

# 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
```

#####

# Learning part

---

## 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？

```
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：

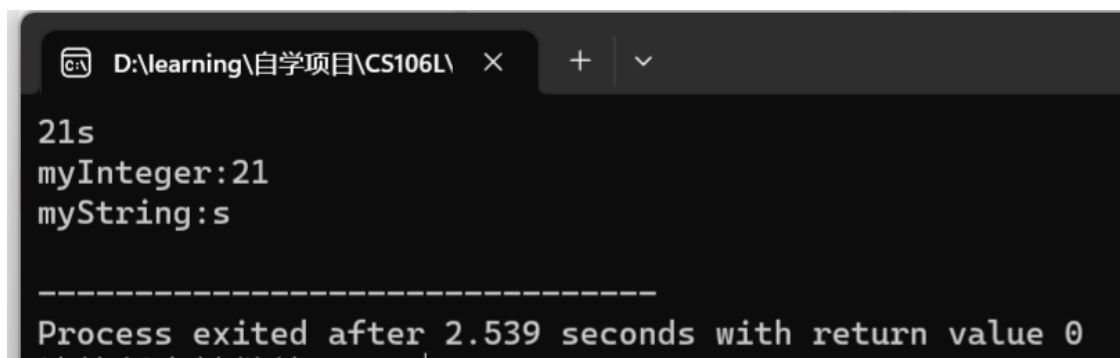
```
1 if (cin.fail()) {
2     cin.clear(); // clear the error flags
3     cin.ignore(numeric_limits<streamsize>::max(), '\n'); //
    discard invalid input
4     cout << "Invalid input. Please enter an integer." <<
    endl;
5 }
```

这样，你就可以捕获并处理无效的输入，确保程序的稳健性

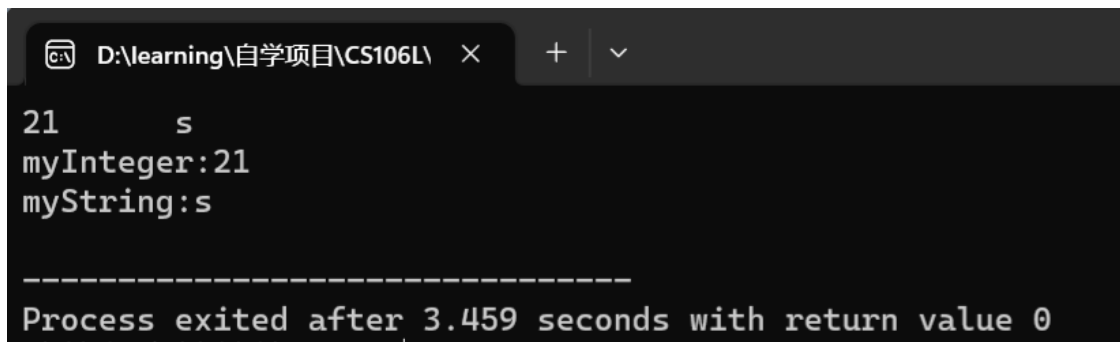
## cin流提取的终止条件

- 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

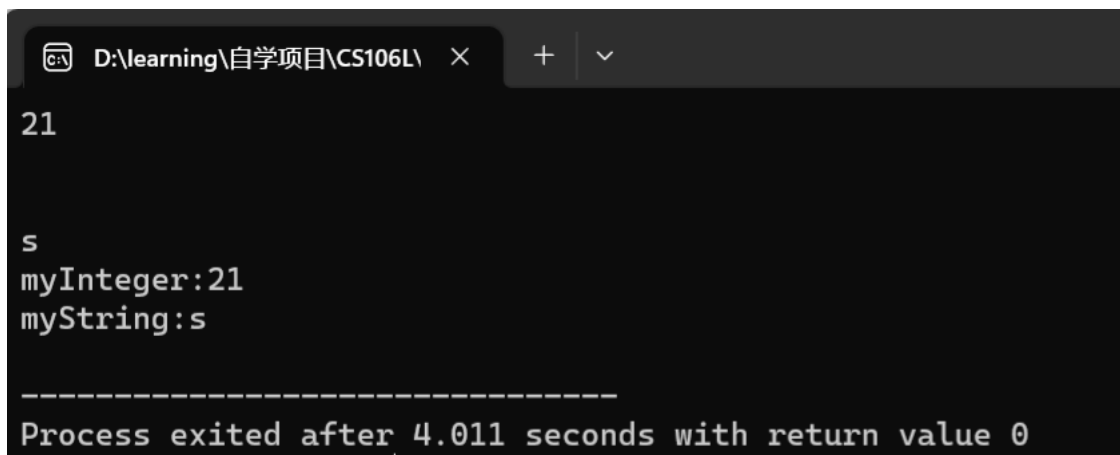
```
1  #include <iostream>
2  using namespace std;
3  int main() {
4      int myInteger;
5      string myString;
6      cin >> myInteger >> myString; // Read an integer and string
   from cin
7      cout<<"myInteger:"<<myInteger<<endl;
8      cout<<"myString:"<<myString<<endl;
9      return 0;
10 }
```



A screenshot of a C++ program execution in a terminal window. The window title is "D:\learning\自学项目\CS106L\". The output shows the program reading the integer "21" and the string "s" from standard input. The output is "myInteger:21" and "myString:s" on separate lines. A separator line of dashes is shown, followed by the message "Process exited after 2.539 seconds with return value 0".



A screenshot of a C++ program execution in a terminal window. The window title is "D:\learning\自学项目\CS106L\". The output shows the program reading the integer "21" and the string "s" from standard input. The output is "21" and "s" on separate lines, followed by "myInteger:21" and "myString:s" on separate lines. A separator line of dashes is shown, followed by the message "Process exited after 3.459 seconds with return value 0".



A screenshot of a C++ program execution in a terminal window. The window title is "D:\learning\自学项目\CS106L\". The output shows the program reading the integer "21" and the string "s" from standard input. The output is "21" and "s" on separate lines, followed by "myInteger:21" and "myString:s" on separate lines. A separator line of dashes is shown, followed by the message "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

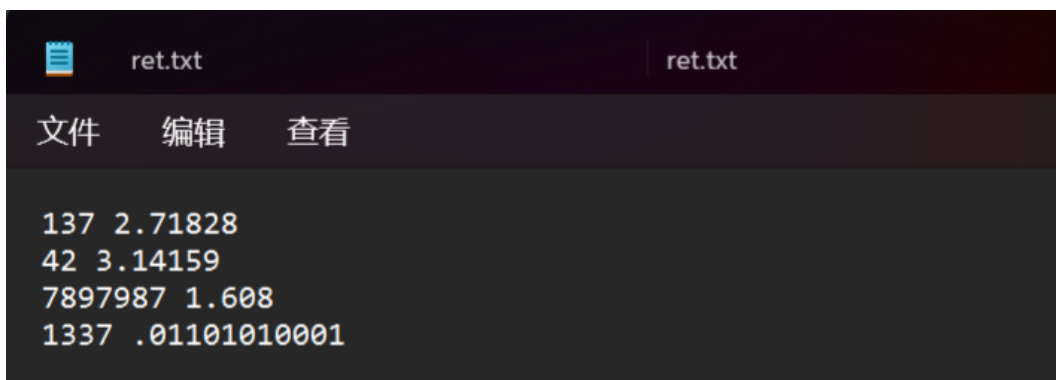
```
1 ifstream input("myfile.txt");
2 if(!input.is_open())
3     cerr << "Couldn't open the file myfile.txt" << endl;
```

## Stream Manipulators

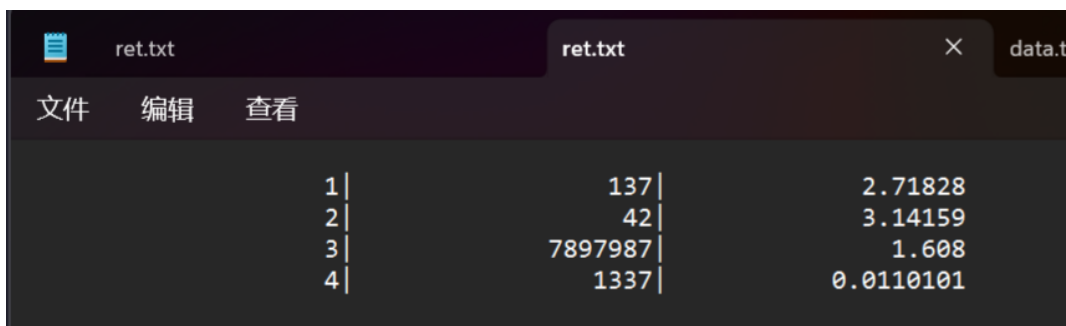
- 常用eg
  - setw
  - setfill
  - hex, dec, oct
- 这个还是挺有用的，可以用来帮处理一些杂事，尤其是写文档时候的格式问题，比如写了给目录加上.....，同时保持每一行的长度一样等等类似的重复性工作

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 }

```

你提到 `cin >> a` 的返回值是 `cin` 对象本身，`if` 语句的括号中应该是一个 `bool` 值，这两者是否冲突？

ChatGPT

不冲突。虽然 `cin >> a` 返回的是 `cin` 对象本身，但 `istream` 类 (`cin` 的类) 重载了 `void*` 类型的类型转换操作符，或者在 C++11 及以后版本中，重载了 `bool` 类型的类型转换操作符。这样，`cin` 对象可以在需要布尔值的地方，如 `if` 语句的条件中被隐式转换为 `bool` 类型。

## 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

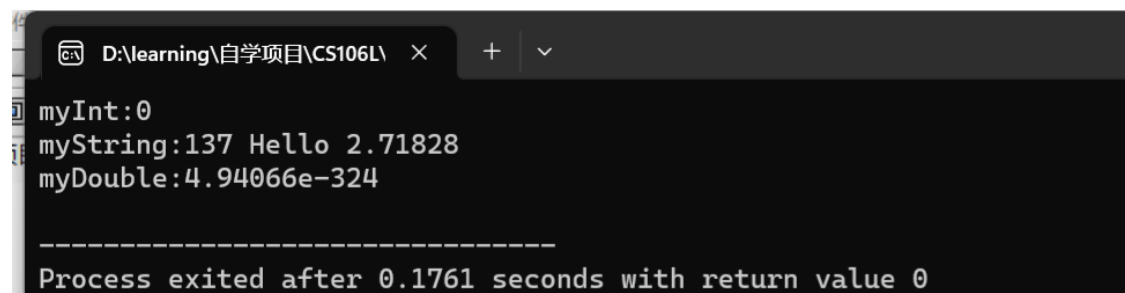
- `getline`是更契合实际读取文件的一种方式, 原因在于大部分case下我们是不希望把whitespace作为终止条件之一的。可以看看cs106L教材的例子

## 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

- 学了编译原理的相关知识之后对这里的理解应该更深了

对于下面这个简单的demo：

```
1  stringstream myConverter;
2  myConverter << "abc 123";
3
4  string temp;
5  int num;
6  myConverter >> temp >> num;
7  cout << temp << endl;
8  cout << num << endl;
9
```

abc 123本质上就是一个字符串，这里使用>>流提取运算符地过程中，本质上是调用了一个类似于lexer的词法分析器，该词法分析器的终止条件是空格或者回车，使用它会根据我们预期的数据类型，使用正则表达式进行匹配。比如这里我们先预期一个string，后预期一个int



## 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 int main()
29 {
30
31     int a;
32     getline(cin, a);
33     cout << "a: " << a << endl;
34     return 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)
  - GetInteger预期功能:
    - 参数: 输入流对象is
    - 返回: 从is中读取到的整数
    - 仅以回车为终止条件, 需要获得一个完整的整数, 完整整数的前后允许空格存在

```
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,
16                input is invalid
17                cout << "Unexpected character: " << remaining
18                << endl;
19        }
20    }
21}
```

```

17         else
18             return result;
19     }
20     else
21         cout << "Please enter an integer." << endl;
22         cout << "Retry: ";
23     }
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
3     #include <iostream>;
4     #include
5     <iostream> // Error: Multi-line preprocessor directives are
6     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

```

```

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.

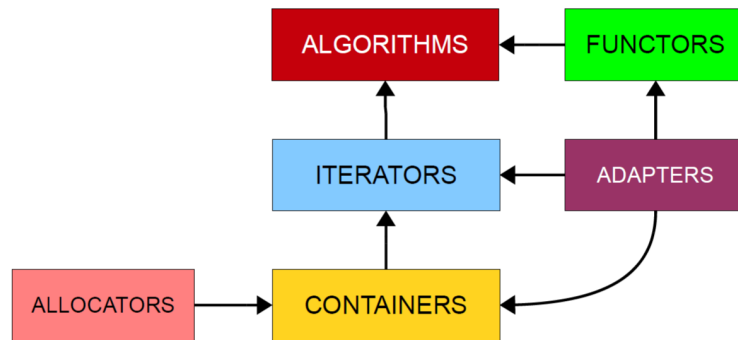
- Adapters. STL adapters are objects which transform an object from one form into another. For example, the stack adapter transforms a regular vector or list into a LIFO container, while the istream\_iterator transforms a standard C++ stream into an STL iterator.

- 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

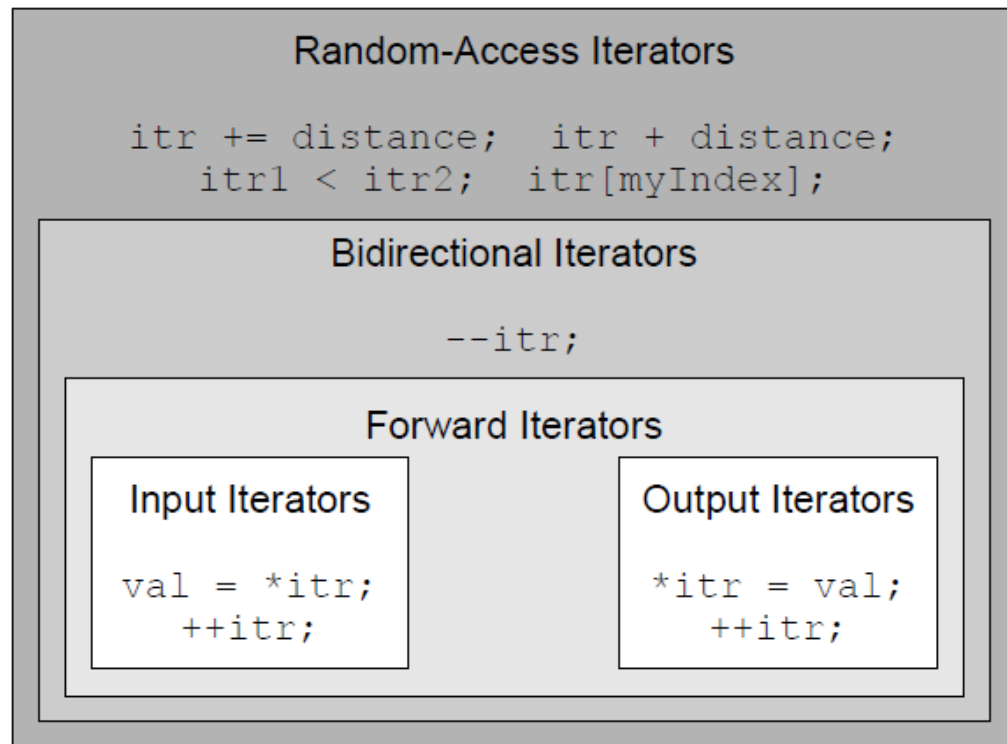
- background: 对于vector/deque，我们可以通过类似数组索引的方式来对其中的element进行traverse，但是对于set不可以
- 这就是Abstraction的魅力了：Iterators provide a clean, consistent mechanism for accessing data stored in containers, **irrespective of** how that data may be stored
- 请明确说出一个container的end()函数的返回值
- more: `std::set` 在 C++ 中是一个有序集合，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, 例如make\_pair就经常被用到

```

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不支持更新已经存在的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 =
3   myMap.insert(make_pair("STL", 137));
4 /* 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
- stringstream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 SimulatedDFA(DFA& d, string input) {
8      int currState = d.startState;
9      for (string::iterator itr = input.begin(); itr !=
input.end(); ++itr) {
10         cout << "当前状态转移: " << currState << "-->"
<< d.transitions[make_pair(currState, *itr)] << endl;;
11         currState = d.transitions[make_pair(currState,
*itr)];
12
13
14     }
15     //cout << "currState:" << currState << endl;
16     return d.acceptingStates.find(currState) !=
d.acceptingStates.end();
17 }
18

```

■ 初始化:

```

1
2  //=====下面是针对邮箱地址设置的相关
内容，上面是通用设计=====
3  struct DFA LoadEmailDFA() {
4      struct DFA ret;
5      ret.startState = 0;
6      ret.acceptingStates.insert(6);
7
8
9      //这里反而是最麻烦的地方hhh，打错一个地方就g
10
11     ret.transitions.insert(make_pair(make_pair(0, 'a'), 1));
//注意，这里外层还需要一次make_pair
12     ret.transitions.insert(make_pair(make_pair(0,
'.'), 7));
13     ret.transitions.insert(make_pair(make_pair(0,
'@'), 7));
14
15     ret.transitions.insert(make_pair(make_pair(1,
'a'), 1));
16     ret.transitions.insert(make_pair(make_pair(1,
'@'), 3));
17     ret.transitions.insert(make_pair(make_pair(1,
'.'), 2));
18
19     ret.transitions.insert(make_pair(make_pair(2,
'a'), 1));
20     ret.transitions.insert(make_pair(make_pair(2,
'@'), 7));

```

```

20     ret.transitions.insert(make_pair(make_pair(2,
21     '.', 7)));
22     ret.transitions.insert(make_pair(make_pair(3,
23     '@'), 7));
24     ret.transitions.insert(make_pair(make_pair(3,
25     '.', 7));
26     ret.transitions.insert(make_pair(make_pair(3,
27     'a'), 4));
28     ret.transitions.insert(make_pair(make_pair(4,
29     '@'), 7));
30     ret.transitions.insert(make_pair(make_pair(4,
31     '.', 5));
32     ret.transitions.insert(make_pair(make_pair(4,
33     'a'), 4));
34     ret.transitions.insert(make_pair(make_pair(5,
35     '@'), 7));
36     ret.transitions.insert(make_pair(make_pair(5,
37     '.', 7));
38     ret.transitions.insert(make_pair(make_pair(5,
39     'a'), 6));
40     ret.transitions.insert(make_pair(make_pair(6,
41     '.', 5));
42     ret.transitions.insert(make_pair(make_pair(6,
43     'a'), 6));
44     ret.transitions.insert(make_pair(make_pair(6,
45     '@'), 7));
46     ret.transitions.insert(make_pair(make_pair(7,
47     '@'), 7));
48     ret.transitions.insert(make_pair(make_pair(7,
49     '.', 7));
50     ret.transitions.insert(make_pair(make_pair(7,
51     'a'), 7));
52
53     return ret;
54 }
55
56 bool IsEmailAddress(string input) {
57     DFA emailChecker = LoadEmailDFA();
58     for (string::iterator itr = input.begin(); itr !=
59     input.end(); ++itr) {
60         if (isalnum(*itr))
61             *itr = 'a';
62         else if (*itr != '.' && *itr != '@')

```

```

52         return false;
53     }
54     return simulateDFA(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(),
7     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

首先结合几个例子体会它的作用：

- ostream\_iterator

```
1 vector<int> myVector({ 1,2,3,4,5,77 });
2
3 copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout,
  " "));
```

- insert\_iterator

```
1 vector<int> myVector; /* Initially empty */
2 /* Create a back_insert_iterator that inserts values into
   myVector. */
3 back_insert_iterator< vector<int> > itr(myVector);
4 for (int i = 0; i < 10; ++i) {
5     *itr = i; // "write" to the back_insert_iterator, appending
   the value.
6     ++itr;
7 }
8 /* Print the vector contents; this displays 0 1 2 3 4 5 6 7 8 9 */
9 copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout,
  " "));
```

第一个例子中体现出的adaptor的作用很明显，但是第二个看起来就有点多此一举？事实并非如此，

makes the iterator adaptors so useful? The key point is that iterator adaptors are *iterators*, and so they can be used in conjunction with the STL algorithms. Whenever an STL algorithm expects a regular iterator,

- 结合下面的case1和case2的运行结果即可体会iterator adaptor的强大作用

```

1  vector<int> vec({ 1,2,3,4,5 });
2  //下面把这五个元素赋值给另一个
3  vector<int> dst;
4  //case 1
5  //copy(vec.begin(), vec.end(), dst.begin());
6
7  //case 2
8  back_insert_iterator<vector<int>> itr(dst);
9  copy(vec.begin(), vec.end(), itr);
10
11 //output
12 copy(dst.begin(), dst.end(), ostream_iterator<int>(cout, "
  ));

```

case2存在更简洁的写法:

```

1  copy(vec.begin(), vec.end(), back_inserter(dst));

```

这里背后的本质是: the line `*itr = i` will implicitly call `myVector.push_back(i)`

这也就意味着, 这个`back_insert_iterator`仅仅对sequence container生效, 对于associative container无效

solution也是有的: `insert_iterator`(相应的简洁写法`inserter`)。有一个注意点:

inserts its elements into the result set. For somewhat technical reasons, when inserting elements into a set, you must specify both the container and the container's begin iterator as parameters, though the generated elements will be stored in sorted order.

- more example:

```

copy(istream_iterator<int>(input), istream_iterator<int>(),
    inserter(values, values.begin()));

```

## 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.

- 呃呃，很合理啊，比如下面这个例子：

```
1 double f(char c) {
2     return toupper(c)+0.1;
3 }
4 void tempTest() {
5     string input("abcdefg");
6     vector<double> output;
7     transform(input.begin(),
8               input.end(), inserter(output, output.begin()), f);
9     copy(output.begin(), output.end(), ostream_iterator<double>
10          (cout, " "));
11 }
```



## 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<cctype>
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>
2 class MyPair {
3 public:
4     FirstType getFirst();
5     void setFirst(FirstType newValue);
6     SecondType getSecond();
7     void setSecond(SecondType newValue);
8 private:
9     FirstType first;
10    SecondType second;
11 };
12
13 FirstType MyPair::getFirst() { // Problem: Not legal syntax
14     return first;
15 }

```



- 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
   operator
7
8      /* ... */
9
10 };

```

- 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

```

o
    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\*?

- 评价为细节拉满

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

- 智能指针的生命周期的结束伴随着其析构函数的执行，其析构函数的执行伴随着 `delete` 函数的调用，因此RAII原则得到实现
- [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 consisitance**)

```
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.

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。

## 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 code  
2      try {  
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.

- `assert`:

意义在于可以在程序break down时输出错误信息，而不只是报一个programmer难以理解的错误。可以说，这是使用cout来debug的进阶版本。

## Generic Programming

### functor

- functor和function的核心区别
- 背景问题



We need a way to construct a function that takes in only one parameter (the string to test), but which can be customized to accept an arbitrary maximum length. How can we do this?

一个典型的例子：count\_if的第三个参数必须是一个仅接受一个参数的函数（指针）

- functor的作用

```
1 public:
2     /* Accept and store an int parameter */
3     explicit ShorterThan(size_t maxLength) : length(maxLength) {}
4     /* Return whether the string length is less than the stored
5     int. */
6     bool operator() (const string& str) const {
7         return str.length() < length;
8     }
9 private:
10    const size_t length;
```

然后我们的需求就被满足了：

```
1 ShorterThan st(length);
2 count_if(myVector.begin(), myVector.end(), st);
```

一个更简单的写法：

```
1 count_if(myVector.begin(), myVector.end(), ShorterThan(length));
```

这里其实不算很好理解：ShorterThan(length)和函数指针显然是不等价的。

在 C++ 中，functor（也被称为函数对象）是一个具有 `operator()` 的对象。由于它们是对象，所以可以拥有状态，这是它们与普通函数和函数指针的主要区别。在给出的示例中，functor `ShorterThan` 存储了一个 `length` 状态，可以在调用时使用这个状态。

## 为什么 Functor 可以替代函数指针？

在你给出的 `count_if` 示例中，functor 可以替代函数指针的原因是，C++ 的标准模板库（STL）算法是模板化的。`count_if` 函数模板不仅可以接受普通函数指针，也可以接受任何可以调用的对象，只要它们接受正确的参数并返回正确的类型。因为 `ShorterThan` 有 `operator()`，所以它是可调用的。

这种灵活性来自于 C++ 模板的特性。在编译时，编译器会为给定的参数类型生成特定的函数或类的实例。所以，无论你传递一个函数指针、lambda 表达式还是 functor，只要它是可调用的，并且满足参数和返回类型的要求，它就可以与 `count_if` 等算法一起使用。

ok,误会解除。下面的问题是, 对于更普遍的case, 函数(指针)和函数对象是不能混为一用的。例如:

```
1  const double kLowerBound = 0.0;
2  const double kUpperBound = 1.0;
3  const int kNumSteps = 25;
4  const double kStepSize = (kUpperBound - kLowerBound) / kNumSteps;
5  void TabulateFunctionValues(double function(double)) {
6      for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
7          cout << "f(" << i << ") = " << function(i) << endl;
8  }
```

这里我们就不能把functor传进去

- solution:

```
1  template <typename UnaryFunction>
2  void TabulateFunctionValues(UnaryFunction function) {
3      for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
4          cout << "f(" << i << ") = " << function(i) << endl;
5  }
```

- 一个further的问题: 改成这种形式之后确实是可以接受各种形式的函数对象了, 但是它还能接受函数指针吗? ?
- 答案是可以:

我不太理解这句话: 函数模板接受任何可调用的对象

ChatGPT

我可以更详细地解释这个概念。在 C++ 中, "可调用的对象" 是一个泛化的概念, 它不仅包括函数和方法, 还包括任何重载了 `operator()` 的对象 (通常被称为 functor) 和 lambda 表达式。

## 函数模板可以接受以下类型的可调用对象:

### 1. 普通函数

- 这是最基本的可调用对象。例如, `void myFunction(int x)` 可以被直接调用, 如 `myFunction(5)`。

### 2. 函数指针

- 函数的地址可以被存储在指针变量中, 并通过这个指针调用函数。

### 3. 成员函数指针

- 类或结构体的成员函数的地址也可以被存储和调用，但需要一个对象实例来进行调用。

#### 4. Functor (函数对象)

- 这些是重载了 `operator()` 的类或结构体的对象。它们可以像函数一样被调用。

#### 5. Lambda 表达式

- 从 C++11 开始，可以在代码中直接定义匿名函数。这些也是可调用的对象。

这就是我所说的 "函数模板可以接受任何可调用的对象" 的含义。也就是说，你可以将以上任何类型的可调用对象传递给一个接受可调用对象参数的函数模板。

以下是一个例子，展示了一个函数模板是如何接受不同类型的可调用对象的：

```
1  cppCopy code#include <iostream>
2  #include <functional> // 为 std::function
3
4  // 一个普通函数
5  void function(int x) {
6      std::cout << "Function called with value: " << x <<
7      std::endl;
8  }
9
10 // 一个 functor
11 struct Functor {
12     void operator()(int x) const {
13         std::cout << "Functor called with value: " << x <<
14         std::endl;
15     }
16 };
17
18 // 一个接受可调用对象的函数模板
19 template <typename Callable>
20 void call(int x, Callable callable) {
21     callable(x);
22 }
23
24 int main() {
25     // 使用普通函数
26     call(5, function);
27
28     // 使用 functor
29     Functor functor;
30     call(10, functor);
31
32     // 使用 lambda 表达式
33     call(15, [](int x) {
```

```

32         std::cout << "Lambda called with value: " << x <<
        std::endl;
33     });
34
35     return 0;
36 }

```

在这个例子中，`call` 函数是一个模板函数，可以接受任何类型的可调用对象——普通函数、functor 或 lambda 表达式，并且以相同的方式调用它们。希望这有助于解释和展示 "可调用对象" 的概念和灵活性。

## 再探STL algorithm

### accumulate

定义：

```

1  template <typename InputIterator, typename Type, typename
   BinaryFn>
2  inline Type accumulate(InputIterator start,
3                          InputIterator stop,
4                          Type accumulator,
5                          BinaryFn fn) {
6      while(start != stop) {
7          accumulator = fn(accumulator, *start);
8          ++start;
9      }
10     return initial;
11 }

```

你不应该认为它仅仅是一个累加的函数，太狭隘了~：

eg:

```

1  class LowerBoundHelper {
2  public:
3      explicit LowerBoundHelper(int lower) : lowestValue(lower) {}
4      int operator() (int bestSoFar, int current) {
5          return current >= lowestValue && current < bestSoFar?
6              current : bestSoFar;
7      }
8  private:
9      const int lowestValue;
10 };

```

```

1 int UnsortedLowerBound(const vector<int>& input, int lowerBound) {
2     return accumulate(input.begin(), input.end(),
3         numeric_limits<int>::max(),
4         LowerBoundHelper(lowerBound));
5 }

```

If you need to transform a range of values into a single result (of any type you wish), use `accumulate`. To transform a range of values into another range of values, use `transform`. We discussed `transform` briefly

## Higher order programming

functors and the STL functional programming libraries, in many cases it is possible to write higher-order code in C++. In the remainder of this chapter, we'll explore the STL functional programming libraries and see how to use higher-order programming to supercharge STL algorithms.

这部分内容 (p397) 感觉可以用functor代替, 略多, 暂时没有看

## modern cpp

- 那这部分内容可以说是无底洞了, 课本上仅仅给出了c++0x, 这里首先整理课本内容, 然后结合《现代c++32讲》的内容进一步拓展

绷不住了, 对面来了一对, 好好好, 这么玩是吧, 他妈的 (10.05)

## lambda expression

- 示例:

```

const int myValue = GetInteger();
count_if(myVector.begin(), myVector.end(),
    [myValue](const string& x) { return x.length() < myValue; });

```

这里主要涉及两个问题的理解:

- 首先是capture list:
  - 你是正确的, lambda 默认只能访问其参数列表中的参数。
  - 通过使用捕获列表, lambda 可以访问其外部作用域的局部变量。
  - Lambda 可以直接访问全局变量, 无需使用捕获列表。

- 然后是整个lambda expression的返回对象

(我们知道cpp与cool语言类似, 本质上都是面向对象的语言, 任何expression都会返回一个“类”)

我的另一个问题是, 我们知道cpp是一个面向对象的语言, 这里的lambda表达式整体最终返回的是一个什么类型的对象呢?

在 C++ 中，lambda 表达式实际上是一个匿名函数对象。当你声明一个 lambda 表达式时，编译器会为其生成一个未命名的、唯一的函数对象类型。因为这个类型是未命名的，所以你不能直接引用它，但你可以使用 `auto` 关键字来自动推断其类型。

下面的例子展示了一个 lambda 表达式的类型是如何被自动推断的：

```
1  cppCopy code#include <iostream>
2
3  int main() {
4      auto myLambda = [](int x, int y) { return x + y; };
5
6      int result = myLambda(3, 4);
7      std::cout << "The sum is: " << result << std::endl; // 输出: The sum is: 7
8
9      return 0;
10 }
```

在上面的例子中，`myLambda` 变量的类型是由编译器生成的一个特定、未命名的函数对象类型。这个类型是唯一的，并且是根据 lambda 表达式的具体形式（包括其捕获列表和参数列表）生成的。

你也可以使用 `std::function` 来存储 lambda 表达式，如果你需要的话。例如：

```
1  cppCopy code#include <iostream>
2  #include <functional>
3
4  int main() {
5      std::function<int(int, int)> myLambda = [](int x, int y) {
6          return x + y; };
7
8      int result = myLambda(3, 4);
9      std::cout << "The sum is: " << result << std::endl; // 输出: The sum is: 7
10
11     return 0;
12 }
```

在这个例子中，`myLambda` 的类型是 `std::function<int(int, int)>`，这是一个通用的函数对象包装器，可以存储任何可以调用的实体，其调用签名与指定的调用签名相匹配。

每天一遍，焦虑再见。太喜欢这种，put everything aside的感觉了。

## move semantics

- 背景：在函数返回时，cpp会默认调用复制构造函数，这会造成毫无必要的空间浪费。
- 概念理解：左值，右值，左值引用，右值引用

评价为，gpttyds

我不太理解你提到的左值引用和右值引用的区别，请结合实例进行详细解释

ChatGPT

左值和右值是 C++ 中两种基本的表达式分类。它们的区别和具体使用场景是 C++ 中较为复杂的概念。以下我将通过例子来说明它们的区别。

### 1. 左值 (Lvalue)

左值是表达式的分类之一，它指的是那些有明确存储位置（例如变量）的对象。你可以把它理解为可以出现在赋值运算符左边的值。例如：

```
1  cppCopy codeint a = 5; // 'a' 是左值
2  a = 10;      // 'a' 可以出现在赋值运算符的左边
```

### 2. 右值 (Rvalue)

右值是指那些不是左值的表达式的分类，通常是临时的、无法寻址的、短暂存在的值。例如：

```
1  cppCopy code
2  int b = a + 5; // 'a + 5' 是右值，是一个临时表达式，不能被赋值
```

### 3. 左值引用和右值引用

左值引用是我们常见的引用类型，它必须绑定到左值上。右值引用是 C++11 新增的类型，用于绑定到右值上，通常用于实现移动语义和完美转发等。

- 左值引用示例：

```
1  cppCopy codeint a = 5;
2  int &ref = a; // 左值引用绑定到左值 'a'
```

- 右值引用示例：

```
1 cppCopy code
2 int &&rref = 5 + 3; // 右值引用绑定到右值 '5+3'
```

## 移动语义

由于右值通常是临时对象，我们可以安全地“窃取”其资源，而不必担心之后会使用这些资源。这就是移动语义的基础。

- 不使用移动语义：

```
1 cppCopy codestd::string str1 = "Hello, world!";
2 std::string str2 = str1; // 这里发生了深拷贝
```

- 使用移动语义：

```
1 cppCopy code
2 std::string str3 = std::move(str1); // str3 窃取了 str1 的资源，
   str1 现在为空
```

在这个例子中，`str3` 成为了 `str1` 的资源的新所有者，避免了深拷贝，提高了程序的效率。

## 总结

- **左值**：有明确存储位置的对象，可以被赋值。
- **右值**：临时的、无法寻址的、短暂存在的值。
- **左值引用**：绑定到左值的引用。
- **右值引用**：绑定到右值的引用，通常用于实现移动语义。

这些概念在理解 C++ 的高级特性，如移动语义、完美转发等方面是非常关键的。希望这些例子能帮助您理解左值和右值，以及它们的引用。

请结合实例解释cpp中的Move Semantics的概念

ChatGPT

C++的移动语义（Move Semantics）允许开发者直接移动资源（例如动态分配的内存），而不仅仅是复制资源。这在处理大对象或者资源管理对象（如智能指针、文件句柄、套接字等）时特别有用，可以大大提高程序的性能和效率。

移动语义通过引入了一个新类型的引用：右值引用（rvalue reference），用 `&&` 表示。与左值引用不同，右值引用可以绑定到将要销毁的对象，从而允许我们安全地移动它的资源。



下面的例子演示了移动语义的基本概念和用法：

```
1  cppCopy code#include <iostream>
2  #include <vector>
3  #include <string>
4
5  class MyString {
6  public:
7      // 构造函数
8      MyString(const char* str) {
9          std::cout << "Copy data from: " << str << std::endl;
10         size_ = strlen(str);
11         data_ = new char[size_ + 1];
12         memcpy(data_, str, size_);
13         data_[size_] = '\0';
14     }
15
16     // 拷贝构造函数
17     MyString(const MyString& other) {
18         std::cout << "Copy constructor called" << std::endl;
19         size_ = other.size_;
20         data_ = new char[size_ + 1];
21         memcpy(data_, other.data_, size_);
22         data_[size_] = '\0';
23     }
24
25     // 移动构造函数
26     MyString(MyString&& other) noexcept {
27         std::cout << "Move constructor called" << std::endl;
28         size_ = other.size_;
29         data_ = other.data_;
30         other.size_ = 0;
31         other.data_ = nullptr;
32     }
33
34     // 析构函数
35     ~MyString() {
36         delete[] data_;
37     }
38
39     // 获取字符串长度
40     size_t size() const {
41         return size_;
42     }
43
44     // 获取字符串数据
45     const char* c_str() const {
46         return data_;
47     }
```

```

48
49 private:
50     char* data_;
51     size_t size_;
52 };
53
54 int main() {
55     MyString str1("Hello, world!"); // 调用构造函数
56     MyString str2 = str1; // 调用拷贝构造函数
57
58     MyString str3("Temporary string");
59     MyString str4 = std::move(str3); // 调用移动构造函数
60
61     std::cout << "str1: " << str1.c_str() << std::endl;
62     std::cout << "str2: " << str2.c_str() << std::endl;
63     std::cout << "str4: " << str4.c_str() << std::endl;
64
65     return 0;
66 }

```

在这个例子中，`MyString` 类有一个构造函数、一个拷贝构造函数和一个移动构造函数。拷贝构造函数用于从一个已存在的对象创建一个新的对象。移动构造函数用于从一个临时对象（或者被明确地转换为右值引用的对象）"窃取" 数据，而不进行复制。这通过转移原对象的内部资源指针实现，并将原对象的内部资源指针设置为 `nullptr`，以防止在原对象析构时释放这些资源。

在 `main()` 函数中，`str2` 是通过拷贝构造函数从 `str1` 创建的；`str4` 是通过移动构造函数从 `str3` 创建的。你可以看到，通过使用移动语义，我们可以避免不必要的数据复制，从而提高程序的性能。

然后我当时提出了一个很sb的问题：

对于上面的例子，我认为移动构造函数写成下面的形式会产生同样的效果：

```

1 // 移动构造函数
2 MyString(MyString& other) noexcept {
3     std::cout << "Move constructor called" << std::endl;
4     size_ = other.size_;
5     data_ = other.data_;
6     other.size_ = 0;
7     other.data_ = nullptr;
8 }

```

提出这个问题说明你还没有完全理解右值引用存在的意义hhh。它就是为了与之区分，让程序明确地知道对于当前传进来的引用应该采取哪种类型的行为。

- `std::move`

它的出现有很现实的背景：以PA2的相关要求为例，我们调用移动构造函数时，大部分时间往往传入的仍然还是一个左值引用，那么这个时候我们就需要一个utility来起到左值向右值转换的功能，这就是std::move的作用，（其对应的头文件是utility也就很自然了）

以前面的代码为例：

```
1 MyString str4 = std::move(str3); // 调用移动构造函数
```

这里涉及到一个比较重要的基础知识：上面这段程序本质上是在执行下面这段程序：（这涉及到你对于复制构造函数和赋值构造函数调用时机的理解）

```
1 MyString str4(std::move(str3));
```

所以下面的这些case也就不难理解了(笑死，一开始忘记了这个point，人完全是懵的)

```
template <typename T>
void MoveIntoArray(T* elems, int size, int position, T&& toAdd) {
    for(int i = size; i > position; ++i)
        elems[i] = move(elems[i - 1]); // Move elements down.

    /* Note that even though toAdd is an rvalue reference, we still must
     * explicitly move it in. This prevents us from accidentally using
     * move semantics in a few edge cases.
     */
    elems[i] = move(toAdd);
}
```

# Assignment Part

## PA1

### std::find vs std::search

结合实例解释下面这段话： use std::search when looking for several elements in a row in a container, and use std::find when looking for a single element in a container).

ChatGPT

这段话说明了在 C++ 中搜索容器中的元素时如何选择 `std::search` 和 `std::find`。当你想要在容器中查找单个元素时，你应该使用 `std::find`；当你想要查找容器中的一个子序列（即几个连续的元素）时，你应该使用 `std::search`。

以下是两个示例，分别演示了如何使用 `std::find` 和 `std::search`。

## 使用 `std::find` 查找单个元素

```
1  cppCopy code#include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  int main() {
6      std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
7
8      // 使用 std::find 查找单个元素
9      auto it = std::find(numbers.begin(), numbers.end(), 5);
10
11     if (it != numbers.end()) {
12         std::cout << "Found value: " << *it << std::endl;
13     } else {
14         std::cout << "Value not found" << std::endl;
15     }
16
17     return 0;
18 }
```

在这个例子中，我们使用 `std::find` 在 `numbers` 容器中查找值 `5`。`std::find` 返回一个迭代器，指向找到的元素，如果没有找到，则返回 `end()`。

## 使用 `std::search` 查找子序列

```
1  cppCopy code#include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  int main() {
6      std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
7      std::vector<int> pattern = {4, 5, 6};
8
9      // 使用 std::search 查找子序列
10     auto it = std::search(numbers.begin(), numbers.end(),
11                           pattern.begin(), pattern.end());
12
13     if (it != numbers.end()) {
14         std::cout << "Found pattern starting at index: " <<
15         std::distance(numbers.begin(), it) << std::endl;
16     } else {
17         std::cout << "Pattern not found" << std::endl;
18     }
19
20     return 0;
21 }
```

在这个例子中，我们使用 `std::search` 在 `numbers` 容器中查找 `pattern` 子序列。如果找到了子序列，`std::search` 返回一个迭代器，指向子序列的开始位置。

这两个例子清晰地展示了如何根据需要查找单个元素或子序列来选择使用 `std::find` 或 `std::search`。

等等，我突然发现这里的search的功能有点强大呀，它在编译器前端的应用我都不敢想啊

## part A

- 这里有一个前提：默认html的语法都是正确的
- 初步体会search、find的强大作用

核心代码：[github链接](#)

```
orange0o0@LAPTOP-8TA12ITF:~/cs106L-assignment1$ ./test-wikiscraper.sh
Building Wikiscraper Test
-- Configuring done
-- Generating done
-- Build files have been written to: /home/orange0o0/cs106L-assignment1/build
Consolidate compiler generated dependencies of target test
[ 25%] Building CXX object CMakeFiles/test.dir/wikiscraper.cpp.o
[ 50%] Linking CXX executable test
[100%] Built target test
Running test: ./build/test 1
Running test: ./build/test 2
Running test: ./build/test 3
Running test: ./build/test 4
Running test: ./build/test 5
Running test: ./build/test 6
Running test: ./build/test 7
Running test: ./build/test 8
All 8 tests passed!
```

十分有九分的绷不住，对面那俩差不多得了

## part B

- first, we need to figure out what this "intelligent way" to visit each link on a given page might be:

解决按理说不难，难在高效

核心问题：

We want to search for a link ladder from the start page to the end page. The hard part in solving a problem like this is dealing with the fact that Wikipedia is enormous. We need to make sure our algorithm makes **intelligent decisions** when deciding which links to follow so that it can find a solution quickly.

solution:

The intuition is that pages dealing with similar content will often have more links in common than unrelated pages.

那么自然而然就会用到优先级队列了

- 然后优先级队列的构建时，lambda expression就有用武之地了

关于lambda表达式中的capture list的细节：

```
1 auto cmp_fn = [&w, &target_set](const vector<string>& left,
2 const vector<string>& right) {
3     // replace all of these lines.
4     (void) w;
5     (void) target_set;
6     (void) left;
7     (void) right;
8     return false; // replace this line! make sure to use
    numCommonLinks.
};
```

对上面这段程序，为什么在capture list中我们要使用&w的形式而不是仅仅使用w

ChatGPT

在这个 lambda 表达式的捕获列表中使用 `&w` 的原因是要以引用的方式捕获 `w` 变量，而不是拷贝它。如果你只使用 `w`，那么 `w` 的一个拷贝将被创建和存储在 lambda 表达式的闭包对象中，原始的 `w` 变量和 lambda 中的 `w` 变量将是两个完全独立的对象。

## 使用 `&w`（按引用捕获）

```
1 cppCopy codeauto cmp_fn = [&w, &target_set](const
2 vector<string>& left, const vector<string>& right) {
3     // 使用原始 w 变量的引用
4 };
```

这里，`w` 是以引用的方式被捕获的，这意味着在 lambda 函数中使用的 `w` 实际上是原始 `w` 变量的引用。对 `w` 的任何修改都将影响原始的 `w` 变量。

## 使用 `w`（按值捕获）

```
1 cppCopy codeauto cmp_fn = [w, target_set](const
2 vector<string>& left, const vector<string>& right) {
3     // 使用 w 变量的一个拷贝
4 };
```

在这种情况下，`w` 是通过值被捕获的，即创建了 `w` 的一个拷贝并存储在 lambda 表达式的闭包对象中。原始的 `w` 变量和 lambda 中的 `w` 变量是两个完全独立的对象。

## 选择哪种捕获方式

选择按值捕获还是按引用捕获，通常取决于你的具体需求：

- 如果你不想在 lambda 表达式中修改外部作用域中的变量，或者你想在 lambda 生命期超出原始变量生命期的情况下继续使用这些变量（例如，在多线程中），你可能会选择按值捕获。
- 如果你想在 lambda 中修改外部作用域中的变量，并且你确信在 lambda 的整个生命期中，这些变量都是可访问的，那么你可能会选择按引用捕获。

有点复杂就开摆？真有你的噯

--事实证明，静心思考，step by step, everything will workout

- 拉满了：因为我们前面贪图方便使用了 auto，在后面可以使用相应的 decltype 来获取类型。好好好，狼狈为奸。

//注意这里要用引用的方式捕获，而不应该使用值得方式进行捕获

```
1 auto cmp_fn = [&w, &target_set](const vector<string>& left, const vector<string>& right) {  
    // replace all of these lines.  
    // (void) w;  
    // (void) target_set;  
    // (void) left;  
    // (void) right;  
    // return false; // replace this line! make sure to use numCommonLinks.
```

//需求：相似度高的放在队列尾部

```
auto lSet=w.getLinkSet(*left.rbegin());  
auto rSet=w.getLinkSet(*right.rbegin());  
return numCommonLinks(lSet,target_set)<numCommonLinks(rSet,target_set);
```

```
};
```

```
// END STUDENT CODE HERE
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
// TODO: ASSIGNMENT 2 TASK 7:
```

```
// Last exercise! please instantiate the priority queue for this algorithm, called "queue". Be  
// to use your work from Task 2, cmp_fn, to instantiate our queue.
```

```
// Estimated length: 1 line
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////  
// BEGIN STUDENT CODE HERE
```

```
2 // something like priority_queue<...> queue(...);  
// please delete ALL 4 of these lines! they are here just for the code to compile.  
std::priority_queue<vector<string>,vector<vector<string>>,cmp_fn> queue;  
//throw std::invalid_argument("Not implemented yet.\n");
```

- 细节2：我们在定义的前半部分只是传递comparison函数的type，其定义是通过构造函数传递的！

- `T` is the type of thing the priority queue will store;
- `Container` is the container the priority queue will use behind the scene to hold items (the priority queue is a container adaptor, just like the stack and queue we studied in lecture!);
- `Compare` is the type of our comparison function that will be used to determine which element has the highest priority.

因此，正确的priority\_queue的定义如下：

```
1 | std::priority_queue<vector<string>, vector<vector<string>>, decltype  
   | e(cmp_fn)> queue(cmp_fn);
```

呃呃，PA1暂时到这就结束了，后半部分由于网络原因无法进行测试。

(命令行似乎无法访问国外网站)

## PA2

- assignment中的一些重要事项

When a user of the hashmap class tries to look up or insert a key, the HashMap class first converts the key, no matter what type it is, to an integer. I

- 核心目标不是通过test，切记

We have implemented all of this logic for you, and we highly encourage you to read through the code to understand what is happening. However, this assignment does not ask you to implement anything except some SMFs, which only deal with initializing the private member variables.

- function signature:函数声明

## 源码分析

首先，最重要的部分当然是源码分析，自然是从main函数开始：

```
1 | int main() {  
2 |  
3 |  
4 |     int answer;  
5 |     cout << "Please enter a number and then hit ENTER\n1)  
   | Milestone 1 code\n2) Milestone 2 tests" << endl;  
6 |     cin >> answer;  
7 |     if(answer == 1){  
8 |         student_main();
```



```

9     } else if (answer == 2) {
10         cout << ":P" << endl;
11         //UNCOMMENT THIS FOR MILESTONE 2
12         // run_test_harness();
13     } else {
14         cout << "Please run your code again and hit 1 or 2
only!";
15     }
16 }

```

## milestone 1

这里我们首先关注answer==1的case, 此时我们跳转到student\_main:

```

1  int student_main() {
2      cout << "This is student main. You can try using HashMap as a
client by editing the code here!" << endl;
3      HashMap<string, int> map;
4      init_map(map);
5      std::set<string> keys = find_keys(map);
6      cout << "Find the difference in time between two lecturers!
\n" <<
7          "Please enter two names from this list, separated by a
space. Then hit ENTER\n" << endl;
8      for(auto it = keys.begin(); it != keys.end(); ++it){
9          cout << *it << endl;
10     }
11     string name1;
12     string name2;
13     cin >> name1;
14     cin >> name2;
15     print_difference(map, name1, name2);
16     //cout<<"map:"<<map<<endl;
17     return 0;
18 }

```

首先, 使用默认构造函数创建了一个HashMap的实例。这里自然涉及到对HashMap结构体的解析, 但是由于该结构体预计解析较长, 因此单独进行讲解而不是在这里按照程序执行顺序进行解释。 [点此跳转到HashMap数据结构](#)

然后执行init\_map:

```

1  //This will fill parameter map with lecturer -> year they
    lectured pairs
2  void init_map(HashMap<string, int>& map) {
3      cout << "Hello from your past and current lecturers!" <<
endl;
4      map.insert({"Anna", 2019});
5      map.insert({"Avery", 2019});

```

```

6      map.insert({"Nikhil", 2020});
7      map.insert({"Ethan", 2020});
8      map.insert({"Frankie", 2021});
9      map.insert({"Sathya", 2021});
10     //demonstrate the map in action!
11     map.debug();
12     map.rehash(5);
13     map.debug();
14 }

```

呃呃，然后可以进行简单地测试：

```

orange0o0@LAPTOP-8TA12ITF:~/cs106L-assignment2/HashMap_Starter$ ./build.sh
orange0o0@LAPTOP-8TA12ITF:~/cs106L-assignment2/HashMap_Starter$ ./main
Please enter a number and then hit ENTER
1) Milestone 1 code
2) Milestone 2 tests
1
This is student main. You can try using HashMap as a client by editing the code here!
Hello from your past and current lecturers!
-----
Printing debug information for your HashMap implementation
Size: 6      Buckets: 10      (load factor: 0.6)

[ 0]: /
[ 1]: -> Ethan:2020 /
[ 2]: -> Sathya:2021 /
[ 3]: /
[ 4]: /
[ 5]: -> Anna:2019 /
[ 6]: -> Nikhil:2020 /
[ 7]: /
[ 8]: -> Frankie:2021 -> Avery:2019 /
[ 9]: /
-----
Printing debug information for your HashMap implementation
Size: 6      Buckets: 5      (load factor: 1.2)

[ 0]: -> Anna:2019 /
[ 1]: -> Nikhil:2020 -> Ethan:2020 /
[ 2]: -> Sathya:2021 /
[ 3]: -> Avery:2019 -> Frankie:2021 /
[ 4]: /
-----
Find the difference in time between two lecturers!
Please enter two names from this list, separated by a space. Then hit ENTER

Anna
Avery
Ethan
Frankie
Nikhil
Sathya
Anna Avery
Anna and Avery lectured the same year, 2019

```

下面开始处理mileston 1要求的const correctness的问题。

首先，根据给出的hint，阅读示例代码begin函数及其重载：

```

1 //一个使用const的example
2 template <typename K, typename M, typename H>
3 typename HashMap<K, M, H>::iterator HashMap<K, M, H>::begin() {

```

```

4     size_t index = first_not_empty_bucket();
5     if (index == bucket_count()) {
6         return end();
7     }
8     return make_iterator(_buckets_array[index]);
9 }
10
11 //注意函数中const关键字的使用
12 template <typename K, typename M, typename H>
13 typename HashMap<K, M, H>::const_iterator HashMap<K, M,
14 H>::begin() const {
15     // This is called the static_cast/const_cast trick, which
16     // allows us to reuse
17     // the non-const version of find to implement the const
18     // version.
19     // The idea is to cast this so it's pointing to a non-const
20     // HashMap, which
21     // calls the overload above (and prevent infinite recursion).
22     // Also note that we are calling the conversion operator in
23     // the iterator class!
24
25     //类似于委托构造函数
26     return static_cast<const_iterator>(const_cast<HashMap<K, M,
27 H>*>(this)->begin());
28 }
29

```

这里主要涉及到cpp的一个static\_cast和const\_cast的trick。

- 当时，我对此感到不解：为什么一定要两个版本的begin呢？
- 事实证明（gpt证明），确实很有必要：

## 需要 const 重载版本的原因：

在 C++ 中，成员函数后面的 `const` 修饰符表示该成员函数是一个常量成员函数，它不会修改对象的状态。这意味着，在一个 `const` 成员函数内部，你不能修改对象的任何成员变量（除非这些成员变量被声明为 `mutable`），也不能调用该对象的任何非 `const` 成员函数。

为一个成员函数提供 `const` 和非 `const` 两个版本的重载，通常是为了在不同情况下提供不同的接口。`const` 成员函数可以被 `const` 对象调用，也可以被非 `const` 对象调用；但非 `const` 成员函数只能被非 `const` 对象调用。

## 示例：

考虑下面的例子：

```
1  const HashMap<int, std::string> constMap;  
2  auto it = constMap.begin(); // 这里应该调用 const 版本的  
   begin()  
3  
4  HashMap<int, std::string> nonConstMap;  
5  auto it2 = nonConstMap.begin(); // 这里可以调用非 const 版本的  
   begin()
```

在第一个调用中，`constMap` 是一个 `const` 对象，所以它只能调用 `const` 成员函数。如果 `HashMap` 类没有提供 `const` 版本的 `begin()`，那么这个调用就会导致编译错误。

## 分析 `const` 和非 `const` 的 `begin()` 函数：

1. 非 `const` 版本的 `begin()` 返回一个普通迭代器，该迭代器可以用来修改 `HashMap` 中的元素。
2. `const` 版本的 `begin()` 返回一个 `const_iterator`。`const_iterator` 是一种特殊类型的迭代器，它不允许通过迭代器修改容器中的元素。这就确保了 `const` 对象的不可变性。

在你提供的代码示例中，`const` 版本的 `begin()` 通过 `const_cast` 临时地移除了 `this` 指针的 `const` 属性，调用了非 `const` 版本的 `begin()`，然后将结果转换为 `const_iterator`。这是一种常见的技巧，可以避免代码重复。但要注意，不当使用 `const_cast` 可能会破坏对象的 `const` 约束，导致未定义行为。

## 总结

为 `begin()` 函数提供 `const` 和非 `const` 两个版本的重载是为了保证在不同的上下文中，都能以适当、安全的方式访问容器的元素，保持对象的不可变性和接口的一致性。

那么到现在为止，我们对于milestone 1给出的const correctness的要求就很明确了：想必我们只要编译不报错，就算大成功了。

- 这个trick感觉还是挺具有通用性的。
- 同时也暴露出oop的基础知识没掌握牢固qwq。const对象只能调用const函数这种知识都需要去问gpt了qwq。
- 有一说一，这个trick确实挺不错的

这里，`const_cast` 是用来移除 `const` 限定符的，因为我们希望调用非 `const` 版本的 `begin()` 函数。原型函数 `begin() const` 是一个 `const` 成员函数，这意味着在这个成员函数内部 `this` 指针是一个指向 `const` 对象的 `const` 指针，即 `const HashMap<K, M, H>* const` 类型。我们使用 `const_cast` 移除 `const` 限定符，将 `this` 指针转换为非 `const` 类型，即 `HashMap<K, M, H>*`，这样就可以调用非 `const` 版本的 `begin()` 函数了。

在代码

```
1  
2 return static_cast<const_iterator>(const_cast<HashMap<K, M,  
    H>*>(this)->begin());
```

中，

1. `const_cast<HashMap<K, M, H>*>(this)` 移除 `this` 指针的 `const` 限定符，使其能够调用非 `const` 成员函数。
2. `const_cast<HashMap<K, M, H>*>(this)->begin()` 在非 `const` 对象上调用非 `const` 版本的 `begin()` 函数。
3. `static_cast<const_iterator>(...)` 将返回的迭代器从非 `const` 迭代器转换为 `const` 迭代器。

因此，这里的 `const_cast` 正确地将 `const` 对象转换为非 `const` 对象，以便能够调用非 `const` 版本的 `begin()` 函数。

另外，你会发现这种方式修改起来并不复杂：下面这段代码只需要把最后一个 `find` 函数调用改掉就可以进行复用了

```
1 return static_cast<const_iterator>(const_cast<HashMap<K, M, H>*>  
    (this)->find());
```

那么下一个需要考虑的问题就是，哪些函数应该进行类似的重载呢？呃呃，看你测试程序那么长，估计凡是不会修改属性值的函数都会被重载hhh。

当然还有一个思路（doge），那就是面向报错编程（🤔🤔🤔🤔）

很难抵挡住后者的诱惑啊qwq

- 先浅浅测试一下面向报错编程的合理性，下面是修改前的报错情况

```

orange0o0@LAPTOP-8TA12ITF:~/cs106L-assignment2/HashMap_Starter$ ./build.sh
In file included from main.cpp:5:
tests.cpp: In function 'void E_at_const_correct()':
tests.cpp:203:40: error: passing 'const HashMap<std::__cxx11::basic_string<char>,
  int>' as 'this' argument discards qualifiers [-fpermissive]
   203 |         VERIFY_TRUE(c_ref_map1.contains(key), __LINE__);
       |                        ^~~~~~
In file included from hashmap.h:583,
      from main.cpp:1:
hashmap.cpp:54:6: note: in call to 'bool HashMap<K, M, H>::contains(const K&) [
with K = std::__cxx11::basic_string<char>; M = int; H = std::hash<std::__cxx11::b
asic_string<char> >]'
   54 | bool HashMap<K, M, H>::contains(const K& key) {
       |      ^~~~~~
In file included from main.cpp:5:
tests.cpp:205:34: error: passing 'const HashMap<std::__cxx11::basic_string<char>,
  int>' as 'this' argument discards qualifiers [-fpermissive]
   205 |         VERIFY_TRUE(c_ref_map1.at(key) == mapped, __LINE__); // confirm p
parameter is a const reference
       |                        ^~~~~~
In file included from hashmap.h:583,
      from main.cpp:1:
hashmap.cpp:45:4: note: in call to 'M& HashMap<K, M, H>::at(const K&) [with K =
  std::__cxx11::basic_string<char>; M = int; H = std::hash<std::__cxx11::basic_str
ing<char> >]'
   45 | M& HashMap<K, M, H>::at(const K& key) {
       |      ^~~~~~
In file included from main.cpp:5:
tests.cpp:214:40: error: passing 'const HashMap<std::__cxx11::basic_string<char>,

```

根据报错信息，定位到test.cpp的203行：很明显是contains不支持被const对象调用

```

for (const auto& [key, mapped] : answer) {
    VERIFY_TRUE(c_ref_map1.contains(key), __LINE__);
    VERIFY_TRUE(map1.contains(key), __LINE__);
    VERIFY_TRUE(c_ref_map1.at(key) == mapped, __LINE__); // confirm parameter is a const reference
    void VERIFY_TRUE(bool condition, int line);
    VERIFY_TRUE(check_map_equal(map1, answer), __LINE__);
}

```

下面按照begin的示例对contains进行修改，修改后我们会发现，error它消失了，它真的，我哭死

```

orange0o0@LAPTOP-8TA12ITF:~/cs106L-assignment2/HashMap_Starter$ ./build.sh
In file included from main.cpp:5:
tests.cpp: In function 'void E_at_const_correct()':
tests.cpp:205:34: error: passing 'const HashMap<std::__cxx11::basic_string<char>,
  int>' as 'this' argument discards qualifiers [-fpermissive]
   205 |         VERIFY_TRUE(c_ref_map1.at(key) == mapped, __LINE__); // confirm p
parameter is a const reference
       |                        ^~~~~~
In file included from hashmap.h:586,
      from main.cpp:1:
hashmap.cpp:45:4: note: in call to 'M& HashMap<K, M, H>::at(const K&) [with K =
  std::__cxx11::basic_string<char>; M = int; H = std::hash<std::__cxx11::basic_str
ing<char> >]'
   45 | M& HashMap<K, M, H>::at(const K& key) {
       |      ^~~~~~
In file included from main.cpp:5:
tests.cpp:215:34: error: passing 'const HashMap<std::__cxx11::basic_string<char>,
  int>' as 'this' argument discards qualifiers [-fpermissive]
   215 |         VERIFY_TRUE(c_ref_map1.at(key) == mapped, __LINE__); // confirm p
parameter is a const reference
       |                        ^~~~~~
In file included from hashmap.h:586,
      from main.cpp:1:
hashmap.cpp:45:4: note: in call to 'M& HashMap<K, M, H>::at(const K&) [with K =
  std::__cxx11::basic_string<char>; M = int; H = std::hash<std::__cxx11::basic_str
ing<char> >]'

```

然后就有惊无险地拿下了捏~🤖🤖🤖🤖🤖🤖🤖🤖🤖 (乐, 当时只是看到了PASS就以  
为拿下了, 下面的6/8没看到吗宝子)

```
orange0o0@LAPTOP-8TA12ITF:~/cs106L-assignment2/HashMap_Starter$ ./build.sh
orange0o0@LAPTOP-8TA12ITF:~/cs106L-assignment2/HashMap_Starter$ ./main
Please enter a number and then hit ENTER
1) Milestone 1 code
2) Milestone 2 tests
2
:P

----- CS 106L SIMPLE TEST HARNESS -----
Written by Avery Wang (2019-2020 lecturer)

----- Milestone 2 Tests -----
Test A_copy_ctor_basic          FAIL: VERIFY_TRUE assertion failure at line r
umber 1266 in file tests.cpp
Test B_copy_assign_basic        FAIL: VERIFY_TRUE assertion failure at line r
umber 1318 in file tests.cpp
Test C_copy_edge                PASS
Test D_move_ctor_basic          PASS
Test E_move_assign_basic        PASS
Test F_move_edge                PASS
Move    10 elements: 130 ns
Move    100 elements: 110 ns
Move    1000 elements: 100 ns
Move    10000 elements: 110 ns
Test G_move_ctor_time           PASS
Move    10 elements: 280 ns
Move    100 elements: 200 ns
Move    1000 elements: 230 ns
Move    10000 elements: 211 ns
Test H_move_assign_time         PASS

----- Test Harness Summary -----
Required Tests passed: 6/8
Some required tests were failed or skipped.

----- End of Test Harness -----
```

## HashMap解析

首先要指出, 源代码的注释部分已经对源码进行了详细的解释, 我这下面最多只能说是对于注释的conclusion

出于简洁的考虑, 我在下面的copy中删除了所有注释, 但是注释中有很多有价值的信息 (比如提供了很多的usage), 如果不熟悉还是很建议去瞅瞅

首先是三个模板参数:

- K = key type
- M = mapped type
- H = hash function type used to hash a key; if not provided, defaults to std::hash

然后是几个using; 结合这里的例子, 对using的使用有了更深的理解!



这using关键字的作用比你想象的要大！我大胆推测，在stl中，正是我们在每一个container中都使用using封装了一个与之对应的iterator，才形成了我们对于container的一种统一的访问方式。

这里的另一个point在于，我们声明了两个iterator，具体来说，分别是：

- iterator
- const\_iterator

两者的区别在于实例化时的模板参数不同，这里具体参见[HashMapIterator解析](#)

- 然后是构造函数的声明
  - 这里，对于默认构造函数，我们使用了委托构造函数的方式（注意这里使用{}的效果和使用（）效果一样）

```
1  HashMap<K, M, H>::HashMap() :  
2      HashMap{kDefaultBuckets} { }
```

- 另一个值得注意的point是，对于有参构造函数，我们明确声明了explicit，**这里的explicit不是可有可无的，而是必要的！！**

```
1  /*  
2   * Constructor with bucket_count and hash function as  
   * parameters.  
3   *  
4   * Creates an empty HashMap with a specified initial  
   * bucket_count and hash function.  
5   * If no hash function provided, default value of H is used.  
6   *  
7   * Usage:  
8   *     HashMap(10) map;  
9   *     HashMap map(10, [](const K& key) {return key % 10; });  
10  *     HashMap map{10, [](const K& key) {return key % 10; }};  
11  *  
12  * Complexity: O(B), B = number of buckets  
13  *  
14  * Notes : what is explicit? Explicit specifies that a  
   * constructor  
15  * cannot perform implicit conversion on the parameters, or  
   * use copy-initialization.  
16  * That's good, as nonsense like the following won't compile:  
17  *  
18  * HashMap<int, int> map(1.0); // double -> int conversion  
   * not allowed.
```



```

19 * HashMap<int, int> map = 1; // copy-initialization, does
   | not compile.
20 */
21
22

```

- 然后是一些比较简单的功能函数，省略
- 然后是insert函数，这里的注意疑惑在于，它似乎没有考虑发生hash冲突的情况？

```

1  std::pair<typename HashMap<K, M, H>::iterator, bool> HashMap<K,
   | M, H>::insert(const value_type& value) {
2      const auto& [key, mapped] = value;
3      auto [prev, node_to_edit] = find_node(key);
4      //这里的index似乎没有考虑发生hash冲突的情况??
5      size_t index = _hash_function(key) % bucket_count();
6
7      if (node_to_edit != nullptr) {
8          return {make_iterator(node_to_edit), false};
9      }
10
11     auto temp = new node(value, _buckets_array[index]);
12     _buckets_array[index] = temp;
13
14     ++_size;
15     return {make_iterator(temp), true};
16 }

```

好吧，是我天真了：

实际上，哈希冲突在该 `insert` 函数中是被考虑的。哈希冲突是指两个或多个不同的键在哈希表中映射到同一个索引的情况。在这个 `HashMap` 的实现中，哈希冲突通过链地址法解决。让我们逐步解析代码。

```

1  size_t index = _hash_function(key) % bucket_count();

```

上面的代码通过哈希函数计算给定键的哈希值，并使用模运算确定该键应该插入到哪个桶（bucket）中。如果其他键也映射到这个桶中，就会发生哈希冲突。

```

1  auto [prev, node_to_edit] = find_node(key);

```

`find_node` 函数在给定的桶中查找是否已经有与给定键相等的节点。如果有，`node_to_edit` 将是一个指向该节点的指针；否则，它将是 `nullptr`。

```

1  if (node_to_edit != nullptr) {
2      return {make_iterator(node_to_edit), false};
3  }

```

如果 `node_to_edit` 不是 `nullptr`，这意味着给定的键已经存在于 `HashMap` 中，因此不会插入新节点，直接返回一个指向已存在节点的迭代器。

```
1 auto temp = new node(value, _buckets_array[index]);
2 _buckets_array[index] = temp;
```

如果 `node_to_edit` 是 `nullptr`，这意味着给定的键不在 `HashMap` 中。在这种情况下，会创建一个新节点并将其插入到正确的桶中。如果该桶已经有其他节点（即发生了哈希冲突），新节点将被插入到链的开头。这里的 `_buckets_array[index]` 存储的是链表的头指针。

所以，哈希冲突实际上是通过链地址法来处理的，这是一种常用的处理哈希冲突的方法。在这种方法中，哈希表的每个桶都存储一个链表，所有映射到该桶的键都存储在这个链表中。

- 到此为止，已经对整个HashMap的运行逻辑比较清晰了（笑死，其实主要也就是看来一个insert函数及其相关函数）

```
1
2
3 #ifndef HASHMAP_H
4 #define HASHMAP_H
5
6 #include <iostream>           // for cout
7 #include <iomanip>             // for setw, setprecision,
8                               // setfill, right
9 #include <sstream>             // for istringstream
10 #include "hashmap_iterator.h"
11 #include <vector>
12
13 template <typename K, typename M, typename H = std::hash<K>>
14 class HashMap {
15 public:
16
17     using value_type = std::pair<const K, M>;
18     using iterator = HashMapIterator<HashMap, false>;
19     using const_iterator = HashMapIterator<HashMap, true>;
20
21     friend class HashMapIterator<HashMap, false>;
22     friend class HashMapIterator<HashMap, true>;
23
24
25     HashMap();
26     explicit HashMap(size_t bucket_count, const H& hash = H());
27
28
29     ~HashMap();
```

```

30
31
32     inline size_t size();
33     inline bool empty();
34     inline float load_factor();
35     inline size_t bucket_count() const;
36     bool contains(const K& key);
37     M& at(const K& key);
38     void clear();
39     iterator find(const K& key);
40
41     /*
42     * Inserts the K/M pair into the HashMap, if the key does not
already exist.
43     * If the key exists, then the operation is a no-op.
44     *
45     * Parameters: const l-value reference to value_type (K/M
pair)
46     * Return value:
47     *             pair<iterator, bool>, where:
48     *             iterator - iterator to the value_type element
with the given key
49     *             this element may have been just
added, or may have already existed.
50     *             bool - true if the element was successfully
added,
51     *             false if the element already existed.
52     *
53     * Usage:
54     *     HashMap<int, std::string> map;
55     *     auto [iter1, insert1] = map.insert({3, "Avery"}); //
inserts {3, "Avery"}, iter1 points to that element, insert1 =
true
56     *     auto [iter2, insert2] = map.insert({3, "Anna"}); //
no-op, iter2 points to {3, "Avery"}, insert2 = false
57     *
58     * Complexity: O(1) amortized average case
59     */
60     std::pair<iterator, bool> insert(const value_type& value);
61
62     /*
63     * Erases a K/M pair (if one exists) corresponding to given
key from the HashMap.
64     * This is a no-op if the key does not exist.
65     *
66     * Parameters: const l-value reference to K, key to be
removed.
67     * Return value: true if K/M pair was found and removed,
false if key was not found.

```

```

68      *
69      * Usage:
70      *      map.erase(3);          // assuming K = int, erases
element with key 3, returns true
71      *
72      * Complexity: O(1) amortized average case, O(N) worst case,
N = number of elements
73      *
74      * Notes: a call to erase should maintain the order of
existing iterators,
75      * other than iterators to the erased K/M element.
76      */
77      bool erase(const K& key);
78
79      /*
80      * Erases the K/M pair that pos points to.
81      * Behavior is undefined if pos is not a valid and
dereferencable iterator.
82      *
83      * Parameters: const_iterator pos, iterator to element to be
removed
84      * Return value: the iterator immediately following pos,
which may be end().
85      *
86      * Usage:
87      *      auto iter = map.find(3);
88      *      auto next = map.erase(iter);    // erases element
that iter is pointing to
89      *
90      * Complexity: O(1) amortized average case, O(N) worst case,
N = number of elements
91      *
92      * Notes: a call to erase should maintain the order of
existing iterators,
93      * other than iterators to the erased K/M element.
94      */
95      iterator erase(const_iterator pos);
96
97      /*
98      * Resizes the array of buckets, and rehashes all elements.
new_buckets could
99      * be larger than, smaller than, or equal to the original
number of buckets.
100     *
101     * Parameters: new_buckets - the new number of buckets. Must
be greater than 0.
102     * Return value: none
103     *
104     * Usage:

```

```

105     *      map.rehash(30)
106     *
107     * Exceptions: std::out_of_range if new_buckets = 0.
108     *
109     * Complexity: O(N) amortized average case, O(N^2) worst
110     * case, N = number of elements
111     *
112     * Notes: our minimal HashMap implementation does not support
113     * automatic rehashing, but
114     * std::unordered_map will automatically rehash, even if you
115     * rehash to
116     * a very small number of buckets. For this reason,
117     * std::unordered_map.rehash(0)
118     * is allowed and forces an unconditional rehash. We will not
119     * require this behavior.
120     * If you want, you could implement this.
121     *
122     * Previously, this function was part of the assignment.
123     * However, it's a fairly challenging
124     * linked list problem, and students had a difficult time
125     * finding an elegant solution.
126     * Instead, we will ask short answer questions on this
127     * function instead.
128     */
129     void rehash(size_t new_buckets);
130
131     /*
132     * Returns an iterator to the first element.
133     * This overload is used when the HashMap is non-const.
134     *
135     * Usage:
136     *      auto iter = map.begin();
137     */
138     iterator begin();
139
140     /*
141     * Returns a const_iterator to the first element.
142     * This overload is used when the HashMap is const.
143     *
144     * Usage:
145     *      auto iter = cmap.begin();
146     */
147     const_iterator begin() const;
148
149     /*
150     * Returns an iterator to one past the last element.
151     * This overload is used when the HashMap is non-const.
152     *
153     * Usage:
154     *      auto iter = map.end();
155     */
156     iterator end();
157
158     /*
159     * Returns a const_iterator to one past the last element.
160     * This overload is used when the HashMap is const.
161     *
162     * Usage:
163     *      auto iter = cmap.end();
164     */
165     const_iterator end() const;

```

```

146         *      while (iter != map.end()) {...}
147         */
148     iterator end();
149
150
151     /*
152     * Function that will print to std::cout the contents of the
153     hash table as
154     * linked lists, and also displays the size, number of
155     buckets, and load factor.
156     *
157     * Parameters: none
158     * Return value: none
159     *
160     * Usage:
161     *      map.debug();
162     *
163     * Complexity: O(N), N = number of elements.
164     *
165     * Notes: debug will not compile if either K or V does not
166     support operator<< for std::ostream.
167     * this function will crash if your linked list logic is
168     incorrect (eg. forgot to reset the
169     * last node's next to nullptr). Check where the source of
170     the compiler error comes from
171     * before complaining to us that our starter code doesn't
172     work!
173     *
174     * Tip: place map.debug() in various places in the test cases
175     to figure out which operation
176     * is failing. Super useful when we debugged our code.
177     */
178     void debug();
179
180     /* EXTRA CONSTRUCTOR */
181
182     /*
183     * Range constructor
184     * Creates a HashMap with the elements in the range [first,
185     last).
186     *
187     * Requirements: InputIt must be iterators to a container
188     whose elements are pair<K, M>.
189     *
190     * Usage:
191     *      std::vector<std::pair<char, int>> vec {'a', 3},
192     {'b', 5}, {'c', 7}};
193     *      HashMap<char, int> map{vec.begin(), vec.end()};
194     */

```

```

185     * Complexity: O(N), where N = std::distance(first, last);
186     */
187     template <typename InputIt>
188     HashMap(InputIt first, InputIt last, size_t bucket_count =
kDefaultBuckets, const H& hash = H());
189
190     /*
191     * Initializer list constructor
192     * Creates a HashMap with the elements in the initializer
list init
193     *
194     * Requirements: init must be an initializer_list whose
elements are pair<K, M>.
195     *
196     * Usage:
197     *     HashMap<char, int> map{{'a', 3}, {'b', 5}, {'c',
7}};
198     *
199     * Complexity: O(N), where N = init.size();
200     *
201     * Notes: you may want to do some research on
initializer_lists. The most important detail you need
202     * to know is that they are very limited, and have three
functions: init.begin(), init.end(), and init.size().
203     * There are no other ways to access the elements in an
initializer_list.
204     * As a result, you probably want to leverage the range
constructor you wrote in the previous function!
205     *
206     * Also, you should check out the delegating constructor
note in the .cpp file.
207     */
208     HashMap(std::initializer_list<value_type> init, size_t
bucket_count = kDefaultBuckets, const H& hash = H());
209
210     /*
211     * Indexing operator
212     * Retrieves a reference to the mapped value corresponding
to this key.
213     * If no such key exists, a key/mapped value pair will be
added to the HashMap.
214     * The mapped value will have the default value for type M.
215     *
216     * Usage:
217     *     HashMap<int, std::string> map;
218     *     map[3] = "Avery"; // creates the pair {3, "Avery"}
219     *     auto name = map[3]; // name is now "Avery"
220     *     auto name2 = map[4]; // creates the pair {4, ""},
name2 is now ""

```

```

221     *
222     * Complexity: O(1) average case amortized plus complexity
of K and M's constructor
223     */
224     M& operator[](const K& key);
225
226     /* Milestone 2 headers (you need to declare these) */
227     // TODO: declare headers for copy constructor/assignment,
move constructor/assignment
228
229 private:
230     /*
231     * node structure represented a node in a linked list.
232     * Each node consists of a value_type (K/M pair) and a next
pointer.
233     *
234     * This is implemented in the private section as clients
should not be dealing
235     * with anything related to the node struct.
236     *
237     * Usage;
238     *     HashMap<K, M, H>::node n;
239     *     n->value = {3, 4};
240     *     n->next = nullptr;
241     */
242     struct node {
243         value_type value;
244         node* next;
245
246         /*
247         * Constructor with default values, so even if you forget
to set next to nullptr it'll be fine.
248         *
249         * Usage:
250         *     node* new_node = node({key, mapped}, next_ptr);
251         */
252         node(const value_type& value = value_type(), node* next
= nullptr) :
253             value(value), next(next) {}
254     };
255
256     /*
257     * Type alias for a pair of node*'s.
258     *
259     * This is used in find_node.
260     *
261     * Usage:
262     *     auto& [prev, curr] = node_pair{nullptr, new node()};
263     */

```



```

264     using node_pair = std::pair<typename HashMap::node*,
typename HashMap::node*>;
265
266     /*
267     * Finds the node N with given key, and returns a node_pair
consisting of
268     * the node whose's next is N, and N. If node is not found,
{nullptr, nullptr}
269     * is returned. If node found is the first in the list,
{nullptr, node} is returned.
270     *
271     * Example given list: front -> [A] -> [B] -> [C] -> /
272     * where A, B, C, D are pointers, then
273     *
274     * find_node(A_key) = {nullptr, A}
275     * find_node(B_key) = {A, B}
276     * find_node(C_key) = {B, C}
277     * find_node(D_key) = {nullptr, nullptr}
278     *
279     * Usage:
280     *     auto& [prev, curr] = find_node(3);
281     *     if (prev == nullptr) { ... }
282     *
283     * Complexity: O(1) amortized average case, O(N) worst case,
N = number of elements
284     *
285     * Notes: this function is necessary because when erasing, we
need to change the
286     * next pointer of the node before the one we are erasing.
287     *
288     * Hint: on the assignment, you should NOT need to call this
function.
289     */
290     node_pair find_node(const K& key) const;
291
292     /*
293     * Finds the first bucket in _buckets_array that is non-
empty.
294     *
295     * Hint: on the assignment, you should NOT need to call this
function.
296     */
297     size_t first_not_empty_bucket() const;
298
299     /*
300     * Creates an iterator that points to the element curr-
>value.
301     *

```

```

302     * Hint: on the assignment, you should NOT need to call this
function.
303     */
304     iterator make_iterator(node* curr);
305
306     /* Private member variables */
307
308     /*
309     * instance variable: _size, the number of elements, which
are K/M pairs.
310     * Don't confuse this with the number of buckets!
311     */
312     size_t _size;
313
314     /*
315     * instance variable: _hash_function, a function (K ->
size_t) that is used
316     * to hash K's to determine which bucket they should be
inserted/found.
317     *
318     * Remember to mod the output of _hash_function by
_bucket_count!
319     *
320     * Usage:
321     *     K element = // something;
322     *     size_t index = _hash_function(element) %
_bucket_count;
323     *
324     */
325     H _hash_function;
326     //咱就是说还是得把握好分寸
327     /*
328     * The array (vector) of buckets. Each bucket is a linked
list,
329     * and the item stored in the bucket is the front pointer of
that linked list.
330     *
331     * Usage:
332     *     node* ptr = _buckets_array[index];           //
_buckets_array is array of node*
333     *     const auto& [key, mapped] = ptr->value;      // each
node* contains a value that is a pair
334     */
335     std::vector<node*> _buckets_array;
336
337     /*
338     * A constant for the default number of buckets for the
default constructor.
339     */

```

```

340     static const size_t kDefaultBuckets = 10;
341
342     /*
343      * A private type alias used by the iterator class so it can
344      * traverse
345      * the buckets.
346      */
347     using bucket_array_type = decltype(_buckets_array);
348 };
349
350 /*
351  * Ask compiler to put the template implementation here.
352  */
353 #include "hashmap.cpp"
354
355 #endif // HASHMAP_H
356

```

## HashMapIterator解析

首先给出去除注释后的源码（类似于前面的HashMap，这里源码的注释同样十分详细，足以让你彻底理解代码）

```

1  template <typename Map, bool IsConst = true>
2  class HashMapIterator {
3
4  public:
5
6      using value_type      = std::conditional_t<IsConst, const
7      typename Map::value_type, typename Map::value_type>;
8
9
10     using iterator_category = std::forward_iterator_tag;
11     using difference_type   = std::ptrdiff_t;
12     using pointer           = value_type*;
13     using reference         = value_type&;
14
15     friend Map;
16     friend HashMapIterator<Map, true>;
17     friend HashMapIterator<Map, false>;
18
19     operator HashMapIterator<Map, true>() const {
20         return HashMapIterator<Map, true>(_buckets_array, _node,
21         _bucket);
22     }
23

```

```

22
23
24     reference operator*() const;
25     pointer operator->() const;
26
27     HashMapIterator<Map, IsConst>& operator++();
28     HashMapIterator<Map, IsConst> operator++(int);
29
30     template <typename Map_, bool IsConst_>
31     friend bool operator==(const HashMapIterator<Map_, IsConst_>&
1hs, const HashMapIterator<Map_, IsConst_>& rhs);
32
33     template <typename Map_, bool IsConst_>
34     friend bool operator!=(const HashMapIterator<Map_, IsConst_>&
1hs, const HashMapIterator<Map_, IsConst_>& rhs);
35
36     HashMapIterator(const HashMapIterator<Map, IsConst>& rhs) =
default;
37     HashMapIterator<Map, IsConst>& operator=(const
HashMapIterator<Map, IsConst>& rhs) = default;
38
39     HashMapIterator(HashMapIterator<Map, IsConst>&& rhs) =
default;
40     HashMapIterator<Map, IsConst>& operator=(HashMapIterator<Map,
IsConst>&& rhs) = default;
41
42 private:
43     /*
44      * Determines what is the type of the nodes that the HashMap
is using.
45      */
46     using node = typename Map::node;
47
48     /*
49      * Determines what is the type of the _buckets_array that the
HashMap is using.
50      */
51     using bucket_array_type = typename Map::bucket_array_type;
52
53     /*
54      * Instance variable: a pointer to the _buckets_array of the
HashMap this iterator is for.
55      */
56     bucket_array_type* _buckets_array;
57
58     /*
59      * Instance variable: pointer to the node that stores the
element this iterator is currently pointing to.
60      */

```

```

61     node* _node;
62
63     /*
64      * Instance variable: the index of the bucket that _node is
        in.
65      */
66     size_t _bucket;
67
68     HashMapIterator(bucket_array_type* buckets_array, node* node,
69                     size_t bucket);
70 };

```

- 关于std::conditional\_t

std::conditional\_t changes the value\_type (at compile-time) to a const one if IsConst is true.

- 然后是4个using: (gpt好样的)

当然，下面我将详细解释每个 `using` 声明，并使用示例来说明其作用。

## 1. using iterator\_category = std::forward\_iterator\_tag;

这个 `using` 声明定义了 `iterator_category` 类型，用于表示迭代器的类型。在这里，迭代器被定义为前向迭代器，意味着它可以从容器的一个元素移动到下一个元素。

### 示例：

```

1  cppCopy code// 当我们有一个前向迭代器时，我们可以这样使用它：
2  for(auto it = container.begin(); it != container.end(); ++it)
3  {
4      // 访问或操作元素 *it
5  }

```

## 2. using difference\_type = std::ptrdiff\_t;

`difference_type` 用于表示两个迭代器之间的距离。这是一个有符号整数类型，通常用于表示迭代器之间的差异。

## 示例:

```
1  cppCopy code// 使用 difference_type 来计算两个迭代器之间的距离
2  auto it1 = container.begin();
3  auto it2 = container.end();
4  std::ptrdiff_t distance = it2 - it1; // 这里 distance 是
    difference_type 类型
```

## 3. using pointer = value\_type\*;

这里的 `pointer` 类型是元素的指针类型。在这个上下文中, `value_type` 是 `HashMap` 中元素的类型, 所以 `pointer` 是指向 `HashMap` 元素的指针。

## 示例:

```
1  cppCopy code// 使用 pointer 类型来访问元素
2  auto it = hashmap.begin();
3  pointer ptr = &(*it); // 获取元素的指针
```

## 4. using reference = value\_type&;

`reference` 类型是 `HashMap` 中元素的引用类型。这使得我们可以直接访问和修改 `HashMap` 中的元素。

## 示例:

```
1  cppCopy code// 使用 reference 类型来访问和修改元素
2  auto it = hashmap.begin();
3  reference ref = *it; // 获取元素的引用
4  ref = newValue; // 修改元素的值
```

这些 `using` 声明使得 `HashMapIterator` 类型更具有可读性和可用性, 它们定义了与迭代器相关的基本类型, 这些类型在算法和其他与迭代器交互的场景中非常有用。

- 然后是friend class的声明, 比较容易理解
- 然后是一段不太理解的代码:按照注释的说法, 它可以实现从iterator到const\_iterator的implicit conversion, 例如:

```
1  iterator iter = map.begin();
2  const_iterator c_iter = iter;    // implicit conversion
```

```
1  operator HashMapIterator<Map, true>() const {
2      return HashMapIterator<Map, true>(_buckets_array, _node,
    _bucket);
3  }
```

这里主要涉及到一个类型转换操作符的概念：

类型转换操作符是类成员函数的一种，用于将类类型的对象转换为其他类型。其基本语法规则如下：

## 语法

```
1  cppCopy code
2  operator TypeName() const;
```

其中：

- `operator`：关键字，用于定义类型转换操作符。
- `TypeName`：要转换的目标类型的名称。
- `const`：(可选) 如果类型转换操作符不修改调用它的对象，则应将其声明为 `const` 成员函数。

## 注意事项

1. **无参数**：类型转换操作符不应有参数。
2. **无返回类型**：不需要（也不允许）指定返回类型，目标类型是通过操作符后面的类型名来指定的。
3. **必须是成员函数**：类型转换操作符必须是类的成员函数，不能是全局函数。

## 示例

```
1  #include <iostream>
2
3  class Meter {
4  private:
5      float distance;
6
7  public:
8      // 构造函数，用于初始化 Meter 对象
9      Meter(float m) : distance(m) {}
10
11     // 类型转换操作符，将 Meter 对象转换为 float
12     operator float() const {
13         return distance;
14     }
15 };
16
17 int main() {
18     Meter m(5.0); // 创建一个表示 5 米的 Meter 对象
19
20     float dist = m; // 使用类型转换操作符将 Meter 对象隐式转换为 float
```

```

21     std::cout << "Distance: " << dist << " meters" <<
    std::endl; // 输出: Distance: 5 meters
22
23     return 0;
24 }

```

然后就很好理解前面的代码了。

小积累，结构化绑定：

这个示例中的用法利用了 C++17 引入的结构化绑定（structured bindings）特性。结构化绑定允许你将一个结构体、类、数组或者元组中的多个成员一次性解包到多个变量中。

在示例 `auto [key, value] = *iter;` 中，假设 `iter` 是一个指向 `std::pair` 的迭代器（比如一个 `std::map` 或者本示例中的 `HashMap` 的迭代器），通过解引用 `*iter` 我们得到一个 `std::pair` 对象。通过结构化绑定，我们可以将这个 `pair` 的 `first` 和 `second` 成员分别赋给 `key` 和 `value` 变量。

下面是一个完整示例以帮助理解：

```

1  #include <iostream>
2  #include <map>
3
4  int main() {
5      std::map<int, std::string> myMap;
6      myMap[1] = "one";
7      myMap[2] = "two";
8      myMap[3] = "three";
9
10     for (const auto& iter : myMap) {
11         // 使用结构化绑定获取键和值
12         auto [key, value] = iter;
13         std::cout << "key: " << key << ", value: " << value
    << std::endl;
14     }
15
16     return 0;
17 }

```

在这个例子中，我们有一个 `std::map` 对象，其中存储了整数键和字符串值。我们使用范围基础的 `for` 循环遍历 `map`，并用结构化绑定 `auto [key, value] = iter;` 来解包每个 `std::pair` 元素的键和值。然后我们可以直接使用 `key` 和 `value` 变量。

这种结构化绑定的特性在处理成对数据或者需要将对象的成员解包到单独变量时非常有用。

- 然后是\*和->的操作符重载：（这里直接给出定义）这里主要是理解\_node的作用



```

1  typename HashMapIterator<Map, IsConst>::reference
   HashMapIterator<Map, IsConst>::operator*() const {
2      return *_node->value; // *_node can't be nullptr - that would be
   dereferencing end()
3  }
4
5  template <typename Map, bool IsConst>
6  typename HashMapIterator<Map, IsConst>::pointer
   HashMapIterator<Map, IsConst>::operator->() const {
7      return &(*_node->value); // *_node can't be nullptr - that would
   be dereferencing end()
8  }

```

小积累，关于前置递增、后置递增运算符的重载：

为什么在参数列表中加了int就表示后置递增？我不太理解？

ChatGPT

在 C++ 中，前置和后置递增/递减运算符的区别是通过 **function signature** 来区分的。为了区分它们，后置版本接受一个额外的 `int` 参数。这个 `int` 参数没有实际的意义和用途，它仅仅是一个占位符，用于在编译时区分前置和后置运算符。

## 例子：

```

1  // 前置递增
2  T& operator++();
3
4  // 后置递增
5  T operator++(int);

```

在后置递增/递减中，我们不需要实际传递任何值给这个 `int` 参数。这个 `int` 参数纯粹是为了在编译时产生一个不同的函数签名，从而区分前置和后置版本。

- 然后是==和!=运算符的重载，这两原来的代码没有实现，应该是我的work
- 然后是许多和HashMap结构体相关的private变量+相应的构造函数，这部分主要涉及[HashMap解析](#)
  - 这里值得一提的是，HashMapIterator的构造函数被设置成了 `private`，其原因如下：

```
1  /*
2   * Private constructor for a HashMapIterator.
3   * Friend classes can access the private members of class
   it is friends with,
4   * so HashMap is able to call HashMapIterator's private
   constructor
5   * (e.g, in begin()). We want the HashMapIterator
   constructor to be private
6   * so a client can't randomly construct a HashMapIterator
   without asking for one
7   * through the HashMap's interface.
8   */
```

## milestone 2

hint中有这样一段话：

Any **good STL-compliant class** must have correct **special member functions**..  
Recall that there are six major special member functions:

- Default constructor (*implemented for you*)
- Destructor (*implemented for you*)
- Copy constructor
- Copy assignment operator
- Move constructor
- Move assignment operator

这里涉及到一个概念：good STL-compliant class

一个 "good STL-compliant class" 指的是一个类，其设计和实现遵循 C++ 标准库（STL, Standard Template Library）的规范和准则。为了保证类的对象能够和 STL 容器、算法等无缝协作，必须确保这个类实现了一系列特殊的成员函数。这些成员函数包括：

1. **默认构造函数**：它可以没有任何参数，或者所有的参数都有默认值，以便可以不带任何参数地创建对象。
2. **析构函数**：负责在对象生命周期结束时进行清理工作，例如释放动态分配的内存等。
3. **拷贝构造函数**：允许以已存在的对象为模板，创建新对象的副本。
4. **拷贝赋值运算符**：允许将一个对象的内容赋给另一个已经存在的对象。
5. **移动构造函数**：允许“移动”一个对象而非复制，通常在对象内部有动态分配内存或其他资源时使用，可以提高效率。
6. **移动赋值运算符**：与移动构造函数类似，允许将一个对象的内容“移动”给另一个已经存在的对象。

好好好，开始还债了。之前的move semantics你因为觉得它篇幅有点长就不看是吧，现在还是得老老实实回头看了。

Test 4G and Test 4H time your move operations to verify that you are actually moving the contents of the HashMaps rather than copying the contents. The tests try the move operations on HashMaps of different sizes, and verify that the runtime of the move operations is  $O(1)$ , not  $O(n)$ . You can see the results of the time test printed in the test harness.