第九周 标准模板库 STL（二）

第一节 关联容器

set/multiset/map/multimap

内部元素有序排列，新元素插入的位置决定于它的值，查找速度快。

除了各容器都有的函数外，还支持以下成员函数：

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

第二节 set 和 multiset

预备知识：pair 模板

```
template<class _T1, class _T2>
struct pair{
    typedef _T1 first_type;
    typedef _T2 second_type;
    _T1 first;
    _T2 second;
    pair():first(),second(){}//如果 first 和 second 是对象的话，就用无参构造函数来初始化
    pair(const _T1&__a,const _T2&__b):first(__a),second(__b){}
//构造函数有两个静态变量__a 和__b，分别用来初始化 first 和 second
    template<class _U1, class _U2>
    pair(const pair<_U1,_U2>&__p):first(__p.first),second(__p.second){}
};
```

第三个构造函数是一个函数模板，用以下例子来示例一下：

```
pair<int, int>
p(pair<double, double>(5.5,4.6));
//p.first = 5, p.second = 4
```

1.multiset

```
template<class Key, class Pred = less<Key>, class A = allocator<Key> >
class multiset {    …… };
```

在这里面，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>返回值为 true，则 x 比 y 小。**Pred 的缺省值类型是 less<Key>**。

less 模板：

```
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

表 2.1 multiset 的成员函数

函数名	作用
iterator find(const T &val)	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。（返回大于等于 val 的第一个数字的地址）
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。（返回比 val 大的第一个数字的地址）
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)。

（1）multiset 的用法

```
#include <set>
using namespace std;
class A {   };
int main()   {
    multiset<A> a;
    a.insert(A());//error
}
```

主函数中第一句话没有什么问题，但是第二句话存在问题。因为 multiset<A> a 等价于 multiset<A, less<A>> a，这样插入元素的时候，multiset 会将插入元素同已有元素进行比较。由于 a 中的元素都是 A 类型的，插入的也是一个 A 类型的，所以需要对 A 类型的对象进行比较，这要求 A 的对象能用<比较，即**需要对<进行重载**。

例程：

```
#include <iostream>
#include <set>
using namespace std;
template<class T>
void Print(T first, T last)
{
    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)//按个位数比较
    {return (a1.n%10)<(a2.n%10);}
};
typedef multiset<A> MSET1;
typedef multiset<A,MyLess> MSET2;//MSET 用 MyLess::operator()比较大小
int main(){
    const int SIZE = 6;
    A a[SIZE]={4,22,19,8,33,40};
    MSET1 m1;
    m1.insert(a,a+SIZE);
    m1.insert(22);
    cout<<"1)"<<m1.count(22)<<endl;//output:1)2
    cout<<"2)"<<Print(m1.begin(),m1.end());//output: 2)4 8 19 22 22 33 40
    MSET1::iterator pp = m1.find(19);
    if(pp!=m1.end())
        cout<<"found"<<endl;
    cout<<"3}";cout<<*m1.lower_bound(22)<<","<<*m1.upper_bound(22)<<endl;//output:22
23
    pp = m1.erase(m1.lower_bound(22),m1.upper_bound(22));//pp 指向被删除元素的下一个元
素
    cout<<"4)";Print(m1.begin(),m1.end());//输出 4 8 19 33 40
    cout<<"5)";cout<<*p>>endl;//输出 33
    MSET2 m2;
    m2.insert(a,a+SIZE);
    cout<<"6)";Print(m2.begin(),m2.end());//输出 6)40 22 33 4 8 19
    return 0;
}
```

以上程序已弄明白，但是要多看看，理解下。

2.set

```
template<class Key, class Pred = less<Key>, class A = allocator<Key>>
class set{}
```

插入 set 中已有元素时，忽略插入。例程如下：

```
#include <iostream>
#include <set>
using namespace std;
int main() {
    typedef set<int>::iterator IT;
```

```
    int a[5] = { 3,4,6,1,2 };
    set<int> st(a, a + 5);
    pair<IT,bool> result;
    result = st.insert(5);
    if (result.second)
        cout << *result.first << "inserted" << endl;
    if (st.insert(5).second) cout << *result.first << endl;
    else
        cout << *result.first << "already exists" << endl;
    pair<IT, IT>bounds = st.equal_range(4);
    cout << *bounds.first << "." << *bounds.second;
    system("pause");
    return 0;
}
```

定义的 result 是一个 pair 类型的对象，其里面的 first 是 IT 类型，second 是 bool 类型（pair 类定义见本节的预备知识）。下面得到的 result 对象里面的 IT 就是迭代器指向 5 所在位置，而 second 的值为 true（因为插入成功）。

if(st.insert(5).second)这句话实际上是先执行了 st.insert(5)之后，判断其 second 值是 false 还是 true。由于前面已经插入了 5，所以这里肯定插入不成功，所以 second 肯定是 false。

第三节 map 和 multimap

1.multimap

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。元素按照 first 成员变量从小到大排列，缺省情况下用 less<Key> 定义关键字的“小于”关系。

例程：

```
#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));
    cout << "2)" << pairs.count(15) << endl;//求关键字等于15的个数，应该输出
    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 << ")" << ",";
    cout << endl;
    system("pause");
    return 0;
}
```

multimap 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”。

输入样例：

Add Jack 12 78

Query 78

Query 81

Add Percy 9 81

Add Marry 8 81

Query 82

Add Tom 11 79

Query 80

Query 81

输出果样例：

Nobody

Jack 12 78

Percy 9 81

Tom 11 79

Tom 11 79

程序如下：

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
class CStudent {
public:
    struct  CInfo
    {
        int id;
```

```
        string name;
    };
    int score;
    CInfo info;
};
typedef multimap<int, CStudent::CInfo> MAP_STD;//pair模板只允许first和second两个参数
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));
            //mp.insert(make_pair(st.scorest.info));也可以
        }
        else if (cmd == "Query") {
            int score;
            cin >> score;
            MAP_STD::iterator p = mp.lower_bound(score);
            if (p != mp.begin()) {
                --p;
                score = p->first;//比要查询分数低的最高分
                MAP_STD::iterator maxp = p;//已经是比分数低的最高分的位置
                int maxId = p->second.id;
                for (; p != mp.begin() && p->first == score; --p) {
                    if (p->second.id > maxId)
                    {
                        maxp = p;
                        maxId = p->second.id;
                    }
                }
                if (p->first == score) {
                    if (p->second.id > maxId) {
                        maxp = p;
                        maxId = p->second.id;
                    }
                }
                cout << maxp->second.name << " " << maxp->second.id << " " <<
maxp->first << endl;
            }
            else
                cout << "Nobody" << endl;
            }
```

```
        }
    system("pause");
    return 0;
}
```

2.map

template<class Key, class T, class Pred = less<Key>,
class A = allocator<T> >
class map {
    typedef pair<const Key, T> value_type;
};

map 中的元素都是 pair 模板类对象。关键字（first 成员变量）各不相同。元素按照关键字从小到大排列，缺省情况下用 less<Key>，即“<”定义“小于”。

3.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 paris 中关键字为 50 的元素，使其值变为 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。

第四节 容器适配器

容器适配器没有迭代器！

1.stack

stack 是后进先出的数据结构，只能插入、删除和访问栈顶的元素。可以用 vector、list 和 deque 来实现，缺省使用 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{ };

stack 可以进行下面的操作：

（1）push：插入元素；

（2）pop：弹出元素；

（3）top：返回栈顶元素的引用。

2.Queue

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

template<class T, class Cont = deque<T> >
class queue{ };

同样有 push，pop 和 top 函数。但是 push 在队尾，pop 和 top 在队头。因为是先进先出。有 back 成员函数可以返回队尾元素的引用。

3.priority_queue

template <class T, class Container = vector<T>. class Compare = less<T> >

class priority_queue;

和 queue 类似，可以用 vector 和 deque 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。默认元素比较器是 less<T>。

push/pop 的时间复杂度是 O(logn)，top()的时间复杂度是 O(1)。

```
#include <queue>
#include <iostream>
using namespace std;
int main()
{
    priority_queue<double> p1;
    p1.push(3.2);p1.push(9.8);p1.push(9.8);p1.push(5.4);
    while(!p1.empty()){
        cout<<p1.top()<<" ";
        p1.pop();
    }//输出 9.8 9.8 5.4 3.2
    cout<<endl;
    priority_queue<double,vector<double>,greater<double> p2;
    p2.push(3.2);p2.push(9.8);p2.push(9.8);p2.push(5.4);
    while(!p1.empty()){
        cout<<p1.top()<<" ";
        p2.pop();
    }//输出 3.2 5.4 9.8 9.8
    return 0;
}
```

stack/queue/priority_queue 都有：empty()（用于判断适配器是否为空）和 size()（返回适配器中元素个数。）

第五节 算法

STL 中的算法大致可以分为以下七类：

（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

大多数重载的算法都是有两个版本：一个版本是用“==”判断元素是否相等，或用“<”来比较大小；还有一个版本是多出一个类型参数 Pred 和函数形参 Pred op，通过表达式 op(x,y)的返回值 true/false 来判断 x 是否等于 y 或者 x 是否小于 y 或者 x 是否大于 y。

1.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（特别是顺序容器和关联

容器）。它的时间复杂度是 **O(n)**的。它包括以下算法：

表 5.1 不变序列算法中的算法

算法名称	功能
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这种操作（**不能改变数值**）
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）
lexicographical_compare	按字典序比较两个区间的大小（可自定义比较器）

我们具体来看：

（1）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。

（2）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。

（3）for_each：

```
template<clas InIt, classFun>
Fun for)each(InIt first, InIt last, Fun f);
```

对区间[first,last)中的每个元素 e，执行 f(e)，要求 f(e)不能改变 e。

（4）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。

（5）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。

（6）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

（7）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。

2.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（因为关联容器是排好序的算法，如果直接值被修改，容器中顺序被打破，再去执行别的操作可能结果就不是预期结果）。**

表 5.2 变值算法的算法

算法名称	算法功能
for_each	对区间中的每个元素都做某种操作（**可以改变数值**）
copy	复制一个区间到别处
copy_backward	复制一个区间到别处，但目标区前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

我们来具体看一下：

（1）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)，x 可以和 first 相等。

（2）copy

template<class InIt, class OutIt>
OutIt copy(InIt first, InIt last, OutIt x);

本函数对每个在区间[0,last-first)中的 N 执行一次*(x+N)=*(first+N)，返回 x+N。copy 的

源代码如下：

```
template<class _II, class _OI>
inline _OI copy(_II_F, _II_L, __OI_X)
{
    for(;_F!=_L;++_X,++_F)
        *_X = *_F;
        return(_X);
}
```

它有两个类型，一个是 II 一个是 OI，暗示分别是输入和输出。函数的三个参数的前两个是区间的开始和结束，后面则可以是某个位置。这个函数做的就是在从_F 走到_L 的过程中，把*_F 的值赋给*_X，然后_F 和_X 不断后移。

对于 copy(v.begin(), v.end(), output);first 和 last 的类型是 vecotr<int>::const_iterator，output 的类型是 osream_iterator<int>。

3.删除算法

删除算法会删除一个容器里的某些元素。这里所说的“删除”，并不会使容器里的元素减少，其工作过程是：将所有应该被删除的元素看做空位子，然后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**。

表 5.3 删除算法的算法

算法名称	算法功能
remove	删除区间中等于某个值的元素
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
uqique	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)

算法复杂度均为 **O(n)**。

我们具体来看：

（1）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
```

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。

```
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);
    //输出语句，输出 1,3,2,5
    cout<<"2)"<<p-a<<endl;//输出 2)3。是指的元素中剩余的有效元素还有 3 个，删除了首
```

地址的元素。

```
    vector<int> v(b,b+6);
    remove(v.begin().v.end(),2);
    //输出语句，结果为 1,3,5,6,5,6
    return 0;
}
```

之所以第一次输出的结果为 1,3,2,5 是因为，当我们删除第一个 2 的时候，后面的 3 移过来，然后后面的 2 页被删了，再后面的 5 移过来，这样最后面空了 2 个位置，则这两个位置的原来的值保持不变。

4.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**。此类算法复杂度都是 O(n)的。

表 5.4 变序算法的算法

算法名称	算法功能
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)
prev_permutation	将区间改为上一个排列(可自定义比较器)
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
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

我们来具体看一下：

（1）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。**用之前要初始化伪随机数种子：**

srand(unsigned(time(NULL)));//需要#include<ctime>

（2）reverse

```
template<class BidIt>
void reverse(BidIt first, BidIt last);
```

颠倒区间[first,last)顺序。

（3）next_permutation

```
template<class InIt>
bool next_permutaion (Init first,Init last);
```

求下一个排列。

例程：

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

```
using namespace std;
int main()
{
	string str = "231";
	char szStr[] = "324";
	while (next_permutation(str.begin(), str.end())){
		cout << str << endl;//输出 312 321
	}
	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;
}
```

5.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**。

表 5.5 排序算法的算法

算法名称	算法功能
sort	将区间从小到大排序(可自定义比较器)。
stable_sort	将区间从小到大排序，并保持相等元素间的相对次序(可自定义比较器)。
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)。

我们来具体看一下：

（1）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) 是否为 true

sort 实际上是快速排序，时间复杂度 O(n*log(n))；平均性能最优。但是最坏的情况下，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**。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。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。

此外，其它排序算法：

partial_sort：部分排序，直到前 n 个元素就位即可。

nth_element：排序，直到第 n 个元素就位，并保证比第 n 个元素小的元素都在第 n 个元素之前即可。

partition：改变元素次序，使符合某准则的元素放在前面。

6.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，而且需要随机访问迭代器的支持。所以有序区间算法不能用于关联容器和 list。

表 5.6 有序区间算法的算法

算法名称	功能
binary_search	判断区间中是否包含某个元素。
includes	判断是否一个区间中的每个元素，都在另一个区间中。
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。
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。

我们来具体看一下：

（1）binary_search 二分查找

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 小于 y。

（2）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 做比较器

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。（**空间必须得充足**）

（3）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，第二个用 pr 作比较器,pr(x,y)==true 说明 x,y 相等。

（4）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。

（5）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)次。

（6）set_symmetric_difference

```
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);
```

把两个区间里相互不在另一区间里的元素放入 x 开始的地方。

（7）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)次。

图 5.1 set_union

7.bitset（非数值算法，是类模板）

template<size_t N>
class bitset {};

是实现标志位。实际使用的时候，N 是个整型常数如：

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。

bitset 的成员函数如下：

```
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。