面向对象编程基础

本课程入选教育部产学合作协同育人项目
课程主页:**http://cpp.njuer.org**
课程老师:陈明 **http://cv.mchen.org**

```
ppt和代码下载地址
git clone https://gitee.com/cpp-njuer-org/book
```

C++标准库

IO库

顺序容器

泛型算法

关联容器

动态内存

第8章

IO库

- IO类
- 文件输入输出
- string流

之前用过的IO库设施

istream：输入流类型，提供输入操作。
ostream：输出流类型，提供输出操作
cin：一个istream对象，从标准输入读取数据。
cout：一个ostream对象，向标准输出写入数据。
cerr：一个ostream对象，向标准错误写入消息。
>>运算符：用来从一个istream对象中读取输入数据。
<<运算符：用来向一个ostream对象中写入输出数据。
getline函数：从给定的istream对象中读取一行数据，存入到一个给定的string对象中。

```
string line;
getline(cin,line);
```

IO类

IO库类型和头文件

头文件	类型	
iostream	istream	wistream
	ostream	wostream
	iostream	iostream
fstream	ifstream	wifstream
	ofstream	wofstream
	fstream	wfstream
sstream	istringstream	wistringstream
	ostringstream	wostringstream
	stringstream	wstringstream

- iostream头文件：从标准流中读写数据
- fstream头文件：从文件中读写数据
- sstream头文件：从字符串中读写数据
- w*stream类，和*stream功能类似，只是用于宽字符版本。
    - wchar_t char
    - wcin wcout wcerr 分别对应cin cout cerr的宽字符版对象

IO类型间的关系

- 设备类型和字符大小不影响执行的IO操作
  - 用>>读取数据，不用管设备类型（控制台\文件\string）字符（char\wchar_t）
- 标准库使我们能忽略不同类型的流之间的差异
  - 这是通过继承机制（inheritance）实现的
    - 继承机制使我们可以声明一个特定的类继承自另一个类。
    - 可以将一个派生类（继承类）对象当作其基类（所继承的类）对象来使用。
  - 类型ifstream和istringstream都继承自istream
    - 可以像使用istream对象一样来使用ifstream和istringstream对象
    - 如何使用cin的，就可以同样地使用这些类型的对象
      - 可以对一个ifstream或istringstream对象调用getline
      - 可以使用>>从一个ifstream或istringstream对象中读取数据
    - 如何使用cout的，就可以同样地使用这些类型的对象。
- 所介绍的标准库流特性都可以无差别地应用于普通流、文件流和string流，
以及char或宽字符流版本。

IO对象无拷贝或赋值

```
//不能拷贝或对IO对象赋值

ofstream out1, out2;
out1=out2;                    //错误 不能对流对象赋值
ofstream print(ofstream);     //错误 不能初始化ofstream参数
out2=print(out2);             //错误 不能拷贝流对象
```

- 由于不能拷贝IO对象，因此我们也不能将形参或返回类型设置为流类型
- 进行IO操作的函数通常以引用方式传递和返回流。
- 读写一个IO对象会改变其状态，因此传递和返回的引用不能是const的。

条件状态

- IO操作可能发生错误，下表帮助访问和操纵流的条件状态(condition state)

strm是一种IO类型，（如istream）， s是一个流对象。

strm::iostate	是一种机器无关的类型，提供了表达条件状态的完整功能
strm::badbit	用来指出流已经崩溃
strm::failbit	用来指出一个IO操作失败了
strm::eofbit	用来指出流到达了文件结束
strm::goodbit	用来指出流未处于错误状态，此值保证为零
s.eof()	若流s的eofbit置位，则返回true
s.fail()	若流s的failbit置位，则返回true
s.bad()	若流s的badbit置位，则返回true
s.good()	若流s处于有效状态，则返回true
s.clear()	将流s中所有条件状态位复位，将流的状态设置成有效，返回void
s.clear(flags)	根据给定的标志位，将流s中指定的条件状态位复位，返回void
s.setstate(flags)	根据给定的标志位，将流s中对应的条件状态位置位，返回void
s.rdstate()	返回流s的当前条件状态，返回值类型为strm::iostate

一个IO错误的例子

```
int ival;
cin >> ival;
//当输入Boo时，期待读取int，得到B，cin进入错误状态
//输入文件结束标志，cin也会进入错误状态
```

- 一个流一旦发生错误，其上后续的IO操作都会失败。

```
//只有当一个流处于无错状态时，我们才可以从它读取数据，向它写入数据。
//流可能处于错误状态，因此代码通常应该在使用一个流之前检查它是否处于良好状态。
//确定一个流对象的状态的最简单的方法是将它当作一个条件来使用：
while(cin >> word)
    //ok 读操作成功...
```

查询流的状态

- 有时我们也需要知道流为什么失败。
  - 例如，文件结束标识应对措施，可能与遇到一个IO设备错误的处理方式是不同的。
- IO库定义了一个与机器无关的iostate类型，它提供了表达流状态的完整功能。
  - 这个类型应作为一个位集合来使用。
  - IO库定义了4个iostate类型的constexpr值，表示特定的位模式。
    - 这些值用来表示特定类型的IO条件，
      可以与位运算符一起使用来一次性检测或设置多个标志位。
    - badbit表示系统级错误，如不可恢复的读写错误。
      通常情况下，一旦badbit被置位，流就无法再使用了。
    - 在发生可恢复错误后，failbit被置位，如期望读取数值却读出一个字符等错误。
      这种问题通常是可以修正的，流还可以继续使用。
    - 如果到达文件结束位置，eofbit和failbit都会被置位。
      goodbit的值为0，表示流未发生错误。
    - 如果badbit、failbit和eofbit任一个被置位，则检测流状态的条件会失败。

标准库定义了一组函数来查询标志位的状态

- 操作good在所有错误位均未置位的情况下返回true
- bad、fail和eof则在对应错误位被置位时返回true
- 在badbit被置位时，fail也会返回true。
    - 使用good或fail是确定流的总体状态的正确方法。
    - 将流当作条件使用的代码就等价于！fail（）
    - eof和bad操作只能表示特定的错误

管理条件状态

- 流对象的rdstate成员返回一个iostate值，对应流的当前状态。
- setstate操作将给定条件位置位，表示发生了对应错误。
- clear成员是一个重载的成员
  - 它有一个不接受参数的版本，而另一个版本接受一个iostate类型的参数。
  - clear不接受参数的版本清除（复位）所有错误标志位。
    执行clear（）后，调用good会返回true。
  - 带参数的clear版本接受一个iostate值，表示流的新状态。
    - 为了复位单一的条件状态位，我们首先用rdstate读出当前条件状态，
      然后用位操作将所需位复位来生成新的状态。
    - 例如，将failbit和badbit复位，但保持eofbit不变

```
//记住cin的当前状态
auto old_state = cin.rdstate();      //记住cin当前状态
cin.clear();                         //使cin有效
process_input(cin);                  //使用cin
cin.setstate(old_state);             //将cin置为原有状态
//复位failbit badbit,其它标志位不变
cin.clear(cin.rdstate()&~cin.failbit&~cin.badbit)
```

```cpp
//condition_state.cpp
#include <iostream>
using std::cin; using std::cout; using std::endl;
#include <sstream>
using std::istringstream;
#include <string>
using std::string;
void read(){
    // turns on both fail and bad bits
    cin.setstate(cin.badbit | cin.eofbit | cin.failbit);
}
void off(){
    // turns off failbit and badbit but all other bits unchanged
    cin.clear(cin.rdstate() & ~cin.failbit & ~cin.badbit);
}
int main()
{
    cout << "before read" << endl;
    if (cin.good()) cout << "cin's good" << endl;
    if (cin.bad()) cout << "cin's bad" << endl;
    if (cin.fail()) cout << "cin's fail" << endl;
    if (cin.eof()) cout << "cin's eof" << endl;
```

```
    read();
    cout << "after read" << endl;
    if (cin.good()) cout << "cin's good" << endl;
    if (cin.bad()) cout << "cin's bad" << endl;
    if (cin.fail()) cout << "cin's fail" << endl;
    if (cin.eof()) cout << "cin's eof" << endl;

    off();
    cout << "after off" << endl;
    if (cin.good()) cout << "cin's good" << endl;
    if (cin.bad()) cout << "cin's bad" << endl;
    if (cin.fail()) cout << "cin's fail" << endl;
    if (cin.eof()) cout << "cin's eof" << endl;
    return 0;
}
//before read
//cin's good
//after read
//cin's bad
//cin's fail
//cin's eof
//after off
//cin's eof
```

练习

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

```
std::istream& func(std::istream &is)
{
    std::string buf;
    while (is >> buf)
        std::cout << buf << std::endl;
    is.clear();
    return is;
}
```

练习

测试函数，调用参数为cin。

```
#include <iostream>
using std::istream;

istream& func(istream &is)
{
    std::string buf;
    while (is >> buf)
        std::cout << buf << std::endl;
    is.clear();
    return is;
}

int main()
{
    istream& is = func(std::cin);
    std::cout << is.rdstate() << std::endl;
    return 0;
}
```

练习

什么情况下，下面的while循环会终止？

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

如badbit、failbit、eofbit 的任一个被置位，那么检测流状态的条件会失败。

管理输出缓冲

- 每个输出流都管理一个缓冲区，执行输出的代码，
  文本串可能立即打印出来，也可能被操作系统保存在缓冲区内，随后再打印。
- 有了缓冲机制，操作系统就可以将程序的多个输出操作组合成单一的系统级写操作
  - 带来很大的性能提升

缓冲刷新(数据写到输出设备或文件)的原因

- 程序正常结束，作为main函数的return操作的一部分，缓冲刷新被执行。
- 缓冲区满时，需要刷新缓冲，而后新的数据才能继续写入缓冲区。
- 使用操纵符如endl显式刷新缓冲区。
- 在每个输出操作之后，用操纵符unitbuf设置流的内部状态，来清空缓冲区。
    - 默认情况下，对cerr是设置unitbuf的，因此写到cerr的内容都是立即刷新的。
- 一个输出流可能被关联到另一个流。在这种情况下，当读写被关联的流时，
  关联到的流的缓冲区会被刷新。
    - 例如，默认情况下，cin和cerr都关联到cout。
      因此，读cin或写cerr都会导致cout的缓冲区被刷新。

刷新输出缓冲区，使用如下IO操纵符

- endl：输出一个换行符并刷新缓冲区。
- flush：刷新流，不添加任何字符。
- ends：在缓冲区插入空字符null，然后刷新。
- unitbuf：告诉流接下来每次操作之后都要进行一次flush操作。
  - nounitbuf：回到正常的缓冲方式。

```
cout << "hi!" <<endl;    //换行 刷新
cout << "hi!" <<flush;   //刷新 不添加字符
cout << "hi!" <<ends;    //添加空字符 刷新

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

- 如果程序崩溃，输出缓冲区不会被刷新
- 当调试一个已经崩溃的程序时，需要确认那些你认为已经输出的数据确实已经刷新了。

关联输入和输出流

- 当一个输入流被关联到一个输出流时，任何试图从输入流读取数据的操作都会先刷新
  关联的输出流。

```
//标准库将cout和cin关联在一起，
cin >> ival;
//导致cout的缓冲区被刷新。
//这意味着所有输出，包括用户提示信息，都会在读操作之前被打印出来。
```

关联输入和输出流

tie有两个重载的版本
- 一个版本不带参数，返回指向输出流的指针。
    - 如果本对象当前关联到一个输出流，则返回的就是指向这个流的指针
    - 如果对象未关联到流，则返回空指针。
- 第二个版本接受一个指向ostream的指针，将自己关联到此ostream。
    - x.tie（&o）将流x关联到输出流o。

我们既可以将一个istream对象关联到另一个ostream，
也可以将一个ostream关联到另一个ostream

关联输入和输出流

```
cin.tie(&cout);//仅仅用来展示 标准库将cin cout关联在一起
//old_tie指向当前关联到cin的流（如果有的话）
ostream *old_tie = cin.tie(nullptr);//cin不再与其它流关联
//将cin与cerr关联；这不是个好主意，因为cin应该关联到cout
cin.tie(&cerr);//读取cin会刷新cerr而不是cout
cin.tie(old_tie);//重建cin cout间的正常关联
```

在这段代码中，为了将一个给定的流关联到一个新的输出流，
我们将新流的指针传递给了tie。
为了彻底解开流的关联，我们传递了一个空指针。
每个流同时最多关联到一个流，但多个流可以同时关联到同一个ostream。

文件输入输出

- 头文件fstream定义了三个类型来支持文件IO：
  - ifstream从一个给定文件读取数据。
  - ofstream向一个给定文件写入数据。
  - fstream可以读写给定文件。
- 这些类型提供的操作与我们之前已经使用过的对象cin和cout的操作一样。
  - 可以用IO运算符（<<和>>）来读写文件
  - 可以用getline从一个ifstream读取数据
  - 之前IO类介绍的内容也都适用于这些类型

FSTREAM特有的操作

操作	解释
`fstream fstrm;`	创建一个未绑定的文件流。
`fstream fstrm(s);`	创建一个文件流并打开名为s的文件，s是string或char指针
`fstream fstrm(s, mode);`	与前一个构造函数类似，但按指定mode打开文件
`fstrm.open(s)`	打开名为s的文件，并和fstrm绑定
`fstrm.close()`	关闭和fstrm绑定的文件
`fstrm.is_open()`	返回一个bool值，指出与fstrm关联的文件是否成功打开且尚未关闭

上表中，fstream是头文件fstream中定义的一个类型，fstrm是一个文件流对象。

使用文件流对象

当我们想要读写一个文件时，可以定义一个文件流对象，并将对象与文件关联起来。
每个文件流类都定义了一个名为open的成员函数，
    - 它完成一些系统相关的操作，来定位给定的文件，并视情况打开为读或写模式。
创建文件流对象时，我们可以提供文件名（可选的）。
    - 如果提供了一个文件名，则open会自动被调用

```
ifstream in(ifile); //构造一个ifstream并打开文件
ofstream out;       //输出文件流未关联到任何文件

//这段代码定义了一个输入流in，它被初始化为从文件读取数据，
//文件名由string类型的参数ifile指定。
//第二条语句定义了一个输出流out，未与任何文件关联。
```

在新C++标准中，文件名既可以是库类型string对象，也可以是C风格字符数组
旧版本的标准库只允许C风格字符数组。

用FSTREAM代替IOSTREAM&

要求使用基类型对象的地方，我们可以用继承类型的对象来替代。

- 接受一个iostream类型引用（或指针）参数的函数，
  可以用一个对应的fstream（或sstream）类型来调用。
- 如果有一个函数接受一个ostream&参数，可以传递给它一个ofstream对象，
  - 对istream&和ifstream也是类似的。

例如：

- 上一章read和print。虽然两个函数定义时指定的形参分别是istream&和ostream&，
  但我们可以向它们传递fstream对象。

```cpp
//假定输入和输出文件的名字是通过传递给main函数的参数来指定的
//avg_price_fstream.cpp
#include <iostream>
#include <fstream>
using std::cerr; using std::cin; using std::cout; using std::endl;
using std::ifstream;using std::ofstream;

#include "Sales_data.h"

int main(int argc,char* argv[])
{
    ifstream input(argv[1]);
    ofstream output(argv[2]);
    Sales_data total;          // variable to hold the running sum
    if (read(input, total))  {  // read the first transaction
        Sales_data trans;     // variable to hold data for the next transaction
        while(read(input, trans)) {      // read the remaining transactions
            if (total.isbn() == trans.isbn())   // check the isbns
                total.combine(trans);  // update the running total
            else {
                print(output, total) << endl;  // print the results
                total = trans;                 // process the next book
            }
        }
```

```
        print(output, total) << endl;           // print the last transaction
    } else {                                    // there was no input
        cerr << "No data?!" << endl;            // notify the user
    }

    return 0;
}

//g++ avg_price_fstream.cpp Sales_data.cpp
//./a.out data.in data.out
//cat data.out
//0-201-78345-X 5 110 22
//cat data.in
//0-201-78345-X 3 20.00
//0-201-78345-X 2 25.00
```

成员函数OPEN和CLOSE

```
//定义了一个空文件流对象，可以随后调用open来将它与文件关联起来：
ifstream in(ifile); //构筑一个ifstream并打开给定文件
oftream out;        //输出文件流未与任何文件相关联
out.open(ifile+".copy"); //打开制定文件

//如果调用open失败，failbit会被置位
//因为调用open可能失败，进行open是否成功的检测通常是一个好习惯
if(out) //检查open是否成功
        //open成功，可以使用文件
//这个条件判断与之前将cin用作条件相似。
//如果open失败，条件会为假，我们就不会去使用out了。
//一旦一个文件流已经打开，它就保持与对应文件的关联。
```

- 对一个已经打开的文件流调用open会失败，并会导致failbit被置位。
  随后的试图使用文件流的操作都会失败。

成员函数OPEN和CLOSE

- 为了将文件流关联到另外一个文件，必须首先关闭已经关联的文件。
  一旦文件成功关闭，我们可以打开新的文件：

```
in.close();//关闭文件
in.open(iflile+"2");// 打开另一个文件
//如果open成功，则open会设置流的状态，使得good（）为true。
```

自动构造和析构

```
//main函数接受一个要处理的文件列表
//这种程序可能会如下:
//fileIO.cpp
#include <iostream>
using std::cerr; using std::cout; using std::endl;
#include <fstream>
using std::ifstream;
#include <string>
using std::string;
#include <stdexcept>
using std::runtime_error;
void process(ifstream &is)
{
    string s;
    while (is >> s)
        cout << s << endl;
}
```

自动构造和析构

```
int main(int argc, char* argv[]){    // for each file passed to the program
    for (auto p = argv + 1; p != argv + argc; ++p) {
        ifstream input(*p);   // create input and open the file
        if (input) {          // if the file is ok, ``process'' this file
            process(input);
        } else
            cerr << "couldn't open: " + string(*p);
    } // input goes out of scope and is destroyed on each iteration
}
/*
g++ fileIO.cpp
./a.out data.in data.out
0-201-78345-X
3
20.00
0-201-78345-X
2
25.00
0-201-78345-X
5
110
22
*/
```

自动构造和析构

每个循环步构造一个新的名为input的ifstream对象，并打开它来读取给定的文件。
检查open是否成功
    - 如果成功，将文件传递给一个函数，该函数负责读取并处理输入数据。
    - 如果open失败，打印一条错误信息并继续处理下一个文件。
input是while循环的局部变量，它在每个循环步中都要创建和销毁一次
    - 当一个fstream对象离开其作用域时，与之关联的文件会自动关闭。
    - 在下一步循环中，input会再次被创建。
- 当一个fstream对象被销毁时，close会自动被调用。

练习

编写函数，以读模式打开一个文件，将其内容读入到一个string的vector中，
将每一行作为一个独立的元素存于vector中。

```
void ReadFileToVec(const string& fileName, vector<string>& vec)
{
    ifstream ifs(fileName);
    if (ifs)
    {
        string buf;
        while (getline(ifs, buf))
            vec.push_back(buf);
    }
}
```

练习

重写上面的程序，将每个单词作为一个独立的元素进行存储。

```
void ReadFileToVec(const string& fileName, vector<string>& vec)
{
    ifstream ifs(fileName);
    if (ifs)
    {
        string buf;
        while (ifs >> buf)
            vec.push_back(buf);
    }
}
```

练习

重写书店程序，从一个文件中读取交易记录。将文件名作为一个参数传递给main。

```
//avg_price_fstream2.cpp
#include <fstream>
#include <iostream>
#include "Sales_data.h"
using std::ifstream; using std::cout; using std::endl; using std::cerr;
int main(int argc, char **argv){
    ifstream input(argv[1]);
    Sales_data total;
    if (read(input, total)){
        Sales_data trans;
        while (read(input, trans)){
            if (total.isbn() == trans.isbn())
                total.combine(trans);
            else{
                print(cout, total) << endl;
                total = trans;
            }
        }
        print(cout, total) << endl;
    }
```

```
    else{
        cerr << "No data?!" << endl;
    }

    return 0;
}
/*
g++ avg_price_fstream2.cpp Sales_data.cpp
./a.out data.in
0-201-78345-X 5 110 22

cat data.in
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
*/
```

文件模式

- 每个流都有一个关联的文件模式（file mode），用来指出如何使用文件。

下表列出了文件模式和它们的含义

文件模式	解释
in	以读的方式打开
out	以写的方式打开
app	每次写操作前均定位到文件末尾
ate	打开文件后立即定位到文件末尾
trunc	截断文件
binary	以二进制方式进行IO操作。

无论用哪种方式打开文件，我们都可以指定文件模式，

- 调用open打开文件时可以
- 用一个文件名初始化流来隐式打开文件时也可以。

文件模式

指定文件模式有如下限制：

- 只可以对ofstream或fstream对象设定out模式。
- 只可以对ifstream或fstream对象设定in模式。
- 只有当out也被设定时才可设定trunc模式。
- 只要trunc没被设定，就可以设定app模式。
    - 在app模式下，即使没有显式指定out模式，文件也总是以输出方式被打开。·
- 默认情况下，即使我们没有指定trunc，以out模式打开的文件也会被截断。
    - 为了保留以out模式打开的文件的内容，
        - 我们必须同时指定app模式，这样只会将数据追加写到文件末尾；
        - 或者同时指定in模式，即打开文件同时进行读写操作
- ate和binary模式可用于任何类型的文件流对象，且可以与其他任何文件模式组合使用。

文件模式

每个文件流类型都定义了一个默认的文件模式，未指定文件模式时，就使用此默认模式。

- 与ifstream关联的文件默认以in模式打开；
- 与ofstream关联的文件默认以out模式打开；
- 与fstream关联的文件默认以in和out模式打开。

以OUT模式打开文件会丢弃

已有数据默认情况下，当我们打开一个ofstream时，文件的内容会被丢弃。
阻止一个ofstream清空给定文件内容的方法是同时指定app模式：

```
//在这几条语句中，file1都被截断
ofstream out("file1");//隐含以输出模式打开文件并截断文件
ofstream out("file1",ofstream::out);//隐含截断文件
ofstream out("file1",ofstream::out|ofstream::trunc);
//为保留文件内容，必须显式指定app模式
ofstream app("file2",ofstream::app);//隐含为输出模式
ofstream app("file2",ofstream::out|ofstream::app);//隐含为输出模式
```

- 保留被ofstream打开的文件中已有数据的唯一方法是显式指定app或in模式。

每次调用OPEN时都会确定文件模式

对于一个给定流，每当打开文件时，都可以改变其文件模式。

```
ofstream out;//未指定文件打开模式
//open调用未显式指定输出模式，文件隐式地以out模式打开。
//通常情况下，out模式意味着同时使用trunc模式。
//因此，当前目录下名为scratchpad的文件的内容将被清空。
out.open("scratchpad");//模式隐含设置为输出和截断
out.close();//关闭out，以便我们将其用于其他文件

//打开名为precious的文件时，指定了append模式。
//文件中已有的数据都得以保留，所有写操作都在文件末尾进行。
out.open("precious",ofstream::app);//模式为输出和追加
out.close();
```

- 在每次打开文件时，都要设置文件模式，可能是显式地设置，也可能是隐式地设置。
  当程序未指定模式时，就使用默认值。

练习

修改书店程序，将结果保存到一个文件中。
将输出文件名作为第二个参数传递给main函数。

```
//avg_price_fstream.cpp
#include <iostream>
#include <fstream>
using std::cerr; using std::cin; using std::cout; using std::endl;
using std::ifstream;using std::ofstream;
#include "Sales_data.h"
int main(int argc,char* argv[]){
    ifstream input(argv[1]);
    ofstream output(argv[2]);
    Sales_data total;          // variable to hold the running sum
    if (read(input, total))  {  // read the first transaction
        Sales_data trans;     // variable to hold data for the next transaction
        while(read(input, trans)) {      // read the remaining transactions
            if (total.isbn() == trans.isbn())   // check the isbns
                total.combine(trans);  // update the running total
            else {
                print(output, total) << endl;  // print the results
                total = trans;                // process the next book
            }
        }
```

```
        print(output, total) << endl;           // print the last transaction
    } else {                                    // there was no input
        cerr << "No data?!" << endl;            // notify the user
    }

    return 0;
}

//g++ avg_price_fstream.cpp Sales_data.cpp
//./a.out data.in data.out
//cat data.out
//0-201-78345-X 5 110 22
//cat data.in
//0-201-78345-X 3 20.00
//0-201-78345-X 2 25.00
```

练习

修改上一题的程序，将结果追加到给定的文件末尾。
对同一个输出文件，运行程序至少两次，检验数据是否得以保留。

```
//avg_price_fstream3.cpp
#include <iostream>
#include <fstream>
using std::cerr; using std::cin; using std::cout; using std::endl;
using std::ifstream;using std::ofstream;
#include "Sales_data.h"
int main(int argc,char* argv[]){
    ifstream input(argv[1]);
    ofstream output(argv[2],ofstream::app);
    Sales_data total;          // variable to hold the running sum
    if (read(input, total))  {  // read the first transaction
        Sales_data trans;     // variable to hold data for the next transaction
        while(read(input, trans)) {      // read the remaining transactions
            if (total.isbn() == trans.isbn())   // check the isbns
                total.combine(trans);  // update the running total
            else {
                print(output, total) << endl;  // print the results
                total = trans;                // process the next book
            }
        }
```

```
        print(output, total) << endl;           // print the last transaction
    } else {                                    // there was no input
        cerr << "No data?!" << endl;            // notify the user
    }

    return 0;
}
/*
g++ avg_price_fstream3.cpp Sales_data.cpp
rm data.out
./a.out data.in data.out
./a.out data.in data.out
cat data.in
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
cat data.out
0-201-78345-X 5 110 22
0-201-78345-X 5 110 22
*/
```

STRING流

头文件sstream定义了三个类型来支持内存IO：

- istringstream从string读取数据。
- ostringstream向string写入数据。
- stringstream可以读写给定string。

与fstream类型类似，头文件sstream中定义的类型都继承自iostream头文件中定义的类型。

- 除了继承得来的操作，sstream中定义的类型还增加了一些成员
  来管理与流相关联的string。

STRINGSTREAM特有的操作

- 下表列出了这些操作，可以对stringstream对象调用这些操作，
  但不能对其他IO类型调用这些操作。

操作	解释
sstream strm	定义一个未绑定的stringstream对象
sstream strm(s)	用s初始化对象
strm.str()	返回strm所保存的string的拷贝
strm.str(s)	将s拷贝到strm中，返回void

sstream是头文件sstream中任意一个类型。s是一个string。

使用ISTRINGSTREAM

当我们的某些工作是对整行文本进行处理，而其他一些工作是处理行内的单个单词时，
    通常可以使用istringstream。

```
//考虑这样一个例子，假定有一个文件，列出了一些人和他们的电话号码。
//文件中每条记录都以一个人名开始，后面跟随一个或多个电话号码。
morgan 2015552368 8625550123
drew 9735550130
lee 6095550132 2015550175 8005550000
```

使用ISTRINGSTREAM

我们首先定义一个简单的类来描述输入数据：

```
// members are public by default
struct PersonInfo {
    string name;
    vector<string> phones;
};
```

使用ISTRINGSTREAM

```
//我们的程序会读取数据文件，并创建一个PersonInfo的vector。
//vector中每个元素对应文件中的一条记录。
//我们在一个循环中处理输入数据，每个循环步读取一条记录，
//提取出一个人名和若干电话号码：

 // will hold a line and word from input, respectively
    string line, word;

    // will hold all the records from the input
    vector<PersonInfo> people;

    // read the input a line at a time until end-of-file (or other error)
    while (getline(is, line)) {//用getline从标准输入读取整条记录
        PersonInfo info;            // object to hold this record's data
        istringstream record(line); // bind record to the line we just read
        record >> info.name;        // read the name
        while (record >> word)      // read the phone numbers
            info.phones.push_back(word);  // and store them
        people.push_back(info); // append this record to people
    }
```

练习

使用所编写的函数打印一个istringstream对象的内容

```
#include <iostream>
#include <sstream>
using std::istream;

istream& func(istream &is)
{
    std::string buf;
    while (is >> buf)
        std::cout << buf << std::endl;
    is.clear();
    return is;
}

int main()
{
    std::istringstream iss("hello");
    func(iss);
    return 0;
}
```

练习

编写程序，将来自一个文件中的行保存在一个vector中。
然后使用一个istringstream从vector读取数据元素，每次读取一个单词。

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>
using std::vector; using std::string; using std::ifstream;
using std::istringstream; using std::cout; using std::endl; using std::cerr;

int main()
{
    ifstream ifs("data.in");
    if (!ifs)
    {
        cerr << "No data?" << endl;
        return -1;
    }
```

```
    vector<string> vecLine;
    string line;
    while (getline(ifs, line))
        vecLine.push_back(line);

    for (auto &s : vecLine)
    {
        istringstream iss(s);
        string word;
        while (iss >> word)
            cout << word << endl;
    }

    return 0;
}
```

练习

本节的程序在外层while循环中定义了istringstream对象。
如果record对象定义在循环之外，你需要对程序进行怎样的修改?
重写程序，将record的定义移到while循环之外，验证你设想的修改方法是否正确。

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
using std::vector; using std::string; using std::cin; using std::istringstream;

struct PersonInfo {
    string name;
    vector<string> phones;
};
```

```
int main()
{
    string line, word;
    vector<PersonInfo> people;
    istringstream record;
    while (getline(cin, line))
    {
        PersonInfo info;
        record.clear();
        record.str(line);
        record >> info.name;
        while (record >> word)
            info.phones.push_back(word);
        people.push_back(info);
    }
```

```
    for (auto &p : people)
    {
        std::cout << p.name << " ";
        for (auto &s : p.phones)
            std::cout << s << " ";
        std::cout << std::endl;
    }

    return 0;
}
```

练习

我们为什么没有在PersonInfo中使用类内初始化?

因为这里只需要聚合类就够了，所以没有必要在PersionInfo中使用类内初始化。

使用OSTRINGSTREAM

当我们逐步构造输出，希望最后一起打印时，ostringstream是很有用的。

- 例如，对上一节的例子，我们可能想逐个验证电话号码并改变其格式。
  - 如果所有号码都是有效的，我们希望输出一个新的文件，包含改变格式后的号码。
  - 对于那些无效的号码，我们不会将它们输出到新文件中，
    而是打印一条包含人名和无效号码的错误信息。
  - 由于我们不希望输出有无效电话号码的人，因此
    - 对每个人，直到验证完所有电话号码后才可以进行输出操作。
    - 我们可以先将输出内容“写入”到一个内存ostringstream中：

```
    for (const auto &entry : people) {    // for each entry in people
        ostringstream formatted, badNums; // objects created on each loop
        for (const auto &nums : entry.phones) {  // for each number
            if (!valid(nums)) {//valid完成电话号码验证功能
                badNums << " " << nums;  // string in badNums
            } else
                // ``writes'' to formatted's string
                formatted << " " << format(nums);//format完成改变格式的功能。
        }
        if (badNums.str().empty())      // there were no bad numbers
            os << entry.name << " "     // print the name
               << formatted.str() << endl; // and reformatted numbers
        else                    // otherwise, print the name and bad numbers
            cerr << "input error: " << entry.name
                 << " invalid number(s) " << badNums.str() << endl;
    }
//对字符串流formatted和badNums的使用。
//  使用标准的输出运算符（<<）向这些对象写入数据，
//  但这些“写入”操作实际上转换为string操作，
//  分别向formatted和badNums中的string对象添加字符。
```

完整程序

```
//sstream.cpp
#include <iostream>
using std::cin; using std::cout; using std::cerr;
using std::istream; using std::ostream; using std::endl;

#include <sstream>
using std::ostringstream; using std::istringstream;

#include <vector>
using std::vector;

#include <string>
using std::string;

// members are public by default
struct PersonInfo {
    string name;
    vector<string> phones;
};
```

完整程序

```
// we'll see how to reformat phone numbers in chapter 17
// for now just return the string we're given
string format(const string &s) { return s; }

bool valid(const string &s)
{
    // we'll see how to validate phone numbers
    // in chapter 17, for now just return true
    return true;
}

vector<PersonInfo>
getData(istream &is)
{
    // will hold a line and word from input, respectively
    string line, word;
    // will hold all the records from the input
    vector<PersonInfo> people;
```

完整程序

```
    // read the input a line at a time until end-of-file (or other error)
    while (getline(is, line)) {
        PersonInfo info;            // object to hold this record's data
        istringstream record(line); // bind record to the line we just read
        record >> info.name;        // read the name
        while (record >> word)      // read the phone numbers
            info.phones.push_back(word);  // and store them
        people.push_back(info); // append this record to people
    }

    return people;
}
```

完整程序

```
ostream& process(ostream &os, vector<PersonInfo> people)
{
    for (const auto &entry : people) {    // for each entry in people
        ostringstream formatted, badNums; // objects created on each loop
        for (const auto &nums : entry.phones) {  // for each number
            if (!valid(nums)) {
                badNums << " " << nums;  // string in badNums
            } else
                // ``writes'' to formatted's string
                formatted << " " << format(nums);
        }
        if (badNums.str().empty())      // there were no bad numbers
            os << entry.name << " "     // print the name
               << formatted.str() << endl; // and reformatted numbers
        else                    // otherwise, print the name and bad numbers
            cerr << "input error: " << entry.name
                 << " invalid number(s) " << badNums.str() << endl;
    }
    return os;
}
```

完整程序

```
int main()
{
    process(cout, getData(cin));
    return 0;
}
```

练习

重写本节的电话号码程序，从一个命名文件而非cin读取数据。

```
//phone_read.cpp
#include <iostream>
#include <sstream>
#include <fstream>
#include <string>
#include <vector>

using std::vector; using std::string; using std::cin; using std::istringstream;
using std::ostringstream; using std::ifstream;
using std::cerr; using std::cout; using std::endl;
using std::isdigit;

struct PersonInfo {
    string name;
    vector<string> phones;
};
```

```
bool valid(const string& str)
{
    return isdigit(str[0]);
}

string format(const string& str)
{
    return str.substr(0,3) + "-" + str.substr(3,3) + "-" + str.substr(6);
}

int main()
{
    ifstream ifs("phone.in");
    if (!ifs)
    {
        cerr << "no phone numbers?" << endl;
        return -1;
    }
```

```
string line, word;
vector<PersonInfo> people;
istringstream record;
while (getline(ifs, line))
{
    PersonInfo info;
    record.clear();
    record.str(line);
    record >> info.name;
    while (record >> word)
        info.phones.push_back(word);
    people.push_back(info);
}
```

```
    for (const auto &entry : people)
    {
        ostringstream formatted, badNums;
        for (const auto &nums : entry.phones)
            if (!valid(nums)) badNums << " " << nums;
            else formatted << " " << format(nums);
        if (badNums.str().empty())
            cout << entry.name << " " << formatted.str() << endl;
        else
            cerr << "input error: " << entry.name
                 << " invalid number(s) " << badNums.str() << endl;
    }

    return 0;
}
/*
morgan  201-555-2368 862-555-0123
drew  973-555-0130
lee  609-555-0132 201-555-0175 800-555-0000
*/
```

练习

我们为什么将entry和nums定义为`const auto&`?

它们都是类类型，因此使用引用避免拷贝。

小结

C++使用标准库类来处理面向流的输入和输出：

- iostream处理控制台IO
- fstream处理命名文件IO
- stringstream完成内存string的IO类

fstream和stringstream都是继承自类iostream的。
输入类都继承自istream，输出类都继承自ostream。

- 可以在istream对象上执行的操作，也可在ifstream或istringstream对象上执行。
- 继承自ostream的输出类也有类似情况。

每个IO对象都维护一组条件状态，用来指出此对象上是否可以进行IO操作。
如果遇到了错误—例如在输入流上遇到了文件末尾，

- 则对象的状态变为失效，
- 所有后续输入操作都不能执行，直至错误被纠正。
- 标准库提供了一组函数，用来设置和检测这些状态。

实践课

- 从课程主页 **cpp.njuer.org** 打开实验课 输入输出 **https://developer.aliyun.com/adc/scenario/f6838abbef584b158a6c2183f2afd3bc**
    - 使用g++编译代码
    - 编辑一个 **readme.md** 文档,键入本次实验心得.
    - 使用git进行版本控制 可使用之前的gitee代码仓库

```
- 云服务器（elastic compute service,简称ecs）
- aliyun linux 2是阿里云推出的 linux 发行版
- vim是从vi发展出来的一个文本编辑器.
- g++ 是c++编译器
```

习题1
构造一个文件，每行若干英语单词，共若干行。
编写一个拷贝程序，能把这个文件的内容复制到另一个文件中。

习题2
构造一个文件，每行若干英语单词，共若干行。
编写程序统计一个文本文件中某个字符串出现的次数，并输出对应出现次数，出现位置。

习题3
构造一个文件，每行若干英语单词，共若干行。
编写程序，把文件中英语单词按词频和字典次序排序，将单词对应词频数一起输出到另一个文件中。

注：试一试，对文本文件 `~/book/test/week9/ted.txt` 做习题3中的统计
(下载地址 `git clone https://gitee.com/cpp-njuer-org/book`)

```
编辑c++代码和markdown文档,使用git进行版本控制
yum install -y git gcc-c++
使用git工具进行版本控制
git clone你之前的网络git仓库test(或其它名字)
cd test 进入文件夹test
(clone的仓库,可移动旧文件到目录weekN:  mkdir -p weekN ; mv 文件名 weekN;)

vim test1.cpp
g++ ./test1.cpp 编译
./a.out 执行程序

vim test2.cpp
g++ ./test2.cpp 编译
./a.out 执行程序

vim test3.cpp
g++ ./test3.cpp 编译
./a.out 执行程序
```

```
git add . 加入当前文件夹下所有文件到暂存区
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
vim readme.md 键入新内容（实验感想),按ESC 再按：wq退出
git add .
git commit -m "weekN" 表示提交到本地,备注weekN

git push 到你的git仓库

git log --oneline --graph 可看git记录
键入命令并截图或复制文字,并提交到群作业.
cat test* readme.md
```

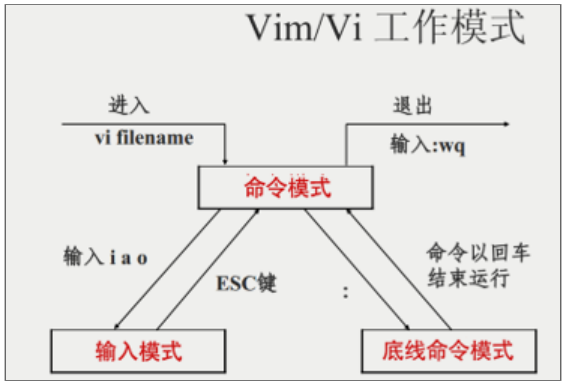
提交

- 截图或复制文字,提交到群作业.
- 填写阿里云平台（本实验）的网页实验报告栏,发布保存.本次报告不需要分享提交
- 填写问卷调查 **https://rnk6jc.aliwork.com/o/cppinfo**

关于使用TMUX

```
sudo yum install -y tmux
cd ~ && wget https://cpp.njuer.org/tmux && mv tmux .tmux.conf
tmux 进入会话 .
前缀按键prefix= ctrl+a,
prefix+c创建新面板,
prefix+"分屏,
prefix+k选上面,prefix+j选下面,
prefix+1选择第一,prefix+n选择第n,
prefix+d脱离会话
tmux attach-session -t 0 回到会话0
```

VIM 共分为三种模式



- 命令模式
  - 刚启动 vim,便进入了命令模式.其它模式下按ESC,可切换回命令模式
    - i 切换到输入模式,以输入字符.
    - x 删除当前光标所在处的字符.
    - : 切换到底线命令模式,可输入命令.
- 输入模式
  - 命令模式下按下i就进入了输入模式.
    - ESC,退出输入模式,切换到命令模式
- 底线命令模式
  - 命令模式下按下:(英文冒号)就进入了底线命令模式.
    - wq 保存退出

VIM 常用按键说明

除了 i, Esc, :wq 之外,其实 vim 还有非常多的按键可以使用.命令模式下:

- 光标移动
  - j下 k上 h左 l右
  - w前进一个词 b后退一个词
  - Ctrl+d 向下半屏 ctrl+u 向上半屏
  - G 移动到最后一行 gg 第一行 ngg 第n行
- 复制粘贴
  - dd 删一行 ndd 删n行
  - yy 复制一行 nyy复制n行
  - p将复制的数据粘贴在下一行 P粘贴到上一行
  - u恢复到前一个动作 ctrl+r重做上一个动作
- 搜索替换
  - /word 向下找word ? word 向上找
  - n重复搜索 N反向搜索
  - :1,$s/word1/word2/g从第一行到最后一行寻找 word1 字符串,并将该字符串取代为 word2

VIM 常用按键说明

底线命令模式下：

- :set nu　显示行号
- :set nonu 取消行号
- :set paste　粘贴代码不乱序

【注：把caps lock按键映射为ctrl,能提高编辑效率.】

MARKDOWN 文档语法

```
# 一级标题
## 二级标题
*斜体* **粗体**
- 列表项
  - 子列表项
> 引用
[超链接](http://asdf.com)
![图片名](http://asdf.com/a.jpg)

|表格标题1|表格标题2|
|-|-|
|内容1|内容2|
```

谢谢