

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

git push前：

```
1 git config --global http.proxy
2 git config --global https.proxy
3 git config --global --unset http.proxy
4 git config --global --unset https.proxy
5
6 (windows需要管理员权限)
7
8 如果上面的方法不行，可以试下面的方法（同样需要管理员权限）
9 git config --global http.sslVerify false
10 git config --global https.sslVerify false
```

```
1 git push github_remote thread:thread --force
```

token:

```
1 ghp_McG8oqfKZok41nJUEMaqHkFgRCM0bp0y67Xr
```

前后lab之间的关系

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？

```
1 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？

```
1  #include <iostream>
2  using namespace std;
3  int main() {
4      cout << "Please enter an integer: ";
5      int myInteger=90;
6      cin >> myInteger; // value stored in myInteger
7      cout<<"myInteger:"<<myInteger<<endl;
8      return 0;
9  }
```

(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：

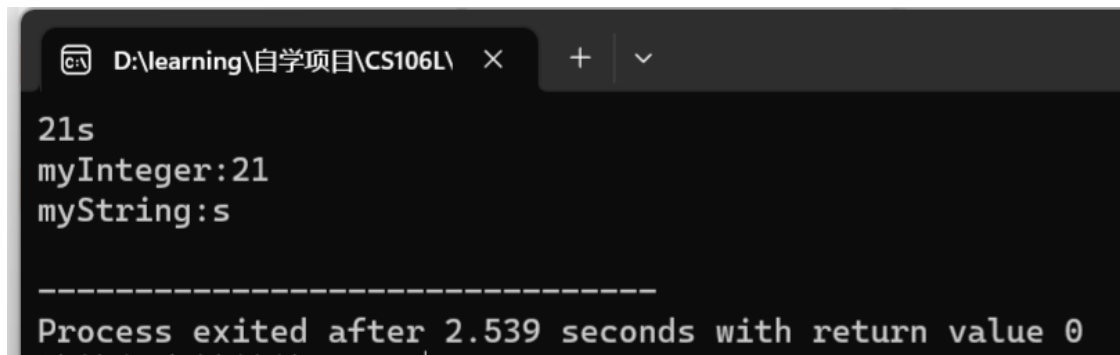
```
1  if (cin.fail()) {
2      cin.clear(); // clear the error flags
3      cin.ignore(numeric_limits<streamsize>::max(), '\n'); //
discard invalid input
4      cout << "Invalid input. Please enter an integer." <<
endl;
5  }
```

这样，你就可以捕获并处理无效的输入，确保程序的稳健性

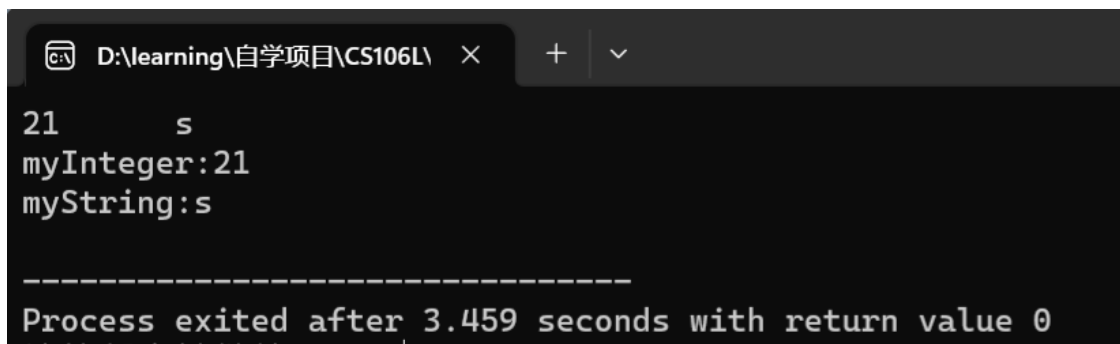
cin流提取的终止条件

- 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

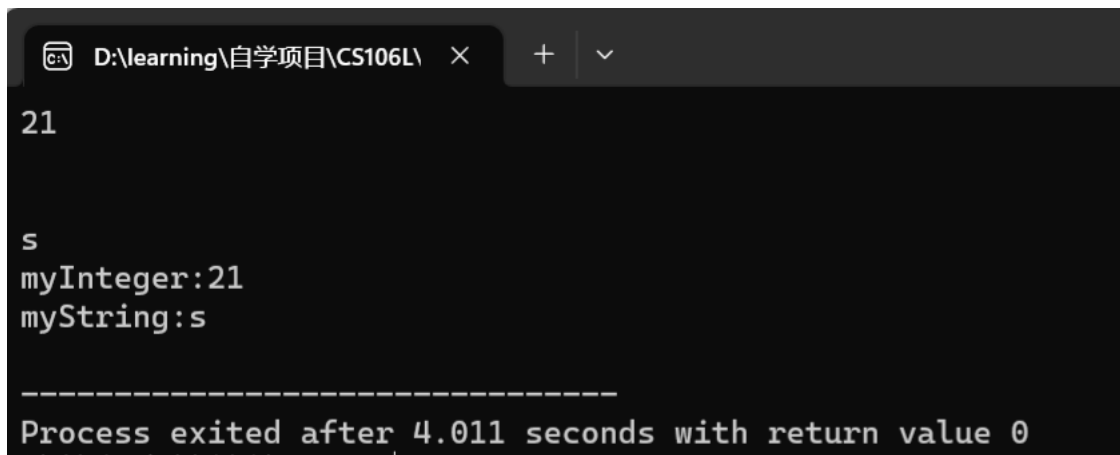
```
1  #include <iostream>
2  using namespace std;
3  int main() {
4      int myInteger;
5      string myString;
6      cin >> myInteger >> myString; // Read an integer and string
    from cin
7      cout<<"myInteger:"<<myInteger<<endl;
8      cout<<"myString:"<<myString<<endl;
9      return 0;
10 }
```



```
D:\learning\自学项目\CS106L>
21s
myInteger:21
myString:s
-----
Process exited after 2.539 seconds with return value 0
```



```
D:\learning\自学项目\CS106L>
21      s
myInteger:21
myString:s
-----
Process exited after 3.459 seconds with return value 0
```



```
D:\learning\自学项目\CS106L>
21
s
myInteger:21
myString:s
-----
Process exited after 4.011 seconds with return value 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。

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，也就是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

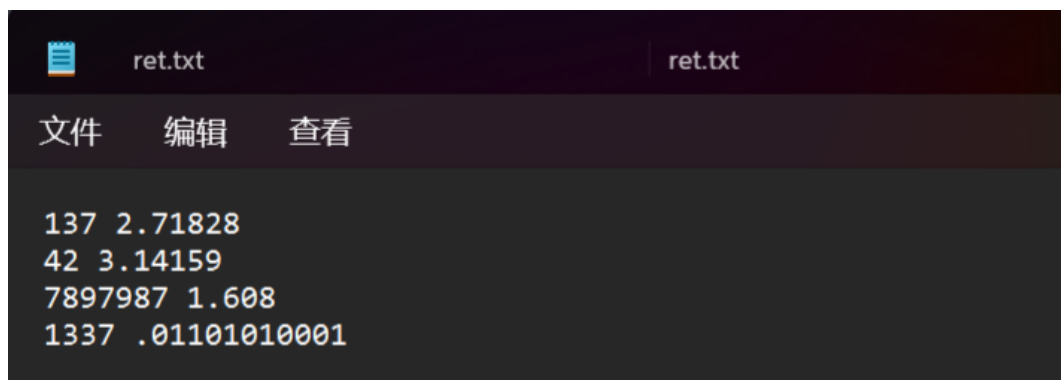
```
1 ifstream input("myfile.txt");
2 if(!input.is_open())
3     cerr << "Couldn't open the file myfile.txt" << endl;
```

Stream Manipulators

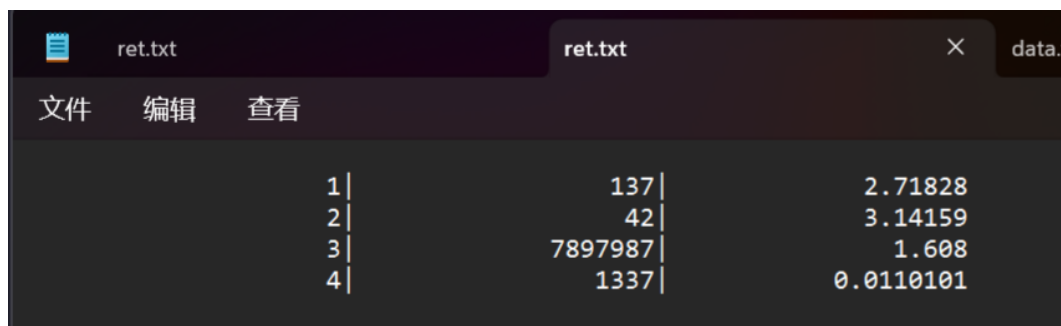
- 常用eg
 - setw
 - setfill
 - hex, dec, oct
- 这个还是挺有用的，可以用来帮处理一些杂事，尤其是写文档时候的格式问题，比如写了给目录加上.....，同时保持每一行的长度一样等等类似的重复性工作

eg:

原来:



处理后:



good practice

替换前:

- 这里也可以学习一下fail的使用

```

1 while(true) {
2     int intValue;
3     double doubleValue;
4     input >> intValue >> doubleValue;
5     if(input.fail()) break;
6     /* ... process values here ... */
7 }

```

替换后:

```

1 int intValue;
2 double doubleValue;
3 while(input >> intValue >> doubleValue) {
4     /* ... process values here ... */
5 }

```

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:

```

1 int age;
2 double hourlyWage;
3 cout << "Please enter your age: ";
4 cin >> age;
5 cout << "Please enter your hourly wage: ";
6 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

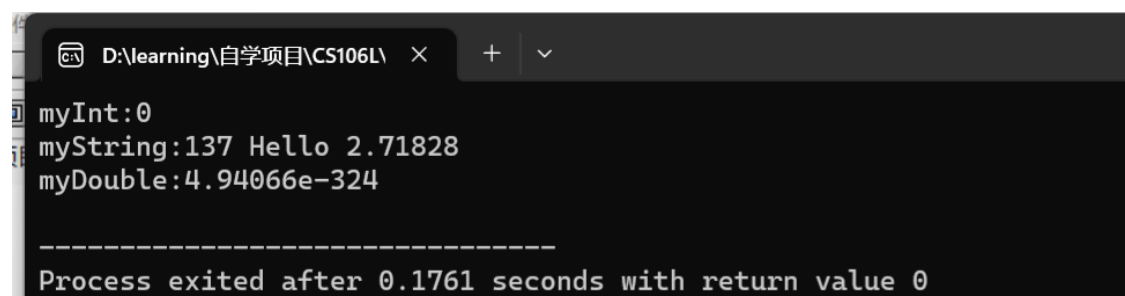
- getline是更契合实际读取文件的一种方式，原因在于大部分case下我们是不希望把 whitespace作为终止条件之一的。可以看看cs106L教材的例子

sstream

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

```
1  int main()
2  {
3      stringstream myConverter;
4      int myInt;
5      string myString;
6      double myDouble;
7      myConverter << "137 Hello 2.71828"; // Insert string data
8      //myConverter >> myInt >> myString >> myDouble; // Extract
mixed data----- line 1
9      getline(myConverter,myString);//-----
----- line 2
10     cout<<"myInt:"<<myInt<<endl;
11     cout<<"myString:"<<myString<<endl;
12
13     cout<<"myDouble:"<<myDouble<<endl;
14
15 }
```

输出如下:



```
myInt:0
myString:137 Hello 2.71828
myDouble:4.94066e-324
-----
Process exited after 0.1761 seconds with return value 0
```

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:

```
27
28  i n t   m a i n ( )
29  {
30
31      i n t   a ;
32      g e t l i n e ( c i n , a ) ;
33      c o u t << " a : " << a << e n d l ;
34      r e t u r n   0 ;
35
36  }
```

- 最终得出GetInteger的实现，（注意，这里没有给出GetLine()函数的实现

```
1
2  #include <iostream>
3  using namespace std;
4  #include <iomanip>
5  #include <sstream>
6
7  int GetInteger() {
8      while (true) { // Read input until user enters valid data
```



```

9      stringstream converter;
10     converter << GetLine();
11     /* Try reading an int, continue if we succeeded. */
12     int result;
13     if (converter >> result) {
14         char remaining;
15         if (converter >> remaining) // Something's left,
input is invalid
16             cout << "Unexpected character: " << remaining
<< endl;
17     }
18     else
19         return result;
20     }
21     else
22         cout << "Please enter an integer." << endl;
23     cout << "Retry: ";
24 }
25
26

```

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？

```

1  #include <iostream>
2  using namespace std;
3  int Factorial(int n); // Prototype for a function to compute n!
4  int main() {
5      cout << Factorial(10) << endl;
6      return 0;
7  }

```

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

```

1      #include <iostream> //前面不能有空格
2      cout << #include <iostream> << endl; // Error: #include must
      start a line.
3      #include <iostream>;
4      #include
5      <iostream> // Error: Multi-line preprocessor directives are
      illegal

```

- 应该把他们看作一种独立于cpp的语言

```

1      #if MY_CONSTANT > 137 // Legal
2      #if MY_CONSTANT * 42 == MY_CONSTANT // Legal
3      #if sqrt(MY_CONSTANT) < 4 // Illegal, cannot call function sqrt
4      #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

```

1      #if statement
2      ...
3      #elif another-statement
4      ...
5      #elif yet-another-statement
6      ...
7      #else
8      ...
9      #endif
10
11
12

```

```

1
2      #if MY_CONSTANT > 137 // Legal
3      #if MY_CONSTANT * 42 == MY_CONSTANT // Legal
4      #if sqrt(MY_CONSTANT) < 4 // Illegal, cannot call function
      sqrt
5      #if MY_CONSTANT == 3.14 // Illegal, can only use integral
      values

```

- 可以和defined结合使用

eg:

```
1  #if defined(A)
2  cout << "A is defined." << endl;
3  #elif defined(B)
4  cout << "B is defined." << endl;
5  #elif defined(C)
6  cout << "C is defined." << endl;
7  #else
8  cout << "None of A, B, or C is defined." << endl;
9  #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：

```
1  #ifndef MyStruct_Included
2  #define MyStruct_Included
3  struct MyStruct {
4      int x;
5      double y;
6      char z;
7  };
8  #endif
```

- 注意，这里是须要#endif的，其实也是理所当然的

Macros

- 标准用法：

```
1  #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

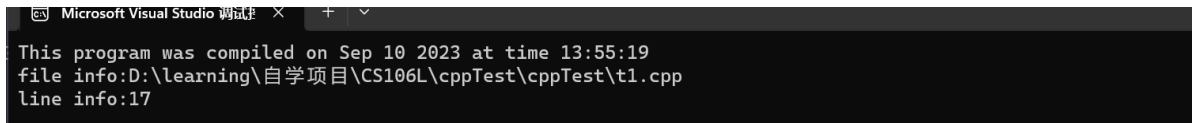
preprocessor预置变量

- eg

```

1 void test0() {
2     cout << "This program was compiled on " << __DATE__ << " at
    time " << __TIME__ << endl;
3     cout << "file info:" << __FILE__ << endl;
4     cout << "line info:" << __LINE__ << endl;
5 }

```



```

Microsoft Visual Studio 调试
This program was compiled on Sep 10 2023 at time 13:55:19
file info:D:\learning\自学项目\CS106L\cppTest\cppTest\t1.cpp
line info:17

```

#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：

```

1 #define DECLARE_MY_VAR(type) type my_##type
2 DECLARE_MY_VAR(int);

```

STL

Sequence Containers

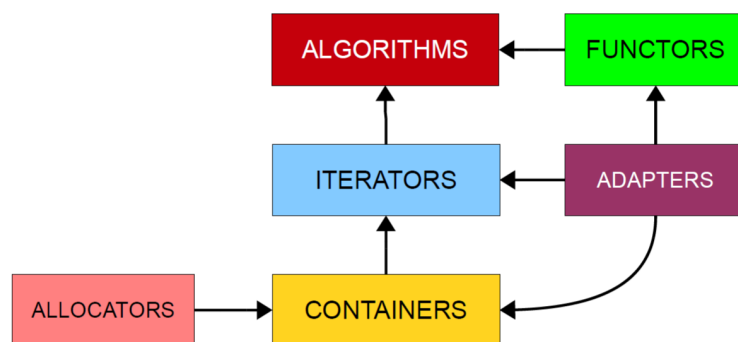
STL总览

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.



Associative Containers

- 着重了解这个“关联型容器”，之前了解不深
- set
 - insert
 - count(不是find)
 - erase

There is a tension here between what the code accomplishes and the way in which it accomplishes it. The reason for this is that we're using the **wrong abstraction**.

iterator

- background: 对于vector/deque，我们可以通过类似数组索引的方式来对其中的element进行traverse，但是对于set不可以

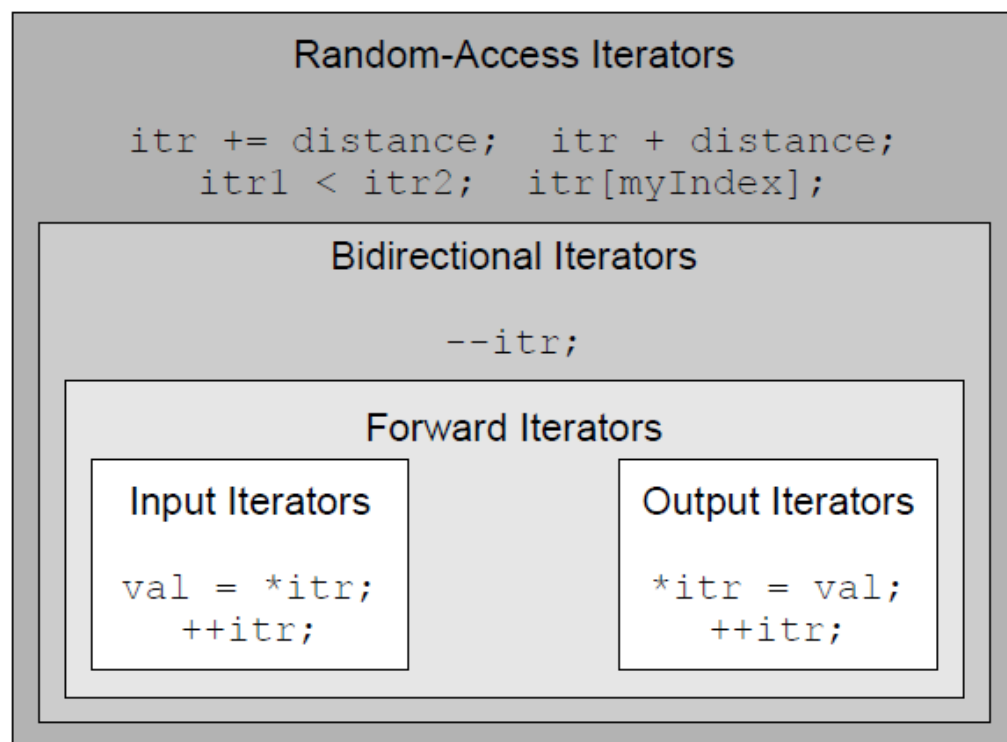
- 这就是Abstraction的魅力了：Iterators provide a clean, consistent mechanism for accessing data stored in containers, **irrespective of** how that data may be stored
- 请明确说出一个container的end()函数的返回值
- more: `std::set` 在 C++ 中是一个有序集合，它会**自动按照元素的值进行排序**。由于 `std::set` 是基于红黑树实现的，所以它的元素总是按照升序排序。
 - 基于其内部实现，我们可以得知，在不考虑运算符重载的前提下，set仅支持比较运算符生效的8中基本数据类型。
- ->相比于.的优势：

```

1  for (set<string>::iterator itr = mySet.begin(); itr !=
    mySet.end(); ++itr)
2      cout << *itr.length() << endl; // Error: Incorrect syntax!
3      cout << (*itr).length() << endl; // right
4
5  for (set<string>::iterator itr = mySet.begin(); itr !=
    mySet.end(); ++itr)
6      cout << itr->length() << endl;

```

- 不同类型迭代器



map

与pair的关系:

- 一个map对象可以视为许多个pair的集合
- 所以pair的api适用于map中的一个element

```
1 numberMap.insert(make_pair("zero", 0));
2 map<string, int>::iterator itr = myMap.begin(); itr !=
  myMap.end(); ++itr)
3     cout << itr->first << ": " << itr->second << endl
```

与vector的对比:

Indexing into a map into a nonexistent key implicitly creates a key/value pair.

- 解释, 下面程序的输出为什么是0?

```
1 int main() {
2
3     map<string, int> test;
4     cout << test["test"] << endl;
5     return 0;
6 }
```

基于此, 输出map[key]的good practice:

```
1 map<string, int>::iterator itr = numberMap.find("key");
2 if (itr == numberMap.end())
3     cout << "Key does not exist." << endl;
4 else
5     cout << numberMap["key"] << endl;
6     /* ... */
```

- map iterators are slightly more complicated because they dereference to a **key/value pair**----- `pair<const KeyType, valueType>`

insert

和使用中括号进行插入不同, insert不支持“更新: 以及存在的key对应的value

- 基于此, insert设计其返回值为: `pair<iterator, bool>`
- good practice:

```

1  /* Try to insert normally. */
2  pair<map<string, int>::iterator, bool> result =
    myMap.insert(make_pair("STL", 137));
3  /* If insertion failed, manually set the value. */
4  if(!result.second)
5      result.first->second = 137;

```

综合案例——关键字数量统计

代码实现

- 自顶向下开发，先确定好有哪些函数，在一步步实现这些函数
- 了解cctype

`<cctype>` 头文件包含了一系列的函数，用于字符的属性检测和转换。下面是这个头文件中的一些常用函数：

1. 属性检测:

- `isalpha()`: 检测字符是否为字母 (a-z, A-Z)。
- `isdigit()`: 检测字符是否为数字 (0-9)。
- `isalnum()`: 检测字符是否为字母或数字。
- `isspace()`: 检测字符是否为空白字符，例如空格、制表符、换行符等。
- `islower()`: 检测字符是否为小写字母。
- `isupper()`: 检测字符是否为大写字母。
- `iscntrl()`: 检测字符是否为控制字符。
- `isgraph()`: 检测字符是否有图形表示（即字符是可打印的并且不是空格）。
- `isprint()`: 检测字符是否是可打印的，包括空格。
- `ispunct()`: 检测字符是否为标点符号。
- `isxdigit()`: 检测字符是否为十六进制数字。

2. 字符转换:

- `tolower()`: 返回字符的小写形式，如果字符已经是小写或者不是字母，则返回原字符。
- `toupper()`: 返回字符的大写形式，如果字符已经是大写或者不是字母，则返回原字符。

- set,map的合理使用
- sstream的妙用——一种简洁有效的逐个获取文本中所有words的方式

DFA

- 凡是能画出状态转移图的，都大概率可以使用这种方法解决，关键在于，状态转移图不一定好画？
- app：邮箱命名格式判断
 - [代码实现](#)
 - q其核心代码相当简洁，核心是画出状态转移图，据此进行transitions的初始化，
 - 核心代码：

```
1 struct DFA {
2     map<pair<int, char>, int> transitions;
3     set<int> acceptingStates;
4     int startState;
5 };
6
7 bool simulateDFA(DFA& d, string input) {
8     int currState = d.startState;
9     for (string::iterator itr = input.begin(); itr !=
10 input.end(); ++itr) {
11         cout << "当前状态转移: " << currState << "-->"
12 << d.transitions[make_pair(currState, *itr)] << endl;;
13         currState = d.transitions[make_pair(currState,
14 *itr)];
15     }
16     //cout << "currState:" << currState << endl;
17     return d.acceptingStates.find(currState) !=
18 d.acceptingStates.end();
19 }
```

- 初始化：

```
1
2 //=====下面是针对邮箱地址设置的相关
3 内容，上面是通用设计=====
4 struct DFA LoadEmailDFA() {
5     struct DFA ret;
6     ret.startState = 0;
7     ret.acceptingStates.insert(6);
8
9     //这里反而是最麻烦的地方hhh，打错一个地方就g
```

```
10 ret.transitions.insert(make_pair(make_pair(0, 'a'), 1));
    //注意，这里外层还需要一次make_pair
11     ret.transitions.insert(make_pair(make_pair(0,
12     '.', 7));
13     ret.transitions.insert(make_pair(make_pair(0,
14     '@', 7));
15     ret.transitions.insert(make_pair(make_pair(1,
16     'a', 1));
17     ret.transitions.insert(make_pair(make_pair(1,
18     '@', 3));
19     ret.transitions.insert(make_pair(make_pair(1,
20     '.', 2));
21     ret.transitions.insert(make_pair(make_pair(2,
22     'a', 1));
23     ret.transitions.insert(make_pair(make_pair(2,
24     '@', 7));
25     ret.transitions.insert(make_pair(make_pair(2,
26     '.', 7));
27     ret.transitions.insert(make_pair(make_pair(3,
28     '@', 7));
29     ret.transitions.insert(make_pair(make_pair(3,
30     '.', 7));
31     ret.transitions.insert(make_pair(make_pair(3,
32     'a', 4));
33     ret.transitions.insert(make_pair(make_pair(4,
34     '@', 7));
35     ret.transitions.insert(make_pair(make_pair(4,
36     '.', 5));
37     ret.transitions.insert(make_pair(make_pair(4,
38     'a', 4));
39     ret.transitions.insert(make_pair(make_pair(5,
40     '@', 7));
41     ret.transitions.insert(make_pair(make_pair(5,
42     '.', 7));
43     ret.transitions.insert(make_pair(make_pair(5,
44     'a', 6));
45     ret.transitions.insert(make_pair(make_pair(6,
46     '.', 5));
47     ret.transitions.insert(make_pair(make_pair(6,
48     'a', 6));
49     ret.transitions.insert(make_pair(make_pair(6,
50     '@', 7));
```

```

37
38     ret.transitions.insert(make_pair(make_pair(7,
39 '@'), 7));
39     ret.transitions.insert(make_pair(make_pair(7,
40 '.'), 7));
40     ret.transitions.insert(make_pair(make_pair(7,
41 'a'), 7));
41
42     return ret;
43 }
44
45
46 bool IsEmailAddress(string input) {
47     DFA emailChecker = LoadEmailDFA();
48     for (string::iterator itr = input.begin(); itr !=
49 input.end(); ++itr) {
49         if (isalnum(*itr))
50             *itr = 'a';
51         else if (*itr != '.' && *itr != '@')
52             return false;
53     }
54     return SimulatedDFA(emailChecker, input);
55 }

```

NFA

- 对它的实际应用还比较迷茫

花式算法

suffix

- `_if`
 - 为什么调用 `IsEven` 时不用传递参数？解释

```

1 bool IsEven(int value) {
2     return value % 2 == 0;
3 }
4
5 cout << count_if(myVec.begin(), myVec.end(), IsEven) << endl;

```

- `_n`
 -

reorder

- 对于不支持<的数据类型，不进行运算符重载，同样可以调用sort:

```
1 bool ComparePlaces(placeT one, placeT two) {
2     if(one.x != two.x)
3         return one.x < two.x;
4     return one.y < two.y;
5 }
6 sort(myPlaceVector.begin(), myPlaceVector.end(),
    ComparePlaces);
```

- sort的第三个是函数参数，默认是使得结果升序，这里可以做文章

Searching Algorithms

Iterator Adaptors

Removal Algorithms

- 这就是我们热血沸腾的组合技 (by remove&erase)

```
1 myVector.erase(remove(myVector.begin(), myVector.end(), 137),
2 myVector.end());
3
4 string StripPunctuation(string input) {
5     input.erase(remove_if(input.begin(), input.end(),
6     ispunct), input.end());
7     return input;
8 }
```

- remove会改变原来的container，如果不想改变，可以使用
remove_copy/remove_copy_if

transform

- 啊？

There is no requirement that the function you pass to transform return elements of the same type as those stored in the container. It's legal to transform a set of strings into a set of doubles, for example.

min_element/max_element

As with other algorithms, by default the elements are compared by `<`, but you can provide a binary comparison function to the algorithms as a final parameter to change the default comparison order.

综合案例——回文字符串判定

```
1  IsPalindrome:
2  bool IsPalindrome(string input) {
3      input.erase(remove_if(input.begin(), input.end(), IsNotAlpha),
4                    input.end());
5      transform(input.begin(), input.end(), input.begin(),
6                ::toupper);
7      return equal(input.begin(), input.begin() + input.size() /
8                  2, input.rbegin());
9  }
```

more——回文文本判定:

```
1  bool IswordPalindrome(string input) {
2      input.erase(remove_if(input.begin(), input.end(),
3                            IsNotAlphaOrSpace), input.end()); //IsNotAlphaOrSpace可以使用
4      transform(input.begin(), input.end(), input.begin(),
5                ::toupper);
6      stringstream tokenizer(input);
7      vector<string> tokens;
8      tokens.insert(tokens.begin(), istream_iterator<string>
9                    (tokenizer), istream_iterator<string>());
10     return equal(tokens.begin(), tokens.begin() + tokens.size() /
11                 2, tokens.rbegin());
12 }
```

- 主要亮点在于: `tokens.insert(tokens.begin(), istream_iterator<string>(tokenizer), istream_iterator<string>());`

template

模板类接口函数实现&注意事项

- 经典错误1

```
1  template <typename FirstType, typename SecondType> class MyPair {
2  public:
```

```

3     FirstType getFirst();
4     void setFirst(FirstType newValue);
5     SecondType getSecond();
6     void setSecond(SecondType newValue);
7 private:
8     FirstType first;
9     SecondType second;
10 };
11
12 FirstType MyPair::getFirst() { // Problem: Not legal syntax
13     return first;
14 }

```

- 经典错误2--想起来了要加上template，但是把函数的prototype和implement放在了不同的文件中

我认为本质原因在于，不同的源文件之间的template <typename FirstType, typename SecondType>是独立的，也就是说，a.h文件中的FirstType为int，b.h中的FirstType仍然是未知的。template的**作用域是单个源文件**

ChatGPT

你是对的。在C++中，模板类和函数与普通的类和函数有一个重要的区别：它们不是在声明时被编译，而是在实例化时被编译。这意味着编译器只有在看到模板的实际使用（即实例化）时才会为其生成代码。

因为C++的**编译模型是基于单个编译单元（通常是一个 .cpp 文件及其包含的所有 .h 文件）进行编译的**，每个编译单元是独立编译的。如果模板的定义（即模板的全部代码，包括所有成员函数的实现）不在一个编译单元中被实例化，那么编译器就没有足够的信息来为这个模板生成代码。

这正是为什么模板的定义通常都放在头文件中，而不是 .cpp 文件中。这样，无论模板在哪个编译单元中被实例化，它的定义都是可用的，编译器都可以为它生成代码。

总之，由于C++的分离式编译模型和模板在实例化时才生成代码的特性，模板的定义必须对其所有的使用者都是可见的，以确保模板可以被正确编译和链接。

一般命名为xx.hpp

资源管理

- 可以再看看modern c++32讲的相关内容

copy constructor vs assignment operator

- assignment 和 copy 的核心区别:

two is now initialized to one because it is declared as a new variable. Always remember that the assignment only occurs when giving an existing object a new value.

- 为什么要区分

When assigning an existing object a new value, the existing object's value ceases to be and we must make sure to clean up any resources the object may have allocated before setting it to the new value.

- 它们都是基于对象进行讨论的, 从这个角度来说, c++确实是一门面向对象的语言:

```
1 class MyClass {
2 public:
3     MyClass();
4     ~MyClass();
5     MyClass(const MyClass& other); // Copy constructor
6     MyClass& operator = (const MyClass& other); // Assignment
7     operator
8 };
```

- cpp默认提供的copy constructor 和assignment operator:

Unless you specify otherwise, C++ will automatically provide any class you write with a basic copy constructor and assignment operator that invoke the copy constructors and assignment operators of all the class's data members. In many cases, this is exactly what you want.

- 在某些case下, 必须弃用默认的复制构造函数和赋值重载函数 (eg: 类中含有指针, 复制之后, 执行析构函数, 将会导致对同一个内存释放两次)

- 一种方法是重写, 老实人做法
- 另一种办法: 写成private hhh

```
class CannotBeCopied {
public:
    CannotBeCopied();
    /* Other member functions. */

private:
    CannotBeCopied(const CannotBeCopied& other);
    CannotBeCopied& operator = (const CannotBeCopied& other);
};
```

- 笑死, 写成这样你就不认识了:

```

1  template <typename T> void Vector<T>::operator= (const Vector&
   other) {
2      logicalLength = other.logicalLength;
3      allocatedLength = other.allocatedLength;
4      array = new T[allocatedLength];
5      copy(other.begin(), other.end(), array);
6  }

```

换个写法:

```

1  template <typename T>
2  void Vector<T>::operator= (const Vector& other) {
3      logicalLength = other.logicalLength;
4      allocatedLength = other.allocatedLength;
5      array = new T[allocatedLength];
6      copy(other.begin(), other.end(), array);
7  }

```

- 人类高质量_类&结构: 尤其是要记忆一些复制构造函数的写法

```

1  MyClass::MyClass() : /* Fill in initializer list. */ {
2      /* Default initialization here. */
3  }
4  MyClass::MyClass(const MyClass& other) {
5      copyOther(other);
6  }
7  MyClass& MyClass::operator =(const MyClass& other) {
8      if(this != &other) {
9          clear();
10         // Note: when we cover inheritance, there's one more step
            here.
11         copyOther(other);
12     }
13     return *this;
14 }
15 MyClass::~MyClass() {
16     clear();
17 }

```

- 深拷贝/浅拷贝问题
- 解释: 为什么当你只定义了一个拷贝构造函数时, 无法使用默认构造函数?

SmartPointer

首先是类的最基本成员的定义:

mine:


```

1  template<typename T>
2  class SmartPointer {
3  private:
4      T* ptr;
5
6  public:
7      SmartPointer(T* mameory);
8      SmartPointer();
9      SmartPointer& operator =(const SmartPoniter& other);
10
11     T* operator ->() const;
12     T* operator *() const;
13 };

```

correct:

```

1  template<typename T>
2  class SmartPointer {
3  private:
4      T* ptr;
5
6  public:
7      explicit SmartPointer(T* mameory);
8      SmartPointer();
9      SmartPointer& operator =(const SmartPoniter& other);
10
11     T& operator ->() const;
12     T* operator *() const;
13 };

```

重载你函数的返回值还比较好理解，不好理解的地方在于explicit关键字的使用：

Constructs a new SmartPointer that manages the resource specified as the parameter. The reference count is initially set to one. We will assume that the provided pointer came from a call to new. This function is marked explicit so that we cannot accidentally convert a regular C++ pointer to a SmartPointer. At first this might seem like a strange design decision, but it prevents a wide range of subtle bugs. For example, suppose that this constructor is not explicit and consider the following function:

```

1  void PrintString(const SmartPointer<string>& ptr) {
2      cout << *ptr << endl;
3  }

```

This function accepts a SmartPointer by reference-to-const, then prints out the stored string. Now, what happens if we write the following code?

```
1 string* ptr = new string("Yay!");  
2 PrintString(ptr);  
3 delete ptr;
```

The first line dynamically-allocates a string, passes it to PrintString, and finally deallocates it.

Unfortunately, this code will almost certainly cause a runtime crash. The problem is that PrintString expects a SmartPointer as a parameter, but we've provided a string. C++ *notices that the*

SmartPointer has a **conversion constructor** that accepts a string, and makes a **temporary**

SmartPointer using the pointer we passed as a parameter. This new SmartPointer starts

tracking the pointer with a reference count of one. After the function returns, the parameter is cleaned

up and its destructor invokes. This decrements the reference count to zero, and then deallocates the

pointer stored in the SmartPointer. The above code then tries to delete ptr a second time, causing a

runtime crash. To prevent this problem, we'll mark the constructor explicit, which **makes the implicit**

conversion illegal and prevents this buggy code from compiling.

为什么重载运算符中，->的返回值是T&而dereference的返回值是T*?

我注意到，我在类内声明时并没有明确设置返回值类型为SmartPointer，为什么在类外实现时就需要？

ChatGPT

我明白你的困惑了。在类模板中，类名后的模板参数 `<T>` 在类内部是可选的，但在类外部定义成员函数时则是必须的。这就是导致你看到这种差异的原因。

让我们详细解释一下。

1. **在类内部**：当你在类的内部声明成员函数时，编译器知道你是在为特定的模板实例 `SmartPointer<T>` 声明这些成员。所以，在返回类型或函数参数中，你可以直接使用 `SmartPointer` 而不用写 `SmartPointer<T>`。

```
1 cppCopy codetemplate<typename T>
2 class SmartPointer {
3 public:
4     SmartPointer& operator =(const SmartPointer& other);
5 };
```

2. **在类外部**：当你在类的外部定义这些成员函数时，你需要明确地告诉编译器这个成员函数是为哪个模板实例定义的。所以，你必须使用 `SmartPointer<T>`，而不能只用 `SmartPointer`。

```
1 cppCopy codetemplate<typename T>
2 SmartPointer<T>& SmartPointer<T>::operator =(const
3     SmartPointer& other) {
4     // ...
5 }
```

这是C++模板语法的一个特性，有时会使人感到困惑，特别是对于初学者。不过一旦你理解了这一点，这个规则就变得很直观了。

RALL & modern SmartPointer

- [我的智能指针的实现](#)

注意，我这里只是一个基本的实现，最好还应该加上get函数和reset函数。get函数的好处在于，

- 对于我的原版智能指针来说，它的作用如下：

对于这种函数，原来需要一些trick才能使得myPtr表现得像是一个string (for **semantical consisittance**)

```
void DoSomething(string* ptr);

SmartPointer myPtr(new string);

DoSomething(&*myPtr);

DoSomething(myPtr.operator-> ());
```

但是有了get函数，就可以直接调用以作为函数参数传入

- 对于unique_ptr来说，其作用就更重要了，因为传入参数时会发生赋值，这会导致我们传入的指针指向的对象的资源被释放，这显然是我们不希望的

<code>T* get() const</code>	<code>SomeFunction(myPtr.get()); // Retrieve stored resource</code>
	Returns the stored pointer. Useful for passing the managed resource to other functions.

- RAII 代表 "Resource Acquisition Is Initialization" (资源获取即初始化)。这个理念的主要思想是，我们将资源（如内存、文件句柄、网络连接等）的生命周期与对象的生命周期绑定。当对象创建时，资源被获取；当对象销毁时，资源被释放。这确保了资源的正确、简单和可预测的管理。
- 智能指针是 RAII 在 C++ 中的一个经典实现。它们的主要目的是管理动态分配的对象的生命周期。传统的 C++ 动态内存管理，使用 `new` 和 `delete`，容易导致内存泄漏、双重释放等问题。

C++11 引入了以下几种智能指针：

1. **`std::unique_ptr`**：它拥有其指向的对象，保证对象有且仅有一个 `unique_ptr` 指向它。当 `unique_ptr` 离开作用域或被销毁时，它所指向的对象也会被删除。
2. **`std::shared_ptr`**：多个 `shared_ptr` 可以指向同一个对象。每个 `shared_ptr` 都有一个引用计数，记录有多少指针指向同一个对象。当最后一个指向对象的 `shared_ptr` 被销毁时，对象也会被删除。
3. **`std::weak_ptr`**：它与 `shared_ptr` 一起使用，不增加引用计数。它主要用于打破潜在的循环引用问题。

`auto_ptr` 是 C++98/03 标准库中的一个早期智能指针。在很多方面，它的行为与 `unique_ptr` 相似，特别是它们都拥有对资源的独占所有权。但是，`auto_ptr` 有一些问题和局限性，这使得它在 C++11 中被废弃并由 `unique_ptr` 取代。

c++实现RAII的方式就是对于所有的对象，都使用`std::unique_ptr`来进行初始化，赋值，这种说法是否准确？

ChatGPT

这种说法是不准确的。虽然 `std::unique_ptr` 是一个很好的RAII工具，用于自动管理动态分配的对象的生命周期，但它只是实现RAII原则的众多工具之一。

RAII 在 C++ 中的应用远不止 `std::unique_ptr`。RAII 的核心思想是将资源管理与对象的生命周期绑定。这意味着任何可以在其析构函数中释放资源的类都实现了RAII原则。

以下是RAII在C++中的几种应用：

1. **智能指针**：包括 `std::unique_ptr`、`std::shared_ptr` 和 `std::weak_ptr`。

2. **容器**：例如 `std::vector`, `std::string` 等，它们在析构时释放所有分配的内存。
3. **文件和流**：例如 C++ 的文件流类，当对象被销毁时，它们自动关闭文件。
4. **锁**：例如 `std::lock_guard` 和 `std::unique_lock`，在离开作用域时自动释放锁。
5. **自定义资源管理**：你可以为自己的资源（如数据库连接、网络套接字、图形上下文等）创建RAII类。

因此，尽管 `std::unique_ptr` 是实现RAII的一个常用工具，但在 C++ 中实现RAII并不意味着要为所有对象使用 `std::unique_ptr`。取决于你要管理的资源和使用场景，你可能会选择其他的工具或**自己定义RAII类**。

error handling

- 结合代码理解这段话：

When this occurs, you can report the error by using the throw keyword to “throw” the exception into the **nearest matching catch clause**.

```
1 try {
2     // Do something
3 }
4 catch(int myInt) {
5     // If the code throws an int, execution continues here.
6 }
7 catch(const vector<int>& myVector) {
8     // Otherwise, if the code throws a vector<int>, execution resumes
   here.
9 }
10 catch(const string& myString) {
11     // Same for string
12 }
```

- 使用stdexcept来处理错误而不是直接将错误信息作为string传给catch的好处：

1. **语义清晰**：使用特定的异常类，如 `invalid_argument`，可以清晰地表达错误的原因。这使得代码的意图更明确，读者可以立即知道错误是由于传入了无效参数。
2. **代码结构化**：使用标准异常类，**你可以更容易地使用多个 catch 块来处理不同的异常类型**。

```
1  cppCopy codetry {  
2      // ...  
3  } catch (const std::invalid_argument& ia) {  
4      // 处理无效参数  
5  } catch (const std::out_of_range& oor) {  
6      // 处理超出范围的错误  
7  }
```

3. **复用性与扩展性**：你可能有很多函数都会因为类似的原因抛出异常。如果你决定在某个时刻改变如何响应这些异常（例如，记录它们、提供不同的反馈等），使用标准异常类会更容易实现。
4. **与标准库和第三方库的兼容性**：标准库和许多第三方库在发生错误时会抛出标准异常。因此，学习和使用它们可以使你的代码与其他代码更加一致。

- **总结：能不用就不用**

in many cases there's a much better alternative that results in concise, readable, and thoroughly exception-safe code – object memory management.