第21章　C++文件的输入/输出

尽管C++的I/O形成了一个完整的系统，但是文件I/O（输入/输出）却十分特殊，以至于被普遍看做是一种特例，它们有自己的约束和特点。之所以会这样，部分原因是因为最常用的文件是磁盘文件，而磁盘文件有一些其他设备不具备的性能和功能。然而要记住，磁盘文件的I/O只是一般I/O系统的一个特例，本章讨论的大部分内容同样适用于与其他类型的设备相连接的流。

21.1　<fstream>和文件类

为了执行文件的I/O，必须在程序中包括头文件<fstream>。该头文件定义了一些类，其中包括ifstream，ofstream和fstream，这些类分别从istream，ostream和iostream派生而来。记住，因为istream，ostream和iostream是从ios派生而来的，所以ifstream，ofstream和fstream同样可以访问ios定义的所有操作（参见上一章的讨论）。文件系统使用的另一个类是filebuf，这个类提供用于管理文件流的低级功能程序，通常情况下，我们不直接使用filebuf，而是将其作为其他文件类的一部分。

21.2　打开和关闭文件

在C++中，可以通过把文件链接到一个流来打开该文件。在打开一个文件之前，先要获得一个流。有三种类型的流：输入流、输出流和输入/输出流。为了创建一个输入流，必须将流声明为ifstream类；为了创建一个输出流，必须将其声明为ofstream类；既执行输入操作又执行输出操作的流必须声明为fstream类。例如，下面这段程序创建了一个输入流、一个输出流和一个既能输入又能输出的流：

```
ifstream in;  // input
ofstream out; // output
fstream io;   // input and output
```

一旦创建了一个流，就可以将它与一个文件联系起来，一种联系方法就是使用open()函数。该函数是这三个流类的成员，其中每一个原型如下所示：

```
void ifstream::open(const char *filename, ios::openmode mode = ios::in);
void ofstream::open(const char *filename, ios::openmode mode = ios::out | ios::trunc);
void fstream::open(const char *filename, ios::openmode mode = ios::in | ios::out);
```

这里，filename是文件名，它可以包括一个路径说明符。mode的值决定文件打开的方式，它必须是由openmode定义的下列值中的一个（或多个），openmode是由ios（通过其基类ios_base）定义的枚举值。

```
ios::app
ios::ate
ios::binary
```

```
ios::in
ios::out
ios::trunc
```

可以通过对这些值进行“或”操作而把它们结合起来使用。

通过包括ios::app，会使所有输出到相应文件的内容都添加到文件末尾，该值只能用于具有输出功能的文件。通过包括 ios::ate 使得在打开文件时能够定位到文件末尾。尽管 ios::ate 最初定位到文件尾，I/O 操作仍然可以在文件内的任何地方进行。

ios::in 值可以将文件指定为具有输入功能，ios::out 值可以将文件指定为具有输出功能。

ios::binary值可以使文件以二进制方式打开。默认情况下，所有文件都以文本方式打开。在文本方式中，可能发生各种字符转换，例如，回车 / 换行序列被转换为新行（即换行）。然而，当文件以二进制方式打开时，将不发生这样的字符转换。要知道，任何文件（无论是包含格式化文本，还是包含原始数据）都既可以用二进制方式打开，也可以用文本方式打开，它们之间惟一的区别是是否发生了字符转换。

ios::trunc 值将销毁具有相同名字的先前文件的内容，并且将文件长度截断为 0。当使用 ofstream 创建一个输出流时，任何先前存在的具有该文件名的文件将被自动截断。

下面的程序段将打开一个普通的输出文件：

```
ofstream out;
out.open("test", ios::out);
```

然而，如上面所示，我们很少看到 open( )被调用，因为 mode 参数对每种类型的流都提供默认值。正如它们各自的原型所示，ifstream 的 mode 参数默认为 ios::in；ofstream 的 mode 参数默认为 ios::out | ios::trunc；fstream 的 mode 参数默认为 ios::in | ios::out。因此，前面的语句通常表示如下：

```
out.open("test"); // defaults to output and normal file
```

注意：根据所使用的编译器，fstream::open( )函数的 mode 参数默认值或许不是 in | out，所以，必须明确指定该参数。

如果 open( )运行失败，相应流的值将为假（当在布尔表达式中使用时），因此，在使用文件之前，应该先对其进行测试以确保打开文件的操作取得成功。为此可使用如下语句：

```
if(!mystream) {
  cout << "Cannot open file.\n";
  // handle error
}
```

虽然利用 open( )函数打开文件完全正确，但大多数情况下都不会这样做，因为 ifstream，ofstream 和 fstream 类都有能够自动打开文件的构造函数，这些构造函数具有与 open( )函数相同的参数和默认值，因此，最常见的文件打开方式如下所示：

```
ifstream mystream("myfile"); // open file for input
```

前面曾经讲到，如果文件由于某些原因不能打开，相关的流变量的值为假，因此，无论使用构造函数还是通过明确地调用 open( )函数打开该文件，你一定想通过测试相应流的值来确定文件是否被真正打开。

还可以利用is_open( )函数检查文件是否被成功打开，该函数是fstream，ifstream和ofstream的一个成员，其原型如下：

```
bool is_open( );
```

如果流被链接到一个打开的文件，该函数返回true；否则，返回false。例如，下面的程序可以检查当前是否打开了mystream：

```
if(!mystream.is_open()) {
  cout << "File is not open.\n";
  // ...
```

为了关闭一个文件，可以使用成员函数close( )。例如，为了关闭与一个称为mystream的流相链接的文件，可使用如下语句：

```
mystream.close();
```

close( )函数没有参数并且不返回任何值。

21.3 读写文本文件

对文本文件进行读写操作非常简单，只要按在处理控制台I/O时所用相同方法使用<<和>>运算符，只是用一个链接到文件的流取代了cin和cout。例如，下面的程序创建一个简单的商品目录文件，其中包括每种商品的名字和价格：

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ofstream out("INVNTRY"); // output, normal file

  if(!out) {
    cout << "Cannot open INVENTORY file.\n";
    return 1;
  }

  out << "Radios " << 39.95 << endl;
  out << "Toasters " << 19.95 << endl;
  out << "Mixers " << 24.80 << endl;

  out.close();
  return 0;
}
```

下面的程序将读取前面程序创建的商品目录文件并将文件内容显示在屏幕上：

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
```

```
  ifstream in("INVNTRY"); // input

  if(!in) {
    cout << "Cannot open INVENTORY file.\n";
    return 1;
  }

  char item[20];
  float cost;

  in >> item >> cost;
  cout << item << " " << cost << "\n";
  in >> item >> cost;
  cout << item << " " << cost << "\n";
  in >> item >> cost;
  cout << item << " " << cost << "\n";

  in.close();
  return 0;
}
```

从某种意义上讲，利用>>和<<读写文件就像使用基于C的fprintf( )和fscanf( )函数，文件中存储的所有信息其格式与在屏幕上显示的格式相同。

下面是另一个磁盘I/O的例子，该程序读取通过键盘输入的字符串并把它们写到磁盘，当用户输入一个惊叹号时，程序停止运行。为了使用这个程序，可在命令行上指定输出文件名。

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Usage: output <filename>\n";
    return 1;
  }

  ofstream out(argv[1]); // output, normal file

  if(!out) {
    cout << "Cannot open output file.\n";
    return 1;
  }

  char str[80];
  cout << "Write strings to disk. Enter ! to stop.\n";

  do {
    cout << ": ";
    cin >> str;
    out << str << endl;
  } while (*str != '!');

  out.close();
  return 0;
}
```

使用>>运算符读取文本文件时，要记住会发生一些字符转换。例如，空白字符将被忽略。如果想要阻止进行字符转换，必须打开一个以二进制方式访问的文件并使用下一节讨论的函数。

如果在输入过程中遇到文件尾，与该文件相链接的流将为false（下一节将对此予以说明）。

21.4 无格式和二进制I/O

虽然读写格式化文本文件非常简单，但它并不总是最有效的文件处理方式。而且，我们有时会需要存储无格式（原始）的二进制数据，而不是文本数据。下面将介绍完成此功能的一些函数。

当对一个文件执行二进制操作时，要确保用方式说明符ios::binary打开该文件。虽然无格式文件函数可以处理以文本方式打开的文件，但这样会发生一些字符转换，而字符转换违背了二进制文件操作的目的。

21.4.1 字符与字节

在介绍无格式I/O之前，先要澄清一个重要概念，这一点非常重要。许多年来，C和C++中的I/O被认为是面向字节的，这是因为一个字符（char）相当于一个字节，而且惟一可用的流类型是char流。然而，随着宽字符（类型为wchar_t）及其流的出现，我们不能再把C++ I/O说成是面向字节的。相反，必须将其说成是面向字符的。当然，char流仍然是面向字节的，我们仍然可以按照字节（特别是在操作非文本数据时）考虑问题，但认为字节是字符的等价物则是错误的。

我们在第20章中讲到，因为字符流是目前为止最常用的流，所以本书使用的都是字符（char）流。由于char流在字节和字符之间建立了一对一的对应关系，所以能够很容易地处理无格式文件，从而有助于读取二进制数据块。

21.4.2 put( )和get( )函数

读写无格式数据的一种方法是使用成员函数get( )和put( )。这些函数对字符进行操作，也就是说，get( )可以读取一个字符，而put( )可以写入一个字符。当然，如果以二进制的操作方式打开一个文件并对一个char流（而不是wchar_t流）进行操作，则这些函数将读写数据字节。

get( )函数具有多种形式，但它和put( )一起使用的最常用的版本如下：

```
istream &get(char &ch);
ostream &put(char ch);
```

get( )函数从调用流中读取一个字符并将该字符值放入ch，它返回对该流的一个引用。put( )函数将ch写入相应的流并返回对该流的一个引用。

下面的程序可以将任何文件的内容显示在屏幕上，无论该文件是包含文本数据还是包含二进制数据。该程序使用了get( )函数。

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
```

```
  char ch;

  if(argc!=2) {
    cout << "Usage: PR <filename>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
  if(!in) {
    cout << "Cannot open file.";
    return 1;
  }

  while(in) { // in will be false when eof is reached
    in.get(ch);
    if(in) cout << ch;
  }

  return 0;
}
```

上一节曾经介绍过，当遇到文件尾时，与该文件相关的流变为false。因此，当in到达文件尾时，其值将变为false，从而导致while循环终止。

实际上还有一种更为简洁的方法可以编写用于读取并显示文件的循环，该方法如下所示：

```
while(in.get(ch))
  cout << ch;
```

之所以能够使用上面的程序，是因为get( )可以返回一个对in的引用，而in在遇到文件尾时将变为false。

下面的程序将使用put( )函数把所有从0到255的字符写入一个称为CHARS的文件。你或许已经知道，ASCII字符大约只占据能够被一个char存储的值的一半，其余的值通常称为扩展字符集并且包含一些诸如外文和数学符号之类的内容（并非所有的系统都支持扩展字符集，但大部分系统支持）。

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  int i;
  ofstream out("CHARS", ios::out | ios::binary);

  if(!out) {
    cout << "Cannot open output file.\n";
    return 1;
  }
  // write all characters to disk
  for(i=0; i<256; i++) out.put((char) i);

  out.close();
  return 0;
}
```

可以查一查CHARS文件的内容，看一看你的计算机可以支持哪些扩展字符，这一定很有趣。

21.4.3 read( )和write( )函数

读写二进制数据块的另一种方法是使用C++的read( )和write( )函数，这两个函数的原型如下所示：

```
istream &read(char *buf, streamsize num);
ostream &write(const char *buf, streamsize num);
```

read( )函数从调用流中读取num字符并将它们放入由buf所指的缓冲区。write( )函数把num字符从由buf所指的缓冲区写入调用流。正如前一章所讲的那样，streamsize是由C++库定义的类型——是某种整型，它可以存储能够被任何一种I/O操作转换的最大字符数。

下面的程序将把一个结构写入磁盘，然后再将其读回：

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

struct status {
  char name[ 80] ;
  double balance;
  unsigned long account_num;
};

int main()
{
  struct status acc;
  strcpy(acc.name, "Ralph Trantor");
  acc.balance = 1123.23;
  acc.account_num = 34235678;

  // write data
  ofstream outbal("balance", ios::out | ios::binary);
  if(!outbal) {
    cout << "Cannot open file.\n";
    return 1;
  }

  outbal.write((char *) &acc, sizeof(struct status));
  outbal.close();

  // now, read back;
  ifstream inbal("balance", ios::in | ios::binary);
  if(!inbal) {
    cout << "Cannot open file.\n";
    return 1;
  }

  inbal.read((char *) &acc, sizeof(struct status));

  cout << acc.name << endl;
```

```
  cout << "Account # " << acc.account_num;
  cout.precision(2);
  cout.setf(ios::fixed);
  cout << endl << "Balance: $" << acc.balance;

  inbal.close();
  return 0;
}
```

可以看到，只调用一次read( )或write( )就可以读写整个结构，而不必分别对每一个单独的域进行读写。正如这个例子说明的那样，缓冲区可以是任何对象类型。

注意： 当对没有定义为字符数组的缓冲区进行操作时，在对read( )和write( )的调用中必须进行强制类型转换。因为C++有很强的类型检查能力，所以一种类型的指针不能自动转换成另一种类型的指针。

如果还没读到num个字符就遇到文件尾，read( )将被终止，缓冲区包含了能得到的那些字符。我们可以利用另一个称为gcount( )的成员函数检查已经有多少字符被读取，该函数的原型如下：

```
streamsize gcount();
```

这个函数返回最后一个二进制输入操作读取的字符数。下面的程序给出了另一个说明read( ), write( )和gcount( )的用途的例子：

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  double fnum[4] = {99.75, -34.4, 1776.0, 200.1};
  int i;

  ofstream out("numbers", ios::out | ios::binary);
  if(!out) {
    cout << "Cannot open file.";
    return 1;
  }

  out.write((char *) &fnum, sizeof fnum);

  out.close();

  for(i=0; i<4; i++) // clear array
    fnum[i] = 0.0;

  ifstream in("numbers", ios::in | ios::binary);
  in.read((char *) &fnum, sizeof fnum);

  // see how many bytes have been read
  cout << in.gcount() << " bytes read\n";

  for(i=0; i<4; i++) // show values read from file
  cout << fnum[i] << " ";
  in.close();
```

```
  return 0;
}
```

上面的程序把一组浮点值写入磁盘，然后再把它们从磁盘读回。在调用read( )后，可用gcount( )确定刚刚读取的字节数。

21.5 其他get( )函数

除前面介绍的形式外，get( )还可以以几种不同的方式被重载。下面是三种最常见的重载形式：

```
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
int get( );
```

第一种形式把字符读入由buf指向的数组，直到读取到第num-1个字符、发现了一个换行符或者是遇到了文件尾。使用get( )时，指针buf所指的数组以null字符结束。如果在输入流中遇到换行符，则不会提取该字符。相反，它将保留在输入流中，直到进行下一个输入操作。

第二种形式把字符读入由buf指向的数组，直到读取到第num-1个字符、发现了由delim指定的字符或者是遇到了文件尾。使用get( )时，指针buf所指的数组以null字符结束。如果在输入流中遇到分隔符字符，则不会提取该字符。相反，它将保留在输入流中，直到进行下一个输入操作。

get( )的第三种重载形式返回相应流中的下一个字符。如果遇到文件尾，则返回EOF。get( )函数的这种形式类似于C的getc( )函数。

21.6 getline( )函数

另一个输入函数是getline( )，该函数是所有输入流类的成员，其原型如下所示：

```
istream &getline(char *buf, streamsize num);
istream &getline(char *buf, streamsize num, char delim);
```

第一种形式把字符读入由buf指向的数组，直到读取到第num-1个字符、发现一个换行符或者是遇到了文件尾。使用getline( )时，指针buf所指的数组以null字符结束。如果在输入流中遇到换行符，则提取该字符，但不会将其放入buf。

第二种形式把字符读入由buf指向的数组，直到读取到第num-1个字符、发现了由delim指定的字符或者是遇到了文件尾。使用getline( )时，指针buf所指的数组以null字符结束。如果在输入流中遇到分隔符，则提取该字符，但不会将其放入buf。

可以看出，getline( )函数的两个形式实际上与get( )的get(buf,num)和get(buf, num, delim)版本相同。这两个函数都从输入流中读取字符并将其放入由buf指针所指的数组，直到读到第num-1个字符或遇到了分隔符。二者的区别是：getline( )从输入流中读取并删除分隔符，get( )则不然。

下面的程序演示了getline( )函数。该程序从一个文本文件中一次读取一行内容并将其显示到屏幕上：

```
// Read and display a text file line by line.

#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Usage: Display <filename>\n";
    return 1;
  }

  ifstream in(argv[1]); // input

  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  char str[255];
  while(in) {
    in.getline(str, 255); // delim defaults to '\n'
    if(in) cout << str << endl;
  }

  in.close();

  return 0;
}
```

21.7　检测 EOF

利用成员函数 eof( )可以检测出是否到达了文件尾，该函数的原型如下：

```
bool eof( );
```

当到达文件尾时，该函数返回 true，否则返回 false。

下面的程序利用 eof( )分别以十六进制形式和 ASCII 码形式显示文件的内容：

```
/* Display contents of specified file
   in both ASCII and in hex.
*/
#include <iostream>
#include <fstream>
#include <cctype>
#include <iomanip>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Usage: Display <filename>\n";
    return 1;
  }
```

```
  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  register int i, j;
  int count = 0;
  char c[16];

  cout.setf(ios::uppercase);
  while(!in.eof()) {
    for(i=0; i<16 && !in.eof(); i++) {
      in.get(c[i]);
    }
    if(i<16) i--; // get rid of eof

    for(j=0; j<i; j++)
      cout << setw(3) << hex << (int) c[j];
    for(; j<16; j++) cout << " ";

    cout << "\t";
    for(j=0; j<i; j++)
      if(isprint(c[j])) cout << c[j];
      else cout << ".";

    cout << endl;

    count++;
    if(count==16) {
      count = 0;
      cout << "Press ENTER to continue: ";
      cin.get();
      cout << endl;
    }
  }

  in.close();

  return 0;
}
```

当用上面的程序显示它自己时，第一屏输出如下所示：

```
2F 2A 20 44 69 73 70 6C 61 79 20 63 6F 6E 74 65   /* Display conte
6E 74 73 20 6F 66 20 73 70 65 63 69 66 69 65 64   nts of specified
20 66 69 6C 65 D A 20 20 20 69 6E 20 62 6F 74     file..    in bot
68 20 41 53 43 49 49 20 61 6E 64 20 69 6E 20 68   h ASCII and in h
65 78 2E D A 2A 2F D A 23 69 6E 63 6C 75 64       ex...*/..#includ
65 20 3C 69 6F 73 74 72 65 61 6D 3E D A 23 69     e <iostream>..#i
6E 63 6C 75 64 65 20 3C 66 73 74 72 65 61 6D 3E   nclude <fstream>
D A 23 69 6E 63 6C 75 64 65 20 3C 63 63 74 79     ..#include <ccty
70 65 3E D A 23 69 6E 63 6C 75 64 65 20 3C 69     pe>..#include <i
6F 6D 61 6E 69 70 3E D A 75 73 69 6E 67 20 6E     omanip>..using n
61 6D 65 73 70 61 63 65 20 73 74 64 3B D A D      amespace std;...
A 69 6E 74 20 6D 61 69 6E 28 69 6E 74 20 61 72    .int main(int ar
```

```
67 63 2C 20 63 68 61 72 20 2A 61 72 67 76 5B 5D   gc, char *argv[]
29 D A 7B D A 20 20 69 66 28 61 72 67 63 21   ).{ .. if(argc!
3D 32 29 20 7B D A 20 20 20 20 63 6F 75 74 20   =2) { .. cout
3C 3C 20 22 55 73 61 67 65 3A 20 44 69 73 70 6C   << "Usage: Displ
Press ENTER to continue:
```

21.8 ignore( )函数

成员函数ignore( )可用来从输入流中读取和丢弃字符，其原型如下所示：

istream &ignore(streamsize *num*=1, int_type *delim*=EOF);

该函数读取和放弃字符，直到num个字符被忽略（默认值为1）或者遇到delim指定的字符（默认值为EOF）。如果遇到分隔字符，将不从输入流中将其删除。这里，int_type被定义为某种整型形式。

下面的程序读取一个名为TEST的文件。该程序将忽略字符，直到遇到空格或者读取了10个字符，然后将显示文件的其余部分：

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ifstream in("test");
  if(!in) {
    cout << "Cannot open file.\n";
    return 1;
  }

  /* Ignore up to 10 characters or until first
     space is found. */
  in.ignore(10, ' ');
  char c;
  while(in) {
    in.get(c);
    if(in) cout << c;
  }

  in.close();
  return 0;
}
```

21.9 peek( )和putback( )函数

利用peek( )，可以获取输入流中的下一个字符，同时不必删除该字符。该函数的原型如下所示：

int_type peek( );

该函数返回流中的下一个字符，如果遇到文件尾，则返回EOF（int_type被定义为某种整型类型）。

利用 putback( )，可以返回从一个流中读取的最后一个字符。该函数的原型如下所示：

```
istream &putback(char c);
```

其中，c 是读取的最后一个字符。

21.10 flush( )函数

当执行输出操作时，数据不一定立即写到与流相链接的物理设备，而是存储在一个内部缓冲区中，直到缓冲区被写满为止，因此只有该缓冲区的内容被写到磁盘。然而，通过调用 flush( )，可以在缓冲区被写满数据前强行将数据写到磁盘。该函数的原型如下：

```
ostream &flush( );
```

当在一些不利的环境中使用程序时（例如，在经常掉电的情况下），调用 flash( )是合适的。

注意：关闭文件或终止一个程序也会刷新所有缓冲区。

21.11 随机访问

在 C++ 的 I/O 系统中，可以利用 seekg( )和 seekp( )函数执行随机访问。这两个函数最常用的形式如下：

```
istream &seekg(off_type offset, seekdir origin);
ostream &seekp(off_type offset, seekdir origin);
```

其中，off_type 是 ios 定义的一个整型类型，可以包含 offset 具有的最大有效值。seekdir 是一个 ios 定义的枚举类型，用来决定查找方式。

C++ I/O 系统管理两个与文件相关的指针：一个是获取指针（get pointer），该指针指定文件中下一个输入操作发生的位置；另一个是放置指针（put pointer），该指针指定文件中下一个输出操作发生的位置。每一次发生输入或输出操作后，相应的指针将自动地顺序前移。然而，利用 seekg( )和 seekp( )函数可以以非顺序的方式访问文件。

seekg( )函数可以把相关文件当前的获取指针从指定的origin处偏移offset 个字符，origin 必须是以下三个值中的一个：

ios::beg	文件头
ios::cur	当前位置
ios::end	文件尾

seekp( )函数可以把相关文件当前的放置指针从指定的origin处偏移offset 个字符，origin 必须是上面所列的三个值中的一个。一般来说，只有对那些用二进制操作方式打开的文件才应执行随机访问 I/O，对文本文件操作时发生的字符转换可能导致想要的位置与真正的文件内容不一致。

下面的程序演示了 seekp( )函数。它允许你可以改变文件中的某个具体字符，方法是在命令行上指定一个文件名，后面加上想要改变的字符序数，再加上改变后的新字符。注意，文件以读/写操作方式打开。

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=4) {
    cout << "Usage: CHANGE <filename> <character> <char>\n";
    return 1;
  }

  fstream out(argv[1], ios::in | ios::out | ios::binary);
  if(!out) {
    cout << "Cannot open file.";
    return 1;
  }

  out.seekp(atoi(argv[2]), ios::beg);

  out.put(*argv[3]);
  out.close();

  return 0;
}
```

例如，为了使用该程序把称为TEST的文件中的第12个字符改为Z，可以使用下面的命令行：

```
change test 12 Z
```

下面的程序使用了 seekg( )函数。该程序从命令行中指定的位置开始显示文件的内容：

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Usage: SHOW <filename> <starting location>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
  if(!in) {
    cout << "Cannot open file.";
    return 1;
  }

  in.seekg(atoi(argv[2]), ios::beg);

  while(in.get(ch))
    cout << ch;
```

```
  return 0;
}
```

下面的程序利用seekp( )和seekg( )反转文件中的前<num>个字符：

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Usage: Reverse <filename> <num>\n";
    return 1;
  }
  fstream inout(argv[1], ios::in | ios::out | ios::binary);

  if(!inout) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  long e, i, j;
  char c1, c2;
  e = atol(argv[2]);

  for(i=0, j=e; i<j; i++, j--) {
    inout.seekg(i, ios::beg);
    inout.get(c1);
    inout.seekg(j, ios::beg);
    inout.get(c2);

    inout.seekp(i, ios::beg);
    inout.put(c2);
    inout.seekp(j, ios::beg);
    inout.put(c1);
  }

  inout.close();
  return 0;
}
```

为了使用这个程序，应指定想要反转的文件名，后面加上要反转的字符个数。例如，为了反转TEST文件中的前10个字符，应使用下面的命令行：

```
reverse test 10
```

如果该文件含有以下内容：

```
This is a test.
```

执行上面的程序后，文件的内容改为：

```
a si sihTtest.
```

21.11.1　获取文件的当前位置

利用下面这些函数可以确定每一个文件指针的当前位置：

```
pos_type tellg( );
pos_type tellp( );
```

其中，pos_type 是 ios 定义的类型，它存储函数可以返回的最大值。可以把 tellg( )和 tellp( )的返回值分别用做下面 seekg( )和 seekp( )的参数：

```
istream &seekg(pos_type pos);
ostream &seekp(pos_type pos);
```

这些函数使你可以保存当前文件位置、执行其他文件操作以及把文件位置重新设置为先前保存的位置。

21.12　I/O 状态

C++ I/O 系统可以保留每一个 I/O 操作的结果信息。I/O 系统的当前状态保存在一个 iostate 类型的对象中，它是 ios 定义的枚举类型，其中包括下列成员：

名字	含义
ios::goodbit	无错误位设置
ios::eofbit	当遇到文件尾时为 1；否则为 0
ios::failbit	当出现非致命的 I/O 错误时为 1；否则为 0
ios::badbit	当出现致命的 I/O 错误时为 1；否则为 0

有两种方法可以获得 I/O 状态信息。第一，可以调用 rdstate( )函数，其原型如下：

```
iostate rdstate( );
```

该函数返回错误标记的当前状态。可能通过前面的标记列表猜到，当没有错误发生时，rdstate( )返回 goodbit，否则，将开启错误标记。

下面的程序说明了 rdstate( )的使用情况。它显示一个文本文件的内容，如果出现错误，程序将利用 checkstatus( )报告之。

```
#include <iostream>
#include <fstream>
using namespace std;

void checkstatus(ifstream &in);

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Usage: Display <filename>\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.\n";
```

```
    return 1;
  }

  char c;
  while(in.get(c)) {
    if(in) cout << c;
    checkstatus(in);
  }

  checkstatus(in); // check final status
  in.close();
  return 0;
}

void checkstatus(ifstream &in)
{
  ios::iostate i;

  i = in.rdstate();

  if(i & ios::eofbit)
    cout << "EOF encountered\n";
  else if(i & ios::failbit)
    cout << "Non-Fatal I/O error\n";
  else if(i & ios::badbit)
    cout << "Fatal I/O error\n";
}
```

上面这个程序总是报告一个“错误”。正如期待的那样，在while循环结束后，最后调用的checkstatus( )报告遇到了一个EOF。你或许会发现checkstatus( )函数在你编写的程序中非常实用。

确定是否出现错误的另一种方法是使用下面列出的一个或多个函数：

```
bool bad( );
bool eof( );
bool fail( );
bool good( );
```

如果设置了badbit，bad( )函数将返回true。eof( )函数前面已经讨论过。如果设置了failbit，fail( )函数返回true。如果没有出现错误，good( )函数返回true，否则，返回false。

一旦出现错误，或许在程序继续运行之前需要清除该错误。为此，可以使用clear( )函数，其原型如下：

```
void clear(iostate flags=ios::goodbit);
```

如果flags是goodbit（默认值），所有错误标记都将被清除。否则，flags将被设置为要求的值。

21.13 定制的I/O和文件

第20章曾经介绍过怎样重载与你自己的类相关的插入和析取运算符。在第20章中，我们只执行了控制台I/O，但由于所有的C++流都是相同的，所以可以利用同样的重载插入器或析

取器函数对控制台或一个文件执行 I/O 操作，二者之间没有变化。作为一个例子，下面的程序重写了第 20 章中列举的电话簿的范例，该程序在磁盘上存储一个电话列表。这个程序非常简单：使你可以把名字添加到列表中或在屏幕上显示该列表。它使用定制的插入器和析取器输入和输出电话号码，你会发现这个程序非常有趣，因为它做了一些改进，从而能够找到一个特定号码或删除不需要的号码。

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

class phonebook {
  char name[ 80] ;
  char areacode[ 4] ;
  char prefix[ 4] ;
  char num[ 5] ;
public:
  phonebook() { } ;
  phonebook(char *n, char *a, char *p, char *nm)
  {
    strcpy(name, n);
    strcpy(areacode, a);
    strcpy(prefix, p);
    strcpy(num, nm);
  }
  friend ostream &operator<<(ostream &stream, phonebook o);
  friend istream &operator>>(istream &stream, phonebook &o);
};

// Display name and phone number.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-";
  stream << o.num << "\n";
  return stream; // must return stream
}

// Input name and telephone number.
istream &operator>>(istream &stream, phonebook &o)
{
  cout << "Enter name: ";
  stream >> o.name;
  cout << "Enter area code: ";
  stream >> o.areacode;
  cout << "Enter prefix: ";
  stream >> o.prefix;
  cout << "Enter number: ";
  stream >> o.num;
  cout << "\n";
  return stream;
```

```
}

int main()
{
  phonebook a;
  char c;

  fstream pb("phone", ios::in | ios::out | ios::app);

  if(!pb) {
    cout << "Cannot open phone book file.\n";
    return 1;
  }

  for(;;) {
    do {
      cout << "1. Enter numbers\n";
      cout << "2. Display numbers\n";
      cout << "3. Quit\n";
      cout << "\nEnter a choice: ";
      cin >> c;
    } while(c<'1' || c>'3');

    switch(c) {
      case '1':
        cin >> a;
        cout << "Entry is: ";
        cout << a; // show on screen
        pb << a; // write to disk
        break;
      case '2':
        char ch;
        pb.seekg(0, ios::beg);
        while(!pb.eof()) {
          pb.get(ch);
          if(!pb.eof()) cout << ch;
        }
        pb.clear(); // reset eof
        cout << endl;
        break;
      case '3':
        pb.close();
        return 0;
    }
  }
}
```

注意，重载的<<运算符可以用来向磁盘文件或向屏幕进行写操作。这是C++ I/O方法中最重要和最实用的特征之一。