stl

佳木斯大学acm内部资料

整理人: ldc

stl:<pair>

构造函数

```
pair<T1,T2> p1; 创建类型为T1，T2的一个pair对象
pair<T1,T2> p1(v1,v2);  为T1，T2初始化
```

操作

```
make_pair(v1,v2)   创建一个新的pair对象 类型分别为v1 ，v2类型
/*两个pair对象之间小于运算，其定义遵循字典次序：如果p1.first < p2.first 或者! （p2.first < p1,first）&&p1.second < p2.second ，则返回true*/
p.first 返回第一个数的值
p.second 返回第二个数的值

例子:
typedef pair<string,string> Author;
Author proust("March","Proust");
Author Joy("James","Joy");

pair<int, double> p1;
p1 = make_pair(1, 1.2);

pair<int, double> p1(1, 1.2);
pair<int, double> p2 = p1;
```

stl: <vector>

构造函数

```
vector<T1> arr;  默认构造函数
vector<T1> arr(n);   创建具有n个变量的数组
vector<T1> arr(T1 *first,T1 *last); 用地址范围（first，last）初始化向量。符号first 和last 是指针标志方法。
```

操作:

```
C++   Vectors 能够使用标准运算符: ==, !=, <=, >=, <, 和 >;
v[]   要访问vector中的某特定位置的元素可以使用 [] 操作符.
v1 == v2    两个vectors被认为是相等的,如果: 1.它们具有相同的容量;2.所有相同位置的元素相等.
v1 != v2
v1 <= v2    vectors 之间大小的比较是按照词典规则.
v1 >= v2
v1 < v2
v1 > v2

arr.assign(input_iterator start, input_iterator end);
  //assign() 函数要么将区间[start, end)的元素赋到当前vector
arr.assign(size_type num, const T1 &val)
  //或者赋num个值为val的元素到vector中.这个函数将会清除掉为vector赋值以前的内容.

arr.at(index)// 返回当前Vector指定位置index的元素的引用.
at() 函数 比 [] 运算符更加安全, 因为它不会让你去访问到Vector内越界的元素


arr.back();//back() 函数返回当前vector最末一个元素的引用.
arr.front();// front() 函数返回当前vector起始元素的引用

arr.begin(); // begin() 函数返回一个指向当前vector起始元素的迭代器.
arr.end();//  end() 函数返回一个指向当前vector末尾元素的下一位置的迭代器.注意,如果你要访问末尾元素,
需要先将此迭代器自减1.

arr.capacity();  //capacity()  函数 返回当前vector在重新进行内存分配以前所能容纳的元素数量.

arr.clear();//clear()函数删除当前vector中的所有元素

arr.empty();// 如果当前vector没有容纳任何元素,则empty()函数返回true,否则返回false.

arr.erase( iterator loc );  // erase 函数删作指定位置loc的元素
arr.erase( iterator start, iterator end ); // 删除区间[start, end)的所有元素.返回值是指向删除
的最后一个元素的下一位置的迭代器


arr.insert(iterator loc, const T1 &val);
//在指定位置loc前插入值为val的元素,返回指向这个元素的迭代器,
arr.insert(iterator loc, size_type num, const T1 &val );
// 在指定位置loc前插入num个值为val的元素
arr.insert(iterator loc, input_iterator start, input_iterator end);
// 在指定位置loc前插入区间[start, end)的所有元素

arr.rbegin();//rbegin 函数返回指向当前vector末尾的逆迭代器.
arr.rend();  //rend()函数返回指向当前vector起始位置的逆迭代器.

arr.reserve(size_type size);// 函数为当前vector预留至少共容纳size个元素的空间.(译注:实际空间可能
大于size)

arr.resize(size_type size,T1 val);//resize()  函数改变当前vector的大小为size,且对新创建的元素
赋值val


arr.size(); //size()  函数返回当前vector所容纳元素的数目
```

```
arr.swap(vector &from );//swap() 函数交换当前vector与vector from的元素

例子:
利用vector创建二维数组
vector<T1> arr[maxm];
for(int i = 0; i< maxn; i++){
    arr[i].reserve(n);
}

int a[3][3]={1,2,3,4,5,6,7,8,9};
vector<int> v(a[0],a[3]);

//数组的默认值是0
```

stl:<string>

构造函数

```
a)      string s;                              // 生成一个空字符串s
b)      string s(str)                        // 拷贝构造函数 生成str的复制品
c)      string s(str,stridx)               // 将字符串str内“始于位置stridx”的部分当作字符串的初值
d)      string s(str,stridx,strlen)  // 将字符串str内“始于stridx且长度顶多strlen”的部分作为字符串的初值
e)      string s(cstr)                      // 将C字符串作为s的初值
f)      string s(chars,chars_len) // 将C字符串前chars_len个字符作为字符串s的初值。
g)      string s(num,c)                 // 生成一个字符串，包含num个c字符
h)      string s(begin,end)              // 以区间begin;end(不包含end)内的字符作为字符串s的初值
```

相关操作

这里是C++字符串的重点，我先把各种操作函数罗列出来，不喜欢把所有函数都看完的人可以在这里找自己喜欢的函数，再到后面看他的详细解释。

a) =,assign() //赋以新值

b) swap() //交换两个字符串的内容

c) +=,append(),push_back() // 在尾部添加字符

d) insert() //插入字符

e) erase() //删除字符

f) clear() //删除全部字符

g) replace() //替换字符

h) + //串联字符串

i) ==,!=,<,<=,>,>=,compare() // 比较字符串

j) size(),length() //返回字符数量

k) max_size() //返回字符的可能最大个数

l) empty() //判断字符串是否为空

m) capacity() //返回重新分配之前的字符容量

n) reserve() //保留一定量内存以容纳一定数量的字符

o) [ ], at() //存取单一字符

p) >>,getline() //从stream读取某值

q) << //将谋值写入stream

r) copy() //将某值赋值为一个C_string

s) c_str() //将内容以C_string返回

t) data() //将内容以字符数组形式返回

u) substr() //返回某个子字符串

v)查找函数

w)begin() end() //提供类似STL的迭代器支持

x) rbegin() rend() //逆向迭代器

y) get_allocator() //返回配置器

例子：
比较函数：

```
string s("abcd");
s.compare("abcd"); // 返回0
s.compare("dcba"); // 返回一个小于0的值
s.compare("ab"); // 返回大于0的值
s.compare(s); // 相等
s.compare(0,2,s,2,2); // 用"ab"和"cd"进行比较 小于零
s.compare(1,2,"bcx",2); // 用"bc"和"bc"比较。
```

更改内容：

```
s.assign(str);
s.assign(str,1,3);// 如果str是"iamangel" 就是把"ama"赋给字符串
s.assign(str,2,string::npos);// 把字符串str从索引值2开始到结尾赋给s
s.assign("gaint");
s.assign("nico",5);// 把’n’ ‘I’ ‘c’ ‘o’ ‘\0’赋给字符串
s.assign(5,’x’);// 把五个x赋给字符串
```

string提供了很多函数用于插入（insert）、删除（erase）、替换（replace）、增加字符。
先说增加字符，函数有 +=、append()、push_back()。举例如下：

```
s+=str;//加个字符串
s+="my name is jiayp";// 加个C字符串
s+=’a’;//加个字符
s.append(str);
s.append(str,1,3);//  同前面的函数参数assign的解释
s.append(str,2,string::npos)// 不解释了
s.append("my name is jiayp");
s.append("nico",5);
s.append(5,’x’);
s.push_back(‘a’);// 这个函数只能增加单个字符
s.insert(0,"my name"); // 从0号后面插入元素
s.insert(1,str); // 从1号后面插入元素
```

```
string s="il8n";
s.replace(1,2,"nternationalizatio");// 从索引1开始的2个替换成后面的C_string
s.erase(13);//从索引13开始往后全删除
s.erase(7,5);//从索引7开始往后删5个
```

提取子串和字符串连接

```
s.substr();//返回s的全部内容
s.substr(11);//从索引11往后的子串
s.substr(5,6);//从索引5开始6个字符
```

搜做与查找

```
s.find('lalala', 2);
```

在str中，从位置2（包括位置2）开始，查找字符'lalala'，返回首次匹配的位置，若匹配失败，返回npos

npos的含义，string::npos 的类型是string::size_type，所以，一旦需要把一个索引与npos相比，这个索引值必须是string::size_type 类型的，更多的情况下，我们可以直接把函数和npos进行比较（如：
if(s.find("jia")== string::npos) ）。

stl<list>

构造函数

```
list<int>lst1;             // 创建空list
list<int> lst2(5);        // 创建含有5个元素的list
list<int>lst3(3,2);       // 创建含有3个元素的list，每个元素的值为2
list<int>lst4(lst2);      // 使用lst2初始化lst4
list<int>lst5(lst2.begin(),lst2.end());  // 同lst4
```

操作

list的非变动性操作	
c.size()	返回容器的大小
c.empty()	判断容器是否为空,等价于size()==0,但可能更快
c.max_size()	返回容器最大的可以存储的元素
reserve()	如果容量不足，扩大之
c1 == c2	判断c1 是否等于c2
c1 != c2	判断c1是否不等于c2
c1 < c2	判断c1 是否小于c2
c1 > c2	判断c1 是否大于c2
c1 <= c2	判断c1是否小于等于c2
c1 >= c2	判断c1是否大于等于c2
list的赋值操作	
c1 = c2	将c2的全部元素赋值给c1
c.assign(n, elem)	复制n个elem，复制给c
c.assign(beg, end)	将区间[beg;end）内的元素赋值给c
c1.swap(c2)	将c1和c2元素互换
swap(c1,c2)	同上，此为全局函数
list元素之间存取	
c.front	返回第一个元素，不检查元素存在与否
c.back	返回最后一个元素，不检查元素存在与否
list迭代器相关函数	
c.begin()	返回一个双向迭代器，指向第一个元素
c.end()	返回一个双向迭代器，指向最后一个元素的下一个位置
c.rbegin()	返回一个逆向迭代器，指向逆向迭代的第一个元素
c.rend()	返回一个逆向迭代器，指向逆向迭代的最后一个元素的下一个位置
list安插、移除操作函数	

list的非变动性操作	
c.insert(pos, elem)	在迭代器pos所指位置上安插一个elem副本，并返回新元素的位置
c.insert(pos,n,elem)	在pos位置上插入n个elem副本，无返回值
c.insert(pos,begin,end)	在pos位置上插入区间[begin,end)内的所有元素的副本没有返回值
c.push_back(elem)	在尾部添加一个elem副本
c.pop_back()	移除最后一个元素，无返回值
c.push_front()	在头部添加一个elem副本
c.pop_front()	移除第一个元素，但不返回
c.remove(val)	移除所有其值为val的元素
c.remove_if()	
c.erase(pos)	移除pos位置上的元素，返回下一个元素的位置
c.erase(begin, end)	移除(begin, end)区间内的所有元素，返回下一个元素的位置
c.resize(num)	将元素数量改为num(如果size()变大了，多出来的新元素都需以default构造函数完成)
c.resize(num,elem)	将元素数量改为num(如果size()变大了，多出来的新元素都elem的副本)
c.clear()	移除所有元素，将容器清空
备注：安插和移除元素，都会使“作用点”之后的各个元素的iterator等失效，若发生内存重新分配，该容器身上的所有iterator等都会失效	
List的特殊变动性操作	
c.unique()	如果存在若干相邻而数值相等的元素，就移除重复元素，之留下一个
c.unique(op)	如果存在若干相邻元素，都使op()的结果为ture，则移除重复元素，只留下一个。
c1.splice(pos, c2)	将c2内的所有元素转移到c1之内，迭代器pos之前
c1.splice(pos, c2, c2pos)	将c2内的c2pos所指元素转移到c1之内的pos所指位置上(c1,c2可相同)
c1.splice(pos, c2, c2beg，c2end)	将c2内的[c2beg，c2end）区间内所有元素转移到c1内的pos之前(c1,c2可相同)

list的非变动性操作	
c.sort()	以operator<为准则，对所有元素排序
c.sort(op)	以op()为准则，对所有元素排序
c1.merge(c2)	假设c1和c2容器都包含已序(相同的排序方式)元素，将c2的全部元素转移到c1，并保证合并后的list还是已序。
c1.merge(c2，op)	假设c1和c2容器都包含op()原则下的已序(相同的排序方式)元素，将c2的全部元素转移到c1，并保证合并后的list在op（）原则仍是已序。
c.reverse()	将所有元素反

//具体操作及样例见STL使用

stl:<stack>,<queue>与<deque>

<stack>

构造函数

```
stack<T1> st;   // 生成一个空栈
```

相关操作

```
s.push(x);      // 入栈
s.pop();        // 出栈，注意，出栈操作只是删除栈顶元素，并不返回该元素。
s.top();        //访问栈顶元素
s.empty(),      // 判断栈空，当栈空时，返回true。
s.size(),       // 访问栈中元素个数
```

<queue>

构造函数

```
queue<T1> qu  // 创建一个空队列
```

相关操作

```
q.push(x);      // 入队，  将x 接到队列的末端。
q.pop();        // 出队，  弹出队列的第一个元素，注意，并不会返回被弹出元素的值。
q.front(),      // 访问队首元素，即最早被压入队列的元素。
q.back(),       //访问队尾元素，即最后被压入队列的元素。
q.empty(),      //判断队空，当队列空时，返回true。

q.size()        // 访问队中元素个数
```

优先队列

构造函数

```
priority_queue<T1> pqu() ; 创建一个空队列
```

相关操作

```
q.empty() const;              // 检查优先级队列是否为空，若为空，返回TRUE，否则返回ELASE
q.pop();                      // 从优先级队列中删除优先级最大的元素，前提条件：队列不为空
q.push(const T1 &item);       // 向优先级队列中插入一个元素
q.size() const;               // 返回优先级队列中元素的个数
T1 & top();                   // 返回具有最高优先级的元素的引用
const T1 & top() const;       // 常量版top（）

例子：

priority_queue<int>que;// 采用默认优先级构造队列

// 使用grater与less时加上<functional> 头文件
priority_queue<int,vector<int>,greater<int> >que3;//  注意“>>”会被认为错误，
priority_queue<int,vector<int>,less<int> >que4;////  最大值优先

//定义比较结构
struct cmp1{
    bool operator ()(int &a,int &b){
        return a>b;// 最小值优先
    }
};

struct cmp2{
    bool operator ()(int &a,int &b){
        return a<b;// 最大值优先
    }
};


priority_queue<int,vector<int>,cmp1>que1;// 最小值优先
priority_queue<int,vector<int>,cmp2>que2;// 最大值优先


//自定义数据结构
struct number1{
    int x;
    bool operator < (const number1 &a) const {
        return x>a.x;// 最小值优先
    }
};
struct number2{
    int x;
    bool operator < (const number2 &a) const {
        return x<a.x;// 最大值优先
    }
};

priority_queue<number1>que5; // 最小优先级队列
priority_queue<number2>que6;  // 最大优先级队列
```

<deque>

构造函数

```
deque<int> deq;// 创建一个没有任何元素的deque对象
deque<int> deq(10)// 创建一个具有10个元素的deque对象，默认值为0
deque<char> deq(5,'k')
deque<char>  deq(d1)// 拷贝构造函数
```

相关操作

```
deq[ ]：用来访问双向队列中单个的元素。
deq.front()：返回第一个元素的引用。
deq.back()：返回最后一个元素的引用。
deq.push_front(x)：把元素x插入到双向队列的头部。
deq.pop_front()：弹出双向队列的第一个元素。
deq.push_back(x)：把元素x插入到双向队列的尾部。
deq.pop_back()：弹出双向队列的最后一个元素。

插入：
将在pos位置之前，插入新元素x。
iterator insert(iterator pos, const T& x)

删除：
iterator erase(iterator pos)
   删除pos所指向的元素。
   iterator erase(iterator first, iterator last)
   删除迭代器区间[first,last)所指向的所有元素。
void clear()
   删除所有元素。

交换
void swap(deque&)
  如d1.swap(d2);

其它
1）bool empty()
   判断deque容器是否已有元素，是则返回true，否则返回false。
2）size_type size()
   当前deque容器的元素个数。
3）size_type max_size()
   系统所支持的deque容器的最大元素个数。
4）reference front()
   deque容器的首元素（引用返回），要求deque不为空。
5）reference back()
   deque容器的末元素（引用返回），要求deque不为空。
```

stl:<set>

构造函数

```
set<T1> se;  创建空的集合，这是默认的构造函数

set<T1> se(T1*first,T1*last);   使用地址区间[first，last]来初始化集合
```

相关操作

```
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
s.end();                // 返回指向最后一个元素的迭代器
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

例子:
自定义比较函数
1，元素不是结构体:
struct myComp
{
     bool operator()(const your_type &a,const your_type &b)
     {
         return a.data-b.data>0;
     }
 }
 set<int,myComp>s;
 ......
 set<int,myComp>::iterator it;

 2，如果元素是结构体，可以直接将比较函数写在结构体内。
  struct Info
 {
     string name;
     float score;
     //重载“<”操作符，自定义排序规则
     bool operator < (const Info &a) const
     {
         //按score从大到小排列
         return a.score<score;
     }
 }
 set<Info> s;
 ......
 set<Info>::iterator it;


 跟多操作见STL使用
```

multiset:

介绍：大部分操作与set相同

```
int count( const T & item) const;   返回多重集中匹配的item的个数

pair < iterator,iterator>equal_range( const T & item);     返回一对迭代器，使得所有匹配的元素都在区间pair的first的成员，pair的second成员之内。

iterator insert （const T & item); 把item添加到多重集中去，返回新元素的迭代器，后置条件：元素item被加入到多重集中去

例子：
#include<stdio.h>
#include<set>
using namespace std;
multiset<int> t;
int main(void)
{
    int i;
    for(i=1;i<=3;i++)
        t.insert(2);
    t.insert(1);
    t.insert(4);
    //printf("%d\n", t.size());
    multiset<int>::iterator it;
    for(it=t.begin();it!=t.end();it++)          //  输出1 2 2 2 4
        printf("%d ", *it);
    printf("\n");

    it = t.find(2);
    t.erase(t.find(2));
    for(it=t.begin();it!=t.end();it++)          //  输出1 2 2 4
        printf("%d ", *it);
    printf("\n");

    printf("%d %d\n", *(t.lower_bound(2)), *(t.upper_bound(2)));        //  输出2 4
    return 0;
}
```

stl:<map>

构造函数：

```
map<T1,T2> ma;

例子:
map<int, string> ID_Name = {
                { 2015, "Jim" },
                { 2016, "Tom" },
                { 2017, "Bob" }
};
```

相关操作

```
C++ Maps是一种关联式容器，包含“关键字/值”对
begin()           //返回指向map头部的迭代器
clear()           //删除所有元素
count()           //返回指定元素出现的次数
empty()           //如果map为空则返回true
end()             //返回指向map末尾的迭代器
equal_range()     //返回特殊条目的迭代器对
erase()           //删除一个元素
find()            //查找一个元素
get_allocator()   //返回map的配置器
insert()          //插入元素
key_comp()        //返回比较元素key的函数
lower_bound()     //返回键值>=给定元素的第一个位置
max_size()        //返回可以容纳的最大元素个数
rbegin()          //返回一个指向map尾部的逆向迭代器
rend()            //返回一个指向map头部的逆向迭代器
size()            //返回map中元素的个数
swap()            //交换两个map
upper_bound()     //返回键值>给定元素的第一个位置
value_comp()      //返回比较元素value的函数


例子:
查找:
find()函数返回一个迭代器指向键值为key的元素，如果没找到就返回指向map尾部的迭代器。
   map<int ,string >::iterator l_it;
   l_it=maplive.find(112);

 交换:
 Map中的swap不是一个容器中的元素交换，而是两个容器交换;

 排序:
 map中的元素是自动按key升序排序,所以不能对map用sort函数;
```

multimap

构造函数:

```
multimap <T1, T2> DNS_daemon;
```

相关操作

提出问题　　与 map 不同，multimap 可以包含重复键。这就带来一个问题：重载下标操作符如何返回相同键的多个关联值？以下面的伪码为例：

```
string phone=phonebook["Harry];
```

标准库设计者的解决这个问题方法是从 multimap 中去掉下标操作符。因此，需要用不同的方法来插入和获取元素以及和进行错误处理。 插入　　假设你需要开发一个 DNS 后台程序（也就是 Windows 系统中的服务程序），该程序将 IP 地址映射匹配的 URL 串。你知道在某些情况下，相同的 IP 地址要被关联到多个 URLs。这些 URLs 全都指向相同的站点。在这种情况下，你应该使用 multimap，而不是 map。例如：

```
#include <map>
            #include <string>
            multimap <string, string> DNS_daemon;
```

用 insert() 成员函数而不是下标操作符来插入元素。insert()有一个 pair 类型的参数。在“使用 库创建关联容器”中我们示范了如何使用 make_pair() 辅助函数来完成此任务。你也可以象下面这样使用它：

```
DNS_daemon.insert(make_pair("213.108.96.7","cppzone.com"));
```

在上面的 insert()调用中，串 “213.108.96.7”是键，“cppzone.com”是其关联的值。以后插入的是相同的键，不同的关联值：

```
DNS_daemon.insert(make_pair("213.108.96.7","cpplusplusZone.com"));
```

因此，DNS_daemon 包含两个用相同键值的元素。注意 multimap::insert() 和 map::insert() 返回的值是不同的。

```
typedef pair <const Key, T> value_type;
            iterator
            insert(const value_type&); // #1 multimap
            pair <iterator, bool>
            insert(const value_type&); // #2 map
```

multimap::insert()成员函数返回指向新插入元素的迭代指针，也就是 iterator（multimap::insert()总是能执行成功）。但是 map::insert() 返回 pair，此处 bool 值表示插入操作是否成功。 查找单个值　　与 map 类似，multimap 具备两个版本重载的 find()成员函数：

```
iterator find(const key_type& k);
            const_iterator find(const key_type& k) const;
```

find(k) 返回指向第一个与键 k 匹配的 pair 的迭代指针，这就是说，当你想要检查是否存在至少一个与该键关联的值时，或者只需第一个匹配时，这个函数最有用。例如：

```
typedef multimap <string, string> mmss;
            void func(const mmss & dns)
            {
            mmss::const_iterator cit=dns.find("213.108.96.7");
            if (cit != dns.end())
            cout <<"213.108.96.7 found" <<endl;
            else
            cout <<"not found" <<endl;
            }
```

处理多个关联值　　count(k) 成员函数返回与给定键关联的值得数量。下面的例子报告了有多少个与键“213.108.96.7” 关联的值：

```
cout<<dns.count("213.108.96.7") //output: 2
            <<" elements associated"<<endl;
```

为了存取 multimap 中的多个值，使用 equal_range()、lower_bound()和 upper_bound()成员函数：equal_range(k)：该函数查找所有与 k 关联的值。返回迭代指针的 pair，它标记开始和结束范围。下面的例子显示所有与键“213.108.96.7”关联的值：

```
typedef multimap <string, string>::const_iterator CIT;
            typedef pair<CIT, CIT> Range;
            Range range=dns.equal_range("213.108.96.7");
            for(CIT i=range.first; i!=range.second; ++i)
            cout << i->second << endl; //output: cpluspluszone.com
            // cppzone.com
```

lower_bound() 和 upper_bound()：lower_bound(k) 查找第一个与键 k 关联的值，而 upper_bound(k) 是查找第一个键值比 k 大的元素。下面的例子示范用 upper_bound()来定位第一个其键值大于“213.108.96.7”的元素。通常，当键是一个字符串时，会有一个词典编纂比较：

```
dns.insert(make_pair("219.108.96.70", "pythonzone.com"));
            CIT cit=dns.upper_bound("213.108.96.7");
            if (cit!=dns.end()) //found anything?
            cout<<cit->second<<endl; //display: pythonzone.com
```

如果你想显示其后所有的值，可以用下面这样的循环：

```
// 插入有相同键的多个值
            dns.insert(make_pair("219.108.96.70","pythonzone.com"));
            dns.insert(make_pair("219.108.96.70","python-zone.com"));
            //  获得第一个值的迭代指针
            CIT cit=dns.upper_bound("213.108.96.7");
            // 输出: pythonzone.com , python-zone.com
            while(cit!=dns.end())
            {
                  cout<<cit->second<< endl;
                  ++cit;
            }
```

stl:<algorithm>

1.不修改内容

adjacent_find	查找两个相邻（**Adjacent**）的等价（**Identical**）元素
all_of	检测在给定范围中是否所有元素都满足给定的条件
any_of	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_not	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_of	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

2.修改内容操作

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_if	将一个范围中满足给定条件的元素拷贝到新的位置处
copy_n	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
move	将一个范围中的元素移动到新的位置处
move_backward	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
shuffle	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

copy	将一个范围中的元素拷贝到新的位置处
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

3.划分操作

is_partitioned	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_copy	拷贝指定范围的划分结果
partition_point	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

4.排序操作

is_sorted	检测指定范围是否已排序
is_sorted_until	返回最大已排序子范围
nth_element	部分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	部分排序
partial_sort_copy	拷贝部分排序的结果
sort	排序
stable_sort	稳定排序

5.查找操作

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

6.集合操作

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

7.堆操作

is_heap	检测给定范围是否满足堆结构
is_heap_until	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

8.最大最小操作

is_permutation	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
minmax	返回两个元素中值最大及最小的元素
minmax_element	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

常用操作：

1，sort();

默认从小到大排序

```
sort(a,a+10);
```

从大到小排序

```
bool complare(int a,int b)
{
 return a>b;
}

sort(a,a+10，complare);
```

使用less<T1>与greater<T1>

```
sort(a,a+10,less<int>());// 从小到大
sort(a,a+10,greater<int>()); 从大到小
```

对字符排序

```
sort(a,a+10,greater<char>());
```

自定义结构排序

```
bool complare(T1. a,T1 b)
{
 return a.x>b.x;
}

sort(a,a+10，complare);
```

2.transform

transform函数有两个重载版本：

transform(first,last,result,op);//first是容器的首迭代器，last为容器的末迭代器，result为存放结果的容器，op为要进行操作的一元函数对象或sturct、class。

transform(first1,last1,first2,result,binary_op);//first1是第一个容器的首迭代器，last1为第一个容器的末迭代器，first2为第二个容器的首迭代器，result为存放结果的容器，binary_op为要进行操作的二元函数对象或sturct、class

1，大小写转换

```
transform(s.begin(),s.end(),s.begin(),::toupper);//  变为大写
transform(s.begin(),s.end(),s.begin(),::tolower);//  变为小写


toupper和tolower在C++中定义分别在std和cctype中
```

2，给你两个vector向量（元素个数相等），利用transform函数将两个vector的每个元素相乘，并输出相乘的结果。

```
int op(int a,int b){return a*b;}
transform(A.begin(),A.end(),B.begin(),SUM.begin(),op);
```

3.find

int *find(int *begin，int *end，int value)

前闭后合的区间 begin，end中，查找value如果查找到了就返回第一个符合条件的元素，否则返回end指针

```
#include <iostream>
#include <algorithm>

using namespace std;

void printElem(int& elem)
{
  cout << elem << endl;
}
int main()
{
    int ia[]={0,1,2,3,4,5,6};

    int *i= find(ia,ia+7,9);// 在整个数组中查找元素 9
    int *j= find(ia,ia+7,3);// 在整个数组中查找元素 3
    int *end=ia+7;// 数组最后位置
    if(i == end)
       cout<<"没有找到元素 9"<<endl;
    else
       cout<<"找到元素9"<<endl;

    if(j == end)
       cout<<"没有找到元素 3"<<endl;
    else
       cout<<"找到元素"<<*j<<endl;
    return 0;
}
```