

新标准C++程序设计

北京大学信息学院 郭 炜

GWPL@PKU.EDU.CN



标准模板库(三)



pair 模板:

```
template<class T, class U>
struct pair {
    typedef T first_type;
    typedef U second_type;
    T first; U second;
    pair();
    pair(const T& x, const U& y);
    template<class V, class W>
        pair(const pair<V, W>& pr);
};
```

pair模板可以用于生成 key-value对



```
#include <set>
```

```
#include <iostream>
```

```
#include <iterator>
```

```
using namespace std;
```

```
main() {
```

```
    typedef set<double,less<double> > double_set;
```

```
    const int SIZE = 5;
```

```
    double a[SIZE] = {2.1,4.2,9.5,2.1,3.7 };
```

```
    double_set doubleSet(a, a+SIZE);
```

```
    ostream_iterator<double> output(cout, " ");
```

```
    cout << "1) ";
```

```
    copy(doubleSet.begin(),doubleSet.end(),output);
```

```
    cout << endl;
```

输出:

1) 2.1 3.7 4.2 9.5



```
pair<double_set::const_iterator, bool> p;
```

```
p = doubleSet.insert(9.5);
```

```
if( p.second )
```

```
    cout << "2) " << * (p.first) << " inserted" << endl;
```

```
else
```

```
    cout << "2) " << * (p.first) << " not inserted" << endl;
```

```
}
```

//insert函数返回值是一个pair对象, 其first是被插入元素的迭代器, second代表是否成功插入了

输出:

1) 2.1 3.7 4.2 9.5

2) 9.5 not inserted



6.3 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
```

- multimap中的元素由 <关键字,值>组成，每个元素是一个pair对象。

multimap 中允许多个元素的关键字相同。元素按照关键字升序排列，缺省情况下用 less<Key> 定义关键字的“小于”关系。

```
#include <iostream>
#include <map>
using namespace std;
int main() {
```

输出:

1) 0

2) 2

```
    typedef multimap<int,double,less<int> > mmid;
    mmid pairs;
    cout << "1) " << pairs.count(15) << endl;
    pairs.insert(mmid::value_type(15,2.7));
    pairs.insert(mmid::value_type(15,99.3));
    cout << "2) " << pairs.count(15) << endl;
    pairs.insert(mmid::value_type(30,111.11));
    pairs.insert(mmid::value_type(10,22.22));
```

//求某关键值的个数



```
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)




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 19.4.4.1.cpp



6.4 map

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

- map 中的元素关键字各不相同。元素按照关键字升序排列，缺省情况下用 less 定义“小于”。



- 可以用 `pairs[key]` 访问map中的元素。`pairs` 为map容器名，`key`为关键字的值。该表达式返回的是对关键值为`key`的元素的值的引用。如果没有关键字为`key`的元素，则会往`pairs`里插入一个关键字为`key`的元素，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



```
#include <iostream>
```

```
#include <map>
```

```
using namespace std;
```

```
ostream & operator <<( ostream & o,const pair< int,double>  
    & 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 ++ )
```

输出：

1) 0

```
    cout << * i << ",";
```

2) 1

```
cout << endl;
```

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

```
cout << "5) ";
```

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

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

```
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),



7 容器适配器

7.1 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 上可以进行以下操作：

push: 插入元素

pop: 弹出元素

top: 返回栈顶元素的引用



7.2 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。

但是push发生在队尾； pop, top发生在队头。先进先出。



7.3 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引用。默认的元素比较器是 less<T>。



```
#include <queue>
#include <iostream>
using namespace std;
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();
    }
} //输出结果: 9.8 5.4 3.2
```



8 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



8 算法

大多重载的算法都是有二个版本的，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true还是false，来判断x是否“等于”y，或者x是否“小于”y。如下面的有二个版本的min_element:

```
iterate min_element(iterate first,iterate last);  
iterate min_element(iterate first,iterate last, Pred op);
```



8.1 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。它们的时间复杂度都是 $O(n)$ 的。

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



8.1 不变序列算法

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)

8.1 不变序列算法

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)

8.1 不变序列算法

lexicographical_compare

按字典序比较两个区间的大小(可自定义比较器)



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 $a \neq b$, $a < b$ 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。

```

#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



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`

- 19.7.1.cpp 不变序列算法



8.2 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。

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



8.2 变值算法

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



8.2 变值算法

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 l ,
执行 $uop(*l)$; 并将结果依次放入从 x 开始的地方。
要求 $uop(*l)$ 不得改变 $*l$ 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; }
```



```
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) ";
```

输出:

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

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

```
    copy( a2,a2+SIZE,v.begin());
```

2) 2

```
    cout << endl << "2) ";
```

3) 6

```
    cout << count(v.begin(),v.end(),8);
```

//1) 是随机的

```
    cout << endl << "3) ";
```

```
    cout << count_if(v.begin(),v.end(),CLessThan9());
```



```
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);  
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);  
}
```

输出:

4)1

5)100

6)193

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

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



输出:

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

2) 2

3) 6

4) 1

5) 100

6) 193

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

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

19.8.1.cpp 变值算法示例



8.3 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。



8.3 删除算法

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)

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。



```
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;
}
```



8.4 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。

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



8.4 变序算法

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



8.4 变序算法

stable_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。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_permutation (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

132

213

231

312

321



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

输出

312

321

342

423

432

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

19.10.1.cpp 变序算法示例



8.5 排序算法

排序算法比前面的变序算法复杂度更高，一般是 $O(n \times \log(n))$ 。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)。

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)。



8.5 排序算法

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)。



8.5 排序算法

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);
    }
};
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] << " ";
}
```

按个位数大小排序，以及
按降序排序
输出：

111 2 14 78 9
111 78 14 9 2



- **sort** 实际上是快速排序，时间复杂度 $O(n \cdot \log(n))$ ；
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 $n \cdot \log(n)$ ，否则复杂度为 $n \cdot \log(n) \cdot \log(n)$ 。
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**。



此外还有其他排序算法：

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

...



堆排序

堆：一种二叉树，最大元素总是在堆顶上，二叉树中任何节点的子节点总是小于或等于父节点的值

◆ 什么是堆？

n 个记录的序列，其所对应的关键字的序列为 $\{k_0, k_1, k_2, \dots, k_{n-1}\}$ ，若有如下关系成立时，则称该记录序列构成一个堆。

$k_i \geq k_{2i+1}$ 且 $k_i \geq k_{2i+2}$ ，其中 $i=0, 1, \dots$ ，

◆ 例如,下面的关键字序列构成一个堆。

96 83 27 38 11 9

y r p d f b k a c

◆ 堆排序的各种算法，如make_heap等，需要随机访问迭代器的支持。



make_heap 函数模板

```
template<class RanIt>
```

```
void make_heap(RanIt first, RanIt last);
```

将区间 [first,last) 做成一个堆。用 < 作比较器

```
template<class RanIt, class Pred>
```

```
void make_heap(RanIt first, RanIt last, Pred pr);
```

将区间 [first,last) 做成一个堆。用 pr 作比较器



push_heap 函数模板

```
template<class RanIt>
```

```
void push_heap(RanIt first, RanIt last);
```

```
template<class RanIt, class Pred>
```

```
void push_heap(RanIt first, RanIt last, Pred pr);
```

- 在 $[first, last-1)$ 已经是堆的情况下，该算法能将 $[first, last)$ 变成堆，时间复杂度 $O(\log(n))$ 。
- 往已经是堆的容器中添加元素，可以在每次 `push_back` 一个元素后，再调用 `push_heap` 算法。



pop_heap 函数模板

取出堆中最大的元素

```
template<class RanIt>
```

```
void pop_heap(RanIt first, RanIt last);
```

```
template<class RanIt, class Pred>
```

```
void pop_heap(RanIt first, RanIt last, Pred pr);
```



- 将堆中的最大元素，即 $*first$ ，移到 $last - 1$ 位置，
原 $*(last - 1)$ 被移到前面某个位置，并且移动后 $[first, last - 1)$ 仍然是个堆。
要求原 $[first, last)$ 就是个堆。
- 复杂度 $O(\log(n))$
- 19.11.1.cpp 排序算法示例



8.6 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。

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。



8.6 有序区间算法

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



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, upper_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 Inlt1, class Inlt2, class Outlt>
```

```
Outlt merge(Inlt1 first1, Inlt1 last1, Inlt2 first2, Inlt2 last2,  
            Outlt x);用 < 作比较器
```

```
template<class Inlt1, class Inlt2, class Outlt, class Pred>
```

```
Outlt merge(Inlt1 first1, Inlt1 last1, Inlt2 first2, Inlt2 last2,  
            Outlt 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 Inlt1, class Inlt2>
```

```
bool includes(Inlt1 first1, Inlt1 last1, Inlt2 first2, Inlt2 last2);
```

```
template<class Inlt1, class Inlt2, class Pred>
```

```
bool includes(Inlt1 first1, Inlt1 last1, Inlt2 first2, Inlt2 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 Inlt1, class Inlt2, class Outlt>
```

```
Outlt set_difference(Inlt1 first1, Inlt1 last1, Inlt2 first2, Inlt2  
    last2, Outlt x);
```

```
template<class Inlt1, class Inlt2, class Outlt, class Pred>
```

```
Outlt set_difference(Inlt1 first1, Inlt1 last1, Inlt2 first2, Inlt2  
    last2, Outlt 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 Inlt1, class Inlt2, class Outlt>
```

```
Outlt set_union(Inlt1 first1, Inlt1 last1, Inlt2 first2, Inlt2 last2,  
                Outlt x); 用<比较大小
```

```
template<class Inlt1, class Inlt2, class Outlt, class Pred>  
Outlt set_union(Inlt1 first1, Inlt1 last1, Inlt2 first2, Inlt2  
last2, Outlt 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。

12. `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

