简介

STL，全称 Standard Template Library，译为标准模板库，由惠普公司开发。它无需额外安装，并且使用起来十分方便，它的出现增强了代码的复用性和可读性，减少了代码手的工作量，让 c++ 得以在算竞中大放异彩。

STL 有六大组件——容器、算法、迭代器、仿函数、适配器、空间配置器。前三个是 STL 广义上的分类，算法竞赛也只需要重点掌握前三个组件。

接下来，我们一一介绍这三大组件。

前置知识

成员函数：在面向对象中，成员函数也被称为方法。类的成员函数是指那些把定义和原型写在类定义内部的函数，就像类定义中的其他变量一样。类成员函数是类的一个成员，它可以操作类的任意对象，可以访问对象中的所有成员。

迭代器

定义

迭代器，是一个指向容器内元素的数据类型。可以简单的理解为**指针**。我们可以通过操作迭代器来遍历/查找容器内特定的元素。

不同的容器，迭代器的使用方式也不相同，具体的内容我们会在介绍容器时一同介绍。

分类及定义方式

- 正向迭代器：

```
<容器类型>::iterator 迭代器名称
```

- 反向迭代器

```
<容器类型>::reverse_iterator 迭代器名称
```

- 常量迭代器

```
<容器类型>::const_iterator 迭代器名称
```

- 常量反向迭代器

```
<容器类型>::const_reverse_iterator 迭代器名称
```

定义迭代器的格式，记住就好了。

容器

容器，是各种数据结构，用来存放数据。

接下来介绍 STL 中常见容器

栈

栈包含在头文件`#include<stack>`

关于栈这个数据结构就不多讲了，我们直接来看如何操作

```
#include<stack> //要包含这个头文件

stack<int> s;  //定义一个名字为 s，存放int类型数据的栈

//stack<数据类型> 变量名    <- 这是命名规则

s.top(); //取出栈顶元素

s.pop(); //删除栈顶元素

s.size(); //返回容器内元素个数（容器大小）

s.push(x); //向栈顶加入新的元素 x

s.empty(); //判断栈是否为空，空则返回1，否则返回0
```

队列

队列，又称**先进先出表**（FIFO）。相当于排队买奶茶，谁先下单谁可以先拿到一样，最先进入容器的元素，也必定是最先取出来的。

队列又分三种：普通队列，优先队列和双端队列。这里我们只讨论前面两种。

普通队列与优先队列均包含在头文件`#include<queue>`中

- 普通队列

```
#include<queue> //记得要包含头文件哦~

queue<int> q; //定义一个普通队列，名字为 q，存放类型为 int 的数据

// queue<数据类型> 变量名 <-定义一个队列的格式

q.front(); //取出队首元素

q.pop(); //清除队首元素

q.size(); //返回当前队列内元素个数（容器大小）

q.push(x); //向队列加入新的元素 x

q.empty(); //判断队列是否为空，空则返回1，否则返回0

q.back(); //返回队列中最后一个元素
```

- 优先队列

优先队列和普通队列区别较大。优先队列，是一个可以**自动排序**的队列。也就是说，先进去的不一定先出来，而是通过比较大小来决定出队顺序的。

优先队列和堆是类似的，当然，你也可以把它理解为一个堆。

优先队列默认以`vector`的作为底层数据结构来实现，当然，也可以以`list`作为底层数据结构实现。

下面是优先队列的声明方式。

```
// priority_queue<数据类型> 变量名
// 认真研究过上面也应该明白是什么意思了

priority_queue<int> q; <=> priority_queue<int,vector<int>,less<int> > q;

// 上面两种声明方法是等价的。在定义时若不指明类型，默认为大根堆（降序排序）

priority_queue<int,vector<int>,greater<int> > q;

//这个是小根堆（升序排序）的声明方式
```

优先队列的成员函数：

```
priority_queue<int> q; //这里直接声明成小根堆了

q.top(); //取出队首元素，和普通队列不太一样

q.pop(); //清除队首元素

q.size(); //返回当前队列内元素个数（容器大小）

q.push(x); //向队列加入新的元素 x

q.empty(); //判断队列是否为空，空则返回1，否则返回0
```

优先队列不仅可以对普通的数据类型排序，还可以对你自定义的数据类型排序，只不过需要自己规定排序方式，方法如下。

```
struct cmp// 方法一：写一个比较函数，定义比较顺序。
{
    bool operator () (int x,int y)
   {
        return x<y;
   }
}

priority_queue<int,vector<int>,cmp> q;//和上面声明小根堆的方法是等价的，这里只是展示写比较函数的方法。



struct node{ //方法二：重载小于符号（可以理解为重新定义小于符号）
    int x,y,sum;
    bool operator<(const node &xx) const{
        return xx.sum<sum;  //对于每一个队内的node类型元素，谁的 sum 小谁在前面
    }
}t,tt;

priority_queue<node> q;
```

优先队列的常用成员函数：

```
priority_queue<int> q; //这里直接声明成小根堆了

q.top(); //取出队首元素，和普通队列不太一样

q.pop(); //清除队首元素

q.size(); //返回当前队列内元素个数（容器大小）

q.push(x); //向队列加入新的元素x

q.empty(); //判断队列是否为空，空则返回1，否则返回0
```

优先队列的增改复杂度为 O(logn)，删查为 O(1)

双端队列

我们知道，队列和优先队列都只能从后端插入，从前端取出。

你是不是觉得这样非常的不爽？你是不是想要一个两边都能插入和删除的队列？

今天，它来了。

它叫做**双端队列**，包含在头文件 `#include<deque>`。

它的作用非常的大(例如，他默认为 stack 和 queue 的底层容器；也可以在单调队列中发挥大作用)

双端队列有很多优点，例如：

- 支持在头部和尾部插入元素
- 支持随机下标访问队列中的元素
- 支持在中间插入一个新元素

缺点：常数大

它的成员函数：

```
#include<deque> //使用时要包含这个头文件

deque<int> q;

// deque<数据类型> 变量名

q.front(); //取队首

q.back(); //取队尾

q.push_front(x); //在队首插入新元素x

q.push_back(x); //在队尾插入新元素x

q.pop_front(); //删除队首元素

q.pop_back(); //删除队尾元素

q.empty(); //判断队列是否为空
```

```
q.size(); //返回队列内元素个数

q.insert();
//insert（插入的位置起点，插入终点，插入的元素）
//insert（插入位置，插入元素）这个只在指定位置插入元素。

q.erase();//清除元素，写法与insert类似。

q.clear(); //一键清空

q.at(1) //访问位于 1 位置的元素，如果越界，则会抛出 out of range 异常

q.swap(p); 和同样数据类型的 p 队列互换元素
```

vector

vector，译为向量，但我更喜欢叫动态数组。

vector 用处十分的大，其不仅**支持下标访问**，而且还能够**动态开辟内存**，这意味着我们不仅可以像正常数组一样使用它，还可以节省较多的空间。同时，它还内置了很多成员函数，例如 size()，能够帮助我们无需开辟新变量记录即可快速获取容器内元素个数。在你有一些静态数组难以满足的需求时，不妨把静态数组换成 vector 试试。

下面是 vector 的声明：

```
#include<vector>  //使用时要包含这个头文件哦~

//vector<数据类型> 变量名(数组大小，初值)

//括号内内容（指数组大小和初值）非必要，可写可不写，看需求

vector<int> a(10); //定义了10个整型元素的向量

vector<int> a(10,1); //定义了10个整型元素的向量,且给出每个元素的初值为1

vector<int> a(b); //用b向量来创建a向量，整体复制性赋值

vector<int> a(b.begin(),b.begin+3); //定义了a值为b中第0个到第2个（共3个）元素

int b[7]={1,2,3,4,5,9,8};

vector<int> a(b,b+7); //从数组中获得初值
```

vector 的成员函数

```
a.assign() //a.assign(左边界l,右边界r) 将[l,r)区间内数据赋值给 a
           //a.assign(元素个数,元素值) 将几个元素赋值给 a

a.front(); //返回第一个元素

a.back();  //返回最后一个元素

a.empty(); //返回数组是否为空，是返回 1 ，否则返回 0
```

```
a.insert();  //a.insert(插入位置，插入元素) 在指定位置插入一个元素
             //a.insert(插入位置，插入个数，插入元素) //在指定位置插入几个元素
             //a.insert(插入位置，左区间l,右边界r) //在指定位置插入[l,r)区间内的元素

a.push_back(x); //在尾部插入x

a.pop_back(); //删除尾部元素

a.size(); //返回 a 中元素个数

a.swap(b); //a,b交换所有元素
```

string

这个东西叫字符串，使用时须包含头文件 `#include<string>`

这个可比字符数组（char[100]）好用多了！

string 不仅是 STL，它同时也是一种**数据类型**。在 c 语言中，字符串需要借助字符数组才能输入，而且十分复杂，但在 c++ 中，你只需要正确声明，就可以直接使用 cin 读取字符串了

string 有着如下的优点：

- 它是动态扩展的（与 vector 一样，允许创建时不指定大小）
- 它允许通过下标访问字符串内的元素
- 两个字符串可以直接比较大小（按位比较）
- 可以使用成员函数 find 查找需要的字符/字符串。
- ...

总之就是好用。

- string 的成员函数

```
#include<string> // 记得不能少了这个哦~

string s; //这样就是定义了一个名为 s 的字符串了。

s+=append() / s. push_back(); //在尾部添加字符

s.insert(); //插入字符

s.erase(); //删除字符

s.clear(); //删除全部字符

s.replace(); //替换字符

s+=ss //串联字符串s,ss

s.size() /  s.length()    //返回字符数量

s.empty()    //判断字符串是否为空

s[] //存、取单一字符(下标访问)
```

```
s.copy() //将某值赋值为一个C_string

c_str() //将内容以C_string返回

s.substr() //返回某个子字符串

s.find(查找的起始位置,查找的终止位置,查找的字符/字符串)  //查找指定位置的字符/字符串

s.begin(); //正向迭代器

s.end();

s.rbegin(); //逆向迭代器

s.rend();

 ==,!=,<,<=,>,>=,compare(s1,s2)    //比较字符串

such as  s1 >= s2
```

list

STL 还有**双向链表!**

使用时须包含头文件 `#include<list>`

和手写链表一样，它只支持顺序遍历，不支持下标访问。

但是你不需要手写链表了，这不很舒服?

list 的成员函数:

```
#include<list> //记得写哦~

list<int> lis;  //声明链表

lis.insert(插入位置，插入元素); //在指定的位置前一个位置插入指定元素。

lis.erase();//清除元素，写法与insert类似。

lis.front(); //取首

lis.back(); //取尾

lis.push_front(); //在链表头部首插入新元素

lis.push_back(); //在链表尾部插入新元素

lis.pop_front(); //删除队首元素

lis.pop_back(); //删除队尾元素

lis.empty(); //判断队列是否为空
```

```
lis.size(); //返回队列内元素个数

lis.clear(); //一键清空

lis.begin(); // 返回首位置的迭代器

lis.end(); //返回尾位置的下一位的迭代器

lis.rbegin(); //返回反向的首位置（也就是最后一位）的迭代器

lis.rend(); //返回反向的尾位置（也就是第一位）的下一位（第一位的前面一位）的迭代器。
```

map

使用时须包含头文件 `#include<map>`

这个不叫地图！！！

用于存储一对映射关系 key-value，

这是一个 STL 的容器，其能够建立起 key-value 两个值之间的一个对应关系，其中，key与value 可以是你想要的任何类型。内部元素会按照 key 的大小进行自动排序。

其底层是一颗红黑树。它的功能类似于 hash。当然，你也可以把它当做一个哈希表来用。

但是，当数据大小 >500000 时建议不要使用，因为它排序也要时间。

map 的增删改查时间均为O(logn)

接下来讲讲怎么用。

- 创建一个 map

```
map<string,int> score; //建立map

//插入元素，建立对应关系

1、 score.insert(pair<string,int>("Li hua",10);

2、 score.insert(map<string,int>::value_type("Li hua",10);

3、 score["Li hua"]=10;
```

需要注意的是，**当关键字已经存在时，insert 操作是无效的**，而数组插入方式则会直接替换原来关键字所关联的值。

- map 的查找

```
if(score.find("Li hua");!=score.end()) cout<<iter->second<<endl;
else cout<<"Not find"<<endl;

// 这里也可以用迭代器。
```

find 函数会返回查找的元素所在的位置，找不到就返回尾指针 `score.end()`。借此可判断该关键字是否存在。

- 其他成员函数：

```
score.size();   //返回map的大小（元素个数）

swap(); //交换两个 map

score.clear()  //删除所有元素

score.count()  //返回指定元素（key）出现的次数。

score.erase()  //删除元素
```

- 迭代器使用方法：

```
map<int,int> mp;

mp.begin(); //返回一个 mp 的起始位置的迭代器位置
mp.end(); //返回一个 mp 的终止位置的下一位的迭代器位置

mp.rbegin(); // 返回一个 mp 的反向起始位置的迭代器
mp.rend(); //返回一个 mp 的反向终止位置的下一位的迭代器位置

map<int,int>::iterator it; // 迭代器声明方式

it -> first //返回 key（第一元素）
it -> second //返回 value （第二元素）

--------------------------------------------------------------------------

map<int,int> mp;

map<int,int>::iterator it;  //(声明一个名为 it 的 map 迭代器)

mp[2]=1; mp[3]=1; mp[4]=1;

for(it = mp.begin();it!=mp.end();++it) {
  cout << it -> first << " " << it -> second << "\n";
}

//我们知道，map 存储的是两个元素的映射关系,key 是第一个元素，value 是第二个元素
//it -> first 指向的是 key，it -> second 指向的是 value
```

unordered_map

unordered_map 是一个将 key 和 value 关联起来的容器，它可以高效的根据单个 key 值查找对应的 value。key 值应该是唯一的，key 和 value 的数据类型可以不相同。

unordered_map 的底层数据结构是哈希表，存储元素时是没有顺序的，同时查找的效率非常高（因为是哈希表），查询一次的时间为 O(1)

unordered_map 查询单个 key 的时候效率比 map 高，但是要查询某一范围内的 key 值时比 map 效率低。
可以使用 [] 操作符来访问key值对应的 value 值。

unordered_map 的增删改查时间均为 O(1)，但其本身常数较大，并且容易被卡，建议谨慎使用。

set/multiset

使用前须包含头文件`#include<set>`

set 和 map 一样，底层结构都是一颗红黑树。

set 的特点：

- 内部不会出现重复元素
- 内部元素是有序的

至于 **multiset**？其与 set 的区别仅在于**内部可以出现重复元素**，成员函数也相差无几，在此省略对 muitiset 的详细讲解

set 的增删改查复杂度均为 O(logn)

set 的成员函数如下：

```
set<int> s;  //这是 set 的声明方式
multiset<int> s; //这是 multiset 的声明方式

s.begin();  // 返回指向第一个元素的迭代器

s.end();   // 返回指向迭代器的最末尾处（即最后一个元素的下一个位置）

s.clear();  // 清除所有元素

s.count();  // 返回某个值元素的个数。在 set 中，其返回值非 0 即 1；在 multiset 中，其值可以大于 1

s.empty(); // 如果集合为空，返回true

s.erase() //删除指定位置的元素

s.insert(pos,x) //在指定位置插入元素

s.size() //返回容器中的元素个数

s.begin(); //返回一个 s 的起始位置的迭代器位置

s.end(); //返回一个 s 的终止位置的下一位的迭代器位置

s.rbegin(); // 返回一个 s 的反向起始位置的迭代器

s.rend(); //返回一个 s 的反向终止位置的下一位的迭代器位置

s.lower_bound(x); //在 s 中查找 x 的值。是内置的二分函数
```

迭代器使用方法

```
set<int> s;

set<int>::iterator it; // 迭代器声明方式

for(it = s.begin();it!=s.end();++it) {
  cout << *it << endl;  //获取 set 内元素
}
```

算法

STL 中还有很多的算法，这些算法拿来即可使用，大大简化了 acmer 的代码长度。

C++STL 中的内置算法主要在头文件 `<algorithm>、<functional>、<numeric>` 中。

- `<algorithm>` 是所有STL头文件中最大的一个，也是包含算法最多，最常用的一个头文件，其中包含比较、交换、查找、遍历、复制、修改等算法
- `<numeric>` 体积很小，只包括几个在序列上面进行简单数学运算的模板函数
- `<functional>` 定义了一些模板类，用以声明函数对象(仿函数)

我们主要介绍 库的常用算法，这也是算竞中用到的比较多的库。

max/min

```
max(a,b); //返回a,b中较大的数
min(a,b); //返回a,b中较小的数
```

注意事项：

- 它们一次只能比较两个元素，若有多个元素同时需要比较，你可以像这样嵌套 `max(a,max(b,c))`
- 比较的元素必须是同类型的，int 只能和 int 比，不能和 long long 比较

min_element() 和 max_element()

这两个函数也是求最大和最小值的，使用规则就是min_element(start,end)
这里传入的是起始地址（或迭代器）和结束地址（或迭代器）返回的也是地址、

```
vector<int> numbers = { 1,3,4,5,6,7,9 };
cout << *min_element(numbers.begin(), numbers.end()) << "\n"; //通过解引用返回1
cout << *max_element(numbers.begin(), numbers.end()) << "\n";//返回9
```

nth_element()

nth_element(start , k ,end)
这个函数的作用就是进行部分排序，返回值为void，传入的是3个地址或迭代器，排序之后， k 的处于正确的位置，其他元素可能是任意的，也就是不一定有序，但是k前面都是比k小的，后面都是比k大的

```
vector<int> numbers = { 4,3,1,10,5,6,9,7,2 };
nth_element(numbers.begin(), numbers.begin() + 4, numbers.end());
for (int i = 0; i < numbers.size(); i++) {
    cout << numbers[i] << " ";
}
//3 2 1 4 5 6 9 7 10
```

find()

作用：查找指定范围内是否存在特定元素

函数原型：`find(iretator begin,iterator end,val)`

解读：`find(查找起点，查找终点，查找值)`

找到会返回指定位置的迭代器，找不到则返回结束的迭代器。

注意事项：如果待查找元素其是自定义数据类型，如类的对象、结构体类型的元素,查找时需要重载 `= =` 运算符

时间复杂度：O(n)

binary_find()

作用：查找特定范围的某个值，不过使用的是二分查找法

函数原型：`bool binary_search(iterator beg,iterator end,value);`

解读：binary_search(起点，终点，查找值)，找到返回 1，否则返回 0

注意事项：查找序列必须是有序的

时间复杂度:O(logn)

count

作用：统计指定元素在指定范围内出现次数

函数原型:`count(iterator begin,iterator end,value);`

解读：count(起点，终点，指定元素)，找到 x 个就返回 x

时间复杂度：O(n)

sort

作用：对容器内部元素进行排序。

函数原型：`sort(iterator beg,iterator end,_Pred);`

解读： sort(起点,终点,排序方式); 排序方式可以不填，不填写则默认为升序排序。可以自定义排序方法。

sort 使用快速排序，时间复杂度为 O(nlogn)

- 拓展—— stable_sort()，用法与 sort 一致，区别在于权值相同的元素，stable_sort 不会改变他们的次序。

random_suhffle

作用：打乱指定范围内的元素顺序

函数原型：`random_shuffle(iterator beg,iterator end);`

解读：random_shuffle(起点,终点);

注意事项：需要注意重置随机种子。

（在一些序列越乱算法越优，或者需要多次打乱序列的情况下，ramdom_shuffle 是一个不错的选择）

reverse

作用：将容器内元素进行反转

函数原型：`reverse(iterator beg,iterator end);`

解读：`reverse(起点，终点)`

超级好用，不需要一个个倒过来了。

时间复杂度：O(n)

swap

作用：交换两个**相同类型**的元素或容器内部元素。

函数原型：`swap(container c1,container c2);`

解读：交换元素/容器 c1 和 元素/容器 c2

replace

作用：将容器内指定范围的旧元素替换为新元素

函数原型：`replace(iterator beg,iterator end,oldvalue,newvalue);`

解读：replace(起点，终点，被替换值，替换后的值);

lower_bound/upper_bound

函数原型（以 lower_bound 为例）：`lower_bound(iterator beg,iterator end,Compare comp)`

函数解析：`lower_bound(起点，终点，比较方式（可以不写，不写默认为升序，结构体排序必须填写这一值）)`

1. 作用

lower_bound和upper_bound都是C++的STL库中的函数，作用差不多，lower_bound所返回的是第一个大于或等于目标元素的元素地址，而upper_bound则是返回第一个大于目标元素的元素地址。

2.使用条件

用lower_bound/upper_bound进行二分查找时必须保证查找区间为升序序列！

3.用法

lower_bound和upper_bound的用法与sort类似：

```
lower_bound(起始位置first,结束位置last,目标元素val);
upper_bound(起始位置first,结束位置last,目标元素val);
 //lower_bound/upperbound的返回值是一个地址值，若要得到目标元素的下标，直接减去数组首地址的值即可
```

根据C++STL的尿性，同理可得，在lower_bound和upper_bound中我们同样可以添加cmp来进行自定义比较：

```
lower_bound(a,a+n,2,cmp);
upper_bound(a,a+n,2,cmp);
```

显然，我们可以通过自定义比较函数来实现降序序列的查找：

```
bool cmp(const int& a,const int& b){
    return a>b;
}
lower_bound(a,a+n,val,cmp);
upper_bound(a,a+n,val,cmp);
```

或者你可以更简单一些：

```
lower_bound(a,a+n,val,greater<int>());
upper_bound(a,a+n,val,greater<int>());
```

最后，如果函数并没有找到目标元素，则会返回last的地址，且last的地址是越界的。

unique

unique用于去除相邻两个重复两个元素 ， 返回的是但并不是真正意义上的删除，可以理解为挪到了后边，所以就需要配合erase去删除

```
vector<int> v = { 1,1,2,3,3,4,5,6 };
auto it = unique(v.begin(), v.end());
v.erase(it, v.end());
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
}
```

memset()

这是一个设置内存块值的函数，包含在< cstring >头文件中

memset(要设置内存块值的指针，要设置的值，要设置的字节数)

```
int a[5];
memset(a,1,sizeof(a));
```

islower()和isupper()

这两个函数就是检查判断一个字符是否为小写字母和大写字母，返回值也就是bool类型，lower为小写，upper为大写

```
char ch = 'A';
if (islower(ch)) cout << "是小写字母" << "\n";
else  cout << "是大写字母" << "\n";
if(isupper(ch)) cout << "是大写字母" << "\n";
else  cout << "是小写字母" << "\n";
```

tolower()和toupper()

toupper和tolower就是将一个小写字母或一个大写字母转化为对应的大写字母或小写字母，返回值是一个字符类型，可以定义一个字符变量来接收

```
 char ch = 'A';
 char c = tolower(ch);
 cout << c << "\n";
```

next_permutation()和prev_permutation()

`next_permutation()` 函数用于生成一个序列的字典序下一个排列。如果当前排列已经是最大的字典序排列，函数会返回 `false` 并生成最小的字典序排列。

`prev_permutation()` 函数用于生成一个序列的字典序上一个排列。如果当前排列已经是最小的字典序排列，函数会返回 `false` 并生成最大的字典序排列。

```
vector<int> v = { 1,2,3 };
while (next_permutation(v.begin(), v.end())) {
    for (int i = 0; i < v.size(); i++) {
        cout <<v[i]<< " ";
    }
    cout << "\n";
}
```

后话

- **谨慎使用迭代器**
- **STL 中涉及区间的，都是左闭右开的（请务必牢记）**
- **当你使用 erase 之后，指向原位置的指针就失效了（俗称野指针），请注意不要操作它**