# cin、cout和cerr 是什么类型？

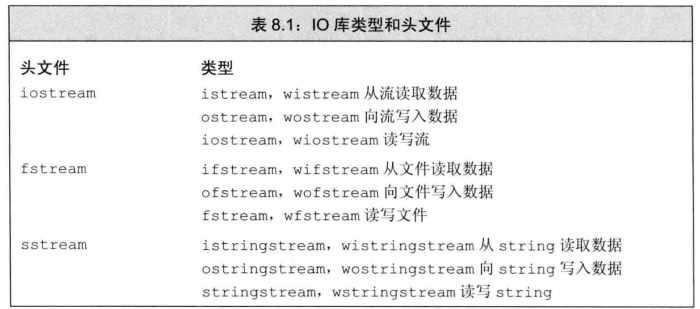
**cin** ：istream类的对象；

**cout**：ostream类的对象；

**cerr** ：ostream类的对象，和cout不一样的是，它用来输出错误信息（写到标准错误中）；

**注意**：istream和ostream都是C++提供的IO类，而cin、cout和cout都是IO对象！只不过它们是由标准库的开发者提前创建好的，可以直接拿来使用。这种在 C++ 中提前创建好的对象称为内置对象。

# C++定义了哪些IO类？



# 使用 IO对象 的时候需要注意什么？

IO对象不能拷贝或赋值：

# IO对象作为参数和返回值的时候需要注意什么？

① 因为IO对象**不能被拷贝和赋值**，这意味着不能将IO对象作为 形参 和 返回值，而应该将IO对象以引用的方式传递和返回；

② 不能将IO对象声明为const，因为不论是读取还是写入都会改变IO对象的状态；

# 为什么IO对象不能被声明为const？

因为不论是读取还是写入都会改变IO对象的状态，所以IO对象不能被声明为const。

1. ofstream out1, out2;
2. out1 = out2;  // 错误: 不能对流对象赋值
3. ofstream print(ofstream);  // 错误: 流对象要以引用的方式传递和返回
4. out2 = print(out2);  // 错误: 流对象要以引用的方式传递和返回

# 如何正确的使用 cin 获取标准输入？

错误的写法：

1. **int** val;
2. cin >> val;//val是int类型，如果用户输入了字符，比如”Boo”,则cin会进入错误状态

正确的写法：最简单的就是将流对象作为一个条件来使用：

1. **while**(cin >> word){
2. // Do sothing here.
3. }

# 流有哪几种状态？

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 常量 | failbit  标记位的值 | eofbit  标记位的值 | Badbit  标记位的值 | 二进制 | 转化为10进制 |
| ios::failbit | 1 | 0 | 0 | 100 | 4 |
| ios::eofbit | 0 | 1 | 0 | 010 | 2 |
| ios::badbit | 0 | 0 | 1 | 001 | 1 |
| ios::goodbit | 0 | 0 | 0 | 000 | 0 |

1) 有badbit、failbit、eofbit、goodbit四种，**这四个都是常量值**（iostate类型的constexpr值），它们任何一个都代表了一种流状态，因此称为“输入状态标记位常量”；

① badbit表示发生系统级的错误，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。

② failbit 表示发生可恢复的错误，如期望读取一个数值，却读出一个字符等错误。这种问题通常是可以修改的，流还可以继续使用。

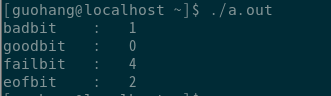
③ 当到达文件的结束位置时，eofbit 和 failbit 都会被置位。

④ goodbit 被置位表示流未发生错误。如果badbit failbit 和eofbit 任何一个被置位，则检查流状态的条件会失败。

2) 其中iostate是IO库定义的一种与机器无关的类型，提供了表达流状态的完整功能。

1. #include <iostream>
3. **int** main()
4. {
5. std::cout << "badbit    :   "<< std::cin.badbit<<std::endl; //1
6. std::cout << "goodbit   :   "<< std::cin.goodbit<<std::endl;//0
7. std::cout << "failbit   :   "<< std::cin.failbit<<std::endl;//4
8. std::cout << "eofbit    :   "<< std::cin.eofbit<<std::endl;//2
9. }

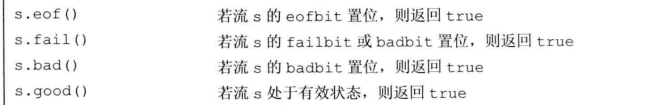
运行结果是：



# 将流作为while(condition)的condition，什么情况下condition是false？

只要badbit、failbit、eofbit有任意一个被置为，condition都会为false

# 如何查询流的状态？



# 如何清除、设置流状态？



# 获取流的当前条件状态？



# C++中io库的clear(flags) 和setstate(flags) 的区别？

## clear()函数：

**cin.clear(ios::failbit);**

　　使 得cin的流状态将按照ios::failbit所描述的样子进行设置：failbit标记位为1，eofbit标记位为0，badbit标记位为0。无 需担心goodbit标记位，failbit、eofbit、badbit任何一个为1，则goodbit为0。(goodbit是另一种流状态的表示方 法）

**cin.clear(ios::goodbit);**

　　使得cin的流状态将按照ios::goodbit所描述的样子进行设置：failbit标记位为0，eofbit标记位为0，badbit标记位为0。此时goodbit标记位为1，从另一个角度表示cin的流状态正常。

　　因此clear() 函数作用是：将流状态设置成括号内参数所代表的状态，强制覆盖掉流的原状态。

## setstate()函数：

与clear()函数不同，setstate()函数并不强制覆盖流的原状态，而是将括号内参数所代表的状态叠加到原始状态上。

比如，假设cin流状态初始正常：

1. cin.setstate (ios::failbit); //在cin流的原状态的基础上将failbit标记位置为1
2. cin.setstate (ios::eofbit); //在上一步的基础上，将cin流的eofbit标记位置为1
3. //上面两条语句结束后，cin的faibit标记位和eofbit标记位均为1，badbit标记位为0

对比clear()函数的效果：

1. cin.clear (ios::failbit);      //将cin的流状态置为ios::failbit所描述的状态
2. cin.clear (ios::eofbit);     //将cin的流状态置为ios::eofbit所描述的状态

**总结，假设流cin的状态为 old\_state，那么：**

cin.clear(flags) ：将流cin的状态直接改为flags（覆盖）；

cin.setstate(flags) ：将流cin原来的基础上，将flags叠加到**old\_state上；**

# 如何在操作cin之前保存条件状态，使用完后恢复操作前的条件状态？

* 1. auto old\_state = cin.rdstate();  // 记住 cin的当前状态
  2. cin.clear();  // 使cin 有效
  3. process\_input(cin);  // 使用cin
  4. cin.setstate(old\_state);  // 恢复cin的原有状态

# 在C++中，缓冲区有几个？

在C++中，每个I/O对象管理一个缓冲区，因此程序里有几个IO对象，就会有几个缓冲区。

# 缓冲区是什么？它的作用是什么？

**是什么？**

缓冲区是用来存储程序读写的数据的一个区域；

**有什么用？**

因为写数据需要用到系统调用write()写到设备中，如果每次读取一部分数据都直接写到设备，那将会是很大的开销，但是如果设置一个缓冲区，把数据写入到这个缓冲区中，等到合适的时候再写到设备中，那么将提高系统性能。

# 什么情况下会导致 缓冲刷新？

**(1) 程序正常结束**：

作为main()返回工作的一部分，将清空所有输出缓存区；

注意：如果程序不正常结束，输出缓冲区将不会刷新。在尝试调试已经崩溃的程序时，通常会根据最后的输出找出程序发生错误的区域。如果崩溃出现在某个特定的输出语句后面，则可能是在程序的这个位置之后出错。

**(2)** **缓冲区满了**：

在这种情况下，缓冲区将会在写下一个值前刷新；

**(3) 使用操作符显示地刷新缓冲区**：

① flush ：刷新缓冲区，但不输出任何额外的字符；

② ends ：向缓冲区插入一个空字符，然后刷新缓冲区；

③ endl ：输出一个换行符，并刷新缓冲区。

注意！C中经常使用的换行符 \n，并不具备刷新缓冲区的作用。

1. std::cout <<"Hello!"<< std::flush;//flushes the buffer; adds no data
2. std::cout <<"Hello!"<< std::ends;//inserts a null, then flushes the buffer
3. std::cout <<"Hello!"<< std::endl;//inserts a newline,then flushes the buffer

**(4) 使用操纵符unitbuf**

每次输出（如cout）后都要用flush来刷新缓冲区太麻烦了，用操纵符unitbuf可以一劳永逸：使用操纵符unitbuf，它会告诉流（如cout）在接下来的每一次操作后都进行一次flush操作：

1. cout << unitbuf;  //所有输出操作后都会立即刷新缓冲区
2. //自此期间，任何输出都立即刷新，无缓冲
3. cout << nounitbuf;  //回到正常的缓冲方式

其中nounitbuf将重置流，使流恢复使用正常的系统管理的缓冲区刷新机制。

**(5) 输出流关联了输入流**

在这种情况下，任何尝试 从输入流尝试读取数据的操作 都将首先刷新其关联的输出流缓冲区。

例如标准库就将cin和cout关联在了一起，因此 用cin读取数据将导致cout被刷新：

cin >> val ; // cout将被刷新，因为标准库就将cin和cout关联在了一起

# 如果希望所有的输出都不缓冲，应该怎么做？

使用操纵符unitbuf：

* 1. cout << unitbuf;  //所有输出操作后都会立即刷新缓冲区
  2. //自此期间，任何输出都立即刷新，无缓冲
  3. cout << nounitbuf;  //回到正常的缓冲方式

# 关联了两个流有什么作用？

一般会将 输出流 和 输入流 关联，此时任何尝试 从输入流尝试读取数据的操作 都将首先刷新其关联的输出流缓冲区。例如标准库就将cin和cout关联在了一起

# 为什么要关联两个流？

拿交互式系统来说，我们希望所有输出（包括用户提示信息）都能在读操作之前被打印出来，此时就应该关联输入流和输出流。

# 如何关联两个流？

用tie函数，它有两个重载的版本：

1. //返回指向绑定的输出流的指针。
2. ostream\* tie ( ) **const**;
4. //将tiestr指向的输出流绑定的该对象上，并返回上一个绑定的输出流指针。
5. ostream\* tie ( ostream\* tiestr );

例子：

1. ostream\* s = cin.tie();//获取cin绑定的流（正常来说应该是cout）
2. ostream \*old\_tie = cin.tie(&cerr);//将流cin绑定到流cerr，并返回的是cin之前绑定的流

c++ primer的例子：

1. cin.tie(&cout);  //仅仅是用来展示，标准库已经将 cin 和 cout 关联在一起
2. //old\_tie 指向当前关联到 cin 的流（如果有的话）
3. ostream \*old\_tie = cin.tie(nullptr);  // cin 不再与其他流关联
4. //将 cin 与 cerr 关联，这不是一个好主意，因为 cin 应该关联到 cout
5. cin.tie(&cerr);  //读取 cin 会刷新 cerr 而不是 cout
6. cin.tie(old\_tie);  //重建 cin 和 cout 间的正常关联

# 如何取消流的关联？

调用tie函数，传一个空指针进去：

* 1. ostream \*old\_tie = cin.tie(nullptr);  // cin 不再与其他流关联

# 关联两个流的时候有什么限制吗？

（1）只能绑定到ostream类型的流上，不能绑定到istream流上：

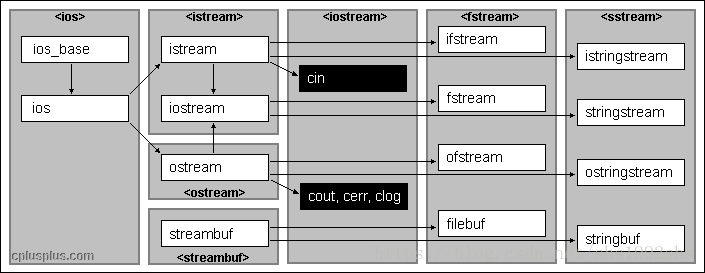
注意看，tie函数的形参类型为ostream ，这就意味着只能接受ostream 形参：

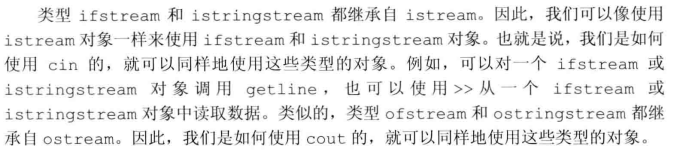
ostream\* tie ( ostream\* tiestr ); // 形参类型为ostream！

（2）一个流只能关联到一个流，要想关联到新流上，必须与旧流解绑；

（3）多个流可以同时关联到同一个ostream

# C++ 标准库提供的几种流的继承关系是？





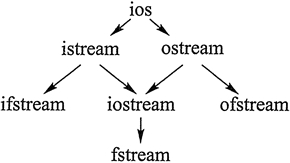
# C++用什么来操作文件？

C++ 标准库中提供了 3 个类用于实现文件操作，它们统称为文件流类，这 3 个类分别为：

* ifstream：专用于从文件中读取数据；
* ofstream：专用于向文件中写入数据；
* fstream：既可用于从文件中读取数据，又可用于向文件中写入数据。

这 3 个文件流类都位于 <fstream> 头文件中，因此在使用它们之前，程序中应先引入此头文件。

# C++ 文件流类 的继承关系是怎样的？



# C++ 有没有为 文件流类定义类似于 cin和cout这样的对象？为什么？

## 有没有？

和 <iostream> 头文件中定义有 ostream 和 istream 类的对象 cin 和 cout 不同，<fstream> 头文件中并没有定义可直接使用的 fstream、ifstream 和 ofstream 类对象。因此，如果我们想使用该类操作文件，需要自己创建相应类的对象。

## 为什么？

为什么 C++ 标准库不提供现成的类似 fin 或者 fout 的对象呢？

① 标准输入/输出一般都是从终端读取或写入，很容易确定；

② 而文件输入流和输出流的输入输出设备是硬盘中的文件，硬盘上有很多文件，到底应该使用哪一个呢？所以，C++ 标准库就把创建文件流对象的任务交给用户了。

# 如何判断文件是否打开成功？

* 1. ofstream out;
  2. out.open("file.txt");
  3. out.close();
  4. if(out){ //若文件打开成功
  5. //Do something here.
  6. }

# 如何以追加模式打开一个文件file.txt ?

(1) 先创建一个ofstream流对象，然后调用open成员函数，同时设置追加ofstream::app)模式

1. ofstream out; // no file mode is set
2. out.open("file.txt", ofstream::app); // mode is out and app
3. out.close();

(2) 创建ofstream流对象的时候直接用构造函数打开文件，并指定加ofstream::app)模式：

1. // 也可以不显示声明为ofstream::out模式，因为ofstream隐含了ofstream::out模式
2. ofstream app("file2", ofstream::out | ofstream::app);

# 打开文件的时候不指定 打开模式会发生什么？

每个 **文件流类型** 都定义了一个默认的文件模式，当我们未指定文件模式时，就使用此默认模式：

ifstream： in模式；

ofstream：out模式；

fstream： in和out模式

# ofstream out("file1");  是以什么模式打开文件？

默认情况下，当我们用ofstream打开一个文件时，将隐式的包含以下模式：

① ofstream::out

这个很好理解，ofstream肯定是以写入（out）模式打开文件；

② ofstream::trunc

用ofstream打开一个文件时，也将隐式的截断文件（即文件中原来的数据将丢失），所以要想保留文件中已有的数据必须以追加模式打开文件

也就是说，下面三个语句的作用是一样的：

1. ofstream out("file1"); // out and trunc are implicit
2. ofstream out2("file1", ofstream::out); // trunc is implicit
3. ofstream out3("file1", ofstream::out | ofstream::trunc);

# 调用open()成员函数的时候若不指定打开模式，会发生什么？

在每次打开文件的时候，都要设置文件模式，可以是显式地设置，也可能是隐式的设置，当程序未指定模式时，就使用默认值：

1. ofstream out; // no file mode is set
2. out.open("scratchpad"); // 文件打开模式 隐式的设置为out 和trunc
3. out.close(); // close out so we can use it for a different file
4. out.open("precious", ofstream::app); // 文件打开模式显式地设置为 out and app
5. out.close();

# 对于fstream对象，close()成员函数什么时候会被自动调用？

当一个fstream对象被销毁的时候，close()会被自动调用，比如在局部作用域的时候：

1. // for each file passed to the program
2. **for** (auto p = argv + 1; p != argv + argc; ++p) {
3. ifstream input(\*p); // 创建输出流，并打开文件
4. **if** (input) { // 如果文件打开成功，则调用process()函数进行处理
5. process(input);
6. } **else**
7. cerr << "couldn't open: " + string(\*p);
8. } // 每次迭代中，流input都会离开作用域，因此它会被销毁，此时会自动调用close()

# 二进制文件 和 文本文件 有什么区别？

**轮子哥：不存在文本文件这种东西，那其实就是某些软件知道怎么去解码成字符串的二进制文件。**

计算机的存储在物理上是二进制的 所以文本文件与二进制文件的区别并不是物理上的 而是逻辑上的，把文本文件看成一种二进制文件的就行了，无论是文本文件 还是unicode 和ascii 这些都是文件的解释方式二进制文件是基于值编码的文件，你可以根据具体应用，指定某个值是什么意思（这样一个过程，可以看作是自定义编码。

**（1）文本文件**

文本文件存储的是常规字符串，由若干文本行组成，通常每行以换行符'\n'结尾。常规字符串是指记事本或其他文本编辑器能正常显示、编辑并且人类能够直接阅读和理解的字符串，如英文字母、汉字、数字字符串。文本文件可以使用字处理软件如gedit、记事本进行编辑。

**（2）二进制文件**

二进制文件把对象内容以字节串(bytes)进行存储，无法用记事本或其他普通字处理软件直接进行编辑，通常也无法被人类直接阅读和理解，需要使用专门的软件进行解码后读取、显示、修改或执行。常见的如图形图像文件、音视频文件、可执行文件、资源文件、各种数据库文件、各类office文档等都属于二进制文件。

## 以下摘自一篇博文：

程序员经常说：“哥，你也别用明文写文件啊，至少也要写成二进制文件啊”。

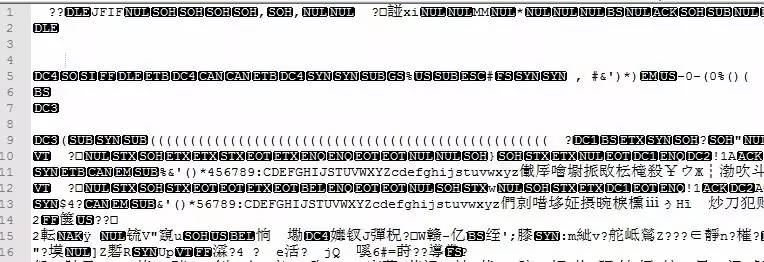
程序员经常说：“哥，这篇文章数字居多，不要写成文本文件哦，好占空间啊”。

程序员经常说：“哥，你是不明白文本文件和二进制文件的区别吧 ：—）”。

带着这些常见的问题，果果带你走进科学，看看文本文件和二进制文件的本质区别以及使用场景。

计算机中的文本文件就指的是你常见到的txt，记事本文件这种，在windows中打开，你是直接可阅读，并可解释其含义的。

而二进制文件通常你用文本打开工具是不能打开的，我们用记事本强行打开，也是一团乱码，下图应该是你常见的，不信你用NotePad等工具打开一张图片看看：



其实，从广义的存储的角度看，计算机中本没有什么文本文件和二进制文件的区别，在计算机的硬盘上存储的文件都是以二进制存储的，也就是01的串。

那为什么程序员口中又要分这两种类型呢？区别何在呢？其实是从狭义的角度划分，我们还是举栗子进行说明：

圆周率π=3.1415926 ，如果按照文本文件存储（在桌面上新建一个txt，然后输入3.1415926，然后保存），这个文件就被存储为一个文本文件，其中一共9个字符，分别是3、.、1、4、1、5、9、2、6，这几个数字分别按照其对应的ASCII码为十进制的63，56，61，64，61，65，71，62，66，每个字符占用一个字节，所以一共占用了9个字节的空间。

如果按照二进制文件存储，那3.1415926是一个浮点数，那最终占用4个字节存储。

可以明显的推导出一个结论：二进制文件在数字上存储要比文本文件省空间，也就是文本文件是按照字符存储，二进制文件按照数据类型存储。

文本文件最终存储的也是二进制文件，只不过每一个字节都是可以转换为相应的字符的，因为要保障其可以还原，而二进制文件根本不关心存储的是什么，就像吃火锅往火锅里面下菜的时候，文本文件像个大家闺秀一样，还要区分蔬菜放在不辣的里面，肉放在辣锅里面一样，效率当然低，而二进制文件不管三七二十一，不按任何规则，只要保证菜品入锅就好了。

就像程序员说的，文本文件打开就是明文，而二进制文件是不定长的，而且存储的是时候，你不知道写入的程序员是按照什么规则写入的，所以会增加一点破解难度。

总结起来，二进制文件更省空间，写入速度更快，因为可读性很差，所以还有一定的加密保护作用。

因为从存储的角度，本来一切公平，大家都是二进制存储的。但是因为人要读文件，所以文本文件委屈求全，作为二进制文件的子集，文本文件开辟了一个新的文件品类，这种品类下，文件的每个字符都是经过了特殊处理（比如转成ASCII码）然后再存储为二进制，这样的二进制因为可以直接对应为ASCII码，所以可供人们阅读。

在程序设计中，经常利用文件流进行二进制文件的读写，程序员会经常跟二进制文件打交道，而且二进制文件的格式经常是程序员自定义的，希望后面你听到这个词的时候，不要太陌生，只把它当作一个普通文件即可。

# 如何打开二进制文件？

指定ofstream::binary模式。

# 写一个程序，从file1读取数据，写到file2里

# 看完了fstream后看这个

<https://blog.csdn.net/qq100440110/article/details/51056306>