Standard Template Library

基础

头文件

```
#include <vector>
#include <deque>
#include <list>

#include <set>
#include <map>

#include <stack>
#include <queue>

#include <iterator>

#include <algorithm>
```

string类

- 构造函数

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

int main(){
  string s1="Hello";
  string s2("Hello");
  string s3(3,"F");//FFF
  //string s4='A'; 无对应构造函数
  //string s4('A'); 无对应构造函数
  string s4;
  s4='A';//或数字 会先被转化为ASCII字符
  return 0;
}
```

- 访问函数

```cpp
int main(){
  string s("Hello");
  s[0]='h';//不做范围检查
  s.at(0)='H';//做范围检查
  string s2=s.substr(3,2);//下标为3开始取2个字符
  const char *pc1=s.c_str();//末尾添加'\0'
  const char *pc2=s.data();//末尾不添加'\0'
  return 0;
}
```

- 其他函数

```cpp
int main(){
  string s1("Hello"),s2;

  if(s2.empty()){cout<<"Empty"<<endl;}//为空返回true
  cout<<s1.length()<<" "<<s1.size();//返回长度

  cout<<s1.find("lo",1);//从下标1开始找字串 返回下标 否则返回-1
  cout<<s1.rfind("lo",3);//从下表3开始向前找

  cout<<s1.compare(s2);//当前对象大或长返回1 小-1 相等0
  cout<<s1.compare(0,3,s2,2,3);//从0和2开始比较3个字符
  cout<<s1>s2;//返回bool型

  s2.insert(s2.length(),s1,0,3);//从s2.length()开始插入3字符
  s2.push_back('A');//在末尾增加一个字符
  s2.pop_back();//删除末尾字符
  s2.erase();//清空
  s2.erase(2);//删除下标2开始以后的字符

  s1.swap(s2);//成员函数
  swap(s1,s2);//全局函数
  s1.replace(2,1,"0-0");//下标2开始的1个字符替换

  s2=to_string(n);//转换数值对象
  return 0;
}
```

容器

顺序容器

元素插入位置与元素值无关

- `vector`：动态数组，尾端增删，随机存取
  - 支持所有STL算法
  - 访问：`*i; v[n]; v.at(n)`
  - 增删
    - 尾端（效率高）：`v.push_back(n); v.pop_back()`
    - 任意位置（效率低）：`v.insert(v.begin(),n); v.erase(v.begin(),v2.begin(),v2.end())`
  - 改变长度：`v.resize(new_l,n);改变长度为new_l 更短舍弃后面 更长添加缺省元素n 默认0`
- `deque`：双端队列，动态数组，两端增删，随机存取
  - 独有成员函数：`dq.push_front(n); dq.pop_front()`
- `list`：双向链表，不支持随机存取，任意位置增删
  - 访问：只能移动迭代器`i++; --i`访问
  - 增删
    - 首尾（效率高）：`li.push_back(n); li.pop_front()`
    - 任意位置（效率高）：`li.insert(...); li.erase(...)`
  - 改变长度：`li.resize(...)`
  - 反序：`li.reverse()`
  - 删除所有指定值：`li.remove(n)`
  - 简单去重：`li.unique(); 去重连续相同元素至一个`
    - 可以自定义相同的含义

```
template<class T>
bool Equal(T a,T b){
  return fabs(a-b)<0.5;
}

int main(){
  ...
  li.unique(Equal);
  return 0;
}
```

  - 排序：`li.sort(); li.sort(CMP)`
    - **不支持STL中的sort算法**
    - 使用自定义类时需要重载`operator <`

      可以自定义比较函数`bool CMP(T a,T b)`

    - 合并：`li1.merge(li2)`
        - 会清空`li2`
    - 插入：`li1.splice(li1.begin(),li2,li2.begin())`
        - 在`li1_start`前插入`li2_start`；省略`li2_start`则插入`li2`
- 顺序容器使用速度

	访问-	-或-	-修改	增加-	-或-	-删除	迭代器
	开头	中间	末尾	开头	中间	末尾	
`vector`	快	快	快	慢	慢	快	随机
`deque`	快	快	快	快	慢	快	随机
`list`	快	慢	快	快	快	快	双向

关联容器

均是双向迭代器

- `pair`：`key-value`对，头文件`utility`
    - `p.first`相当于`key`，`p.second`相当于`value`
    - 大小比较：`p1<p2`相当于`p1.first<p2.first`或`key`相同且`p1.second<p2.second`
    - 创建：`pair<T1,T2> p(v1,v2); make_pair(v1,v2)`
        - 后者的`<T1,T2>`为`v1,v2`的类型
- `set`：集合，平衡二叉树，元素自动**由小到大**排列，不允许相同元素
    - 初始化只需要一个类型`set<T> s`

      双向迭代器，不支持`s[n]`访问

      自定义类重载`operator <`
    - 插入：`s.insert(n)`
    - 删除：`s.erase(n)`
        - 传入元素，迭代器或两个迭代器（左闭右开的区间）
    - 查找：`s.find(x)`
        - 找到返回迭代器，否则`s.end()`

      查找下限：`s.lower_bound(x)`
        - 找到返回第一个大于等于`x`的迭代器，否则`s.end()`

      查找上限：`s.upper_bound(x)`
        - 找到返回第一个大于`x`的迭代器，否则`s.end()`

查找下限和上限：`s.equal_range(x)`

- 返回`pair<T1,T2> p(s.lower_bound(x),s.upper_bound(x))`

- 计算元素个数：`s.cound(n)`

- `multiset`：映射表，平衡二叉树，元素自动由小到大排列，允许相同元素
  - `ms.find()`返回第一个等于该值的迭代器
  - `ms.erase(n)`删除所有等于该值的元素
- `map`：平衡二叉树，元素按键值排列，不允许相同元素
  - 初始化声明两个类型`map<T1,T2> m`
  - 插入：`m.insert(make_pair(a,b))`
    - 不能直接初始化?
  - 访问或修改：迭代器（返回的是`<key,value>`的`pair`）或`m[key]`（`value`）
  - 删除：`m.erase(n)`
    - 传入参数为`key`或迭代器或两个迭代器（左闭右开的区间）
  - 查找：类似`set`
    - 注意返回的迭代器是`pair`，所以`equal_range(x)`返回的是两个`pair`类型的迭代器组成的`pair`
- `multimap`：平衡二叉树，元素按键值排列，允许相同元素
  - 不支持下标访问`m[key]`，因为`key`可能重复

容器适配器

不支持迭代器

- `stack`：栈，插入和修改都在尾部，后进先出
  - 缺省使用`deque`实现；可初始化为其他容器`stack<T,list<T>> s`
  - 插入/入栈：`s.push(n)`
    - 初始化最多使用复制构造函数
  - 删除/出栈：`s.pop()`
  - 访问/返回栈顶元素引用：`s.top()`
  - 清空：`while(!s.empty()) s.pop();`
- `queue`：队列，插入尾部，其他修改在头部，先进先出
  - 缺省使用`deque`实现；可初始化为其他容器
  - 插入/入队：`q.push(n)`
  - 删除/出队：`q.pop()`
  - 访问/返回队首元素引用：`q.front()`

- `priority_queue`：堆，保证最大元素在最前面
  - 缺省使用`vector`实现；可初始化为其他容器
  - 插入：`pq.push(n)`
    - 插入后队首元素可能变化
  - 删除：`pq.pop()`
    - 总是删除最大元素（队首）
  - 访问队首/最大元素的引用：`pq.top()`
    - 注意：不是`pq.front()`
  - 可以视作完全排好的序列，因为随时队首都是最大/最小（或符合规律），而又只能访问队首

容器总结

```
vector<T> v1,v2(5),v3(5,2),v4(v3);//v3:2,2,2,2,2
vector<T>::iterator i1=v3.begin();
//访问
cout<<*i1<<" "<<v3[1]<<" "<<v3.at(2)<<endl;
//增删
v1.push_back(3);
v1.pop_back();
v1.insert(v1.begin(),4);
v1.erase(v1.begin(),v3.begin(),v3.end());
//其他
v3.resize(new_length,n);//缺省元素n 默认0

deque<T> d;
//增删 其余和vector完全相同
d.push_front(5);
d.pop_front();

list<T> li(5,2);
list<T>::iterator itr_li=li.begin();
//访问
cout<<(*itr_li)<<" "<<(*(++itr_li))<<endl;
//增删
li.push_back(3);
li.pop_back();
li.push_front(4);
li.pop_front();
li1.insert(li1.begin(),4);
li1.erase(li1.begin(),li2.begin(),li2.end());
//其他
li.resize(new_length,n);//改变长度，缺省默认0
li.reverse();//反序
li.remove(n);//删除所有的n
li.unique();//简单去重 去重连续相同的元素至一个
li.unique(EQUAL);//简单去重 自定义想等函数
li.sort();//排序
li.sort(CMP)//排序 自定义比较函数
li1.merge(li2)//合并 会清空li2
li1.spice(li1.begin(),li2,li.begin())//在第一个迭代器前插入第二个迭代器所指值 省略第二个迭代器插入整个li2
```

```cpp
pair<T1,T2> p1,p2(var1,var2),p3=make_pair(var1,var2);
p1.first=var1;
p1.second=var2;

set<T> set;
set<T>::iterator itr_set;
multiset<T> mlset;
multiset<T>::iterator itr_mlset;
//访问
cout<<*itr_set<<endl;
//增删
set.insert(var);
set.erase(var);//或一个迭代器 或两个左开右闭的迭代器 mlset删除所有符合
//其他
set.find(var);//返回迭代器 未找到返回set.end()
set.lower_bound(var);//第一个值大于等于var的迭代器
set.upper_bound(var);//第一个值大于var的迭代器
set.equal_range(var);//返回pair(lower_bound,upper_bound)

map<T> map;
map<T>::iterator itr_map;
multimap<T> mlmap;
multimap<T> itr_mlmap;
//访问
cout<<(*itr_map).first<<" "<<(*itr_map).second<<endl
cout<<map[key]<<endl;//mlmap不能使用
//增删
map.insert(make_pair(key,value));
map[key]=value;
map.erase(key);//或一个迭代器 或两个左开右闭的迭代器
```

```cpp
stack<T,vector<T>> stk;//第二参数默认deque 可用所有顺序容器
stk.push(var);//入栈
stk.pop();//出栈
T var=stk.top();//返回栈顶元素引用
while(!stk.empty()) stk.pop();//清空

queue<T,vector<T>> que;//第二参数默认deque 可用所有顺序容器
que.push(var);//入队
que.pop();//出队
T var=que.front();//返回队首元素引用

priority_queue<T,deque<T>> pque;//第二参数默认vector 可用deque 不能用list
pque.push(var);
pque.pop();
T var=pque.top();
```

容器相关

构造函数

- 顺序容器构造函数
  - `v`：无参构造函数
  - `v(5); v(5,"SCP")`：两个参数，第一个为初始长度，第二个为初始值（无则为默认值）
  - `v(a,a+5); v2(itr1_v1,itr2_v1)`：两个参数，数组地址或容器迭代器，左开右闭

全部容器成员函数

- `operator`：各类比较运算符
- `empty()`：判断是否为空
- `max_size()`：判断最多能装多少元素
- `size()`：元素个数
- `swap()`：交换两个**容器**内容

顺序容器和关联容器成员函数

- `begin()`：第一个元素迭的代器
- `end()`：最后一个元素下一个位置的迭代器
- `rbegin()`：最后一个元素的迭代器
- `rend()`：第一个元素上一个位置的迭代器
- `erase()`：删除一个或多个元素
- `clear()`：清空容器

顺序容器成员函数

- `front()`：第一个元素的引用
- `back()`：最后一个元素的引用
- `push_back()`：在容器末尾增加元素
- `pop_back()`：删除容器末尾的元素

函数对象类模版

- 使用
  - 初始化声明时：`set<T,greater<T>>; priority_queue<T,greater<T>>`
    - 此时声明了对象类型
  - 函数内使用：`list.sort<greater<T>()>`
    - 此时创建了一个临时对象
- `greater<T>`：前者比后者大，返回`true`（交换）
  - `set`变为降序，`priority_queue`变为升序

    即`operator <`含义变得相反了！

迭代器

容器不同，对应的迭代器不同；由上自下越来越强，且强迭代器具有弱迭代器所有功能

注意：关联容器的迭代器会在区间修改后失效

- 输入/输出迭代器：只支持输入/输出（读/写）
- 正向迭代器：单向推进
- 双向迭代器：双向推进
  - 双向迭代器不能相加减、比较大小等
- 随机访问迭代器：随机移动

迭代器支持

- 随机迭代器：`vector; deque`
- 双向迭代器：`list; set/multiset; map/multimap`

迭代器操作

- 所有迭代器：`++i; i++`
- 输出迭代器：`*i; i=j`

  输入迭代器：以上，`i==j; i!=j`
- 正向迭代器：以上
- 双向迭代器：以上，`--i; i--`
- 随机访问迭代器：以上，`i+n; i-n; i<j; i[n]`

算法

- 函数对象：重载了`operator ()`的类
- 实现自定义算法的途径
  - 传入自定义函数对象
  - 传入函数
  - 在自定义类中重载`operator <`

不修改值的算法

- `for_each(itr1,itr2,op)`：遍历区间内元素执行op，需要支持迭代器；不能修改关联容器的元素值
- `equal(itr1,itr2,itr3,itr4)`：判断两个区间是否相等，需要支持迭代器
- `count(itr1,itr2,var)`：区间内计数指定元素，需要支持迭代器

修改值的算法

不适用于关联容器，关联容器的迭代器会在区间修改后失效

- `copy(itr1,itr2,itr3)`：复制区间内容
- `transform(itr1,itr2,itr3,op); transform(itr1,itr2,itr3,itr4,op)`：将`[itr1,itr2)`根据`op`写到`itr3`，或`[itr1,itr2)`与`itr3`开头的区间进行`op`写到`itr4`
- `swap()`：交换对象内容
- `replace()`：替换区间中的某个值

删除的含义是将删除的空位留出，剩下元素前移，最后多余的位置保持以前的值

删除算法返回删除后新区间后一个迭代器（新区间末尾到原区间末尾值不变，仍然可以访问）

- `remove(itr1,itr2,var)`
- `unique(itr1,itr2)`：去重（删除连续相同元素）
- `random_shuffle()`：随机排序

分割合并算法

- `partition()`：将区间按规则分割，满足排前面，不满足排后面
- `merge(itr1,itr2,itr3,itr4,itr5)`：合并两个区间到第三个区间，要求两个合并区间已经有序，必须支持随机迭代器；需要保证第三个区间接收长度足够
- `set_intersection()`：找出两个区间相同元素
- `set_difference()`：找出两个区间不同元素

查找算法

无序区间查找，需要用迭代器便利，不支持容器适配器

- `find(itr1,itr2,var)`：返回迭代器，未找到返回该区间末尾的迭代器
- `search(itr1,itr2,itr3,itr4,EQUAL)`：查找子区间`[itr3,itr4)`，可以传入自定义相等

有序区间查找，需要支持随机迭代器，`vector`或`deque`或普通数组

需要保证区间已经排好序

- `binary_search(itr1,itr2,var)`：二分查找，返回bool

排序算法

- `sort()`：快速排序，需要支持随机迭代器，只能用于`vector`和`deque`

堆算法

- `make_heap()`：根据指定序列构建堆
- `push_heap()`：向堆中插入元素
- `pop_heap()`：弹出堆顶元素

排列算法

不支持关联容器

- `reverse(itr1,itr2)`：将区间反序
- `next_permutation()`：产生指定序列下一个排列
- `prev_permutation()`：产生指定序列上一个排列