一个参数是要移动的字节数，它可以是正的或负的。第二个参数是查找方向：

Ios::beg	从流的开始位置
Ios::cur	从流的当前位置
Ios::end	从流的末尾位置

下面是一个说明在文件中移动的例子，记住，不仅限于在文件里查找，就像在 C和C++中的STDIO.H一样。在C++中，我们可以在任何类型的流中查找(虽然查找时， cin和cout的方式未被定义)：

```
//: SEEKING.CPP -- Seeking in iostreams
#include <fstream.h>
#include <assert.h>

main(int argc, char* argv[]) {
  assert(argc == 2);
  ifstream in(argv[1]);
  assert(in); // File must already exist
  in.seekg(0, ios::end); // End of file
  streampos sp = in.tellg(); // Size of file
```

China-pub.com
下载


```
  cout << "file size = " << sp << endl;
  in.seekg(-sp/10, ios::end);
  streampos sp2 = in.tellg();
  in.seekg(0, ios::beg); // Start of file
  cout << in.rdbuf(); // Print whole file
    in.seekg(sp2); // Move to streampos
    // Prints the last 1/10th of the file:
    cout << endl << endl << in.rdbuf() << endl;
}
```

这个程序从命令行中读取文件名，并作为一个文件输入流（ifstream）打开。assert（）检测打开是否失败。由于这是一种输入流，因此用 seekg（）来定位“取指针”，第一次调用从文件末查找零字节，即到末端。由于 streampos是一个long的typedef，那里调用tellg（），返回被打印文件的大小。然后执行查找，移取指针至文件大小的 1/10处——注意那是文件尾的反向查找，所以指针从尾部退回。如我们想进行从文件尾的正向查找，取指针刚好停在文件尾。那里的streampos被读进sp2，然后，seekg（）会到文件开始处执行，整个过程可通过由rdbuf（）产生的streambuf指针打印出来。最后，seekg（）的重载版与streampos sp2一起使用，移到先前的位置，文件的最后部分被打印出来。

建立读/写文件

既然了解streambuf并知道怎样查找，我们会明白怎样建立一个既能读又能写文件的流对象。下面的代码首先建立一个有标志的 ifstream，它既是一个输入文件又是一个输出文件，编译器不会让我们向ifstream写，因此，需要建立具有基本流缓冲区的输出流（ostream）：

```
ifstream in("filename", ios::in|ios::out);
ostream out(in.rdbuf());
```

我们可能想知道向这样一个对象写内容时，会发生什么。下面是一个例子：

```
//: IOFILE.CPP -- Reading & writing one file
#include <fstream.h>
main() {
  ifstream in("iofile.cpp");
  ofstream out("iofile.out");
  out << in.rdbuf(); // Copy file
  in.close();
  out.close();
  // Open for reading and writing:
  ifstream in2("iofile.out",ios::in|ios::out);
  ostream out2(in2.rdbuf());
  cout << in2.rdbuf();  // Print whole file
  out2 << "Where does this end up?";
  out2.seekp(0, ios::beg);
  out2 << "And what about this?";
  in2.seekg(0, ios::beg);
  cout << in2.rdbuf();
}
```

China-pub.com 下载

前五行把这个程序的源代码拷贝进一个名叫 iofile.out的文件，然后关闭这个文件。这给了我们一个可在其周围操作的安全的文本文件。那么前面提及的技术被用来建立两个对象，这两个对象向同一个文件读和写。在 cout<<in2.rdbuf()里，可看到“ 取 ”指针在文件的开始被初始化。“ 放 ”指针被放到文件的末尾，这是由于“ Where does this end up? ”追加到这个文件里。然而，如果“ 放 ”指针移到 seekp（）的开始处，所有插入的文本覆盖现成的文本。当“ 取 ”指针用seekg（）移回到开始处时，两次写结果均可见到，而且文件被打印出来。当然，当 out2 脱离范围时，析构函数被调用，这个文件被自动保存和关闭。

6.6 strstreams

第三个标准型输入输出流可直接与内存而不是一个文件或标准输出一起工作。它允许我们用同样的读函数和格式函数去操作内存里的字节。旧式的计算机，内存是指内核，所以，这种功能有时叫内核格式。

strstream的类名字回显文件流的类名字。如想建立一个从中提取字符的 strstream，我们就建立一个istrstream。如想把字符放进一个strstream，我们就建立一个ostrstream。

串流与内存一起工作，所以我们必须处理这样的问题：内存来自哪里又去哪里。这个问题并不复杂到令人害怕的程度，但我们必须弄懂并注意它。从 strstreams中得到的好处远大于这一微小的不利。

6.6.1 为用户分配的存储

由用户负责分配存储空间，恰好是弄懂这个问题的最容易的途径。用 istrstreams，这是唯一允许的方法。下面是两个构造函数：

```
istrstream::istrstream(char* buf);
istrstream::istrstream(char* buf, int size);
```

第一个构造函数取一个指向零终止符数组的指针；我们可以提取字节直到零为止。第二个构造函数另外还需要这个数组的大小，这个数组不必是零终止的。我们可以一直提取字节到buf[size]，而不管是否遇到一个零。

当移交数组地址给一个istrstream构造函数时，这个数组必须已经填充了我们要提取的并且假定格式化成某种其他数据类型的字符。下面是一个简单的例子 [1]：

```
//: ISTRING.CPP -- Input strstreams
#include <strstrea.h>

main() {
  istrstream s("1.414 47 This is a test");
  int i;
  float f;
  s >> i >> f; // Whitespace-delimited input
  char buf2[100];
  s >> buf2;
  cout << "i = " << i << ", f = " << f;
```

[1] 注意文件名必须被截断，以处理DOS对文件名的限制。如果我们的系统支持长文件名，我们必须调整头文件名（否则只拷贝头文件）。

China-pub.com
下载


```
  cout << " buf2 = " << buf2 << endl;
  cout << s.rdbuf(); // Get the rest...
}
```

比起标准C库里atof()、atoi()等等这样的函数，可以看到，这是把字符串转换成类型值更加灵活和更加一般的途径。

编译器在下面的代码里处理这个串的静态存储分配：

```
istrstream s("1.414 47 This is a test");
```

我们还可以移交一个在栈或堆里分配的有零终止符的指针给它。

在s>>i>>f里，第一个数被提取到i，第二数被提取到f，这不是“字符的第一个空白分隔符”，这是因为它依赖于它正被提取的数据类型。例如，如果这个串被替换成“1.414 47 This is a test”，那么i取值1，这是因为输入程序停留在小数点上。然后 f取0.414。把一浮点数分成整数部分和小数部分，是有用的。否则看起来似乎是一个错误。

就像已猜测的一样，buf2没有取串的其余部分，仅取空白分隔符的下一个字。一般地，使用输入输出流提取符最好是当我们仅知道输入流里的确切的数据序列并且正在转换某些类型而不是一字符串。然而，如果我们想立即提取这个串的其余部分并把它送给另一个输入输出流，我们可以使用显示过的rdbuf（）。

输出strstreams也允许我们提供自己的存储空间。在这种情况下，字节在内存中被格式化。相应的构造函数是：

```
ostrstream::ostrstream(char*,int,int=ios::out);
```

第一个参数是预分配的缓冲区，在那里字符将结束，第二个参数是缓冲区的大小，第三个参数是模式。如果模式是缺省值，字符从缓冲区的开始地址格式化。如果模式是 ios::ate或ios::app（效果一样），字符缓冲区被假定已经包含了一个零终止字符串，而任何新的字符只能从零终止符开始添加。

第二个构造函数参数表示数组大小，且被对象用来保证它不覆盖数组尾。如我们已填满数组而又想添加更多的字节，这些字节是加不进去的。

关于ostrstreams，记住重要的是：没有为我们插入一般在字符数组末尾所需要的零终止符。当我们准备好零终止符时，用特别操纵算子ends。

一旦已建立一个ostrstream，就可以插入我们需要插入的任何东西，而且它将在内存缓冲区里完成格式化。下面是一个例子：

```
//: OSTRING.CPP -- Output strstreams
#include <strstrea.h>
#define SZ 100

main() {
cout << "type an int, a float and a string:";
int i;
float f;
cin >> i >> f;
cin >> ws; // Throw away white space
char buf[SZ];
cin.getline(buf, SZ); // Get rest of the line
```

China-pub.com 下载

```
  // (cin.rdbuf() would be awkward)
  ostrstream os(buf, SZ, ios::app);
  os << endl;
  os << "integer = " << i << endl;
  os << "float = " << f << endl;
  os << ends;
  cout << buf;
  cout << os.rdbuf(); // Same effect
  cout << os.rdbuf(); // NOT the same effect
}
```

这类似于前面int和float的例子。我们可能认为取一行其余部分的逻辑方法是使用 rdbuf（）；这个当然可以，但它很笨拙，因为所有包括回车的输入一直被收集起来，直到用户按control-Z(在unix中control-D)表明输入结束时才停下来。使用getline（）所表明的方法，一直取输入直到用户按下回车才停下来。这个输入被取进 buf，buf用来构造ostrstream os 。如果未提供第三个参数ios::app，构造函数缺省地写在buf的开头，覆盖刚被收集的行。然而，“追加”标志使它把被格式化后的信息放在这个串的末尾。

像其他的输出流一样，可以用平常的格式化工具发送字节到 ostrstream。区别是仅用ends在末尾插入零。注意，endl是在流中插入一个新行，而不是插入零。

现在信息在buf里格式化，可用cout<<buf直接发送它。然而，也有可能用os.rdbuf()发送它。当我们这样做的时候，在streambuf里的“取”指针随这字符被输出而向前移动。正因如此，第二次用到cout<<os.rdbuf()时，什么也没有发生——“取”指针已经在末端。

6.6.2 自动存储分配

输出strstreams（但不是istrstreams）是另一种分配存储空间的方法：它们自己完成这个操作，我们所做的是建立一个不带构造函数参数的ostrstream：

```
ostrstream A;
```

现在，A关心它自己在堆中存储空间的分配，可以在 A中想放多少字节就放多少字节，它用完存储空间，如有必要，它将移动存储块以分配更多的存储空间。

如果不知道需要多少空间，这是一个很好的解决办法，因为它很灵活。格式化数据到strstream，然后把它的streambuf移给另一个输入输出流。这样做很完美：

```
A << "hello, world. i = " << i << endl << ends;
cout << A.rdbuf();
```

这是所有解决办法中最好的。但是，如果要 A的字符被格式化到内存物理地址，会发生什么呢？这是很容易做到的——只要调用 str（）成员函数即可：

```
char* cp=A.str();
```

还有一个问题，如果想在 A中放进更多的字符会发生什么呢？如果我们知道 A分配的存储空间足够放进更多的字符，就好了，但那是不正确的。一般地，如给 A更多的字符，它将用完存储空间。通常 A试图在堆中分配更多的存储空间，这经常需要移动存储块。但是流对象刚刚把它的存储块的地址交给我们，所以我们不能很好地移动那个块，因为我们期望它处于特定的位置。

ostrstream处理这个问题的方法是“冻结”它自己。只要不用 str（）请求内部char*，就可

China-pub.com
下载

以尽可能向串输出流中追加字符。它将从堆中分配所需的存储空间，当对象脱离作用域时，堆存储空间自动被释放。

然而，如果调用str（），ostrstream就“冻结”了，就不能再向给它添加字符，无需通过实现来检测错误。向冻结的ostrstream添加字符导致未被定义的行为。另外，ostrstream不再负责清理存储器。当我们用str()请求char*时，要负责清除存储器。

为了不让内存泄漏，必须清理存储器。有两种清理办法。较普通的办法是直接释放要处理的内存。为搞懂这个问题，我们得预习一下C++中两个新的关键字：new和delete。就像第12章学到的一样，这两个关键字用得相当多，但目前可认为它们是用来替代C中malloc()和free()的。操作符new返回一个存储块，而delete释放它。重要的是必须知道它们，因为实际上C++中所有内存分配是由new完成的。ostrstream也是这样的。如果内存是由new分配的，它必须由delete释放。所以，如果有一个ostrstream A，用str()取得char*，清理存储器的办法是：

```
delete A.str();
```

这能满足大部分需要，但还有另一个不是很普通的释放存储器的办法：解冻ostrstream，可通过调用freeze()来做。freeze()是ostrstream的streambuf成员函数。freeze有一个缺省参数，这个缺省参数冻结这个流。用零参数对它解冻：

```
A.rdbuf()->freeze(0);
```

当A脱离作用域时，存储被重新分配，而且它的析构函数被调用。另外，可添加更多的字节给A。但是这可能引起存储移动，所以最好不要用以前通过调用str（）得到的指针——在添加更多的字符后，这个指针将不可靠。

下面的例子测试在一个流被解冻后追加字符的能力：

```
//: WALRUS.CPP -- Freezing a strstream
#include <strstrea.h>

main() {
  ostrstream s;
  s << "'The time has come', the walrus said,";
  s << ends;
  cout << s.str() << endl; // String is frozen
  // S is frozen; destructor won't delete
  // the streambuf storage on the heap
  s.seekp(-1, ios::cur); // Back up before NULL
  s.rdbuf()->freeze(0); // Unfreeze it
  // Now destructor releases memory, and
  // you can add more characters (but you
  // better not use the previous str() value)
  s << " 'To speak of many things'" << ends;
  cout << s.rdbuf();
}
```

在放第一个串到s后，添加一个ends，所以这个串能用由str()产生的char*打印出来。在这个意义上，s被冻结了。为了添更多的字节给s，“放”指针必须后移一步，这样下一个字符被放到由ends插入的零的上面。（否则，这个串仅被打印到原来的零的上面）。这是由seekp（）完成的。然后通过使用rdbuf()和调用freeze(0)取回基本streambuf指针，s被解冻。在这个意义上，

China-pub.com 下载

s就像它以前调用str()一样：我们可以添加更多的字符，清理由析构函数自动完成。

解冻一个ostrstream并继续添加字符是有可能的，但通常不这样做。正常的情况下，如果我们获得ostrstream的char*时想添加更多的字符，就建立一个新的ostrstream，通过使用rdbuf()把旧的流灌到这个新的流中去，并继续添加新的字符到新的ostrstream中。

1. 检验移动

如果我们仍不相信调用str()就得对ostrstream的存储空间负责，下面的例子说明存储定位被移动了，因而由str（）返回的旧指针是无效的：

```
//: STRMOVE.CPP -- Ostrstream memory movement
#include <strstrea.h>

main() {
  ostrstream s;
  s << "hi";
  char* old = s.str(); // Freezes s
  s.rdbuf()->freeze(0); // Unfreeze
  for(int i = 0; i < 100; i++)
    s << "howdy"; // Should force reallocation
  cout << "old = " << (void*)old << endl;
  cout << "new = " << (void*)s.str(); // Freezes
  delete s.str(); // Release storage
}
```

在插入一个串到s中并用str（）捕获char*后，这个串被解冻而且有足够的新字节被插入，真正确保了内存被重新分配且大多数被移动。在打印出旧的和新的 char*值后，存储明确地由delete释放，因为第二次调用str()又冻结了这个串。

为了打印出它们指向的串的地址而不是这个串，必须把 char*指派为void*。char*的操作符“ << ”打印出它正指向的串，而对应于void*的操作符“ << ”打印出指针的十六进制表示值。

有趣的是应注意到：在调用 str()前，如不插一个串到s中，结果则为0。这意味着直到第一次插入字节到ostrstream时，存储才被重新分配。

2. 一个更好的方法

标准C++ string类以及与其联系在一起工作的stringstream类，对解决这个问题做了很大的改进。使用这两个类代替char*和strstream时，不必担心负责存储空间的事——一切都被自动清理[1]。

6.7 输出流格式化

全部的努力以及所有这些不同类型的输入输出流的目的，是让我们容易地从一个地方到另一个地方移动并翻译字节。如果不能用 printf()族函数完成所有的格式化，这当然是没有用的。我们将学到输入输出流所有可用的输出格式化函数，得到所需要的那些字节。

输入输出流的格式化函数开始有点使人混淆，这是因为经常有一种以上的方式控制格式化：通过成员函数控制，也可以通过操纵算子来控制。更容易混淆的是，有一个派生的成员函数设置控制格式化的状态标志，如左对齐或右对齐，对十六进制表示法是否用大写字母，是否总是用十进制数表示浮点数的值等等。另一方面，这里有特别的成员函数用以设置填充字符、域宽

[1] 这本书中，这些类仅是草稿，不能在编译器上实现。

China-pub.com
下载

和精度，并读出它们的值。

6.7.1 内部格式化数据

ios类（在头文件IOSTREAM.H中可看到）包含数据成员以存储属于那个流的所有格式化数据。有些数据的值有一定范围并被储存在变量里：浮点精度、输出域宽度和用来填充输出（通常是一空格）的字符。格式化的其余部分是由标志所决定的，这些标志通常被连在一起以节省空间，并一起被指定为格式标志。可以用 ios::flags()成员函数发现格式化标志的值，这个成员函数没带参数并返回一个包含当前格式化标志的 long(typedefed to fmtflags)型值。函数的所有其余部分使格式化标志发生改变并返回格式化标志先前的值。

```
fmtflags ios::flags(fmtflags newflags);
fmtflags ios::setf(fmtflags ored_flag);
fmtflags ios::unsetf(fmtflags clear_flag);
fmtflags ios::setf(fmtflags bits, fmtflags field);
```

有时第一个函数迫使所有的标志改变。更多的是每次用剩下的三个函数来改变一个标志。

setf()的用法看来更加令人混淆：要想知道用哪个重载版本，必须知道正要改变的是哪类标志。这里有两类标志：一类是简单的 on或off，一类是与其他标志在一个组里工作的标志。on/off标志理解起来最简单，因为我们可用 setf(fmtflags)将它们变为on，用unsetf(fmtflags)将它们变为off。这些标志是：

on/off标志	作　　用
ios::skipws	跳过空白字符（对于输入这是缺省值）
ios::showbase	打印一个整数值时标明数值基数（十进制，八进制或十六进制），使用的格式能被C++编译器读出
ios::showpoint	表明浮点数的小数点和后面的零
ios::uppercase	显示十六进制数值的大写字母 A-F和科学记数法中的大写字母 E
ios::showpos	显示加号（+）代表正值
ios::unitbuf	“设备缓冲区”；在每次插入操作后，这个流被刷新
ios::stdio	使这个流与C标准I/O系统同步

例如，为 cout显示加号，可写成 cout.setf(ios::showpos)；停止显示加号，可写成 cout.unsetf(ios::showpos)。应该解释一下最后两个标志。当一个字符一旦被插进一个输出流，如果想确信它是一个输出时，可启用缓冲设备。也可以不用缓冲输出，但用缓冲设备会更好。

有一个程序用了输入输出流和C标准I/O库（用C库不是不可能的），标志ios::stdio就被采用。如果发现输入输出流的输出和printf()输出出现了错误的次序，就要设置这个标志。

1. 格式域

第二类格式化标志在一个组里工作，一次只能用这些标志中的一种，就像旧式的汽车收音机按钮一样 —— 按下一个按钮，其余的弹出。可惜的是，这是不能自动发生的，我们必须注意正在设置的是什么标志，这样就不会偶然调用错误的 setf（）函数。例如，每一个数字基数有一个标志：十六进制，十进制和八进制。这些标志一起被指定为 ios::basefield。如果ios：：dec标志被设置而调用setf（ios::hex），将设置ios::hex标志，但不会清除ios::dec位，结果出现未被定义的方式。适当的方法是像这样调用 setf()的第二种形式：setf(ios::hex,ios::basefield)。这个函数首先清除ios::basefield里的所有位，然后设置ios::hex。这样，setf()的这个形式保证无论什么时候设置一个标志，这个组里的其他标志都会“弹出”。当然，所有这些操作由hex()操纵算子自动完成。所以不必了解这个类实现的内部细节，甚至不必关心它是一个二进制标志的设置。

China-pub.com 下载

以后将会看到有一个与set()有提供同样的功能操纵算子。

下面是标志组和它们的作用：

iso::basefield	作　　用
ios::dec	十进制格式整数值（十进制）（缺省基数）
ios::hex	十六进制格式整数值（十六进制）
ios::oct	八进制格式整数值（八进制）

ios::floatfield	作　　用
ios::scientific	科学记数法表示浮点数值，精度域指小数点后面的数字数目
ios::fixed	定点格式表示浮点数值，精度域指小数点后面的数字数目
“automatic”(Neither bit is set)	精度域指整个有效数字的数目

ios::adjustfield	作　　用
ios::left	左对齐，向右边填充字符
ios::right	右对齐，向左边填充字符
ios::internal	在任何引导符或基数指示符之后但在值之前添加填充字符

2. 域宽、填充字符和精度

一些内部变量，用于控制输出域的宽度，或当数据没有填入时，用作填充的字符，或控制打印浮点数的精度。它被与变量同名字的成员函数读和写。

function	作　　用
int ios::width()	读当前宽度（缺省值为0），用于插入和提取
int ios::width(int n)	设置宽度，返回以前的宽度
int ios::fill()	读当前填充字符（缺省值为空格）
int ios::fill(int n)	设置填充字符，返回以前的填充字符
int ios::precision()	读当前浮点数精度（缺省值为6）
int ios::precision(int n)	设置浮点精度，返回以前的精度；“精度”含义见 ios::floatfield表

填充和精度值是相当直观的，但宽度值需要一些解释。当宽度为0时，插入一个值将产生代表这个值所需字符的最小数。一个正值的宽度意味着插入一个值将产生至少与宽度一样多的字符。假如值小于字符宽度，填充字符用来填这个域。然而，这个值决不被截断。所以，如果打印123而宽度为2，我们仍将得到123。域宽标识了字符的最小数目。没有标识字符最大数目的办法。

宽度也是明显不同的，因为每个插入符或提取符可能受到它的值的影响，它被每个插入符或提取符重新设置为0。它不是一个真正的静态变量，而是插入符和提取符的一个隐含参数。如我们想有一个恒定的宽度，得在每一个插入或提取它之后调用width()。

6.7.2 例子

为了确信懂得怎样调用以前讨论过的所有函数，下面举一个全部调用这些函数的例子：

```
//: FORMAT.CPP -- Formatting functions
#include <fstream.h>
#define D(a) T << #a << endl; a
ofstream T("format.out");

main() {
```

China-pub.com
下载


```
D(int i = 47;)
D(float f = 2300114.414159;)
char* s = "Is there any more?";

D(T.setf(ios::unitbuf);)
D(T.setf(ios::stdio);)

D(T.setf(ios::showbase);)
D(T.setf(ios::uppercase);)
D(T.setf(ios::showpos);)
D(T << i << endl;) // Default to dec
D(T.setf(ios::hex, ios::basefield);)
D(T << i << endl;)
D(T.unsetf(ios::uppercase);)
D(T.setf(ios::oct, ios::basefield);)
D(T << i << endl;)
D(T.unsetf(ios::showbase);)
D(T.setf(ios::dec, ios::basefield);)
D(T.setf(ios::left, ios::adjustfield);)
D(T.fill('0');)
D(T << "fill char: " << T.fill() << endl;)
D(T.width(10);)
T << i << endl;
D(T.setf(ios::right, ios::adjustfield);)
D(T.width(10);)
T << i << endl;
D(T.setf(ios::internal, ios::adjustfield);)
D(T.width(10);)
T << i << endl;
D(T << i << endl;) // Without width(10)

D(T.unsetf(ios::showpos);)
D(T.setf(ios::showpoint);)
D(T << "prec = " << T.precision() << endl;)
D(T.setf(ios::scientific, ios::floatfield);)
D(T << endl << f << endl;)
D(T.setf(ios::fixed, ios::floatfield);)
D(T << f << endl;)
D(T.setf(0, ios::floatfield);) // Automatic
D(T << f << endl;)
D(T.precision(20);)
D(T << "prec = " << T.precision() << endl;)
D(T << endl << f << endl;)
D(T.setf(ios::scientific, ios::floatfield);)
D(T << endl << f << endl;)
D(T.setf(ios::fixed, ios::floatfield);)
```

China-pub.com 下载

```
  D(T << f << endl;)
  D(T.setf(0, ios::floatfield);) // Automatic
  D(T << f << endl;)

  D(T.width(10);)
  T << s << endl;
  D(T.width(40);)
  T << s << endl;
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.width(40);)
  T << s << endl;

  D(T.unsetf(ios::showpoint);)
  D(T.unsetf(ios::unitbuf);)
  D(T.unsetf(ios::stdio);)
}
```

这个例子使用技巧建立一个跟踪文件，这样我们就能监控所发生的事情。宏D（a）使用预处理器“ stringizing ”把a转变为一个串打印出来。然后，宏D（a）反复处理a，使a产生作用。宏发送所有的信息到一个叫作T的文件中，这就是那个跟踪文件。输出是：

```
int i = 47;
float f = 2300114.414159;
T.setf(ios::unitbuf);
T.setf(ios::stdio);
T.setf(ios::showbase);
T.setf(ios::uppercase);
T.setf(ios::showpos);
T << i << endl;
+47
T.setf(ios::hex, ios::basefield);
T << i << endl;
+0X2F
T.unsetf(ios::uppercase);
T.setf(ios::oct, ios::basefield);
T << i << endl;
+057
T.unsetf(ios::showbase);
T.setf(ios::dec, ios::basefield);
T.setf(ios::left, ios::adjustfield);
T.fill('0');
T << "fill char: " << T.fill() << endl;
fill char: 0
T.width(10);
+470000000
T.setf(ios::right, ios::adjustfield);
T.width(10);
```

China-pub.com
下载


```
0000000+47
T.setf(ios::internal, ios::adjustfield);
T.width(10);
+000000047
T << i << endl;
+47
T.unsetf(ios::showpos);
T.setf(ios::showpoint);
T << "prec = " << T.precision() << endl;
prec = 6
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.300115e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.500000
T.setf(0, ios::floatfield);
T << f << endl;
2.300115e+06
T.precision(20);
T << "prec = " << T.precision() << endl;
prec = 20
T << endl << f << endl;

2300114.50000000020000000000
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.30011450000000020000e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.50000000020000000000
T.setf(0, ios::floatfield);
T << f << endl;
2300114.50000000020000000000
T.width(10);
Is there any more?
T.width(40);
0000000000000000000000Is there any more?
T.setf(ios::left, ios::adjustfield);
T.width(40);
Is there any more?0000000000000000000000
T.unsetf(ios::showpoint);
T.unsetf(ios::unitbuf);
T.unsetf(ios::stdio);
```

China-pub.com 下载

研究这个输出会使我们清楚地理解输入输出流格式化成员函数。

6.8 格式化操纵算子

就像我们在前面的例子中看到的一样，调用成员函数有点乏味。为使读和写更容易， C++提供了一套操纵算子以起到与成员函数同样的作用。

提供在IOSTREAM.H里的是不带参数的操纵算子。这些操纵算子包括 dec、oct和hex。它们各自更简明扼要地完成与 setf(ios::dec,ios::basefield)、setf(ios::oct,ios::basefield)和setf(ios::hex,ios::basefield)同样的任务。IOSTREAM.H [1] 还包括ws、endl、ends和flush以及如下所示的其他操纵算子：

操纵算子	作　　用
showbase	在打印一整数值时，标明数字基数（十进制，八进制和十六进制）；所用的格式能被C++编译器读出
noshowbase	
showpos	显示正值符号加（+）
noshowpos	
uppercase	显示代表十六进制值的大写字母A-F以及科学记数法中的E
nouppercase	
showpoint	表明浮点数值的小数点和后面的零
noshowpoint	
skipws	跳过输入中的空白字符
noshipws	
left	左对齐，右填充
right	右对齐，左填充
internal	在引导符或基数指示符和值之间填充
scientific	使用科学记数法
fixed	setprecision()或ios::precision()设置小数点后面的位数

带参数的操纵算子

如果正在使用带参数的操纵算子，必须也包含头文件IOMANIP.H。这包含了解决建立带参数操纵算子所遇到的一般问题的代码。另外，它有六个预定义的操纵算子：

操纵算子	作　　用
setiosflags(fmtflags n)	设置由n指定的格式标志；设置一直起作用直到下一个变化为止，像ios::setf()一样
resetiosflags(fmtflags n)	清除由n指定的格式标志。设置一直起作用直到下一个变化为止，像ios::unsetf()一样
setbase(base n)	把基数改成n，这里n取10、8或16（任何别的值结果为0）。如果n是0，输出基数为10，但输入使用C约定：10是10，010是8而0xf是15。我们还是使用dec、oct和hex输出为好
setfill(char n)	把填充字符改成n，像ios∷fill（）一样
setprecision(int n)	把精度改成n，像ios∷precision（）一样
setw(int n)	把域宽改成n，像ios∷width（）一样

如果正在使用很多的插入符，我们会看到这是怎样清理的。作为一个例子，下面是用操纵

[1] 这些仅在修改库中出现，老的输入输出流实现中没包括它们。

China-pub.com
下载

算子对前面程序的重写（宏已被去掉以使其更容易阅读）：

```
//: MANIPS.CPP -- FORMAT.CPP using manipulators
#include <fstream.h>
#include <iomanip.h>

main() {
  ofstream T("trace.out");
  int i = 47;
  float f = 2300114.414159;
  char* s = "Is there any more?";

  T << setiosflags(
         ios::unitbuf | ios::stdio
         | ios::showbase | ios::uppercase
         | ios::showpos
       );
  T << i << endl; // Default to dec
  T << hex << i << endl;
  T << resetiosflags(ios::uppercase)
    << oct << i << endl;
  T.setf(ios::left, ios::adjustfield);
  T << resetiosflags(ios::showbase)
    << dec << setfill('0');
  T << "fill char: " << T.fill() << endl;
  T << setw(10) << i << endl;
  T.setf(ios::right, ios::adjustfield);
  T << setw(10) << i << endl;
  T.setf(ios::internal, ios::adjustfield);
  T << setw(10) << i << endl;
  T << i << endl; // Without setw(10)

  T << resetiosflags(ios::showpos)
    << setiosflags(ios::showpoint)
    << "prec = " << T.precision() << endl;
  T.setf(ios::scientific, ios::floatfield);
  T << f << endl;
  T.setf(ios::fixed, ios::floatfield);
  T << f << endl;
  T.setf(0, ios::floatfield); // Automatic
  T << f << endl;
  T << setprecision(20);
  T << "prec = " << T.precision() << endl;
  T << f << endl;
  T.setf(ios::scientific, ios::floatfield);
  T << f << endl;
  T.setf(ios::fixed, ios::floatfield);
```

China-pub.com 下载

```
  T << f << endl;
  T.setf(0, ios::floatfield); // Automatic
  T << f << endl;

  T << setw(10) << s << endl;
  T << setw(40) << s << endl;
  T.setf(ios::left, ios::adjustfield);
  T << setw(40) << s << endl;

  T << resetiosflags(
         ios::showpoint | ios::unitbuf
         | ios::stdio
       );
}
```

许多的多重语句已被精简成单个的链插入。注意调用 setiosflags()和resetiosflags()，在这两个函数里，标志被按“位OR”运算成一个。在前面的例子里，这个工作是由 setf()和unsetf()完成的。

6.9 建立操纵算子

（注意：这部分包含一些以后章节中才介绍的内容）有时，我们可能想建立自己的操纵算子，这是相当简单的。一个像endl这样的不带参数的操纵算子只是一个函数，这个函数把一个ostream引用作为它的参数。（引用是一种不同的参数传送方式，在第 10章中讨论）对endl的声明是：

```
ostream& endl(ostream&)；
```

现在，当我们写：

```
cout<<"howdy" <<endl；
```

endl产生函数的地址。这样，编译器问：“有能被我调用的把函数的地址作为它的参数的函数吗？”确实有一个这样的函数，是在IOSTREAM.H里预先定义的函数；它被称作“应用算子”。这个应用算子调用这个函数，把ostream对象作为一个参数传送给这个函数。

不必知道应用算子怎样建立我们自己的操纵算子；我们只要知道应用算子存在就行了。下面是建立一个操纵算子的例子，这个操纵算子叫nl，它产生一个换行而不刷新这个流：

```
//: NL.CPP -- Creating a manipulator
#include <iostream.h>

ostream& nl(ostream& os) {
  return os << '\n';
}

main() {
  cout << "newlines" << nl << "between" << nl
       << "each" << nl << "word" << nl;
}
```

China-pub.com
下载

表达式

```
os<<'\n';
```

调用一个返回os的函数，它是从nl中返回的[1]。

人们经常认为，如上所示的nl比使用endl要好，因为后者总是清空输出流，这可能引起执行故障。

效用算子

正如我们已看到的，零参数操纵算子相当容易建立。但是如果建立带参数的操纵算子又怎样呢？要这样做，输入输出流有一个相当麻烦且易混淆的方法。但是 Jerry Schwarz，输入输出流库的建立者，提出一个方案[2]，他称之为效用算子。一个效用算子是一个简单的类，这个类的构造函数与工作在这个类里的一个重载操作符“ << ”一起执行想要的操作。下面是一个有两个效用算子的例子。第一个输出是一个被截断的字符串，第二个打印出一个二进制数（定义一个重载操作符“ << ”的过程将在第11章讨论）：

```
//: EFFECTOR.CPP -- Jerry Schwarz's "effectors"
#include<iostream.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <limits.h> // ULONG_MAX

// Put out a portion of a string:
class fixw {
  char* s;
public:
  fixw(const char* S, int width);
  ~fixw();
  friend ostream& operator<<(ostream&, fixw&);
};

fixw::fixw(const char* S, int width) {
  s = (char*)malloc(width + 1);
  assert(s);
  strncpy(s, S, width);
  s[width] = 0; // Null-terminate
}

fixw::~fixw() { free(s); }

ostream& operator<<(ostream& os, fixw& fw) {
  return os << fw.s;
}
```

[1] 把n1放入头文件之前，应该把它变成内联函数（见第8章）。

[2] 在私人谈话中。

China-pub.com 下载

```
// Print a number in binary:
typedef unsigned long ulong;

class bin {
  ulong n;
public:
  bin(ulong N);
  friend ostream& operator<<(ostream&, bin&);
};

bin::bin(ulong N) { n = N; }

ostream& operator<<(ostream& os, bin& b) {
  ulong bit = ~(ULONG_MAX >> 1); // Top bit set
  while(bit) {
    os << (b.n & bit ? '1' : '0');
    bit >>= 1;
  }
  return os;
}
main() {
  char* string =
    "Things that make us happy, make us wise";
  for(int i = 1; i <= strlen(string); i++)
    cout << fixw(string, i) << endl;
  ulong x = 0xFEDCBA98UL;
  ulong y = 0x76543210UL;
  cout << "x in binary: " << bin(x) << endl;
  cout << "y in binary: " << bin(y) << endl;
}
```

fixw的构造函数产生char*参数的一个缩短的副本，析构函数释放产生这个副本的内存。重载操作符“ << ”取第二个参数的内容，即fixw对象，并把它插入第一个参数ostream，然后返回ostream，所以它可以用在一个链接表达式里。下面的表达式里用fixw时：

```
cout <<fixw(string,i)<<endl；
```

一个临时对象通过调用fixw构造函数被建立了，那个临时对象被传送给操作符“ << ”。带参数的操纵算子产生作用了。

bin 效用算子依赖于这样的事实：对一个无符号数向右移位，把零移成高位。 ULONG_MAX（最大的长型值unsigned long，来自标准包含文件LIMITS.H）用来产生一个带高位设置的值，这个值移过正被讨论的数（通过移位），屏蔽每一位。

起初，这个技术上的问题是：一旦为char*建立一个叫fixw的类，或为unsign long建立一个叫bin的类，没有别的人能为他们的类型建立一个不同的 fixw类或bin类。然而，有了名字空间（在第9章），这个问题被解决了。

China-pub.com
下载

6.10 输入输出流实例

本节，将看到怎样处理本章学到的所有知识的一些例子。虽然存在能操纵字节的很多工具（像unix中的sed和awk这样的流编辑器可能是广为人知的，而文本编辑器也属于此类），但它们一般都有一些限制。sed和awk可能比较慢，而且仅能处理向前序列里的行。文本编辑器通常需要人的交互作用或至少要学一种专有的宏语言。用输入输出流写的程序就没有任何此类限制：这些程序运行快，可移植而且很灵活。输入输出流是工具箱里的非常有用的一个工具。

6.10.1 代码生成

第一个例子涉及程序的产生，比较巧的是，这个程序也符合本书的格式。在开发代码时保证供快速和连贯性。第一个程序建立一个文件保存 main（）（假定它带有非命令行参数并且使用输入输出流库）：

```
//: MAKEMAIN.CPP -- Create a shell main() File
#include <fstream.h>
#include <strstrea.h>
#include <assert.h>
#include <string.h>
#include <ctype.h>

main(int argc, char* argv[]) {
  assert(argc == 2);
  // Don't replace it if it exists:
  ofstream mainfile(argv[1], ios::noreplace);
  assert(mainfile);
  istrstream name(argv[1]);
  ostrstream CAPname;
  char c;
  while(name.get(c))
    CAPname << char(toupper(c));
  CAPname << ends;
  mainfile << "//:"  << ' ' << CAPname.rdbuf()
    << " -- " << endl
    << "#include <iostream.h>" << endl
    << endl
    << "main() {" << endl << endl
    << "}" << endl;
}
```

这个文件被打开，使用ios::noreplace，以保证不会偶然地覆盖一个现成文件。然后用命令行中的那个参数来建立一个 istrstream。这样，字符每次一个地被提取。有了标准 C库宏toupper()，这些字符可被转变成大写字母。这个变量返回一个 int，这样，它必须很明确地转换给char。此名字用在头一行里，后跟产生文件的余项。

1. 维护类库资源

第二个例子执行一个更复杂和更有用的任务。总之，建立一个类时，要想想库术语，而且

China-pub.com 下载

要在类声明中创建一个头文件NAME.H和一个名叫NAME.CPP的文件——成员函数在这个文件里实现。这些文件有一定的要求：一个特别的代码标准（这儿显示的程序是用这本书里的代码格式），而且这个头文件的声明一般被一些预处理器说明所包围，目的是避免类的多重说明。（多重说明使编译器混淆——编译器不知道我们要使用哪个类。这些类可能是不同的，所以编译器发出一个出错信息）。

这个例子建立一对新的头——实现文件，不然就修改现存的文件。如果这对文件已存在，它检查这对文件并潜在地修改这对文件，但是如果这对文件不存在，它用合适的格式建立这对文件：

```
//: CPPCHECK.CPP -- Configures .H & .CPP files
// To conform to style standard.
// Tests existing files for conformance
#include <fstream.h>
#include <strstrea.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>
#define SZ 40  // Buffer sizes
#define BSZ 100

main(int argc, char* argv[]) {
  assert(argc == 2); // File set name
  enum bufs { base, header, implement,
    Hline1, guard1, guard2, guard3,
    CPPline1, include, bufnum };
  char b[bufnum][SZ];
  ostrstream osarray[] = {
    ostrstream(b[base], SZ),
    ostrstream(b[header], SZ),
    ostrstream(b[implement], SZ),
    ostrstream(b[Hline1], SZ),
    ostrstream(b[guard1], SZ),
    ostrstream(b[guard2], SZ),
    ostrstream(b[guard3], SZ),
    ostrstream(b[CPPline1], SZ),
    ostrstream(b[include], SZ),
  };
  osarray[base] << argv[1] << ends;
  // Find any '.' in the string using the
  // Standard C library function strchr():
  char* period = strchr(b[base], '.');
  if(period) *period = 0; // Strip extension
  // Force to upper case:
  for(int i = 0; b[base][i]; i++)
    b[base][i] = toupper(b[base][i]);
  // Create file names and internal lines:
```

China-pub.com
下载


```
osarray[header] << b[base] << ".H" << ends;
osarray[implement] << b[base] << ".CPP" << ends;
osarray[Hline1] << "//:" << ' ' << b[header]
  << " -- " << ends;
osarray[guard1] << "#ifndef " << b[base]
                << "_H_" << ends;
osarray[guard2] << "#define " << b[base]
                << "_H_" << ends;
osarray[guard3] << "#endif // " << b[base]
                << "_H_" << ends;
osarray[CPPline1] << "//:" << ' '
                  << b[implement]
                  << " -- " << ends;
osarray[include] << "#include \""
                 << b[header] << "\"" <<ends;
// First, try to open existing files:
ifstream existh(b[header]),
         existcpp(b[implement]);
if(!existh) { // Doesn't exist; create it
  ofstream newheader(b[header]);
  newheader << b[Hline1] << endl
    << b[guard1] << endl
    << b[guard2] << endl << endl
    << b[guard3] << endl;
}
if(!existcpp) { // Create cpp file
  ofstream newcpp(implement);
  newcpp << b[CPPline1] << endl
    << b[include] << endl;
}
if(existh) { // Already exists; verify it
  strstream hfile; // Write & read
  ostrstream newheader; // Write
  hfile << existh.rdbuf() << ends;
  // Check that first line conforms:
  char buf[BSZ];
  if(hfile.getline(buf, BSZ)) {
    if(!strstr(buf, "//:") ||
      !strstr(buf, b[header]))
     newheader << b[Hline1] << endl;
  }
  // Ensure guard lines are in header:
  if(!strstr(hfile.str(), b[guard1]) ||
     !strstr(hfile.str(), b[guard2]) ||
     !strstr(hfile.str(), b[guard3])) {
```

China-pub.com 下载

```
      newheader << b[guard1] << endl
        << b[guard2] << endl
        << buf
        << hfile.rdbuf() << endl
        << b[guard3] << endl << ends;
    } else
      newheader << buf
        << hfile.rdbuf() << ends;
    // If there were changes, overwrite file:
    if(strcmp(hfile.str(),newheader.str())!=0){
      existh.close();
      ofstream newH(b[header]);
      newH << "//@//" << endl // change marker
        << newheader.rdbuf();
    }
    delete hfile.str();
    delete newheader.str();
  }
  if(existcpp) { // Already exists; verify it
    strstream cppfile;
    ostrstream newcpp;
    cppfile << existcpp.rdbuf() << ends;
    char buf[BSZ];
    // Check that first line conforms:
    if(cppfile.getline(buf, BSZ))
      if(!strstr(buf, "//:") ||
         !strstr(buf, b[implement]))
        newcpp << b[CPPline1] << endl;
    // Ensure header is included:
    if(!strstr(cppfile.str(), b[include]))
        newcpp << b[include] << endl;
      // Put in the rest of the file:
      newcpp << buf << endl; // First line read
      newcpp << cppfile.rdbuf() << ends;
      // If there were changes, overwrite file:
      if(strcmp(cppfile.str(),newcpp.str())!=0){
        existcpp.close();
        ofstream newCPP(b[implement]);
        newCPP << "//@//" << endl // change marker
          << newcpp.rdbuf();
      }
      delete cppfile.str();
      delete newcpp.str();
    }
  }
```

China-pub.com
下载

这个例子需要不同缓冲区中的串格式化。不是建立单个命名的缓冲区和 ostrstream对象，而是在enum bufs缓冲区中建立 一组名字。因而要建立两个数组：一个字符缓冲区数组和一个从字符缓冲区里建立的ostrstream对象数组。注意在char缓冲区b的二维数组定义里， char数组的数目是由bufnum决定的， bufnum是bufs中的最后一个枚举常量。当建立一个枚举变量时，编译器赋一个整数值给所有那些标明从零开始的 enum，所以bufnum的唯一目的是成为统计buf中枚举常量数目的计数器。b中每一个串的长度是SZ。

枚举变量里的名字是： base，即大写字母的不带扩展名的基文件名； header，即头文件名；implement，即实现文件名（CPP）；Hline1，即头文件第一行框架； guard1 、guard2和guard3，即头文件里的“保护”行（阻止多重包含）； CPPlinel，即CPP文件的第一行框架；include，即包含头文件的CPP文件的行。

osarray是一个通过集合初始化和自动计数建立起来的 ostrstream对象数组。当然，这是带两个参数（缓冲区地址和大小）形式的 ostrstream构造函数，所以构造函数调用必须相应地建立在集合初始化表里。利用 bufs枚举常量， b的适当数组元素结合到相应的 osarray对象。一旦数组被建立，利用枚举常量就可选择数组里的对象，作用是填充相应的 b元素。我们可看到，每个串是怎样建立在ostrstream数组定义后面的行里的。

一旦串被建立，程序试图打开头文件和 CPP文件的现行版本作为ifstreams。如果用操作符“！”测试对象而这个文件不存在，测试将失败。如果头文件或实现文件不存在，利用以前建立的文本的适当的行来建立它。

如果文件确实存在，那么这些文件后面跟着适当的格式，这是有保证的。在这两种情况下，一个strstream被建立而且整个文件被读进；然后第一行被读出并被检查，看看这一行是否包含一个“//:”和文件名以确信它跟在格式的后面。这是由标准 C库函数strstr()完成的。如果第一行不符合，较早时建立的内容被插进一个已建好的 ostrstream中，目的是保存被编辑的那个文件。

在头文件里，整个文件被搜索（再次使用 strstr()函数）以确保它包含三个“保护”行；如果没有包含这三行，要插入它们。检查实现文件，看看包含头文件那一行是否存在（虽然编译器有效地保证它的存在）。

在两种情况下，比较原始文件（在它的 strstream里）和被编辑的文件（在 ostrstream里），看看它们是否有变化。如果有变化，现存文件被关闭， 一个新的 ofstream对象被建立，目的是覆盖这个现存文件。一个特别的变化标志被添加到开始处之后， ostrstream被输出到这个文件，所以可使用一文本搜索程序，通过检查快速发现产生另外变化的文件。

2. 检测编译器错误

这本书里的所有代码都已经设计好了，没有编译错误。任何产生编译错误的代码行都由特别注释顺序符“//！”注解出来。下面的程序将移去这些特别注解并在原处添加一个已编号的注解。这样，运行编译器时，它应该产生出错信息。当编译所有文件时，应该看到所有的号码。它也可以添加修改过的行到一个特别文件里，这样可容易定位任何不产生错误的行：

```
//: SHOWERR.CPP -- Un-comment error generators
#include <fstream.h>
#include <strstrea.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>
```

China-pub.com 下载

```
char* marker = "//!";

char* usage =
"usage: showerr filename chapnum\n"
"where filename is a C++ source file\n"
"and chapnum is the chapter name it's in.\n"
"Finds lines commented with //! and removes\n"
"comment, appending //(#) where # is unique\n"
"across all files, so you can determine\n"
"if your compiler finds the error.\n"
"showerr /r\n"
"resets the unique counter.";

// File containing error number counter:
char* errnum = "..\\errnum.txt";
// File containing error lines:
char* errfile = "..\\errlines.txt";
ofstream errlines(errfile,ios::app);

main(int argc, char* argv[]) {
  if(argc < 2) {
    cerr << usage << endl;
    return 1;
  }
  if(argv[1][0] == '/' || argv[1][0] == '-') {
    // Allow for other switches:
    switch(argv[1][1]) {
      case 'r': case 'R':
        cout << "reset counter" << endl;
        remove(errnum); // Delete files
        remove(errfile);
        return 0;
      default:
        cerr << usage << endl;
        return 1;
    }
  }
  char* chapter = argv[2];
  strstream edited; // Edited file
  int counter = 0;
  {
    ifstream infile(argv[1]);
    assert(infile);
    ifstream count(errnum);
    if(count) count >> counter;
    int linecount = 0;
```

China-pub.com
下载


```
    #define sz 255
    char buf[sz];
    while(infile.getline(buf, sz)) {
      linecount++;
      // Eat white space:
      int i = 0;
      while(isspace(buf[i]))
      i++;
    // Find marker at start of line:
    if(strstr(&buf[i], marker) == &buf[i]) {
      // Erase marker:
      memset(&buf[i], ' ', strlen(marker));
      // Append counter & error info:
      ostrstream out(buf, sz, ios::ate);
      out << "//(" << ++counter << ") "
          << "Chapter " << chapter
          << " File: " << argv[1]
          << " Line " << linecount << endl
          << ends;
        edited << buf;
      errlines << buf; // Append error file
    } else
      edited << buf << "\n"; // Just copy
  }
 } // Closes files
 ofstream outfile(argv[1]); // Overwrites
 outfile << edited.rdbuf();
 ofstream count(errnum); // Overwrites
 count << counter; // Save new counter
}
```

这个maker指针可被我们的一种选择所代替。

每个文件每次读出一行，在每一行中搜索出现在行开头的 maker。这个行被修改并放进错误行表、放进strstream edited里。当整个文件被处理完时，它被关闭（通过到达范围的末端），重新作为一个输出文件打开， edited被灌进这个文件。注意计数器被保存在内部文件里。所以下一次调用这个程序时，继续由计数器按顺序计数。

6.10.2 一个简单的数据记录

这个例子显示了记录数据到磁盘上而后检索数据并作处理的一种途径。这个例子的意思是产生一个海洋各处温度——深度的曲线图。为保存数据，要用到一个类：

```
//: DATALOG.H -- Datalogger record layout
#ifndef DATALOG_H_
#define DATALOG_H_
#include <time.h>
#include <iostream.h>
```

China-pub.com 下载

```
#define BSZ 10

class datapoint {
  tm Tm; // Time & day
  // Ascii degrees (*) minutes (') seconds ("):
  char Latitude[BSZ], Longitude[BSZ];
  double Depth, Temperature;
public:
  tm Time(); // read the time
  void Time(tm T); // set the time
  const char* latitude(); // read
  void latitude(const char* l); // set
  const char* longitude(); // read
  void longitude(const char* l); // set
  double depth(); // read
  void depth(double d); // set
  double temperature(); // read
  void temperature(double t); // set
  void print(ostream& os);
};
#endif // DATALOG_H_
```

这个存取函数为每个数据成员提供受控制的读和写。 printf()函数用一个可读的形式格式化datapoint到一个ostream对象中（printf()的参数），下面是一个定义文件：

```
//: DATALOG.CPP -- Datapoint member functions
#include "..\5\datalog.h"
#include <iomanip.h>
#include <string.h>

tm datapoint::Time() { return Tm; }

void datapoint::Time(tm T) { Tm = T; }

const char* datapoint::latitude() {
  return Latitude;
}

void datapoint::latitude(const char* l) {
  Latitude[BSZ - 1] = 0;
  strncpy(Latitude, l, BSZ - 1);
}

const char* datapoint::longitude() {
  return Longitude;
}
```

China-pub.com
下载


```
void datapoint::longitude(const char* l) {
  Longitude[BSZ - 1] = 0;
  strncpy(Longitude, l, BSZ - 1);
}

double datapoint::depth() { return Depth; }

void datapoint::depth(double d) { Depth = d; }

double datapoint::temperature() {
  return Temperature;
}

void datapoint::temperature(double t) {
  Temperature = t;
}

void datapoint::print(ostream& os) {
  os.setf(ios::fixed, ios::floatfield);
  os.precision(4);
  os.fill('0'); // Pad on left with '0'
  os << setw(2) << Time().tm_mon << '\\'
     << setw(2) << Time().tm_mday << '\\'
     << setw(2) << Time().tm_year << ' '
     << setw(2) << Time().tm_hour << ':'
     << setw(2) << Time().tm_min << ':'
     << setw(2) << Time().tm_sec;
  os.fill(' '); // Pad on left with ' '
  os << " Lat:" << setw(9) << latitude()
     << ", Long:" << setw(9) << longitude()
     << ", depth:" << setw(9) << depth()
     << ", temp:" << setw(9) << temperature()
     << endl;
}
```

在printf()函数里，调用setf()引起浮点数以定点精度的形式输出，而精度函数 precision()设置4位小数。

域里的数据缺省向右对齐，时间信息由时、分和秒各两位数字组成，这样。在每种情况下，宽度由setw()函数设置为2。（记住域宽的任何变化影响下次输出操作，所以setw()函数必须配给每个输出）。但是首先如果值小于10，填充字符被设置为“ 0 ”，在值的左边放一个零。以后它又被设置为空格。

纬度和经度是零终止符域，它保存度（这里‘ * ’表示度）、分（`）和秒（``）的信息。如我们愿意的话，当然能设计出一个更有效的存储方案。

1. 生成测试数据

下面是一个程序，这个程序（用 write()）建立一个二进制形式的测试数据文件 ,并且用

China-pub.com 下载

datapoint::print()建立另一个ASCⅡ形式的文件。我们也可以把它打印到屏幕上，但是在文件形式里更容易检测：

```
//: DATAGEN.CPP -- Test data generator
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include "..\5\datalog.h"

main() {
  ofstream data("data.txt");
  ofstream bindata("data.bin", ios::binary);
  time_t timer = time(NULL);  // Get time
  // Seed random generator:
  srand((unsigned)timer);
  for(int i = 0; i < 100; i++) {
    datapoint d;
    // Convert date/time to a structure:
    d.Time(*localtime(&timer));
    timer += 55; // Reading each 55 seconds
    d.latitude("45*20'31\"");
    d.longitude("22*34'18\"");
    // Zero to 199 meters:
    double newdepth  = rand() % 200;
    double fraction = rand() % 100 + 1;
    newdepth += double(1) / fraction;
    d.depth(newdepth);
    double newtemp = 150 + rand()%200; //Kelvin
    fraction = rand() % 100 + 1;
    newtemp += (double)1 / fraction;
    d.temperature(newtemp);
    d.print(data);
    bindata.write((unsigned char*)&d,
                  sizeof(d));
  }
}
```

文件DATA.TXT作为一个ASCⅡ文件用通常的方法建立了。而DATA.BIN有标志ios::binary，用以告诉构造函数把它设置成二进制文件。

标准C库函数time(),当带一个零参数调用它时，返回当前时间为一个time_t值，它是自1970年1月1日00：00：00 GMT后的秒数。当前时间是用标准C库函数scrand()为随机数产生器设置种子的最方便的方法，这里就是这样做的。

有时候，存储时间的更方便的方法是一个tm结构，它含有时间和日期的每个元素，而时间和日期被分成如下所示的构成成分：

```
struct tm {
   int tm_sec; // 0-59 seconds
```

China-pub.com
下载


```
    int tm_min; // 0-59 minutes
    int tm_hour; // 0-23 hours
    int tm_mday; // day of month
    int tm_mon; // 1-12 months
    int tm_year; // calendar year
    int tm_wday; // Sunday == 0, etc.
    int tm_yday; // 0-365 day of year
    int tm_isdst; // daylight savings?
};
```

为把以秒计数的时间转换成tm格式的本地时间，利用标准C库localtime()函数，它取时间秒数并返回一个指针指向tm中的结果。然而，tm是localtime()函数里面的一个静态结构，每当localtime()被调用时，localtime()被重写。为把内容拷进datapoint中的tm struct中，我们可能认为必须逐个拷贝每个元素。然而，我们所必须做的是一个结构分配，其余的由编译器来做，这意味着右边的一定是一个结构，不是一个指针，所以，localtime()的结果被间接引用。想要得到的结果是：

```
d.Time=*localtime(&timer)；
```

在这之后，timer增加了55秒，与读之间有一个有趣的间隔。

使用的纬度和经度是定点值，这个值表明在某一位置的一组读出值。深度和温度是由标准C库rand()函数产生的，这个函数返回零和常数RAND_MAX之间的一个伪随机数。为把这个数放进希望得到的范围里，使用模操作符%和范围的上界。这些数是整型的；为添加小数部分，对函数rand()做第二次调用，得到的值加一以后取倒数（加一是为了防止除数为零的错误）。

事实上，文件DATA.BIN作为数据容器在程序里使用，即使这个容器存在于磁盘上而不存在RAM中。为了以二进制形式发送这些数据到磁盘上，write()被使用。第一个参数是源块的开始地址——注意它必须指派为unsigned char*，因为这正是这个函数所期望的。第二参数代表要写的字节数，就是datapoint对象的大小。由于没有指针包含在datapoint里，因此向盘中写入对象时不会出现问题。如果对象更复杂，我们必须实现一个顺序化方案（大多数类库厂家已在里面建立了某种顺序化）。

2. 检验和观察数据

为检查以二进制格式存储的数据的有效性，数据从盘中读出并被放进文本文件DATA2.TXT中，所以这个文件可与DATA.TXT比较以供检验。在下面的程序里，我们会看到这个数据恢复是多么简单。在测试文件被建立后，记录在用户命令上被读出。

```
//: DATASCAN.CPP -- Verify and view logged data
#include <fstream.h>
#include <strstrea.h>
#include <iomanip.h>
#include <assert.h>
#include "..\5\datalog.h"

main() {
  ifstream bindata("data.bin", ios::binary);
  assert(bindata); // Make sure it exists
  // Create comparison file to verify data.txt:
```

China-pub.com 下载

```
  ofstream verify("data2.txt");
  datapoint d;
  while(bindata.read((unsigned char*)&d,
                     sizeof d))
    d.print(verify);
  bindata.clear(); // Reset state to "good"
  // Display user-selected records:
  int recnum = 0;
  // Left-align everything:
  cout.setf(ios::left, ios::adjustfield);
  // Fixed precision of 4 decimal places:
  cout.setf(ios::fixed, ios::floatfield);
  cout.precision(4);
  for(;;) {
    bindata.seekg(recnum* sizeof d, ios::beg);
    cout << "record " << recnum << endl;
    if(bindata.read((unsigned char*)&d,
                    sizeof d)) {
      cout << asctime(&(d.Time()));
      cout << setw(11) << "Latitude"
           << setw(11) << "Longitude"
           << setw(10) << "Depth"
           << setw(12) << "Temperature"
           << endl;
      // Put a line after the description:
      cout << setfill('-') << setw(43) << '-'
           << setfill(' ') << endl;
      cout << setw(11) << d.latitude()
           << setw(11) << d.longitude()
           << setw(10) << d.depth()
           << setw(12) << d.temperature()
           << endl;
    } else {
      cout << "invalid record number" << endl;
      bindata.clear(); // Reset state to "good"
    }
    cout << endl
         << "enter record number, x to quit:";
    char buf[10];
    cin.getline(buf, 10);
    if(buf[0] == 'x') break;
    istrstream input(buf, 10);
    input >> recnum;
  }
}
```

ifstream bindata是从文件DATA.BIN中产生并作为一个二进制文件建立起来的，带有

China-pub.com
下载

ios::nocreate标志。如果这个文件不存在，这个标志被设置导致函数 assert()失败。函数read()说明读出一个单个记录并把它直接放进datapoint d。（假如datapoint包含指针，这将产生无意义的指针值）。到文件尾时，函数read()的作用是设置bindata的failbit。它将导致while语句失败。然而，在这一点上，不能移回“取”指针并读更多的记录，因为流的状态不允许进一步读出。所以clear()函数被调用，重新设置failbit。

一旦记录从盘里读进，我们就可以做我们想做的任何事情，如执行计算或作图。这里，它被显示出来以进一步练习关于输入输出流格式方面的知识。

程序的其余部分显示用户选择的一个记录号（由 recnum代表）。像以前一样，精度是固定在4个小数的地方。但这时都是左对齐的。

这个输出的格式看来与以前不同：

```
record 0
Tue Nov 16 18:15:49 1993
Latitude    Longitude   Depth      Temperature
----------------------------------------
45*20'31"   22*34'18"   186.0172   269.0167
```

为确信标号和数据栏纵向对齐，利用 setw()，标号被放入与栏同样的宽度域内。通过设置填字符‘－’，设置希望得到的行宽而且输出单个的‘－’，这样，行间隔符产生了。

如果read()失败，我们将在else部分结束，这部分告诉用户记录数是无效的。然后，由于failbit被设置，必须调用clear()重新设置它。这样，下一个read()是成功的（假定它在正确的范围内）。

当然，也可以打开二进制数据文件来读和写。这样可以检索记录，修改它们并把它们写回到同样的位置，建立了flat-file数据库管理系统。就在我第一次做程序设计工作时，我也不得不建立乏味的文件数据库管理系统 DBMS——在Apple Ⅱ上用BASIC。它花费了几个月时间，然而，现在这样做只需几分钟。当然，现在使用一个封装的 DBMS会更有意义，但是用C++和输入输出流，仍可做在实验室里需要做的所有低级操作。

6.11 小结

这一章给出了关于输入输出流类库的一个相当完整的介绍。很可能这就是用输入输出流建立程序所必需的。（在以后的章节里，我们会看到向我们自己的类里添加输入输出流功能的简单例子。）然而，应该知道，输入输出流还有一些不常使用的其他性能，可通过查看输入输出流头文件和读我们自己编译器中的关于输入输出流的文档来发现这些性能。

6.12 练习

1) 通过创建一个叫in 的ifstream对象来打开一个文件。创建一个叫os的ostrstream对象，并通过rdbuf()成员函数把整个内容读进ostrstream。用str()函数取出os的char*地址，并利用标准C toupper() 宏使文件里每个字符大写。把结果写到一新的文件中，并删除由 os分配的内存。

2) 创建一个能打开文件（命令行中的第一个参数）的程序，并从中搜索一组字中的任何一个（命令行中其余的参数）。每次，读入一行输入并打印出与之匹配的行（带行数）。

3) 写一个在所有源代码文件的开始处添加版权注意事项的程序。只需对练习 1）稍作修改。

4) 用你最喜爱的文本搜索程序（如grep）输出包含一特殊模式的所有文件名字（仅是名字）。重定向输出到一个文件中。写一个用那个文件里的内容产生批处理文件的程序，这个批处理文件对每个由这个搜索程序找到的文件调用你的编辑器。