第6章 C++标准数据库STL介绍

---writerd

by Tang

第6章 C++标准数据库STL介绍

6.1 vector的常用解法详解

vector，翻译为向量，但在C++中实际上是一种变长数组(长度可以根据需要来自动改变的数组)

```
#include <vector>
using namespace std;
```

1、vector的定义

```
vector<typename> name;
```

(1) 一维数组

```
vector<int> nums(n, value)
//将nums设置为大小为n的数组，并且数组中的值都为value
```

(2) 二维数组

```
vector<vector<int>> matrix(m, vector<int> (n, value))
//将nums设置为一个m * n的二维数组，数组中的值都初始化为value
```

利用数组来初始化vector，vector不能像nums一样初始化，`vector nums(10) = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };会产生错误

但是可以用数组来进行初始化

```
int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
vector<int> nums(arr, arr+10);
```

2、vector内容器的访问

(1) 通过下标进行访问

nums[index]

(2) 通过迭代器进行访问

迭代器(iterator)类似于一种指针的东西

迭代器和指针的区别：https://www.cnblogs.com/depend-wind/articles/10101643.html

```
vector<typename> :: iterator it;
//等价于auto it，这里auto -> vector<typename> :: interator

for (auto it = nums.begin(); it != nums.end(); it ++) {
    cout << *it;    // nums[i]等价于*(nums.begin() + i)
}
```

C++11的特性：**基于范围遍历**

```
for (auto value : nums) printf("%d", value);

//nums.begin(), nums.end()分别表示数组的首地址和数组尾元素地址的下一个地址
```

3、vector常用实例函数解析

(1) push_back()

push_back(x)，在vector之后添加元素x，时间复杂度O(1)

(2) pop_back()

pop_back()删除vector的尾元素，时间复杂度O(1)

(3) front()、back()

front()访问vector的首元素，时间复杂度O(1)

back()访问vector的尾元素，时间复杂度O(1)

通过push_back()、 pop_back()、back()，可以将vector当做stack来使用

(4) size()、resize()

size()获取vector中元素的个数

vector可以开始不定义大小，之后resize()更改vector的大小

(5) clear()

clear()，将vector中的所有元素清空,时间复杂度O(N)

(6) insert()

insert(it, x)用来在迭代器it处插入元素x， 时间复杂度O(N)

在下标为index的位置上插入元素x，`x nums.insert(nums.begin() + index, x);`

(7) earse()

1)、删除单个元素

erase(it)删除迭代器it处的元素

删除下标为index的位置上的元素，`nums.erase(nums.begin() + index);`

2)、删除一个区间的所有元素

erase(first, last)删除[first, last)之内的所有元素

vector支持比较运算(字典序)

4、vector的常见用途

1)、作为数组使用

2)、用邻接表存储图

6.2 set常用解法详解

set翻译为集合，是一个内部自动排序且不含重复元素的容器

```
#include <set>
using namespace std;
```

1、set的定义

```
set<typename> name;
```

2、set容器内元素的访问

set只能通过迭代器进行访问

只有vector和string这两个STL容器支持*(it + i)的访问方式

```
set<typename> ::iterator it; //等价于-> auto it

for (auto it = st.begin(); it != st.end(); it++){
    cout << *it << endl;
}
```

3、set常用函数实例解析

(1) insert()

insert(x)可以将x插入到set容器内，并自动递增排序和去重，时间复杂度O(log N)

(2) find()

```
find(value) //返回set中值为value的迭代器
auto it = st.find(value)
st.find(value) != st.end() //则说明st中存在value的元素,如不存在则返回st.end()
```

(3) erase()

1) 删除单个元素

erase(it)删除迭代器it处的元素，时间复杂度O(1)

erase(value)，value为需要删除的元素值，时间复杂度为O(logn)

2) 删除一个区间的所有元素

erase(first, last)删除[first, last)之内的所有元素，时间复杂度O(last-first)

set内部使用红黑树来实现，erase的时间复杂度为O(logn)

(4) size()

size()用来获得set内元素的个数,时间复杂度O(1)

(5) clear()

clear()用来清空set内的所有元素,时间复杂度O(1)

(6) lower_bound()、upper()_bound()

lower_bound(x)，返回>=x的第一个元素的迭代器
upper_bound(x)，返回>x的第一个元素的迭代器

4、 set的常见用途

set的主要作用是自动排序并去重
使用vector来实现去重和自动排序:

```
sort(nums.begin(), nums.end())
nums.earse(unique(nums.begin(), nums.end()), nums.end())
```

1)元素不唯一

使用multiset

2)去重，但是不排序

使用unordered_set #inclue <unordered_set>

6.3 string的常用解法详解

在C语言中，使用char[]数组来存储字符串，但操作起来比较麻烦
C++中的STL加入了string类

```
#include <string>   注意和#inlucde <cstring>不同
using namespace std;
```

1、string的定义

```
string str;
string str = "abcdefg";
```

2、string中内容的访问

(1)、通过下标进行访问

```
for (int i = 0; i < str.size(); i++)
    cout << str[i] << " ";

for (char ch : str) cout << ch << " ";  //基于范围的访问
```

(2)、通过迭代器访问

```
for (auto it = str.begin(); it != str.end(); it++)
    cout << *it << " ";
```

3、string常用函数实例解析

(1)、operator +=

str = str1 + str2可以直接将两个字符串进行拼接

(2)、compare operator

将两个字符串进行比较> < == >= <= !=
比较的规则是字典序

(3)、length() size()

length()/size()返回字符串的长度

(4)、insert()

字符串的插入

1)、insert(pos, str)

在pos号位置上插入字符串str

2)、insert(it, it2, it 3)

it为待插入的位置,将字符串[it2, it3)中的内容插入到str中

(5)、earse()

删除字符串中的元素

1)、删除单个元素

srt.erase(it)

2)、删除一个区间内的所有元素

```
str.earse(first, last)
//其中first为需要删除区间的起始迭代器，last为需要删除的区间的末尾迭代器的下一个地址
str.earse(pos, length)
//删除str中第pos位置开始，长度为length的元素
```

(6)、clear()

用于清空string中的数据

(7)、substr()

返回子字符串

```
str.substr(pos)
//返回pos位置开始到str末尾结束的字符串
str.substr(pos, length)
//返回pos位置开始，长度为length的字符串
```

(8)、string :: npos

```
string :: npos  用以作为find函数失配时的返回值
str.find(s) != string :: npos说明在str中可以找到s的字符串
```

(9)、find()

```
str.find(s)
//当s是str的字符串时，返回其在str中第一次出现的位置，否则返回string :: npos
str.find(s, pos)
//从第pos号开始匹配s
```

(10)、replace()

```
str.replace(pos, len, s)
//把str的第pos位置开始长度为len的字符串替换为s
sre.replace(it1, it2, s)
//把str的迭代器[it1, it2)范围内的子串替换为s
```

6.4 map的常用用法详解

map，翻译为映射，也是常用的STL容器。
map可以将任何基本类型(包括STL容器)映射到任何基本类型(包括STL容器)

（有时候用unordered_map不可以实现，但却用map可以，例如将二维坐标映射map<pair<int, int>>mp可以，但unordered_map会报错）

```
#include <map>
using namespace std;
```

1、map的定义

```
map<typename1, typename2> mp;
     键的类型    值的类型
map<string, int> mp;
map<set<int>, int> mp;
```

2、map容器内元素的访问

(1)、通过下标进行访问

mp[key] = value **map中的键是唯一的**

(2)、通过迭代器访问

```
auto it = mp.begin()
    //访问键it->first
    //访问值it->second
```

map会按照键的大小，从小到大自动排序

3、map常用函数实例解析

(1)、find()

find(key)返回键为key的映射的迭代器，时间复杂度O(log N)，其中N为map中元素的个数

(2)、earse()

1)、删除单个元素

```
mp.erase(it)    //it为需要删除元素的迭代器
mp.erase(key)   //key为欲删除的键
```

2)、删除一个区间内的所有元素

```
mp.erase(first, last)   //删除[fiest, last)区间内的所有元素
```

(3)、size()

size()获取mp中映射到个数

(4)、clear()

clear()用来清空map中的所有元素，时间复杂度O(N)，其中N为map中元素的个数

4、map的常见用途

1)、建立char、string和int之间的映射，使用map可以减少代码量

2)、需要判断大整数或者其他类型的数据是否存在的题目，可以吧map当bool数组只用

3)、字符串和字符串之间的映射

map中的键和值是唯一的，而如果一个键需要对应多个值，就只能使用multimap
不对map中的键进行排序，可以使用unordered_map来加快程序的运行**

5、map和unordered_map的异同

C++中unordered_map的用法详解

（1）相同：

map和unordered_map都是

6.5 queue的常用解法详解

queue翻译为队列，在STL中则是实现了一个先进先出的容器

1、queue的定义

```
#include<queue>
using namespace std;

queue<typename> name;
```

2、queue容器内元素的访问

由于queue是一种先进先出的限制性数据结构，
因此在queue中只能通过front来访问队首元素，back来访问队尾元素

3、queue常用函数解析

(1)push()

push(x)将x入队，时间复杂度O(1)

(2)front()、back()

front()获得队首元素，back()获取队尾元素，时间复杂度O(1)

(3)pop()

令队首元素出队，时间复杂度O(1)

(4)empty()

empty检测queue是否为空，为空返回true，不空返回false，时间复杂度O(1)

(5)size()

size()返回queue内元素的个数，时间复杂度O(1)

4、queue的常见用途

树的层次遍历、图的广度优先遍历都会用到queue

双端队列deque

```
#include <deque>
using namespace std;
```

双端队列可以在对头队尾两端进行插入和删除

1、push_back()、push_front()

push_back()、push_front()分别在队尾、队头进行插入

2、pop_back()、pop_front()

pop_back()、pop_front()分别将队尾元素和队头元素弹出

单调队列

Acwing 窗口内的最小值和最大值

1 3 -1 -3 5 3 6 7

(1)使用普通的队列

让队列的大小为k，每扫描到一个数就将这个数加入到队列中
再在队列中扫描一遍，获得最大值和最小值，算法的时间复杂度O(n * k)

(2)使用单调队列

首先解决最小值

对1 3 -1而言，1、3都要比-1小并且在-1前面，也就是说窗口内的1、3永远不会成为最小值，所以可以将1、3删除只剩下-1。对-3而言，它比窗口内的-1要小，所以-1不会成为最小值，-1可以删除....

最后删除前面这些>nums[i]的数之后，整个窗口内的数成一个递增有序的状态

始终维持队列中的元素递增(可以存在重复元素)

在这里需要注意的是，当nums[i] < nums[q.back()]时，我们**应该将队尾元素出队**，所以使用的是**双端队列**

当遍历到某个元素时，我们就可以用过单调队列来寻找它的最小值(单调队列的队头元素)

```
#include <iostream>
#include <deque>
using namespace std;

deque<int> dq;
const int N = 1000010;
int nums[N];

int main(){
    int n, k;
    cin>>n>>k;
    for (int i = 0; i < n; i++){
        cin>>nums[i];
    }
    for (int i = 0; i < n; i++){
        if (dq.size() && i-dq.front()+1 > k)  dq.pop_front(); //只保留i和i之前的k个元素
        while(dq.size() && nums[dq.back()] > nums[i]){//维护一个单调递增的双向队列
            dq.pop_back();
        }
        dq.push_back(i);
        if (i >= k-1)   cout<<nums[dq.front()]<<" ";
    }
    cout<<endl;

    dq.clear();
    for (int i = 0; i < n; i++){
        if (dq.size() && i-dq.front()+1 > k)  dq.pop_front(); //只保留i和i之前的k个元素
        while(dq.size() && nums[dq.back()] < nums[i]){//维护一个单调递减的双向队列
            dq.pop_back();
        }
        dq.push_back(i);
        if (i >= k-1)   cout<<nums[dq.front()]<<" ";
    }
    return 0;
}
```

6.6 priority_queue的常用解法详解

priority_queue又称为优先队列，其底层是用堆(heap)来实现的
在优先队列中，队首元素是当前队列中优先级最高的一个

1、priority_queue的定义

```
#include <queue>
using namespace std;

priority_queue<typename> name;
```

2、priority_queue容器内元素的访问

和队列不一样，**优先队列没有front()、back()函数，**
只能通过top()函数来访问队首元素(堆顶元素)，也就是优先级最高的元素

3、priority_queue常用函数实例解析

(1)push()

push(x)将x入队，时间复杂度O(log N)，其中N为当前优先队列中元素的个数

(2)top()

top()可以获得队首(堆顶)元素，时间复杂度O(log N)

(3)pop()

pop()令队首元素出队，时间复杂度O(log N)

(4)empty()

empty()检测priority_queue是否为空，为空返回true，不空返回false，时间复杂度O(1)

(5)size()

size()返回priority_queue内元素的个数，时间复杂度O(1)

4、priority_queue内元素优先级的设置(重点)

(1) 基本数据类型的优先级设置

在这里指的基本数据类型是int, char, double可以直接使用的数据类型
默认是数值越大优先级越高(大根堆)

```
priority_queue<int> q;
priority_queue<int, vector<int>, less<int>> q;
/*
int表示priority_queue中存储元素的类型，vector<int>是承载数据结构堆(heap)底层容器
less<int>表示数字越大优先级越高
greater<int>表示数据越小优先级越高
*/
```

(2)结构体的优先级的设置

```
struct cmp{
    bool operator()(ListNode *a, ListNode *b){
        return a->val > b->val;
    }
};
//优先队列中cmp函数的效果和sort中cmp的效果刚好相反
```

notes:
如果结构体中的数据较为庞大(例如出现字符串或数组)，可以使用引用来提高效率（在Leetcode中常用）
此时在参数中加上"const"、"&"

```
struct cmp{
    bool operator()(const ListNode *a, const ListNode *b){
        return a->val > b->val;
    }
};
```

优先队列相关题目：

743网络延迟时间

23. 合并K个升序链表

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    struct cmp{
        bool operator()(ListNode* a, ListNode *b){
            return a->val > b->val;
        }
    };

    ListNode* mergeKLists(vector<ListNode*>& lists) {
        priority_queue<ListNode*, vector<ListNode*>, cmp> pq;
        for (ListNode* list : lists){    //初始时将每个链表的头结点入队
            if (list)   pq.push(list);
        }

        ListNode *dummy = new ListNode();
        ListNode *r = dummy;
        while(pq.size()){
            ListNode *cur = pq.top(); pq.pop();
            r->next = cur, r = cur;
```

```
            if (cur->next)  pq.push(cur->next);
        }
        r->next = nullptr;
        return dummy->next;
    }
};
```

6.7 stack的常用解法详解

stack，翻译为栈，是STL中一个后进先出的容器

1、stack的定义

```
#include <stack>
using namespace std;

stack<typename> name;
```

2、stack容器内元素的访问

由于stack本身就是一种后进先出的数据结构，在STL的stack中只能通过top()来访问栈顶元素

3、stack常用函数实例解析

(1)push()

push(x)，将x入栈，时间复杂度O(1)

(2)top()

top()访问栈顶元素，时间复杂度O(1)

(3)pop()

pop()可以弹出栈顶元素，时间复杂度O(1)

(4)empty()

empty()可以检测stack()内是否为空，为空则返回true，时间复杂度O(1)

(5)size()

size()返回stack内元素的个数，时间复杂度O(1)

4、stack的常见用途

stack用来模拟实现一些递归，防止程序对栈的内存限制从而导致程序运行出错

5、单调栈

单调栈，通过自己定义一些规则，让栈内的元素满足一定的条件，来进行模拟

单调递减栈：①在一个数组中针对每一个元素从它右边寻找第一个比它大的元素

②在一个数组中针对每一个元素从它左边寻找第一个比它大的元素（从后往前遍历

739. 每日温度

```
vector<int> dailyTemperatures(vector<int>& temperatures) {
    //利用单调递减栈，来寻找下一个更大的值
    int n = temperatures.size();
    vector<int> res(n, 0);
    stack<int> st;  //需要用到下标
    for (int i = 0; i < n; i++){
        while(!st.empty() && temperatures[i] > temperatures[st.top()]){
            //栈顶元素之后更大的值是temperatures[i]
            int cur = st.top();
            st.pop();
            res[cur] = i - cur;
        }
        st.push(i);
    }
    return res;
}
```

1019. 链表中的下一个更大节点

```
vector<int> nextLargerNodes(ListNode* head) {
    vector<int> nums;//从链表中获取数据
    while(head){
        nums.push_back(head->val);
        head = head->next;
    }
    int len = nums.size();
    vector<int> res(len, 0);//首先默认所有节点都没有比他更大的结点，方便处理
    stack<int> st;          //维护一个单调递减的栈，来寻找下一个更大的值,存放的是节点的下标
    for (int i = 0; i < nums.size(); i++){
        while(!st.empty() && nums[i] > nums[st.top()]){
            int cur = st.top();//说明nums[i]比当前栈顶元素更大，也就是当前栈顶元素更大的值就是nums[i]
            st.pop();
            res[cur] = nums[i];
        }
        st.push(i);
    }
    return res;
}
```

单调递增栈

利用单调递减栈，可以很快找到>栈顶元素的元素
当前元素 > 栈顶元素
说明已经找到了右边第一个 >栈顶元素
根据单调栈的性质>栈顶元素的距离左边最近的元素位于栈顶的下一个位置
在这里就需要判断，出栈后栈是否为空，如果出栈后栈空，说明左边没有比栈顶元素更大的值

84. 柱状图中最大的矩形

```
int largestRectangleArea(vector<int>& heights) {
    heights.push_back(0);
    int res = 0;
    stack<int> st;
    for (int i = 0; i < heights.size(); i++){
        while(!st.empty() && heights[i] < heights[st.top()]){
            //当前元素比栈顶元素要低，我们就可以确定一栈顶元素为高的矩形面积
            int cur = st.top();
            st.pop();                   //cur指向栈顶元素并出栈
            int left;
            if (st.empty()) {
                left = 0;   //栈顶元素出栈后，栈为空说明左边没有比他要小的元素，left = 0
                res = max(res, i * heights[cur]);
            }else    {
                left = st.top();    //当栈不空，left指向比cur低的元素
                res = max(res, (i - left - 1) * heights[cur]);
            }

        }
        st.push(i);
    }
    return res;
}
```

6.8 pair的常用解法详解

pair将两个元素绑定在一起合成一个元素

1、pair的定义

```
#include <map>
using namespace std;

pair<typename1, typename2> name;
pair初始化
    1)、
        pair<string, int> p("hello word", 1);
        p = {"hello word", 1}
    2)、
        make_pair("hello word", 1)
```

2、pair中元素的访问

pair中只有两个元素，分别是first和second，只需按照正常接结构体的方式访问即可

3、pair常用函数实例解析

两个pair类型数据可以直接使用 == ！ = < <= > >=比较大小
比较规则是先以first的大小作为标准，只有first相等时才会比较second的大小
p.first　　p.second

4、pair的常见用途

1)、代替二元结构体和构造函数

2)、作为map的键值对来进行插入

```
map<string, int> mp;
    mp.insert(make_pair("hello", 1));
    mp.insert(make_pair("word", 2));
    for (auto it = mp.beegin(); it != mp.end(); it++)
        cout<<it->first<<" "<<it->second<<endl;
```

6.9 algorithm头文件下常用函数

1、max()、min()和abs()

max(x, y)、min(x, y)分别返回x、y中的最大值和最小值
abs(x) 返回x的绝对值(x是整数) fabs(x)(x是浮点数)
int index = max_element(nums.begin(), nums.end()) - nums.begin()
获取nums中元素值最大的下标

2、swap()

swap(x, y)交换x和y的值

3、reverse()

reverse(it1, it2)将容器的迭代器在[it1, it2)之间的元素进行反转

4、next_permutaion()

next_permutation()给出一个序列在全排列中的下一个序列
输出123构成的全排列

```
int nums[] = {1, 2, 3};
do {
    cout<<nums[0]<<" "<<nums[1]<<" "<<nums[2]<<endl;
}while(next_permutation(a, a+3))
//next_permutation在达到全排列的最后一个排列时会返回false
```

5、fill()

fill()可以将数组或容器中某一段区间赋为某个相同的值

```
fiil(arr, arr + N, value)

memset()    //常用来将数组中的值设置为0和-1
memset(arr, 0, sizeof(arr))
memset(dist, 0x3f, sizeof dist)
```

6、sort()

sort()用来进行排序

```
sort(首地址， 尾地址的下一个地址， [比较函数])
```

6.9.7 lower_bound()、upper_bound()

```
lower_bound(first, last, val)   //返回第一个>= val的元素的位置
upper_bound(first, last, val)   //返回第一个>  val的元素的位置
```