11.8. Mutating Algorithms

Mutating algorithms change the order of elements but not their values. Because elements of associative and unordered containers have an order defined by the container, you can't use these algorithms as a destination for mutating algorithms.

11.8.1. Reversing the Order of Elements

Click here to view code image

```
void
reverse (BidirectionalIterator beg, BidirectionalIterator end)

OutputIterator
reverse_copy (BidirectionalIterator sourceBeg, BidirectionalIterator
sourceEnd,
              OutputIterator destBeg)
```

- `reverse()` reverses the order of the elements inside the range [*beg* , *end*) .
- `reverse_copy()` reverses the order of the elements while copying them from the source range [*sourceBeg,sourceEnd*) to the destination range starting with *destBeg*.
- `reverse_copy()` returns the position after the last copied element in the destination range (the first element that is not overwritten).
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- Lists provide an equivalent member function, `reverse()` , which offers better performance because it relinks pointers instead of assigning element values (see Section 8.8.1, page 423).
- Complexity: linear (*numElems* /2 swaps or *numElems* assignments, respectively).

The following program demonstrates how to use `reverse()` and `reverse_copy()` :

Click here to view code image

```
// algo/reverse1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    // reverse order of elements
    reverse (coll.begin(), coll.end());
    PRINT_ELEMENTS(coll,"coll: ");

    // reverse order from second to last element but one
    reverse (coll.begin()+1, coll.end()-1);
    PRINT_ELEMENTS(coll,"coll: ");

    // print all of them in reverse order
    reverse_copy (coll.cbegin(), coll.cend(),          // source
                  ostream_iterator<int>(cout," "));  // destination
    cout << endl;
}
```

The program has the following output:

```
coll: 1 2 3 4 5 6 7 8 9
coll: 9 8 7 6 5 4 3 2 1
coll: 9 2 3 4 5 6 7 8 1
1 8 7 6 5 4 3 2 9
```

11.8.2. Rotating Elements

Rotating Elements inside a Sequence

```
ForwardIterator
rotate (ForwardIterator beg, ForwardIterator newBeg, ForwardIterator end)
```

- Rotates elements in the range `[` *beg* `,` *end* `)` so that `*newBeg` is the new first element after the call.
- Since C++11, returns *beg* `+(` *end* `-` *newbeg* `)` , which is the new position of the first element. Before C++11, the return type was `void` .
- The caller must ensure that *newBeg* is a valid position in the range `[` *beg* `,` *end* `)` ; otherwise, the call results in undefined behavior.
- Complexity: linear (at most, *numElems* swaps).

The following program demonstrates how to use `rotate()` :

Click here to view code image

```
// algo/rotate1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll:      ");

    // rotate one element to the left
    rotate (coll.begin(),         // beginning of range
            coll.begin() + 1,     // new first element
            coll.end());          // end of range
    PRINT_ELEMENTS(coll,"one left:  ");

    // rotate two elements to the right
    rotate (coll.begin(),         // beginning of range
            coll.end() - 2,       // new first element
            coll.end());          // end of range
    PRINT_ELEMENTS(coll,"two right: ");

    // rotate so that element with value 4 is the beginning
    rotate (coll.begin(),                      // beginning of range
            find(coll.begin(),coll.end(),4),   // new first element
            coll.end());                       // end of range
    PRINT_ELEMENTS(coll,"4 first:   ");
}
```

As the example shows, you can rotate to the left with a positive offset for the beginning and rotate to the right with a negative offset to the end. However, adding the offset to the iterator is possible only when you have random-access iterators, as you have for vectors. Without such iterators, you must use `advance()` (see the example of `rotate_copy()` on page 586).

The program has the following output:

```
coll:      1 2 3 4 5 6 7 8 9
one left:  2 3 4 5 6 7 8 9 1
two right: 9 1 2 3 4 5 6 7 8
4 first:   4 5 6 7 8 9 1 2 3
```

Rotating Elements While Copying

Click here to view code image

```
OutputIterator
rotate_copy (ForwardIterator sourceBeg, ForwardIterator newBeg,
             ForwardIterator sourceEnd,
             OutputIterator destBeg)
```

- Is a combination of `copy()` and `rotate()` .
- Copies the elements of the source range `[` *sourceBeg,sourceEnd* `)` into the destination range starting with *destBeg* in rotated order so that `*` *newBeg* is the new first element.
- Returns *destBeg* `+(` *sourceEnd* `-` *sourceBeg* `)` , which is the position after the last copied element in the destination range.

- The caller must ensure that *newBeg* is an element in the range `[` *beg* `,` *end* `)` ; otherwise, the call results in undefined behavior.
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- The source and destination ranges should not overlap.
- Complexity: linear (*numElems* assignments).

The following program demonstrates how to use `rotate_copy()` :

Click here to view code image

```
// algo/rotate2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    set<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll);

    // print elements rotated one element to the left
    set<int>::const_iterator pos = next(coll.cbegin());
    rotate_copy(coll.cbegin(),                         // beginning of source
                pos,                                   // new first element
                coll.cend(),                           // end of source
                ostream_iterator<int>(cout," "));      // destination
    cout << endl;

    // print elements rotated two elements to the right
    pos = coll.cend();
    advance(pos,-2);
    rotate_copy(coll.cbegin(),                         // beginning of source
                pos,                                   // new first element
                coll.cend(),                           // end of source
                ostream_iterator<int>(cout," "));      // destination
    cout << endl;

    // print elements rotated so that element with value 4 is the beginning
    rotate_copy(coll.cbegin(),                         // beginning of source
                coll.find(4),                          // new first element
                coll.cend(),                           // end of source
                ostream_iterator<int>(cout," "));      // destination
    cout << endl;
}
```

Unlike the previous example of `rotate()` (see Section 11.8.2, page 584), here a set is used instead of a vector. This has two consequences:

1. You must use `advance()` (see Section 9.3.1, page 441) or `next()` (see Section 9.3.2, page 443) to change the value of the iterator, because bidirectional iterators do not provide operator `+` .
2. You should use the `find()` member function instead of the `find()` algorithm, because the former has better performance.

The program has the following output:

```
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 1
8 9 1 2 3 4 5 6 7
4 5 6 7 8 9 1 2 3
```

11.8.3. Permuting Elements

Click here to view code image

```
bool
next_permutation (BidirectionalIterator beg, BidirectionalIterator end)
bool
next_permutation (BidirectionalIterator beg, BidirectionalIterator end,
                  BinaryPredicate op)
bool
prev_permutation (BidirectionalIterator beg, BidirectionalIterator end)
bool
prev_permutation (BidirectionalIterator beg, BidirectionalIterator end,
                    BinaryPredicate op)
```

- `next_permutation()` changes the order of the elements in [*beg* , *end*) according to the next permutation.
- `prev_permutation()` changes the order of the elements in [*beg* , *end*) according to the previous permutation.
- The first forms compare the elements by using operator `<` .
- The second forms compare the elements by using the binary predicate

  ```
  op ( elem1 , elem2 )
  ```

  which should return `true` if *elem1* is "less than" *elem2*.
- Both algorithms return `false` if the elements got the "normal" (lexicographical) order: that is, ascending order for `next_permutation()` and descending order for `prev_permutation()` . So, to run through all permutations, you have to sort all elements (ascending or descending), and start a loop that calls `next_permutation()` or `prev_permutation()` as long as these algorithms return `true` .[3] See Section 11.5.4, page 548, for an explanation of lexicographical sorting.

[3] `next_permutation()` and `prev_permutation()` could also be used to sort elements in a range. You just call them for a range as long as they return `true` . However, doing so would produce really bad performance.

- Complexity: linear (at most, *numElems* /2 swaps).

The following example demonstrates how `next_permutation()` and `prev_permutation()` run through all permutations of the elements:

Click here to view code image

```
// algo/permutation1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;
    INSERT_ELEMENTS(coll,1,3);
    PRINT_ELEMENTS(coll,"on entry:   ");

    // permute elements until they are sorted
    // - runs through all permutations because the elements are sorted now
    while (next_permutation(coll.begin(),coll.end())) {
        PRINT_ELEMENTS(coll," ");
    }
    PRINT_ELEMENTS(coll,"afterward:  ");

    // permute until descending sorted
    // - this is the next permutation after ascending sorting
    // - so the loop ends immediately
    while (prev_permutation(coll.begin(),coll.end())) {
        PRINT_ELEMENTS(coll," ");
    }
    PRINT_ELEMENTS(coll,"now:        ");

    // permute elements until they are sorted in descending order
    // - runs through all permutations because the elements are sorted in descending
order now
    while (prev_permutation(coll.begin(),coll.end())) {
        PRINT_ELEMENTS(coll," ");
    }
    PRINT_ELEMENTS(coll,"afterward:  ");
}
```

The program has the following output:

```
on entry:  1 2 3
 1 3 2
 2 1 3
 2 3 1
 3 1 2
 3 2 1
```

```
afterward: 1 2 3
now:       3 2 1
 3 1 2
 2 3 1
 2 1 3
 1 3 2
 1 2 3
afterward: 3 2 1
```

11.8.4. Shuffling Elements

Shuffling Using the Random-Number Library

Click here to view code image

```
void
shuffle (RandomAccessIterator beg, RandomAccessIterator end,
         UniformRandomNumberGenerator&& eng)

void
random_shuffle (RandomAccessIterator beg, RandomAccessIterator end)

void
random_shuffle (RandomAccessIterator beg, RandomAccessIterator end,
                RandomFunc&& op)
```

- The first form, available since C++11, shuffles the order of the elements in the range [*beg*, *end*), using an *engine eng* as introduced by the random numbers and distributions library (see Section 17.1.2, page 912).
- The second form shuffles the order of the elements in the range [*beg*, *end*), using an implementation-defined uniform distribution random-number generator, such as the C function `rand()`.
- The third form shuffles the order of the elements in the range [*beg*, *end*), using *op*. *op* is called with an integral value of `difference_type` of the iterator:

  op (*max*)

  which should return a random number greater than or equal to zero and less than *max*. Thus, it should not return *max* itself.
- For `shuffle()`, you should not pass an engine just temporarily created. See Section 17.1.1, page 911, for details.
- Before C++11, *op* was declared as `RandomFunc&`, so you couldn't pass a temporary value or an ordinary function.
- Complexity: linear (*numElems* `-1` swaps).

Note that old global C functions, such as `rand()`, store their local states in a static variable. However, this has some disadvantages: For example, the random-number generator is inherently thread unsafe, and you can't have two independent streams of random numbers. Therefore, function objects provide a better solution by encapsulating their local states as one or more member variables. For this reason, the algorithms change the state of the passed generator while generating a new random number.

The following example demonstrates how to shuffle elements by calling `random_shuffle()` without passing a random-number generator or by using `shuffle()`:

Click here to view code image

```
// algo/shuffle1.cpp

#include <cstdlib>
#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll:     ");

    // shuffle all elements randomly
    random_shuffle (coll.begin(), coll.end());

    PRINT_ELEMENTS(coll,"shuffled: ");

    // sort them again
    sort (coll.begin(), coll.end());
    PRINT_ELEMENTS(coll,"sorted:   ");

    // shuffle elements with default engine
```

```
        default_random_engine dre;
        shuffle (coll.begin(), coll.end(),   // range
                 dre);                        // random-number generator

        PRINT_ELEMENTS(coll,"shuffled: ");
    }
```

A possible (but not portable) output of the program is as follows:

```
    coll:     1 2 3 4 5 6 7 8 9
    shuffled: 8 2 4 9 5 7 3 6 1
    sorted:   1 2 3 4 5 6 7 8 9
    shuffled: 8 7 5 6 2 4 9 3 1
```

See Section 17.1, page 907, for details about engines you can pass to `shuffle()` .

The following example demonstrates how to shuffle elements by using your own random-number generator passed to `random_shuffle()` :

Click here to view code image

```
    // algo/randomshuffle1.cpp

    #include <cstdlib>
    #include "algostuff.hpp"
    using namespace std;

    class MyRandom {
      public:
        ptrdiff_t operator() (ptrdiff_t max) {
            double tmp;
            tmp = static_cast<double>(rand())
                    / static_cast<double>(RAND_MAX);
            return static_cast<ptrdiff_t>(tmp * max);
        }
    };

    int main()
    {
        vector<int> coll;

        INSERT_ELEMENTS(coll,1,9);
        PRINT_ELEMENTS(coll,"coll:     ");

        // shuffle elements with self-written random-number generator
        MyRandom rd;
        random_shuffle (coll.begin(), coll.end(),    // range
                        rd);                         // random-number generator

        PRINT_ELEMENTS(coll,"shuffled: ");
    }
```

The call of `random()` uses the self-written random-number generator `rd()` , an object of the auxiliary function object class `MyRandom` , which uses a random-number algorithm that often is better than the usual direct call of `rand()` .[4] Note that before C++11, you couldn't pass a temporary object as random-number generator:

[4] The way `MyRandom` generates random numbers is introduced and described in [Stroustrup:C++].

Click here to view code image

```
    random_shuffle (coll.begin(), coll.end(),
                    MyRandom());                  // ERROR before C++11
```

Again, a possible but not portable output of the program is as follows:

```
    coll:     1 2 3 4 5 6 7 8 9
    shuffled: 1 8 6 2 4 9 3 7 5
```

See Section 17.1.1, page 912, for some general comments about the use of `rand()` .

11.8.5. Moving Elements to the Front

Click here to view code image

```
    ForwardIterator
    partition (ForwardIterator beg, ForwardIterator end,
               UnaryPredicate op)
```

```
BidirectionalIterator
stable_partition (BidirectionalIterator beg, BidirectionalIterator end,
                  UnaryPredicate op)
```

- Both algorithms move all elements in the range [*beg*, *end*) to the front, for which the unary predicate

  op(*elem*)

  yields `true`.
- Both algorithms return the first position for which *op*() yields `false`.
- The difference between `partition()` and `stable_partition()` is that the algorithm `stable_partition()` preserves the relative order of elements that match the criterion and those that do not.
- You could use this algorithm to split elements into two parts according to a sorting criterion. The `nth_element()` algorithm has a similar ability. See Section 11.2.2, page 514, for a discussion of the differences between these algorithms and `nth_element()`.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- Before C++11, `partition()` required bidirectional iterators instead of forward iterators and guaranteed at most *numElems* /2 swaps.
- Use `partition_copy()` (see Section 11.8.6, page 594) to copy elements into one destination range for fulfilling and one for not fulfilling a predicate (available since C++11).
- Complexity:
  - For `partition()`: linear (at most *numElems* /2 swaps and *numElems* calls of *op*() if bidirectional iterators or random-access iterators are used; at most *numElems* swaps if the iterators are only forward iterators).
  - For `stable_partition()`: linear if there is enough extra memory (*numElems* swaps and calls of *op*()); otherwise, n-log-n (*numElems* calls of *op*() but *numElems* *log(*numElems*) swaps).

The following program demonstrates the use of and the difference between `partition()` and `stable_partition()`:

Click here to view code image

```
// algo/partition1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll1;
    vector<int> coll2;

    INSERT_ELEMENTS(coll1,1,9);
    INSERT_ELEMENTS(coll2,1,9);
    PRINT_ELEMENTS(coll1,"coll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");
    cout << endl;

    // move all even elements to the front
    vector<int>::iterator pos1, pos2;
    pos1 = partition(coll1.begin(), coll1.end(),          // range
                     [](int elem){                         // criterion
                         return elem%2==0;
                     });
    pos2 = stable_partition(coll2.begin(), coll2.end(),   // range
                            [](int elem){                  // criterion
                                return elem%2==0;
                            });

    // print collections and first odd element
    PRINT_ELEMENTS(coll1,"coll1: ");
    cout << "first odd element: " << *pos1 << endl;
    PRINT_ELEMENTS(coll2,"coll2: ");
    cout << "first odd element: " << *pos2 << endl;
}
```

The program has the following output:

```
coll1: 1 2 3 4 5 6 7 8 9
coll2: 1 2 3 4 5 6 7 8 9
```

```
coll1: 8 2 6 4 5 3 7 1 9
first odd element: 5
coll2: 2 4 6 8 1 3 5 7 9
first odd element: 1
```

As this example shows, `stable_partition()`, unlike `partition()`, preserves the relative order of the even and the odd elements.

11.8.6. Partition into Two Subranges

Click here to view code image

```
pair<OutputIterator1,OutputIterator2>
partition_copy (InputIterator sourceBeg, InputIterator sourceEnd,
                OutputIterator1 destTrueBeg, OutputIterator2 destFalseBeg,
                UnaryPredicate op)
```

- Splits all elements in the range [*beg* , *end*) according to the predicate *op* () into two subranges.
- All elements for which the unary predicate

  op (*elem*)

  yields `true` are copied into the range starting with *destTrueBeg*. All elements for which the predicate yields `false` are copied into the range starting with *destFalseBeg*.
- The algorithm returns a pair of the position after the last copied elements of the destination ranges (the first element that is not overwritten).
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- This algorithm is available since C++11.
- Use `copy_if()` (see Section 11.6.1, page 557) or `remove_copy_if()` (see Section 11.7.1, page 577) if you need only the elements that either fulfill or do not fulfill the predicate.
- Complexity: linear (at most *numElems* applications of *op* ()).

The following program demonstrates the use of `partition_copy()`:

```
// algo/partitioncopy1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll = { 1, 6, 33, 7, 22, 4, 11, 33, 2, 7, 0, 42, 5 };
    PRINT_ELEMENTS(coll,"coll: ");

    // destination collections:
    vector<int> evenColl;
    vector<int> oddColl;

    // copy all elements partitioned accordingly into even and odd elements
    partition_copy (coll.cbegin(),coll.cend(),   // source range
                    back_inserter(evenColl),     // destination for even elements
                    back_inserter(oddColl),      // destination for odd elements
                    [](int elem){                // predicate: check for even
elements
                        return elem%2==0;
                    });
    PRINT_ELEMENTS(evenColl,"evenColl: ");
    PRINT_ELEMENTS(oddColl, "oddColl:  ");
}
```

The program has the following output:

```
coll: 1 6 33 7 22 4 11 33 2 7 0 42 5
evenColl: 6 22 4 2 0 42
oddColl:  1 33 7 11 33 7 5
```