程序设计实习

郭炜　微博 http://weibo.com/guoweiofpku
http://blog.sina.com.cn/u/3266490431
刘家瑛 微博 http://weibo.com/pkuliujiaying

标准模板库STL

set和multiset

关联容器

set, multiset, map, multimap

- 内部元素有序排列，新元素插入的位置取决于它的值，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

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() { }
    pair(const _T1& __a, const _T2& __b)
    : first(__a), second(__b) { }
    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
```

multiset

template<class Key, class Pred = less<Key>,

class A = allocator<Key> >

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) 返回值为true,则 x比y小。

  Pred的缺省类型是 less<Key>。

multiset

```
template<class Key, class Pred = less<Key>,
        class A = allocator<Key> >
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) 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);
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。

multiset的成员函数

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

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

multiset 的用法

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

multiset <A> a;

就等价于

multiset<A, less<A>> a;

插入元素时，multiset会将被插入元素和已有元素进行比较。由于less模板是用 < 进行比较的，所以,这都要求 A 的对象能用 < 比较，即适当重载了 <

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)
{    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 };
            MSET1 m1;
            m1.insert(a,a+SIZE);
            m1.insert(22);
            cout << "1) " << m1.count(22) << endl;       //输出 1) 2
            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);
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;
}
```

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。

输出：

1) 2

2) 4 8 19 22 22 33 40

3) 22,33

4) 4 8 19 33 40

5) 33

6) 40 22 33 4 8 19

set

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

插入set中已有的元素时，忽略插入。

set用法示例

```
#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);    // 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;
}
```

输出结果：

5 inserted
5 already exists
4,5

In-Video Quiz

1. 以下哪个对象定义语句是错的?

A)pair<int,int> a(3.4,5.5);

B)pair<string,int> b;

C)pair<string,int> k(pair<char*,double> (“this”,4.5));

D)pair<string,int> x(pair<double,int> b(3.4,100));

2. 要让下面一段程序能够编译通过，需要重载哪个运算符？

class A { };

multiset<A,greater<A> > b;

b.insert(A());

A)== B)= C)> D)<

In-Video Quiz

1. 以下哪个对象定义语句是错的?

A)pair<int,int> a(3.4,5.5);

B)pair<string,int> b;

C)pair<string,int> k(pair<char*,double> (“this”,4.5));

D)pair<string,int> x(pair<double,int> b(3.4,100));

2. 要让下面一段程序能够编译通过，需要重载哪个运算符？

class A { };

multiset<A,greater<A> > b;

b.insert(A());

A)== B)= C)> D)<

In-Video Quiz

3. 下面程序片段输出结果是：

```
int a[] = { 2,3,4,5,7,3};
multiset<int> mp(a,a+6);
cout << * mp.lower_bound(4);
```

A)2 B)3 C)4 D)5

4. set<double> 类的equal_range成员函数的返回值类型是：

A)void

B)pair<set<double>::iterator, set<double>::iterator>

C)pair<int,int>

D)pair<set<double>::iterator,bool>

程序设计实习

郭炜　微博 http://weibo.com/guoweiofpku
http://blog.sina.com.cn/u/3266490431
刘家瑛 微博 http://weibo.com/pkuliujiaying

标准模板库STL

map和multimap

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() { }
    pair(const _T1& __a, const _T2& __b)
    : first(__a), second(__b) { }
    template<class _U1, class _U2>
    pair(const pair<_U1, _U2>& __p)
            : first(__p.first), second(__p.second) { }
  };
```

map/multimap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
```

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。

multimap示例

```
#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;  //求关键字等于某值的元素个数
        pairs.insert(mmid::value_type(30,111.11));
        pairs.insert(mmid::value_type(10,22.22));
```

输出:

1) 0

2) 2

```
    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   << ")" << ",";
}
```

输出：

1) 0

2) 2

(10,22.22),(15,2.7),(15,99.3),(20,9.3),(25,33.333),(30,111.11)

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
```

```
#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()) {
                            --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;
```

```
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()) {
                            --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;
```

iterator lower_bound (const T & val);
查找一个最大的位置 it,使得 [begin(),it) 中所有元素的first 都比 val 小。

```
            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) {
        //如果上面循环是因为 p == mp.begin()
        // 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;
}
```

```
                                        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;
}

 mp.insert(MAP_STD::value_type(st.score,st.info ));
//mp.insert(make_pair(st.score,st.info )); 也可以
```

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”。

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.

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示例

```
#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));
    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;
```

输出：

1) 0

2) 1

3) (15,2.7),(20,9.3),

```
    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  << ",";
}
```

输出：

1) 0

2) 1

3) (15,2.7),(20,9.3),

4) (15,2.7),(20,9.3),(40,0),

5) (15,6.28),(20,9.3),(40,0),

In-Video Quiz

1. 下面三段程序，哪个不会导致编译出错?(提示，本题非常坑)

A)multimap <string,greater<string> > mp;

B)multimap <string,double,less<int> > mp1; mp1.insert(make_pair(“ok”,3.14));

C)multimap<string,double> mp2; mp2.insert(“ok”,3.14);

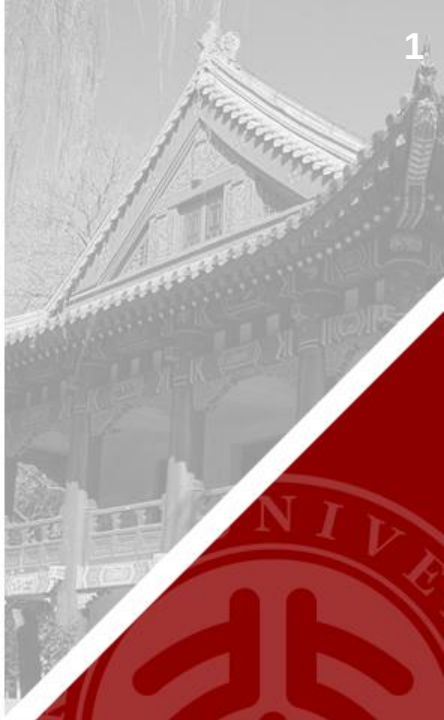
D)都会导致编译出错

2. 有对象map<string,int> mp; 则表达式mp[“ok”] 的返回值类型是：

A)Int B)int & C)string D)string &

容器适配器

郭 炜 刘家瑛



北京大学

容器适配器

- 可以用某种顺序容器来实现

(让已有的顺序容器以栈/队列的方式工作)

- 1) stack: 头文件 <stack>
  - 栈 -- 后进先出
- 2) queue: 头文件 <queue>
  - 队列 -- 先进先出
- 3) 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

容器适配器

- 都有3个成员函数:
  - push: 添加一个元素;
  - top: 返回栈顶部或队头元素的引用
  - pop: 删除一个元素
- 容器适配器上没有迭代器

→ STL中各种排序, 查找, 变序等算法都不适合容器适配器

stack

- stack 是后进先出的数据结构
- 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 {
  …
};
```

- stack 中主要的三个成员函数:
  - void push(const T & x);

    将x压入栈顶
  - void pop();

    弹出(即删除)栈顶元素
  - T & top();

    返回栈顶元素的引用. 通过该函数, 可以读取栈顶元素的值, 也可以修改栈顶元素

queue

- 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 {
……
};
```

- 同样也有push, pop, top函数
  - push发生在队尾
  - pop, top发生在队头, 先进先出

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
  - 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引用
- 默认的元素比较器是 less<T>

```
#include <queue>
#include <iostream>
using namespace std;
int main()  {
      priority_queue<double> priorities;
      priorities.push(3.2);
      priorities.push(9.8);
      priorities.push(5.4);
      while( !priorities.empty() ) {
            cout << priorities.top() << " ";
            priorities.pop();
      }
      return 0;
} //输出结果: 9.8 5.4 3.2
```

STL算法

郭 炜 刘家瑛

北京大学

STL算法分类

- STL中的算法大致可以分为以下七类:
  - 不变序列算法
  - 变值算法
  - 删除算法
  - 变序算法
  - 排序算法
  - 有序区间算法
  - 数值算法

算法

- 大多重载的算法都是有两个版本的
  - 用 “==” 判断元素是否相等, 或用 “<” 来比较大小
  - 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: ture/false

    →判断x是否 “等于” y，或者x是否 “小于” y
- 如下面的有两个版本的min_element:

  iterator min_element(iterator first, iterator last);

  iterator min_element(iterator first, iterator last, **Pred op**);

1. 不变序列算法

- 该类算法不会修改算法所作用的容器或对象
- 适用于顺序容器和关联容器
- 时间复杂度都是O(n)

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某种操作

1. 不变序列算法

算法名称	功 能
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)

1. 不变序列算法

算法名称	功 能
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)
lexicographical_compare	按字典序比较两个区间的大小(可自定义比较器)

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

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(x==y为true算等于)

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(不小于任何其他元素)的迭代器
- 以 “<” 作比较器

```
#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” << endl;
       if( a1.n == 3 && a2.n == 7 )
             return true;
       return false;
}
```

```
int main()  {
      A aa[] = { 3,5,7,2,1 };
      cout << min_element(aa,aa+5)->n << endl;
      cout << max_element(aa,aa+5)->n << endl;
      return 0;
}
```

输出：

```
< called
< called
< called
< called
3
< called
< called
< called
< called
7
```

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

算法名称	功 能
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
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

2. 变值算法

算法名称	功 能
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

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

```
#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," ");
    random_shuffle(v.begin(),v.end());
    cout << endl << "1) ";
    copy( v.begin(),v.end(),output);
    copy( a2,a2+SIZE,v.begin());
    cout << endl << "2 ) ";
    cout << count(v.begin(),v.end(),8);
    cout << endl << "3 ) ";
    cout << count_if(v.begin(),v.end(),CLessThen9());
```

输出:

1) 5 4 1 3 7 8 9 10 6 2

2) 2

3) 6

//1) 是随机的

```
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
```

输出:

4) 1

5) 100

6) 193

```
    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);
    return 0;
}
```

输出：

7)10000 4 64 1 2500 9 64 81 100 4

8)1 8 27 64 125 216 343 512 729 1000

ostream_iterator<int> output(cout ,“ ”);

- 定义了一个 ostream_iterator<int> 对象，
  可以通过cout输出以 “ ”(空格) 分隔的一个个整数

copy (v.begin(), v.end(), output);

- 导致v的内容在 cout上输出

copy 函数模板(算法)

```
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

对于copy(v.begin(),v.end(),output);

- first 和 last 的类型是 vector<int>::const_iterator
- output 的类型是 ostream_iterator<int>

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);
}
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;
int main(){
    int a[4] = { 1,2,3,4 };
    My_ostream_iterator<int> oit(cout,"*");
    copy(a,a+4,oit);    //输出 1*2*3*4*
    ofstream oFile("test.txt", ios::out);
    My_ostream_iterator<int> oitf(oFile,"*");
    copy(a,a+4,oitf);    //向test.txt文件中写入 1*2*3*4*
    oFile.close();
    return 0;}
// 如何编写 My_ostream_iterator?
```

- 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);
}
```

- 上面程序中调用语句 “copy(a,a+4,oit)” 实例化后得到copy如下:

```
My_ostream_iterator<int> copy(int * _F, int * _L, My_ostream_iterator<int> _X)
{
    for (; _F != _L; ++_X, ++_F)
            *_X = *_F;
    return (_X);
}
```

- My_ostream_iterator类应该重载 “++” 和 “*” 运算符
- “=” 也应该被重载

```
#include <iterator>
template<class T>
class My_ostream_iterator:public iterator<output_iterator_tag, T>{
    private:
        string sep; //分隔符
        ostream & os;
    public:
        My_ostream_iterator(ostream & o, string s):sep(s), os(o){ }
        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;      }
};
```

In-Video Quiz

1. find算法判断两个元素相等时，用哪个运算符？
A)== B)= C)< D)!=

2. 若要使下面这段程序编译能够通过，以下哪个表达式不是必须有定义的？
class A { };
A obj;
int a[] = {1,2,3,4,5};
copy(a,a+5,obj);
A)++obj
B)-- obj
C)*obj
D)以上三个都可以没定义

3. 删除算法

- 删除一个容器里的某些元素
- 删除 -- 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

3. 删除算法

算法名称	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
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)

- 算法复杂度都是O(n)的

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 (x,y)为 true说明x和y相等
- 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

```
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,
    cout << "2) " << p - a << endl;    //输出 2) 3
    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;
}
```

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
- 算法复杂度都是O(n)的

算法名称	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

4. 变序算法

算法名称	功 能
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

4. 变序算法

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

```
#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;
	 }
	 cout << "****" << endl;
	 while (next_permutation(szStr,szStr + 3)){
		cout << szStr << endl;
	 }
```

输出:
312
321
****
342
423
432

```
    sort(str.begin(), str.end());
    cout << "****" << endl;
    while (next_permutation(str.begin(), str.end()))
    {
        cout << str << endl;
    }
    return 0;
}
```

输出:
132
213
231
312
321

```
#include <iostream>
#include <algorithm>
#include <string>
#include <list>
#include <iterator>
using namespace std;
int main(){
    int a[] = { 8,7,10 };
    list<int> ls(a, a+3);
    while( next_permutation(ls.begin(), ls.end())) {
        list<int>::iterator i;
        for( i = ls.begin(); i != ls.end(); ++i)
            cout << * i << " ";
        cout << endl;
    }
}
```

输出：

8 10 7

10 7 8

10 8 7

5. 排序算法

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

算法名称	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)

5. 排序算法

算法名称	功能
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）

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

```
#include <iostream>
#include <algorithm>
using namespace std;
class MyLess  {
public:
    bool operator()( int n1,int n2) {
        return (n1 % 10) < ( n2 % 10);
    }
};
```

按个位数大小排序,
按降序排序
输出:
111 2 14 78 9
111 78 14 9 2

```
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>());
    for( i = 0;i < 5;i ++)
        cout << a[i] << " ";
}
```

- 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

6. 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

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

6. 有序区间算法

算法名称	功能
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

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小于y

```
#include <vector>
#include <bitset>
#include <iostream>
#include <numeric>
#include <list>
#include <algorithm>
using namespace std;
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()) {
        cout << endl << "1) " << location - v.begin();
    }
    location = find_if( v.begin(),v.end(),Greater10);
    if( location != v.end())
        cout << endl << "2) " << location - v.begin();
```

输出:

1) 8

2) 3

```
    sort(v.begin(),v.end());
    if( binary_search(v.begin(),v.end(),9)) {
        cout << endl << "3) " << "9 found";
    }
}
```

输出：

1) 8

2) 3

3) 9 found

lower_bound, uper_bound, equal_range

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

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

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)次

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

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

bitset

```
template<size_t N>
class bitset
{
      .....
};
```

- 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:

- 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

In-Video Quiz

1. 下面的一些算法，哪个可以用于关联容器？

A)find B)sort C)remove D)random_shuffle