STL概述

目录

1.string类

```
string 类是下面的模板类实例化出来的
```

```
typedef basic_string<char> string;
```

1.1 sting对象的实例化

```
string s1("Hello");
string month = "March";
string s2(8,'x');
```

错误的范例：

```
string error1 = 'c'; // 错
string error2('u'); // 错
string error3 = 22; // 错
string error4(8); // 错
```

1.2 string相关函数

–length()

–string 支持流读取运算符

```
string stringObject;
cin >> stringObject;
```

–string 支持getline函数

```
- string s;
- getline(cin ,s);
```

–用 = 赋值

```
string s1("cat"), s2;
s2 = s1;
```

–用 assign 成员函数复制

```
string s1("cat"), s3;
s3.assign(s1);
```

–用 assign 成员函数部分复制

```
string s1("catpig"), s3;
s3.assign(s1, 1, 3); //从s1 中下标为1的字符开始复制3个字符给s3
```

–逐个访问string对象中的字符

```
string s1("Hello");
for(int i=0;i<s1.length();i++)
cout << s1.at(i) << endl;
```

成员函数at会做范围检查，如果超出范围，会抛出out_of_range异常，而下标运算符[]不做范围检查。

–string 的赋值和连接

- 用 + 运算符连接字符串

```
string s1("good "), s2("morning! ");
s1 += s2;
cout << s1;
```

- 用成员函数 append 连接字符串

```
string s1("good "), s2("morning! ");
s1.append(s2);
cout << s1;
s2.append(s1, 3, s1.size());//s1.size() , s1字符数
cout << s2;
```

下标为3开始，append()个字符，如果字符串内没有足够字符，则复制到字符串最后一个字符

–比较string

- 用关系运算符比较string的大小

> == , >, >=, <, <=, != 返回值都是bool类型，成立返回true, 否则返回false

- 用成员函数compare比较string的大小

```
string s1("hello"),s2("hello"),s3("hell");
int f1 = s1.compare(s2);
int f2 = s1.compare(s3);
int f3 = s3.compare(s1);
int f4 = s1.compare(1,2,s3,0,3); //s1 1-2; s3 0-3
int f5 = s1.compare(0,s1.size(),s3);//s1 0-end
cout << f1 << endl << f2 << endl << f3 << endl;
cout << f4 << endl << f5 << endl;
```

- 输出

```
0 // hello == hello
1 // hello > hell
-1 // hell < hello
-1 // el < hell
1 // hello > hell
```

–子串

- 成员函数 substr

```
string s1("hello world"), s2;
s2 = s1.substr(4,5); // 下标4开始5个字符
cout << s2 << endl;
```

–交换string

- 成员函数 swap

```
string s1("hello world"), s2("really");
s1.swap(s2);
cout << s1 << endl;
cout << s2 << endl;
```

–寻找string中的字符

- 成员函数 find()

```
string s1("hello world");
s1.find("lo");
```

在s1中从前向后查找 “lo” 第一次出现的地方，如果找到，返回 “lo” 开始的位置，即l所在的位置下标。如果找不到，返回string::npos （string中定义的静态常量）

- 成员函数 rfind()

```
string s1("hello world");
s1.rfind("lo");
```

在s1中从后向前查找 “lo” 第一次出现的地方，如果找到，返回 “lo” 开始的位置，即l所在的位置下标。如果找不到，返回string::npos 。
- 指定查找位置

```
string s1("hello worlld");
cout << s1.find("ll",1) << endl;
cout << s1.find("ll",2) << endl;
cout << s1.find("ll",3) << endl; // 分别从下标1，2，3开始查找“ll”
```

- 成员函数 find_first_of()

```
string s1("hello world");
s1.find_first_of(“abcd");
```

在s1中从前向后查找 “abcd” 中任何一个字符第一次出现的地方，如果找到，返回找到字母的位置，如果找不到，返回string::npos。 • 成员函数 find_last_of()

```
string s1("hello world");
s1.find_last_of(“abcd");
```

在s1中查找 “abcd” 中任何一个字符最后一次出现的地方，如果找到，返回找到字母的位置，如果找不到，返回string::npos。
- 成员函数 find_first_not_of()

```
- string s1("hello world");
- s1.find_first_not_of(“abcd");
```

在s1中从前向后查找不在 “abcd” 中的字母第一次出现的地方，如果找到，返回找到字母的位置，如果找不到，返回string::npos。
- 成员函数 find_last_not_of()

```
string s1("hello world");
s1.find_last_not_of(“abcd");
```

在s1中从后向前查找不在 “abcd” 中的字母第一次出现的地方，如果找到，返回找到字母的位置，如果找不到，返回string::npos。

–删除string中的字符

- 成员函数erase()

```
string s1("hello worlld");
s1.erase(5);
cout << s1;
cout << s1.length();
cout << s1.size();// 去掉下标 5 及之后的字符
```

–替换string中的字符

- 成员函数 replace()

```
string s1("hello world");
s1.replace(2,3, “haha");
cout << s1;//将s1中下标2 开始的3个字符换成“haha”
```

输出

```
hehaha world
```

指定要替换字符的长度

```
string s1("hello world");
s1.replace(2,3, "haha", 1,2);
cout << s1;// 将s1中下标2 开始的3个字符换成“haha” 中下标1开始的2个字符
```

–在string中插入字符

- 成员函数insert()

```
string s1("hello world");
string s2(“show insert");
s1.insert(5,s2); // 将s2插入s1下标5的位置
cout << s1 << endl;
s1.insert(2,s2,5,3); //将s2中下标5开始的3个字符插入s1下标2的位置
```

cout << s1 << endl;
输出：

```
helloshow insert world
heinslloshow insert world
```

–转换成C语言式char *字符串

- 成员函数 c_str()

```
string s1("hello world");
printf("%s\n", s1.c_str());// s1.c_str() 返回传统的const char * 类型字符串，且该字符串以‘\0’结尾
```

- 成员函数data()

```
string s1("hello world");
const char * p1=s1.data();
for(int i=0;i<s1.length();i++)
printf("%c",*(p1+i));//s1.data() 返回一个char * 类型的字符串，对s1 的修改可能会使p1出错。
```

–字符串拷贝

- 成员函数copy()

```
string s1("hello world");
int len = s1.length();
char * p2 = new char[len+1];
s1.copy(p2,5,0);
p2[5]=0;
cout << p2 << endl;// s1.copy(p2,5,0) 从s1的下标0的字符开始制作一个最长5个字符长度的字符串副本并将其赋值给p2。返回值表明实际复制字符串的长度。
```

–字符串流处理

- 除了标准流和文件流输入输出外，还可以从string进行输入输出；
- 类似 istream和ostream进行标准流输入输出，我们用istringstream 和 ostringstream进行字符串上的输入输出，也称为内存输入输出。

- 字符串流处理 - 字符串输入流 istringstream

```
string input("Input test 123 4.7 A");
istringstream inputString(input);
string string1, string2;
int i;
double d;
char c;
inputString >> string1 >> string2 >> i >> d >> c;
cout << string1 << endl << string2 << endl;
cout << i << endl << d << endl << c <<endl;
long L;
if(inputString >> L) cout << "long\n";//inputString流中没有东西了
else cout << "empty\n";
```

输出

```
Input
test
123
4.7
A
empty
```

- 字符串流处理 - 字符串输出流 istringstream

```
ostringstream outputString;
int a = 10;
outputString << "This " << a << "ok" << endl;//将东西放在流中
cout << outputString.str();//将流中的东西取出来
```

2.STL概述

```
标准模板库（Standard Template Library）就是一些常用数据结构和算法的模板的集合

容器：可容纳各种数据类型的通用数据结构，是类模板

迭代器：可用于依次存取容器中元素，类似于指针

算法：用来操作容器中的元素的函数模板
```

算法本身与他们操作的数据的类型无关。因此他们可以在从简单数组到高度复杂容器的任何数据结构上使用。

2.1容器概述

可以用于存放各种类型的数据（基本类型的变量，对象等）的数据结构，都是类模版。
分为三种：

1)顺序容器

vector, deque,list----一维可变数组、双向队列、双向列表
容器并非排序的，元素的插入位置同元素的值无关。

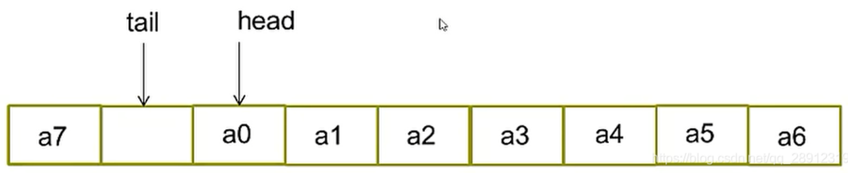
vector 头文件 <vector>

动态数组。元素在内存连续存放。随机存取任何元素都能在常数时间完成。在尾端增删元素具有较佳的性能(大部分情况下是常数时间—需要重新分配空间的时候时间才会变长）

deque 头文件 <deque>

双向队列。元素在内存连续存放。随机存取任何元素都能在常数时间完成(但次于vector)。在两端增删元素具有较佳的性能(大部分情况下是常数时间）。

```
这里讨论一下为什么随机存取的时间次于vector
```



有这样的一种情况，在队尾添加元素的时候，超过了边界，那就回过头来在队列前端添加元素，在存取数据的时候，head+【存取下标】之后，还要判断是否发生了这种情况

list 头文件 <list>

双向链表。元素在内存不连续存放。在任何位置增删元素都能在常数时间完成（不计算查找这个元素的时间）。不支持随机存取。

三者的效率比较

操作	vector	deque	list
在头部插入和删除元素	线性	恒定	恒定
在尾部插入和删除元素	恒定	恒定	恒定
在中间插入和删除元素	线性	线性	恒定
访问头部的元素	恒定	恒定	恒定
访问尾部的元素	恒定	恒定	恒定
访问中间的元素	恒定	恒定	线性

2)关联容器

set, multiset, map, multimap

```
元素是排序的
插入任何元素，都按相应的排序规则来确定其位置
在查找时具有非常好的性能
通常以平衡二叉树方式实现，插入和检索的时间都是 O(log(N))
```

set/multiset 头文件 <set>

set 即集合。set中不允许相同元素，multiset中允许存在相同的元素----都是排好序的

map/multimap 头文件 <map>

map与set的不同在于map中存放的元素有且仅有两个成员变量，一个名为first,另一个名为second, map根据first值对元素进行从小到大排序，并可快速地根据first来检索元素。
map同multimap的不同在于是否允许相同first值的元素。

3)容器适配器

stack, queue, priority_queue

stack :头文件 <stack>

栈。是项的有限序列，并满足序列中被删除、检索和修改的项只能是最近插入序列的项（栈顶的项）。后进先出。

queue 头文件 <queue>

队列。插入只可以在尾部进行，删除、检索和修改只允许从头部进行。先进先出。

priority_queue 头文件 <queue>

优先级队列。最高优先级元素总是第一个出列

4)顺序容器和关联容器中都有的成员函数

```
begin 返回指向容器中第一个元素的迭代器
end 返回指向容器中最后一个元素后面的位置的迭代器
rbegin 返回指向容器中最后一个元素的迭代器
rend 返回指向容器中第一个元素前面的位置的迭代器
erase 从容器中删除一个或几个元素
clear 从容器中删除所有元素
insert(迭代器,插入元素)
```

5)顺序容器的常用成员函数

```
front :返回容器中第一个元素的引用
back ： 返回容器中最后一个元素的引用
push_back ： 在容器末尾增加新元素
pop_back ： 删除容器末尾的元素
erase(迭代器) :删除迭代器指向的元素(可能会使该迭代器失效)，或删
除一个区间，返回被删除元素后面的那个元素的迭代器
```

```
对象被插入容器中时，被插入的是对象的一个复制
品。许多算法，比如排序，查找，要求对容器中的元
素进行比较，有的容器本身就是排序的，所以，放入
容器的对象所属的类，往往还应该重载 == 和 < 运算符。
```

2.2迭代器

```
用于指向顺序容器和关联容器中的元素
迭代器用法和指针类似
有const 和非 const两种
通过迭代器可以读取它指向的元素
通过非const迭代器还能修改其指向的元素
```

定义一个容器类的迭代器的方法可以是：

```
容器类名::iterator 变量名;
```

```
容器类名::const_iterator 变量名;
```

访问一个迭代器指向的元素：

```
* 迭代器变量名//类似指针
```

注意：迭代器上可以执行 ++ 操作，以使其指向容器中的下一个元素。如果迭代器到达了容器中的最后一个元素的后面，此时再使用它，就会出错，类似于使用NULL或未初始化的指针一样。

例子：

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> v; //一个存放int元素的数组，一开始里面没有元素
    v.push_back(1); v.push_back(2); v.push_back(3);
    v.push_back(4);
    vector<int>::const_iterator i; //常量迭代器
    for( i = v.begin();i != v.end();++i )
    cout << * i << ",";
    cout << endl;
    vector<int>::reverse_iterator r; //反向迭代器
    for( r = v.rbegin();r != v.rend();r++ ) //++是往前走的
    cout << * r << ",";
    cout << endl;
    vector<int>::iterator j; //非常量迭代器
    for( j = v.begin();j != v.end();j ++ )
    * j = 100;
    for( i = v.begin();i != v.end();i++ )
    cout << * i << ",";
}
```

迭代器类型

1）双向迭代器
若p和p1都是双向迭代器，则可对p、p1可进行以下操作：

```
++p，p++ 使p指向容器中下一个元素
--p，p-- 使p指向容器中上一个元素
* p 取p指向的元素
p = p1 赋值
p == p1 ，p!= p1 判断是否相等、不等
```

2）随机访问迭代器
若p和p1都是随机访问迭代器，则可对p、p1可进行以下操作：

```
双向迭代器的所有操作
p += i 将p向后移动i个元素
p -= i 将p向向前移动i个元素
p + i 值为：指向 p 后面的第i个元素的迭代器
p - i 值为：指向 p 前面的第i个元素的迭代器
p[i] 值为：p后面的第i个元素的引用
p < p1，p <= p1，p > p1，p>= p1
p – p1 ： p1和p之间的元素个数
```

不同容器有不同的迭代器

容器	容器上的迭代器类别
vector	随机访问
deque	随机访问
list	双向
set/multiset	双向
map/multimap	双向
stack	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

```
有的算法，例如sort，binary_search需要通过随机访问迭代器来访问容器中的元素，那么list以
及关联容器就不支持该算法！
```

迭代器的使用

```
遍历 vector 可以有以下几种做法(deque亦然)：
```

```
vector<int> v(100);
int i;
for(i = 0;i < v.size() ; i ++)
cout << v[i]; //根据下标随机访问
vector<int>::const_iterator ii;
for( ii = v.begin(); ii != v.end ();++ii)//双向迭代器的性质
cout << * ii;
for( ii = v.begin(); ii < v.end ();++ii )//随机迭代器的性质
cout << * ii;
//间隔一个输出：
ii = v.begin();
while( ii < v.end()) {
cout << * ii;
ii = ii + 2;
}
```

```
list 的迭代器是双向迭代器，
```

```
list<int> v;
list<int>::const_iterator ii;
for( ii = v.begin(); ii != v.end ();++ii )//这里不能是 >
cout << * ii;
```

2.3算法

```
算法就是一个个函数模板，大多数在<algorithm> 中定义

STL中提供能在各种容器中通用的算法，比如查找，排序等

算法通过迭代器来操纵容器中的元素。许多算法可以对容器中的一个局部区间进行操作，
因此需要两个参数，一个是起始元素的迭代器，一个是终止元素的后面一个元素的迭代器。
比如，排序和查找

有的算法返回一个迭代器。比如 find() 算法，在容器中查找一个元素，并返回一个指向该元素的迭代器

算法可以处理容器，也可以处理普通数组
```

find()

```
template<class InIt, class T>
InIt find(InIt first, InIt last, const T& val);
```

```
first 和 last 这两个参数都是容器的迭代器，它们给出了容器中的查找
区间起点和终点[first,last)。区间的起点是位于查找范围之中的，而终
点不是。find在[first,last)查找等于val的元素

用 == 运算符判断相等

函数返回值是一个迭代器。如果找到，则该迭代器指向被找到的元素。
如果找不到，则该迭代器等于last
```

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
int main() { //find算法示例
    int array[10] = {10,20,30,40};
    vector<int> v;
    v.push_back(1); v.push_back(2);
    v.push_back(3); v.push_back(4);
    vector<int>::iterator p;
    p = find(v.begin(),v.end(),3);
    if( p != v.end())
    cout << * p << endl; //输出3
    p = find(v.begin(),v.end(),9);
    if( p == v.end())
    cout << "not found " << endl;
    p = find(v.begin()+1,v.end()-2,1); //整个容器：[1,2,3,4]， 查找区间：[2,3)
    if( p != v.end())
    cout << * p << endl;
    int * pp = find( array,array+4,20);//数组名是迭代器
    cout << * pp << endl;
}
```

generate()

replace_if()

sort()

for_each

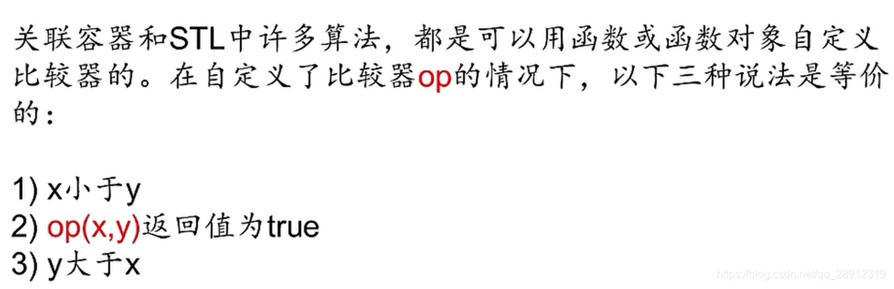
2.4STL中概念

1) STL中“大” “小” 的概念

```
关联容器内部的元素是从小到大排序的
有些算法要求其操作的区间是从小到大排序的，称为“有序区间算法”
    例：binary_search（折半查找）
有些算法会对区间进行从小到大排序，称为“排序算法”
    例： sort
```

使用STL时，在缺省的情况下即没有自定义大小的概念时，以下三个说法等价：
1) x比y小
2) 表达式“x<y”为真
3) y比x大

这个题图是什么情况？



2) STL中“相等”的概念

```
有时，“x和y相等”等价于“x==y为真”
    例：在未排序的区间上进行的算法，如顺序查找find
有时“x和y相等”等价于“x小于y和y小于x同时为假”
    例：
    有序区间算法，如binary_search
    关联容器自身的成员函数find
```

例子：

```
#include <iostream>
#include <algorithm>
using namespace std;
class A {
    int v;
    public:
    A(int n):v(n) { }
    bool operator < ( const A & a2) const { //必须为常量成员函数
        cout << v << "<" << a2.v << "?" << endl;//自定义大小
        return false;
    }
    bool operator ==(const A & a2) const {
        cout << v << "==" << a2.v << "?" << endl;
        return v == a2.v;
    }
};
int main()
{
    A a [] = { A(1),A(2),A(3),A(4),A(5) };
    cout << binary_search(a,a+4,A(9)); //折半查找
    return 0;
}
```

输出：

```
3<9?
2<9?
1<9?
9<1?
1
```

解释：

```
在折半查找中，没有用到==比较，用的是><,在最后面的，在判断的时候，发现1<9并且9<1，
就认定为相等
```

3.vector deque

3.1vector

```
#include <iostream>
#include <vector>
using namespace std;
template<class T>
void PrintVector( T s, T e)
{
    for(; s != e; ++s)
    cout << * s << " ";
    cout << endl;
}
int main() {
    int a[5] = { 1,2,3,4,5 };
    vector<int> v(a,a+5); //将数组a的内容放入v
    cout << "1) " << v.end() - v.begin() << endl;//两个随机迭代器相减输出 1) 5
    cout << "2) "; PrintVector(v.begin(),v.end()); //2) 1 2 3 4 5
    v.insert(v.begin() + 2, 13); //在begin()+2位置插入 13
    cout << "3) "; PrintVector(v.begin(),v.end()); //3) 1 2 13 3 4 5
    v.erase(v.begin() + 2); //删除位于 begin() + 2的元素
    cout << "4) "; PrintVector(v.begin(),v.end()); //4) 1 2 3 4 5
    vector<int> v2(4,100); //v2 有4个元素，都是100
    v2.insert(v2.begin(),v.begin()+ 1,v.begin()+3);//将v的一段插入v2开头
    cout << "5) v2: "; PrintVector(v2.begin(),v2.end());
    //5) v2: 2 3 100 100 100 100 47
    v.erase(v.begin() + 1, v.begin() + 3); //删除 v 上的一个区间,即 2,3
    cout << "6) "; PrintVector(v.begin(),v.end()); //6) 1 4 5
    return 0;
}
```

用vector实现二维数组

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<vector<int> > v(3); //这里的>间必须有空格，不然会算作右移运算法
    //v有3个元素，每个元素都是vector<int> 容器
    for(int i = 0;i < v.size(); ++i)
    for(int j = 0; j < 4; ++j)
    v[i].push_back(j);
    for(int i = 0;i < v.size(); ++i) {
    for(int j = 0; j < v[i].size(); ++j)
    cout << v[i][j] << " ";
    cout << endl;
return 0;
}
```

3.2Deque

```
所有适用于 vector的操作都适用于 deque。
```

deque还有 push_front（将元素插入到前面） 和pop_front(删除最前面的元素）操作。复杂度是O(1)

3.3list——双向链表

```
在任何位置插入删除都是常数时间—前提是找到这个元素，不支持随机存取。
除了具有所有顺序容器都有的成员函数以外，还支持8个成员函数：
```

```
push_front：在前面插入
pop_front：删除前面的元素
sort：排序—自己的成员函数（ list 不支持 STL 的算法 sort）
```

```
remove：删除和指定值相等的所有元素
unique：删除所有和前一个元素相同的元素(要做到元素不重复，则
unique之前还需要 sort)
merge：合并两个链表，并清空被合并的那个
reverse：颠倒链表
splice：在指定位置前面插入另一链表中的一个或多个元素，并在另一链表中删除被插入的元素
```

例：

```
#include <list>
#include <iostream>
#include <algorithm>
using namespace std;
class A {
private:
    int n;
public:
    A( int n_ ) { n = n_; }
    friend bool operator<( const A & a1, const A & a2);
    friend bool operator==( const A & a1, const A & a2);
    friend ostream & operator <<( ostream & o, const A & a);
};
bool operator<( const A & a1, const A & a2) {
    return a1.n < a2.n;
}
bool operator==( const A & a1, const A & a2) {
    return a1.n == a2.n;
}
ostream & operator <<( ostream & o, const A & a) {
    o << a.n;
    return o;
}
template <class T>
void PrintList(const list<T> & lst) {
    //不推荐的写法，还是用两个迭代器作为参数更好
    int tmp = lst.size();
    if( tmp > 0 ) {
    typename list<T>::const_iterator i; //定义容器类型不确定的迭代器，需要用typename
    i = lst.begin();
    for( i = lst.begin();i != lst.end(); i ++)
        cout << * i << ",";
} // typename用来说明 list<T>::const_iterator是个类型
    //在vs中不写也可以
int main() {
    list<A> lst1,lst2;
    lst1.push_back(1);lst1.push_back(3); //这里的1是临时对象
    lst1.push_back(2);lst1.push_back(4);
    lst1.push_back(2);
    lst2.push_back(10);lst2.push_front(20);
    lst2.push_back(30);lst2.push_back(30);
    lst2.push_back(30);lst2.push_front(40);
    lst2.push_back(40);
    cout << "1) "; PrintList( lst1); cout << endl;
    // 1) 1,3,2,4,2,
    cout << "2) "; PrintList( lst2); cout << endl;
    // 2) 40,20,10,30,30,30,40,
    lst2.sort();
    cout << "3) "; PrintList( lst2); cout << endl;
    //3) 10,20,30,30,30,40,40,
    lst2.pop_front();
    cout << "4) "; PrintList( lst2); cout << endl;
    //4) 20,30,30,30,40,40,
    lst1.remove(2); //删除所有和A(2)相等的元素
    cout << "5) "; PrintList( lst1); cout << endl;
    //5) 1,3,4,
    lst2.unique(); //删除所有和前一个元素相等的元素
    cout << "6) "; PrintList( lst2); cout << endl;
    //6) 20,30,40,
    58
    lst1.merge (lst2); //合并 lst2到lst1并清空lst2
    cout << "7) "; PrintList( lst1); cout << endl;
    //7) 1,3,4,20,30,40,
    cout << "8) "; PrintList( lst2); cout << endl;
    //8)
    lst1.reverse();
    cout << "9) "; PrintList( lst1); cout << endl;
    //9) 40,30,20,4,3,1,
    59
    60
    lst2.push_back (100);lst2.push_back (200);
    lst2.push_back (300);lst2.push_back (400);
    list<A>::iterator p1,p2,p3;
    p1 = find(lst1.begin(),lst1.end(),3);//返回一个迭代器
    p2 = find(lst2.begin(),lst2.end(),200);
    p3 = find(lst2.begin(),lst2.end(),400);
    lst1.splice(p1,lst2,p2, p3);
    //将[p2,p3)插入p1之前，并从lst2中删除[p2,p3)
    cout << "10) "; PrintList( lst1); cout << endl;
    //10) 40,30,20,4,200,300,3,1,
    cout << "11) "; PrintList( lst2); cout << endl;
    //11) 100,400,
    return 0;
}
```

4.函数对象

```
是个类的对象，但是用起来看上去象函数调用，实际上也执行了函数调用。
对()的重载必须重载成类的成员函数，和[]一样，不能重载成顶层函数
```

```
class CMyAverage {
public:
    double operator()( int a1, int a2, int a3 ) { //重载 () 运算符
    return (double)(a1 + a2+a3) / 3;
    }
};
CMyAverage average; //函数对象—实际上还是一个类的对象
cout << average(3,2,3); // average.operator()(3,2,3) 用起来看上去象函数调用 输出 2.66667
```

4.1函数对象的实例：

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>
using namespace std;
int sumSquares( int total, int value){ //计算二次方
    return total + value * value;
}
template <class T>
void PrintInterval(T first, T last){ //输出区间[first,last)中的元素
    for( ; first != last; ++ first)
    cout << * first << " ";
    cout << endl;
}
template<class T>
class SumPowers//计算次方
{
private:
    int power;
public:
    SumPowers(int p):power(p) { }
    const T operator() ( const T & total, const T & value) { //计算 value的power次方，加到total上
        T v = value;
        for( int i = 0;i < power - 1; ++ i)
        v = v * value;
        return total + v;
    }
};
int main()
{
    const int SIZE = 10;
    int a1[] = { 1,2,3,4,5,6,7,8,9,10 };
    vector<int> v(a1,a1+SIZE);
    cout << "1) "; PrintInterval(v.begin(),v.end());
    int result = accumulate(v.begin(),v.end(),0,SumSquares);
    cout << "2) 平方和: " << result << endl;
    result =
    accumulate(v.begin(),v.end(),0,SumPowers<int>(3));
    cout << "3) 立方和: " << result << endl;
    result =
    accumulate(v.begin(),v.end(),0,SumPowers<int>(4));
    cout << "4) 4次方和: " << result;
    return 0;
}
```

代码解析：
首先来看accumulate 源代码：

```
template<typename _InputIterator, typename _Tp>
_Tp accumulate(_InputIterator __first, _InputIterator __last, _Tp __init){
    for ( ; __first != __last; ++__first)
    __init = __init + *__first;
    return __init;
}
```

```
template<typename _InputIterator, typename _Tp, typename _BinaryOperation>
_Tp accumulate(_InputIterator __first, _InputIterator __last,
_Tp __init, _BinaryOperation __binary_op)//这里的OP相当调用了函数对象
{
    for ( ; __first != __last; ++__first)
    __init = __binary_op(__init, *__first);
    return __init;
}
```

在本例中调用了acccumulate的第二个代码：
具体实例化如下：

```
int result = accumulate(v.begin(),v.end(),0,SumSquares);
实例化出：
int accumulate(vector<int>::iterator
first,vector<int>::iterator last,int init,int ( * op)( int,int))
{//这里定义了指向函数的指针
    for ( ; first != last; ++first)
    init = op(init, *first);
    return init;
}
```

```
accumulate(v.begin(),v.end(),0,SumPowers<int>(3));
实例化出：
int accumulate(vector<int>::iterator first,
vector<int>::iterator last,int init, SumPowers<int> op)
{//这里直接使用了函数对象
```

```
    for ( ; first != last; ++first)
    init = op(init, *first);
    return init;
}
```

4.2STL 函数对象类模板

```
equal_to
greater
less//类中都有对()的重载
```

greater的例子

```
template<class T>
struct greater : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const {
        return x > y;//与正常的大小比较是相反的
    }
};
```

应用：

```
list 中的sort函数：

sort（）按 < 规定的比较方法 升序排列。

template <class T2>
void sort(T2 op);
可以用 op来比较大小，即 op(x,y) 为true则认为x应该排在前面
```

代码测试:

```
#include <list>
#include <iostream>
#include <iterator>
using namespace std;
class MyLess {//自定义比较器
public:
    bool operator()( const int & c1, const int & c2 )
    {
        return (c1 % 10) < (c2 % 10);
    }
};
int main(){
    const int SIZE = 5;
    int a[SIZE] = {5,21,14,2,3};
    list<int> lst(a,a+SIZE);
    lst.sort(MyLess()); //函数对象
    ostream_iterator<int> output(cout,",");
    copy( lst.begin(),lst.end(),output); cout << endl;
    lst.sort(greater<int>()); //greater<int>()是个对象
    //本句进行降序排序—即使用了sort ( ) 的第二个代码
    copy( lst.begin(),lst.end(),output); cout << endl;
    return 0;
}
```

4.3自定义比较器

```
#include <iostream>
#include <iterator>
using namespace std;
class MyLess{//自定义比较器
public:
    bool operator() (int a1,int a2)
    {
        if( ( a1 % 10 ) < (a2%10) )
            return true;
        else
            return false;
    }
};
bool MyCompare(int a1,int a2)//自定义比较器
{
    if( ( a1 % 10 ) < (a2%10) )
        return false;
    else
        return true;
}
int main()
{
    int a[] = {35,7,13,19,12};
    cout << MyMax(a,5,MyLess()) << endl;
    cout << MyMax(a,5,MyCompare) << endl;
    return 0;
}
```

```
MyMax的实现
template <class T, class Pred>
T MyMax( T * p, int n, Pred myless)
{//依据比较器不同类型产生不同的结果
    T tmpmax = p[0];
    for( int i = 1;i < n;i ++ )
    if( myless(tmpmax,p[i]))
    tmpmax = p[i];
    return tmpmax;
};
```

5.set multiset

```
按照一定的比大小方式对容器中的元素从小打到排序
```

5.1独有的函数

```
find: 查找等于某个值的元素(x小于y和y小于x同时不成立即为相等)
lower_bound : 查找某个下界
upper_bound : 查找某个上界
equal_range : 同时查找上界和下界
count :计算等于某个值的元素个数(x小于y和y小于x同时不成立即为相等)
insert: 用以插入一个元素或一个区间
```

5.2STL 中预先定义的pair类模板

```
template<class _T1, class _T2>
struct pair//和class一样可以定义一个类，只不过成员默认公有
{
    typedef _T1 first_type;
    typedef _T2 second_type;
    _T1 first;
    _T2 second;
    pair(): first(), second() {}//如果first和second为对象的话，调用默认构造函数
    pair(const _T1& __a, const _T2& __b): first(__a), second(__b) { }
    //是一个函数模板，使用的时候才会实例化
    //参数是pair模板类的对象
    template<class _U1, class _U2>
    pair(const pair<_U1, _U2>& __p): first(__p.first), second(__p.second) { }
};
```

map/multimap容器里放着的都是pair模版类的对象，且按first从小到大排序
第三个构造函数用法示例：
pair<int,int>
p(pair<double,double>(5.5,4.6));// p.first = 5, p.second =4

5.3multiset

multiset数据结构

```
template<class Key, class Pred = less<Key>, class A = allocator<Key> >//第三个类型参数不重要不做研究
class multiset { …… };
```

Pred类型的变量:决定了multiset 中的元素，“一个比另一个小”是怎么定义的。multiset运行过程中，比较两个元素x,y的大小的做法，就是生成一个 Pred类型的变量，假定为 op,若表达式op(x,y) (通常为函数指针，或者是函数对象)返回值为true,则 x比y小。

Pred的缺省类型是 less< Key >
less 模板的定义：

```
template<class T>
struct less : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) {
        return x < y ;
    }
    const;
};//less模板是靠 < 来比较大小的
```

multiset的成员函数

```
iterator find(const T & val);//相等的意思是x<y和y>x同时不成立
在容器中查找值为val的元素，返回其迭代器。如果找不到，返回end()。
iterator insert(const T & val); 将val插入到容器中并返回其迭代器。
void insert( iterator first,iterator last); 将区间[first,last)插入容器。
int count(const T & val); 统计有多少个元素的值和val相等。
iterator lower_bound(const T & val);
查找一个最大的位置 it,使得[begin(),it) 中所有的元素都比 val 小。
iterator upper_bound(const T & val);
查找一个最小的位置 it,使得[it,end()) 中所有的元素都比 val 大
pair<iterator,iterator> equal_range(const T & val);
同时求得lower_bound和upper_bound
iterator erase(iterator it);
删除it指向的元素，返回其后面的元素的迭代器(Visual studio 2010上如此，但是在
C++标准和Dev C++中，返回值不是这样)。
```

multiset 的用法示例

```
#include <iostream>
#include <set> //使用multiset须包含此文件
using namespace std;
template <class T>
void Print(T first, T last) { //输出一个区间的所有元素
    for(;first != last ; ++first)
        cout << * first << " ";
    cout << endl;
}

class A {
private:
    int n;
public:
    A(int n_ ) { n = n_; }
    friend bool operator< ( const A & a1, const A & a2 ) {
```

```
        return a1.n < a2.n;
    }
    friend ostream & operator<< ( ostream & o, const A & a2 {
        o << a2.n; return o;
    }
    friend class MyLess;
};

struct MyLess {
    bool operator()( const A & a1, const A & a2){ //按个位数比大小
        return ( a1.n % 10 ) < (a2.n % 10);
    }
};

typedef multiset<A> MSET1; //MSET1用 "<"比较大小
typedef multiset<A,MyLess> MSET2; //MSET2用 MyLess::operator()比较大小
int main()
{
    const int SIZE = 6;
    A a[SIZE] = { 4,22,19,8,33,40 };//直接用临时对象赋值
    MSET1 m1;
    m1.insert(a,a+SIZE);
    m1.insert(22);
    cout << "1) " << m1.count(22) << endl; //输出 1) 2—输出22的个数
    cout << "2) "; Print(m1.begin(),m1.end()); //输出 2) 4 8 19 22 22 33 40
    //m1元素：4 8 19 22 22 33 40
    MSET1::iterator pp = m1.find(19);//如果找不到就会返回end ( )
    if( pp != m1.end() ) //条件为真说明找到
        cout << "found" << endl; //本行会被执行，输出 found
    cout << "3) "; cout << * m1.lower_bound(22) << "," <<* m1.upper_bound(22)<< endl;
    //输出 3) 22,33
    pp = m1.erase(m1.lower_bound(22),m1.upper_bound(22));
    //pp指向被删元素的下一个元素
    cout << "4) "; Print(m1.begin(),m1.end()); //输出 4) 4 8 19 33 40
    cout << "5) "; cout << * pp << endl; //输出 5) 33
    MSET2 m2; // m2里的元素按n的个位数从小到大排
    m2.insert(a,a+SIZE);
    cout << "6) "; Print(m2.begin(),m2.end()); //输出 6) 40 22 33 4 8 19
    return 0;
    }
    //m1元素：4 8 19 22 22 33 40
    MSET1::iterator pp = m1.find(19);
    if( pp != m1.end() ) //条件为真说明找到
        cout << "found" << endl; //本行会被执行，输出 found
    cout << "3) "; cout << * m1.lower_bound(22) << "," <<* m1.upper_bound(22)<< endl;
    //输出 3) 22,33
    pp = m1.erase(m1.lower_bound(22),m1.upper_bound(22));
    //pp指向被删元素的下一个元素
    cout << "4) "; Print(m1.begin(),m1.end()); //输出 4) 4 8 19 33 40
    cout << "5) "; cout << * pp << endl; //输出 5) 33
    MSET2 m2; // m2里的元素按n的个位数从小到大排
    m2.insert(a,a+SIZE);
    cout << "6) "; Print(m2.begin(),m2.end()); //输出 6) 40 22 33 4 8 19
    return 0;
}
```

5.4set

```
和multiset的区别是不能有重复的元素—即x<y和x>y都不能成立
```

相关函数

```
insert()返回值为 pair< set<class T>::iterator, bool>
equal_range()返回值为pair< set<class T>::iterator, set<class T>::iterator>
```

用法实例

```
int main() {
    typedef set<int>::iterator IT;
    int a[5] = { 3,4,6,1,2 };
    set<int> st(a,a+5); // st里是 1 2 3 4 6
    pair< IT,bool> result;
    result = st.insert(5); // st变成 1 2 3 4 5 6
    if( result.second ) //插入成功则输出被插入元素
        cout << * result.first << " inserted" << endl; //输出: 5 inserted
    if( st.insert(5).second )
        cout << * result.first << endl;
    else
        cout << * result.first << " already exists" << endl; //输出 5 already exists
    pair<IT,IT> bounds = st.equal_range(4);
    cout << * bounds.first << "," << * bounds.second ; //输出：4,5
    return 0;
}
```

6.map multimap

6.1multimap

```
map/multimap容器里放着的都是pair模版类的对象，且按first从小到大排序
```

```
template<class Key, class T, class Pred = less<Key>,class A = allocator<T> >
class multimap {
    ….
    typedef pair<const Key, T> value_type;
    …….
}; //Key 代表关键字的类型
```

multimap中的元素由 <关键字,值>组成，每个元素是一个pair对象，关键字就是first成员变量,其类型是Key

multimap 中允许多个元素的关键字相同。但关键字可以不是唯一的元素按照first成员变量从小到大排列，缺省情况下用 less 定义关键字的"小于"关系。

用例示范

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    typedef multimap<int,double,less<int> > mmid;
    mmid pairs;
    cout << "1) " << pairs.count(15) << endl;
    pairs.insert(mmid::value_type(15,2.7));//typedef pair<const Key, T> value_type;
    pairs.insert(mmid::value_type(15,99.3)); 输出:
    cout << “2) ” << pairs.count(15) << endl; //求关键字等于某值的元素个数
    pairs.insert(mmid::value_type(30,111.11));
    pairs.insert(mmid::value_type(10,22.22));
    pairs.insert(mmid::value_type(25,33.333));
    pairs.insert(mmid::value_type(20,9.3));
    for( mmid::const_iterator i = pairs.begin();
        i != pairs.end() ;i ++ )
    cout << "(" << i->first << "," << i->second << ")" << ",";
}
```

例题

一个学生成绩录入和查询系统，
接受以下两种输入:

```
Add name id score
Query score
```

name是个字符串，中间没有空格，代表学生姓名。id是个整数，代表学号
score是个整数，表示分数。学号不会重复，分数和姓名都可能重复。
两种输入交替出现。第一种输入表示要添加一个学生的信息，碰到这种输入，就记下学生的姓名、id和分数。第二种输入表示要查询，碰到这种输入，就输出已有记录中分数比score低的最高分获得者的姓名、学号和分数。如果有多个学生都满足条件，就输出学号最大的那个学生的信息。如果找不到满足条件的学生，则输出“Nobody”

```
#include <iostream>
#include <map> //使用multimap需要包含此头文件
#include <string>
using namespace std;
class CStudent
{
    public:
    struct CInfo //类的内部还可以定义类
    {
        int id;
        string name;
    };
    int score;
    CInfo info; //学生的其他信息
};

typedef multimap<int, CStudent::CInfo> MAP_STD;

int main() {
    MAP_STD mp;
    CStudent st;
    string cmd;
    while( cin >> cmd ) {
    if( cmd == "Add") {
        cin >> st.info.name >> st.info.id >> st.score ;
        mp.insert(MAP_STD::value_type(st.score,st.info ));
    }
    else if( cmd == "Query" ){
        int score;
        cin >> score;
        MAP_STD::iterator p = mp.lower_bound (score);
        if( p!= mp.begin()) { //说明找到了
            --p;
            score = p->first; //比要查询分数低的最高分
            MAP_STD::iterator maxp = p;
            int maxId = p->second.id;
            for( ; p != mp.begin() && p->first == score; --p) {
            //遍历所有成绩和score相等的学生
                if( p->second.id > maxId ) {
                    maxp = p;
                    maxId = p->second.id ;
                }
            }
            if( p->first == score) {
                p == mp.begin()
                //如果上面循环是因为
                // 而终止，则p指向的元素还要处理
                if( p->second.id > maxId ) {
                    maxp = p;
                    maxId = p->second.id ;
                }
            }
            cout << maxp->second.name << " “ << maxp->second.id << " " << maxp->first << endl;
            }
        else
            //lower_bound的结果就是 begin，说明没人分数比查询分数低
```

```
            cout << "Nobody" << endl;
    } }
    return 0;
}
```

6.2map

```
multimap和map 的区别：map不允许key重复，且map中有[]
```

map的[ ]成员函数

若pairs为map模版类的对象，
　　pairs[key]
返回对关键字等于key的元素的值(second成员变量）的引用。若没有关键字为key的元素，则会往pairs里插入一个关键字为key的元素，其值用无参构造函数初始化，并返回其值的引用
如：
map<int,double> pairs;
则
pairs[50] = 5; 会修改pairs中关键字为50的元素，使其值变成5。
若不存在关键字等于50的元素，则插入此元素，并使其值变为5。

map 实例

```
#include <iostream>
#include <map>
using namespace std;
template <class Key,class Value>
ostream & operator <<( ostream & o, const pair<Key,Value> & p){
    o << "(" << p.first << "," << p.second << ")";
    return o;
}
int main() {
    typedef map<int, double,less<int> > mmid;
    mmid pairs;
    cout << "1) " << pairs.count(15) << endl;
    pairs.insert(mmid::value_type(15,2.7));//插入是否成功的判断方法和set很类似
    pairs.insert(make_pair(15,99.3)); //make_pair生成一个pair对象
    cout << "2) " << pairs.count(15) << endl;
    pairs.insert(mmid::value_type(20,9.3));
    mmid::iterator i;
    cout << "3) ";
    for( i = pairs.begin(); i != pairs.end();i ++ )
    cout << * i << ",";
    cout << endl;
    cout << "4) ";
    int n = pairs[40];//如果没有关键字为40的元素，则插入一个
    for( i = pairs.begin(); i != pairs.end();i ++ )
    cout << * i << ",";
    cout << endl;
    cout << "5) ";
    pairs[15] = 6.28; //把关键字为15的元素值改成6.28
    for( i = pairs.begin(); i != pairs.end();i ++ )
    cout << * i << ",";
}
```

7 容器适配器

```
没有迭代器！很多STL算法不适用，只能用他自身的成员函数
```

7.1stack

```
后进先出的，只能插入，删除，访问栈顶的元素。
```

可用 vector, list, deque来实现。缺省情况下，用deque实现。
用 vector和deque实现，比用list实现性能好。

```
template<class T, class Cont = deque<T> >
class stack {
    ……
};
push 插入元素
pop 弹出元素
top 返回栈顶元素的引用
```

7.2Queue

```
先进先出
```

和stack 基本类似，可以用 list和deque实现。缺省情况下用deque实现。

```
template<class T, class Cont = deque<T> >
class queue {
    ……
};
push 在队尾插入元素
back 返回队尾元素的引用
pop 弹出队头元素
top 返回队头元素的引用
```

7.3priority_queue

```
template <class T, class Container = vector<T>,
class Compare = less<T> >
class priority_queue;
```

可以用vector和deque实现。不能用list实现缺省情况下用vector实现。
priority_queue 通常用堆排序技术实现，保证（优先级）最大的元素总是在最前面。即执行pop操作时，删除的是最大的元素；执行top操作时，返回的是最大元素的常引用。默认的元素比较器是
less。

```
push、pop 时间复杂度O(logn)
top()时间复杂度O(1)
```

7.4三个公有的函数

empty() 成员函数用于判断适配器是否为空
size() 成员函数返回适配器中元素个数

8算法

```
1)不变序列算法
2)变值算法
3)删除算法
4)变序算法
5)排序算法
6)有序区间算法
7)数值算法
```

8.1重载的版本说明

大多重载的算法都是有两个版本的：
其中一个是用“==”判断
元素是否相等，或用“<”来比较大小；
而另一个版本多出来一个类型参数“Pred”，以及函数形参“Pred op”,该版本通过表达式“op(x,y)”的返回值是ture还是false，来判断x是否“等于”y，或者x是否“小于”y。——可以自定义比较器
min_element:
iterate min_element(iterate first,iterate last);
iterate min_element(iterate first,iterate last, Pred op);

8.2不变序列算法

此类算法不会修改算法所作用的容器或对象
适用于所有容器——即顺序容器和关联容器
时间复杂度都是O(n)

全部算法

```
min求两个对象中较小的(可自定义比较器)
max求两个对象中较大的(可自定义比较器)
min_element求区间中的最小值(可自定义比较器)
max_element求区间中的最大值(可自定义比较器)
for_each对区间中的每个元素都做某种操作
count计算区间中等于某值的元素个数
count_if计算区间中符合某种条件的元素个数
find在区间中查找等于某值的元素
find_if在区间中查找符合某条件的元素
find_end在区间中查找另一个区间最后一次出现的位置(可自定义比较器)
find_first_of在区间中查找第一个出现在另一个区间中的元素 (可自定义比较器)
adjacent_find在区间中寻找第一次出现连续两个相等元素的位置(可自定义比较器)
search在区间中查找另一个区间第一次出现的位置(可自定义比较器)
search_n在区间中查找第一次出现等于某值的连续n个元素(可自定义比较器)
equal判断两区间是否相等(可自定义比较器)
mismatch逐个比较两个区间的元素，返回第一次发生不相等的两个元素的位置(可自定义比较器)
exicographical_compare按字典序比较两个区间的大小(可自定义比较器)
```

常用算法：

```
find:
template<class InIt, class T>
InIt find(InIt first, InIt last, const T& val);
返回区间[first,last)中的迭代器i ,使得*i== val,没有找到返回end()迭代器

find_ if:
template<class InIt, class Pred>
InIt find_ if(InIt first, InIt last, Pred pr);
返回区间[first,last) 中的迭代器i,使得pr(*i) == true

for_each:
template<class InIt, class Fun>
Fun for_each(InIt first, InIt last, Fun f);
对[first,last)中的每个元素 e ,执行 f(e) , 要求 f(e)不能改变e

count:
template<class InIt, class T>
size_t count(InIt first, InIt last, const T& val);
计算[first,last) 中等于val的元素个数

count_if:
template<class InIt, class Pred>
size_t count_if(InIt first, InIt last, Pred pr);
计算[first,last) 中符合pr(e) == true 的元素 e的个数

min_element:
template<class FwdIt>
FwdIt min_element(FwdIt first, FwdIt last);
```

```
返回[first,last) 中最小元素的迭代器,以 “< ”作比较器。
最小指没有元素比它小，而不是它比别的不同元素都小
因为即便a!= b，a<b 和b<a有可能都不成立

max_element:
template<class FwdIt>
FwdIt max_element(FwdIt first, FwdIt last);
返回[first,last) 中最大元素(它不小于任何其他元素，但不见得其他不同元素都小于它）的迭代器，
以 “< ”作比较器。
```

例子：

```
#include <iostream>
#include <algorithm>
using namespace std;
class A {
    public: int n;
    A(int i):n(i) { }
};
bool operator<( const A & a1, const A & a2) {
    cout << "< called,a1=" << a1.n << " a2=" << a2.n << endl;
    if( a1.n == 3 && a2.n == 7)
        return true;
    else
        return false;
}
int main() {
    A aa[] = { 3,5,7,2,1};
    //假设第一个元素为最小最大值，然后按照规定的比较器进行比较
    cout << min_element(aa,aa+5)->n << endl;
    cout << max_element(aa,aa+5)->n << endl;
    return 0;
}
```

8.3变值算法

```
此类算法会修改源区间或目标区间元素的值。
值被修改的那个区间，不可以是属于关联容器的——会破坏关联容器的有序性
```

全部算法

```
for_each对区间中的每个元素都做某种操作
copy复制一个区间到别处，目标区间是从前往后被修改的
copy_backward复制一个区间到别处，但目标区间是从后往前被修改的
transform将一个区间的元素变形后拷贝到另一个区间
swap_ranges交换两个区间内容
fill用某个值填充区间
fill_n用某个值替换区间中的n个元素
generate用某个操作的结果填充区间
generate_n用某个操作的结果替换区间中的n个元素
replace将区间中的某个值替换为另一个值
replace_if将区间中符合某种条件的值替换成另一个值
replace_copy将一个区间拷贝到另一个区间，拷贝时某个值要换成新值拷过去
replace_copy_if将一个区间拷贝到另一个区间，拷贝时符合某条件的值要换成新值拷过去
```

常用算法

```
transform
template<class InIt, class OutIt, class Unop>
OutIt transform(InIt first, InIt last, OutIt x, Unop uop);
对[first,last)中的每个迭代器 I ，
    执行 uop( * I ) ; 并将结果依次放入从 x 开始的地方。
    要求 uop( * I ) 不得改变 * I 的值。
本模板返回值是个迭代器，即 x + (last-first)——拷贝过去的区间的最后一个元素
    x 可以和 first相等。
```

例子

```
#include <vector>
#include <iostream>
#include <numeric>
#include <list>
#include <algorithm>
#include <iterator>
using namespace std;
class CLessThen9 {
public:
    bool operator()( int n) { return n < 9; }
};
void outputSquare(int value ) { cout << value * value << " "; }
int calculateCube(int value) { return value * value * value; }
int main() {
    const int SIZE = 10;
    int a1[] = { 1,2,3,4,5,6,7,8,9,10};
    int a2[] = { 100,2,8,1,50,3,8,9,10,2 };
    vector<int> v(a1,a1+SIZE);
    ostream_iterator<int> output(cout," ");
    //这是一个类模板，用int实例化
    //output具有cout的输出功能，且只输出int，输出一个就加一个空格
    random_shuffle(v.begin(),v.end());//随机打乱顺序
    cout << endl << "1) ";
    copy( v.begin(),v.end(),output);//copy实现了输出的功能
    copy( a2,a2+SIZE,v.begin());//要保证目标区间有SIZE大小的空间，不然会数组越界
    cout << endl << "2) ";
    cout << count(v.begin(),v.end(),8);
    cout << endl << "3) ";
    cout << count_if(v.begin(),v.end(),CLessThen9());//将满足条件的元素计数
    cout << endl << "4) ";
    cout << * (min_element(v.begin(),v.end()));
    cout << endl << "5) ";
    cout << * (max_element(v.begin(),v.end()));
    cout << endl << "6) ";
    cout << accumulate(v.begin(),v.end(),0);//求和
    cout << endl << "7) ";
    for_each(v.begin(),v.end(),outputSquare);
    vector<int> cubes(SIZE);
    transform(a1,a1+SIZE,cubes.begin(),calculateCube);
    cout << endl << "8) ";
    copy( cubes.begin(),cubes.end(),output);
}
```

cpoy的理解：

```
int main(){
    int a[4]={ 1,2,3,4 };
    My_ ostream_ _iterator<int> oit(cout,"*");//输出到屏幕上
    copy(a,a+4,oit); //输出 1*2*3*4*
    ofstream oFile("test.txt' , ios::out);
    My_ ostream_ iterator<int> oitf(oFile ,"*");//输出到文件里
    copy(a,a+4,oitf); //向test.xt文件中写入 1*2*3*4*
    oFile.close();
    return 0;
}
```

从以上例子中我们可以看到，copy起到了一个输出的作用。
copy的源代码：

```
template <class _II, class _OI>
inline _OI copy(_II _F,_II _L,_OI _X){
    for(;_ F!=_ L; ++_ X, ++_ F)
    *_X=*_F;
    return( X);
}
```

调用copy(a,a+4,oit);的时候会被实例化为：

```
My_ostream_iterator<int> copy(int *_F, int *_L, My_ostream_ iterator<int> _X){
    for(;_ F!=_ L; ++_ X, ++_ F)
    *_X=*_F;
    return( X);
}
```

如何定义一个My_ostream_iterator< int>类模板才能够满足上面的实例化呢？
首先要重载++只需要保证通过就可以了
重载*，因为在等号的左边，所以返回值My_ostream_iterator< int>的引用
重载==实现输出的功能
代码如下：

```
#include <iterator>
template<class T>
class My_ostream_ iterator:public iterator<output _iterator_ _tag, T>{//public部分编译器需要
private:
    string sep; //分隔符
    ostream & OS;
public:
    My_ostream_ iterator(ostream & 0, string s):sep(s), os(0){ }
    void operator ++() {}; // ++只需要有定义即可，不需要做什么
    My_ostream_ iterator & operator* () { return * this; }
    My_ostream_ iterator & operator= ( const T & val){
        OS << val << sep;
        return * this;
    }
};

```

8.4删除算法

```
 删除算法会删除一个容器里的某些元素。
“删除”——并不会使容器里的元素减少，
    将所有应该被删除的元素看做空位子
    用留下的元素从后往前移，依次去填空位子。
    元素往前移后，它原来的位置也就算是空位子，
    也应由后面的留下的元素来填上。
    最后，没有被填上的空位子，维持其原来的值不变。
删除算法不应作用于关联容器
```

全部算法

```
remove删除区间中等于某个值的元素——返回的是指向最后一个有效元素的后面
remove_if删除区间中满足某种条件的元素
remove_copy拷贝区间到另一个区间。等于某个值的元素不拷贝
remove_copy_if拷贝区间到另一个区间。符合某种条件的元素不拷贝
unique删除区间中连续相等的元素，只留下一个(可自定义比较器)
unique_copy拷贝区间到另一个区间。连续相等的元素，只拷贝第一个到目标区间 (可自定义比较器)
```

常用算法

```
unique
template<class FwdIt>
```

```
FwdIt unique(FwdIt first, FwdIt last);
    用 == 比较是否等
template<class FwdIt, class Pred>
FwdIt unique(FwdIt first, FwdIt last, Pred pr);
    用 pr 比较是否等
对[first,last) 这个序列中连续相等的元素，只留下第一个。
返回值是迭代器，指向元素删除后的区间的最后一个元素的后面。
```

例子

```
int main()
{
    int a[5] = { 1,2,3,2,5};
    int b[6] = { 1,2,3,2,5,6};
    ostream_iterator<int> oit(cout,",");
    int * p = remove(a,a+5,2);
    cout << "1) "; copy(a,a+5,oit); cout << endl;
    //输出 1) 1,3,5,2,5,—模拟这个过程，填充空位子
    cout << "2) " << p - a << endl;
    //输出 2) 3，说明有效的元素还有三个
    vector<int> v(b,b+6);
    remove(v.begin(),v.end(),2);
    cout << "3) ";copy(v.begin(),v.end(),oit);cout << endl;
    //输出 3) 1,3,5,6,5,6,
    cout << "4) "; cout << v.size() << endl;
    //v中的元素没有减少,输出 4) 6
    return 0;
}
```

8.5变序算法

```
变序算法改变容器中元素的顺序，但是不改变元素的值。
变序算法不适用于关联容器
排序算法需要随机访问迭代器的支持
```

全部算法

```
reverse颠倒区间的前后次序
reverse_copy把一个区间颠倒后的结果拷贝到另一个区间，源区间不变
rotate将区间进行循环左移
rotate_copy将区间以首尾相接的形式进行旋转后的结果拷贝到另一个区间，源区间不变
next_permutation将区间改为下一个排列(可自定义比较器)//会修改作用区间
prev_permutation将区间改为上一个排列(可自定义比较器)
//这里说的排序是排列，比如说123是最小的排序，321是最大的排序
random_shuffle随机打乱区间内元素的顺序
partition把区间内满足某个条件的元素移到前面，不满足该条件的移到后面
```

常用算法

```
stable_patition
    把区间内满足某个条件的元素移到前面，
    不满足该条件的移到后面。
    而且对这两部分元素，分别保持它们原来的先后次序不变
random_shuffle :
template<class RanIt>
void random_shuffle(RanIt first, RanIt last);
    随机打乱[first,last) 中的元素，适用于能随机访问的容器。
reverse
template<class BidIt>
void reverse(BidIt first, BidIt last);
    颠倒区间[first,last)顺序
next_permutation
template<class InIt>
bool next_permutaion (Init first,Init last);
    求下一个排列
```

例子

```
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
int main()
{
    string str = "231";
    char szStr[] = "324";
    while (next_permutation(str.begin(), str.end()))//会改变区间
        cout << str << endl;
    cout << "****" << endl;
    while (next_permutation(szStr,szStr + 3))
        cout << szStr << endl;
    sort(str.begin(),str.end());
    cout << "****" << endl;
    while (next_permutation(str.begin(), str.end()))
        cout << str << endl;
    return 0;
}
```

8.7排序算法

```
排序算法比前面的变序算法复杂度更高，一般是O(n×log(n))。
排序算法需要随机访问迭代器的支持
因而不适用于关联容器和list
```

```
sort将区间从小到大排序(可自定义比较器)——不稳定！
stable_sort将区间从小到大排序，并保持相等元素间的相对次(可自定义比较器)——稳定！
partial_sort对区间部分排序，直到最小的n个元素就位(可自定义比较器)
partial_sort_copy将区间前n个元素的排序结果拷贝到别处。源区间不变(可自定义比较器)。
nth_element对区间部分排序，使得第n小的元素（n从0开始算）就位，而且比它小的都在它前面，比它大的都在它后面(可自定义比较器)。
make_heap使区间成为一个“堆”(可自定义比较器)。
push_heap将元素加入一个是“堆”区间(可自定义比较器)。
pop_heap从 “堆”区间删除堆顶元素(可自定义比较器)
sort_heap将一个“堆”区间进行排序，排序结束后，该区间就
是普通的有序区间，不再是 “堆”了(可自定义比较器)。
```

常用算法

```
sort 快速排序
template<class RanIt>
void sort(RanIt first, RanIt last);
    按升序排序
    判断x是否应比y靠前，就看 x < y 是否为true
template<class RanIt, class Pred>
void sort(RanIt first, RanIt last, Pred pr);
    按升序排序。
    判断x是否应比y靠前，就看 pr(x,y)——函数对象或函数指针，是否为true

void nth_element (Iterator first, Iterator first+nth, Iterator last, Compare comp);
重新排列range [first,last)中的元素，使第n个位置的元素是按排序顺序在该位置的元素。
其他元素没有任何特定的顺序，只是第n个元素之前的元素都不大于该元素，而第n个元素后面的元素均不小于该元素。
```

例子

```
#include <iostream>
#include <algorithm>
using namespace std;
class MyLess {
public:
    bool operator()( int n1,int n2) {
        return (n1 % 10) < ( n2 % 10);
    }
};
int main() {
    int a[] = { 14,2,9,111,78 };
    sort(a,a + 5,MyLess());
    int i;
    for( i = 0;i < 5;i ++)
    cout << a[i] << " ";
    cout << endl;
    sort(a,a+5,greater<int>());//greater谁大谁就小
    for( i = 0;i < 5;i ++)
    cout << a[i] << " ";
}
```

sort的相关说明

```
sort 实际上是快速排序，时间复杂度 O(n*log(n));
    平均性能最优
    最坏的情况下，性能可能非常差
如果要保证“最坏情况下”的性能，那么可以使用
    stable_sort 实际上是归并排序，特点是能保持相等元素之间的先后次
    序——稳定！
    在有足够存储空间的情况下，复杂度为 n * log(n)，否则复杂度为n*log(n)*log(n)。
    stable_sort 用法和 sort相同。
排序算法要求随机存取迭代器的支持，所以list不能使用排序算法，要使用list::sort。
```

8.8有序区间算法

```
有序区间算法要求所操作的区间是已经从小到大排好序的
需要随机访问迭代器的支持。
有序区间算法不能用于关联容器和list
```

全部算法

```
binary_search判断区间中是否包含某个元素，这里的相等不是==
includes判断是否一个区间中的每个元素，都在另一个区间中—判断一个集合是不是另外一个集合的子集
lower_bound查找最后一个不小于某值的元素的位置。
upper_bound查找第一个大于某值的元素的位置。
equal_range同时获取lower_bound和upper_bound。
merge合并两个有序区间到第三个区间。
//集合运算
set_union将两个有序区间的并拷贝到第三个区间
set_intersection将两个有序区间的交拷贝到第三个区间
set_difference将两个有序区间的差拷贝到第三个区间
set_symmetric_difference将两个有序区间的对称差拷贝到第三个区间
inplace_merge将两个连续的有序区间原地合并为一个有序区间
```

常用算法

```
binary_search 折半查找：
要求容器已经有序且支持随机访问迭代器，返回是否找到
template<class FwdIt, class T>
bool binary_search(FwdIt first, FwdIt last, const T& val);
    上面这个版本，比较两个元素x,y 大小时，看 x < y
template<class FwdIt, class T, class Pred>
bool binary_search(FwdIt first, FwdIt last, const T& val, Pred pr);
    上面这个版本，比较两个元素x,y 大小时，若 pr(x,y) 为true，则认为x于y

lower_bound:
```

```
template<class FwdIt, class T>
FwdIt lower_bound(FwdIt first, FwdIt last, const T& val);
    要求[first,last)是有序的，
    查找[first,last)中的，最大的位置 FwdIt，使得[first,FwdIt) 中所有的元素都比 val 小


upper_bound
template<class FwdIt, class T>
FwdIt upper_bound(FwdIt first, FwdIt last, const T& val);
    要求[first,last)是有序的，
    查找[first,last)中的，最小的位置 FwdIt，使得[FwdIt,last) 中所有的元素都比 val 大

equal_range
template<class FwdIt, class T>
pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last, const T& val);
    要求[first,last)是有序的，
    返回值是一个pair，假设为 p，则：
        [first,p.first) 中的元素都比 val 小
        [p.second,last)中的所有元素都比 val 大
        p.first 就是lower_bound的结果
        p.last 就是 upper_bound的结果

merge
template<class InIt1, class InIt2, class OutIt>
OutIt merge(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
    用 < 作比较器
template<class InIt1, class InIt2, class OutIt, class Pred>
OutIt merge(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);
    用 pr 作比较器
    把[first1,last1)，[ first2,last2) 两个升序序列合并，形成第3 个升序序列，第3个升序序列以 x 开头    ——x后面空间必须足够

includes：
template<class InIt1, class InIt2>
bool includes(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2);

template<class InIt1, class InIt2, class Pred>
bool includes(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, Pred pr);
判断 [first2,last2)中的每个元素，是否都在[first1,last1)中
    第一个用 <作比较器—— x<y&&y<x不成立
    第二个用 pr 作比较器，pr(x,y) == true说明 x,y相等。


set_difference：
template<class InIt1, class InIt2, class OutIt>
OutIt set_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);

template<class InIt1, class InIt2, class OutIt, class Pred>
OutIt set_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);
求出[first1,last1)中，不在[first2,last2)中的元素，放到 从 x开始的地方。
如果 [first1,last1) 里有多个相等元素不在[first2,last2)中，则这多个元素也都会被放入x代表的目标区间里

set_intersection：
template<class InIt1, class InIt2, class OutIt>
OutIt set_intersection(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);

template<class InIt1, class InIt2, class OutIt, class Pred>
OutIt set_intersection(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);
求出[first1,last1)和[first2,last2)中共有的元素，放到从 x开始的地方。
若某个元素e 在[first1,last1)里出现 n1次，在[first2,last2)里出现n2次，则该元素在目标区间里出现min(n1,n2)——求交

set_symmetric_difference———并减去交：
template<class InIt1, class InIt2, class OutIt>
OutIt set_symmetric_difference(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x);

template<class InIt1, class InIt2, class OutIt, class Pred>
OutIt set_symmetric_difference(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x, Pred pr);

set_union
template<class InIt1, class InIt2, class OutIt>
OutIt set_union(InIt1 first1, InIt1 last1, InIt2 first2,InIt2 last2, OutIt x);
    用<比较大小
template<class InIt1, class InIt2, class OutIt, class Pred>
OutIt set_union(InIt1 first1, InIt1 last1, InIt2 first2,InIt2 last2, OutIt x, Pred pr);
    用 pr 比较大小
    求两个区间的并，放到以 x开始的位置。
    若某个元素e 在[first1,last1)里出现 n1次，在[first2,last2)里出现n2次，则该元素在目标区间里出现max(n1,n2)
```

标志位函数

```
bitset
template<size_t N>
class bitset
{
    ……
};
实际使用的时候，N是个整型常数
    bitset<40> bst;
    bst是一个由40位组成的对象，
    用bitset的函数可以方便地访问任何一位
成员函数：
bitset<N>& operator&=(const bitset<N>& rhs);
bitset<N>& operator|=(const bitset<N>& rhs);
bitset<N>& operator^=(const bitset<N>& rhs);
bitset<N>& operator<<=(size_t num);
bitset<N>& operator>>=(size_t num);
bitset<N>& set(); //全部设成1
bitset<N>& set(size_t pos, bool val = true); //设置某位
bitset<N>& reset(); //全部设成0
bitset<N>& reset(size_t pos); //某位设成0
bitset<N>& flip(); //全部翻转
bitset<N>& flip(size_t pos); //翻转某位
reference operator[](size_t pos); //返回对某位的引用
bool operator[](size_t pos) const; //判断某位是否为1
reference at(size_t pos);
bool at(size_t pos) const;
unsigned long to_ulong() const; //转换成整数
string to_string() const; //转换成字符串
size_t count() const; //计算1的个数
size_t size() const;
bool operator==(const bitset<N>& rhs) const;
bool operator!=(const bitset<N>& rhs) const;
bool test(size_t pos) const; //测试某位是否为 1
bool any() const; //是否有某位为1
bool none() const; //是否全部为0
bitset<N> operator<<(size_t pos) const;
bitset<N> operator>>(size_t pos) const;
bitset<N> operator~();
static const size_t bitset_size = N;
注意：第0位在最右边
```

例子

```
bool Greater10(int n){
    return n > 10;
}
int main() {
    const int SIZE = 10;
    int a1[] = { 2,8,1,50,3,100,8,9,10,2 };
    vector<int> v(a1,a1+SIZE);
    ostream_iterator<int> output(cout," ");
    vector<int>::iterator location;
    location = find(v.begin(),v.end(),10);
    if( location != v.end())//是否找到
        cout << endl << "1) " << location - v.begin();
    location = find_if( v.begin(),v.end(),Greater10);
    if( location != v.end())
        cout << endl << "2) " << location - v.begin();
    sort(v.begin(),v.end());//先排序
    if( binary_search(v.begin(),v.end(),9)) {//折半查找前提是排序好的
        cout << endl << "3) " << "9 found";
} }
```