## 前言

本人笔记都是以Q&A的形式记录，旨在：

* 引导思考
* 真实记录思考过程

## stream

### cin/cout

#### <</>>返回值

* 下面这段程序，为什么不仅Here is text followed by a number:可以被输出，31415也可以？

cout << "Here is text followed by a number: " << 31415 << endl;

在C++中，<< 流插入运算符的返回值确实是它的左操作数，也就是说，它返回了一个对应的输出流对象。这允许我们链式地使用多个 << 运算符。

对于一个运算符来说，应当关注下面三个方面的内容：

* 优先级
* 结合性
* 返回值

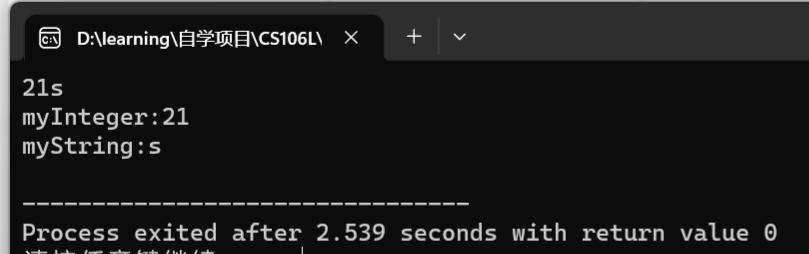
#### 错误处理

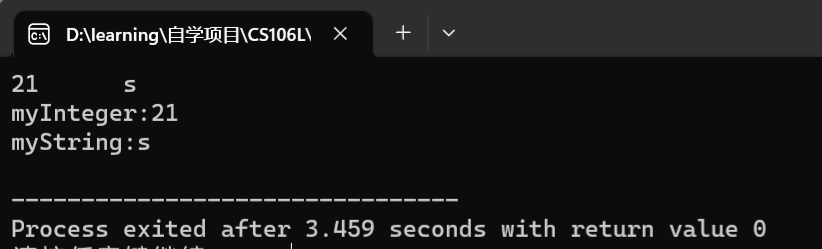
* 下面这段程序：当我输入S后，输出myInteger为0，这是为什么?
* #include <iostream>  
  using namespace std;  
  int main() {  
   cout << "Please enter an integer: ";  
   int myInteger=90;  
   cin >> myInteger; // Value stored in myInteger  
   cout<<"myInteger:"<<myInteger<<endl;  
   return 0;  
  }
* (ChatGPT)当你尝试使用 cin >> myInteger; 从标准输入读取一个整数，但实际上输入了一个非整数字符（例如"S"），cin 会进入一个"失败"状态。在这种失败状态下，它不会修改 myInteger 的值。但由于你已经初始化 myInteger 为90，所以你可能期望看到90作为输出。
* 然而，这里的关键是当 cin 进入失败状态时，它实际上会设置一些内部标志，这些标志表示输入流已经遇到了错误。在这种情况下，任何后续的输入操作都不会成功，除非你首先清除这些错误标志。
* 在你的例子中，当你输入"S"时，cin 进入失败状态，并设置了其内部的错误标志。由于你没有清除这些标志，所以 myInteger 的值不会被修改，它仍然保持其初始值90。
* 但是，你提到当你输入"S"后，输出的 myInteger 为0。这是因为当输入流遇到一个无效的输入（如一个字符而不是一个整数）时，它不会改变变量的值，但在某些情况下，它可能会设置变量为0。这取决于编译器和库的实现。
* 为了处理这种情况，你可以检查 cin 的状态，并在必要时清除错误标志：
* if (cin.fail()) {  
   cin.clear(); // clear the error flags  
   cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // discard invalid input  
   cout << "Invalid input. Please enter an integer." << endl;  
  }
* 这样，你就可以捕获并处理无效的输入，确保程序的稳健性

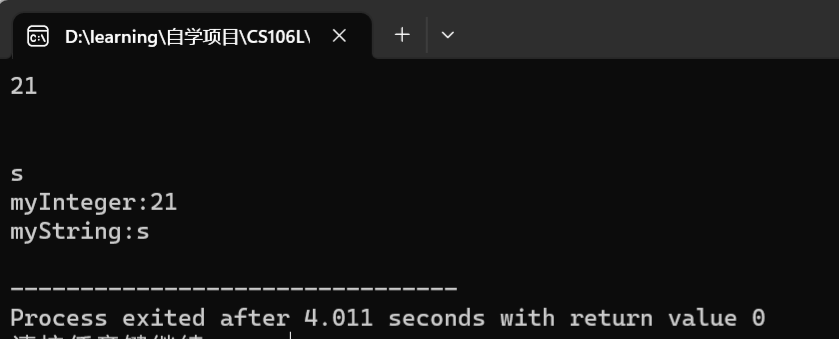
#### cin流提取的终止条件

* 观察下面程序及其运行结果，可以得出几个重要的cin流提取的终止条件

#include <iostream>  
using namespace std;  
int main() {  
 int myInteger;  
 string myString;  
 cin >> myInteger >> myString; // Read an integer and string from cin  
 cout<<"myInteger:"<<myInteger<<endl;  
 cout<<"myString:"<<myString<<endl;  
 return 0;  
}







In practice, it is not a good idea to read values directly from cin. Unlike GetInteger and the like, cin  
does not perform any safety checking of user input and if the user does not enter valid data, cin will begin  
behaving unusually.

#### cerr vs cout

* 错误处理优先使用cerr，尤其是类似于无限，没有出口的递归这种case

std::cerr是标准错误输出流，不需要缓存，直接显示在显示器上，而且只能显示到显示器上，即std::cerr流不能转向。  
std::cout需要缓冲区（目的是为了减少屏幕刷新次数，一次全部显示），std::cout流可以转向，例如可将cout流流向文件操作中去。  
1.区别  
cout对应于标准输出流，默认情况下是显示器。这是一个被缓冲的输出，可以被重定向。  
cerr对应标准错误流，用于显示错误消息。默认情况下被关联到标准输出流，但它不被缓冲，也就说错误消息可以直接发送到显示器，而无需等到缓冲区或者新的换行符时，才被显示。一般情况下不被重定向。  
2.为什么要用cerr  
比如，你的程序遇到调用栈用完了的威胁（无限，没有出口的递归）。  
你说，你到什么地方借内存，存放你的错误信息？  
所以有了cerr。其目的，就是在你最需要它的紧急情况下，还能得到输出功能的支持。 缓冲区的目的，就是减少刷屏的次数——比如，你的程序输出圣经中的一篇文章。不带缓冲的话，就会每写一个字母，就输出一个字母，然后刷屏。有了缓冲，你将看到若干句子“同时”就出现在了屏幕上（由内存翻新到显存，然后刷新屏幕）  
————————————————  
版权声明：本文为CSDN博主「「已注销」」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：<https://blog.csdn.net/hyqsong/article/details/38453511>

### fstream

#### 对比

Unlike cin and cout, which are concrete stream objects, ifstream and ofstream are types.To read or write  
from a file, you will create an object of type ifstream or ofstream, much in the same way that you would  
create an object of type string to store text data or a variable of type double to hold a real number

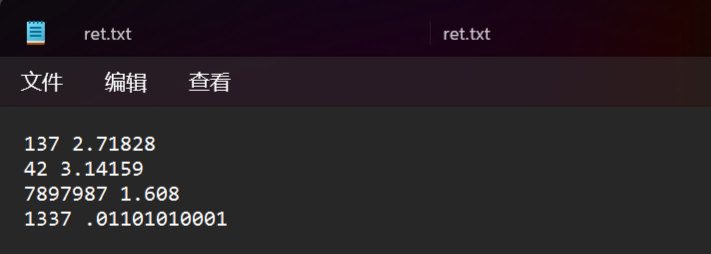
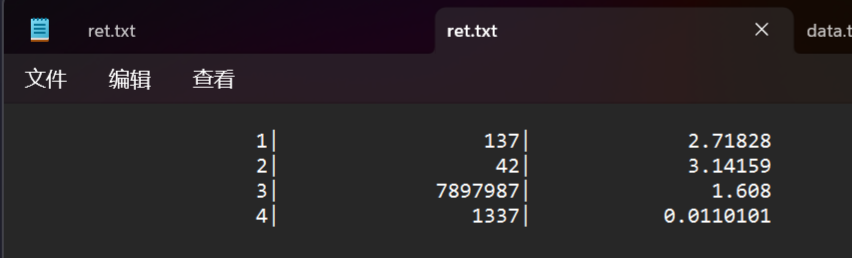
#### 记忆

要知道，程序也是以自我为中心的，所以ifstream的“in”是相对于程序而言的，因此ifstream类型的对象就是从别的文件读取内容到程序中。ofstream同理

#### good practice

ifstream input("myfile.txt");  
if(!input.is\_open())  
cerr << "Couldn't open the file myfile.txt" << endl;

#### Stream Manipulators

* 常用eg
  + setw
  + setfiill
  + hex, dec, oct
* 这个还是挺有用的，可以用来帮处理一些杂事，尤其是写文档时候的格式问题，比如写了给目录加上.........，同时保持每一行的长度一样等等类似的重复性工作
* eg：
* 原来：
* 
* 处理后：
* 

#### good practice

替换前：

* 这里也可以学习一下fail的使用

while(true) {  
 int intValue;  
 double doubleValue;  
 input >> intValue >> doubleValue;  
 if(input.fail()) break;  
 /\* ... process values here ... \*/  
}

替换后：

int intValue;  
double doubleValue;  
while(input >> intValue >> doubleValue) {  
 /\* ... process values here ... \*/  
}

注意区分，运算符的返回值和整个表达式的返回值不一样

* >> 操作符的返回值是用于输入的流对象（在这个情况下是 cin），它用于链式输入多个值。这是为了支持链式输入，例如 cin >> a >> b;，其中多个输入操作可以连续执行。
* 整个表达式 cin >> a 的返回值是一个布尔值，表示输入操作的成功或失败。如果输入成功，它返回 true，否则返回 false。这个返回值通常用于条件检查，以确定输入是否成功。

#### When Streams Do Too Much

举个栗子：

int age;  
double hourlyWage;  
cout << "Please enter your age: ";  
cin >> age;  
cout << "Please enter your hourly wage: ";  
cin >> hourlyWage;

输入2.1234+回车，会发生什么？

* reading directly from cin is unsafe and poses more problems than it solves

#### >> vs getline

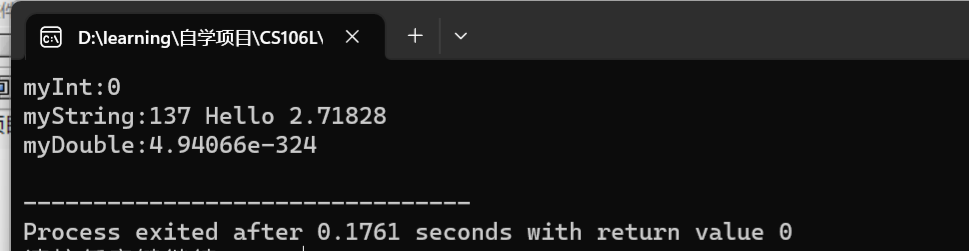
* 核心区别在于终止条件不同。
* 比如，cin>>a;的终止条件是空格或其他分隔符（例如换行符），而getline(cin,a)的终止条件是遇到换行
* getline是更契合实际读取文件的一种方式，原因在于大部分case下我们是不希望把whitespace作为终止条件之一的。可以看看cs106L教材的例子

#### sstream

* 全称：string streams，可以用来模仿java那种字符串拼接
* 头文件:
* stringstreams are an example of an iostream, a stream that can perform both input and output

int main()  
{  
 stringstream myConverter;  
 int myInt;  
 string myString;  
 double myDouble;  
 myConverter << "137 Hello 2.71828"; // Insert string data  
 //myConverter >> myInt >> myString >> myDouble; // Extract mixed data--------------------- line 1  
 getline(myConverter,myString);//---------------------------------------------------------- line 2  
 cout<<"myInt:"<<myInt<<endl;  
 cout<<"myString:"<<myString<<endl;  
   
 cout<<"myDouble:"<<myDouble<<endl;  
  
}

输出如下：



对比line1和line2作用下的输出结果，可以进一步加深上一小节的知识

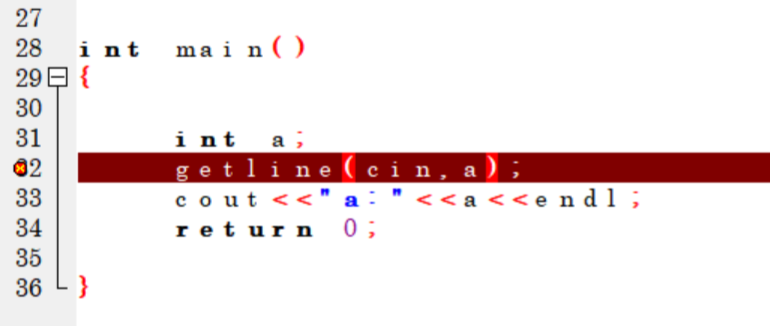
#### formatting

* 解释原因：内存中，int型数据78979871通过iostream的方式存储到一个txt文件中，其实际占用的内存由4bytes变为8bytes。

read and write: When you write numeric data to a stream, you're actually converting them into  
sequences of characters that represent those numbers. For example, when you print out the fourbyte  
value 78979871, you're using eight bytes to represent the data on screen or in a file – one for  
each character. These extra bytes can quickly add up, and it's actually possible to have on-disk  
representations of data that are more than twice as large as the data stored in memory. To get  
around this, C++ streams let you directly write data from memory onto disk without any  
formatting. All ostreams support a write function that writes unformatted data to a stream, and  
istreams support read to read unformatted data from a stream into memory. When used well,  
these functions can cut file loading times and reduce disk space usage. For example, The  
CS106B/X Lexicon class uses read to quickly load its data file into memory

#### 综合案例------GetInteger

核心在于解决cin和getline的缺点

* cin:可能“do too much"
* cin:可能fail
* getline可以避免上述问题
* 但是，getline只能读string：
* 
* 最终得出GetInteger的实现，（注意，这里没有给出GetLine()函数的实现
* #include <iostream>  
  using namespace std;  
  #include <iomanip>  
  #include <sstream>  
    
  int GetInteger() {  
   while (true) { // Read input until user enters valid data  
   stringstream converter;  
   converter << GetLine();  
   /\* Try reading an int, continue if we succeeded. \*/  
   int result;  
   if (converter >> result) {  
   char remaining;  
   if (converter >> remaining) // Something's left, input is invalid  
   cout << "Unexpected character: " << remaining << endl;  
   else  
   return result;  
   }  
   else  
   cout << "Please enter an integer." << endl;  
   cout << "Retry: ";  
   }  
  }

## Multi-File Programs, Abstraction, and the Preprocessor

### 链接

* 下面的程序会在什么阶段报错：为什么？

#include <iostream>  
using namespace std;  
int Factorial(int n); // Prototype for a function to compute n!  
int main() {  
 cout << Factorial(10) << endl;  
 return 0;  
}

### preprocess

#### #include

* 类似于#include，#define的directive都不属于c++语法，所以下面都是错误的
* #include <iostream>//前面不能有空格
    
   cout << #include <iostream> << endl; // Error: #include must start a line.
    
   #include <iostream>;
    
   #include
    
  <iostream> // Error: Multi-line preprocessor directives are illegal
* 应该把他们看作一种独立于cpp的语言

#if MY\_CONSTANT > 137 // Legal
  
#if MY\_CONSTANT \* 42 == MY\_CONSTANT // Legal
  
#if sqrt(MY\_CONSTANT) < 4 // Illegal, cannot call function sqrt
  
#if MY\_CONSTANT == 3.14 // Illegal, can only use integral values

#### #define

* 首先要理解，在预编译阶段的程序都是且仅是文本
* 在此基础上，由#define带来的种种tricks也就不难理解了

#### #Include Guards Explained && #if

* 为什么存在Include Guards Explained？
* #if：
  + 格式
  + #if statement
      
    ...
      
    #elif another-statement
      
    ...
      
    #elif yet-another-statement
      
    ...
      
    #else
      
    ...
      
    #endif
* #if MY\_CONSTANT > 137 // Legal
    
  #if MY\_CONSTANT \* 42 == MY\_CONSTANT // Legal
    
  #if sqrt(MY\_CONSTANT) < 4 // Illegal, cannot call function sqrt
    
  #if MY\_CONSTANT == 3.14 // Illegal, can only use integral values
  + 可以和defined结合使用
  + eg:
  + #if defined(A)
      
    cout << "A is defined." << endl;
      
    #elif defined(B)
      
    cout << "B is defined." << endl;
      
    #elif defined(C)
      
    cout << "C is defined." << endl;
      
    #else
      
    cout << "None of A, B, or C is defined." << endl;
      
    #endif

#### 综合app

* 确保一个头文件即使被include多次，保证其中的变量声明且仅声明一次：

#ifndef MyStruct\_Included
  
#define MyStruct\_Included
  
struct MyStruct {
  
 int x;
  
 double y;
  
 char z;
  
};
  
#endif

* 注意，这里是须要#endif的，其实也是理所当然的

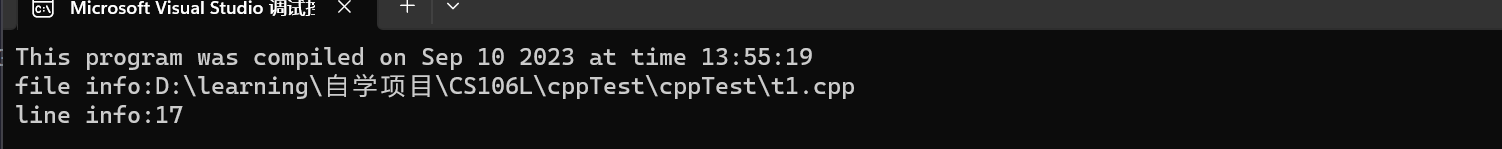
#### Macros

* 标准用法：
* #define macroname(parameter1, parameter2, ... , parameterN) macro-body\*
* 准则：凡是变量能用括号括起来就括，防止运算优先级等问题产生

#### preprocessor预置变量

* eg

void test0() {
  
 cout << "This program was compiled on " << \_\_DATE\_\_ << " at time " << \_\_TIME\_\_ << endl;
  
 cout << "file info:" << \_\_FILE\_\_ << endl;
  
 cout << "line info:" << \_\_LINE\_\_ << endl;
  
}



#### #define vs 函数

* 和preprocessor directives结合，可以做到一些c++函数无法实现的功能：

#define DECLARE\_MY\_VAR(type) type my\_##type
  
DECLARE\_MY\_VAR(int);

## STL

### Sequence Containers

#### STL总览

The STL is logically divided into six pieces, each consisting of generic components that interoperate with  
the rest of the library:  
• Containers. At the heart of the STL are a collection of container classes, standard C++'s analog to  
the CS106B/X ADTs. For example, you can store an associative collection of key/value pairs in an  
STL map, or a growing list of elements in an STL vector.  
• Iterators. Each STL container exports iterators, objects that view and modify ranges of stored  
data. Iterators have a common interface, allowing you to write code that operates on data stored  
in arbitrary containers.  
• Algorithms. STL algorithms are functions that operate over ranges of data specified by iterators.  
The scope of the STL algorithms is staggering – there are algorithms for searching, sorting,  
reordering, permuting, creating, and destroying sets of data.

• Adapters. STL adapters are objects which transform an object from one form into another. For  
example, the stack adapter transforms a regular vector or list into a LIFO container, while the  
istream\_iterator transforms a standard C++ stream into an STL iterator.  
• Functors. Because so much of the STL relies on user-defined callback functions, the STL provides  
facilities for creating and modifying functions at runtime. We will defer our discussion of functors  
to much later in this text, as they require a fairly nuanced understanding of C++.  
• Allocators. The STL allows clients of the container classes to customize how memory is allocated  
and deallocated, either for diagnostic or performance reasons. While allocators are fascinating  
and certainly worthy of discussion, they are beyond the scope of this text and we will not cover  
them here.

