# STL初步

自己定义的struct必须要重定义小于号

## 全排列

不用写dfs了。

// 如果vec不是有序的，先排序可以枚举所有排列  
 do {  
 ++tot;  
 for (int i = 0; i < 8; ++i) {  
 sort(line[i], line[i] + 3, [&](int aa, int bb) {  
 return ord[aa] < ord[bb];   
 });  
 if (a[line[i][0]] == a[line[i][1]] && a[line[i][1]] != a[line[i][2]]) {  
 ++cnt;  
 break;  
 }  
 }  
 } while(next\_permutation(ord + 1, ord + 10));  
// 如果是C数组，就传两个指针进去

这个函数会修改容器。

## string

定义

string s;

长度（复杂度是O(1)的）

s.length();

定义流

#include <sstream>  
//...  
string line;  
stringstream ss(line);  
ss >> a; //就可以直接从line里面读或写

读取一整行

getline(cin, line);

## 重定义运算符

struct note{...}  
note operator +(note aa, note bb) {...}

## vector（可变长数组）

vector<int> a; //定义  
a.size(); //a的大小  
a.resize(); //改变a的大小  
a.push\_back(); //插入元素  
a.pop\_back(); //删除元素

## set（集合）

**集合中的元素总是按照升序排列好的**，也就是说集合可以直接lower\_bound()以及upper\_bound()

set<string> dict; //定义  
dict.insert(s); //插入元素  
for (set<string>::iterator it = dict.begin(); it != dict.end(); ++it)  
//↑遍历，其中定义了set<string>类型的名为it的迭代器  
dict.erease(a); //删除元素  
dict.find(s); //查找

### multiset

不去重的set，可以有效平替平衡树

## map（映射）

对于集合的大部分操作也可适用于map，但更多时候map是作为下标强大的数组使用的。

map<string, int> m; //这里m是string为下标，里面存int的数组  
m["July"] = 7; //于是就可以有这样的操作

由于其下标功能强大，map通常可以用来做离散处理，只是在复杂度上套了一个log。（还有一个很大的常数）

如果访问到了不存在的键，map会自动给该键赋值，int类型的默认值为0。若需要判断某个键是否存在，可以用count()函数，返回某个键出现的次数（map中的键是唯一的）。

### unordered\_map

unordered\_map是基于哈希实现的，其内部没有顺序，但复杂度是O(1)的。支持如下操作：

unorderd\_map<string, int> list; //定义  
list["233"] = 233; //赋值  
list.count(233); //可以用来判断是否存在  
list.erease(233); //删除映射关系  
list.clear(); //清空

## stack, queue（栈、队列）

int a;  
stack<int> s;  
s.push(1);  
a = s.top();  
s.pop();

对于队列而言，操作是类似的，只有取队首不同：a = q.front()

不过于我个人而言，我更倾向于手写队列

### priority\_queue（优先队列）

常用于Dijkstra的堆优化，求单源最短路。默认的堆顶是最大的元素。其操作和stack相同，也是用top()取堆顶。

如果想要堆顶元素最小，可以有以下几种方法：

* 存相反数进去；
* 定义一个struct，然后把小于号重定义为大于号；
* 按照如下方法定义优先队列，但需要注意的是，最后“< <”中间必须有空格，不然有些（并非所有）编译器会认为这是位移运算符。

priority\_queue<int, vector<int>, greater<int> > //top()取出来的就是最大的