一、I/O操作

> 在针对I/O操作时，标准库提供一组操作符（ manipulator ），允许程序堆输出的内容进行格式化，比如：输出数字的十六进制、浮点值的精度等。类似以前的 endl 就是一个操作符，但它并不是一个普通的值，是用于输出一个换行符并且兼具刷新缓冲区的功能。

1. 基本输入输出

1. 输出布尔数据

> 在c/c++中，在对bool类型的数据做输出的时候，打印的是 0 、1 ，如果希望看到的是 true 和 false ，那么可以使用 boolalpha 操作符。

```
#include <iostream>

using namespace std;

int main (){

        bool flag = false;
    cout << "flag的值是: " << flag << endl; // 打印 0

    //操作符只会影响后续的输出  打印 0  false
    cout << "flag的值是: " << flag  <<" 添加操作符后: "<<boolalpha << flag << endl;

        return 0 ;
}
```

2. 输出整形数字

> 在输出数字时，可以选择使用 十进制 、 八进制 、 十六进制 输出，它们只会影响整形数字， 默认会采用十进制输出数字

```cpp
#include <iostream>
using namespace std;
int main (){

        cout <<"十进制: " << dec  <<9 << endl;  // 9
    cout <<"八进制: " << oct  <<9 << endl;  // 10
    cout <<"十六进制: " << hex <<10 << endl; // a

    //若想在输出前面表示打印的数值显示前缀进制标识，可以使用showbase关键字
    cout <<showbase;
    //默认即采用十进制输出
    cout <<"十进制: " << dec  <<9 << endl; 9
    cout <<"八进制: " << oct  <<9 << endl; 011
    cout <<"十六进制: " << hex <<10 << endl;  //0xa
    cout<<noshowbase;

        return 0 ;
}
```

3. 输出浮点数

> c++ 对浮点数的输出默认只会输出六位， 那么在应对较多浮点数的时候，则常常会丢失精度，导致后面的小数位无法输出。标准库也提供了针对浮点数的输出格式 `cout.precision()` | `setprecision()`， 允许指定以多高的精度输出浮点数。

```
#include<iostream>

using namespace std;

int main(){

    double a = 10.12345678901234;
    cout.precision(3); //  设置输出多少位数字 ,该函数有返回值，为设置的精度值。
    cout  <<" a ="<<a << endl;  // 10.1

    //或者使用setprecision() ，不过需要导入 #include <iomanip>
    //做一个标记位的设置，所以还是要连上  <<
    cout  << setprecision(5)<<" a ="<<a << endl;  // 10.123

    return 0 ;
}
```

小数点后面都是0，默认是不会被输出的，若想强制输出，可以使用showpoint关键字，配合precision 可以精确打印

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

int main(){

        float  f =10.00;
        cout.precision(4);
        cout <<showpoint <<"f="<< f <<noshowpoint<< endl;  //10.00

        return 0 ;
}
```

4. 禁止忽略空白符号

> 默认情况下，获取键盘的输入内容时会自动忽略空白符（空格符 、制表符、换行符、回车符） ， 若不想忽略可以使用 `noskipws` 禁止忽略空白符， 使用 `skipws` 还原忽略即可。

```
#include<iostream>

using namespace std;

int main(){

        cin>>noskipws;
    char c ;
    cout <<"请输入字符: " << endl;
    while(cin >> c){
        cout << c ;
    }
    cin >> skipws;

       return 0 ;
}
```

2. string 流

> c++中的I/O 操作，有提供专门三个针对字符串操作的流对象，它们定义在 `#include < sstream>` 头文件中
>
> `istringtream` :从string读取数据
>
> `ostringstream` ：向string写入数据
>
> `stringstream` ：既能读取数据，也能写入数据

1. istringstream【截断字符串，默认按空格来截断】

> istringstream 是以空格为分隔符，将字符串从字符串流中在依次拿出，比较好的是它不会将空格作为流。这样就实现了字符串的空格切割。

```cpp
#include<sstream>

using namespace std;

int main(){
    string s = "我是黑马程序员 我爱黑马程序员";
    istringstream stream(str); //或者使用  stream.str(s);
    string s;
    while(stream>>s) { // 抽取stream中的值到s
        cout<<s<<endl;      //依次输出s
    }
    return 0 ;
}
```

2. ostringstream【拼接字符串】

> 我们需要格式化一个字符串，但通常并不知道需要多大的缓冲区。为了保险常常申请大量的缓冲区以防止缓冲区过小造成字符串无法全部存储。这时我们可以考虑使用 ostringstream 类，该类能够根据内容自动分配内存，并且其对内存的管理也是相当的到位。

```cpp
#include<sstream>
#include <string>

using namespace std;

int  main(){

     int a = 55;
     double b = 65.123;
     string str = "";

     //头文件是sstream
     ostringstream oss;
     oss << a << "---" << b;

     str = oss.str();
     cout << str << endl;
     return 0;
}
```

3. 文件操作

> c++的文件操作和Python的文件操作有许多相似之处，其实不止它们两，大多数编程语言在对待文件处理上都大同小异。 在Python中针对文件的操作，使用了几个函数即可 `open` 、 `read` 、 `write` ，而在C++中处理文件操作的有三个主要对象 `ifstream` 、 `ofstream` 、 `fstream` 。 需要添加头文件 `#include <fstream>`

- 文件操作常用类

> 到底是输入还是输出，是站在程序的角度看就ok.

数据类型	描述
ofstream	表示输出文件流，用于创建文件并向文件写入信息。
ifstream	表示输入文件流，用于从文件读取信息。
fstream	表示文件流，且同时具有 ofstream 和 ifstream

数据类型	描述
	两种功能，这意味着它可以创建文件，向文件写入信息，也可以从文件读取信息。

- 文件操作模式

模式标志	描述
ios::app	追加模式。所有写入都追加到文件末尾。
ios::ate	文件打开后定位到文件末尾。
ios::in	打开文件用于读取。
ios::out	打开文件用于写入。
ios::trunc	如果该文件已经存在，其内容将在打开文件之前被截断，即把文件长度设为0。

1. 读取文件

> 标准库中提供了针对文件读写操作的类，可以对文件的每行进行读取，由于后续操作矩阵数据的机会更多，所以此处就以文件存储的是矩阵数据为例。

a. 简单示例

> 文件仅仅是一些简单的字符串。

```cpp
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

int main(){

    //构建文件对象， ../表示上一级目录，因为执行的时候是exe文件，它位于debug目录，并不和
    //test.txt 处于同级目录， in表示仅仅是读取文件内容
    //ios:: 表示域操作符，表示使用ios这个对象里面的in静态常量
    fstream file{"../test.txt" , ios::in};

    //文件是否能正常打开
    if(file.is_open()){
        string line ;

        //getline 用于读取文件的整行内容，直到碰到文件末尾或者超过字符串可存储的最大值才会返回
        while (getline(file , line)){
            cout << line << endl;
        }
        //关闭文件
        file.close();
    }else{
        cout << "文件无法正常打开！"<<endl;
    }
        return 0 ;
}
```

b. 二维vector操作示例

> 文件中存放如下所示的 3 * 4 的矩阵数据，需要读取出来，然后使用vector来存储。

```
1, 6, 2, 10.5
11, 15.2, 2, 21
3, 9, 1, 7.5
```

- 示例代码

```cpp
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

int main(){

        fstream matrixFile{"../matrix.txt" , ios_base::in};

        string line;

        if(matrixFile.is_open()){
          //循环读取每一行，直到文末

          //针对每一行操作的对象，用于字符串切割、转化
          stringstream ss ;

          //用于接收每一个数字
          float number ;

          //整个矩阵数据
          vector <vector<float>> matrixVector;


          //存储每一行的矩阵
          vector <float> rowVector;


          while(getline(matrixFile , line)){
              cout << "line = " << line << endl;


              //每次开始前都清空行vector的数据
              rowVector.clear();

              //每次包装前，都清空内部数据
              ss.clear();
```

```
            //包装line数据，以便一会进行切割，
            ss.str(line);


            //循环进行转化。
            while(ss >> number){

                //获取下一个字符，如果是逗号或者空白 则忽略它，丢弃它。
                //然后while循环继续执行，获取下一个数字
                if(ss.peek() == ',' || ss.peek() == ' '){
                    ss.ignore();
                }
                //往行里面追加数据
                rowVector.push_back(number);
            }
            //每一行填充完毕之后，再存储到大的vector中
            matrixVector.push_back(rowVector);

        }

        //关闭文件
        matrixFile.close();


        //最后在这里，遍历打印二维vector即可。

          for (int i = 0; i <matrixVector.size() ; ++i) {

              for (int j = 0; j <matrixVector[i].size() ; ++j) {
                  cout << matrixVector[i][j] << " ";
              }

              cout << endl;

          }
    }else{
        cout << "文件打开失败"<<endl;
    }
return 0 ;
```

```
}
```

2. 写入文件

> 实际上和读取文差不多，如果能从文件中读取内容，那么往文件中写入内容也没有多大难处。

a. 简单示例

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){

    //若问写入操作，文件不存在，会自动创建文件
    //out:  每次写入都会覆盖源文件内容
    //app: 在末尾处追加内容
    fstream file{"../test2.txt",ios::app};

    if(file.is_open()){
        cout << "正常打开文件"<<endl;

        //写入数据
        file << ", hi c++";

        //写入换行
        file << endl;
        //写入结束
        file.close();
    }else{
        cout << "无法正常打开文件" << endl;
    }
        return 0 ;
}
```

b. 二维vector操作示例

> 现在要做的是把二维vector的矩阵数据，写入到文件中。

```cpp
#include <iostream>
#include <vector>
#include <fstream>

using namespace std;

int main(){
  //构建vector
    vector<vector<float >> matrixVector{
            {1, 6, 2, 10.5},
            {11, 15.2, 2, 21},
            {3, 9, 1, 7.5}
    };

    //构建文件对象
    fstream matrixFile{"../matrix.txt",ios::app};
    //文件能否正常使用
    if(matrixFile.is_open()){
        //开始遍历vector
        for (int i = 0; i < matrixVector.size(); ++i) {
            //遍历每一行
            for (int j = 0; j <matrixVector[i].size(); ++j) {
                //往文件写入数字
                matrixFile << matrixVector[i][j] ;

                //只要不是末尾的数字，那么都追加一个 ，
                if(j != matrixVector[i].size()-1){
                    matrixFile << ",";
                }
            }
            //写入换行
            matrixFile << endl;
        }
        //关闭文件
        matrixFile.close();
        cout << "文件写入完毕" << endl;
    }else{
        cout << "文件写入失败" << endl;
    }
```

```
    return 0 ;
}
```