机试技巧与STL

vs2018 快捷键

`CTRL + J`	列出成员
`Ctrl+E,D`	格式化全部代码
`Ctrl+K,F`	格式化选中的代码
`CTRL + SHIFT + E`	显示资源视图

```
F12                         转到定义
CTRL + F12                  转到声明
CTRL + ALT + J              对象浏览
CTRL + ALT + F1             帮助目录
CTRL + F1                   动态帮助
CTRL + K, CTRL + C          注释选择的代码
CTRL + K, CTRL + U          取消对选择代码的注释
CTRL + U                    转小写
CTRL + SHIFT + U            转大写
F5                          运行调试
CTRL + F5                   运行不调试
F10                         跨过程序执行
F11                         单步逐句执行
```

头文件

标准c库

头文件	说明	头文件	说明	头文件	说明
assert.h	断言相关	ctype.h	字符类型判断	errno.h	标准错误机制
float.h	浮点限制	limits.h	整形限制	locale.h	本地化接口
math.h	数学函数	setjmp.h	非本地跳转	signal.h	信号相关
stdarg.h	可变参数处理	stddef.h	宏和类型定义	stdio.h	标准I/O
stdlib.h	标准工具库	string.h	字符串和内存处理	time.h	时间相关

c++ STL

using namespace std;

头文件	说明	头文件	说明	头文件	说明
algorithm	通用算法	deque	双端队列	vector	向量
iterator	迭代器	stack	栈	map	图（键值对）
list	列表	string	字符串	set	集合
queue	队列	bitset	bit类	numeric	数值算法

常用头

```
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<iostream>
#include<string>
#include<vector>
#include<stack>
#include<bitset>
#include<cstdlib>
#include<cmath>
#include<set>
#include<list>
#include<deque>
#include<map>
#include<queue>
using namespace std;
```

常用宏定义

```
//求最大值和最小值
#define  MAX(x,y) (((x)>(y)) ? (x) : (y))
#define  MIN(x,y) (((x) < (y)) ? (x) : (y))

//取余
#define  mod(x) ((x)%MOD)

//for循环
#define  FOR(i,f_start,f_end) for(int i=f_start;i<=f_end;++i)

//返回数组元素的个数
```

```
#define  ARR_SIZE(a)  (sizeof((a))/sizeof((a[0])))

//初始化数组
#define MT(x,i) memset(x,i,sizeof(x))
#define MEM(a,b) memset((a),(b),sizeof(a))

//符号重定义
#define LL long long
#define ull unsigned long long
#define pii pair<int,int>

//常见常数
#define PI acos(-1.0)
#define eps 1e-12
#define INF 0x3f3f3f3f //int最大值
const int INF_INT = 2147483647;
const ll INF_LL = 9223372036854775807LL;
const ull INF_ULL = 18446744073709551615Ull;
const ll P = 92540646808111039LL;
const ll maxn = 1e5 + 10, MOD = 1e9 + 7;
const int Move[4][2] = {-1,0,1,0,0,1,0,-1};
const int Move_[8][2] = {-1,-1,-1,0,-1,1,0,-1,0,1,1,-1,1,0,1,1};
```

结构体

```
typedef struct{int id;int h;} node;
bool operator <(const node& a,const node & b){return (a.h)
<(b.h);}
```

常用STL

> 参考：

https://blog.csdn.net/f_zyj/article/details/51594851
https://download.csdn.net/download/f_zyj/9988653

algorithm

头文件：lgorithm

函数参数，返回值以及具体的使用方法请自行去头文件找定义！！！

不修改内容的序列操作

函数	说明
adjacent_find	查找两个相邻（Adjacent）的等价（Identical）元素
all_ofC++11	检测在给定范围中是否所有元素都满足给定的条件
any_ofC++11	检测在给定范围中是否存在元素满足给定条件
count	返回值等价于给定值的元素的个数
count_if	返回值满足给定条件的元素的个数
equal	返回两个范围是否相等
find	返回第一个值等价于给定值的元素
find_end	查找范围*A*中与范围*B*等价的子范围最后出现的位置
find_first_of	查找范围*A*中第一个与范围*B*中任一元素等价的元素的位置
find_if	返回第一个值满足给定条件的元素
find_if_notC++11	返回第一个值不满足给定条件的元素
for_each	对范围中的每个元素调用指定函数
mismatch	返回两个范围中第一个元素不等价的位置
none_ofC++11	检测在给定范围中是否不存在元素满足给定的条件
search	在范围*A*中查找第一个与范围*B*等价的子范围的位置
search_n	在给定范围中查找第一个连续*n*个元素都等价于给定值的子范围的位置

修改内容的序列操作

函数	说明
copy	将一个范围中的元素拷贝到新的位置处
copy_backward	将一个范围中的元素按逆序拷贝到新的位置处
copy_ifC++11	将一个范围中满足给定条件的元素拷贝到新的位置处
copy_nC++11	拷贝 n 个元素到新的位置处

fill	将一个范围的元素赋值为给定值
fill_n	将某个位置开始的 n 个元素赋值为给定值
generate	将一个函数的执行结果保存到指定范围的元素中，用于批量赋值范围中的元素
generate_n	将一个函数的执行结果保存到指定位置开始的 n 个元素中
iter_swap	交换两个迭代器（Iterator）指向的元素
moveC++11	将一个范围中的元素移动到新的位置处
move_backwardC++11	将一个范围中的元素按逆序移动到新的位置处
random_shuffle	随机打乱指定范围中的元素的位置
remove	将一个范围中值等价于给定值的元素删除
remove_if	将一个范围中值满足给定条件的元素删除
remove_copy	拷贝一个范围的元素，将其中值等价于给定值的元素删除
remove_copy_if	拷贝一个范围的元素，将其中值满足给定条件的元素删除
replace	将一个范围中值等价于给定值的元素赋值为新的值
replace_copy	拷贝一个范围的元素，将其中值等价于给定值的元素赋值为新的值
replace_copy_if	拷贝一个范围的元素，将其中值满足给定条件的元素赋值为新的值
replace_if	将一个范围中值满足给定条件的元素赋值为新的值
reverse	反转排序指定范围中的元素
reverse_copy	拷贝指定范围的反转排序结果
rotate	循环移动指定范围中的元素
rotate_copy	拷贝指定范围的循环移动结果
shuffleC++11	用指定的随机数引擎随机打乱指定范围中的元素的位置
swap	交换两个对象的值
swap_ranges	交换两个范围的元素

transform	对指定范围中的每个元素调用某个函数以改变元素的值
unique	删除指定范围中的所有连续重复元素，仅仅留下每组等值元素中的第一个元素。
unique_copy	拷贝指定范围的唯一化（参考上述的 unique）结果

划分操作

函数	说明
is_partitionedC++11	检测某个范围是否按指定谓词（Predicate）划分过
partition	将某个范围划分为两组
partition_copyC++11	拷贝指定范围的划分结果
partition_pointC++11	返回被划分范围的划分点
stable_partition	稳定划分，两组元素各维持相对顺序

排序操作

函数	说明
is_sortedC++11	检测指定范围是否已排序
is_sorted_untilC++11	返回最大已排序子范围
nth_element 部份排序指定范围中的元素，使得范围按给定位置处的元素划分	
partial_sort	部份排序
partial_sort_copy	拷贝部分排序的结果
sort	排序
stable_sort	稳定排序

二分法查找操作

函数	说明
binary_search	判断范围中是否存在值等价于给定值的元素
equal_range	返回范围中值等于给定值的元素组成的子范围
lower_bound	返回指向范围中第一个值大于或等于给定值的元素的迭代器
upper_bound	返回指向范围中第一个值大于给定值的元素的迭代器

集合操作

函数	说明
includes	判断一个集合是否是另一个集合的子集
inplace_merge	就绪合并
merge 合并	
set_difference	获得两个集合的差集
set_intersection	获得两个集合的交集
set_symmetric_difference	获得两个集合的对称差
set_union	获得两个集合的并集

堆操作

函数	说明
is_heap	检测给定范围是否满足堆结构
is_heap_untilC++11	检测给定范围中满足堆结构的最大子范围
make_heap	用给定范围构造出一个堆
pop_heap	从一个堆中删除最大的元素
push_heap	向堆中增加一个元素
sort_heap	将满足堆结构的范围排序

最大/最小操作

函数	说明
is_permutationC++11	判断一个序列是否是另一个序列的一种排序
lexicographical_compare	比较两个序列的字典序
max	返回两个元素中值最大的元素
max_element	返回给定范围中值最大的元素
min	返回两个元素中值最小的元素
min_element	返回给定范围中值最小的元素
minmaxC++11	返回两个元素中值最大及最小的元素
minmax_elementC++11	返回给定范围中值最大及最小的元素
next_permutation	返回给定范围中的元素组成的下一个按字典序的排列
prev_permutation	返回给定范围中的元素组成的上一个按字典序的排列

vector

头文件：vector

在STL的vector头文件中定义了vector（向量容器模版类），vector容器以连续数组的方式存储元素序列，可以将vector看作是以顺序结构实现的线性表。当我们在程序中需要使用动态数组时，vector将会是理想的选择，vector可以在使用过程中动态地增长存储空间。vector模版类需要两个模版参数，第一个参数是存储元素的数据类型，第二个参数是存储分配器的类型，其中第二个参数是可选的，如果不给出第二个参数，将使用默认的分配器

下面给出几个常用的定义vector向量对象的方法示例：

```
vector<int> s;
//  定义一个空的vector对象，存储的是int类型的元素
vector<int> s(n);
//  定义一个含有n个int元素的vector对象
vector<int> s(first, last);
//  定义一个vector对象，并从由迭代器first和last定义的序列[first, last)中复制初值
```

vector的基本操作：

```
s[i]                    //  直接以下标方式访问容器中的元素
s.front()               //  返回首元素
s.back()                //  返回尾元素
s.push_back(x)          //  向表尾插入元素x
s.size()                //  返回表长
s.empty()               //  表为空时，返回真，否则返回假
s.pop_back()            //  删除表尾元素
s.begin()               //  返回指向首元素的随机存取迭代器
s.end()                 //  返回指向尾元素的下一个位置的随机存取迭代器
s.insert(it, val)       //  向迭代器it指向的元素前插入新元素val
s.insert(it, n, val)//  向迭代器it指向的元素前插入n个新元素val
s.insert(it, first, last)
//  将由迭代器first和last所指定的序列[first, last)插入到迭代器it指向的元素前面
s.erase(it)             //  删除由迭代器it所指向的元素
s.erase(first, last)//  删除由迭代器first和last所指定的序列[first, last)
s.reserve(n)            //  预分配缓冲空间，使存储空间至少可容纳n个元素
s.resize(n)             //  改变序列长度，超出的元素将会全部被删除，如果序列需要扩展（原空间小于n），元素默认值将填满扩展出的空间
s.resize(n, val)        //  改变序列长度，超出的元素将会全部被删除，如果序列需要扩展（原空间小于n），val将填满扩展出的空间
s.clear()               //  删除容器中的所有元素
s.swap(v)               //  将s与另一个vector对象进行交换
s.assign(first, last)
//  将序列替换成由迭代器first和last所指定的序列[first, last)，[first, last)不能是原序列中的一部分

//  要注意的是，resize操作和clear操作都是对表的有效元素进行的操作，但并不一定会改变缓冲空间的大小
//  另外，vector还有其他的一些操作，如反转、取反等，不再一一列举
//  vector上还定义了序列之间的比较操作运算符（>、<、>=、<=、==、!=），可以按照字典序比较两个序列。
//  还是来看一些示例代码吧……

/*
 * 输入个数不定的一组整数，再将这组整数按倒序输出
 */
```

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> L;
    int x;
    while(cin >> x)
    {
        L.push_back(x);
    }
    for (int i = L.size() - 1; i >= 0; i--)
    {
        cout << L[i] << " ";
    }
    cout << endl;
    return 0;
}
```

list

头文件：list

下面给出几个常用的定义list对象的方法示例：

```cpp
list<int>a{1,2,3}
list<int>a(n)    //声明一个n个元素的列表，每个元素都是0
list<int>a(n, m)  //声明一个n个元素的列表，每个元素都是m
list<int>a(first, last)  //声明一个列表，其元素的初始值来源于由区间所指定的序列中的元素，first和last是迭代器
```

list的基本操作：

```cpp
a.begin()            //  返回指向首元素的随机存取迭代器
a.end()              //  返回指向尾元素的下一个位置的随机存取迭代器
a.push_front(x)      //  向表头插入元素x
a.push_back(x)       //  向表尾插入元素x
```

```
a.pop_back()          //  删除表尾元素
a.pop_front()         //  删除表头元素
a.size()              //  返回表长
a.empty()             //  表为空时，返回真，否则返回假
a.resize(n)           //  改变序列长度，超出的元素将会全部被删除，如果序列
需要扩展（原空间小于n），元素默认值将填满扩展出的空间
a.resize(n, val)      //  改变序列长度，超出的元素将会全部被删除，如果序列
需要扩展（原空间小于n），val将填满扩展出的空间
a.clear()             //  删除容器中的所有元素
a.front()             //  返回首元素
a.back()              //  返回尾元素
a.swap(v)             //  将a与另一个list对象进行交换
a.merge(b)            //  调用结束后b变为空，a中元素包含原来a和b的元素
a.insert(it, val)     //  向迭代器it指向的元素前插入新元素val
a.insert(it, n, val)//  向迭代器it指向的元素前插入n个新元素val
a.insert(it, first, last)
//  将由迭代器first和last所指定的序列[first, last)插入到迭代器it指向的元
素前面
a.erase(it)           //  删除由迭代器it所指向的元素
a.erase(first, last)//  删除由迭代器first和last所指定的序列[first,
last)
a.remove(x)           //  删除了a中所有值为x的元素
a.assign(n, val)      // 将a中的所有元素替换成n个val元素
a.assign(b.begin(), b.end())
//将a变成b
```

string

头文件：string

string是STL的字符串类型，通常用来表示字符串。而在使用string之前，字符串通常是用`char*`表示的。
string和`char*`的区别
string是一个类，`char*`是一个指向字符的指针。
string封装了`char*`，管理这个字符串，是一个`char*`型的容器。也就是说string是一个容器，里面元素的数据类型是`char*`。
string不用考虑内存释放和越界。
string管理`char*`所分配的内存。每一次string的复制，取值都由string类负责维护，不用担心复制越界和取值越界等。 string提供了一系列的字符串操作函数
查找find，拷贝copy，删除erase，替换replace，插入insert.

构造和析构函数:

表达式	效果
string s	生成一个空字符串
string s(str)	copy构造函数，生成一个str的复制品
string s(str,idx)	将string内始于位置idx的部分当作字符串s的初值
string s(str,idx,len)	将string内始于位置idx且长度最多为len的部分当作字符串s的初值
string s(cstr)	以C-string字符串cstr作为字符串s的初值
string s(cstr, len)	以C-string字符串cstr的前len个字符作为字符串s的初值
string s(num，c)	生成一个字符串，包含num个字符c
string s(beg, end)	以区间[beg,end]内所有字符作为字符串s的初值

操作函数:

操作函数	效果
=，assign()	赋以新值
swap()	交换两个字符串的内容
+=, append(),push_back()	添加字符
insert()	插入字符
erase()	删除字符
clear()	移除全部字符
resize()	改变字符数量
replace()	替换字符
+	串联字符串
==，!=，<，<=，>，>=，compare()	比较字符串内容

size(),length()	返回字符数量,等效函数
max_size()	返回字符的最大可能个数
empty()	判断字符串是否为空
capacity()	返回重新分配之前的字符容量
reserve()	保留一定量内存以容纳一定数量的字符
[ ]，at()	存取单一字符
>>，getline()	从stream中读取某值
<<	将某值写入stream
copy()	将内容复制为一个C-string
c_str()	将内容以C-string形式返回
data()	将内容以字符数组形式返回
substr()	返回某个子字符串
begin()，end()	提供正常的迭代器支持
rbegin()，rend()	提供逆向迭代器支持

pair

头文件：utility

STL的utility头文件中描述了一个看上去非常简单的模版类pair，用来表示一个二元组或元素对，并提供了按照字典序对元素对进行大小比较运算符模版函数。 Example，想要定义一个对象表示一个平面坐标点，则可以：

```
pair<double, double> p;
cin >> p.first >> p.second;
```

pair模版类需要两个参数：首元素的数据类型和尾元素的数据类型。pair模版类对象有两个成员：first和second，分别表示首元素和尾元素。 在其中已经定义了pair上的六个比较运算符：<、>、<=、>=、==、!=，其规则是先比较first，first相等时再比较second，这符合大多数应用的逻辑。当然，也可以通过重载这几个运算符来重新指定自己的比较逻辑。 除了直接定义一个pair对象外，如果需要即时生成一个pair对象，也可以调用在其中定义的一个模版函数：make_pair。make_pair需要两个参数，分别为元素对的首元素和尾元素。

map

头文件：map

在STL的头文件中map中定义了模版类map和multimap，用有序二叉树表存储类型为`pair<const Key, T>`的元素对序列。序列中的元素以const Key部分作为标识，map中所有元素的Key值必须是唯一的，multimap则允许有重复的Key值。

可以将map看作是由Key标识元素的元素集合，这类容器也被称为“关联容器”，可以通过一个Key值来快速决定一个元素，因此非常适合于需要按照Key值查找元素的容器。 map模版类需要四个模版参数，第一个是键值类型，第二个是元素类型，第三个是比较算子，第四个是分配器类型。其中键值类型和元素类型是必要的。

定义map对象的代码示例：

```
map<string, int> m;
```

map的基本操作：

```
/*  向map中插入元素  */
m[key] = value; //  [key]操作是map很有特色的操作,如果在map中存在键值为key的元素对, 则返回该元素对的值域部分,否则将会创建一个键值为key的元素对,值域为默认值。所以可以用该操作向map中插入元素对或修改已经存在的元素对的值域部分。
m.insert(make_pair(key, value));    //  也可以直接调用insert方法插入元素对,insert操作会返回一个pair,当map中没有与key相匹配的键值时,其first是指向插入元素对的迭代器,其second为true;若map中已经存在与key相等的键值时,其first是指向该元素对的迭代器,second为false。

/*  查找元素  */
int i = m[key]; //  要注意的是,当与该键值相匹配的元素对不存在时,会创建键值为key（当另一个元素是整形时，m[key]=0）的元素对。
map<string, int>::iterator it = m.find(key);    //  如果map中存在与key相匹配的键值时,find操作将返回指向该元素对的迭代器,否则,返回的迭代器等于map的end()(参见vector中提到的begin()和end()操作)。

/*  删除元素  */
m.erase(key);   //  删除与指定key键值相匹配的元素对,并返回被删除的元素的个数。
m.erase(it);    //  删除由迭代器it所指定的元素对,并返回指向下一个元素对的迭代器。

/*  其他操作  */
m.size();       //  返回元素个数
```

```
m.empty();      //  判断是否为空
m.clear();      //  清空所有元素
```

stack

头文件：stack

stack模版类的定义在stack头文件中。 stack模版类需要两个模版参数，一个是元素类型，另一个是容器类型，但是只有元素类型是必要的，在不指定容器类型时，默认容器的类型为deque。

定义stack对象的示例代码如下:

```
stack<int> s;
stack<string> ss;
```

stack的基本操作有：

```
s.push(x);  //  入栈
s.pop();    //  出栈
s.top();    //  访问栈顶
s.empty();  //  当栈空时，返回true
s.size();   //  访问栈中元素个数
```

queue

头文件：queue

queue模版类的定义在queue头文件中。 queue与stack相似，queue模版类也需要两个模版参数，一个元素类型，一个容器类型，元素类型时必须的，容器类型时可选的，默认为deque类型。

定义queue对象的示例代码必须如下：

```
queue<int> q;
queue<double> qq;
```

queue的基本操作：

```
q.push(x);  //  入队列
q.pop();    //  出队列
q.front();  //  访问队首元素
q.back();   //  访问队尾元素
q.empty();  //  判断队列是否为空
q.size();   //  访问队列中的元素个数
```

set

头文件：set

set是与集合相关的容器，STL为我们提供了set的实现，在编程题中遇见集合问题直接调用是十分方便的。

定义set对象的示例代码如下：

```
set<int> s;
set<double> ss;
```

set的基本操作：

```
s.begin()        //  返回指向第一个元素的迭代器
s.clear()        //  清除所有元素
s.count()        //  返回某个值元素的个数
s.empty()        //  如果集合为空，返回true(真)
s.end()          //  返回指向最后一个元素之后的迭代器，不是最后一个元素
s.equal_range()  //  返回集合中与给定值相等的上下限的两个迭代器
s.erase()        //  删除集合中的元素
s.find()         //  返回一个指向被查找到元素的迭代器
s.get_allocator()   //  返回集合的分配器
s.insert()       //  在集合中插入元素
s.lower_bound()  //  返回指向大于（或等于）某值的第一个元素的迭代器
s.key_comp()     //  返回一个用于元素间值比较的函数
s.max_size()     //  返回集合能容纳的元素的最大限值
s.rbegin()       //  返回指向集合中最后一个元素的反向迭代器
s.rend()         //  返回指向集合中第一个元素的反向迭代器
s.size()         //  集合中元素的数目
s.swap()         //  交换两个集合变量
s.upper_bound()  //  返回大于某个值元素的迭代器
s.value_comp()   //  返回一个用于比较元素间的值的函数
```

multiset

头文件：set

在set头文件中，还定义了另一个非常实用的模版类multiset（多重集合）。多重集合与集合的区别在于集合中不能存在相同元素，而多重集合中可以存在。

定义multiset对象的示例代码如下：

```
multiset<int> s;
multiset<double> ss;
```

multiset和set的基本操作相似，需要注意的是，集合的count()能返回0（无）或者1（有），而多重集合是有多少个返回多少个。

bitset

头文件：bitset

在 STLSTL 的头文件中 bitset中定义了模版类 bitsetbitset，用来方便地管理一系列的 bitbit 位的类。bitsetbitset 除了可以访问指定下标的 bitbit 位以外，还可以把它们作为一个整数来进行某些统计。

bitsetbitset 模板类需要一个模版参数，用来明确指定含有多少位。

定义 bitsetbitset 对象的示例代码：

```
const int MAXN = 32;
bitset<MAXN> bt;             //  bt 包括 MAXN 位，下标 0 ~ MAXN - 1，默认初始化为 0
bitset<MAXN> bt1(0xf);       //  0xf 表示十六进制数 f，对应二进制 1111，将 bt1 低 4 位初始化为 1
bitset<MAXN> bt2(012);       //  012 表示八进制数 12，对应二进制 1010，即将 bt2 低 4 位初始化为 1010
bitset<MAXN> bt3("1010");    //  将 bt3 低 4 位初始化为 1010
bitset<MAXN> bt4(s, pos, n);//  将 01 字符串 s 的 pos 位开始的 n 位初始化 bt4
```

bitsetbitset 基本操作：

```
bt.any()         //  bt 中是否存在置为 1 的二进制位?
bt.none()        //  bt 中不存在置为 1 的二进制位吗?
bt.count()       //  bt 中置为 1 的二进制位的个数
bt.size()        //  bt 中二进制位的个数
bt[pos]          //  访问 bt 中在 pos 处的二进制位
```

```
bt.test(pos)     //  bt 中在 pos 处的二进制位是否为 1
bt.set()         //  把 bt 中所有二进制位都置为 1
bt.set(pos)      //  把 bt 中在 pos 处的二进制位置为 1
bt.reset()       //  把 bt 中所有二进制位都置为 0
bt.reset(pos)    //  把 bt 中在pos处的二进制位置为0
bt.flip()        //  把 bt 中所有二进制位逐位取反
bt.flip(pos)     //  把 bt 中在 pos 处的二进制位取反
bt[pos].flip()   //  同上
bt.to_ulong()    //  用 bt 中同样的二进制位返回一个 unsigned long 值
os << bt         //  把 bt 中的位集输出到 os 流
```