STL 参考资料

袁辉勇 整理

2009 年 11 月

目录

STL 介绍.....3
1、STL 简介.....3
2、算法.....3
3、容器.....3
4、迭代器.....4
5、使用注意.....4
一、stack 堆栈.....5
成员函数：.....5
实例程序：.....5
二、queue 队列.....6
成员函数：.....6
实例程序：.....6
三、Priority Queues 优先队列.....7
成员函数：.....7
实例程序：.....7
四、Bitset 位集合.....9
成员函数：.....9
实例程序：.....9
五、list 列表.....11
成员函数：.....11
实例程序：.....12
六、vector 向量.....13
成员函数：.....13
实例程序：.....14
七、map / multimap 映射与多重映射.....15
map 成员函数：.....15
Map 实例程序：.....17
MultiMap 实例程序：.....18
八、set / multiset 集合与多重集合.....19
成员函数：.....19
Set 实例程序：.....20
MultiSet 实例程序：.....21
九、deque (Double Ended Queue) 双端队列.....22
成员函数：.....22
实例程序：.....23
十、string 字符串.....24
成员函数：.....24
实例程序：.....28
十一、常用算法调用.....29
1. for_each.....29
2. min_element / max_element.....29
3. copy / copy_n /copy_backward.....29
4. fill / fill_n.....30

5. remove / remove_if....30
6. unique....31
7. rotate....32
8. random_shuffle....32
9. partition / stable_partition....33
10. sort / stable_sort....33
11. partial_sort....34
12. nth_element....34
13. lower_bound / upper_bound //要求区间有序....34
14. binary_search //要求有序区间....35
15. merge / inplace_merge....35
16. includes....36
17. set_union, set_intersection, set_difference, set_symmetric_diffrece....36
18. next_permutation / prev_permutation....37
19. power....37
20. heap operations....38
21. min / max / swap....39
22. numeric_limits....39

STL 介绍

1、STL 简介

STL（Standard Template Library，标准模板库)是惠普实验室开发的一系列软件的统称。它是由 Alexander Stepanov、Meng Lee 和 David R Musser 在惠普实验室工作时所开发出来的。现在虽说它主要出现在 C++中，但在被引入 C++之前该技术就已经存在了很长的一段时间。

STL 的代码从广义上讲分为三类：algorithm（算法）、container（容器）和 iterator（迭代器），几乎所有的代码都采用了模板类和模版函数的方式，这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。在 C++标准中，STL 被组织为下面的 13 个头文件：<algorithm>、<deque>、<functional>、<iterator>、<vector>、<list>、<map>、<memory>、<numeric>、<queue>、<set>、<stack>和<utility>。以下笔者就简单介绍一下 STL 各个部分的主要特点。

2、算法

大家都能取得的一个共识是函数库对数据类型的选择对其可重用性起着至关重要的作用。举例来说，一个求方根的函数，在使用浮点数作为其参数类型的情况下的可重用性肯定比使用整型作为它的参数类性要高。而 C++通过模板的机制允许推迟对某些类型的选择，直到真正想使用模板或者说对模板进行特化的时候，STL 就利用了这一点提供了相当多的有用算法。它是在一个有效的框架中完成这些算法的—你可以将所有的类型划分为少数的几类，然后就可以在模版的参数中使用一种类型替换掉同一种类中的其他类型。

STL 提供了大约 100 个实现算法的模版函数，比如算法 for_each 将为指定序列中的每一个元素调用指定的函数，stable_sort 以你所指定的规则对序列进行稳定性排序等等。这样一来，只要我们熟悉了 STL 之后，许多代码可以被大大的化简，只需要通过调用一两个算法模板，就可以完成所需要的功能并大大地提升效率。

算法部分主要由头文件<algorithm>，<numeric>和<functional>组成。<algorithm>是所有 STL 头文件中最大的一个（尽管它很好理解），它是由一大堆模版函数组成的，可以认为每个函数在很大程度上都是独立的，其中常用到的功能范围涉及到比较、交换、查找、遍历操作、复制、修改、移除、反转、排序、合并等等。<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，包括加法和乘法在序列上的一些操作。<functional>中则定义了一些模板类，用以声明函数对象。

3、容器

在实际的开发过程中，数据结构本身的重要性不会逊于操作于数据结构的算法的重要性，当程序中存在着对时间要求很高的部分时，数据结构的选择就显得更加重要。

经典的数据结构数量有限，但是我们常常重复着一些为了实现向量、链表等结构而编写的代码，这些代码都十分相似，只是为了适应不同数据的变化而在细节上有所出入。STL 容器就为我们提供了这样的方便，它允许我们重复利用已有的实现构造自己的特定类型下的数据结构，通过设置一些模板类，STL 容器对最常用的数据结构提供了支持，这些模板的参数允许我们指定容器中元素的数据类型，可以将我们许多重复而乏味的工作简化。

容器部分主要由头文件<vector>,<list>,<deque>,<set>,<map>,<stack>和<queue>组成。对于常用的一些容器和容器适配器（可以看作由其它容器实现的容器），可以通过下表总结一下它们和相应头文件的对应关系。

数据结构	描述	实现头文件
向量(vector)	连续存储的元素	〈vector〉
列表(list)	由节点组成的双向链表，每个结点包含着一个元素	〈list〉
双队列(deque)	连续存储的指向不同元素的指针所组成的数组	〈deque〉
集合(set)	由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种作用于元素对的谓词排列，没有两个不同的元素能够拥有相同的次序	〈set〉
多重集合(multiset)	允许存在两个次序相等的元素的集合	〈set〉
栈(stack)	后进先出的值的排列	〈stack〉
队列(queue)	先进先出的执的排列	〈queue〉
优先队列(priority_queue)	元素的次序是由作用于所存储的值对上的某种谓词决定的的一种队列	〈queue〉
映射(map)	由{键，值}对组成的集合，以某种作用于键对上的谓词排列	〈map〉
多重映射(multimap)	允许键对有相等的次序的映射	〈map〉

4、迭代器

下面要说的迭代器从作用上来说是最基本的部分，可是理解起来比前两者都要费力一些（至少笔者是这样）。软件设计有一个基本原则，所有的问题都可以通过引进一个间接层来简化，这种简化在 STL 中就是用迭代器来完成的。概括来说，迭代器在 STL 中用来将算法和容器联系起来，起着一种黏和剂的作用。几乎 STL 提供的所有算法都是通过迭代器存取元素序列进行工作的，每一个容器都定义了其本身所专有的迭代器，用以存取容器中的元素。

迭代器部分主要由头文件<utility>,<iterator>和<memory>组成。<utility>是一个很小的头文件，它包括了贯穿使用在 STL 中的几个模板的声明，<iterator>中提供了迭代器使用的许多方法，而对于<memory>的描述则十分的困难，它以不同寻常的方式为容器中的元素分配存储空间，同时也为某些算法执行期间产生的临时对象提供机制,<memory>中的主要部分是模板类 allocator，它负责产生所有容器中的默认分配器。

5、使用注意

STL 的区间都是左闭右开的。例如： [start, end) 表示从 start 开始到 end 之前一个位置。

一、stack 堆栈

头文件： `#include<stack>`
实例化： `stack<类型[, 存储容器]>StackName`

成员函数：

bool empty();	栈为空返回 **true** 否则返回 **false**.
void pop();	移除堆栈中最顶层元素。
void push(**const** TYPE &val);	将 val 值压栈，使其成为栈顶的第一个元素
size_type size();	返当前堆栈中的元素数目
TYPE &top();	返回对栈顶元素的引用

实例程序：

```
#include <iostream>
#include <stack>
#include <algorithm>
using namespace std;
int main()
{   stack<int>s;
    s.push(1);    s.push(2);    s.push(3);
    cout << "Top: " << s.top() << endl;
    cout << "Size: " << s.size() << endl;
    s.pop();
    cout << "Size: " << s.size() << endl;
    if(s.empty()) cout << "Is empty" << endl;
       else  cout << "Is not empty" << endl;
    return 0;
}
```

二、queue 队列

头文件： #include<queue>
实例化： queue<类型[, 存储容器]>QueueName

成员函数：

bool empty();	队列为空返回 **true** 否则返回 **false**.
void pop();	删除队列的一个元素。
void push(**const** TYPE &val);	将 val 元素加入队列。
size_type size();	返当前队列中的元素数目
TYPE &back();	返回一个引用，指向队列的最后一个元素。
TYPE &front();	返回队列第一个元素的引用。

实例程序：

```
#include <iostream>
#include <queue>
#include <algorithm>
using namespace std;
int main()
{   queue<int>s;
    s.push(100);    s.push(200);    s.push(300);
    cout << "Size: " << s.size() << endl;
    cout << "Front: " << s.front() << endl;
    cout << "Back: " << s.back() << endl;
    cout << endl;

    s.pop();
    cout << "Size: " << s.size() << endl;
    cout << "Front: " << s.front() << endl;
    cout << "Back: " << s.back() << endl;
    if(s.empty())  cout << "Is empty" << endl;
        else  cout << "Is not empty" << endl;
    return 0;
}
```

三、Priority Queues 优先队列

类似队列，但是在这个数据结构中的元素默认使用小于号(std::less<typename>)进行部分排序，使得每次出栈的元素都是最小(优先级最高的)元素。可以使用其他的比较方式。

头文件： `#include<queue>`

实例化： `priority_queue<类型[, 存储容器, 比较谓词]>PriorityQueueName`

成员函数：

bool empty();	优先队列为空返回 **true** 否则返回 **false**.
void pop();	删除优先队列中的第一个元素。
void push(**const** TYPE &val);	添加一个元素到优先队列中，值为 val。
size_type size();	返当前队列中的元素数目
TYPE &top ();	返回一个引用，指向最高优先级的元素。

默认的比较方式是使用小于号运算符(<)进行比较，如果是系统提供的能够使用小于号比较的元素类型就可以只写元素类型；如果想用系统提供的大于号进行比较，则还需要给出存储容器和比较谓词；如果使用自定义的 struct/class，则需要重载小于号运算符。举 3 例：

```
priority_queue<int> minIntQ;        //注意 v 下面的空格：v
priority_queue<float, vector<float>, greater<float> > maxFloatQ;
struct node
{   int i;
    bool operator<(const node &a)const{return (i < a.i);}
};
priority_queue<node> minNodeQ;
```

以上三个优先队列出队的元素分别是最小的整数、最大的浮点数、成员 i 最小的 node。

实例程序：

```
#include <iostream>
#include <queue>
#include <algorithm>
using namespace std;
#define pow2(a) ((a)*(a))
#define dist2(x, y) (pow2(x) + pow2(y))
struct coord
{   int x, y;
    const bool operator<(const coord &b)const
    {   return (dist2(x, y) < dist2(b.x, b.y));}
};

struct cmp
{   const bool operator()(const coord &a, const coord &b)
    {       return (dist2(a.x, a.y) < dist2(b.x, b.y));    }
};
int main()
```

```
{   priority_queue<coord> s;
    coord a;
    a.x = 3, a.y = 2;
    s.push(a);
    a.x = 1, a.y = 2;
    s.push(a);
    a.x = 2, a.y = 2;
    s.push(a);
    cout << "Size: " << s.size() << endl;
    cout << "Top: " << s.top().x << ", " << s.top().y << endl;
    s.pop();
    cout << "Size: " << s.size() << endl;
    cout << "Top: " << s.top().x << ", " << s.top().y << endl;
    if(s.empty()) cout << "Is empty" << endl;
          else   cout << "Is not empty" << endl;
    return 0;
}
```

四、Bitset 位集合

顾名思义就是一种位集合的数据结构。Bitsets 使用许多二元操作符，比如逻辑和，或等。

头文件： #include<bitset>

实例化： bitset<unsigned long bits>bs; //容纳 bits 位，不设置初值(全零)
bitset<bits>bs(unsigned long &value); //用 unsigned long 初始化
bitset<bits>bs(const string &str); //用 string 来初始化

成员函数：

构造函数:
bitset();
bitset(unsigned long val);
	以无参的形式创建 bitset，或把一个长无符号整数转为二进制插入到 bitset 中。模板中提供的数字决定 bitset 有多长。

操作符 !=, ==, &=, ^=, |=, ~, <<=, >>=, []	不等，相等，与，异或，或，非，左移，右移(和普通位运算一样)，取某一位(像数组)
bool any();	如果有位被置 1 返回 true，否则返回 false
size_type count();	返回被设置成 1 的位的个数。
bitset &flip(); bitset &flip(size_t pos);	反转 bitset 中所有的位 反转 pos 上的位(pos 是 unsigned long)
bool none();	如果没有位被置 1 返回 true，否则 false
bitset &reset(); bitset &reset(size_t pos);	置零 bitset 中所有的位 置零 pos 上的位(pos 是 unsigned long)
bitset &set(); bitset &set(size_t pos, int val=1);	置 1 bitset 中所有的位 置 1 pos 上的位(pos 是 unsigned long)
bool test(size_t pos);	如果指定位为 1 返回 true，否则 false
string to_string();	转换成 string 返回以便输出，可直接 cout
unsigned long to_ulong();	转换成 unsigned long 返回.

操作符的使用：

bitset<bits>::reference bitset<bits>::operator [ ] (size_t idx)
bool bitset<bits>::operator [ ] (size_t idx) const
第一种形式返回的是对在 idx 位置的那个位的引用，所以可以把它当作左值进行修改。
第二种类型返回的是 bool 类型，输出前需要进行强制类型转换。
istream& operator>> (istream& strm, bitset<bits>& bits)
尽可能多地读入包含 0，1 的字符串，如果长度小于 bitset 的长度，使用前导 0 填充

实例程序：

```
#include <iostream>
#include <bitset>
using namespace std;
int main(){
    bitset<64> bs;
    bs = 1;
```

```
    cout << bs << endl;
    bs ^= 63;
    cout << bs << endl;
    bs >>= 1;
    cout << bs << endl;
    cout << ~bs << endl;
    cout << "1 Bits: " << bs.count() << endl;
    cout << bs[32] << endl;
    bs.set(32);
    cout << bs[32] << endl;
    bs.flip(32);
    cout << bs[32] << endl;
    return 0;
}
```

五、list 列表

头文件： #include<list>
实例化： list<类型>ListName

成员函数：

构造函数(使得可以在定义时赋初值) list (); list (size_type n, **const** TYPE &v) list (const list &from) list (input_iterator start, input_iterator end)	• 无初值 • 给出 n 个初值 • 由另一个 list 初始化 • 由[start, end)区间内的值初始化
void assign(input_iterator start, input_iterator end); **void** assign(size_type num, **const** TYPE &val);	• 清空链表，插入区间[start, end)的内容到 list 中 • 清空链表，插入 num 个值为 val 的元素
TYPE &back()	返回对最后一个元素的引用
TYPE &front()	返回对第一个元素的引用
iterator begin()	返回指向第一个元素的迭代器
iterator end()	返回指向**链表末尾(最后一个元素之后的那个位置)**的迭代器
void clear()	清空链表
bool empty()	如果链表为空返回 **true**，否则返回 **false**
iterator erase(iterator pos); **iterator** erase(iterator start, iterator end);	• 删除 pos 所指元素并返回下一元素迭代器 • 删除[start, end)之间的元素，并返回最后一个被删除元素的下个元素的迭代器
iterator insert(iterator pos, **const** TYPE &val); **void** insert(iterator pos, size_type num, **const** TYPE &val); **void** insert(iterator pos, input_iterator start, input_iterator end);	•插入一个值为 value 的元素在 pos 位置并返回其迭代器，原 pos 及以后的元素后移。 • 插入 num 个值为 value 的元素在 pos 位置，原 pos 及以后元素后移。 • 插入[start, end)之间的元素到 pos 位置，原 pos 及以后元素后移
void merge(list &lst); **void** merge(list &lst, bool Cmpfunc) // bool Cmpfunc(Type &a, Type &b)	将链表 lst 有序地合并到原链表中，默认使用小于号进行比较插入，可指定比较函数 Cmpfunc，对两个 TYPE 类型元素进行比较
void pop_back();	删除链表的最后一个元素。
void pop_front();	删除链表的第一个元素。
void push_back(**const** TYPE &val);	将 val 连接到链表的最后。
void push_front(**const** TYPE &val);	将 val 连接到链表的头部。
void remove(**const** TYPE &val);	删除链表中所有值为 val 的元素。
void remove_if(bool testfunc) // bool testfunc(TYPE &val)	用 testfunc 一元函数来判断是否删除元素如果 testfunc 返回 true 则删除该元素。

size_type size()	返回 list 中元素的数量。
void resize(size_type n, TYPE val = 0)	将链表大小重置为 n，若 n < size()只保留前面 n 个元素，否则最后 size() - n 个元素置为 value(如果不给出，默认为 0)
void reverse();	将链表所有元素倒转。
void sort(); **void** sort(Comp compfunc);	提供 $n\log_2 n$ 的排序效率，默认使用小于号排序，可以自己指定排序函数。
void splice(iterator pos, list &lst) **void** splice(iterator pos, list &lst, iterator del); **void** splice(iterator pos, list &lst, iterator start, iterator end)	• 将 lst 链表插入到这个链表的 pos 位置(pos 及其后元素后移) • 将 lst 链表中 del 所指元素插入到这个链表的 pos 位置 • 将 lst 链表中[start, end)之间的元素插入到这个链表的 pos 位置
void swap(list &lst);	交换两个链表中的元素
void unique();	去除链表中重复元素(离散化)。
reverse_iterator rbegin();	返回一个逆向迭代器，指向链表的末尾
reverse_iterator rend();	返回一个指向开头之前位置的逆向迭代器

实例程序：

```
#include <iostream>
#include <list>
using namespace std;
int main()
{   list<int>lis;
    list<int>::iterator li;
    for (int i = 0; i < 2; ++i)
    {   lis.push_back(i);       lis.push_front(i);    }
    lis.sort();
    cout << "Size: " << lis.size() << endl;
    for (li = lis.begin(); li != lis.end(); li++)
        cout << *li << endl;
    lis.pop_front();
    cout << "Size: " << lis.size() << endl;
    for (li = lis.begin(); li != lis.end(); li++)
        cout << *li << endl;
    lis.pop_back();
    cout << "Size: " << lis.size() << endl;
    for (li = lis.begin(); li != lis.end(); li++)
        cout << *li << endl;
    return 0;
}
```

六、vector 向量

头文件： #include<vector> **实例化：** vector<类型>VectorName

成员函数：

构造函数(使得可以在定义时赋初值) vector(); vector(size_type n, **const** TYPE &v) vector(const vector &from) vector (input_iterator start, input_iterator end)	• 无初值 • 给出 n 个初值 • 由另一个 vector 初始化 • 由[start, end)区间内的值初始化
运算符，包括==, !=, <=, >=, < , >, []	[]运算符使 vector 可以像数组一样操作
void assign(input_iterator start, input_iterator end); **void** assign(size_type num, **const** TYPE &val);	• 清空 vector，插入区间[start, end)的内容到 list 中 • 清空 vector，插入 num 个值为 val 的元素
TYPE at(size_type loc)	返回在 loc 位置元素的值的引用，有左值
TYPE &back()	返回对最后一个元素的引用
TYPE &front()	返回对第一个元素的引用
iterator begin()	返回指向第一个元素的迭代器
iterator end()	返回指向 **vector 末尾(最后一个元素之后的那个位置)** 的迭代器
void clear()	清空 vector（未回收空间!!!）
bool empty()	如果为空返回 **true**，否则返回 **false**
iterator erase(iterator loc); **iterator** erase(iterator start, iterator end);	• 删除 loc 所指元素并返回下一元素迭代器 • 删除[start, end)之间的元素，并返回最后一个被删除元素的下个元素的迭代器
iterator insert(iterator loc, **const** TYPE &val); **void** insert(iterator loc, size_type num, **const** TYPE &val); **void** insert(iterator loc, input_iterator start, input_iterator end);	•插入一个值为 value 的元素在 loc 位置并返回其迭代器，原 loc 及以后的元素后移。 • 插入 num 个值为 value 的元素在 loc 位置，原 loc 及以后元素后移。 • 插入[start, end)之间的元素到 loc 位置，原 loc 及以后元素后移
void pop_back();	删除 vector 的最后一个元素。
void push_back(**const** TYPE &val);	将 val 放置到 vector 的最后。
void reserve(size_type size)	预留**至少**共容纳 size 个元素的空间
size_type size()	返回 list 中元素的数量。
void resize(size_type n, TYPE val = 0)	将 vector 大小重置为 n，若 n < size()只保留前面 n 个元素，否则最后 size() - n 个元素置为 value(如果不给出，默认为 0)
void swap(vector &from);	交换两个数组中的元素
reverse_iterator rbegin();	返回一个逆向迭代器，指向链表的末尾
reverse_iterator rend();	返回一个指向开头之前位置的逆向迭代器

对 vector 进行排序可以使用 STL 的 sort, stable_sort, partition, partial_sort, nth_element, 可以用 STL 的 unique 算法对其进行排重，但是一定要这么写：

```
vt.erase(unique(vt.begin(), vt.end()), vt.end());
```

使用 STL 的 remove 或 remove_if 算法删除指定元素：

```
vt.erase(remove(vt.begin(), vt.end(), Type &value), vt.end());
vt.erase(remove_if(vt.begin(), vt.end(), testfunc), vt.end());
```

回收 vector 占用的空间：

```
vector<TYPE>(vt).swap(vt); //回收 vt 中多余元素占用的空间
vector<TYPE>().swap(vt); //回收 vt 占用的所有空间
```

也就是创建一个匿名的空的 vector<TYPE>类型变量(前一句还执行了用 vt 对其初始化)并与 vt 交换，然后这个变量在这条语句结束时被自动释放。

实例程序：

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main()
{   vector<int> a;
    for (int i = 0; i < 5; ++i)
        a.push_back(5 - i);
    cout << "Size: " << a.size() << endl;
    a.pop_back();
    a[0] = 1;
    cout << "Size: " << a.size() << endl;
    for (int i = 0; i < (int)a.size(); ++i)
        cout << a[i] << ", " << endl;
    cout << endl;
    sort(a.begin(), a.end());
    cout << "Size: " << a.size() << endl;
    for (int i = 0; i < (int)a.size(); ++i)
        cout << a[i] << ", " << endl;
    cout << endl;
    a.clear();
    cout << "Size: " << a.size() << endl;
    return 0;
}
```

七、map / multimap 映射与多重映射

红黑树实现的关联式容器，包含“关键字/值”对，在插入时默认使用小于号进行比较，在对数时间内插入到相应位置。可以指定比较谓词(使用 std 中的 less/greater 或仿函数)。Map 中保存的元素的键值是唯一的，而 multimap 可以有键值重复的元素存在。

头文件： #include<map>

原型：

```
namespace std {
    template <class Key, class T, class Compare = less<Key>,
              class Allocator = allocator<pair<const Key,T> > >
    class map;
    template <class Key, class T, class Compare = less<Key>,
              class Allocator = allocator<pair<const Key,T> > >
    class multimap;
}
```

这里解释一下“关键字/值”对：在 std 里面定义的一个结构体，有 first 和 second 两个不同类型的成员，原型如下：

```
namespace std{
    template <class T1, class T2>;
    struct pair{
        T1 first;
        T2 second;
        … //other functions
    }
}
```

由于 pair 是个结构体，而 map/multimap 里面存放的是 pair，所以对于这两个容器中的元素，需要使用 element.first 和 element.second 来访问其中的值。在插入的时候可以使用 pair<keytype, valuetype>(key, value)产生一个临时变量来插入，或者使用 make_pair(key, value)来产生，但是要严格注意 key 和 value 的类型，比如 make_pair(1, 0)是不等于 make_pair(1, 0.0)的。

实例化(map)： map<key 类型, value 类型>MapName;

实例化(multimap)： multimap<key 类型, value 类型>MultiMapName;

map 成员函数：

构造函数(使得可以在定义时赋初值) map(); map(const map &from) map (input_iterator start, input_iterator end) 对应 multimap 也有这三个构造函数	• 无初值 • 由另一个 map 初始化 • 由[start, end)区间内的值初始化
运算符，包括==, !=, <=, >=, < , >, []	[]运算符使 map 可像数组一样操作 (multimap 不行，因为可以有重复元素)
iterator begin()	返回指向第一个元素的迭代器
iterator end()	返回指向**末尾(最后一个元素之后的那个位**

	置) 的迭代器
void clear()	清空容器。
bool empty()	如果为空返回 **true**，否则返回 **false**
insert(pair<keytype, valuetype> &val); (上面的 insert 对 multimap 无返回值) **iterator** insert(iterator loc, pair<keytype, valuetype> &val); **void** insert(input_iterator start, input_iterator end);	• 插入一个 pair 类型的元素，对 map 返回一个 pair，其 first 是指向插入元素的迭代器，second 表示是否插入成功 • 从 loc 开始寻找一个可以插入值为 value 的元素的位置将其插入并返回其迭代器 • 插入[start, end)之间的元素到容器中
void erase(iterator loc) **void** erase(iterator start, iterator end) **size_type** erase(const key_type &key)	• 删除 loc 所指元素 • 删除[start, end)之间的元素 •删除 **key 值为 value** 的元素并返回被删除元素的个数
pair<iterator start, iterator end> **equal_range**(const key_type &key)	查找**键值等于 key** 的元素区间为[start, end)，指示范围的两个迭代器以 pair 返回
iterator find(const key_type &key)	返回一个迭代器指向**键值为 key 的元素**，如果没找到就返回 end()
size_type count(const KeyType &key)	返回**键值等于 key** 的元素的个数
iterator lower_bound(const key_type &key);	返回一个迭代器，指向**键值 >= key** 的第一个元素。
iterator upper_bound(const key_type &key);	返回一个迭代器，指向**键值 > key** 的第一个元素。
size_type size()	返回元素的数量。
void swap(map &from);	交换两个 map 中的元素
key_compare key_comp();	返回一个比较 key 的函数。
value_compare value _comp();	返回一个比较 value 的函数。
reverse_iterator rbegin();	返回一个逆向迭代器，指向链表的末尾
reverse_iterator rend();	返回一个指向开头之前位置的逆向迭代器

默认的比较方式是使用小于号运算符(<)进行比较，如果是系统提供的能够使用小于号比较的元素类型就可以只写元素类型；如果想用系统提供的大于号进行比较，则还需要给出比较谓词（std 中的 less/greater 或者自定义的仿函数）；如果使用自定义的 struct/class，则需要重载小于号运算符。举 3 例：

```
map<int, int> minIntMap;        //注意 v 下面的空格：v
map<float, int, greater<float> > maxFloatMap;
struct node{
    int i;
    bool operator<(const node &a)const{return (i < a.i);}
};
multimap<node, int> minNodeMap;
```

以上三个 map/multimap 的元素分别按照整数升序、浮点降序、node.i 升序排列的。

Map 实例程序：

```
#include <iostream>
#include <map>
#include <algorithm>
using namespace std;
struct T1
{   int v;
    bool operator<(const T1 &a)const
    {       return (v < a.v);    }
};

struct T2{    int v;};
struct cmp
{   const bool operator()(const T2 &a, const T2 &b)
    {       return (a.v < b.v);    }
};

int main()
{   map<T1, int>mt1; //example for user-defined class
    map<T2, int, cmp>mt2; //example for user-defined class(functor)
    map<string, int> m2;
    map<string, int>::iterator m2i, p1, p2;
    //map<string, int, greater<string> >m2;
    //map<string, int, greater<string> >::iterator m2i, p1, p2;
    m2["abd"] = 2;    m2["abc"] = 1;    m2["cba"] = 2;
    m2.insert(make_pair("aaa", 9));
    m2["abf"] = 4;    m2["abe"] = 2;
    cout << m2["abc"] << endl;

    m2i = m2.find("cba");
    if(m2i != m2.end())
        cout << m2i->first << ": " << m2i->second << endl;
    else    cout << "find nothing" << endl;

    cout << "Iterate" << endl;
    for(m2i = m2.begin(); m2i != m2.end(); m2i++)
        cout << m2i->first << ": " << m2i->second << endl;
    return 0;
}
```

MultiMap 实例程序：

```
#include <iostream>
#include <map>
#include <algorithm>
using namespace std;
int main()
{   multimap<string, int> mm1;
    multimap<string, int>::iterator mm1i, p1, p2;
    mm1.insert(make_pair("b", 3));
    mm1.insert(make_pair("a", 0));
    mm1.insert(make_pair("b", 5));
    mm1.insert(make_pair("c", 4));
    mm1.insert(make_pair("b", 2));
    cout << "Size: " << mm1.size() << endl;
    for(mm1i = mm1.begin(); mm1i != mm1.end(); mm1i++)
        cout << mm1i->first << ": " << mm1i->second << endl;

    cout << "COUNT: " << mm1.count("b") << endl;
    cout << "Bound: " << endl;
    p1 = mm1.lower_bound("b");
    p2 = mm1.upper_bound("b");
    for(mm1i = p1; mm1i != p2; mm1i++)
        cout << mm1i->first << ": " << mm1i->second << endl;

    return 0;
}
```

八、set / multiset 集合与多重集合

头文件： #include<set>
实例化(set)： set<类型>SetName
实例化(multiset)： multiset<类型>SetName

成员函数：

构造函数(使得可以在定义时赋初值) set(); set(const set &from) set (input_iterator start, input_iterator end) 对应 multiset 也有这三个构造函数	• 无初值 • 由另一个 set 初始化 • 由[start, end)区间内的值初始化
运算符，包括==, !=, <=, >=, < , >	
iterator begin()	返回指向第一个元素的迭代器
iterator end()	返回指向**末尾(最后一个元素之后的那个位置)**的迭代器
void clear()	清空容器。
bool empty()	如果为空返回 **true**，否则返回 **false**
iterator insert(TYPE &val); (上面的 insert 对 multiset 无返回值) **iterator** insert(iterator loc, TYPE &val) **void** insert(input_iterator start, input_iterator end);	• 插入一个元素，对于 set 返回一个 pair 分别是插入元素的迭代器和是否插入成功 • 从 loc 开始寻找一个可以插入值为 value 的元素的位置将其插入并返回其迭代器 • 插入[start, end)之间的元素到容器中
void erase(iterator loc) **void** erase(iterator start, iterator end) **size_type** erase(const key_type &key)	• 删除 loc 所指元素 • 删除[start, end)之间的元素 •删除 key 值为 value 的元素并返回被删除元素的个数
pair<iterator start, iterator end> equal_range(const key_type &key)	查找 multiset 中**键值等于 key** 的所有元素，返回指示范围的两个迭代器以 pair 返回
size_type count(const KeyType &key)	查找容器中值为 key 的元素的个数
iterator find(const key_type &key)	返回一个迭代器指向**键值为 key 的元素**，如果没找到就返回 end()
iterator lower_bound(const key_type &key);	返回一个迭代器，指向**键值 >= key** 的第一个元素。
iterator upper_bound(const key_type &key);	返回一个迭代器，指向**键值 > key** 的第一个元素。
size_type size()	返回元素的数量。
void swap(vector &from);	交换两个链表中的元素
key_compare key_comp();	返回一个比较 key 的函数。
value_compare value _comp();	返回一个比较 value 的函数。

reverse_iterator rbegin();	返回一个逆向迭代器，指向链表的末尾
reverse_iterator rend();	返回一个指向开头之前位置的逆向迭代器

Set 实例程序：

```
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
struct T1
{   int key,value1, value2;
    bool operator<(const T1 &b)const
    {     return (key < b.key);    }
};
struct T2
{   int key,v1, v2;  };
struct T2cmp
{   bool operator()(const T2 &a, const T2 &b)
    {       return (a.key < b.key);    }
};

int main()
{   set<T1> s2;    set<T2, T2cmp> s3;
    #if 1
       set<string>s1;
       set<string>::iterator iter1;
    #else
       set<string, greater<string> >s1;
       set<string, greater<string> >::iterator iter1;
    #endif
    s1.insert("abc");    s1.insert("abc");    s1.insert("abc");
    s1.insert("bca");    s1.insert("aaa");
    cout << "ITERATE:" << endl;
    for (iter1 = s1.begin(); iter1 != s1.end(); iter1++)
       cout << (*iter1) << endl;
    cout << "FIND:" << endl;
    iter1 = s1.find("abc");
    if(iter1 != s1.end())   cout << *iter1 << endl;
       else       cout << "NOT FOUND" << endl;
      return 0;
}
```

MultiSet 实例程序：

```
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
struct T1
{   int key,value1, value2;
    bool operator<(const T1 &b)const
    {       return (key < b.key);    }
};
struct T2
{   int key,v1, v2; };
struct T2cmp
{   bool operator()(const T2 &a, const T2 &b)
    {       return (a.key < b.key);    }
};

int main()
{   multiset<T1> s2;
    multiset<T2, T2cmp> s3;
    #if 1
        multiset<string>s1;
        multiset<string>::iterator iter1;
    #else
        multiset<string, greater<string> >s1;
        multiset<string, greater<string> >::iterator iter1;
    #endif
    s1.insert("abc");    s1.insert("abc");    s1.insert("abc");
    s1.insert("bca");    s1.insert("aaa");
    cout << "ITERATE:" << endl;
    for (iter1 = s1.begin(); iter1 != s1.end(); iter1++)
        cout << (*iter1) << endl;
    cout << "FIND:" << endl;    iter1 = s1.find("abc");
    if(iter1 != s1.end())    cout << *iter1 << endl;
        else    cout << "NOT FOUND" << endl;
    cout << "COUNT: " << s1.count("abc") << endl;
    cout << "BOUND: " << endl;
    multiset<string>::iterator s1i, p1, p2;
    p1 = s1.lower_bound("abc");    p2 = s1.upper_bound("abc");
    for(s1i = p1; s1i != p2; s1i++)
        cout << (*s1i) << endl;
    return 0;
}
```

九、deque (Double Ended Queue) 双端队列

deque 和 vector 很相似,但是它允许在容器头部快速插入和删除(就像在尾部一样),并提供对其内部元素随机访问的能力(但速度稍慢于 vector)。

头文件: #include<deque>

实例化: deque<类型>DequeName

成员函数:

构造函数(使得可以在定义时赋初值) deque (); deque (size_type n, **const** TYPE &v) deque (const deque &from) deque (input_iterator start, input_iterator end)	• 无初值 • 给出 n 个初值 • 由另一个 vector 初始化 • 由[start, end)区间内的值初始化
运算符[]	[]运算符使 deque 可以像数组一样操作
void assign(input_iterator start, input_iterator end); **void** assign(size_type num, **const** TYPE &val);	• 清空 vector,插入区间[start, end)的内容到 list 中 • 清空 vector,插入 num 个值为 val 的元素
TYPE at(size_type loc)	返回在 loc 位置元素的值的引用,有左值
TYPE &back()	返回对最后一个元素的引用
TYPE &front()	返回对第一个元素的引用
iterator begin()	返回指向第一个元素的迭代器
iterator end()	返回指向 **deque 末尾(最后一个元素之后的那个位置)** 的迭代器
void clear()	清空 **deque,但是不释放空间**。
bool empty()	如果为空返回 **true**, 否则返回 **false**
iterator erase(iterator loc); **iterator** erase(iterator start, iterator end);	• 删除 loc 所指元素并返回下一元素迭代器 • 删除[start, end)之间的元素,并返回最后一个被删除元素的下个元素的迭代器
iterator insert(iterator loc, **const** TYPE &value); **void** insert(iterator loc, size_type num, **const** TYPE &val); **void** insert(iterator loc, input_iterator start, input_iterator end);	•插入一个值为 value 的元素在 loc 位置并返回其迭代器,原 loc 及以后的元素后移。 • 插入 num 个值为 value 的元素在 loc 位置,原 loc 及以后元素后移。 • 插入[start, end)之间的元素到 loc 位置,原 loc 及以后元素后移
void pop_front();	删除 **deque** 的第一个元素。
void push_front(**const** TYPE &val);	将 val 放置到 **deque** 的开头。
void pop_back();	删除 **deque** 的最后一个元素。
void push_back(**const** TYPE &val);	将 val 放置到 **deque** 的最后。
size_type size()	返回 list 中元素的数量。
void resize(size_type n,	将 **deque** 大小重置为 n, 若 n < size()只

TYPE val = 0)	保留前面 n 个元素，否则最后 size() - n 个元素置为 value(如果不给出，默认为 0)
void swap(vector &from);	交换两个链表中的元素
reverse_iterator rbegin();	返回一个逆向迭代器，指向链表的末尾
reverse_iterator rend();	返回一个指向开头之前位置的逆向迭代器

对 deque 进行排序可以使用 STL 的 sort, stable_sort, partition, partial_sort, nth_element, 可以用 STL 的 unique 算法对其进行排重，但是一定要这么写：

que.erase(unique(vt.begin(),que.end()),que.end());

使用 STL 的 remove 或 remove_if 算法删除指定元素：

```
que.erase(remove(que.begin(),que.end(), Type &value), que.end());
que.erase(remove_if(que.begin(),que.end(), testfunc), que.end());
```

实例程序：

```
#include <iostream>
#include <algorithm>
#include <deque>
using namespace std;
int main()
{   deque<int> a;
    for (int i = 0; i < 5; ++i)
        a.push_back(5 - i);
    cout << "Size: " << a.size() << endl;
    a.push_front(0);

    cout << "Size: " << a.size() << endl;
    for (int i = 0; i < (int)a.size(); ++i)
        cout << a[i] << ", " << endl;
    cout << endl;

    sort(a.begin(), a.end());
    cout << "Size: " << a.size() << endl;
    for (int i = 0; i < (int)a.size(); ++i)
        cout << a[i] << ", " << endl;
    cout << endl;

    a.clear();
    cout << "Size: " << a.size() << endl;
    return 0;
}
```

十、string 字符串

头文件：#include<string> //默认情况下包含了 iostream 就可以用了
实例化：string StringName;

成员函数：

构造函数
string();
string(const string& s);
string(size_type length, const char& ch)
string(const char* str);
string(const char* str, size_type length);
string(const string& str, size_type index, size_type length);
string(input_iterator start, input_iterator end);
	
	• 无操作
	• 使用一个 string 来初始化
	• 使用 length 个字符 ch 来初始化
	
	•使用一个字符数组初始化(ASCII0 结束)
	•用一个字符数组最多前 length 个字符初始化
	• 用一个 string 从 index 开始的最多 length 个字符初始化
	
	• 用[start, end)之间的元素来初始化

=, ==, !=, +, +=, <, <=, >, >=, []	[]可以像字符数组一样随机读取和写入

string& append(const string& str);
string& append(const char* str);
string& append(const string& str, size_type index, size_type len);
string& append(const char* str, size_type num);
string& append(size_type num, char ch);
string& append(input_iterator start, input_iterator end);
	• 末尾追加一个 string
	
	• 追加一个字符数组
	• 追加一个 string 从 index 开始的最多 len 个字符
	• 追加一个字符数组最多 num 个字符
	
	• 追加 num 个字符 ch
	
	• 追加[start, end)之间的元素

void assign(size_type num, const char& val);
void assign(input_iterator start, input_iterator end);
string& assign(const string& str);
string& assign(const char* str);
string& assign(const char* str, size_type num);
string& assign(const string& str,
	• 赋值，num 个字符 val
	
	• 赋值，[start, end)之间的元素
	
	• 赋值，string str 的内容
	
	• 赋值，用字符数组
	• 赋值，用字符数组最多 num 个字符
	
	• 赋值，用 string 从 index 开始的最多

size_type index, size_type len);	len 个字符
TYPE& at(size_type loc);	返回在指定位置 loc 的字符以读取或写入
const_iterator begin() const;	返回指向头部的迭代器
const char * c_str();	返回一个标准 c 字符串，**但是不允许修改**，否则会破坏 string 的内部结构
size_type capacity() const;	返回已分配空间可容纳的最大字符数
void clear()	清空 string，但是不回收空间
int compare(string a, string b)	比较两个字符串，a<b 返回负数，a==b 返回零，a>b 返回正数，可以使用字符数组。
size_type copy(char* str, size_type num, size_type index = 0); //**注意**！调用这个函数的时候会自动调用 memset(str, NULL, sizeof(str));	将 string 中从 index(默认为 0)开始的最多 num 个字符 copy 到字符数组 str 中，返回 copy 的字符数。**不建议使用。**
const char *data();	返回指向第一个字符的指针(不要修改!)
bool empty() const;	返回 true 如果字符串长为 0
const_iterator end() const;	返回指向最后一个字符下一位置的迭代器

iterator erase(iterator loc);
iterator erase(iterator start, iterator end);
string& erase(size_type index = 0, size_type num = npos);
	• 删除 loc 位置的字符
	• 删除[start,end)之间的字符
	• 删除从 index 开始的 num 个字符，返回 *this

size_type find(const string& str, size_type index);
size_type find(const char* str, size_type index);
size_type find(const char* str, size_type index, size_type length);
size_type find(char ch, size_type index);
	• 返回从 index 开始 str 第一次出现的位置，找不到就返回 string::npos(常量)
	• 返回从 index 开始 str 第一次出现的位置，找不到就返回 string::npos(常量)
	• 返回从 index 开始 str 前 length 个字符第一次出现的位置，找不到就返回 string::npos(常量)
	• 返回从 index 开始，字符 ch 第一次出现的位置，找不到就返回 string::npos

find_first_not_of	格式同 find，返回第一个不是给定的字符串串中字符的位置
find_first_of	格式同 find，返回第一个是给定的字符串串中字符的位置

find_last_not_of	格式同 find，反向查找，返回第一个不是给定的字符串串中字符的位置
find_last_of	格式同同 find，反向查找，返回第一个是给定的字符串串中字符的位置
istream& getline(istream& is, string& s, char delimiter = '\n');	从输入流 is 中读入一些字符到 str 中，以 delimiter(默认为'\n')结束

iterator insert(iterator i, const char& ch);
string& insert(size_type index, const string& str);
string& insert(size_type index, const char* str);
string& insert(size_type index1, const string& str, size_type index2, size_type num);
string& insert(size_type index, const char* str, size_type num);
string& insert(size_type index, size_type num, char ch);
void insert(iterator i, size_type num, const char& ch);
void insert(iterator i, iterator start, iterator end);
	•在迭代器 i 指向的位置插入一个字符 ch
	•在位置 index 插入一个 string
	•在位置 index 插入一个 C 字符串
	•在 index1 位置插入 string 从 index2 开始的最多 num 个字符
	•在 index 位置插入 C 字符串的最多 num 个字符
	•在 index 位置插入 num 个字符 ch
	•在迭代器 i 指向的位置插入 num 个字符 ch
	•在迭代器 i 指向的位置插入[start, end)之间的字符。

size_type length()	返回 string 的长度(和 size()一样)
size_type max_size()	返回字符串最大能容纳的字符数
void push_back(char &c)	插入字符 c 到 string 的末尾
reverse_iterator rbegin()	返回指向最后一个字符的反向迭代器
reverse_iterator rend()	返回指向第一个字符之前位置反向迭代器

string& replace(size_type index, size_type num, const string& str);
string& replace(size_type index1, size_type num1, const string& str, size_type index2, size_type num2);
string& replace(size_type index, size_type num, const char* str);
string& replace(size_type index, size_type num1, const char* str, size_type num2);
	•替换(从 index 开始最多 num 个字符)为(string str)
	•替换从(index1 开始最多 num1 个字符)为(string str 从 index2 开始的最多 num2 个字符)
	•替换(从 index 开始的最多 num 个字符)为(C 字符串 str)
	•替换(从 index 开始最多 num1 个字符)为(C 字符串 str 中的 num2 个字符，有可能越界)

string& replace(size_type index, size_type num1, size_type num2, char ch);
string& replace(iterator start, iterator end, const string& str);
string& replace(iterator start, iterator end, const char* str);
string& replace(iterator start, iterator end, const char* str, size_type num);
string& replace(iterator start, iterator end, size_type num, char ch);
	• 替换(从 index 开始最多 num1 个字符)为(num2 个字符 ch)
	• 替换(本串中[start, nd)之间字符)为(string str)
	• 替换(本串中[start,end)之间字符)为（C 字符串 str)
	• 替换(本串中[start,end)之间字符)为(c 字符串 str 中最多 num 个字符,会越界)
	• 替换(本串中[start,end)之间字符)为 num 个字符 ch

void reserve(size_type size);	为 string 开辟至少能寸下 size 个字符的内存，但不会删除现有字符
void resize(size_type size, const TYPE& val = TYPE());	改变 string 的字符数为 size 个，如果 size<size()删除多余的(但不回收空间)； size>size()后面的填充为 val
rfind	格式同 find，逆向查找
size_type size();	返回字符串中现有字符的个数
string substr(size_type index, size_type length = npos);	返回包含从 index 的最多 length 个字符的 string，不指定 length 就到结尾
void swap(string &str)	和 str 交换内容

string::npos

string 类的常量，如果查找时返回这个值表示没有查找成功。

回收 str 占用的空间

```
string (str).swap(str); //保留现有有效元素
string ().swap(str); //删除所有元素
```

删除排序后的重复字符

```
str.erase(unique(str.begin(), str.end()), str.end());
```

输出字符串，不用 cout

```
printf("%s", str.c_str());
```

string 的忽略大小写排序仿函数：

```
struct stringcmp{
    bool operator()(const string &a, const string &b){
        for(unsigned int i=0; i<a.size() && i<b.size(); i++){
            if(toupper(a[i]) < toupper(b[i])){ return true; }
            if(toupper(a[i]) > toupper(b[i])){ return false; }
        }
        return a.size() < b.size();
```

```
        }
    };
    set<string, stringcmp>ignoreCaseStringSet;
map<string, value_type, stringcmp>ignoreCaseStringMap;
```

实例程序：

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
int main()
{   string t1, t2("a");
    t1 = "b";
    cout << "t1: " << t1 << ", t2: " << t2 << endl;
    cout << "t2[0] = " << t2[0] << endl;
    t1 += "def";
    cout << "t1 = " << t1 << endl;

    if(t1 == t2)     cout << "t1 == t2" << endl;
       else      cout << "t1 != t2" << endl;
    printf("t1 = %s\n", t1.c_str());

    unsigned idx = t1.find("de");
    if(idx != string::npos)   cout << "find @ index " << idx << endl;
       else    cout << "not found " << endl;

    string t3 = "abcd";
    t3.replace(1, 2, "ooxx");
    cout << "t3: " << t3 << ", size = " << t3.size() << endl;
    return 0;
}
```

十一、常用算法调用

1. for_each

```
void foreach(iterator begin, iterator end, class func);
```
将[begin, end)上的每个元素传给只有一个参数(一元)的 func 函数/仿函数进行处理。

例：
```
void neg(int &i){ i = -i; }
void f(){
    int nums[5] = {1, 2, 3, 4, 5);
    vector<int>a(num, num+5);
    for_each(a.begin(), a.end(), neg);//a 中元素变为其相反数
}
```
注意：如果需要对元素进行改动，定义的函数和仿函数中需要使用引用传参。

2. min_element / max_element

```
iterator min_element(iterator begin, iterator end);
iterator min_element(iterator begin, iterator end, Cmpfunc func);
iterator max_element(iterator begin, iterator end);
iterator max_element(iterator begin, iterator end, Cmpfunc func);
```
返回区间[being, end)之间的最小/最大元素，可以提供比较函数/仿函数(二元)，返回一个指向该元素的迭代器(对于数组则返回相应的指针)。

3. copy / copy_n /copy_backward

```
iterator copy(iterator begin, iterator end, iterator to);
iterator copy_n(iterator begin, size_t n, iterator to);
```
copy 把[begin, end)之间的元素拷贝到从 to 开始的一段区间，拷贝完毕后返回目标区间的结尾(最后一个拷贝位置的下一个位置)。copy_n 把从 begin 开始的 n 个元素拷贝到从 to 开始的位置,返回目标区间的结尾(同 copy)。iterator 和可以是普通的 iterator 或者 reverse_iterator, 也可以是数组的指针。

例：
```
int a[3] = {1, 2, 3}, b[3];
copy(a, a+3, b); //把区间[a, a+3)的元素拷贝到从 b 开始的等长区间
copy_n(a, 3, b); //把从 a 开始的 3 个元素拷贝到从 b 开始的等长区间
```

4. fill / fill_n

```
void fill(iterator first, iterator last, const T& value);
```
将[first, end)之间的元素赋值为 value
```
iterator fill_n(iterator first, Size n, const T& value);
```
将从 first 开始的 n 个元素赋值为 value
```
int a[10];
```
例：
```
    fill(a, a+10, 0); //用 0 填充区间[a, a+10)
    fill(a, 10, 0); //用 0 填充从 a 开始的 10 个元素
```

实例程序：
```
#include <iostream>
using namespace std;
int main()
{   int a[10];
    fill(a, a + 10, 1);
    for (int i = 0; i < 10; ++i)     printf("%d ", a[i]);
    printf("\n");
    fill_n(a, 10, 2);
    for (int i = 0; i < 10; ++i)      printf("%d ", a[i]);
    printf("\n");

    string b[10];
    fill(b, b + 10, "a");
    for (int i = 0; i < 10; ++i)   printf("%s\n", b[i].c_str());
    printf("\n");

    return 0;
}
```

5. remove / remove_if

```
iterator remove(iterator begin, iterator end, const T &value)
iterator remove_if(iterator begin, iterator end, function test)
```
按区间中原有元素的相对次序将不需要移除的元素提前，覆盖需要被删除的元素(指定值，或是(一元判断函数 test)返回 true 的值)，返回新的结尾。它不删除新的结尾和旧的结尾之间的元素，所以一般是结合 erase 使用：
例：
```
vector<int>vt;
for( i=0; i<10; i++) vt.push_back(i);
vt.erase(remove(vt.begin(), vt.end(), 3), vt.end());
```

实例程序：

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

using namespace std;
bool test(int i){    return (i < 5);    }

int main()
{   int a[5] = {1,2,1,3,1};
    int *e = remove(a, a + 5, 1);
    cout << "END: " << (e - a) << endl;
    for (int i = 0; i < 5; ++i)     cout << a[i] << " ";
    cout << endl;
    cout << "Left: " << endl;
    for(int *s = a; s < e; s++) cout << *s << " ";
    cout << endl;

    int d[10] = {1,3,5,7,9,2,4,6,8,10};
    e = remove_if(d, d + 10, test);
    cout << "END: " << (e - d) << endl;
    for (int i = 0; i < 10; ++i)    cout << d[i] << " ";
    cout << endl;
    cout << "Left: " << endl;
    for(int *s = d; s < e; s++) cout << *s << " ";
    cout << endl;

    int b[7] = {1,2,1,3,1,4,1};
    vector<int>v;
    vector<int>::iterator vi;
    copy(b, b + 7, back_inserter(v));
    vi = remove(v.begin(), v.end(), 1);
    //v.erase(remove(v.begin(), v.end(), 1), v.end());
    cout << "Size: " << v.size() << endl;
    for (int i = 0; i < (int)v.size(); ++i)  cout << v[i] << " ";
    return 0;
}
```

6. unique

`iterator unique (iterator begin, iterator end[, Cmpfunc cmp])`
按区间中原有元素的相对次序将多余的重复元素用其后的有效元素覆盖，返回新的结尾。如果区间是将(二元比较函数 cmp)传给 sort 函数排序的，那么需要将(cmp 函数)传给 unique 才可以得到想要的结果。和 remove 一样 unique 并不真的移除新的结尾和旧的

结尾之间的元素，所以一般也需要结合 erase 使用。例：

```
vector<int>vt;
for( i=0; i<10; i++) {vt.push_back(i); vt.push_back(i+1); }
vt.erase(unique(vt.begin(), vt.end()), vt.end());
```

实例程序：

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{  int a[7] = {1,1,1,1,2,2,3};
   int *e = unique(a, a + 7);
   cout << "END: " << (e - a) << endl;
   for (int i = 0; i < 7; ++i)  cout << a[i] << endl;

   int b[7] = {1,1,1,1,2,2,3};
   vector<int>v;
   copy(b, b + 7, back_inserter(v));
   unique(v.begin(), v.end());
      //v.erase(unique(v.begin(), v.end()), v.end());
   cout << "Size: " << v.size() << endl;
   for (int i = 0; i < (int)v.size(); ++i) cout << v[i] << endl;
   return 0;
}
```

7. rotate

```
iterator rotate(iterator begin, iterator middle, iterator end);
```

将[begin, middle)和[middle, end)两个区间内的元素互换。最多只需要 end-begin 次交换。例：

```
char alpha[] = "abcdefghijklmnopqrstuvwxyz";
rotate(alpha, alpha + 13, alpha + 26);
printf("%s\n", alpha); // 输出为 nopqrstuvwxyzabcdefghijklm
```

8. random_shuffle

```
random_shuffle (iterator begin, iterator end)
```

将[begin, end)之间的元素随机重排列,例：

```
const int N = 8;
int A[] = {1, 2, 3, 4, 5, 6, 7, 8};
random_shuffle(A, A + N);
copy(A, A + N, ostream_iterator<int>(cout, " "));
// 输出可能为 7 1 6 3 2 5 4 8, 或其它 40,319 中排列中的任意一种
```

9. partition / stable_partition

```
iterator partition (iterator begin, iterator end, function test)
iterator stable_partition (iterator begin, iterator end, function
    test)
```

把符合条件(一元判断函数 test)的元素移动到区间的前面，不符合条件的移动到后面，返回指向第一个不符合条件元素的迭代器 mid，则符合条件的元素在[begin, mid)之间，不符合条件的元素在[mid, end)之间。注意，partition 是不稳定的，stable_partition 是稳定的。例：

```
bool test(int i){  return i > 5;}
void f()
{   int b[10] = {0, 5, 2, 6, 7, 3, 9, 1, 8, 4};
    vector<int>a.assign(b, b+10);
    vector<int>::iterator p;
    p = stable_partition(a.begin(), a.end(), test);
    copy(a.begin(), p, ostream_iterator<int>(cout, " "));
    cout << endl;
    copy(p, a.end(), ostream_iterator<int>(cout, " "));
} //大于 5 的元素先输出，换行，输出小于等于 5 的元素
```

10. sort / stable_sort

```
void sort(iterator begin, iterator end[, Cmpfunc func])
void stable_sort(iterator begin, iterator end[, Cmpfunc func])
```

将[begin, end)之间的元素使用默认的(operator<)或给定的(二元比较函数 func)进行排序，sort 使用随机化快速排序，不稳定；stable_sort 使用归并排序，稳定。例：

```
int a[10] = {0, 5, 2, 6, 7, 3, 9, 1, 8, 4};
sort(a, a+10);
copy(a, a + 10, ostream_iterator<int>(cout, ", "));
//输出 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

实例程序：

```
#include <iostream>
using namespace std;
int main()
{   int a[10] = {1,3,5,2,4,6,7,5,2,5};
    sort(a, a + 10);
    for (int i = 0; i < 10; ++i)   printf("%d  ", a[i]);
```

```
    printf("\n");

    int *s, *e;
    s = lower_bound(a, a + 10, 5); e = upper_bound(a, a + 10, 5);
    printf("[%d, %d)\n", s - a, e - a);
    while(s != e)     printf("%d ", *s++);
    return 0;
}
```

11. partial_sort

```
    void partial_sort( iterator begin, iterator middle, iterator end[,
        Cmpfunc cmp]);
```

将区间[begin, end)之间的元素部分排序，使得在[begin, middle)之间的元素是最小的(middle-begin)个元素，且是升序排列的(默认使用 operator<排序，或使用提供的 cmp 函数排序)。[middle, end)之间的元素是乱序的。

12. nth_element

```
    void nth_element( iterator begin, iterator middle, iterator end[,
         Cmpfunc cmp]);
```

假设 m 是排序后第(middle-begin)个元素,那么 nth_element 将 m 放置在 middle 位置，排在 m 前面的元素在[begin, middle)，排在 m 后面的元素在[middle, end)。两个区间都是无序的。例：

```
    int a[10] = { 2, 0, 5, 6, 7, 1, 9, 3, 8, 4};
    nth_element(a, a + 2, a + 10);
    copy(a, a + 10, ostream_iterator<int>(cout, " "));
    //输出 1 0 2 3 4 7 9 6 8 5
```

13. lower_bound / upper_bound //要求区间有序

```
  iterator lower_bound(iterator begin, iterator end, Const Type
       &value[, Cmpfunc func])
  iterator upper_bound(iterator begin, iterator end, Const Type
       &value[, Cmpfunc func])
```

函数在**有序区间**中查找，lower_bound 返回指向区间中>=value 的第一个元素的迭代器，upper_bound 返回区间中>value 的第一个元素的迭代器。如果有序区间排序时（比如 sort）使用的是 func 函数进行比较，那么也需要该函数进行判断。如果区间可随机访问，效率是 log2n； 否则需要进行最多 n 次移动和 log2n 次比较。例：

```
//区间中有指定元素
int a[8] = {0, 1, 2, 2, 3, 4, 5, 6};
lower_bound(a, a + 8, 2) 返回的是 (a+2) 指向第一个 2
upper_bound(a, a + 8, 2) 返回的是 (a+4) 指向 3
//区间中没有指定元素，lower_bound 和 upper_bound 的返回值是一样的
int a[8] = {0, 1, 3, 4, 5, 6, 7, 8};
lower_bound(a, a + 8, 2) 返回的是 (a+2) 指向 3
upper_bound(a, a + 8, 2) 返回的是 (a+2) 指向 3
```

实例程序：

```
#include <iostream>
using namespace std;
int main()
{   int a[10] = {1,3,5,2,4,6,7,5,2,5};
    sort(a, a + 10);
    for (int i = 0; i < 10; ++i)     printf("%d  ", a[i]);
    printf("\n");

    int *s, *e;
    s = lower_bound(a, a + 10, 5);    e = upper_bound(a, a + 10, 5);
    printf("[%d, %d)\n", s - a, e - a);
    while(s != e)      printf("%d ", *s++);
    return 0;
}
```

14. binary_search //要求有序区间

```
void binary_search( iterator begin, iterator end,
                    const Type &value[, Cmpfunc func]);
```

查找指定的**有序区间**中是否有值等于 value 的元素，如果有，返回 true，否则返回 false。如果有序区间排序时（比如 sort）使用的是 func 函数进行比较，那么 binary_search 也需要该函数进行判断。如果区间可随机访问，效率是 log2n； 否则需要进行最多 n 次移动和 log2n 次比较。

15. merge / inplace_merge

```
iterator merge(iterator begin1, iterator end1, iterator begin2,
    iterator end2,iterator dest[, Cmpfunc func]);
```

将两个**有序区间**合并到从 dest 开始的区间，返回目的区间的结尾。

```
void inplace_merge(iterator begin, iterator middle, iterator end
    [, Cmpfunc func]);
```

将**有序区间**[begin, middle)和[middle, end)合并到[begin, end)。

16. includes

```
bool includes(iterator begin1, iterator end1,iterator begin2,
   iterator end2[, StrictWeakOrdering comp]);
```

判断有序区间[begin2, end2)中每个元素是否都在[begin1, end1)中。

17. set_union, set_intersection, set_difference, set_symmetric_diffrece

这四个算法是求**有序区间**的并、交、差、对称差，并保存到一个新的区间中去。最好是只对没有重复元素的有序区间使用这四个算法，答案比较直观。

```
1)  iterator set_union(iterator begin1, iterator end1,
                iterator begin2, iterator end2
                iterator dest[, StrictWeakOrdering comp]);
```

求**有序区间** a[begin1, end1)和 b[begin2, end2)的并，保存在 dest 开始的区间，并返回其结尾。如果同一个元素在 a 出现 m 次，在 b 出现 n 次，则会保留 max(m,n)个。

```
2)  iterator set_union(iterator begin1, iterator end1,
                iterator begin2, iterator end2
                iterator dest[, Cmpfunc func]);
```

求有序区间 a[begin1, end1)和 b[begin2, end2)的交，保存在 dest 开始的区间，并返回其结尾。如果同一个元素在 a 出现 m 次，在 b 出现 n 次，则会保留 min(m,n)个。

```
3) iterator set_intersection(iterator begin1, iterator end1,
                      iterator begin2, iterator end2
                      iterator dest[, Cmpfunc func])
```

求有序区间 a[begin1, end1)对 b[begin2, end2)的差，保存在 dest 开始的区间，并返回其结尾。如果同一个元素在 a 出现 m 次，在 b 出现 n 次，则会保留 max(m-n, 0)个。

4) 既不在 a 也不在 b 中的所有元素

```
   iterator set_union(iterator begin1, iterator end1,
                iterator begin2, iterator end2
                iterator dest[, Cmpfunc func]);
```

求有序区间 a[begin1, end1)和 b[begin2, end2)的对称差，保存在 dest 开始的区间，并返回其结尾。如果同一个元素在 a 出现 m 次，在 b 出现 n 次，则会保留 abs(m-n)个。

18. next_permutation / prev_permutation

```
bool next_permutation(iterator begin, iterator end[, Cmpfunc func]);
bool prev_permutation(iterator begin, iterator end[, Cmpfunc func]);
```

将**能够随机访问**的[begin, end)之间的序列转换到上一个/下一个序列，可以给定比较函数。如果已经是最后一个序列，就把序列修改为最小/大的状态(就像整形溢出一样)。效率是一般 dfs 枚举的 2 倍(n=11 的测试结果)。

注意：在开始使用这两个函数之前首先对[begin, end)进行初始化，并且最好采用 do{ //something}while(next_permutation(begin, end);这样的循环，保证初始个序列会被处理。例：

```
int perm[6], count1 = 0;
for(int i = 0; i < 6; i++){ perm[i] = i; }
do
{   count1 ++;} while(next_permutation(perm, perm+n));
cout << endl << count1 << endl;
//输出 720，数组最后的状态为 1 2 3 4 5 6
```

实例程序：

```
#include <iostream>
using namespace std;
int main()
{   int a[3] = {0,1,2}, i;
    do
    {  for (i = 0; i < 3; ++i)   printf("%d ", a[i]);
       printf("\n");
    } while(next_permutation(a, a + 3));
    printf("NEXT\n");
    for (i = 0; i < 3; ++i)
    {   printf("%d ", a[i]);      a[i] = 3 - i;    }
    printf("\nPREV\n");
    do
    {   for (i = 0; i < 3; ++i)   printf("%d ", a[i]);
        printf("\n");
    }while(prev_permutation(a, a + 3));
    printf("\n");
}
```

19. power

```
Type power(Type a, int n[, multiplyFunc func);
```

计算 n 个 a 的乘积(默认使用 operator*，或者指定二元函数 func)。要求 operator*(或者 func)是可结合的，否则结果无法预料。复杂度为 $\log_2 n$ + n 的二进制表示中 1 的个数，也就是使用快速幂，要求 n>=0。

20. heap operations

在**可以随机访问**的容器上进行堆操作。

1) 判断是否是堆

bool is_head(iterator begin, iterator end[, Cmpfunc cmp]);

按照指定的顺序(默认 operator<，或指定 cmp 比较)如果[begin, end)是堆即返回 true

2) 建立堆

void make_head(iterator begin, iterator end[, Cmpfunc cmp]);

在[begin, end)的区间上建立堆。可指定比较函数

3) 插入元素到堆

void push_heap(iterator begin, iterator end[, Cmpfunc cmp]);

前提:[begin, end)是堆，将 end 指向的元素插入堆中，新堆为[begin, end+1)。使用之前需要先把元素插入到 end 所指位置(比如 push_back)，然后调用 push_heap。

4) 从堆中移除

void pop_heap(iterator begin, iterator end[, Cmpfunc cmp]);

前提:[begin, end)是堆，将 beign 指向的元素移除，新堆为[begin, end-1)。被移除的元素实际上放置在 end-1 位置，**该位置不会被释放。**

5) 堆排序

void sort_heap(iterator begin, iterator end[, Cmpfunc cmp]);

前提:[begin, end)是堆，将[begin, end)排序(不稳定)。例:

```
vector<int>a;
int v[6] = {4,2,5,3,1};
a.assign(v, v+5);
make_heap(a.begin(), a.end());
copy(a.begin(), a.end(), ostream_iterator<int>(cout, " "));
//输出 5 3 4 2 1
a.push_back(6);
push_heap(a.begin(), a.end());
copy(a.begin(), a.end(), ostream_iterator<int>(cout, " "));
//输出 6 3 5 2 1 4
pop_heap(a.begin(), a.end());
copy(a.begin(), a.end(), ostream_iterator<int>(cout, " "));
//输出 5 3 4 2 1 6  ---- a.end()返回的值没有变化!
a.erase(a.end()-1, a.end());
copy(a.begin(), a.end(), ostream_iterator<int>(cout, " "));
//输出 5 3 5 2 1  ---- 最后一个元素从 a 中删除了
sort_heap(a.begin(), a.end());
copy(a.begin(), a.end(), ostream_iterator<int>(cout, " "));
//输出 1 2 3 4 5
```

21. min / max / swap

对两个数求最大值、最小值或交换其值。

```
TYPE min(TYPE a, TYPE b){return a<b?a:b;}
TYPE max(TYPE a, TYPE b){return b<a?a:b;}
void swap(TYPE &a, TYPE &b){TPYE t = a; a = b; b = t;}
```

实例程序：

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{   int a = 1, b = 2;
    cout << min(a, b) << endl;    cout << max(a, b) << endl;
    swap(a, b);
    cout << a << ", " << b << endl;
    return 0;
}
```

22. numeric_limits

numeric_limits<TYPE>::min() 返回 type 类型的最小值
numeric_limits<TYPE>::max() 返回 type 类型的最大值
numeric_limits<TYPE>::epsilon() 返回 type 类型 1 和最接近 1 值的差(最小精度)