第七章　输入输出流

本章导读

C++将输入(input)和输出(output)看着字节流。输入时，程序从输入流中抽取字节；输出时，程序将字节插入到输出流中。在C++中，cin代表标准输入，从键盘输入字符到程序；cout代表标准输出，从程序输出到屏幕。在C++中，除了从标准输入和输出外，还可以从文件实现输入和输出。

流(stream)是中介。流与键盘或文件等建立联系，程序与流交互，由流在与键盘或文件等交互。

学习目标：

1. 了解stream的概念；
2. 掌握标准输入输出；
3. 掌握文件的输出；
4. 掌握文本文件和二进制文件；
5. 掌握顺序文件和随机文件；

本章目录

第一节　快速入门

1、基本数据类型输入输出

例程7-1的功能是在vector中追加100个0-99之间的随机整数，并用copy()函数显示。在第12行代码中，ostream_iterator<int>是输出流迭代器，其中stream的含义是“流”，一个一个逐一显示，构成字符流；o是output的简写，与stream一起构成输出流；ostream_iterator的含义是输出流迭代器，其int表示该输出流迭代器处理int型数据。输出流迭代器中的cout表示从cout中输出，即标准输出流，一般代表的是屏幕。

例程7-1

```
第1行  #include<iostream>
第2行  #include<vector>
```

```
#include<algorithm>
#include<iterator>
using namespace std;

int main(){
    vector<int> vecNum;
    for(int i=0;i<100;++i)
        vecNum.push_back(rand()%100);

    copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(cout," "));

    return 0;
}
```

能输出到cout，能否输出到文件呢？观察例程7-2，代码能实现输出到文件。在第5行，包含了fstream头文件，这是文件功能的前提，其含义是file stream；第13行，申明fileOut为ofstream类，即fileOut为ofstream类的实例或对象；第14行代码打开一个名为num.txt的文件，实现文件名与对象fileOut的联系；第18行的功能是切断fileOut与文件num.txt的联系。重点是第16行，与例程7-1第12行相比，会发现cout被替换为fileOut。

例程7-2

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<iterator>
#include<fstream>//file stream文件流
using namespace std;

int main(){
    vector<int> vecNum;
    for(int i=0;i<100;++i)
        vecNum.push_back(rand()%100);

    ofstream fileOut;//output file stream，输出文件流
    fileOut.open("num.txt");//打开文件

    copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(fileOut," "));

    fileOut.close();//关闭文件

    return 0;
}
```

例程7-3是从键盘输入整数，并插入到vector中。键盘输入采用cin，构成输入字符串流，第11行是输入开始，第12行表示输入结束，构成输入流迭代器的开始和结束。第14行实现输入流到vecNum插入，第16行通过cout显示。

例程7-3

```
#include<iostream>
#include<vector>
```

```
第3行  #include<algorithm>
第4行  #include<iterator>
第5行  #include<fstream>//file stream文件流
第6行  using namespace std;
第7行
第8行  int main(){
第9行      vector<int> vecNum;
第10行
第11行     istream_iterator<int> iterFromCin(cin);
第12行     istream_iterator<int> endOfCin;
第13行
第14行     copy(iterFromCin,endOfCin,back_inserter(vecNum));
第15行
第16行     copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(cout," "));
第17行
第18行     return 0;
第19行 }
```

例程7-4实现从文件输入数据。第11行的ifstream表示input file stream，即输入文件流；第12行实现fileIn对象与num.txt的联系。第14、15行的功能实现文件输入流迭代器的开始和结束。第17行就实现把数据从文件到流传到vecNum，并在第18行输出。

例程7-4

```
第1行  #include<iostream>
第2行  #include<vector>
第3行  #include<algorithm>
第4行  #include<iterator>
第5行  #include<fstream>//file stream文件流
第6行  using namespace std;
第7行
第8行  int main(){
第9行      vector<int> vecNum;
第10行
第11行     ifstream fileIn;//input file stream，输出文件流
第12行     fileIn.open("num.txt");//打开文件
第13行
第14行     istream_iterator<int> iterFileIn(fileIn);
第15行     istream_iterator<int> endOfFile;
第16行
第17行     copy(iterFileIn,endOfFile,back_inserter(vecNum));
第18行
第19行     copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(cout," "));
第20行     fileIn.close();//关闭文件
第21行
第22行     return 0;
第23行 }
```

例程7-5也能实现文件中数据的输入，其第14-18行与例程7-4第14-17行相比发生了变化，但功能一致，其余部分没有发生变化，这是

一种传统数据读入方式。在第15行，fileIn.eof()判断流是否结束，或者说是否到文件尾部，eof是end of file的简写，返回值为true或false。如未结束，则执行循环体内的代码。当执行第16行时，自动读取内容直到空格或eof为止，并将读取内容放入变量numFromFile中，然后将指针指向下一个读取位置。第17行将读取内容追加到容器vecNum中。然后回到循环开始，判断是否到文件尾决定是否继续循环。

例程7-5

```
第1行  #include<iostream>
第2行  #include<vector>
第3行  #include<algorithm>
第4行  #include<iterator>
第5行  #include<fstream>//file stream文件流
第6行  using namespace std;
第7行
第8行  int main(){
第9行      vector<int> vecNum;
第10行
第11行     ifstream fileIn;//input file stream，输出文件流
第12行     fileIn.open("num.txt");//打开文件
第13行
第14行     int numFromFile;
第15行     while(!fileIn.eof()){
第16行         fileIn>>numFromFile;//从文件中读取数据，并放入变量numFromFile
第17行         vecNum.push_back(numFromFile);
第18行     }
第19行
第20行     copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(cout," "));
第21行     fileIn.close();//关闭文件
第22行
第23行     return 0;
第24行 }
```

例程7-5和例程7-4是两种不同读取文件内容的方法，往文件写入数据也有类似的模式(如例程7-6所示)。采用迭代器模式，适合大量同类型数据，而传统模式则能适应更加多变的场景等，如int、double等多类型数据等。

例程7-6

```
第1行  #include<iostream>
第2行  #include<vector>
第3行  #include<algorithm>
第4行  #include<iterator>
第5行  #include<fstream>//file stream文件流
第6行  using namespace std;
第7行
第8行  int main(){
第9行      vector<int> vecNum;
第10行
第11行     ofstream fileOut;//Output file stream，输出文件流
第12行     fileOut.open("num.txt");//打开文件
```

```
第13行
第14行      if(!fileOut.fail()){//判断打开文件是否是否失败
第15行          for(int i=0;i<100;++i)
第16行              fileOut<<(rand()%100)<<" ";//输出100个随机整数
第17行      }
第18行      fileOut.close();//关闭文件
第19行
第20行      return 0;
第21行  }
```

2、自定义类输入输出

输入输出流不仅能处理基本数据类型，也同样能处理自定义数据类型即class或struct，如例程7-7即是UpDown类输出到文件。

例程7-7

```
第1行   #include<fstream>
第2行   #include<iostream>
第3行   #include<algorithm>
第4行   #include<iterator>
第5行   #include<vector>
第6行   using namespace std;
第7行   class UpDown{
第8行   public:
第9行       UpDown(int U=1,int D=1):Up(U),Down(D){}
第10行      friend inline ostream& operator<<(ostream &out,const UpDown &UD);//重载<<输出运算符
第11行  private:
第12行      int Up,Down;
第13行  };
第14行  inline ostream& operator<<(ostream &out,const UpDown &UD){
第15行      return out<<'('<<UD.Up<<'/'<<UD.Down<<')';
第16行  }
第17行
第18行  int main(){
第19行      vector<UpDown> vecUpDown;
第20行      for(int i=0;i<100;++i)//随机产生100个分数
第21行          vecUpDown.push_back(UpDown(rand()%10,rand()%10+1));
第22行
第23行      ofstream outFile;
第24行      outFile.open("UD.data");
第25行      ostream_iterator<UpDown>iter_oFile(outFile);
第26行      copy(vecUpDown.begin(),vecUpDown.end(),iter_oFile);
第27行      //文件中记录形式形如：(7/2)(0/5)(4/10)(8/9)(4/3)……
第28行
第29行      outFile.close();
第30行
第31行      return 0;
第32行  }
```

从例程7-7中可以看出，第26行代码与之前的文件输出代码相似，但是为什么能(7/2)(0/5)(4/10)(8/9)(4/3)……呢？原因是类UpDown中重载了输出运算符<<，在其中定义了输出的格式。

例程7-8是从例程7-7中产生的文件UD.txt中读入数据。

例程7-8

```
第1行  #include<fstream>
第2行  #include<iostream>
第3行  #include<algorithm>
第4行  #include<iterator>
第5行  #include<vector>
第6行  using namespace std;
第7行  class UpDown{
第8行  public:
第9行      UpDown(int U=1,int D=1):Up(U),Down(D){}
第10行     friend istream& operator>>(istream &in,UpDown &UD);
第11行     friend ostream& operator<<(ostream &out,const UpDown &UD);
第12行     void Print(){
第13行         cout<<Up<<"/"<<Down<<" ";
第14行     }
第15行 private:
第16行     int Up,Down;
第17行 };
第18行 ostream& operator<<(ostream &out,const UpDown &UD){
第19行     return out<<'('<<UD.Up<<'/'<<UD.Down<<')';
第20行 }
第21行 istream& operator>>(istream &in,UpDown &UD){
第22行     char ch1,ch2,ch3;
第23行     int Up,Down;
第24行     in>>ch1>>UD.Up>>ch2>>UD.Down>>ch3;
第25行     //注意不能写成：in>>'('>>UD.Up>>'/'>>UD.Down>>')';
第26行     return in;
第27行 }
第28行 int main(){
第29行     ifstream inFile;
第30行     inFile.open("UD.data");
第31行     istream_iterator<UpDown>iterBeg(inFile);
第32行     istream_iterator<UpDown>iterEnd;
第33行     vector<UpDown> vecUD(iterBeg,iterEnd);
第34行     copy(vecUD.begin(),vecUD.end(),ostream_iterator<UpDown>(cout));
第35行     inFile.close();
第36行
第37行     return 0;
第38行 }
```

程序运行时，不管int还是UpDown，都存储在内存中，无需顾虑数据类型的不同。存储在硬盘中的文件也应该与内存相似，是否更通用的输入输出方式呢？如例程7-9所示。

例程7-9

```
第1行  #include <iostream>
第2行  #include <fstream>
第3行  #include <vector>
第4行  #include <string>
第5行  #include <iterator>
第6行  using namespace std;
第7行
第8行  class UpDown{
第9行  public:
第10行      UpDown(int U=1,int D=1):Up(U),Down(D){}
第11行      friend inline std::ostream& operator<<(std::ostream& os, const UpDown& UD);
第12行 private:
第13行      int Up,Down;
第14行 };
第15行
第16行 inline std::ostream& operator<<(std::ostream& os, const UpDown& UD){
第17行      os.write(reinterpret_cast<const char*>(&UD),sizeof(UpDown));
第18行      return os;
第19行 }
第20行
第21行 int main()
第22行 {
第23行      vector<UpDown> vecUD;
第24行      for(int i=0;i<10;++i)
第25行          vecUD.push_back(UpDown(rand()%10,rand()%10+1));
第26行
第27行      ofstream fileOutBin;
第28行      fileOutBin.open("binData.dat",ios::binary | ios::out);//定义打开文件的方式
第29行
第30行      cout<<"\n\n开始写数据\n";
第31行      std::ostream_iterator<UpDown>os(fileOutBin,"");
第32行      std::copy(vecUD.begin(),vecUD.end(),os);
第33行      cout<<"\n\n结束写数据\n";
第34行      fileOutBin.close();//关闭文件
第35行
第36行      return 0;
第37行 }
```

当用文本编辑器打开例程7-9产生的数据文件binOut.dat时，会发现有很多奇怪的符号，而不像前面产生的文件，能轻松阅读。粗略查看代码，似乎与前面具有相同功能的例程没有太大区别。关键在第28行和第17行。第28行的含义是以二进制方式(ios::binary)打开文件用于输出(ios::out)，即在输出时将采用二进制方式。第17行中的os.write()的含义向os写入内容；os是ostream类的实例名称；reinterpret_cast<const char*>(&UD)功能是将UD变量所在地址所指向内存单元中的内容转换为字符指针(数组)，sizeof(UpDown)计算出长度，即UpDown所占长度。通过reinterpret_cast()模板函数，实现UpDown类对象所在内存向字符指针转换。当读取时按相同方式即可，如例程7-10所示。

例程7-10

```
第1行  #include <iostream>
第2行  #include <fstream>

第3行  #include <vector>
第4行  #include <string>
第5行  #include <iterator>
第6行  using namespace std;
第7行
第8行  class UpDown{
第9行  public:
第10行      UpDown(int U=1,int D=1):Up(U),Down(D){}
第11行      friend inline std::ostream& operator<<(std::ostream& os, const UpDown& UD);
第12行 private:
第13行      int Up,Down;
第14行 };
第15行 inline std::istream& operator>>(std::istream& is, UpDown& UD)
第16行 {
第17行      is.read(reinterpret_cast<char*>(&UD), sizeof(UpDown));
第18行      return is;
第19行 }
第20行 inline std::ostream& operator<<(std::ostream& os, const UpDown& UD){
第21行      os<<"("<<UD.Up<<"/"<<UD.Down<<")";
第22行      return os;
第23行 }
第24行
第25行 int main()
第26行 {
第27行      vector<UpDown> vecUD;
第28行
第29行      ifstream fileInBin;
第30行      fileInBin.open("binData.dat", std::ios::binary | std::ios::in);//以二进制方式打开文件
第31行
第32行      //从文件中读入数据
第33行      istream_iterator<UpDown> is(fileInBin);

第34行      istream_iterator<UpDown> eof;

第35行      copy(is, eof, back_inserter(vecUD));
第36行
第37行      fileInBin.close();//关闭文件
第38行      //显示读出的数据
第39行      copy(vecUD.begin(),vecUD.end(),ostream_iterator<UpDown>(cout));
第40行
第41行      return 0;
第42行 }
```

比较例程7-10第17行以及例程7-9第17行，会发现其输出与输入保持一致，都是将UpDown类型的数据转换为char*。另外，当

reinterpret_cast()模板函数用于文件二进制输入或输出时，几乎都是char*，虽然理论上可以用其他数据类型，但可能存在这样或那样的问题。

在例程7-10和例程7-9中，<<即用于输出到文件，也用于输出到标准设备屏幕。由于两个读写分别在两个文件，因此可以改写operator<<中的代码。当读写文件以及屏幕输出在同一个文件时，该怎么办呢？如例程7-11所示即可，对于cin默认所代表的键盘输入，同样可以在operator>>中做类似处理。

例程7-11

```
第1行  inline std::ostream& operator<<(std::ostream& os, const UpDown& UD){
第2行      if(os==std::cout){//用于屏幕输出时
第3行          os<<"("<<UD.Up<<"/"<<UD.Down<<")";
第4行      }else{//用于文件输出时
第5行          os.write(reinterpret_cast<const char*>(&UD),sizeof(UpDown));
第6行      }
第7行
第8行      return os;
第9行  }
```

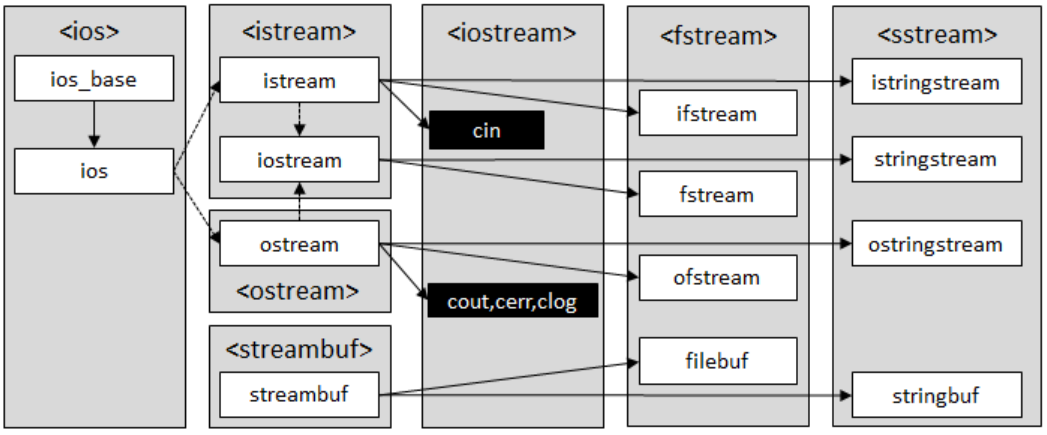
另外，在例程7-10中，operator<<()函数是友元函数，而operator>>()则不是。实际上，这两个都可以不是友元函数，原因这两个函数都无需访问UpDown中的数据成员，在reinterpret_cast()模板函数中，是将UpDown类实例化后的对象作为一个整体转化为char*，因此无需设置为友元函数。

第二节　基本原理

1、流标准库

与流相关的主要标准库如图7-1所示，另有<iofwd>、<iomanip>未在其中列出。<ios>、<istream>、 <ostream>、 <streambuf>和<iosfwd>一般不直接使用，常是其他类库的基类。直接使用的标准库如下：

- <iostream>是最常用的标准库，定义有标准输入cin和标准输出cout以及cerr和clog；
- <fstream>定义文件流类，如ifstream、fstream以及fstream类等，以及内置其中的缓存，如filebuf等。文件流类是文件操作的基础，是程序与文件交互的桥梁；
- <sstream>用于操作字符串流；



尖括号中为库文件；白底黑字为库文件中定义的类；黑底白字为库文件中的实例对象

图7-1　输入输出流关系图

从图7-1能梳理出输入输出流对象的继承关系图，如图7-2所示，其中iostream继承自istream类和ostream类，根据类的继承关系，此处为虚拟继承。istream和ostream也是虚拟继承自ios类。

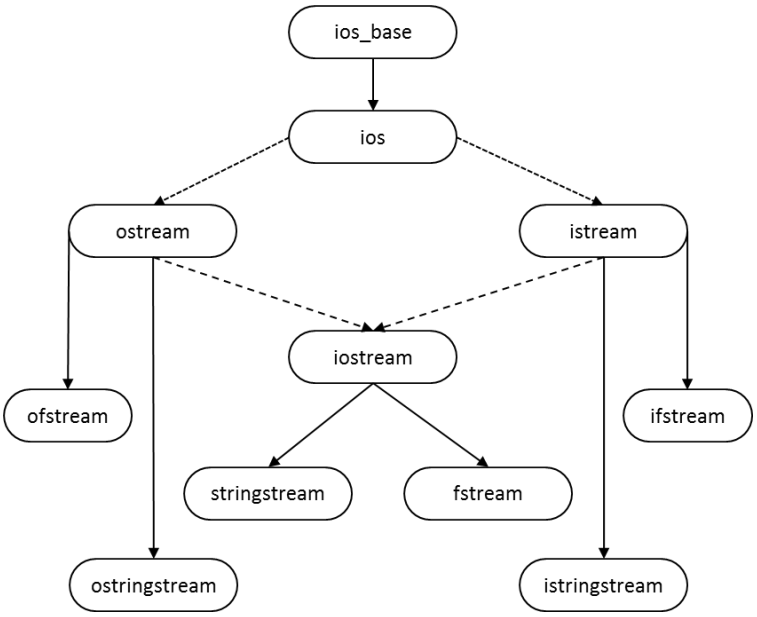


图7-2　输入输出流继承关系图

2、文件流与标准流

图7-3是C++输入输出流示意图。在C++中，程序从流输入或输出，然后流再与设备或文件打交道。当流与设备或文件等建立起联系后，程序就可以与流交互，并且程序无需区分是设备还是文件等，即程序操作文件与设备具有一致性。例程7-1第12行与例程7-2第16行相比，仅cout与fileOut不同。cout输入到标准输出设备即屏幕，fileOut是输出到文件。fileOut是ofstream类的实例对象，其通过打开文件num.txt建立起文件与流之间的联系。

总之，当输入输出流建立起与文件的联系后，操作文件与操作标准设备方式极为相似，无论输入还是输出。

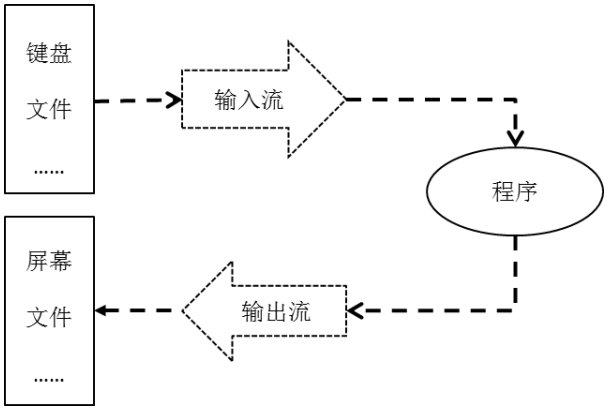


图7-3　输入输出流示意图

3、流的状态

在程序执行过程中，流存在各种可能，如：读取数据时企图打开不存在的文件、写入数据时文件定位到不能写入的硬件(如光盘等)、读写过程中设备发生故障，此时就必须获知流的状态并执行相关处理，表7-1是流状态的标志位常量。

表7-1：流状态及其检测函数

I/O状态常量值	描述	检测函数				
		good()	eof()	fail()	bad()	rdstate()
goodbit	正常状态，常量值为0	true	false	false	false	goodbit
eofbit	当输入模式时，到达文件尾的状态，常量值为1。	false	true	false	false	eofbit
failbit	输入输出操作时发生逻辑错误，常量值为2。	false	false	true	false	failbit
badbit	输入输出操作时发生读写错误，常量值为4。	false	false	true	true	badbit

例程7-12

```
第1行  #include<iostream>
第2行  using namespace std;
```

```
第3行
第4行  int main(){
第5行      cout<<"ios::badbit="<<ios::badbit<<endl;//输出：4
第6行      cout<<"ios::failbit="<<ios::failbit<<endl;//输出：2
第7行      cout<<"ios::eofbit="<<ios::eofbit<<endl;//输出：1
第8行      cout<<"ios::goodbit="<<ios::goodbit<<endl;//输出：0
第9行
第10行     return 0;
第11行 }
```

流对象有3个成员函数，分别是rdstate()、setstate()以及clear()用于处理流状态标志位。rdstate()用于读取流状态标志位；setstate()用于设置流状态位；clear()用于清除流状态位。例程7-13是上述3个成员函数的应用示例。

例程7-13

```
第1行  #include<iostream>
第2行  #include<fstream>//file stream文件流
第3行  using namespace std;
第4行
第5行  int main(){
第6行      ofstream fileOut;//Output file stream，输出文件流
第7行      fileOut.open("num.txt");//打开文件
第8行
第9行      cout<<"fileOut.rdstate()="<<fileOut.rdstate()<<endl;//输出值为：0
第10行
第11行     fileOut.setstate(ios::badbit);
第12行     cout<<"fileOut.rdstate()="<<fileOut.rdstate()<<endl;//输出值为：4
第13行
第14行     fileOut.setstate(ios::failbit);
第15行     cout<<"fileOut.rdstate()="<<fileOut.rdstate()<<endl;//输出值为：6
第16行
第17行     fileOut.clear(ios::failbit);
第18行     cout<<"fileOut.rdstate()="<<fileOut.rdstate()<<endl;//输出值为：2
第19行
第20行     if(fileOut.rdstate()==ios::failbit)exit(1);//直接退出
第21行
第22行     return 0;
第23行 }
```

当文件被打开时，应在数据输入输出前，首先判断文件是否打开成功，如果未打开成功就对文件进行读写操作，将带来难以预期的效果。如例程7-14所示。

例程7-14

```
第1行  #include<iostream>
第2行  #include<vector>
第3行  #include<algorithm>
第4行  #include<iterator>
第5行  #include<fstream>//file stream文件流
第6行  using namespace std;
```

```
第7行
第8行  int main(){
第9行      vector<int>vecNum;
第10行     ofstream fileOut;//Output file stream，输出文件流
第11行     fileOut.open("num.txt");//打开文件
第12行
第13行     //判断文件打开是否正确
第14行     if(fileOut.rdstate()>0)exit(1);
第15行
第16行     for(int i=0;i<100;++i)//产生100个随机整数
第17行         vecNum.push_back(rand()%100);
第18行
第19行     //输出到文件
第20行     copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(fileOut,""));
第21行
第22行     //输出到cout，即标准输出设备，此处是屏幕
第23行     copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(cout," "));
第24行
第25行     fileOut.close();
第26行
第27行     return 0;
第28行 }
```

直接使用流的状态标志位不是很方便，C++还提供了流对象的成员函数来检测标志位，例程7-15是检测函数的应用示例，注意：例程7-15第14行与例程7-14第14行的比较。

例程7-15

```
第1行  #include<iostream>
第2行  #include<vector>
第3行  #include<algorithm>
第4行  #include<iterator>
第5行  #include<fstream>//file stream文件流
第6行  using namespace std;
第7行
第8行  int main(){
第9行      vector<int>vecNum;
第10行     ofstream fileOut;//Output file stream，输出文件流
第11行     fileOut.open("num.txt");//打开文件
第12行
第13行     //判断文件打开是否正确
第14行     if(fileOut.fail())exit(1);//用fail()函数检测文件打开状态
第15行
第16行     for(int i=0;i<100;++i)//产生100个随机整数
第17行         vecNum.push_back(rand()%100);
第18行
第19行     //输出到文件
```

```
第20行        copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(fileOut,""));
第21行
第22行        //输出到cout，即标准输出设备，此处是屏幕
第23行        copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(cout," "));
第24行
第25行        fileOut.close();
第26行
第27行        return 0;
第28行    }
```

4、文件打开模式

在C++中，ofstream仅用于输出数据到文件，ifsteam仅用于从文件读入数据，除此之外，还可以用fstream创建输出流或输入流，甚至既可以输入也可以输出的文件流。当用fstream打开文件时，必须指定文件打开模式，而ofstream和ifstream则无需指定文件打开模式，或者说这两个类已经在代码内部固定文件打开模式。fstream打开文件的模式如表7-2所示。

表7-2：文件模式

文件模式	描述
ios::in	in是input的简写，功能是打开文件用于输入数据
ios::out	out是output的简写，功能是打开文件用于输出数据
ios::app	app是append的简写，功能是打开文件用于输出，输出数据附加于文件末尾
ios::ate	ate是at end的简写，功能是打开文件用于输入。如果文件存在则移动到文件尾部能写入数据位置
ios::truct	truct是truncate的简写。如果文件已存在，丢弃文件内容，此为ios::out的默认方式
ios::binary	打开文件用于二进制输入输出。在C++中如未指明用二进制打开文件，默认以文件方式打开文件。

有些文件打开模式也可以用于ifstream和ofstream，如：ofstream类的对象实例打开文件时，可以使用ios::app模式，用于向文件追加数据。但是，文件打开模式，最好仅用于fstream对象。

在实际使用过程中，可以使用单个模式也可以组合多个模式，如fileOutBin.open("binData.dat",ios::binary | ios::out)，其中fileOutBin为fstream类的实例对象，ios::binary与ios::out组合使用，其间的|运算符是位或运算符。例程7-9第28行与例程7-10第30行就是两种文件模式的联合使用。

5、文件指针

此处的指针，与C++中一般定义的指针没有太多关系，也与C语言的文件指针无关，当然其内在本质实现本质，与指针关系密切。

在C++中，当文件被打开后，会自动生成一个隐含的文件指针，文件的读或写就从这个指针所在的位置开始。当用ios::app或ios::ate方式打开一个文件后，文件指针自动指向文件的末尾；当用其他方式打开，则文件指针都指向文件的开头。

在C++中，每执行一次读或写操作后，文件指针自动移动到下一个读或写的操作位置，移动量与文件的打开方式，读写参数有关。当文件以文本模式打开时，默认以空格或设定分隔符作为下一个读或写的操作位置；如果以二进制方式打开，则以设定移动量作为下一个读或写的操作位置。

6、输入流(istream)

istream是ifstream、istringstream以及iostream的基类；由于fstream以及stringstream继承自iostream，因此istream也间接成为fstream和stringstream的基类。根据C++类关系的基本原理，派生类有基类的特征，因此，了解istream，也就认知了输入流的基本特征。

7、输出流(ostream)

ostream是ofstream、ostringstream以及iostream的基类；由于fstream以及stringstream继承自iostream，因此ostream也间接成为fstream和stringstream的基类。根据C++类关系的基本原理，派生类有基类的特征，因此，了解ostream，也就认知了输出流的基本特征。

第三节　文本文件

文本文件由于结构简单，被广泛地用于记录各种信息。文本文件除了用专用程序产生和读取外，还可以用文本编辑器方便地编辑，但其往往占据的空间较大(在辅助存储器越来越廉价的今天，这个缺点有时可以忽略不计)。

1、向文本文件写入

例程7-16是向文本文件输出数据，其输出结果在文件中的形态如“John190”和“Rose085”。由于1与90、0与85之间没有空格，因此不利于程序读取和管理。为此，在输出时，可以在每个输出项之间增加空格，形如：fileOut<<"John"<<" "<<1<<" "<<90<<endl。

例程7-16

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

int main(){

    ofstream fileOut;
    fileOut.open("Data.txt");
    //写入第1行
    fileOut<<"John"<<1<<90<<endl;//姓名，性别，成绩
    fileOut<<"Rose"<<0<<85<<endl;//姓名，性别，成绩
    //在文件中的形态
    //John190
    //Rose085
    fileOut.close();

    return 0;
}
```

2、从文件文本读取

例程7-17是从文本文件中读出数据。在文本文件读取中，以空格作为分隔符。在C语言中，字符可以保存在字符数组中，为避免字符长度超过定义长度，可以加大字符数组的长度。在C++中，用string数据类型更加方便。

例程7-17

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

int main(){
    ifstream fileIn;
    fileIn.open("Data.txt");

    string Name;
    int Score;
    bool Sex;
    if(!fileIn.eof()){
        fileIn>>Name>>Sex>>Score;//姓名，性别，成绩
        cout<<Name<<" "<<Sex<<" "<<Score<<endl;
        fileIn>>Name>>Sex>>Score;//姓名，性别，成绩
        cout<<Name<<" "<<Sex<<" "<<Score<<endl;
    }
```

```
第19行      fileIn.close();
第20行
第21行      return 0;
第22行  }
```

如是连续汉字或英文单词构成的文章，用<<方式未必方便，C++提供的getline()函数能实现逐行读入，非常方便，如例程7-18所示。另外，流提取运算符<<默认以空白符作为分隔符，当空白符是文本的一部分时，采用<<就难以胜任。

getline(char array[],int size,char delimitChar)

当函数读取到间隔符(delimitChar)或者达到文件尾，或者已读取size-1个字符时，就会停止读取。数组中最后一个位置放置空结束符('\0')。如果函数是在读取到间隔符后终止，间隔符虽然被读入，但不保存在数组中。第3个参数delimitChar的默认值为('\n')。

例程7-18

```
第1行   #include<iostream>
第2行   #include<string>
第3行   #include<fstream>
第4行   using namespace std;
第5行
第6行   int main(){
第7行       ifstream readingText;
第8行       readingText.open("cpp.txt");
第9行       char textIn[1000];
第10行      if(!readingText.eof()){
第11行          while(!readingText.eof()){
第12行              readingText.getline(textIn,sizeof(textIn));
第13行              cout<<textIn<<endl;
第14行          }
第15行      }
第16行      readingText.close();
第17行
第18行      return 0;
第19行  }
```

例程7-18可以用迭代器方式更加简单地实现，如例程7-19所示。相比例程7-18代码更加简洁，但也存在问题，即不能区分换行符号，将空格以及换行都视为空白符。

例程7-19

```
第1行   #include<iostream>
第2行   #include<string>
第3行   #include<iterator>
第4行   #include<fstream>
第5行   using namespace std;
第6行
第7行   int main(){
第8行       ifstream readingText;
第9行       readingText.open("cpp.txt");
第10行
```

```
第11行      istream_iterator<string> stringFromFile(readingText);
第12行      istream_iterator<string>Eof;
第13行      copy(stringFromFile,Eof,ostream_iterator<string>(cout," "));
第14行      readingText.close();
第15行
第16行      return 0;
第17行  }
```

3、get()和put()

get()函数从输入流对象读取一个字符，put()向输出流对象写入一个字符。get()函数有3个版本：

```
char get()//返回一个字符
istream* get(char &reading)//ch为变量引用，读入字符到ch
char get(char reading[],int size, char delimitChar)//读入字符到数组
```

get()系列函数将文件全部视为字符，不再区分空白符等，能将空格、换行等字符全部正确读入。第一个版本返回从输入流对象读入一个字符，其返回值为读入的字符；第二个版本需要变量引用，从输入流读取的字符存入该变量中，另外，此版本还把引用返回给所使用的对象；第3个版本与getline()几乎完全相同，但该函数不会把字符串结束符('\0')放在字符串数组最后。注意：第二版本中的istream是ifstream的基类。例程7-20是部分get()函数的应用示例。

例程7-20

```
第1行   #include<iostream>
第2行   #include<string>
第3行   #include<iterator>
第4行   #include<fstream>
第5行   using namespace std;
第6行
第7行   int main(){
第8行       ifstream readingText;
第9行       readingText.open("cpp.txt");
第10行
第11行      while(!readingText.eof()){
第12行          cout<<(char)readingText.get();
第13行      }
第14行      readingText.close();
第15行
第16行      ifstream is("cpp.txt");
第17行      while(!is.good()){
第18行          cout<<is.get();
第19行      }
第20行      is.close();
第21行
第22行      return 0;
第23行  }
```

put()函数只有一个版本，其功能是将指定的字符写出到输出流对象，其使用格式如下所示。例程7-21是get()和put()函数的示例，其功能是复制文件。

```
void put(char reading)
```

例程7-21

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

int main(){
    ifstream readingText;//读入数据
    ofstream writingFile;//写出数据

    readingText.open("cpp.txt");
    writingFile.open("CppCopy.txt");

    while(!readingText.eof()){
        writingFile.put(readingText.get());//读取，然后写入。
    }
    readingText.close();
    writingFile.close();

    return 0;
}
```

观察复制后的目标文件会发现其长度比源文件长1个字节，原因是第15行代码是先写入文件，然后再判断是否结束，因此目标文件实际上是将文件结束符也当成普通字符写入了目标文件。复制的流程应该是先判断是否结束，然后再决定是否写入。因此，可将14-16行代码修改如下：

例程7-22

```
char readingChar;
readingChar=readingText.get();
while(!readingText.eof()){
    writingFile.put(readingChar);//读取，然后写入。
    readingChar=readingText.get();
}
```

第四节　二进制文件

二进制文件和文本文件没有本质的区别，因为硬盘和内存都只能按二进制存储。也可以认为文本文件仅是二进制文件的特例，其中仅仅保存有ASCII字符，而二进制文件则可以保存各种类型的文件。对于图片文件、可执行文件等，非常适合于用二进制方式读取和解读。在C++中文件默认以文本模式打开，如果需要以二进制方式工作，打开文件时必须指定为二进制模式ios::binary。二进制文件的读写采用流对象的read()和write()成员函数，其格式如下：

```
fileStream.read(char *address,int count)
fileStream.write(char *address,int count)
```

从格式中可以看出，二进制文件的读写，仅能适用于字符地址，或者说字符数组(数组也可视为指针)。对于非字符序列，可以将之转

换为字符序列。在二进制文件操作中，空白符、换行符等都是二进制，都是用相同的规则处理。

另外，二进制文件一般采用fstream类，而不用ifstream或ofstream类。例程7-23是二进制文件的示例之一。

例程7-23

```
第1行  #include<iostream>
第2行  #include<fstream>
第3行  using namespace std;
第4行
第5行  int main(){
第6行      //随机生成100个字符并以二进制方式存入文件
第7行      fstream binFile("binData.dat",ios::out|ios::binary);
第8行      char singleChar;
第9行      for(int i=0;i<100;++i){
第10行         singleChar=65+rand()%26;
第11行         binFile.write(&singleChar,1);
第12行     }
第13行     binFile.close();
第14行
第15行     binFile.open("binData.dat",ios::in|ios::binary);
第16行     while(!binFile.eof()){
第17行         binFile.read(&singleChar,1);
第18行         cout<<singleChar;
第19行     }
第20行     binFile.close();
第21行
第22行     return 0;
第23行 }
```

例程7-23虽然是二进制文件，但由于写入的是文本信息，仍然可以通过文本编辑器查看。例程7-24是将随机生成的100个double型的数据写入文件并读出。

例程7-24

```
第1行  #include<iostream>
第2行  #include<fstream>
第3行  using namespace std;
第4行
第5行  int main(){
第6行      //随机生成100个double型数据并以二进制方式存入文件
第7行      fstream binFile("binData.dat",ios::out|ios::binary);
第8行      double dblNum;
第9行      for(int i=0;i<100;++i){
第10行         dblNum=rand()/100.0;
第11行         binFile.write((char*)&dblNum,sizeof(dblNum));
第12行     }
第13行     binFile.close();
第14行
第15行     binFile.open("binData.dat",ios::in|ios::binary);
```

```
第16行      while(!binFile.eof()){
第17行          binFile.read((char*)&dblNum,sizeof(dblNum));
第18行          cout<<dblNum<<" ";
第19行      }
第20行      binFile.close();
第21行
第22行      return 0;
第23行  }
```

二进制文件不仅适用于基本数据类型，也适用于自定义的类或结构，如例程7-25所示。如果说例程7-23和例程7-24生成的文件可读或者基本可读，则例程7-25则可读性就已经非常差。与此同时，如果不知道文件数据的生成规则，则解读文件中的数据，则变得难度很大。

例程7-25

```
第1行   #include<iostream>
第2行   #include<fstream>
第3行   using namespace std;
第4行   class UpDown{
第5行   public:
第6行       UpDown(int U=1,int D=1):Up(U),Down(D){}
第7行       friend ostream & operator<<(ostream &Out,const UpDown&UD);
第8行   private:
第9行       int Up,Down;
第10行  };
第11行  ostream & operator<<(ostream &Out,const UpDown&UD){
第12行      return Out<<"("<<UD.Up<<"/"<<UD.Down<<")";
第13行  }
第14行  int main(){
第15行      //随机生成100个UpDown数据
第16行      fstream binFile("binData.dat",ios::out|ios::binary);
第17行      UpDown UD;
第18行      for(int i=0;i<100;++i){
第19行          UD=UpDown(rand()%10,rand()%10+1);
第20行          binFile.write((char*)&UD,sizeof(UD));
第21行      }
第22行      binFile.close();
第23行
第24行      binFile.open("binData.dat",ios::in|ios::binary);
第25行      while(!binFile.eof()){
第26行          binFile.read((char*)&UD,sizeof(UD));
第27行          cout<<UD;
第28行      }
第29行      binFile.close();
第30行
第31行      return 0;
第32行  }
```

从例程7-24和例程7-25可以看出，对于非字符型数据，必须进行转换后才能以二进制方式读写文件。转换时，必须指定内存位置，长

度，程序依此转换为对应的字符形态。这种转换还可以用reinterpret_cast()函数，其格式如下，其中dataType是转换后的数据类型，address是被转换数据的地址。例程7-26是reinterpret_cast()函数的应用示例，注意与例程7-25的比较。注意：二进制文件同样适用于迭代器模式，如例程7-9和例程7-10所示。

```
reinterpret_cast<dataType>(address)
```

例程7-26

```
#include<iostream>
#include<fstream>
using namespace std;
class UpDown{
public:
    UpDown(int U=1,int D=1):Up(U),Down(D){}
    friend ostream & operator<<(ostream &Out,const UpDown&UD);
private:
    int Up,Down;
};
ostream & operator<<(ostream &Out,const UpDown&UD){
    return Out<<"("<<UD.Up<<"/"<<UD.Down<<")";
}
int main(){
    //随机生成100个UpDown数据
    fstream binFile("binData.dat",ios::out|ios::binary);
    UpDown UD;
    for(int i=0;i<100;++i){

        UD=UpDown(rand()%10,rand()%10+1);
         binFile.write(reinterpret_cast<char*>(&UD),sizeof(UD));
    }
    binFile.close();

    binFile.open("binData.dat",ios::in|ios::binary);
    while(!binFile.eof()){
         binFile.read(reinterpret_cast<char*>(&UD),sizeof(UD));
        cout<<UD;
    }
    binFile.close();

    return 0;
}
```

当自定义结构或者类中含有动态内存分配时要特别注意。在例程7-27中，STUD类含有string类型的Name数据成员，其本意是可以保存“学生姓名”，学生姓名可长可短。但是无论姓名长短，sizeof(STUD)的结果在笔者系统中都是40(注：不同的系统，其值可能不同)，但很明显，学生姓名的长短在这个程序中是随机变化的。将不同长度的数据用同样的长度保存，必然错误。将这个程序拆分成两个独立程序，其中一个用于写入数据，另外一个用于读出数据，结果明显不正确；而如果将这两个部分合并在一个程序中，则结果似乎正确，原因是什么呢？当利用reinterpret_cast()函数转换时，实际上是将studA或者studB所占据的内存对应转换成字符，而对于Name，由于是

string类型，其表现为地址，实际存储空间是动态分配。write()时，实际上是将Sex、Score以及Name的地址保存在文件中。当两个部分在一个程序内时，分配的内存空间在程序结束前继续留存，因此还能几乎正确获取；而分配为两个程序时，写入程序结束后，分配的内容空间被收回，读取时就不能在原有地址获得正确的数据。

例程7-27

```
第1行  #include<iostream>
第2行  #include<string>
第3行  #include<fstream>
第4行  using namespace std;
第5行  class STUD{
第6行  public:
第7行      STUD(string N="",bool Sex=1,int S=0):Name(N),Sex(Sex),Score(S){}
第8行      friend ostream & operator<<(ostream &out,const STUD &stud);

第9行  private:
第10行     string Name;//姓名
第11行     int Sex;//1代表Male，0代表Female
第12行     int Score;//成绩
第13行 };
第14行 ostream & operator<<(ostream &out,const STUD &stud){
第15行     return out<<stud.Name<<(stud.Sex?"Male":"Female")<<stud.Score<<endl;
第16行 }
第17行 int main(){
第18行     //随机生成100个UpDown数据
第19行     fstream binFileOut("binData.dat",ios::out|ios::binary);
第20行     STUD studA;
第21行     string tempName;
第22行     for(int i=0;i<100;++i){
第23行         tempName=string("ABCDEFGHIJKLMNOPQRSTUVWXYZ").substr(0,rand()%25+1);
第24行         studA=STUD(tempName,rand()%2,rand()%100);
第25行         cout<<studA;
第26行         binFileOut.write(reinterpret_cast<char*>(&studA),sizeof(studA));
第27行     }
第28行     binFileOut.close();
第29行
第30行     fstream binFileIN;
第31行     STUD studB;
第32行     binFileIN.open("binData.dat",ios::in|ios::binary);
第33行     while(!binFileIN.eof()){
第34行         binFileIN.read(reinterpret_cast<char*>(&studB),sizeof(studB));
第35行         cout<<studB;
第36行     }
第37行     binFileIN.close();
第38行
第39行     return 0;
```

```
}
```

第五节　随机文件

文件中数据的访问方式可以分为：顺序访问(Sequential Access)和随机访问(Random Access)。在顺序访问中，文件指针只能一直向前移动。当文件以输入方式打开时，从起始位置向末尾读取数据；当以输出方式打开时，从开始位置或末尾位置(追加模式ios::app)逐一写入数据。在顺序访问中，为了读取特定位置的数据，必须逐项读取其前面的所有字节；而在随机访问中，可以随意向前或者向后移动文件指针。在C++中，当以ios::binary(二进制方式)打开文件时可以用流对象的成员函数seekp()或seekg()移动文件指针。seekp()用于输出流，是seek put的简写；seekg()用于输入流，是seek get的简写。这两个函数都有单参数和双参数版本。单参数版本采用绝对定位，即相对于文件开始位置的绝对位置；双参数版本则是相对于定位基址的偏移量。定位基址有3种，分别是：ios::beg、ios::end和ios::cur分别指文件开始、文件结束以及当前位置。另外，还可以用tellp()或tellg()两个无参函数返回文件指针的当前位置。例程7-28是seekp()与tellp()的应用示例。

例程7-28

```
#include<iostream>
#include<fstream>
using namespace std;
struct UpDown{
    UpDown(int U=1,int D=1):Up(U),Down(D){}
    void Print(){
        cout<<" ("<<(this->Up)<<"/"<<(this->Down)<<") ";
    }
    int Up,Down;
};

fstream udFile;
UpDown tmpUD;

void read_show(){
    cout<<"tellg()="<<udFile.tellg();
    udFile.read(reinterpret_cast<char*>(&tmpUD),sizeof(tmpUD));
    tmpUD.Print();
    cout<<"tellg()="<<udFile.tellg()<<endl;
}
int main(){
    udFile.open("ud.dat",ios::binary|ios::out);//以二进制输出方式打开文件
    if(udFile.fail())return 1;//判断文件打开是否失败
    for(int i=1;i<=1000;++i){
        tmpUD=UpDown(i,i+1);//注意分数的规则，形如1/2,2/3,3/4
        udFile.write(reinterpret_cast<char *>(&tmpUD),sizeof(tmpUD));
    }
    udFile.close();

    udFile.open("ud.dat",ios::in|ios::binary|ios::out);
    int UDSize=sizeof(UpDown);

    udFile.seekg(15);read_show();//输出：tellg()=15 (768/1024) tellg()=23
```

```
第34行        udFile.seekg(15*UDSize);read_show();//输出：tellg()=120 (16/17) tellg()=128
第35行        udFile.seekg(100*UDSize,ios::beg);read_show();//输出：tellg()=800 (101/102) tellg()=808
第36行        udFile.seekg(99*UDSize,ios::beg);read_show();//输出：tellg()=792 (100/101) tellg()=800
第37行        udFile.seekg(0*UDSize,ios::cur);read_show();//输出：tellg()=800 (101/102) tellg()=808
第38行        udFile.seekg(0*UDSize,ios::cur);read_show();//输出：tellg()=808 (102/103) tellg()=816
第39行        udFile.seekg(10*UDSize,ios::cur);read_show();//输出：tellg()=896 (113/114) tellg()=904
第40行        udFile.seekg(-30*UDSize,ios::end);read_show();//输出：tellg()=7760 (971/972) tellg()=7768
第41行        udFile.seekg(-1*UDSize,ios::end);read_show();//输出：tellg()=7992 (1000/1001) tellg()=8000
第42行
第43行        //改变数据
第44行        udFile.seekp(15*UDSize);
第45行        tmpUD=UpDown(1,3);
第46行        udFile.write(reinterpret_cast<char *>(&tmpUD),sizeof(UpDown));
第47行        udFile.seekg(15*UDSize);read_show();//输出：tellg()=120 (1/3) tellg()=128
第48行        udFile.seekp(0,ios::end);
第49行        tmpUD=UpDown(1,5);
第50行        udFile.write(reinterpret_cast<char *>(&tmpUD),sizeof(UpDown));
第51行        udFile.seekg(0,ios::end);read_show();//输出：tellg()=8008 (1/5) tellg()=-1

第52行        udFile.close();
第53行
第54行        return 0;
第55行    }
```

在例程7-28中，第22行和第30行分别是打开文件，从中可以看出，随机文件要求必须以ios::binary方式打开，但从第30行能看出，文件可以同时具有输入和输出状态，而以前的文件打开模式，仅能在输入或输出二者之间选择其一。

观察第33行，发现输出与预期不同，udFile.seekg(15)是绝对定位到第15个字节，而udFile.seekg(15*udSize)相当于以udSize为单位，跳动15个单位，由于udSize是sizeof(UpDown)，因此每次跳动长度与UpDown相同，每次能正确取得UpDown数据，而绝对跳动15个字节，则可能取到两个UpDown之间，因此发生错误。因此，在二进制文件或随机文件，要特别注意该问题。

第35-41行都是采用相对定位，即以ios::beg、ios::cur或ios::end为基础跳动，如果输入负数，则反向跳动。虽然第37、38行的seekg()参数相同，但输出结果完全不同，原因是每次读取后，文件指针自动向后跳跃一个读取单元。在文件体系中，尤其要注意文件指针的变化。在第15-20行之间，tellg()值不同，相差8个字节，即相差sizeof(UpDown)，即每执行一次读写，文件指针跳跃一次读写长度。

在第43-51之间，在原有执行读取的文件中，开始写入数据。在写入数据时，原有数据被覆盖。在写入过程中，也要考虑文件指针的位置，如果指向两个数据单元之间，则会覆盖错误，影响一个或多个数据单元。另外，数据写入，可以在开始覆盖，也可以在数据中间覆盖，也可以在文件尾部追加。

另外，将第40行udFile.seekg(-30*UDSize,ios::end)修改为udFile.seekg(-30*sizeof(UpDown),ios::end)将导致错误，虽然有第31行int UDSize=sizeof(UpDown)，即UDSize与sizeof(UpDown)相等。原因是sizeof()返回值为unsigned int(无符号整数，表示0和正整数)，当其结果与-30相乘时，其结果将被转换为unsigned int，导致结果与预期严重不符。当两个不同数据类型的操作数一起运算时，C++约定了自动转换规则。为了避免类似错误，可以显式转换其类型，将其修改为udFile.seekg(-30*int(sizeof(UpDown)),ios::end)或udFile.seekg(-30*static_cast<int>(sizeof(UpDown)),ios::end)即可，可以认为前一种方式C模式，后一种是C++模式。

第六节　字符串流

流可以附着在string上，也就是说，可以通过流所提供的功能从string读出或向string写入，这样的流称之为stringstream，相关定义在sstream库中。与文件流相似，字符串流也分为输入流istringstream、输出流ostringstream和输入输出流stringstream。istringstream继承自istream；ostringstream继承自ostream；stringstream则继承自iostream。正因为有如此继承关系，所以与cin、cout以及ifstream、ofstream和fstream都有相似的特征。例程7-29是istringstream的简单应用。

例程7-29

```
第1行  #include<iostream>
第2行  #include<string>
第3行  #include<iterator>
第4行  #include<sstream>

第5行  using namespace std;
第6行
第7行  int main(){
第8行      string myStr("C++ is a programming language.");
第9行      istringstream ssIN(myStr);//等效表达：istringstream ssIN;ssIN.str(myStr);
第10行
第11行     istream_iterator<string> isBeg(ssIN);
第12行     istream_iterator<string> isEnd;
第13行     copy(isBeg,isEnd,ostream_iterator<string>(cout,""));//输出时删除空格
第14行
第15行     copy(myStr.begin(),myStr.end(),ostream_iterator<char>(cout,""));//原样输出
第16行
第17行     return 0;
第18行 }
```

在例程7-29中，第9行申明ssIN为istringstream类的实例对象，并用myStr所代表的string对象初始化，其含义是将string类的实例myStr作为字符串流。第11-13行是构造输入流迭代器，与文件输入流或cin等相似。第15行是将string类的myStr作为string容器，逐个字符输出。

字符串流广泛地用于从字符串中提取数据，如例程7-30所示。从字符串中提取数据，如果不用字符串流将比较麻烦，而用字符串流只需列出对照关系即可。第11行、第13行是具体实现从ssIN输出到变量中，如果将ssIN替换成cin，就会更加熟悉，实际上ssIN与cin在功能上类似。

例程7-30

```
第1行  #include<iostream>
第2行  #include<string>
第3行  #include<sstream>
第4行  using namespace std;
第5行
第6行  int main(){
第7行      string myStr("John 1 85 Rose 0 90");
第8行      istringstream ssIN(myStr);//等效表达：istringstream ssIN;ssIN.str(myStr);
第9行      string Name;bool Sex;int Score;
第10行
第11行     ssIN>>Name>>Sex>>Score;
第12行     cout<<"姓名:"<<Name<<" 性别："<<(Sex?"男":"女")<<" 成绩:"<<Score<<endl;
第13行     ssIN>>Name>>Sex>>Score;
第14行     cout<<"姓名:"<<Name<<" 性别："<<(Sex?"男":"女")<<" 成绩:"<<Score<<endl;
第15行
第16行     return 0;
第17行 }
```

在例程7-30第12行申明ostringstream类的实例对象os，第13行则将vector中的数据输出到os中，如同输出到cout或输出文件流中。

第16行的os.str()是将os中的数据转换为字符串。第19行的isIN是输入字符串流，以os.str()初始化，然后将该字符串流输入到dblNum中并通过cout显示。第20行的isIN.eof()是判断输入字符串流是否结束。

例程7-31

```
第1行  #include<iostream>
第2行  #include<string>
第3行  #include<iterator>
第4行  #include<vector>
第5行  #include<sstream>
第6行  using namespace std;
第7行
第8行  int main(){
第9行      vector<double> dblData;
第10行     for(int i=0;i<100;++i)
第11行         dblData.push_back(rand()/100.0);
第12行     ostringstream os;
第13行     copy(dblData.begin(),dblData.end(),ostream_iterator<double>(os," "));
第14行     //数据输出到ostringstream类的os中
第15行
第16行     cout<<os.str()<<endl;//os.str()将输出流转换成字符串
第17行
第18行     double dblNum;
第19行     istringstream isIN(os.str());
第20行     while(!isIN.eof()){
第21行         isIN>>dblNum;
第22行         cout<<dblNum<<endl;
第23行     }
第24行
第25行     return 0;
第26行 }
```

第七节　输入输出格式化

第八节　cout

cout作为ostream的实例对象，代表标准的输出流，用于输出char类型数据，相当于C语言的stdout。可以用插入操作符<<使cout接受格式化的数据，也可以用write()函数让cout接受非格式化数据。cout被定义在iostream库中，使用时必须包含iostream库，且在整个程序运行期间一直有效。

第九节　cin

cin作为istream的实例对象，代表标准的输入流，用于输入char类型数据或其转换类型，相当于C语言的stdin。可以用抽取操作符>>使cout接受格式化的数据，也可以用read()函数让cout接受非格式化数据。cin被定义在iostream库中，使用时必须包含iostream库，且在整个程序运行期间一直有效。

第十节　小结

第十一节　阅读材料