Module04-04
C++ 标准库：算法与函数对象

- 数据结构简介
- 标准容器
- 常用算法简介
- → 算法与函数对象
- 迭代器
- 字符串
- I/O 流
- 数值

标准库 - 算法与函数对象

- 算法与函数对象 (STL Algorithms & Functor)
  - 标准算法一览
  - 函数对象
  - 非修改性算法
  - 修改性算法
  - 排序算法
  - 堆算法
  - 最大和最小
  - 排列算法
  - C 风格的算法

算法与函数对象 - 标准算法一览

- 非修改性操作列表

for_each()	对序列中每个元素执行某操作
find()	在序列中查找某个值的第一次出现的位置
find_if()	在序列中查找复合某个条件的第一个元素
find_first_of()	在一个序列中查找处另一个序列里的值
adjacent_find()	查找相邻的两个值
count()	统计某个值在序列中出现的次数
count_if()	统计序列中符合某种条件的次数
mismatch()	找出两个序列中相异的第一个元素
equal()	如果两个序列对应的元素都相同则为真
search()	找出一个序列作为子序列的第一个出现的位置
find_end()	找出一个序列作为子序列的最后一个出现位置
search_n()	找出一个序列作为子序列的第 n 个出现位置

算法与函数对象 - 标准算法一览

- 修改性操作列表

transform()	将操作应用于序列中每个元素
copy()	复制序列中的元素
copy_backward()	从序列后面开始复制
swap()	交换 2 个元素
iter_swap()	交换迭代器指向的 2 个元素
swap_ranges()	交换 2 个序列中的元素
replace()	替换
replace_if()	替换符合某种条件的元素
replace_cpoy()	复制序列时用给定值替换元素
replace_copy_if()	赋值序列时替换满足某种条件的元素
fill()	用一个给定值取代序列中的所有元素
fill_n()	用一个给定值取代序列中的前 n 个元素
generate()	用一个操作的结果取代所有元素

算法与函数对象 - 标准算法一览

- 修改性操作列表（续）

generate_n()	用一个操作的结果取代前 n 个元素
remove()	删除具有给定值的元素
remove_if()	删除符合某种条件的元素
remove_copy()	复制序列时删除符合给定值的元素
remove_copy_if()	复制序列时删除符合某种条件的元素
unique()	删除相邻的重复元素
unique_copy()	复制序列时删除重复的元素
reverse()	反转序列
reverse_copy()	复制时反转序列
rotate()	循环移动元素
rotate_copy()	复制序列时循环移动元素
random_shuffle()	随机打乱元素的次序

算法与函数对象 - 标准算法一览

- 排序操作列表

sort()	排序（很好的平均效率）
stable_sort()	稳定排序
partial_sort()	将序列的前一部分排序
partial_sort_copy()	赋值且将序列的前一部分排好序
nth_element()	将第 n 个元素放到其正确的位置
lower_bound()	找到某个值的第一个出现
upper_bound()	找到大于某个值的的第一个元素
equal_range()	找出匹配某个值的一个子序列
binary_search()	二分搜索，在已序序列中查找某个元素
merge()	合并 2 个已序的序列
inplace_merge()	合并 2 个接续的排好序的序列
partition()	将符合某种条件的元素放到前面
stable_partition()	将符合某种条件的元素放到前面，且维持原来的次序

算法与函数对象 - 标准算法一览

- 集合操作列表

include()	判断一个序列是否为另一个序列的子序列
set_union()	构建已序序列的并集
set_intersection()	构建已序序列的交集
set_difference()	构建已序序列的差集
set_symmetric_defference()	构建一个已序序列，只包含两个序列中不同的元素

- 堆操作列表

make_heap()	将一个序列堆化
push_heap()	向堆中添加一个元素
pop_heap()	将堆顶元素删除
sort_heap()	对堆排序

算法与函数对象 - 标准算法一览

- 最大和最小操作列表

max()	两个值中的较大者
min()	两个值中的较小者
max_element()	序列中最大的元素
min_element()	序列中最小的元素
lexicographic_compare()	按字典次序比较

- 排列操作列表

next_permutation()	按字典序的下一个排列
prev_permutation()	按字典序的前一个排列

算法与函数对象 - 序列

- 关于序列
  - 一个序列通常指某个容器中的一个区间（或整个容器）
  - 标准算法中，区间为半开区间，包含区间的开始位置但不含区间的结束位置
    - 由 first, last 构成的区间，包含 first ，但不含 last
    - 区间通常由迭代器来指示

- 函数对象的基类
  - 一元函数对象和二元函数对象（定义于 <functional> ）

```
// 一元函数对象
template<typename Arg, typename Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};

// 二元函数对象
template<typename Arg1, typename Arg2, typename Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

- 判断式 (Predicates)
  - 判断式是返回 bool 的函数对象（或函数），一个判断式的定义大致如下：

```
// 一元判断式
template<typename T>
struct logical_not: public unary_function<T, bool> {
    bool operator()(const T& x) const {
        return !x;
    }
};

// 二元判断式
template<typename T>
struct less: public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const {
        return x < y;
    }
};
```

算法与函数对象 - 函数对象

- 预定义的判断式

equal_to	二元	arg1 == arg2
not_equal_to	二元	arg1 != arg2
greater	二元	arg1 > arg2
less	二元	arg1 < arg2
greater_equal	二元	arg1 >= arg2
less_equal	二元	arg1 <= arg2
logical_and	二元	arg1 && arg2
logical_or	二元	arg1 || arg2
logical_not	一元	!arg

算法与函数对象 - 函数对象

- 算术函数对象

plus	二元	arg1 + arg2
minus	二元	arg1 - arg2
multiplies	二元	arg1 * arg2
divides	二元	arg1 / arg2
modulus	二元	arg1 % arg2
negate	一元	-arg

算法与函数对象 - 函数对象

- Binder、 Adapter、 Negater
  - Binder： 通过将一个参数约束到一个值，这样便可以将两个参数的函数对象当作一个参数的函数对象使用
  - Member Function Adapter： 使类的成员函数可以作为算法的参数
  - Function Pointer Adapter： 使函数指针可以当算法的参数
  - Negater： 表示某个判断式 (Predicate) 的否定
- Adapter 的实现：
  - 每个 Adapter 都提供了一个协助函数，以一个函数对象为参数，返回另一个合适的函数对象

算法与函数对象 - 函数对象

- Binder、 Adapter、 Negater（列表）

函数	函数对象	说明
bind2nd(y)	binder2nd	以 y 为第二参数调用 2 元函数
bind1st(x)	binder1st	以 x 为第一参数调用 2 元函数
mem_fun()	mem_fun_t	通过指针调用 0 元成员函数
	mem_fun1_t	通过指针调用 1 元成员函数
	const_mem_fun_t	通过指针调用 0 元 const 成员函数
	const_mem_fun1_t	通过指针调用 1 元 cosnt 成员函数
mem_fun_ref()	mem_fun_ref_t	通过引用调用 0 元成员函数
	mem_fun_ref1_t	通过引用调用 1 元成员函数
	const_mem_fun_ref_t	通过引用调用 0 元 const 成员函数
	const_mem_fun_ref1_t	通过引用调用 1 元 cosnt 成员函数
ptr_fun()	pointer_to_unary_function	调用 1 元函数指针
	pointer_to_binary_function	调用 2 元函数指针
not1()	unary_negate	否定 1 元判断式
not2()	binary_negate	否定 2 元判断式

算法与函数对象 - 函数对象

- Binder、 Adapter、 Negater（示例）

```
class Shape {
public:
    virtual void draw();
    virtual void rotate(double ang);
};

void f(vector<Shape*>& s, double ang) {
    // mem_fun Adapter
    for_each(s.begin(), s.end(), mem_fun(&Shape::draw));

    const int SIZE = 9;
    int a[SIZE] = { 12, 45, 23, 2, 89, 7, 6, 66, 5 };
    // Binder
    int* p = find_if(a, a + SIZE, bind2nd(less<int> (), 8));
}
```

算法与函数对象 - 非修改性算法

- for_each
  - 定义：

```
template<typename InputIterator, typename Function>
Function for_each(InputIterator first, InputIterator last,
Function f) {
    for (; first != last; ++first)
        f(*first);
    return f;
}
```

算法与函数对象 - 非修改性算法

- find

```
template<typename InputIterator, typename T>
inline InputIterator find(InputIterator first,
        InputIterator last, const T& val);

template<typename InputIterator, typename Predicate>
inline InputIterator find_if(InputIterator first,
        InputIterator last, Predicate pred);

template<typename InputIterator, typename ForwardIterator>
InputIterator find_first_of(InputIterator first1,
        InputIterator last1, ForwardIterator first2,
        ForwardIterator last2);

template<typename InputIterator, typename ForwardIterator,
        typename BinaryPredicate>
InputIterator find_first_of(InputIterator first1,
        InputIterator last1, ForwardIterator first2,
        ForwardIterator last2, BinaryPredicate comp);
```

算法与函数对象 - 非修改性算法

- adjacent_find

```
template<typename ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first,
        ForwardIterator last);

template<typename ForwardIterator, typename BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first,
        ForwardIterator last, BinaryPredicate binary_pred);
```

```
int a[] = { 12, 23, 2, 2, 89, 7, 6 };
int b[] = { 56, 7, 23 };

int* p0 = find_if(a, a + 7, bind1st(less<int> (), 56));
assert(p0 == a + 4);
int* p1 = find_first_of(a, a + 7, b, b + 3); // &a[1]
assert(p1 == a + 1);
int* p2 = adjacent_find(a, a + 7); // &a[2]
assert(p2 == a + 2);
```

算法与函数对象 - 非修改性算法

- count

```
template<typename InputIterator, typename _Tp>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const _Tp&
value);

template<typename InputIterator, typename Predicate>
typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate
pred);
```

算法与函数对象 - 非修改性算法

- equal 和 mismatch

```
template<typename IIter1, typename IIter2>
bool equal(IIter1 first, IIter1 last, IIter2 first2);

template<typename IIter1, typename IIter2, typename
BinaryPredicate>
bool equal(IIter1 first, IIter1 last, IIter2 first2,
BinaryPredicate pred);

template<typename IIter1, typename IIter2>
pair<IIter1, IIter2> mismatch(IIter1 first, IIter1 last,
IIter2 first2);

template<typename IIter1, typename IIter2, typename
BinaryPredicate>
pair<IIter1, IIter2> mismatch(IIter1 first, IIter1 last,
IIter2 first2, BinaryPredicate pred);
```

- search

```
// 在第一个序列中查找第二个序列是否存在
template<typename ForwardIterator1, typename ForwardIterator2>
ForwardIterator1
search(ForwardIterator1 first1, ForwardIterator1 last1,
       ForwardIterator2 first2, ForwardIterator2 last2);

template<typename ForwardIterator1, typename ForwardIterator2,
       typename BinaryPredicate>
ForwardIterator1
search(ForwardIterator1 first1, ForwardIterator1 last1,
       ForwardIterator2 first2, ForwardIterator2 last2,
       BinaryPredicate predicate);

// 在一个序列中查找连续count次出现的val，返回指向第一个匹配的元素的迭代器（位置）
template<typename ForwardIterator, typename Integer,
       typename T>
ForwardIterator
search_n(ForwardIterator first, ForwardIterator last, Integer
count, const T& val);
```

- search（续）

```
template<typename ForwardIterator, typename Integer,
        typename T, typename BinaryPredicate>
ForwardIterator
search_n(ForwardIterator first, ForwardIterator last, Integer
count, const T& val, BinaryPredicate binary_pred);

// search函数的反向版本，在一个序列中查找是否包含第二个序列，如果找到，返回最后一个匹配的位置
template<typename ForwardIterator1, typename ForwardIterator2>
inline ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);

template<typename ForwardIterator1, typename ForwardIterator2,
        typename BinaryPredicate>
inline ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate comp);
```

算法与函数对象 - 修改性算法

- copy

```
template<typename IIter, typename OIter>
OIter copy(IIter first, IIter last, OIter res);

template<typename BIter1, typename BIter2>
BIter2 copy_backward(BIter1 first, BIter1 last, BIter2);

// 注意：该函数不在C++98标准中，C++0x中已包含此函数
template<typename IIter, typename OIter, typename Predicate>
OIter copy_if(IIter first, IIter last, OIter res, Predicate
pred);
```

算法与函数对象 - 修改性算法

- copy（示例）

```
int a[] = { 12, 23, 2, 2, 89, 7, 6 };

list<int> ls(a, a + 7); // size == 7
vector<int> v(7); // size == 7

// 注意，下面的操作假定v的size至少与ls一样大，否则危险
copy(ls.begin(), ls.end(), v.begin());
vector<int> v1; // size == 0
copy(ls.begin(), ls.end(), v1.begin()); // 危险！runtime crash!!
copy(ls.begin(), ls.end(), back_inserter(v1)); // OK

// copy到输出流
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
```

算法与函数对象 - 修改性算法

- transform
  - transform 并不改变输入序列，只是将操作的结果输出到输出序列

```
template<typename InputIterator, typename OutputIterator,
        typename _UnaryOperation>
OutputIterator
transform(InputIterator first, InputIterator last,
        OutputIterator result, _UnaryOperation unary_op);

template<typename InputIterator1, typename InputIterator2,
        typename OutputIterator, typename BinaryOperation>
OutputIterator
transform(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, OutputIterator result,
        BinaryOperation binary_op) {
    while(first1 != last1)
        *result++ = binary_op(*first1++, *first2++);
    return result;
}
```

算法与函数对象 - 修改性算法

- unique
  - 该族函数操作已序的序列，消除连续重复的元素
  - 其不会更改容器的 size
  - 返回不重复的序列的末端

```
template<typename ForwardIterator>
ForwardIterator
unique(ForwardIterator first, ForwardIterator last);

template<typename ForwardIterator, typename BinaryPredicate>
ForwardIterator
unique(ForwardIterator first, ForwardIterator last,
       BinaryPredicate binary_pred);

template<typename InputIterator, typename OutputIterator,
       typename BinaryPredicate>
OutputIterator
unique_copy(InputIterator first, InputIterator last,
       OutputIterator result, BinaryPredicate binary_pred);
```

算法与函数对象 - 修改性算法

- unique（续）
  - 示例：

```
int a[] = { 23, 2, 2, 89, 7, 2, 7, 6 };

vector<int> v(a, a + 8);
// step 1: sort
sort(v.begin(), v.end());
// step 2: unique
vector<int>::iterator last = unique(v.begin(), v.end());
// step 3: erase
v.erase(last, v.end()); // 真正意义上的删除

// output: 2 6 7 23 89
copy(v.begin(), v.end(), ostream_iterator<int> (cout, " "));
```

算法与函数对象 - 修改性算法

- replace

```
template<typename FIter, typename T>
void replace(FIter, FIter, const T&, const T&);

template<typename FIter, typename Predicate, typename T>
void replace_if(FIter, FIter, Predicate, const T&);

template<typename IIter, typename OIter, typename T>
OIter replace_copy(IIter, IIter, OIter, const T&, const T&);

template<typename Iter, typename OIter, typename Predicate,
typename T>
OIter replace_copy_if(Iter, Iter, OIter, Predicate, const T&);
```

算法与函数对象 - 修改性算法

- replace（示例）

```
int a[] = { 23, 2, 2, 89, 7, 2, 7, 6 };
vector<int> v(a, a + 8);
replace_if(v.begin(), v.end(), bind2nd(less<int> (), 8), 16);

// output: 23 16 16 89 16 16 16 16
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));

string dir("c:\\a\\b\\c\\d");
replace(dir.begin(), dir.end(), '\\', '/');
cout << dir << endl;    // c:/a/b/c/d
```

算法与函数对象 - 修改性算法

- remove
  - 注意： remove 不会改变输入序列的 size

```
template<typename FIter, typename _Tp>
FIter remove(FIter, FIter, const _Tp&);

template<typename FIter, typename Predicate>
FIter remove_if(FIter, FIter, Predicate);

template<typename IIter, typename OIter, typename _Tp>
OIter remove_copy(IIter, IIter, OIter, const _Tp&);

template<typename IIter, typename OIter, typename Predicate>
OIter remove_copy_if(IIter, IIter, OIter, Predicate);
```

算法与函数对象 - 修改性算法

- remove（示例）

```
int a[] = { 23, 2, 2, 89, 7, 2, 7, 6 };
vector<int> v(a, a + 8);
vector<int>::iterator last = remove(v.begin(), v.end(), 2);

// output: 23 89 7 7 6 2 7 6
copy(v.begin(), v.end(), ostream_iterator<int> (cout, " "));
cout << endl;

v.erase(last, v.end());
// output: 23 89 7 7 6
copy(v.begin(), v.end(), ostream_iterator<int> (cout, " "));
cout << endl;
```

算法与函数对象 - 修改性算法

- fill 和 generate

```
template<typename FIter, typename T>
void fill(FIter, FIter, const T&);

template<typename OIter, typename Size, typename T>
void fill_n(OIter, Size, const T&);

template<typename FIter, typename Generator>
void generate(FIter, FIter, Generator);

template<typename OIter, typename Size, typename T,
         typename Generator>
void generate_n(OIter, Size, Generator);
```

算法与函数对象 - 修改性算法

- fill 和 generate（示例）

```
struct Generator {
    Generator(int i) : n(i) {
    }
    int operator()() {
        return int((double(rand()) / RAND_MAX) * n);
    }
private:
    int n;
};

int a1[16];
int a2[16];
vector<int> v;

fill(a1, a1 + 16, 88);

generate(a2, a2 + 16, Generator(100));

fill_n(back_inserter(v), 20, 88);
```

算法与函数对象 - 修改性算法

- reverse 和 rotate

```
template<typename BIter>
void reverse(BIter, BIter);

template<typename BIter, typename OIter>
OIter reverse_copy(BIter, BIter, OIter);

template<typename FIter>
void rotate(FIter, FIter, FIter);

template<typename FIter, typename OIter>
OIter rotate_copy(FIter, FIter, FIter, OIter);

template<typename RAIter>
void random_shuffle(RAIter, RAIter);

template<typename RAIter, typename Generator>
void random_shuffle(RAIter, RAIter, Generator&);
```

算法与函数对象 - 修改性算法

- reverse 和 rotate（示例）

```
int a1[] = { 66, 88, 99, 12, 28, 11 };

reverse(a1, a1 + 6);
// 11 28 12 99 88 66
copy(a1, a1 + 6, ostream_iterator<int> (cout, " "));

rotate(a1, a1 + 2, a1 + 4);
// 12 99 11 28 88 66
copy(a1, a1 + 6, ostream_iterator<int> (cout, " "));

random_shuffle(a1, a1 + 6);
// 88 28 99 11 12 66 (也有可能是其它的次序)
copy(a1, a1 + 6, ostream_iterator<int> (cout, " "));
```

算法与函数对象 - 修改性算法

- swap

```
template<typename T>
void swap(T&, T&);

template<typename T, size_t N>
void swap(T(&)[N], T(&)[N]);

template<typename FIter1, typename FIter2>
void iter_swap(FIter1, FIter2);

template<typename FIter1, typename FIter2>
FIter2 swap_ranges(FIter1, FIter1, FIter2);
```

算法与函数对象 - 排序算法

- sort
  - sort 需要随机迭代器，也即需要 array 、 vector 类似的容器
  - sort 的效率平均为 O(N*log(N))，但最差性能却是 O(N*N)
  - stable_sort 则总可以保证 O(N*log(N)*log(N))，而且在提供额外存储时，性能可以接近 O(N*log(N))
  - sort 不能保证元素的相对位置，而 stable_sort 可以

算法与函数对象 - 排序算法

- sort（续）

```
template<typename RAIter>
void sort(RAIter, RAIter);

template<typename RAIter, typename Compare>
void sort(RAIter, RAIter, Compare);

template<typename RAIter>
void stable_sort(RAIter, RAIter);

template<typename RAIter, typename Compare>
void stable_sort(RAIter, RAIter, Compare);
```

算法与函数对象 - 排序算法

- partial_sort
  - 对局部进行排序
  - 与 sort 类似， partial_sort 需要随机迭代器

```
template<typename RAIter>
void partial_sort(RAIter, RAIter, RAIter);

template<typename RAIter, typename Compare>
void partial_sort(RAIter, RAIter, RAIter, Compare);

template<typename IIter, typename RAIter>
RAIter partial_sort_copy(IIter, IIter, RAIter, RAIter);

template<typename IIter, typename RAIter, typename Compare>
RAIter partial_sort_copy(IIter, IIter, RAIter, RAIter,
Compare);
```

算法与函数对象 - 排序算法

- nth_element
  - 将 N 个元素放到其正确的位置，且保证序列中比第 N 个元素小的元素不会出现在它之后

```
template<typename RAIter>
void nth_element(RAIter first, RAIter nth, RAIter last);

template<typename RAIter, typename Compare>
void nth_element(RAIter first, RAIter nth, RAIter last,
Compare cmp);
```

```
int a1[] = { 66, 88, 99, 12, 28, 11, 2, 9 };
// 取序列中3个最小的元素放到最前
nth_element(a1+0, a1 + 3, a1 + 8);

// output: 9 2 11 12 28 99 88 66
copy(a1, a1 + 8, ostream_iterator<int> (cout, " "));
```

- 对已序区间的搜索
  - 注意：是对已序序列操作
  - binary_search 只是判断某个给定的值是否出现在序列中
  - equal_range 可以返回序列中出现给定值出现的位置

```
template<typename FIter, typename T>
bool binary_search(FIter, FIter, const T&);

template<typename FIter, typename T, typename Compare>
bool binary_search(FIter, FIter, const T&, Compare);

template<typename FIter, typename T>
pair<FIter, FIter>
equal_range(FIter, FIter, const T&);

template<typename FIter, typename T, typename Compare>
pair<FIter, FIter> equal_range(FIter, FIter, const T&,
Compare);
```

- 对已序区间的搜索（续）
  - lower_bound 返回给定值出现在序列中第一次出现的位置
  - upper_bound 返回给定值最后出现在序列中的位置之后的位置

```
template<typename FIter, typename T>
FIter lower_bound(FIter, FIter, const T&);

template<typename FIter, typename T, typename Compare>
FIter lower_bound(FIter, FIter, const T&, Compare);

template<typename FIter, typename T>
FIter upper_bound(FIter, FIter, const T&);

template<typename FIter, typename T, typename Compare>
FIter upper_bound(FIter, FIter, const T&, Compare);
```

- merge
  - merge 合并两个已序序列，产生的新序列也是有序的
  - inplace_merge 合并同一个序列中的两个已序子序列

```
template<typename IIter1, typename IIter2, typename OIter>
OIter merge(IIter1, IIter1, IIter2, IIter2, OIter);

template<typename IIter1, typename IIter2,
         typename OIter, typename Compare>
OIter merge(IIter1, IIter1, IIter2, IIter2, OIter, Compare);

template<typename BIter>
void inplace_merge(BIter, BIter, BIter);

template<typename BIter, typename Compare>
void inplace_merge(BIter, BIter, BIter, Compare);
```

算法与函数对象 - 排序算法

- merge（示例）

```
int a1[] = { 1, 3, 5, 7 };
int a2[] = { 2, 4, 6, 8 };
int a3[8];

merge(a1, a1 + 4, a2, a2 + 4, a3);
// output: 1 2 3 4 5 6 7 8
copy(a3, a3 + 8, ostream_iterator<int> (cout, " "));

int a4[] = { 1, 3, 5, 7, 2, 4, 6, 8 };
inplace_merge(a4, a4 + 4, a4 + 8);
// output: 1 2 3 4 5 6 7 8
copy(a4, a4 + 8, ostream_iterator<int> (cout, " "));
```

- partition
  - 将符合某种条件的元素放到不符合的元素前面

```
template<typename BIter, typename Predicate>
BIter partition(BIter, BIter, Predicate);

template<typename BIter, typename Predicate>
BIter stable_partition(BIter, BIter, Predicate);
```

```
int a1[] = { 21, 13, 65, 37, 56, 23 };
partition(a1, a1 + 6, bind2nd(less<int> (), 30));
// output: 21 13 23 37 56 65  (不保证元素的相对次序)
copy(a1, a1 + 6, ostream_iterator<int> (cout, " "));

int a2[] = { 21, 13, 65, 37, 56, 23 };
stable_partition(a2, a2 + 6, bind2nd(less<int> (), 30));
// output: 21 13 23 37 56 65  (保证元素的相对次序)
copy(a2, a2 + 6, ostream_iterator<int> (cout, " "));
```

- 序列中的集合运算
  - 最好是已序序列，否则效率将很低

```
template<typename IIter1, typename IIter2>
bool includes(IIter1, IIter1, IIter2, IIter2);

template<typename IIter1, typename IIter2,
         typename Compare>
bool includes(IIter1, IIter1, IIter2, IIter2, Compare);

template<typename IIter1, typename IIter2,
         typename OIter>
OIter set_difference(IIter1, IIter1, IIter2, IIter2, OIter);

template<typename IIter1, typename IIter2,
         typename OIter, typename Compare>
OIter set_difference(IIter1, IIter1, IIter2,
                     IIter2, OIter, Compare);
```

- 序列中的集合运算（续）

```
template<typename IIter1, typename IIter2,
        typename OIter>
OIter set_intersection(IIter1, IIter1, IIter2, IIter2,
OIter);

template<typename IIter1, typename IIter2,
        typename OIter, typename Compare>
OIter set_intersection(IIter1, IIter1, IIter2,
        IIter2, OIter, Compare);

template<typename IIter1, typename IIter2, typename OIter>
OIter set_symmetric_difference(IIter1, IIter1,
        IIter2, IIter2, OIter);

template<typename IIter1, typename IIter2,
        typename OIter, typename Compare>
OIter set_symmetric_difference(IIter1, IIter1, IIter2,
        IIter2, OIter, Compare);
```

- 序列中的集合运算（续）

```
template<typename IIter1, typename IIter2, typename OIter>
OIter set_union(IIter1, IIter1, IIter2, IIter2, OIter);

template<typename IIter1, typename IIter2,
         typename OIter, typename Compare>
OIter set_union(IIter1, IIter1, IIter2, IIter2, OIter,
Compare);
```

算法与函数对象 - 堆算法

- heap
  - 堆算法需要随机迭代器

```
template<typename RAIter>
void push_heap(RAIter, RAIter);

template<typename RAIter, typename Compare>
void push_heap(RAIter, RAIter, Compare);

template<typename RAIter>
void pop_heap(RAIter, RAIter);

template<typename RAIter, typename Compare>
void pop_heap(RAIter, RAIter, Compare);
```

算法与函数对象 - 堆算法

- heap（续）

```
template<typename RAIter>
void make_heap(RAIter, RAIter);

template<typename RAIter, typename Compare>
void make_heap(RAIter, RAIter, Compare);

template<typename RAIter>
void sort_heap(RAIter, RAIter);

template<typename RAIter, typename Compare>
void sort_heap(RAIter, RAIter, Compare);
```

算法与函数对象 - 最小和最大

- max 和 min

```
template<typename T>
const T& max(const T&, const T&);

template<typename T, typename Compare>
const T& max(const T&, const T&, Compare);

template<typename T>
const T& min(const T&, const T&);

template<typename T, typename Compare>
const T& min(const T&, const T&, Compare);
```

算法与函数对象 - 最小和最大

- max_element 和 min_element

```
template<typename FIter>
FIter max_element(FIter, FIter);

template<typename FIter, typename Compare>
FIter max_element(FIter, FIter, Compare);

template<typename FIter>
FIter min_element(FIter, FIter);

template<typename FIter, typename Compare>
FIter min_element(FIter, FIter, Compare);
```

算法与函数对象 - 排列算法

- next_permutation 和 prev_permutation

```
template<typename BIter>
bool next_permutation(BIter, BIter);

template<typename BIter, typename Compare>
bool next_permutation(BIter, BIter, Compare);

template<typename BIter>
bool prev_permutation(BIter, BIter);

template<typename BIter, typename Compare>
bool prev_permutation(BIter, BIter, Compare);
```

```
char ss[] = "abcd";
cout << ss << endl;
while (next_permutation(ss, ss + 3)) // 4 * 3 * 2 个排列组合
    cout << ss << endl;
```

算法与函数对象 - C 风格的算法

- qsort 和 bsearch
  - 包含头文件 <cstdlib>

```
typedef int (*cmp)(const void*, const void*);

void qsort(void* base, size_t nmemb, size_t size, cmp c);

void* bsearch(const void* key, const void* base,
        size_t nmemb, size_t size, cmp c);
```

算法与函数对象 - Bjarne's Advices

- Bjarne's Advices
  - 多用算法，少用循环
  - 在写循环时，考虑是否能将其描述为一个通用算法
  - 保证一对迭代器参数确实是表述了一个序列
  - 把测试表达式表述成能够作为判断式 (Predicate) 使用的形式
  - 记住判断式是函数或对象，不是类型
  - 利用 mem_fun() 和 mem_fun_ref() 将算法应用于容器
  - 利用 ptr_fun() 将函数作为算法的参数
  - 切记 strcmp() 用 0 表示相等，与 == 不同
  - 利用判断式，以便能以各种比较准则和相等准则使用算法

算法与函数对象 - Bjarne's Advices

- Bjarne's Advices （续）
  - 利用判断式和其它函数对象，使标准算法能用于表示范围广泛的意义
  - 操作符 == 和 < 在指针上的默认意义很少适用于标准算法
  - 算法并不会直接为其参数序列增加或减少元素
  - 应保证用于同一个序列的小于和相等判断式相互匹配
  - 有时排好序的序列用起来更有效且优雅