

第十一讲 标准模板库3

—— STL算法

刘家瑛

北京大学



上节课回顾

- 函数对象
 - 函数指针, 函数模板和函数对象
 - accumulate, greater
- 关联容器 multiset / set / multimap / map
 - multiset / set 容器中的对象类 **必须重载<**
 - find, lower_bound / upper_bound, count, insert
 - less 函数模板
 - pair对象
- 容器适配器
 - stack
 - queue
 - priority_queue



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
- 适用于所有容器
- 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)



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(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



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 $pr(*i) == true$



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, 但目标区间是从后往前被修改的
transform	将一个区间的元素变形后拷贝到另一个区间



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; }
```



```
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(),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
```

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



3. 删除算法

- 删除 一个容器里的某些元素
- 删除 一 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)



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;
}
```




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_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



```
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 $O(n\log(n))$
- 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）



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”
- ▲ 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 $O(n \cdot \log(n))$
 - 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:

算法名称	功 能
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

- 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, 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 n_1 次, 在 $[first2, last2)$ 里出现 n_2 次, 则该元素在目标区间里出现 $\min(n_1, n_2)$ 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



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