



北京大学  
PEKING UNIVERSITY

信息科学技术学院

# 程序设计实习

## C++面向对象程序设计

郭 炜

微博: <http://weibo.com/guoweiofpku>



北京大学  
PEKING UNIVERSITY

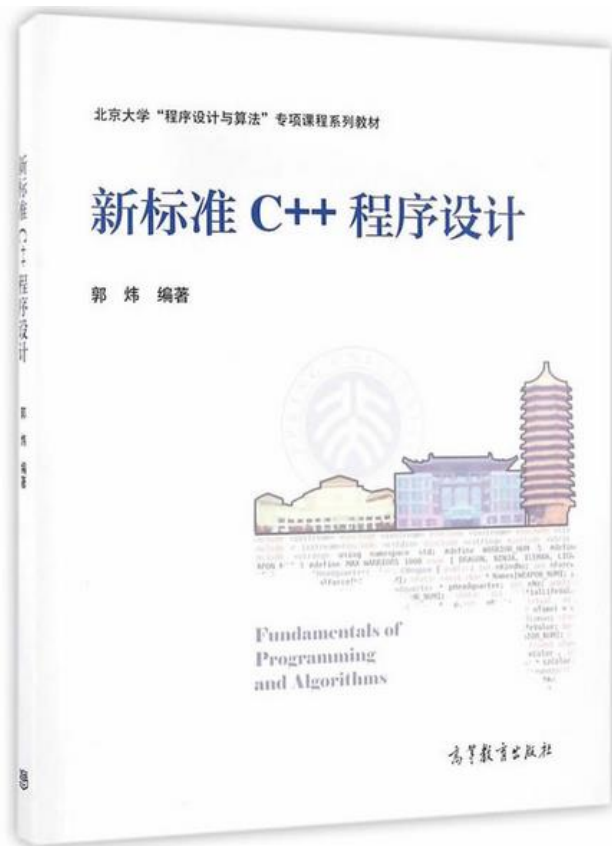
信息科学技术学院

配套教材：

高等教育出版社

《新标准C++程序设计》

郭炜 编著





# 标准模板库STL

(二)



北京大学  
PEKING UNIVERSITY

信息科学技术学院 郭炜

set和multiset



瑞士少女峰

# 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

以下说法错误的是

- ☐ A multiset里可以有重复的元素
- ☒ B 要将对象放入multiset，则必须重载能比较两个对象大小的 "<"
- ☐ C 不可以对multiset中的元素进行修改，只能添加或者删除
- ☐ D multiset插入元素的复杂度是 $O(\log n)$

提交

# 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的insert成员函数返回值是:

- ☐ A 被插入元素的迭代器
- ☐ B 被插入元素的后面元素的迭代器
- ☐ C 没有返回值
- ☒ D 以上都不对

提交

# 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



北京大学  
PEKING UNIVERSITY

信息科学技术学院 郭炜

map和multimap



瑞士少女峰

## 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模板类的对象,

`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模板类的对象,

**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),



以下说法哪个正确

- ☒ A multimap没有重载 [] 运算符
- ☐ B map没有重载 [] 运算符
- ☐ C 假设a是某map<int,string>对象, a中没有关键字是3的元素, 则 a[3]=5; 导致runtime error
- ☐ D map和multimap都重载了[] 运算符

提交



北京大学  
PEKING UNIVERSITY

信息科学技术学院 郭炜

## 容器适配器



瑞士少女峰

# 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

## 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。
- 有 back成员函数可以返回队尾元素的引用

# 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。默认的元素比较器是 less<T>。

## priority\_queue

- push、pop 时间复杂度 $O(\log n)$
- top()时间复杂度 $O(1)$

# priority\_queue

```
#include <queue>
#include <iostream>
using namespace std;
int main()
{
    priority_queue<double> pq1;
    pq1.push(3.2); pq1.push(9.8); pq1.push(9.8); pq1.push(5.4);
    while( !pq1.empty() ) {
        cout << pq1.top() << " ";
        pq1.pop();
    } //上面输出 9.8 9.8 5.4 3.2
```

```
cout << endl;
priority_queue<double,vector<double>,greater<double> > pq2;
pq2.push(3.2); pq2.push(9.8); pq2.push(9.8); pq2.push(5.4);
while( !pq2.empty() ) {
    cout << pq2.top() << " ";
    pq2.pop();
}
//上面输出 3.2 5.4 9.8 9.8
return 0;
}
```



# 容器适配器的元素个数

stack,queue,priority\_queue 都有

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



# 标准模板库STL

(算法)

# 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

# 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ture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);
```

# 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

# 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)

# 不变序列算法

## **adjacent find**

在区间中查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)

# 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,a1="
        << a1.n << " a2=" << a2.n << 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,a1=5 a2=3
< called,a1=7 a2=3
< called,a1=2 a2=3
< called,a1=1 a2=3
3
< called,a1=3 a2=5
< called,a1=3 a2=7
< called,a1=7 a2=2
< called,a1=7 a2=1
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

# 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

## 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

# 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) ";  
    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(),CLessThan9());  
}
```

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  
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

# 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。

# 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;
}
```



## 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

## 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

## 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。

用之前要初始化伪随机数种子:

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

## 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



## 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)。

# 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)。

# 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

...

以下有几个算法可以用于关联容器  
find, sort, binary\_search, lower\_bound,

☒ A 1

☐ B 2

☐ C 3

☐ D 4

提交



# 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

## 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。

# 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 Inlt1, class Inlt2, class Outlt>
```

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

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

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

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

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

```
Outlt set_symmetric_difference(Inlt1 first1, Inlt1 last1, Inlt2 first2,  
                               Inlt2 last2, Outlt 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