第三章 C++输入输出流

- **输入输出的含义**

以前所用到的输入和输出，都是以终端为对象的，即从键盘输入数据，运行结果输出到显示器屏幕上。从操作系统的角度看，每一个与主机相连的输入输出设备都被看作一个**文件**。除了以终端为对象进行输入和输出外，还经常用磁盘(光盘)作为输入输出对象，磁盘文件既可以作为**输入文件**，也可以作为**输出文件**。

在编程语言中的输入输出含义有所不同。**程序的输入**指的是从输入文件将数据传送给程序(内存)，**程序的输出**指的是从程序(内存)将数据传送给输出文件。

- **C++输入输出流机制**

C++ 的 I/O 发生在流中，流是字节序列。如果字节流是从设备（如键盘、磁盘驱动器、网络连接等）流向内存，这叫做输入操作。如果字节流是从内存流向设备（如显示屏、打印机、磁盘驱动器、网络连接等），这叫做输出操作。

就 C++ 程序而言， I/O 操作可以简单地看作是从程序移进或移出字节，程序只需要关心是否正确地输出了字节数据，以及是否正确地输入了要读取字节数据，特定 I/O 设备的细节对程序员是隐藏的。

- **C++常用流类型**

C++ 的输入与输出包括以下3方面的内容：

（1） 对系统指定的标准设备的输入和输出。即从键盘输入数据，输出到显示器屏幕。这种输入输出称为标准的输入输出，简称**标准** I/O 。

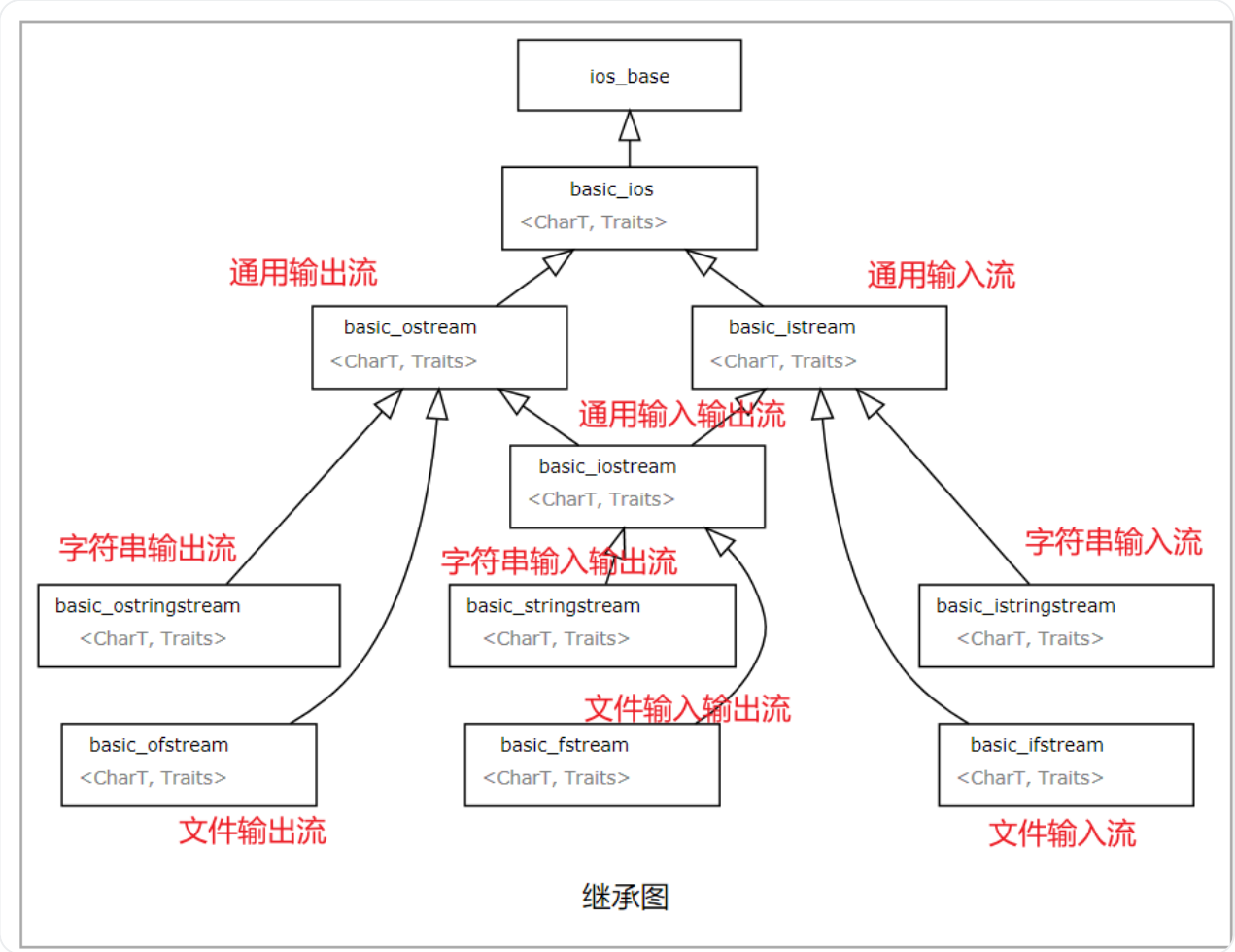
（2） 以外存磁盘文件为对象进行输入和输出，即从磁盘文件输入数据，数据输出到磁盘文件。以外存文件为对象的输入输出称为文件的输入输出，简称**文件** I/O 。

（3） 对内存中指定的空间进行输入和输出。通常指定一个字符数组作为存储空间（实际上可以利用该空间存储任何信息）。这种输入和输出称为字符串输入输出，简称**串** I/O 。

常用的输入输出流如下：

类名	作用	头文件
istream	通用输入流	iostream
ostream	通用输出流	iostream

iostream	通用输入输出流	iostream
ifstream	文件输入流	fstream
oftream	文件输出流	fstream
fstream	文件输入输出流	fstream
istringstream	字符串输入流	sstream
ostringstream	字符串输出流	sstream
stringstream	字符串输入输出流	sstream



继承图

流的四种状态（重点）

IO 操作与生俱来的一个问题是可能会发生错误，一些错误是可以恢复的，另一些是不可以的。在C++ 标准库中，用 iostate 来表示流的状态，不同的编译器 iostate 的实现可能不一样，不过都有四种状态：

- **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。
- **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。
- **eofbit**表示**到达流结尾位置**， 此时eofbit 和 failbit 都会被置位。
- **goodbit** 表示流处于**有效状态**。流在有效状态下，才能正常使用。如果 badbit 、 failbit 和 eofbit 任何一个被置位，则流无法正常使用。

这四种状态都定义在类 ios_base 中，作为其数据成员存在。在 GNU GCC7.4 的源码中，流状态的实现

如下：

ios_base.h

```
153   enum _Ios_Iostate
154     {
155       _S_goodbit         = 0,
156       _S_badbit          = 1L << 0,
157       _S_eofbit          = 1L << 1,
158       _S_failbit         = 1L << 2,
159       _S_ios_iostate_end = 1L << 16,
160       _S_ios_iostate_max = __INT_MAX__,
161       _S_ios_iostate_min = ~__INT_MAX__
162     };
```

通过流的状态函数实现

```
1  bool good() const       //流是goodbit状态，返回true，否则返回false
2  bool bad() const        //流是badbit状态，返回true，否则返回false
3  bool fail() const       //流是failbit状态，返回true，否则返回false
4  bool eof() const        //流是eofbit状态，返回true，否则返回false
```

标准输入输出流

对系统指定的标准设备的输入和输出。即从键盘输入数据，输出到显示器屏幕。这种输入输出称为标准输入输出，简称**标准** I/O

C++标准库定义了三个预定义的标准输入输出流对象，分别是 `std::cin`、`std::cout` 和 `std::cerr`。它们分别对应于标准输入设备（通常是键盘）、标准输出设备（通常是显示器）和标准错误设备（通常是显示器）。

标准输入、输出的内容包含在头文件iostream中。

有时候会看到通用输入输出流的说法，这是一个更广泛的概念，可以与各种类型的输入输出设备进行交互，包括标准输入输出设备、文件、网络等。

标准输入流

istream 类定义了一个全局输入流对象，即 cin，代表的是**标准输入**，它从标准输入设备(键盘)获取数据，程序中的变量通过流提取符 “>>”（输入流符号） 从流中提取数据。

流提取符 “>>” 从流中提取数据时通常跳过输入流中的空格、 tab 键、换行符等空白字符。只有在输入完数据再按回车键后，该行数据才被送入键盘**缓冲区**，形成输入流，提取运算符 “>>” 才能从中提取数据。需要注意保证从流中读取数据能正常进行。（流的缓冲机制在下一节中学习）

下面来看一个例子，每次从 cin 中获取一个字符：

```
void printStreamStatus(std::istream & is){
    cout << "is's goodbit:" << is.good() << endl;
    cout << "is's badbit:" << is.bad() << endl;
    cout << "is's failbit:" << is.fail() << endl;
    cout << "is's eofbit:" << is.eof() << endl;
}

void test0(){
    printStreamStatus(cin);  //goodbit状态
    int num = 0;
    cin >> num;
    cout << "num:" << num << endl;
    printStreamStatus(cin);  //进行一次输入后再检查cin的状态
}
```

如果没有进行正确的输入，输入流会进入failbit的状态，无法正常工作，需要恢复流的状态。

查看C++参考文档，需要利用**clear和ignore**函数配合，实现这个过程

```
   if(!cin.good()){
       //恢复流的状态
       cin.clear();
       //清空缓冲区，才能继续使用

cin.ignore(std::numeric_limits<std::streamsize>::max(),'\n');
       cout << endl;
       printStreamStatus(cin);
   }
```

```
void test0(){
    int num = 10;
    cout << "执行输入操作前检查流的状态" << endl;
    checkStreamStatus(cin);

    cout << endl;
    cin >> num;
    cout << "执行输入操作后检查流的状态" << endl;
    checkStreamStatus(cin);

    cout << endl;
    if(!cin.good()){
        //恢复流的状态
        cin.clear();
        //还需要清空缓冲区，才能继续使用
        /* cin.ignore(1024,'\n'); */
        cin.ignore(std::numeric_limits<std::streamsize>::max(),'\n');
        checkStreamStatus(cin);
    }

    string line;
    cin >> line;
    cout << "line:" << line << endl;


    //没有进行正常输入，会将num设为0
    cout << "num:" << num << endl;
```

思考，如何完成一个输入整型数据的实现（如果是非法输入则继续要求输入）

```
void InputInteger(int num){
    cout << "请输入一个int型数据" << endl;
    //逗号表达式整体的值为最后一个逗号之后的表达式的值
    while(cin >> num, !cin.eof()){
        if(cin.bad()){
            cout << "cin has broken!" << endl;
            return;
        }else if(cin.fail()){
            cin.clear();
            cin.ignore(std::numeric_limits<std::streamsize>::max(),'\n');
            cout << "请输入一个int型数据" << endl;
        }else{
            //输入是合法的
            cout << "num:" << num << endl;
            break;
        }
    }
}
```

```cpp
/* if(cin.good()){ */
if(cin){
    cout << "hello" << endl;
}

//cin >> number这个表达式的返回值就是cin
//cin对象是good状态时就可以视为true,还可以继续正常工作
cin >> number >> number2;
cout << number << endl;
cout << number2 << endl;
```

缓冲机制

在标准输入输出流的测试中发现，流有着缓冲机制。**缓冲区**又称为缓存，它是内存空间的一部分。也就是说，在内存空间中预留了一定的存储空间，这些存储空间用来缓冲输入或输出的数据，这部分预留的空间就叫做缓冲区。缓冲区根据其对应的是输入设备还是输出设备，分为**输入缓冲区**和**输出缓冲区**。

输入或输出的内容会存在流对象对应的缓冲区，在特定情景下会从缓冲区释出。

- **为什么要引入缓冲区?**

  比如我们从磁盘里取信息，我们先把读出的数据放在缓冲区，计算机再直接从缓冲区中取数据，等缓冲区的数据取完后再去磁盘中读取，这样就可以减少磁盘的读写次数，再加上计算机对缓冲区的操作大大快于对磁盘的操作，故应用缓冲区可大大提高计算机的运行速度。

  又比如，我们使用打印机打印文档，由于打印机的打印速度相对较慢，我们先把文档输出到打印机相应的缓冲区，打印机再自行逐步打印，这时我们的 CPU 可以处理别的事情。因此缓冲区就是一块内存区，它用在输入输出设备和 CPU 之间，用来缓存数据。它使得低速的输入输出设备和高速的CPU 能够协调工作，避免低速的输入输出设备占用 CPU，解放出 CPU，使其能够高效率工作。

- **缓冲区要做哪些工作?**

从上面的描述中，不难发现缓冲区向上连接了程序的输入输出请求，向下连接了真实的 I/O 操作。作为中间层，必然需要分别处理好与上下两层之间的接口，以及要处理好上下两层之间的协作。

输入或输出的内容会存在流对象对应的缓冲区，在特定情景下会从缓冲区释出。

- **缓冲机制**

  缓冲机制分为三种类型：**全缓冲、行缓冲和不带缓冲**。

  全缓冲：在这种情况下，当填满缓冲区后才进行实际 I/O 操作。全缓冲的典型代表是对磁盘文件的读写。

  行缓冲：在这种情况下，当在输入和输出中遇到换行符时，执行真正的 I/O 操作。这时，我们输入的字符先存放在缓冲区，等按下回车键换行时才进行实际的 I/O 操作。典型代表是cin。

  不带缓冲：也就是不进行缓冲，有多少数据就刷新多少。标准错误输出 cerr是典型代表，这使得出错信息可以直接尽快地显示出来。

cout既有全缓冲的机制，又有行缓冲的机制；cin通常体现行缓冲机制；cerr属于不带缓冲机制，通常用于处理错误信息。

标准输出流

ostream 类定义了全局输出流对象 cout，即标准输出，在缓冲区刷新时将数据输出到终端。

如下几种情况会导致输出缓冲区内容被刷新：

1. **程序正常结束**；

马上输出了1025个a

```
void test0(){
    for(int i = 0; i < 1025; ++i){
        cout << 'a';
    }
}
```

2. **缓冲区满**；

马上输出了1024个a,等待2秒后输出了最后一个a

（在实验环境中cout对象的默认缓冲区大小是1024个字节，缓冲区满了就刷新出了所有内容，后面还有一个字符，就要等程序正常结束时刷新出来）

```
void test0(){
    for(int i = 0; i < 1025; ++i){
        cout << 'a';
    }
    sleep(2);
}
```

3. 使用**操纵符**显式地刷新输出缓冲区，如endl ；

加上endl这种操作符，直接输出了5个a，等待2秒程序结束；如果不加endl，等待2秒程序结束时才会输出5个a

```
void test0(){
    for(int i = 0; i < 5; ++i){
        cout << 'a' << endl;
    }
    sleep(2);
}
```

—— 查看ostream头文件中endl的定义（刷新缓冲区 + 换行）

```
ray@ubuntu:/usr/include/c++/7$ vim ostream
ray@ubuntu:/usr/include/c++/7$
```

```
template<typename _CharT, typename _Traits>
  inline basic_ostream<_CharT, _Traits>&
  endl(basic_ostream<_CharT, _Traits>& __os)
  { return flush(__os.put(__os.widen('\n'))); }
```

来看一个简单的例子：在使用cout时，如果在输出流语句末尾使用了endl函数，会进行换行，并刷新缓冲区

```
1  void test0(){
2      for(int i = 0; i < 1025; ++i){
3          cout << 'a' << endl;
4      }
5  }
```

如果在使用cout时，没有使用endl函数，键盘输入的内容会存在输出流对象的缓冲区中，当缓冲区满或遇到换行符时，将缓冲区刷新，内容传输到终端显示。可使用sleep函数查看缓冲的效果。

```
1  #include <unistd.h>
2  void test0(){
3      for(int i = 0; i < 1024; ++i){
4          cout << 'a';
5      }
6      sleep(2);
7      cout << 'b';
8      sleep(2);
9  }
```

GCC中标准输出流的默认缓冲区大小就是1024个字节。

如果不用sleep函数，即使没有endl或换行符，所有内容依然是直接输出

——因为程序执行完时也会刷新缓冲区。

- **关于操作符**

endl：用来完成换行，并刷新缓冲区

ends：在输入后加上一个空字符('\0')，然后再刷新缓冲区

flush：用来直接刷新缓冲区的　　cout.flush();

- **标准错误流**

ostream 类还定义了全局输出流对象 cerr，标准错误流（不带缓冲）

试试看如下的代码运行会有什么效果

```
#include <unistd.h>
void test1(){
    cerr << 1;
    cout << 3;
    sleep(2);

}
```

文件输入输出流（重点）

所谓“文件”，一般指存储在外部介质上数据的集合。一批数据是以文件的形式存放在外部介质上的。操作系统是以文件为单位对数据进行管理的。要向外部介质上存储数据也必须先建立一个文件（以文件名标识），才能向它输出数据。外存文件包括磁盘文件、光盘文件和U盘文件。目前使用最广泛的是磁盘文件。

文件流是以外存文件为输入输出对象的数据流。

文件输入流是从外存文件流向内存的数据，**文件输出流**是从内存流向外存文件的数据。每一个文件流都有一个内存缓冲区与之对应。**文件流**本身不是文件，而只是以文件为输入输出对象的流。若要对磁盘文件输入输出，就必须通过文件流来实现。

C++ 对文件进行操作的流类型有三个：

ifstream（文件输入流）

ofstream（文件输出流）

fstream （文件输入输出流）

他们的构造函数形式都很类似：

```
ifstream();
explicit ifstream(const char* filename, openmode mode =
ios_base::in);
explicit ifstream(const string & filename, openmode mode =
ios_base::in);

ofstream();
explicit ofstream(const char* filename, openmode mode =
ios_base::out);
explicit ofstream(const string & filename, openmode mode =
ios_base::out);

fstream();
explicit fstream(const char* filename, openmode mode =
ios_base::in|out);
explicit fstream(const string & filename, openmode mode =
ios_base::in|out);
```

补充：explicit关键字的意义 —— 禁止隐式转换

```
class Point{
public:
    //这个关键字加到隐式转换时需要调用的构造函数前
    //禁止隐式转换
    explicit
    Point(int x = 0, int y = 0)
    : _ix(x)
    , _iy(y)
    { cout << "Point(int,int)" << endl; }
```

```
    //隐式转换
    //Point pt6 = Point(1);
    Point pt6 = 1;
    pt6.print();

```

文件输入流

文件输入流对象的创建

首先我们要明确使用文件输入流的信息传输方向：文件 --》 文件输入流对象的缓冲区 --》 程序中的数据结构

根据上述的说明，我们可以将输入流对象的创建分为两类：

1. 可以使用无参构造创建ifstream对象，再使用open函数将这个文件输入流对象与文件绑定（若文件不存在，则文件输入流进入failbit状态）；
2. 也可以使用有参构造创建ifstream对象，在创建时就将流对象与文件绑定，后续操作这个流对象就可以对文件进行相应操作。

通过参考文档中对ifstream的构造函数的描述，文件输入流对象的有参构造需要输入文件名，可以指定打开模式（不指定则使用in模式，为读打开）

```cpp
#include <fstream>
void test0(){
    ifstream ifs;
    ifs.open("test1.cc");

    ifstream ifs2("test2.cc");

    string filename = "test3.cc";
    ifstream ifs3(filename);
}
```

```cpp
//默认以换行符、空格作为间隔符
//一次读取一个字符串
string word;
//只要ifs是goodbot状态就会一直读取
while(ifs >> word){
    cout << word << endl;
}

//规范操作，使用完之后关闭流
ifs.close();
```

- **文件模式**

根据不同的情况，对文件的读写操作，可以采用不同的文件打开模式。文件模式在 GNU GCC7.4 源码实现中，是用一个叫做 openmode 的枚举类型定义的，它位于 ios_base 类中。文件模式一共有六种，它们分别是：

in：输入，文件将允许做读操作；如果文件不存在，打开失败

out：输出，文件将允许做写操作；如果文件不存在，则直接创建一个

app：追加，写入将始终发生在文件的末尾

ate：末尾，写入最初在文件的末尾

trunc：截断，如果打开的文件存在，其内容将被丢弃，其大小被截断为零

binary：二进制，读取或写入文件的数据为二进制形式

ios_base.h

```
111   enum _Ios_Openmode
112     {
113       _S_app        = 1L << 0,
114       _S_ate        = 1L << 1,
115       _S_bin        = 1L << 2,
116       _S_in         = 1L << 3,
117       _S_out        = 1L << 4,
118       _S_trunc      = 1L << 5,
119       _S_ios_openmode_end = 1L << 16,
120       _S_ios_openmode_max = __INT_MAX__,
121       _S_ios_openmode_min = ~__INT_MAX__
122     };
```

按行读取

方法一：使用ifstream类中的成员函数getline，这种方式是兼容C的写法

std::basic_istream<CharT,Traits>::getline

```
basic_istream& getline( char_type* s, std::streamsize count );          (1)
basic_istream& getline( char_type* s, std::streamsize count, char_type delim );    (2)
```

参数

s	-	指向要存储字符到的字符串的指针
count	-	s 所指向的字符串的大小
delim	-	释出所终止于的分隔字符。释出但不存储它。

```
1  ifstream ifs("test.cc");
2  //方法一，兼容C的写法，使用较少
3  char buff[100] = {0};
4  while(ifs.getline(buff,sizeof(buff))){
5      cout << buff << endl;
6      memset(buff,0,sizeof(buff));
7  }
```

准备好一片空间存放一行的内容，但是有一个弊端就是我们并不知道一行的内容会有多少个字符，如果超过了设置的字符长度将无法完成该行的读取，也将跳出循环。

方法二：

使用<string>提供的getline方法，**工作中更常用**

std::getline

定义于头文件 <string>

```
template< class CharT, class Traits, class Allocator >
std::basic_istream<CharT,Traits>& getline( std::basic_istream<CharT,Traits>& input,
                                           std::basic_string<CharT,Traits,Allocator>& str,
                                           CharT delim );                                    (1)
```

```
template< class CharT, class Traits, class Allocator >
std::basic_istream<CharT,Traits>& getline( std::basic_istream<CharT,Traits>& input,
                                           std::basic_string<CharT,Traits,Allocator>& str );  (2)
```

2) 同 `getline(input, str, input.widen('\n'))` ，即默认分隔符是换行符。

传入输入流对象、string、分隔符（默认换行符为分隔符）

```
//更方便，使用更多
string line;
while(getline(ifs,line)){
    cout << line << endl;
}
```

将一行的内容交给一个string对象去存储，不用再关心字符数了。

```
#if 1
    //方式一：兼容C的写法
    char buff[100] = {0};
    while(ifs.getline(buff,sizeof(buff))){
        cout << buff << endl;
        memset(buff,0,sizeof(buff));
    }
#endif

    //方式二：使用C++的string的getline函数
    string line;
    while(std::getline(ifs,line)){
        cout << line << endl;
    }


    //规范操作，使用完之后关闭流
    ifs.close();
}
```

读取指定字节数的内容

read函数 + seekg函数 + tellg函数

通过文件输入流对象读取到的内容交给字符数组，同时需要传入要读取的字符数

std::basic_istream<CharT,Traits>::read

```
basic_istream& read( char_type* s, std::streamsize count );
```

参数

s	-	指向要存储字符到的字符数组的指针
count	-	要读取的字符数

要知道字符数就需要用上tellg函数了，可以这样理解，从文件中读取内容时存在一个文件游标，读取是从文件游标的位置开始读取的。tellg就是用来获取游标位置的，而seekg则是用来设置游标位置的。

std::basic_istream<CharT,Traits>::tellg

```
pos_type tellg();
```

返回当前关联的 streambuf 对象的输入位置指示器。

调用seekg时有两种方式，一种是绝对位置（比如将游标设为流的开始位置，可以直接传参数0）；一种是相对位置，传入偏移量和基准点——第一个参数：相对基准点需要向前偏移则传入负数，不偏移则传入0，需要向后偏移则传入正数。第二个参数格式为**std::ios::beg**(以流的开始位置为例)

std::basic_istream<CharT,Traits>::seekg

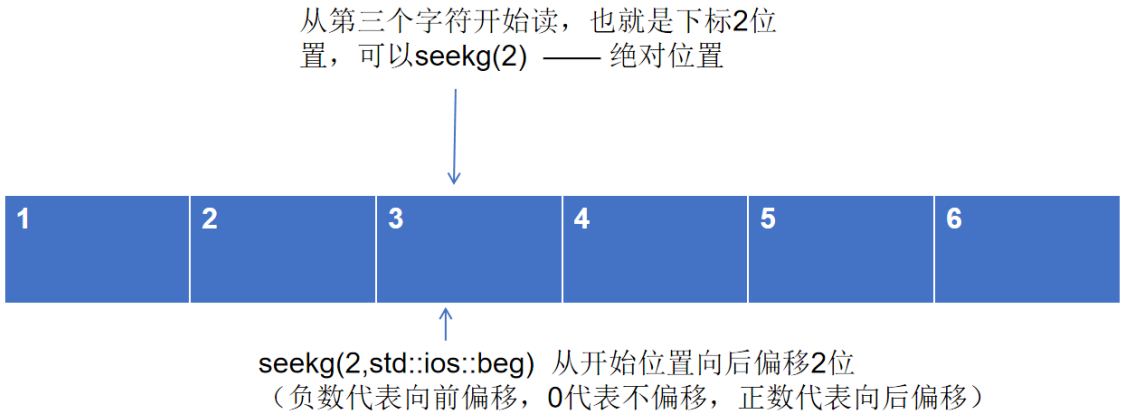
```
basic_istream& seekg( pos_type pos );
basic_istream& seekg( off_type off, std::ios_base::seekdir dir);
```

参数

pos	-	设置输入位置指示器到的绝对位置。
off	-	设置输入位置指示器到的相对位置。
dir	-	定义应用相对偏移到的基位置。它能为下列常量之一：

常量	解释
beg	流的开始
end	流的结尾
cur	流位置指示器的当前位置

如图示：



read(char * s , size_t count)

tellg()　获取游标位置

seekg函数设置游标位置
（1）seekg(size_t pos)　绝对位置
（2）seekg(off, dir)　相对位置　off代表偏移量，dir代表锚点（游标偏移的基准点）

dir - 定义应用相对偏移到的基位置。它能为下列常量之一：

常量	解释
beg	流的开始
end	流的结尾
cur	流位置指示器的当前位置

例子：读取一个文件的全部内容

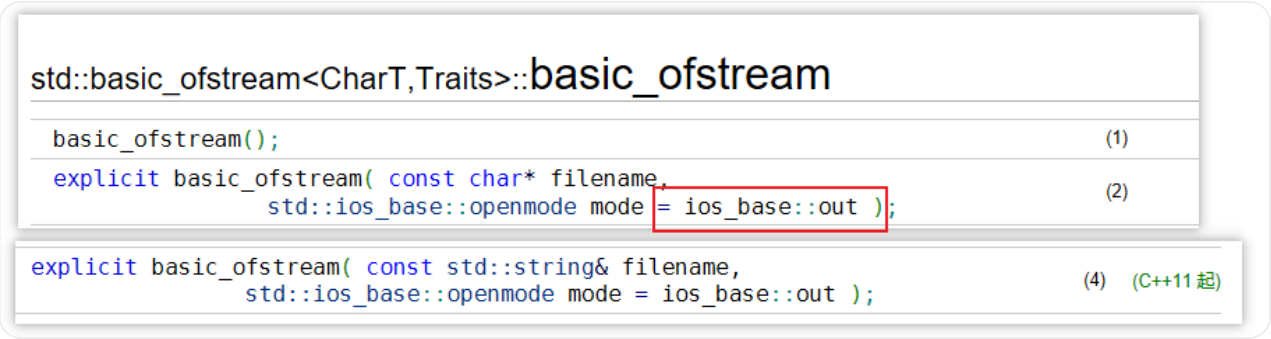
```
void test0(){
    string filename = "test.cc";
    ifstream ifs(filename);

    if(!ifs){
        cerr << "ifs open file fail!";
        return;
    }

    //读取一个文件的所有内容先要获取文件的大小
    //将游标放到了文件的最后（尾后）
    fs.seekg(0,std::ios::end);
    long length = ifs.tellg();//获取尾后下标，实际就是总的字符数
    cout << length << endl;

    char * pdata = new char[length]();
    //需要将游标再放置到文件开头
    ifs.seekg(0,std::ios::beg);
    ifs.read(pdata,length);

    //content包含了文件的所有内容，包括空格、换行
    string content(pdata);
    cout << "content:" << content << endl;
    /* cout << pdata << endl; */
    ifs.close();
```

```
26  }
27
```

还可以在创建输入流对象时指定ate模式，省去第一步将游标置流末尾处的操作。

文件输出流

文件输出流的作用是将流对象保存的内容传输给文件



ofstream对象的创建与ifstream对象的创建类似

```
1   #include <fstream>
2   void test0(){
3       ofstream ofs;
4       ofs.open("test1.cc");
5
6       ofstream ofs2("test2.cc");
7
8       string filename = "test3.cc";
9       ofstream ofs3(filename);
10  }
```

推测一下，如果文件输出流对象绑定的文件不存在，可以吗？

—— 可以，如果文件不存在，就创建出来

- **通过输出流运算符写内容**

ofstream对象绑定文件后，可以往该文件中写入内容

```
1   string filename = "test3.cc";
2   ofstream ofs3(filename);
3
4   string line("hello,world!\n");
5   ofs << line;
6
7   ofs.close();
```

内容传输的过程是string中的内容传给ofs对象，再传给这个对象绑定的文件。

但是我们会发现进行多次写入，并没有保留下多次的内容，因为这种创建方式会使打开模式默认为std::ios::out，**每次都会清空文件的内容**。

为了实现在文件流结尾追加写入内容的效果，可以在创建流对象时指定打开模式为**std::ios::app**（追加模式）

```
string filename = "test3.cc";
ofstream ofs3(filename, std::ios::app);
```

- **通过write函数写内容**

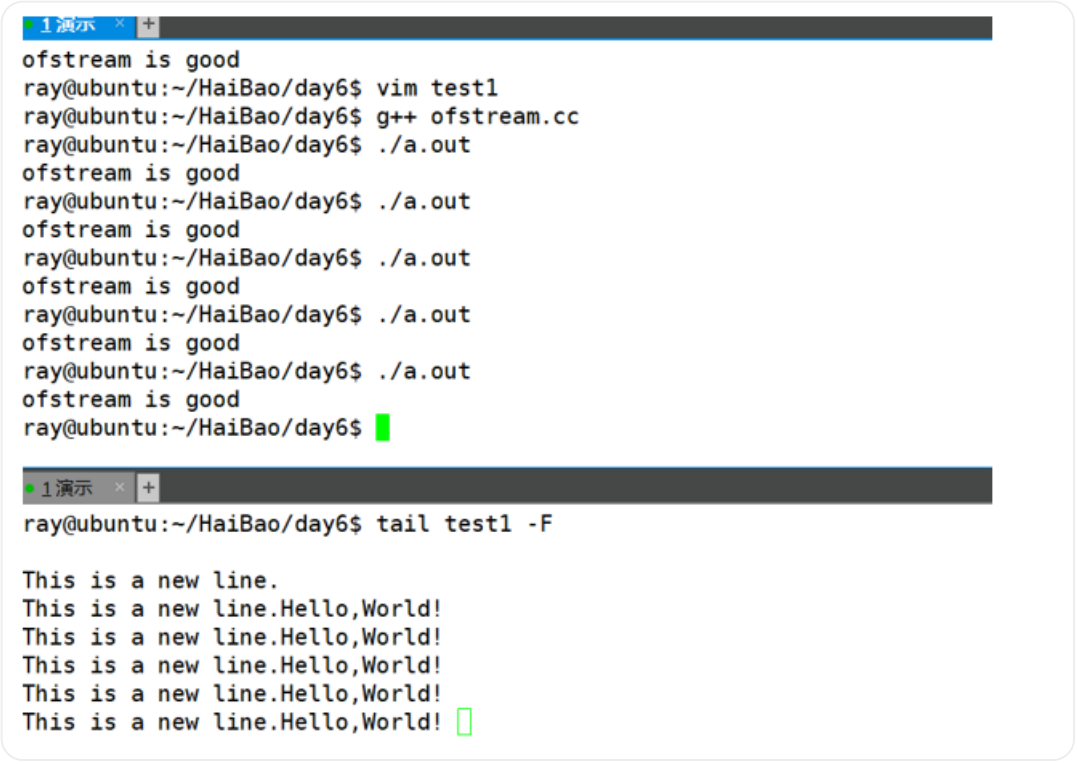
除了使用输出流运算符<< 将内容传输给文件输出流对象（传给ofstream对象就是将内容传到其绑定的文件中），还可以使用write函数进行传输


```
char buff[100] = "hello,world!";
ofs.write(buff,strlen(buff));
```

- **动态查看指令**

为了更方便地查看多次写入的效果（动态查看文件的内容）可以使用指令

```
tail 文件名 -F    //动态查看文件内容

ctrl + c          //退出查看
```



字符串输入输出流

字符串I/O是内存中的字符串对象与字符串输入输出流对象之间做内容传输的数据流，通常用来做格式转换。

C++ 对字符串进行操作的流类型有三个：

istringstream （字符串输入流）

ostringstream （字符串输出流）

stringstream （字符串输入输出流）

它们的构造函数形式都很类似：

```
istringstream(): istringstream(ios_base::in) { }
explicit istringstream(openmode mode = ios_base::in);
explicit istringstream(const string& str, openmode mode =
ios_base::in);

ostringstream(): ostringstream(ios_base::out) { }
explicit ostringstream(openmode mode = ios_base::out);
explicit ostringstream(const string& str, openmode mode =
ios_base::out);

stringstream(): stringstream(in|out) { }
explicit stringstream(openmode mode = ios_base::in|ios_base::out);
explicit stringstream(const string& str, openmode mode =
ios_base::in|ios_base::out);
```

字符串输入流

将字符串的内容传输给字符串输入流对象，再通过这个对象进行字符串的处理（解析）

创建字符串输入流对象时传入c++字符串，字符串的内容就被保存在了输出流对象的缓冲区中。之后可以通过输入流运算符将字符串内容输出给不同的变量，起到了字符串分隔的作用。

```
explicit basic_istringstream( const std::basic_string<CharT,Traits,Allocator>& str,
                              std::ios_base::openmode mode = std::ios_base::in );   (3)
```

——如下，将字符串s的内容传给了两个int型数据

```
void test0(){
    string s("123 456");
    int num = 0;
    int num2 = 0;
    //将字符串内容传递给了字符串输入流对象
    istringstream iss(s);
    iss >> num >> num2;
    cout << "num:" << num << endl;
    cout << "num2:" << num2 << endl;
}
```

因为输入流运算符会默认以空格符作为分隔符，字符串123 456中含有一个空格符，那么传输时会将空格前的123传给num，空格后的456传给num2，因为num和num2是int型数据，所以编译器会以int型数据来理解缓冲区释出的内容，将num和num2赋值为123和456

字符串输入流通常用来处理字符串内容，比如读取配置文件

```
//myserver.conf
ip 192.168.0.0
port 8888
dir ~HaiBao/53th/day06

//readConf.cc
void readConfig(const string & filename){
    ifstream ifs(filename);
    if(!ifs.good()){
        cout << "open file fail!" << endl;
        return;
    }

    string line;
    string key, value;
    while(getline(ifs,line)){
        istringstream iss(line);
        iss >> key >> value;
        cout << key << " -----> " << value << endl;
    }
}

void test0(){
    readConfig("myserver.conf");
}
```

```
void readConfig(const string & filename){
    ifstream ifs(filename);
    if(!ifs.good()){
        cout << "open file fail!" << endl;
        return;
    }

    string line;
    string key, value;
    //从文件中读取一行的内容保存到line
    while(getline(ifs,line)){
        //创建字符串输入流对象，接收line的内容
        istringstream iss(line);
        //从iss对象的缓冲区输入内容给key和value
        iss >> key >> value;
        cout << key << " -----> " << value << endl;
    }
}
```

字符串输出流

通常的用途就是将各种类型的数据转换成字符串类型

```
void test0(){
    int num = 123, num2 = 456;
    ostringstream oss;
    //把所有的内容都传给了字符串输出流对象
    oss << "num = " << num << " , num2 = " << num2 << endl;
    cout << oss.str() << endl;
}
```

将字符串、int型数据、字符串、int型数据统统传给了字符串输出流对象，存在其缓冲区中，利用它的str函数，全部转为string类型并完成拼接。