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 O(long(n))
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