C++输入输出流

1、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。

2、C++输入输出机制

2.1、"流"的概念

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。

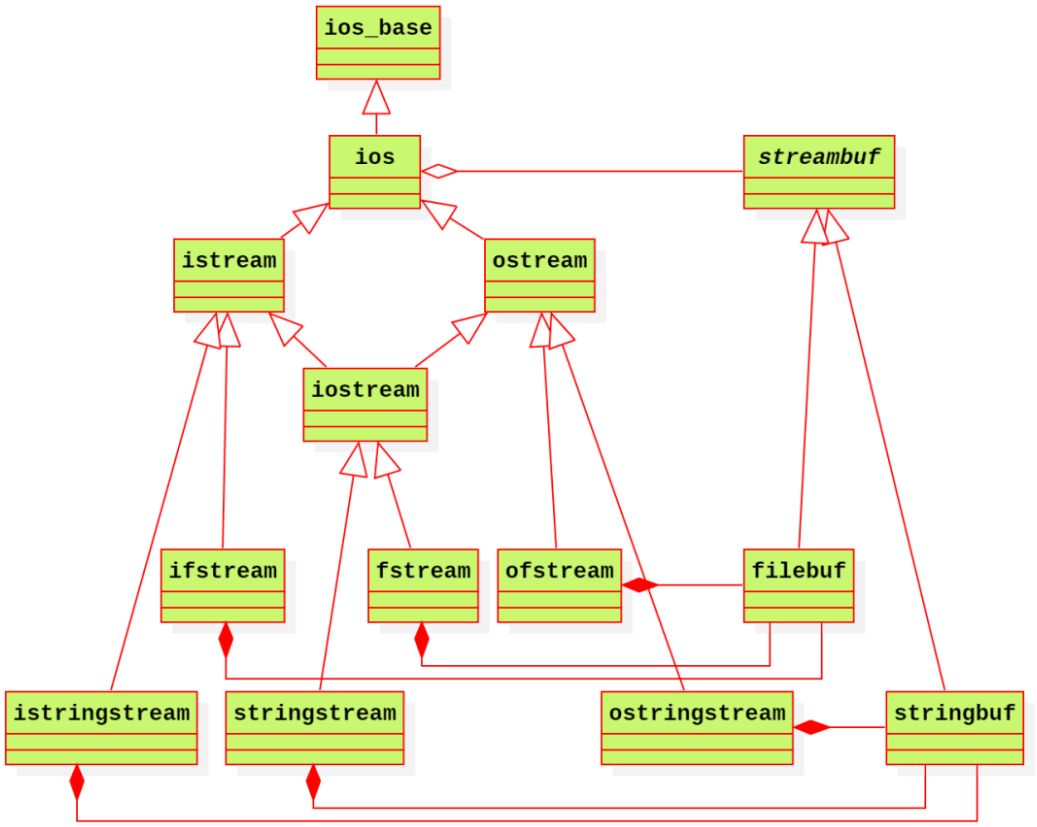
2.2、C++常用流类型

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**。

C++标准库提供了一组丰富的具有输入/输出功能的流类型。常用流类如下:

类名	作用	头文件
ios_base	抽象基类，管理格式化标志和输入/输出异常	iostream
ios	抽象基类，管理任意流缓冲	iostream
istream ostream iostream	通用输入流 通用输出流 通用输入输出流	iostream
ifstream ofstream fstream	文件输入流 文件输出流 文件输入输出流	fstream
istringstream ostringstream stringstream	字符串输入流 字符串输出流 字符串输入输出流	sstream

2.3、流类型之间的关系



ios是抽象基类，由它派生出istream类和ostream类，iostream类支持输入输出操作，iostream类是从istream类和ostream类通过多重继承而派生的类。类ifstream继承了类istream，类ofstream继承了类ostream，类fstream继承了类iostream.

2.4、流的状态

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

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     };
398     typedef _Ios_Iostate iostate;
399
400     /// Indicates a loss of integrity in an input or output sequence (
401     /// as an irrecoverable read error from a file).
402     static const iostate badbit =   _S_badbit;
403
404     /// Indicates that an input operation reached the end of an input
405     static const iostate eofbit =   _S_eofbit;
406
407     /// Indicates that an input operation failed to read the expected
408     /// characters, or that an output operation failed to generate the
409     /// desired characters.
410     static const iostate failbit =  _S_failbit;
411
412     /// Indicates all is well.
413     static const iostate goodbit =  _S_goodbit;
```

2.5、管理流的状态

C++标准库还定义了一组成员函数来查询或者操作这些状态。

```
bool bad() const;   //若流的badbit置位，则返回true;否则返回false
bool fail() const;  //若流的failbit或badbit置位，则返回true;
bool eof() const;   //若流的eofbit置位，则返回true;
bool good() const;  //若流处于有效状态，则返回true;
iostate rdstate() const;        //获取流的状态
void setstate(iostate state);   //设置流的状态

//clear的无参版本会复位所有错误标志位*(重置流的状态)
void clear(std::ios_base::iostate state = std::ios_base::goodbit);
```

2.6、流的通用操作

输入输出流同时还提供了一些其他通用的操作:

```
//----以下输入流操作----
int_type get();//读取一个字符
istream & get(char_type & ch);
//读取一行数据
istream & getline(char_type * s, std::streamsize count, char_type delim
='\n');

//读取count个字节的数据
istream & read(char_type * s, std::streamsize count);
//最多获取count个字节，返回值为实际获取的字节数
std::streamsize readsome(char_type * s, std::streamsize count);

//读取到前count个字符或在读这count个字符进程中遇到delim字符就停止，并把读取的这些东西丢
掉
istream & ignore(std::streamsize count = 1, int_type delim = Traits::eof());

//查看输入流中的下一个字符，但是并不将该字符从输入流中取走
//不会跳过输入流中的空格、回车符；在输入流已经结束的情况下，返回 EOF。
int_type peek();

//获取当前流中游标所在的位置
pos_type tellg();

//偏移游标的位置
basic_istream & seekg(pos_type pos);
basic_istream & seekg(off_type off, std::ios::seekdir dir);

//----以下为输出流操作----
//往输出流中写入一个字符
ostream & put(char_type ch);
//往输出流中写入count个字符
ostream & write(const char_type * s, std::streamsize count);

//获取当前流中游标所在的位置
pos_type tellp();

//刷新缓冲区
ostream & flush();

//偏移游标的位置
ostream & seekp(pos_type pos);
ostream & seekp(off_type off, std::ios_base::seekdir dir);
```

2.7、缓冲区

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**。

2.7.1、为什么要引入缓冲区呢?

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。

2.7.2、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。

2.7.3、缓冲区的类型

缓冲区分为三种类型：全缓冲、行缓冲和不带缓冲。

- 全缓冲：在这种情况下，当填满标准I/O缓存后才进行实际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。典型代表是键盘输入数据。
- 不带缓冲：也就是不进行缓冲，标准出错情况cerr/stderr是典型代表，这使得出错信息可以直接尽快地显示出来。

2.7.4、C++中的流缓冲区

在C++中，流的缓冲区之基类是定义在 头文件当中的 streambuf.

在头文件 当中，定义着两个类: ios_base和 ios。ios_base是所有I/O类的祖先，提供了**状态信息**、**控制信息**、**内部存储**、**回调**等设施。ios继承自ios_base，额外提供了与streambuf的接口；同时允许多个ios对象绑定同一个streambuf对象。

由于ios_base没有提供与streambuf的接口，ios才是标准库内所有I/O类（模板）事实上的最近共同祖先。ios的成员函数rdbuf是读取和设置流对象（ios的对象）绑定缓冲区的成员函数，它有两个不同的重载形式，分别如下：

```
//返回与之关联的streambuf；如果没有，则返回nullptr
streambuf *rdbuf() const;

//重新设置streambuf
//如果有，与先前绑定的streambuf解绑，再绑定传入的streambuf；
//如果传入的是nullptr，则流对象不与任何缓冲区对象绑定
streambuf *rdbuf(streambuf * sb);
```

streambuf本身不可以直接创建对象，它是一个抽象类型；但它向下派生出了2个派生类型，如我们在类图中看到的filebuf和stringbuf，这两个类分别是作为文件流和字符串流的子对象的，可以直接创建对象，后面我们会再次看到。

3、C++标准IO

3.1、标准输入流

istream类定义了1个输入流对象，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>>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。

下面来看一个例子，每次从cin中获取一个字符:

```
void test()
{
    char ch;
    while((ch = cin.get()) != '\n')
    {
        cout << ch;
    }
    cout << ch;
}
```

在界面输入"123456abcd回车"，当敲下回车（'\n'）时，数据就会被存入输入缓冲区中，字符'\n'也一起存入了缓冲区，然后每次get()操作就会从缓冲区中获取一个字符。

下面这个例子，通过cin获取一个单词：

```
void test1()
{
    int value;
    cin >> value;
    cout << "value:" << value << endl;

    string word;
    cin >> word;
    cout << "line:" << line << endl;
}
```

首先程序输入123abc，看看程序会发生什么?

接下来，我们关注一下**流的状态**:

```
void printStreamStatus(istream & is)
{
    cout << "is.good() = " << is.good() << endl;
    cout << "is.bad() = " << is.bad() << endl;
    cout << "is.fail() = " << is.fail() << endl;
    cout << "is.eof() = " << is.eof() << endl;
}

void test2()
{
    int value;
    printStreamStatus(cin);
    while(cin >> value)
    {
        cout << value << endl;
    }
    printStreamStatus(cin);
```

```
    std::cin.clear();
    printStreamStatus(cin);
    cin.ignore(1024, '\n');

    std::string s;
    std::cin >> s;
    std::cout << s << std::endl;
}

void test3()
{
    int value;
    while(cin >> value, !cin.eof())
    {
        if(cin.bad())
        {
            throw runtime_error("IO stream is corrupted!");
        }
        else if(cin.fail())
        {
            cerr << "bad data, pls input a valid integer number!" << endl;
            cin.clear();
            cin.ignore(1024, '\n');
            continue;
        }
        else
        {
             cout << "value = " << value << endl;
        }
    }
}
```

下面这个例子，从cin中获取一行数据:

```
void test4()
{
    char buffer[1024] = {0};
    cout << "pls input a line string:" << endl;
    cin.getline(buffer, 1024);
    cout << "the line is:" << buffer << endl;
}
```

3.2、标准输出流

ostream类定义了3个全局输出流对象，即cout,cerr,clog，平常用的最多的就是cout,即**标准输出。**cout将数据输出到终端，它与标准C输出stdout关联。cerr是标准错误流（非缓冲），clog也是标准错误流（带缓冲）。注意：在C语言中，标准输入、标准输出和标准错误分别用0, 1, 2文件描述符代表。

下面我们来看一个例子:

```
void test5()
{
    cout.put('a');
    cout.put('\n');

    char str[] = "hello,world";
    cout.write(str, sizeof(str));

    int x = 0x61626364;
    cout.write((char*)&x, sizeof(x)) << endl;
}
```

3.2.1、cout与cerr/clog的区别

我们来看一个例子:

```
void test6()
{
    cout << "hello, cout" << endl;
    cerr << "hello, cerr" << endl;
    clog << "hello, clog" << endl;
    sleep(2);
}
```

假设我们经过编译后得到的可执行程序为test，然后我们在命令行输入:

```
$ ./test > a.txt
```

执行完毕后，我们会发现屏幕上输出的是:

```
hello, cerr
hello, clog
```

但hello, cout没有输出到屏幕上，但却出现在了文件a.txt之中。

若执行的是:

```
$ ./test 2>a.txt
```

我们会发现屏幕上输出的是:

```
hello, cout
```

而另外2句被写入了文件a.txt。这就是区别啦。

3.2.2、cerr与clog的区别

它们俩都是标准错误流，区别在于cerr不经过缓冲区，直接向终端输出信息，而clog中的信息是存放在缓冲区的，缓冲区满后或遇到endl向终端输出。

3.3、输出缓冲区

输出缓冲区内容刷新的意思是：输出缓冲区的内容写入到真实的输出设备或者文件。

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（有一个收尾操作就是清空缓冲区）；
2. 缓冲区满（包含正常情况和异常情况）；
3. 使用操纵符显式地刷新输出缓冲区，如：endl、flush、ends(没有刷新功能)；
4. 使用unitbuf操纵符设置流的内部状态；
5. 输出流与输入流相关联，此时在读输入流时将刷新其关联的输出流的输出缓冲区。

3.3.1、演示缓冲区满

```
void test7()
{
    //在Ubuntu18.04上演示
    for(size_t idx = 0; idx < 1024; ++idx)
    {
        cout << 'a';
    }

    sleep(5);
    cout << 'b';
}
```

3.3.2、使用操纵符

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，但是没有刷新缓冲区（这个需要注意，很多书上说可以刷新缓冲区）

flush: 用来直接刷新缓冲区的

unitbuf: 在每次执行完写操作后都刷新输出缓冲区

nounitbuf: 让流回到正常的缓冲方式

```
void test8()
{
    cout << "hello, world!" << endl;//立刻换行输出
    cout << "hello, 德玛西亚Garen";//不确定啥时候会输出
    sleep(5);

    cout << "hello, 寒冰射手Ashe" << ends;
    cout << "hello, 无极剑圣Master Yi" << flush;
    cout << unitbuf << "hello, 齐天大圣Wukong" << nounitbuf;
}
```

3.3.3、输出流与输入流相关联

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，可以测试:

```
void test9()
{
    auto stream = cin.tie();
    cout << "stream:" << stream << endl;
    cout << "&cout:" << &cout << endl;

    cin.tie(nullptr);//解除关联
}
```

交互式系统通常应该关联输入流和输出流。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。

用来关联流的操作是tie:

```
ostream *tie () const;  //返回指向绑定的输出流的指针。
ostream *tie (ostream *os);  //将os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。
```

3.4、C++文件IO

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。根据文件中数据的组织形式，可分为**ASCII文件**和**二进制文件**。

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 = in);
explicit ifstream(const string &filename, openmode mode = in);

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

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

3.4.1、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，读操作始终发生在文件的末尾
- 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

源文件中的形式:

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     };
429     typedef _Ios_Openmode openmode;
430
431     /// Seek to end before each write.
432     static const openmode app =     _S_app;
433
434     /// Open and seek to end immediately after opening.
435     static const openmode ate =     _S_ate;
436
437     /// Perform input and output in binary mode (as opposed to text mo
438     /// This is probably not what you think it is; see
439     /// https://gcc.gnu.org/onlinedocs/libstdc++/manual/fstreams.html#
440     static const openmode binary =  _S_bin;
441
442     /// Open for input.  Default for @c ifstream and fstream.
443     static const openmode in =      _S_in;
444
445     /// Open for output.  Default for @c ofstream and fstream.
446     static const openmode out =     _S_out;
447
448     /// Open for input.  Default for @c ofstream.
449     static const openmode trunc =   _S_trunc;
```

简单读文件可以用ifstream的相关方法>>，简单写文件可以用ofstream的相关方法<<.

每次读取文件中的一行数据:

```
void test()
{
    ifstream ifs("test.txt");
    if(!ifs.good())
    {
        cerr << ">> ifstream open file error!\n";
        return;
    }

    string line;
    vector<string> vec;
    while(getline(ifs, line)
    {
        //cout << line << endl;
        vec.push_back(line);
    }
    ifs.close();

    ofstream ofs("a.txt");
    if(!ofs.good())
    {
        cerr << ">> ofstream open file error!\n";
        return;
    }

    for(auto &elem : vec)
    {
        ofs << elem << '\n';
    }

    ofs.close();
}
```

每次在文件的末尾添加数据：

```
int test(void)
{
    std::ofstream ofs("text1.txt", std::ios::app);
    if(!ofs)
    {
        cerr << "ofstream error!" << endl;
        return -1;
    }
    cout << ofs.tellp() << endl;
    ofs << "that's new line" << std::endl;
    ofs.close();

    return 0;
}
```

随机存取：利用对文件指针的操作可以实现随机存取数据。对于文件指针的偏移操作涉及到三个值:

```
ios_base::beg    文件开头
ios_base::cur    当前指针位置
ios_base::end    文件结尾位置
```

对于短文件，如果想要一次性拿到所有的数据，该怎么操作?

```
void test12()
{
    ifstream ifs("a.txt");
    if(!ifs.is_open())
    {
        cerr << "ifstream open file error!\n";
        return;
    }

    ifs.seekg(std::ios_base::end);

    auto length = ifs.tellg();
    ifs.seekg(std::ios_base::beg);
    char *buff = new char[length + 1]();
    ifs.read(buff, length + 1);
    string content(buff, length + 1);
    cout << content << endl;

    delete [] buff;
}
```

复制文件:

```
void test13()
{
    fstream in("a.txt", std::ios::in|std::ios::binary);
    if(!in.is_open())
    {
        cerr << "fstream open file error!\n";
        return;
    }

    fstream out("a.txt", std::ios::out|std::ios::binary);
    if(!out.is_open())
    {
        cerr << "fstream open file error!\n";
        return;
    }

    out << in.rdbuf();//流的重定向
    out.close();
    in.close();
}
```

复制文件应该用ios::binary(二进制模式)，原因是使用二进制文件模式时，程序将数据从内存传递给文件，将不会发生任何隐藏的转换，而默认状态下是文本模式，复制的内容可能会发生改变。

3.5、C++字符串IO

字符串流是以内存中的字符串类对象或者字符数组为输入输出对象的数据流，也即是将数据输出到字符串流对象或者从字符串流对象读入数据，也称之为内存流。

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），他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I/O经常用来做格式转换，下面来看一看例子:

```
void testStringStream()
{
    string s1 = "hello, world. This is C++ world";
    istringstream iss(s1);
    string word;
    while(iss >> word)
    {
        cout << word << endl;
    }
}

string int2str(int number)
{
    ostringstream oss;
    oss << number;
    return oss.str();
}

void testStringCat()
{
    stringstream ss;
    int idx;
    string prefix = "~/documents/photos/";
    string postfix = ".jpg";
```

```
    string filename;
    for(idx = 0; idx < 100; ++idx)
    {
        ss << prefix << idx << postfix << '\n';
        ss >> filename;
        ss.clear();
        ss.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        cout << filename << endl;
    }
}
```