MODULE 36
--THE STL--
ALGORITHM PART IV

My Training Period: hours

Note: Compiled using Microsoft Visual C++ .Net, win32 empty console mode application. **g++** compilation examples given at the end of this Module.

Abilities

- Able to understand and use the member functions of the algorithm.
- Appreciate how the usage of the template classes and functions.
- Able to use containers, iterators and algorithm all together.

36.1 Continuation from previous Module…

push_heap()

- Adds an element that is at the end of a range to an existing heap consisting of the prior elements in the range.

```
template<class RandomAccessIterator>
   void push_heap(
      RandomAccessIterator _First,
      RandomAccessIterator _Last
   );
template<class RandomAccessIterator, class BinaryPredicate>
   void push_heap(
      RandomAccessIterator _First,
      RandomAccessIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

Parameter	Description
_First	A random-access iterator addressing the position of the first element in the heap.
_Last	A random-access iterator addressing the position one past the final element in the range to be converted into a heap.
_Comp	User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns true when satisfied and false when not satisfied.

Table 36.1

- The element must first be pushed back to the end of an existing heap and then the algorithm is used to add this element to the existing heap. Repeat again, heaps have two properties:

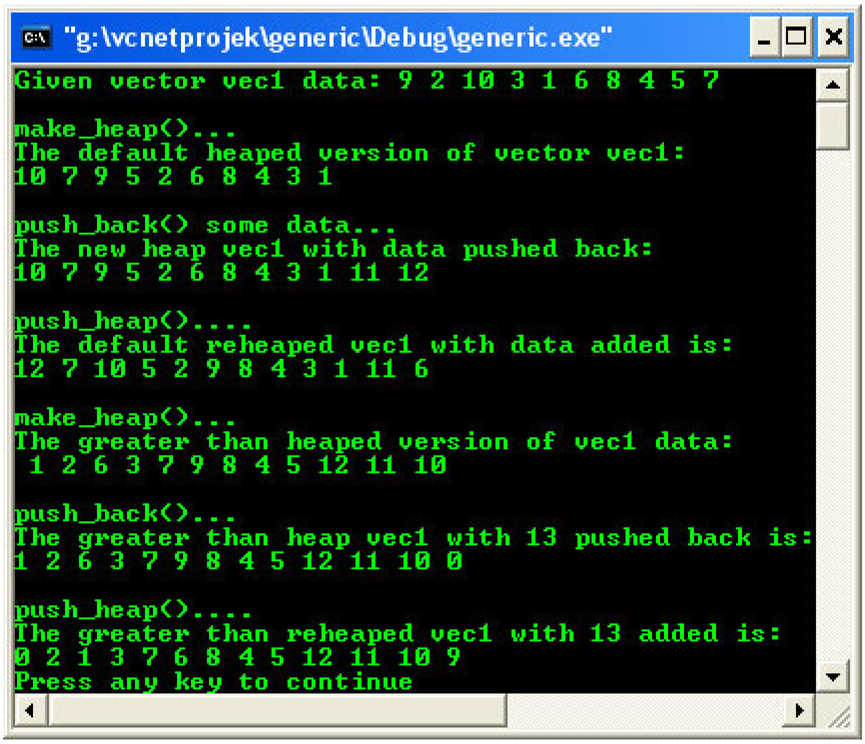
  - The first element is always the largest.
  - Elements may be added or removed in logarithmic time.

- Heaps are an ideal way to implement priority queues and they are used in the implementation of the STL container adaptor `priority_queue` Class.
- The range referenced must be valid; all pointers must be de-referenceable and within the sequence the last position is reachable from the first by incrementation.
- The range excluding the newly added element at the end must be a heap.
- The complexity is logarithmic, requiring at most log `(_Last – _First)` comparisons.

```
//algorithm, push_heap()
//make_heap()
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
using namespace std;
```

```
int main()
{
   vector <int> vec1;
   vector <int>::iterator Iter1;

   int i;
   for(i = 1; i <= 10; i++)
     vec1.push_back(i);

   random_shuffle(vec1.begin(), vec1.end());

   cout<<"Given vector vec1 data: ";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Make vec1 a heap with default less than ordering
   cout<<"\nmake_heap()..."<<endl;
   make_heap(vec1.begin(), vec1.end());
   cout<<"The default heaped version of vector vec1:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Add elements to the heap
   cout<<"\npush_back() some data...\n";
   vec1.push_back(11);
   vec1.push_back(12);
   cout<<"The new heap vec1 with data pushed back:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   cout<<"\npush_heap()...."<<endl;
   push_heap(vec1.begin(), vec1.end());
   cout<<"The default reheaped vec1 with data added is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Make vec1 a heap with greater than ordering
   cout<<"\nmake_heap()..."<<endl;
   make_heap(vec1.begin(), vec1.end(), greater<int>());
   cout<<"The greater than heaped version of vec1 data:\n ";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   cout<<"\npush_back()..."<<endl;
   vec1.push_back(0);
   cout<<"The greater than heap vec1 with 13 pushed back is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   cout<<"\npush_heap()...."<<endl;
   push_heap(vec1.begin(), vec1.end(), greater<int>());
   cout<<"The greater than reheaped vec1 with 13 added is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;
}
```

Output:



```
Given vector vec1 data: 9 2 10 3 1 6 8 4 5 7

make_heap()...
The default heaped version of vector vec1:
10 7 9 5 2 6 8 4 3 1

push_back() some data...
The new heap vec1 with data pushed back:
10 7 9 5 2 6 8 4 3 1 11 12

push_heap()....
The default reheaped vec1 with data added is:
12 7 10 5 2 9 8 4 3 1 11 6

make_heap()...
The greater than heaped version of vec1 data:
 1 2 6 3 7 9 8 4 5 12 11 10

push_back()...
The greater than heap vec1 with 13 pushed back is:
1 2 6 3 7 9 8 4 5 12 11 10 0

push_heap()....
The greater than reheaped vec1 with 13 added is:
0 2 1 3 7 6 8 4 5 12 11 10 9
Press any key to continue
```

random_shuffle()

- Rearranges a sequence of *N* elements in a range into one of $N!$ possible arrangements selected at random.

```
template<class RandomAccessIterator>
   void random_shuffle(
      RandomAccessIterator _First,
      RandomAccessIterator _Last
   );
template<class RandomAccessIterator, class RandomNumberGenerator>
   void random_shuffle(
      RandomAccessIterator _First,
      RandomAccessIterator _Last,
      RandomNumberGenerator& _Rand
   );
```

Parameters

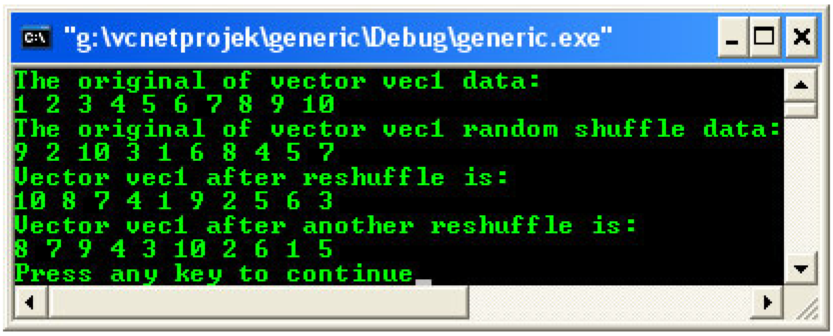
Parameter	Description
_First	A random-access iterator addressing the position of the first element in the range to be rearranged.
_Last	A random-access iterator addressing the position one past the final element in the range to be rearranged.
_Rand	A special function object called a random number generator.

Table 36.2

- The two versions (refer to the templates) of the function differ in how they generate random numbers.
- The first version uses an internal random number generator and the second a random number generator function object that is explicitly passed and can accept a seed value.

```
//algorithm, random_shuffle()
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
using namespace std;

int main()
{
   vector <int> vec1;
   vector <int>::iterator Iter1;

   int i;
   for(i = 1; i <= 10; i++)
     vec1.push_back(i);
   cout<<"The original of vector vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;

   //random shuffle…
   random_shuffle(vec1.begin(), vec1.end());
   cout<<"The original of vector vec1 random shuffle data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;

   //Shuffled once
   random_shuffle(vec1.begin(), vec1.end());
   push_heap(vec1.begin(), vec1.end());
   cout<<"Vector vec1 after reshuffle is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;

   //Shuffled again
   random_shuffle(vec1.begin(), vec1.end());
   push_heap(vec1.begin(), vec1.end());
   cout<<"Vector vec1 after another reshuffle is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;
}
```

Output:



```
The original of vector vec1 data:
1 2 3 4 5 6 7 8 9 10
The original of vector vec1 random shuffle data:
9 2 10 3 1 6 8 4 5 7
Vector vec1 after reshuffle is:
10 8 7 4 1 9 2 5 6 3
Vector vec1 after another reshuffle is:
8 7 9 4 3 10 2 6 1 5
Press any key to continue_
```

remove()

- Eliminates a specified value from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.

```
template<class ForwardIterator, class Type>
   ForwardIterator remove(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val
   );
```

Parameters

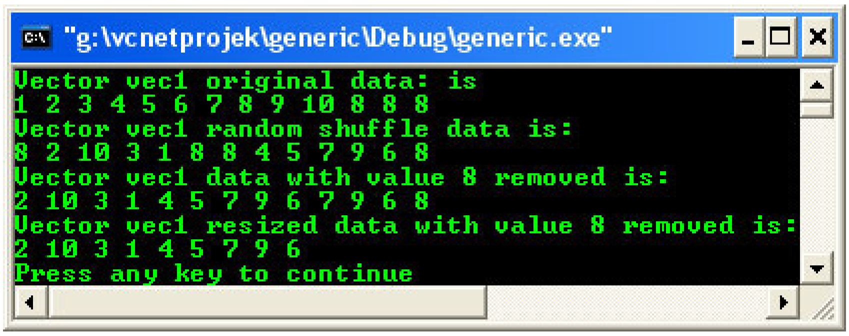
Parameter	Description
_First	A forward iterator addressing the position of the first element in the range from which elements are being removed.
_Last	A forward iterator addressing the position one past the final element in the range from which elements are being removed.
_Val	The value that is to be removed from the range.

Table 36.3

- The return value is a forward iterator addressing the new end position of the modified range, one past the final element of the remnant sequence free of the specified value.
- The range referenced must be valid; all pointers must be de-referenceable and within the sequence the last position is reachable from the first by incrementation.
- The order of the elements not removed remains stable.
- The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.
- The complexity is linear; there are (*_Last – _First*) comparisons for equality.
- The `list` Class class has a more efficient member function version of `remove()`, which also re-links pointers.

```
//algorithm, remove()
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
   vector <int> vec1;
   vector <int>::iterator Iter1, new_end;

   int i;
   for(i = 1; i <= 10; i++)
   vec1.push_back(i);

   int j;
   for(j = 0; j <= 2; j++)
     vec1.push_back(8);

   cout<<"Vector vec1 original data: is\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   random_shuffle(vec1.begin(), vec1.end());
   cout<<"Vector vec1 random shuffle data is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;

   //Remove elements with a value of 8
   new_end = remove(vec1.begin(), vec1.end(), 8);

   cout<<"Vector vec1 data with value 8 removed is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;

   //using erase, to change the sequence size
   vec1.erase(new_end, vec1.end());

   cout<<"Vector vec1 resized data with value 8 removed is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;
}
```

Output:



```
Vector vec1 original data: is
1 2 3 4 5 6 7 8 9 10 8 8 8
Vector vec1 random shuffle data is:
8 2 10 3 1 8 8 4 5 7 9 6 8
Vector vec1 data with value 8 removed is:
2 10 3 1 4 5 7 9 6 7 9 6 8
Vector vec1 resized data with value 8 removed is:
2 10 3 1 4 5 7 9 6
Press any key to continue
```

remove_copy()

- Copies elements from a source range to a destination range, except that elements of a specified value are not copied, without disturbing the order of the remaining elements and returning the end of a new destination range.

```
template<class InputIterator, class OutputIterator, class Type>
   OutputIterator remove_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      const Type& _Val
   );
```

Parameters

Parameter	Description
_First	An input iterator addressing the position of the first element in the range from which elements are being removed.
_Last	An input iterator addressing the position one past the final element in the range from which elements are being removed.
_Result	An output iterator addressing the position of the first element in the destination range to which elements are being removed.
_Val	The value that is to be removed from the range.

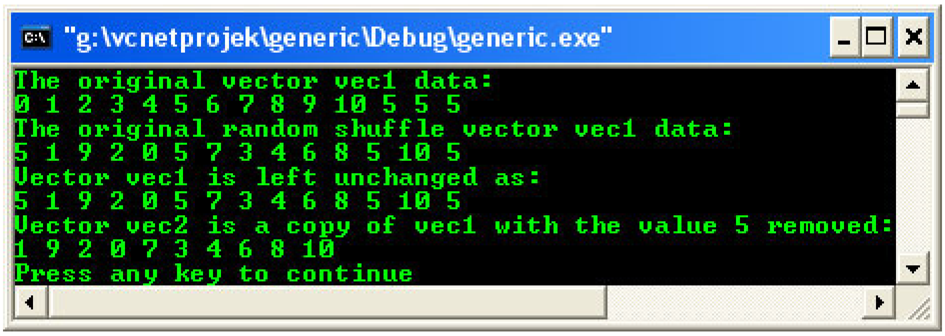
Table 36.4

- The return value is a forward iterator addressing the new end position of the destination range, one past the final element of the copy of the remnant sequence free of the specified value.
- The source and destination ranges referenced must be valid; all pointers must be de-referenceable and within the sequence the last position is reachable from the first by incrementation.
- There must be enough space in the destination range to contain the remnant elements that will be copied after elements of the specified value are removed.
- The order of the elements not removed remains stable.
- The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.
- The complexity is linear; there are (*_Last – _First*) comparisons for equality and at most (*_Last – _First*) assignments.

```
//algorithm, remove_copy()
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
   vector <int> vec1, vec2(10);
   vector <int>::iterator Iter1, Iter2, new_end;

   int i;
   for(i = 0; i <= 10; i++)
     vec1.push_back(i);

   int j;
   for(j = 0; j <= 2; j++)
     vec1.push_back(5);

   cout<<"The original vector vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   random_shuffle(vec1.begin(), vec1.end());
   cout<<"The original random shuffle vector vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Remove elements with a value of 5
   new_end = remove_copy(vec1.begin(), vec1.end(), vec2.begin(), 5);
```

```
    cout<<"Vector vec1 is left unchanged as:\n";
    for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
       cout<<*Iter1<<" ";
    cout<<endl;

    cout<<"Vector vec2 is a copy of vec1 with the value 5 removed:\n";
    for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
       cout<<*Iter2<<" ";
    cout<<endl;
}
```

Output:



```
The original vector vec1 data:
0 1 2 3 4 5 6 7 8 9 10 5 5 5
The original random shuffle vector vec1 data:
5 1 9 2 0 5 7 3 4 6 8 5 10 5
Vector vec1 is left unchanged as:
5 1 9 2 0 5 7 3 4 6 8 5 10 5
Vector vec2 is a copy of vec1 with the value 5 removed:
1 9 2 0 7 3 4 6 8 10
Press any key to continue
```

remove_copy_if()

- Copies elements from a source range to a destination range, except that satisfying a predicate are not copied, without disturbing the order of the remaining elements and returning the end of a new destination range.

```
template<class InputIterator, class OutputIterator, class Predicate>
   OutputIterator remove_copy_if(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      Predicate _Pred
   );
```

Parameters

Parameter	Description
_First	An input iterator addressing the position of the first element in the range from which elements are being removed.
_Last	An input iterator addressing the position one past the final element in the range from which elements are being removed.
_Result	An output iterator addressing the position of the first element in the destination range to which elements are being removed.
_Pred	The unary predicate that must be satisfied is the value of an element is to be replaced.

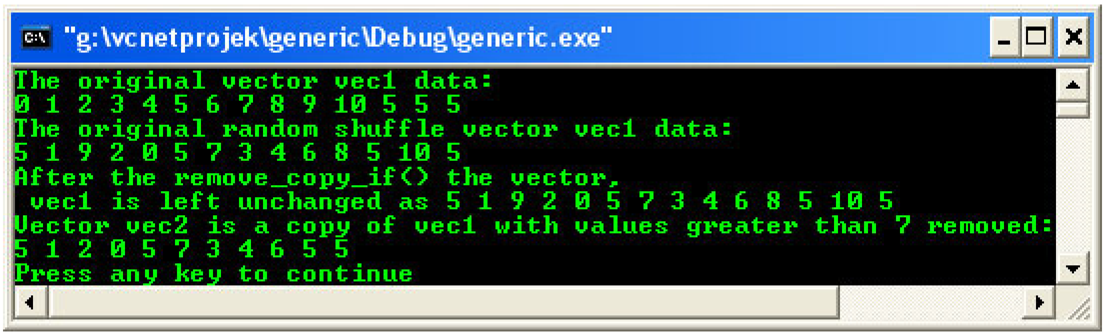
Table 36.5

- The return value is a forward iterator addressing the new end position of the destination range, one past the final element of the remnant sequence free of the elements satisfying the predicate.
- The source range referenced must be valid; all pointers must be de-referenceable and within the sequence the last position is reachable from the first by incrementation.
- There must be enough space in the destination range to contain the remnant elements that will be copied after elements of the specified value are removed.
- The order of the elements not removed remains stable.
- The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.
- The complexity is linear: there are (*_Last – _First*) comparisons for equality and at most (*_Last – _First*) assignments.

```
//algorithm, remove_copy_if()
#include <vector>
```

```
#include <algorithm>
#include <iostream>
using namespace std;

bool greathan(int value)
{ return value >7;}

int main()
{
   vector <int> vec1, vec2(14);
   vector <int>::iterator Iter1, Iter2, new_end;

   int i;
   for(i = 0; i <= 10; i++)
    vec1.push_back(i);

   int j;
   for(j = 0; j <= 2; j++)
    vec1.push_back(5);

   cout<<"The original vector vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   random_shuffle(vec1.begin(), vec1.end());
   cout<<"The original random shuffle vector vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Remove elements with a value greater than 7
   new_end = remove_copy_if(vec1.begin(), vec1.end(),
      vec2.begin(), greathan);

   cout<<"After the remove_copy_if() the vector,\n"
        <<" vec1 is left unchanged as ";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   cout<<"Vector vec2 is a copy of vec1 with values greater "
        <<"than 7 removed:\n";
   for(Iter2 = vec2.begin(); Iter2 != new_end; Iter2++)
      cout<<*Iter2<<" ";
   cout<<endl;
}
```

Output:



```
The original vector vec1 data:
0 1 2 3 4 5 6 7 8 9 10 5 5 5
The original random shuffle vector vec1 data:
5 1 9 2 0 5 7 3 4 6 8 5 10 5
After the remove_copy_if() the vector,
 vec1 is left unchanged as 5 1 9 2 0 5 7 3 4 6 8 5 10 5
Vector vec2 is a copy of vec1 with values greater than 7 removed:
5 1 2 0 5 7 3 4 6 5 5
Press any key to continue
```

remove_if()

- Eliminates elements that satisfy a predicate from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.

```
template<class ForwardIterator, class Predicate>
   ForwardIterator remove(
      ForwardIterator _First,
      ForwardIterator _Last,
      Predicate _Pred
   );
```

Parameters

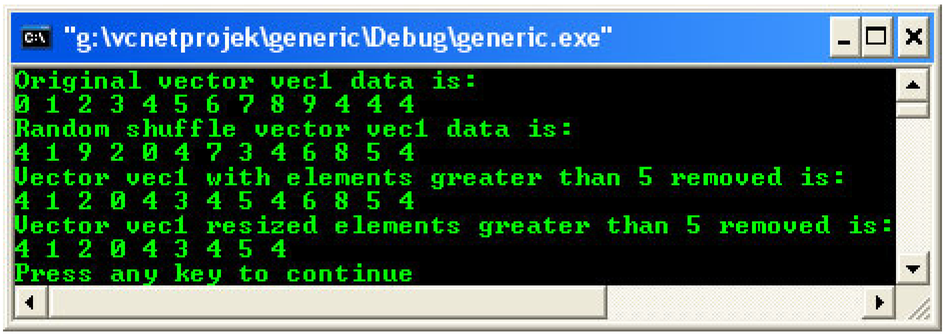
Parameter	Description
`_First`	A forward iterator pointing to the position of the first element in the range from which elements are being removed.
`_Last`	A forward iterator pointing to the position one past the final element in the range from which elements are being removed.
`_Pred`	The unary predicate that must be satisfied is the value of an element is to be replaced.

Table 36.6

- The return value is a forward iterator addressing the new end position of the modified range, one past the final element of the remnant sequence free of the specified value.
- The range referenced must be valid; all pointers must be de-referenceable and within the sequence the last position is reachable from the first by incrementation.
- The order of the elements not removed remains stable.
- The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.
- The complexity is linear: there are `(_Last – _First)` comparisons for equality.
- List has a more efficient member function version of remove which re-links pointers.

```
//algorithm, remove_if()
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

bool greathan(int value)
{return value >5;}

int main()
{
   vector <int> vec1;
   vector <int>::iterator Iter1, new_end;

   int i;
   for(i = 0; i <= 9; i++)
     vec1.push_back(i);

   int j;
   for(j = 0; j <= 2; j++)
     vec1.push_back(4);

   cout<<"Original vector vec1 data is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   random_shuffle(vec1.begin(), vec1.end());
   cout<<"Random shuffle vector vec1 data is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Remove elements satisfying predicate greater than
   new_end = remove_if(vec1.begin(), vec1.end(), greathan);

   cout<<"Vector vec1 with elements greater than 5 removed is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //using erase, to change the sequence size,
   vec1.erase(new_end, vec1.end());

   cout<<"Vector vec1 resized elements greater than 5 removed is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;
}
```

Output:



```
Original vector vec1 data is:
0 1 2 3 4 5 6 7 8 9 4 4 4
Random shuffle vector vec1 data is:
4 1 9 2 0 4 7 3 4 6 8 5 4
Vector vec1 with elements greater than 5 removed is:
4 1 2 0 4 3 4 5 4 6 8 5 4
Vector vec1 resized elements greater than 5 removed is:
4 1 2 0 4 3 4 5 4
Press any key to continue
```

replace()

- Examines each element in a range and replaces it if it matches a specified value.

```
template<class ForwardIterator, class Type>
   void replace(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _OldVal,
      const Type& _NewVal
   );
```

Parameters

Parameter	Description
_First	A forward iterator pointing to the position of the first element in the range from which elements are being replaced.
_Last	A forward iterator pointing to the position one past the final element in the range from which elements are being replaced.
_OldVal	The old value of the elements being replaced.
_NewVal	The new value being assigned to the elements with the old value.

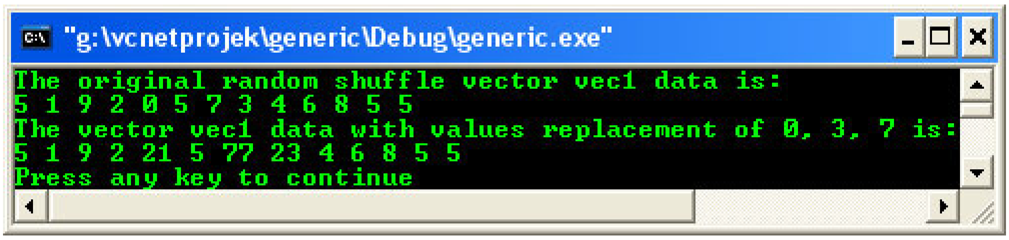
Table 36.7

- The range referenced must be valid; all pointers must be de-referenceable and within the sequence the last position is reachable from the first by incrementation.
- The order of the elements not replaced remains stable.
- The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.
- The complexity is linear; there are (*_Last – _First*) comparisons for equality and at most (*_Last – _First*) assignments of new values.

```
//algorithm, replace()
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
   vector <int> vec1;
   vector <int>::iterator Iter1;

   int i;
   for(i = 0; i <= 9; i++)
   vec1.push_back(i);

   int j;
   for(j = 0; j <= 2; j++)
   vec1.push_back(5);

   random_shuffle(vec1.begin(), vec1.end());
   cout<<"The original random shuffle vector vec1 data is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;

   //Replace elements with other values…
```

```
    replace (vec1.begin( ), vec1.end( ), 3, 23);
    replace (vec1.begin( ), vec1.end( ), 7, 77);
    replace (vec1.begin( ), vec1.end( ), 0, 21);

    cout<<"The vector vec1 data with values replacement of 0, 3, 7 is:\n";
    for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
    cout<<*Iter1<<" ";
    cout<<endl;
}
```

Output:



```
The original random shuffle vector vec1 data is:
5 1 9 2 0 5 7 3 4 6 8 5 5
The vector vec1 data with values replacement of 0, 3, 7 is:
5 1 9 2 21 5 77 23 4 6 8 5 5
Press any key to continue
```

replace_copy()

- Examines each element in a source range and replaces it if it matches a specified value while copying the result into a new destination range.

```
template<class InputIterator, class OutputIterator, class Type>
   OutputIterator replace_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      const Type& _OldVal,
      const Type& _NewVal
   );
```

Parameters

Parameter	Description
`_First`	An input iterator pointing to the position of the first element in the range from which elements are being replaced.
`_Last`	An input iterator pointing to the position one past the final element in the range from which elements are being replaced.
`_Result`	An output iterator pointing to the first element in the destination range to where the altered sequence of elements is being copied.
`_OldVal`	The old value of the elements being replaced.
`_NewVal`	The new value being assigned to the elements with the old value.

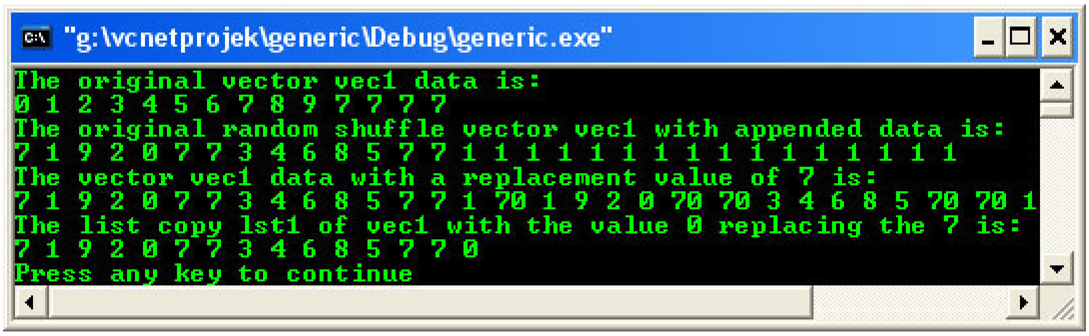
Table 36.8

- The value return is an output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.
- The source and destination ranges referenced must not overlap and must both be valid: all pointers must be de-referenceable and within the sequences the last position is reachable from the first by incrementation.
- The order of the elements not replaced remains stable.
- The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.
- The complexity is linear: there are (`_Last – _First`) comparisons for equality and at most (`_Last – _First`) assignments of new values.

```
//algorithm, replace_copy()
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
```

```
{
   vector <int> vec1;
   list <int> lst1 (15);
   vector <int>::iterator Iter1;
   list <int>::iterator lstIter;

   int i;
   for (i = 0; i <= 9; i++)
     vec1.push_back(i);

   int j;
   for (j = 0; j <= 3; j++)
     vec1.push_back(7);

   cout<<"The original vector vec1 data is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   random_shuffle(vec1.begin(), vec1.end());
   int k;
   for (k = 0; k <= 15; k++)
     vec1.push_back(1);

   cout<<"The original random shuffle vector vec1 with appended data is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Replace elements in one part of a vector with a value of 7
   //with a value of 70 and copy into another part of the vector
   replace_copy(vec1.begin(), vec1.begin() + 14, vec1.end( )-15, 7, 70);

   cout<<"The vector vec1 data with a replacement value of 7 is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Replace elements in a vector of a value 70
   //with a value of 1 and copy into a list
   replace_copy(vec1.begin(), vec1.begin() + 14, lst1.begin(), 70, 1);

   cout<<"The list copy lst1 of vec1 with the value 0 replacing the 7 is:\n";
   for(lstIter = lst1.begin(); lstIter != lst1.end(); lstIter++)
      cout<<*lstIter<<" ";
   cout<<endl;
}
```

Output:



```
The original vector vec1 data is:
0 1 2 3 4 5 6 7 8 9 7 7 7 7
The original random shuffle vector vec1 with appended data is:
7 1 9 2 0 7 7 3 4 6 8 5 7 7 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
The vector vec1 data with a replacement value of 7 is:
7 1 9 2 0 7 7 3 4 6 8 5 7 7 1 70 1 9 2 0 70 70 3 4 6 8 5 70 70 1
The list copy lst1 of vec1 with the value 0 replacing the 7 is:
7 1 9 2 0 7 7 3 4 6 8 5 7 7 0
Press any key to continue
```

replace_copy_if()

- Examines each element in a source range and replaces it if it satisfies a specified predicate while copying the result into a new destination range.

```
template<class InputIterator, class OutputIterator, class Predicate, class Type>
   OutputIterator replace_copy_if(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      Predicate _Pred,
      const Type& _Val
   );
```

Parameters

Parameter	Description
`_First`	An input iterator pointing to the position of the first element in the range from which elements are being replaced.
`_Last`	An input iterator pointing to the position one past the final element in the range from which elements are being replaced.
`_Result`	An output iterator pointing to the position of the first element in the destination range to which elements are being copied.
`_Pred`	The unary predicate that must be satisfied is the value of an element is to be replaced.
`_Val`	The new value being assigned to the elements whose old value satisfies the predicate.

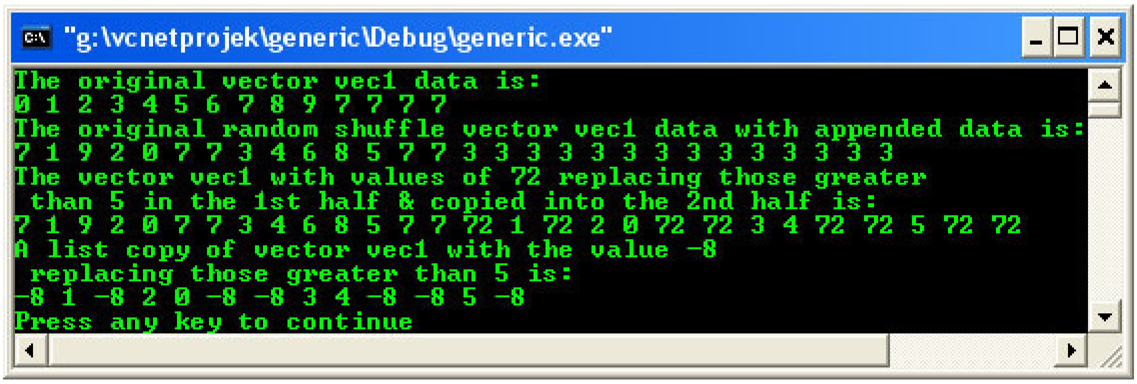
Table 36.9

- The source and destination ranges referenced must not overlap and must both be valid: all pointers must be de-referenceable and within the sequences the last position is reachable from the first by incrementation.
- The order of the elements not replaced remains stable.
- The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.
- The complexity is linear; there are `(_Last – _First)` comparisons for equality and at most `(_Last – _First)` assignments of new values.

```
//algorithm, replace_copy_if()
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

bool greaterthan(int value)
{return value > 5;}

int main()
{
   vector <int> vec1;
   list <int> lst1 (13);
   vector <int>::iterator Iter1;
   list <int>::iterator lstIter1;

   int i;
   for (i = 0; i <= 9; i++)
     vec1.push_back(i);
      int j;
   for (j = 0; j <= 3; j++)
     vec1.push_back(7);

   cout<<"The original vector vec1 data is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;

   random_shuffle(vec1.begin(), vec1.end());

   int k;
   for(k = 0; k <= 13; k++)
     vec1.push_back(3);

   cout<<"The original random shuffle vector vec1 data with appended data is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;

   //Replace elements with a value of 7 in the 1st half of a vector
   //with a value of 72 and copy it into the 2nd half of the vector
   replace_copy_if(vec1.begin(), vec1.begin() + 14, vec1.end() -14, greaterthan, 72);

   cout<<"The vector vec1 with values of 72 replacing those greater"
       <<"\n than 5 in the 1st half & copied into the 2nd half is:\n";
```

```
    for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
       cout<<*Iter1<<" ";
    cout<<endl;

    //Replace elements in a vector with a value of 72
    //with a value of -8 and copy into a list
    replace_copy_if(vec1.begin(), vec1.begin() + 13, lst1.begin(), greaterthan, -8);

    cout<<"A list copy of vector vec1 with the value -8\n replacing "
         <<"those greater than 5 is:\n";
    for(lstIter1 = lst1.begin(); lstIter1 != lst1.end(); lstIter1++)
       cout<<*lstIter1<<" ";
    cout<<endl;
}
```

Output:



```
The original vector vec1 data is:
0 1 2 3 4 5 6 7 8 9 7 7 7 7
The original random shuffle vector vec1 data with appended data is:
7 1 9 2 0 7 7 3 4 6 8 5 7 7 3 3 3 3 3 3 3 3 3 3 3 3 3 3
The vector vec1 with values of 72 replacing those greater
 than 5 in the 1st half & copied into the 2nd half is:
7 1 9 2 0 7 7 3 4 6 8 5 7 7 72 1 72 2 0 72 72 3 4 72 72 5 72 72
A list copy of vector vec1 with the value -8
 replacing those greater than 5 is:
-8 1 -8 2 0 -8 -8 3 4 -8 -8 5 -8
Press any key to continue
```

replace_if()

- Examines each element in a range and replaces it if it satisfies a specified predicate.

```
template<class ForwardIterator, class Predicate, class Type>
   void replace_if(
      ForwardIterator _First,
      ForwardIterator _Last,
      Predicate _Pred,
      const Type& _Val
   );
```

Parameters

Parameter	Description
_First	A forward iterator pointing to the position of the first element in the range from which elements are being replaced.
_Last	An iterator pointing to the position one past the final element in the range from which elements are being replaced.
_Pred	The unary predicate that must be satisfied is the value of an element is to be replaced.
_Val	The new value being assigned to the elements whose old value satisfies the predicate.

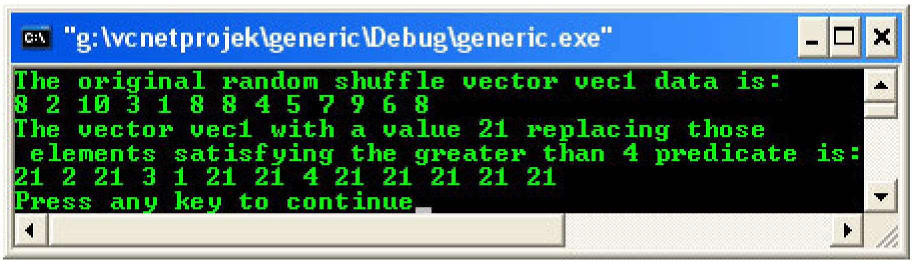
Table 36.10

- The range referenced must be valid; all pointers must be de-referenceable and within the sequence the last position is reachable from the first by incrementation.
- The order of the elements not replaced remains stable.
- The algorithm `replace_if()` is a generalization of the algorithm `replace()`, allowing any predicate to be specified, rather than equality to a specified constant value.
- The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.
- The complexity is linear: there are (*_Last – _First*) comparisons for equality and at most (*_Last – _First*) assignments of new values.

```
//algorithm, replace_if()
```

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

bool greaterthan(int value)
{return value > 4;}

int main()
{
   vector <int> vec1;
   vector <int>::iterator Iter1;

   int i;
   for (i = 1; i <= 10; i++)
     vec1.push_back(i);

   int j;
   for (j = 0; j <= 2; j++)
      vec1.push_back(8);

   random_shuffle(vec1.begin(), vec1.end());
   cout<<"The original random shuffle vector vec1 data is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Replace elements satisfying the predicate greaterthan
   //with a value of 21
   replace_if(vec1.begin(), vec1.end(), greaterthan, 21);

   cout<<"The vector vec1 with a value 21 replacing those\n "
       <<"elements satisfying the greater than 4 predicate is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;
}
```

Output:



```
The original random shuffle vector vec1 data is:
8 2 10 3 1 8 8 4 5 7 9 6 8
The vector vec1 with a value 21 replacing those
 elements satisfying the greater than 4 predicate is:
21 2 21 3 1 21 21 4 21 21 21 21 21
Press any key to continue
```

reverse()

- Reverses the order of the elements within a range.

```
template<class BidirectionalIterator>
   void reverse(
      BidirectionalIterator _First,
      BidirectionalIterator _Last
   );
```

Parameters

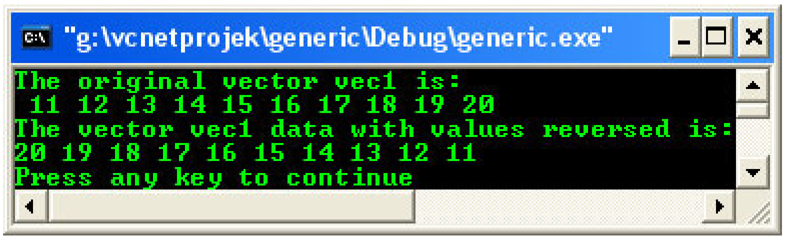
Parameter	Description
_First	A bidirectional iterator pointing to the position of the first element in the range within which the elements are being permuted.
_Last	A bidirectional iterator pointing to the position one past the final element in the range within which the elements are being permuted.

Table 36.11

- The source range referenced must be valid; all pointers must be de-referenceable and within the sequence the last position is reachable from the first by incrementation.

```
//algorithm, reverse()
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
   vector <int> vec1;
   vector <int>::iterator Iter1;

   int i;
   for(i = 11; i <= 20; i++)
      vec1.push_back(i);

   cout<<"The original vector vec1 is:\n ";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Reverse the elements in the vector
   reverse(vec1.begin(), vec1.end());

   cout<<"The vector vec1 data with values reversed is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;
}
```

Output:



```
The original vector vec1 is:
 11 12 13 14 15 16 17 18 19 20
The vector vec1 data with values reversed is:
20 19 18 17 16 15 14 13 12 11
Press any key to continue
```

reverse_copy()

- Reverses the order of the elements within a source range while copying them into a destination range

```
template<class BidirectionalIterator, class OutputIterator>
   OutputIterator reverse_copy(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      OutputIterator _Result
   );
```

Parameters

Parameter	Description
_First	A bidirectional iterator pointing to the position of the first element in the source range within which the elements are being permuted.
_Last	A bidirectional iterator pointing to the position one past the final element in the source range within which the elements are being permuted.
_Result	An output iterator pointing to the position of the first element in the destination range to which elements are being copied.

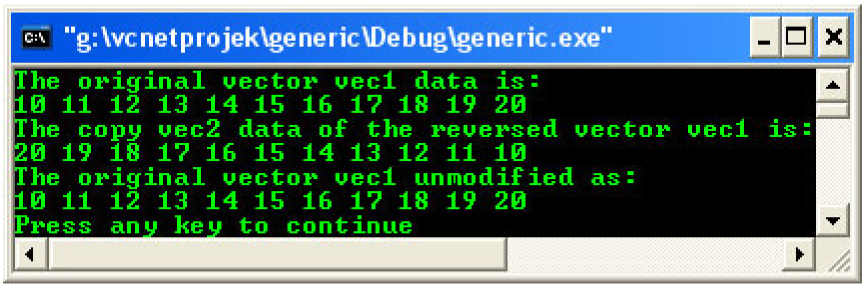
Table 36.12

- The return value is an output iterator pointing to the position one past the final element in the destination range to where the altered sequence of elements is being copied.
- The source and destination ranges referenced must be valid; all pointers must be de-referenceable and within the sequence the last position is reachable from the first by incrementation.

```
//algorithm, reverse_copy()
#include <vector>
#include <algorithm>
```

```
#include <iostream>
using namespace std;

int main()
{
   vector <int> vec1, vec2(11);
   vector <int>::iterator Iter1, Iter2;

   int i;
   for(i = 10; i <= 20; i++)
      vec1.push_back(i);

   cout<<"The original vector vec1 data is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Reverse the elements in the vector
   reverse_copy(vec1.begin(), vec1.end(), vec2.begin());

   cout<<"The copy vec2 data of the reversed vector vec1 is:\n";
   for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
      cout<<*Iter2<<" ";
   cout<<endl;

   cout<<"The original vector vec1 unmodified as:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;
}
```

Output:



```
The original vector vec1 data is:
10 11 12 13 14 15 16 17 18 19 20
The copy vec2 data of the reversed vector vec1 is:
20 19 18 17 16 15 14 13 12 11 10
The original vector vec1 unmodified as:
10 11 12 13 14 15 16 17 18 19 20
Press any key to continue
```

rotate()

- Exchanges the elements in two adjacent ranges.

```
template<class ForwardIterator>
   void rotate(
      ForwardIterator _First,
      ForwardIterator _Middle,
      ForwardIterator _Last
   );
```

Parameters

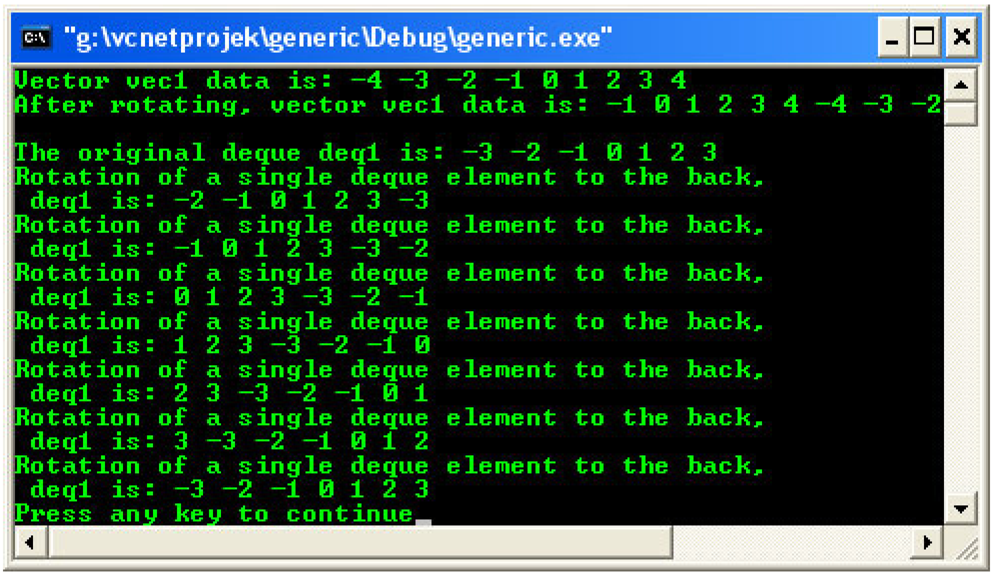
Parameter	Description
`_First`	A forward iterator addressing the position of the first element in the range to be rotated.
`_Middle`	A forward iterator defining the boundary within the range that addresses the position of the first element in the second part of the range whose elements are to be exchanged with those in the first part of the range.
`_Last`	A forward iterator addressing the position one past the final element in the range to be rotated.

Table 36.13

- The ranges referenced must be valid; all pointers must be de-referenceable and within the sequence the last position is reachable from the first by incrementation.
- The complexity is linear with at most (`_Last – _First`) swaps.

```
//algorithm, rotate()
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
   vector <int> vec1;
   deque <int> deq1;
   vector <int>::iterator vec1Iter1;
   deque<int>::iterator deq1Iter1;

   int i;
   for(i = -4; i <= 4; i++)
      vec1.push_back(i);

   int j;
   for(j = -3; j <= 3; j++)
     deq1.push_back(j);

   cout<<"Vector vec1 data is: ";
   for(vec1Iter1 = vec1.begin(); vec1Iter1 != vec1.end(); vec1Iter1++)
      cout<<*vec1Iter1 <<" ";
   cout<<endl;

   //Let rotates...
   rotate(vec1.begin(), vec1.begin() + 3, vec1.end());
   cout<<"After rotating, vector vec1 data is: ";
   for(vec1Iter1 = vec1.begin(); vec1Iter1 != vec1.end(); vec1Iter1++)
      cout<<*vec1Iter1<<" ";
   cout<<endl;

   cout<<"\nThe original deque deq1 is: ";
   for(deq1Iter1 = deq1.begin(); deq1Iter1 != deq1.end(); deq1Iter1++)
      cout<<*deq1Iter1<<" ";
   cout<<endl;

   //Let rotates…
   int k = 1;
   while(k <= deq1.end() - deq1.begin())
   {
      rotate(deq1.begin(), deq1.begin() + 1, deq1.end());
  cout<<"Rotation of a single deque element to the back,\n deq1 is: ";
      for(deq1Iter1 = deq1.begin(); deq1Iter1 != deq1.end(); deq1Iter1++)
         cout<<*deq1Iter1<<" ";
      cout<<endl;
      k++;
   }
}
```

Output:



```
Vector vec1 data is: -4 -3 -2 -1 0 1 2 3 4
After rotating, vector vec1 data is: -1 0 1 2 3 4 -4 -3 -2

The original deque deq1 is: -3 -2 -1 0 1 2 3
Rotation of a single deque element to the back,
 deq1 is: -2 -1 0 1 2 3 -3
Rotation of a single deque element to the back,
 deq1 is: -1 0 1 2 3 -3 -2
Rotation of a single deque element to the back,
 deq1 is: 0 1 2 3 -3 -2 -1
Rotation of a single deque element to the back,
 deq1 is: 1 2 3 -3 -2 -1 0
Rotation of a single deque element to the back,
 deq1 is: 2 3 -3 -2 -1 0 1
Rotation of a single deque element to the back,
 deq1 is: 3 -3 -2 -1 0 1 2
Rotation of a single deque element to the back,
 deq1 is: -3 -2 -1 0 1 2 3
Press any key to continue_
```

rotate_copy()

- Exchanges the elements in two adjacent ranges within a source range and copies the result to a destination range.

```
template<class ForwardIterator, class OutputIterator>
   OutputIterator rotate_copy(
      ForwardIterator _First,
      ForwardIterator _Middle,
      ForwardIterator _Last,
      OutputIterator _Result
   );
```

Parameters

Parameter	Description
_First	A forward iterator addressing the position of the first element in the range to be rotated.
_Middle	A forward iterator defining the boundary within the range that addresses the position of the first element in the second part of the range whose elements are to be exchanged with those in the first part of the range.
_Last	A forward iterator addressing the position one past the final element in the range to be rotated.
_Result	An output iterator addressing the position of the first element in the destination range.

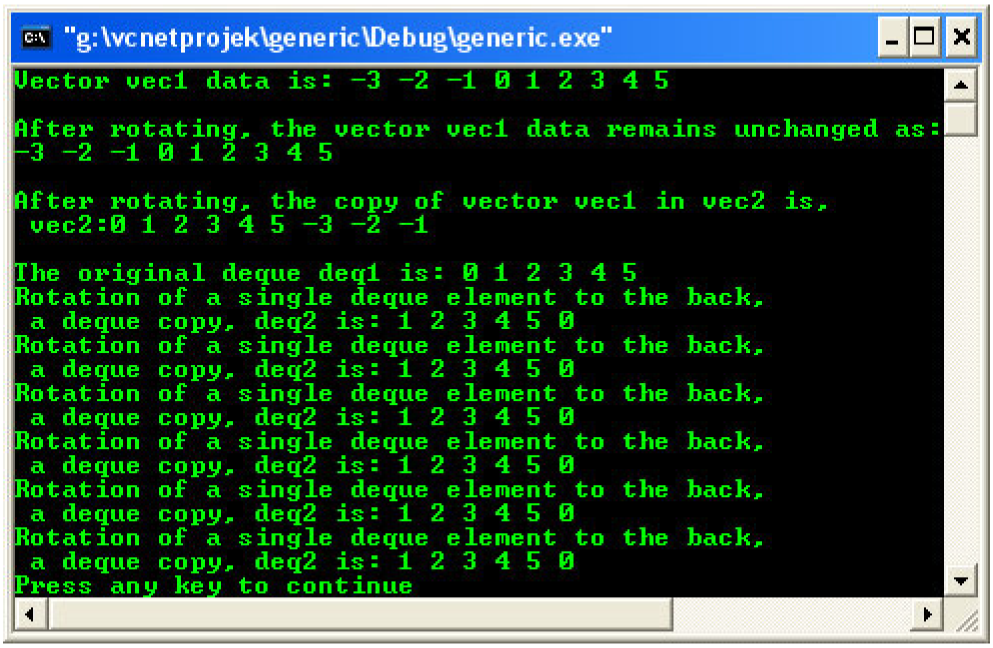
Table 36.14

- The return value is an output iterator addressing the position one past the final element in the destination range.
- The ranges referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.
- The complexity is linear with at most (_Last – _First) swaps.

```
//algorithm, rotate_copy()
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
   vector <int> vec1, vec2(9);
   deque <int> deq1, deq2(6);
   vector <int>::iterator vec1Iter, vec2Iter;
   deque<int>::iterator deq1Iter, deq2Iter;

   int i;
   for(i = -3; i <= 5; i++)
     vec1.push_back(i);

   int j;
   for(j =0; j <= 5; j++)
     deq1.push_back(j);

   cout<<"Vector vec1 data is: ";
   for(vec1Iter = vec1.begin(); vec1Iter != vec1.end(); vec1Iter++)
      cout<<*vec1Iter<<" ";
   cout<<endl;

   rotate_copy(vec1.begin(), vec1.begin() + 3, vec1.end(), vec2.begin());
   cout<<"\nAfter rotating, the vector vec1 data remains unchanged as:\n";
   for(vec1Iter = vec1.begin(); vec1Iter != vec1.end(); vec1Iter++)
      cout<<*vec1Iter<<" ";
   cout<<endl;

   cout<<"\nAfter rotating, the copy of vector vec1 in vec2 is,\n vec2:";
   for(vec2Iter = vec2.begin(); vec2Iter != vec2.end(); vec2Iter++)
      cout<<*vec2Iter<<" ";
   cout<<endl;
```

```
    cout<<"\nThe original deque deq1 is: ";
    for(deq1Iter = deq1.begin(); deq1Iter != deq1.end(); deq1Iter++)
       cout<<*deq1Iter<<" ";
    cout<<endl;

    int k = 1;
    while(k <= deq1.end() - deq1.begin())
    {
       rotate_copy(deq1.begin(), deq1.begin() + 1, deq1.end(), deq2.begin());
   cout<<"Rotation of a single deque element to the back,\n a deque copy, deq2 is: ";
       for(deq2Iter = deq2.begin(); deq2Iter != deq2.end(); deq2Iter++)
          cout<<*deq2Iter<<" ";
       cout<<endl;
       k++;
    }
}
```

Output:



```
Vector vec1 data is: -3 -2 -1 0 1 2 3 4 5

After rotating, the vector vec1 data remains unchanged as:
-3 -2 -1 0 1 2 3 4 5

After rotating, the copy of vector vec1 in vec2 is,
 vec2:0 1 2 3 4 5 -3 -2 -1

The original deque deq1 is: 0 1 2 3 4 5
Rotation of a single deque element to the back,
 a deque copy, deq2 is: 1 2 3 4 5 0
Rotation of a single deque element to the back,
 a deque copy, deq2 is: 1 2 3 4 5 0
Rotation of a single deque element to the back,
 a deque copy, deq2 is: 1 2 3 4 5 0
Rotation of a single deque element to the back,
 a deque copy, deq2 is: 1 2 3 4 5 0
Rotation of a single deque element to the back,
 a deque copy, deq2 is: 1 2 3 4 5 0
Rotation of a single deque element to the back,
 a deque copy, deq2 is: 1 2 3 4 5 0
Press any key to continue
```

search()

- Searches for the first occurrence of a sequence within a target range whose elements are equal to those in a given sequence of elements or whose elements are equivalent in a sense specified by a binary predicate to the elements in the given sequence.

```
template<class ForwardIterator1, class ForwardIterator2>
   ForwardIterator1 search(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      ForwardIterator2 _First2,
      ForwardIterator2 _Last2
   );
template<class ForwardIterator1, class ForwardIterator2, class Pr>
   ForwardIterator1 search(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      ForwardIterator2 _First2,
      ForwardIterator2 _Last2
      BinaryPredicate _Comp
   );
```

Parameters

Parameter	Description
_First1	A forward iterator addressing the position of the first element in the range to be searched.
_Last1	A forward iterator addressing the position one past the final element in the range to be searched.

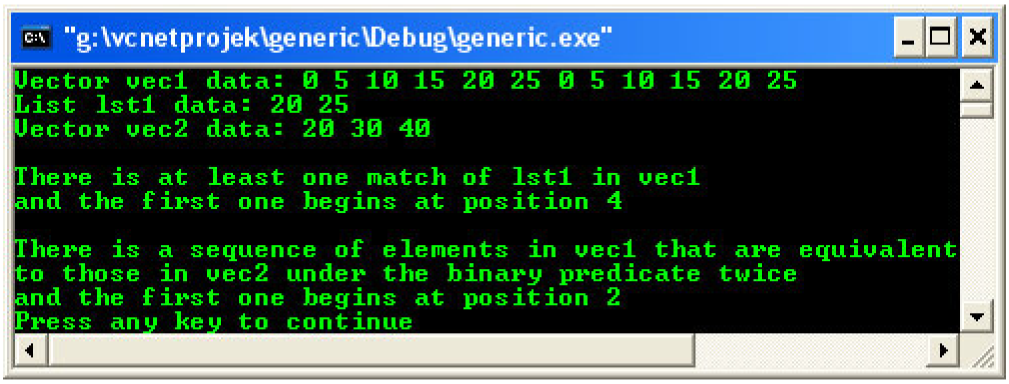
_First2	A forward iterator addressing the position of the first element in the range to be matched.
_Last2	A forward iterator addressing the position one past the final element in the range to be matched.
_Comp	User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns true when satisfied and false when not satisfied.

Table 36.15

- The return value is a forward iterator addressing the position of the first element of the first subsequence that matches the specified sequence or that is equivalent in a sense specified by a binary predicate.
- The `operator==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.
- The ranges referenced must be valid; all pointers must be de-referenceable and within each sequence the last position is reachable from the first by incrementation.
- Average complexity is linear with respect to the size of the searched range, and worst case complexity is also linear with respect to the size of the sequence being searched for.

```
//algorithm, search()
//compiled with some type conversion
//warning…
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

//Return whether second element is twice the first
bool twice(int elem1, int elem2)
{return (2 * elem1 == elem2);}

int main()
{
   vector <int> vec1, vec2;
   list <int> lst1;
   vector <int>::iterator Iter1, Iter2;
   list <int>::iterator lst1_Iter, lst1_inIter;

   int i;
   for(i = 0; i <= 5; i++)
      vec1.push_back(5*i);

   for(i = 0; i <= 5; i++)
      vec1.push_back(5*i);

   int j;
   for(j = 4; j <= 5; j++)
      lst1.push_back(5*j);

   int k;
   for(k = 2; k <= 4; k++)
      vec2.push_back(10*k);

   cout<<"Vector vec1 data: ";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   cout<<"List lst1 data: ";
   for(lst1_Iter = lst1.begin(); lst1_Iter!= lst1.end(); lst1_Iter++)
      cout<<*lst1_Iter<<" ";
   cout<<endl;

   cout<<"Vector vec2 data: ";
   for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
      cout<<*Iter2<<" ";
      cout<<endl<<endl;

   //Searching vec1 for first match to lst1 under identity
   vector <int>::iterator result1;
   result1 = search (vec1.begin(), vec1.end(), lst1.begin(), lst1.end());
   if(result1 == vec1.end())
      cout<<"There is no match of lst1 in vec1."<<endl;
   else
      cout<<"There is at least one match of lst1 in vec1"
           <<"\nand the first one begins at "
           <<"position "<< result1 - vec1.begin( )<<endl;

   //Searching vec1 for a match to lst1 under the binary predicate twice
   vector <int>::iterator result2;
   result2 = search(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(), twice);

   if(result2 == vec1.end())
      cout<<"\nThere is no match of lst1 in vec1."<<endl;
   else
      cout<<"\nThere is a sequence of elements in vec1 that "
          <<"are equivalent\nto those in vec2 under the binary "
          <<"predicate twice\nand the first one begins at position "
          <<result2 - vec1.begin()<<endl;
}
```

Output:



```
Vector vec1 data: 0 5 10 15 20 25 0 5 10 15 20 25
List lst1 data: 20 25
Vector vec2 data: 20 30 40

There is at least one match of lst1 in vec1
and the first one begins at position 4

There is a sequence of elements in vec1 that are equivalent
to those in vec2 under the binary predicate twice
and the first one begins at position 2
Press any key to continue
```

search_n()

- Searches for the first subsequence in a range that of a specified number of elements having a particular value or a relation to that value as specified by a binary predicate.

```
template<class ForwardIterator1, class Diff2, class Type>
   ForwardIterator1 search_n(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      Size2 _Count,
      const Type& _Val
   );
template<class ForwardIterator1, class Size2, class Type, class BinaryPredicate>
   ForwardIterator1 search_n(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      Size2 _Count,
      const Type& _Val,
      BinaryPredicate _Comp
   );
```

Parameters

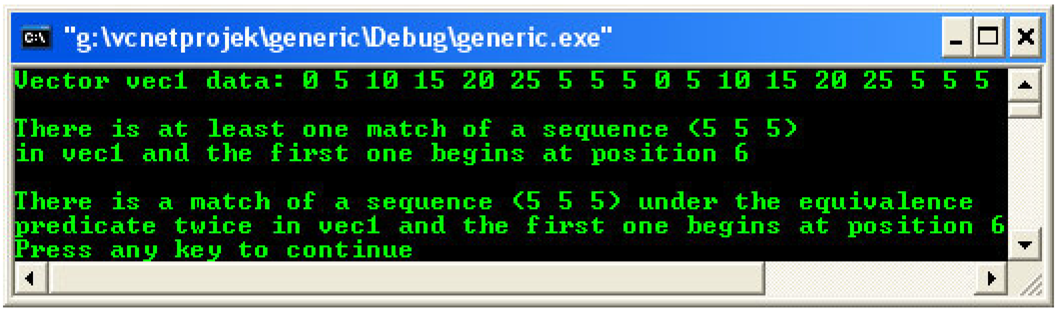
Parameter	Description
_First1	A forward iterator addressing the position of the first element in the range to be searched.
_Last1	A forward iterator addressing the position one past the final element in the range to be searched.
_Count	The size of the subsequence being searched for.
_Val	The value of the elements in the sequence being searched for.
_Comp	User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns true when satisfied and false when not satisfied.

Table 36.16

- The return value is a forward iterator addressing the position of the first element of the first subsequence that matches the specified sequence or that is equivalent in a sense specified by a binary predicate.
- The `operator==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.
- The range referenced must be valid; all pointers must be de-referenceable and within the sequence the last position is reachable from the first by incrementation.
- Complexity is linear with respect to the size of the searched.

```
//algorithm, search_n()
//some type conversion warning
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

//Return whether second element is twice the first
bool twice(int elem1, int elem2)
{return 2 * elem1 == elem2;}

int main()
{
   vector <int> vec1, vec2;
   vector <int>::iterator Iter1;

   int i;
   for(i = 0; i <= 5; i++)
     vec1.push_back(5*i);

   for(i = 0; i <= 2; i++)
     vec1.push_back(5);

   for(i = 0; i <= 5; i++)
     vec1.push_back(5*i);


   for(i = 0; i <= 2; i++)
     vec1.push_back(5);

   cout<<"Vector vec1 data: ";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Searching vec1 for first match to (5 5 5) under identity
   vector <int>::iterator result1;
   result1 = search_n(vec1.begin(), vec1.end(), 3, 5);

   if(result1 == vec1.end())
      cout<<"\nThere is no match for a sequence (5 5 5) in vec1."<<endl;
   else
      cout<<"\nThere is at least one match of a sequence (5 5 5)"
          <<"\nin vec1 and the first one begins at "
          <<"position "<<result1 - vec1.begin()<<endl;

   //Searching vec1 for first match to (5 5 5) under twice
   vector <int>::iterator result2;
   result2 = search_n(vec1.begin(), vec1.end(), 3, 5);

   if(result2 == vec1.end())
      cout<<"\nThere is no match for a sequence (5 5 5) in vec1"
          <<" under the equivalence predicate twice."<<endl;
   else
      cout<<"\nThere is a match of a sequence (5 5 5) "
          <<"under the equivalence\npredicate twice"
          <<" in vec1 and the first one begins at "
          <<"position "<<result1 - vec1.begin()<<endl;
}
```

Output:



```
Vector vec1 data: 0 5 10 15 20 25 5 5 5 0 5 10 15 20 25 5 5 5

There is at least one match of a sequence (5 5 5)
in vec1 and the first one begins at position 6

There is a match of a sequence (5 5 5) under the equivalence
predicate twice in vec1 and the first one begins at position 6
Press any key to continue
```

set_difference()

- Unites all of the elements that belong to one sorted source range, but not to a second sorted source range, into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator set_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class
BinaryPredicate>
   OutputIterator set_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

Parameter	Description
_First1	An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.
_Last1	An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.
_First2	An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.
_Last2	An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.
_Result	An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the difference of the two source ranges.
_Comp	User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return true when the first element is less than the second element and false otherwise.

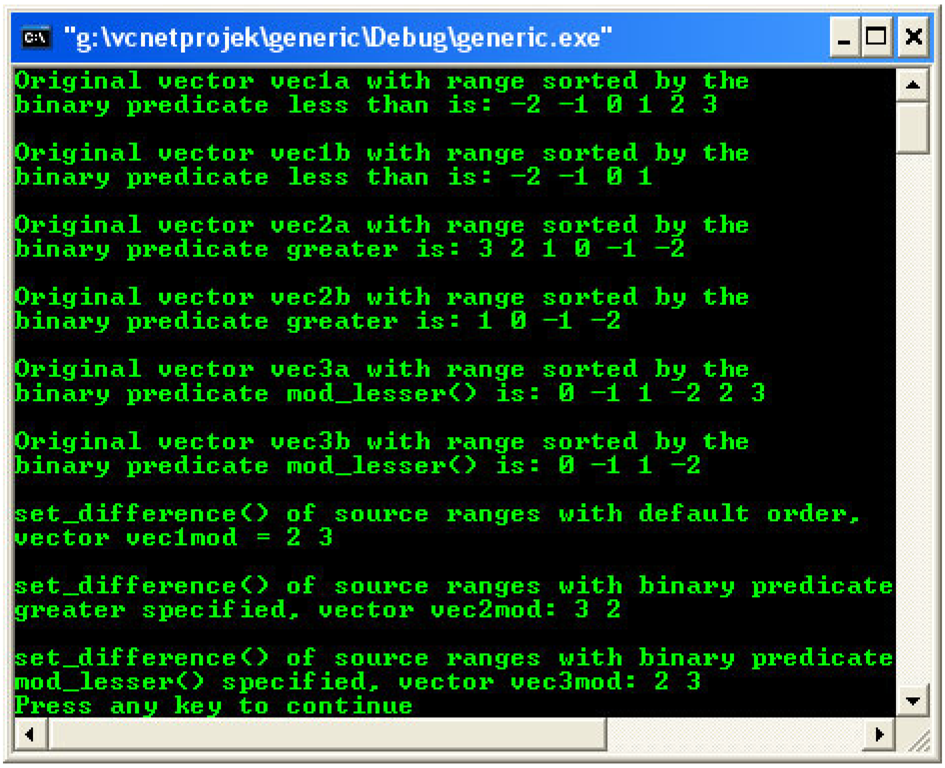
Table 36.17

- The return value is an output iterator addressing the position one past the last element in the sorted destination range representing the difference of the two source ranges.
- The sorted source ranges referenced must be valid; all pointers must be de-referenceable and within each sequence the last position must be reachable from the first by incrementation.

- The destination range should not overlap either of the source ranges and should be large enough to contain the destination range.
- The sorted source ranges must each be arranged as a precondition to the application of the `set_difference()` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.
- The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges are not modified by the algorithm merge.
- The value types of the input iterators need be less-than-comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent, in the sense that neither is less than the other or that one is less than the other. This results in an ordering between the nonequivalent elements.
- When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range.
- If the source ranges contain duplicates of an element such that there are more in the first source range than in the second, then the destination range will contain the number by which the occurrences of those elements in the first source range exceed the occurrences of those elements in the second source range.
- The complexity of the algorithm is linear with at most `2*((_Last1 – _First1)-(_Last2 – _First2))–1` comparisons for nonempty source ranges.

```
//algorithm, set_difference()
#include <vector>
#include <algorithm>
//For greater<int>()
#include <functional>
#include <iostream>
using namespace std;

//Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser(int elem1, int elem2)
{
   if (elem1 < 0)
      elem1 = - elem1;
   if (elem2 < 0)
      elem2 = - elem2;
   return (elem1 < elem2);
}

int main()
{
   vector <int> vec1a, vec1b, vec1(12);
   vector <int>::iterator Iter1a, Iter1b, Iter1, Result1;

   //Constructing vectors vec1a & vec1b with default less-than ordering
   int i;
   for(i = -2; i <= 3; i++)
      vec1a.push_back(i);

   int j;
   for(j =-2; j <= 1; j++)
      vec1b.push_back(j);

   cout<<"Original vector vec1a with range sorted by the\n"
   <<"binary predicate less than is: ";
   for(Iter1a = vec1a.begin(); Iter1a != vec1a.end(); Iter1a++)
      cout<<*Iter1a<<" ";
   cout<<endl;

   cout<<"\nOriginal vector vec1b with range sorted by the\n"
   <<"binary predicate less than is: ";
   for(Iter1b = vec1b.begin(); Iter1b != vec1b.end(); Iter1b++)
      cout<<*Iter1b<<" ";
   cout<<endl;

   //Constructing vectors vec2a & vec2b with ranges sorted by greater
   vector <int> vec2a(vec1a), vec2b(vec1b), vec2(vec1);
   vector <int>::iterator Iter2a, Iter2b, Iter2, Result2;
   sort(vec2a.begin(), vec2a.end(), greater<int>());
   sort(vec2b.begin(), vec2b.end(), greater<int>());

   cout<<"\nOriginal vector vec2a with range sorted by the\n"
   <<"binary predicate greater is: ";
   for(Iter2a = vec2a.begin(); Iter2a != vec2a.end(); Iter2a++)
      cout<<*Iter2a<<" ";
   cout<<endl;
   cout<<"\nOriginal vector vec2b with range sorted by the\n"
   <<"binary predicate greater is: ";
   for(Iter2b = vec2b.begin(); Iter2b != vec2b.end(); Iter2b++)
      cout<<*Iter2b<<" ";
   cout<<endl;

   //Constructing vectors vec3a & vec3b with ranges sorted by mod_lesser
   vector <int> vec3a(vec1a), vec3b(vec1b), vec3(vec1);
   vector <int>::iterator Iter3a, Iter3b, Iter3, Result3;
   sort(vec3a.begin(), vec3a.end(), mod_lesser);
   sort(vec3b.begin(), vec3b.end(), mod_lesser);

   cout<<"\nOriginal vector vec3a with range sorted by the\n"
   <<"binary predicate mod_lesser() is: ";
   for(Iter3a = vec3a.begin(); Iter3a != vec3a.end(); Iter3a++)
      cout<<*Iter3a<<" ";
   cout<<endl;

   cout<<"\nOriginal vector vec3b with range sorted by the\n"
   <<"binary predicate mod_lesser() is: ";
   for(Iter3b = vec3b.begin(); Iter3b != vec3b.end(); Iter3b++)
      cout<<*Iter3b<<" ";
   cout<<endl;

   //To combine into a difference in ascending
   //order with the default binary predicate less <int>()
   Result1 = set_difference(vec1a.begin(), vec1a.end(),
      vec1b.begin(), vec1b.end(), vec1.begin());
   cout<<"\nset_difference() of source ranges with default order,"
       <<"\nvector vec1mod = ";
   for(Iter1 = vec1.begin(); Iter1 != Result1; Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //To combine into a difference in descending
   //order specify binary predicate greater<int>()
   Result2 = set_difference(vec2a.begin(), vec2a.end(),
      vec2b.begin(), vec2b.end(), vec2.begin(), greater <int>());
   cout<<"\nset_difference() of source ranges with binary"
   <<" predicate\ngreater specified, vector vec2mod: ";
   for(Iter2 = vec2.begin(); Iter2 != Result2; Iter2++)
      cout<<*Iter2<<" ";
   cout<<endl;

   //To combine into a difference applying a user
   //defined binary predicate mod_lesser()
   Result3 = set_difference(vec3a.begin(), vec3a.end(),
      vec3b.begin(), vec3b.end(), vec3.begin(), mod_lesser);
   cout<<"\nset_difference() of source ranges with binary "
   <<"predicate\nmod_lesser() specified, vector vec3mod: ";
   for(Iter3 = vec3.begin(); Iter3 != Result3; Iter3++)
      cout<<*Iter3<<" ";
   cout<<endl;
}
```

Output



```
Original vector vec1a with range sorted by the
binary predicate less than is: -2 -1 0 1 2 3

Original vector vec1b with range sorted by the
binary predicate less than is: -2 -1 0 1

Original vector vec2a with range sorted by the
binary predicate greater is: 3 2 1 0 -1 -2

Original vector vec2b with range sorted by the
binary predicate greater is: 1 0 -1 -2

Original vector vec3a with range sorted by the
binary predicate mod_lesser() is: 0 -1 1 -2 2 3

Original vector vec3b with range sorted by the
binary predicate mod_lesser() is: 0 -1 1 -2

set_difference() of source ranges with default order,
vector vec1mod = 2 3

set_difference() of source ranges with binary predicate
greater specified, vector vec2mod: 3 2

set_difference() of source ranges with binary predicate
mod_lesser() specified, vector vec3mod: 2 3
Press any key to continue
```

set_intersection()

- Unites all of the elements that belong to both sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
   OutputIterator set_intersection(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class
BinaryPredicate>
   OutputIterator set_intersection(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

Parameter	Description
_First1	An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.
_Last1	An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.
_First2	An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.
_Last2	An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.
_Result	An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range

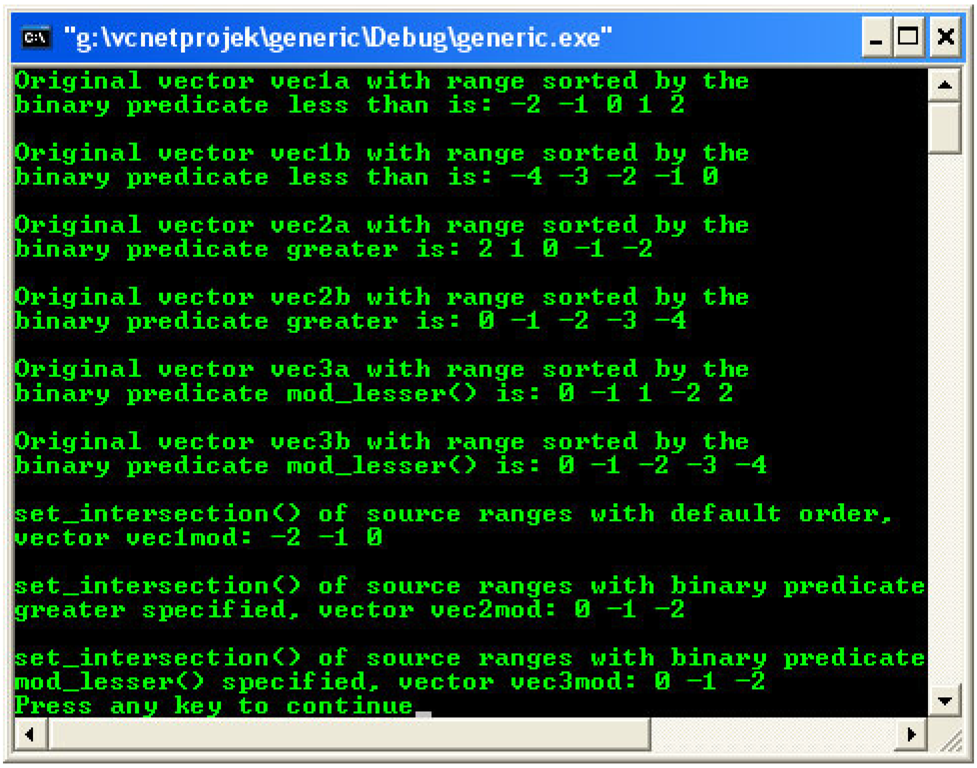
	representing the intersection of the two source ranges.
`_Comp`	User-defined predicate function object that defines the sense in which one element is greater than another. The binary predicate takes two arguments and should return true when the first element is less than the second element and false otherwise.

Table 36.18

- The return value is an output iterator addressing the position one past the last element in the sorted destination range representing the intersection of the two source ranges.
- The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.
- The destination range should not overlap either of the source ranges and should be large enough to contain the destination range.
- The sorted source ranges must each be arranged as a precondition to the application of the merge algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.
- The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges are not modified by the algorithm.
- The value types of the input iterators need be less-than comparable to be ordered, so that, given two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other.
- This results in an ordering between the nonequivalent elements.
- When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range.
- If the source ranges contain duplicates of an element, then the destination range will contain the maximum number of those elements that occur in both source ranges.
- The complexity of the algorithm is linear with at most `2*((_Last1 – _First1)-(_Last2 – _First2))-1` comparisons for nonempty source ranges.

```
//algorithm, set_intersection()
#include <vector>
#include <algorithm>
//For greater<int>()
#include <functional>
#include <iostream>
using namespace std;

//Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser(int elem1, int elem2)
{
   if(elem1 < 0)
      elem1 = - elem1;
   if(elem2 < 0)
      elem2 = - elem2;
   return (elem1 < elem2);
}

int main()
{
   vector <int> vec1a, vec1b, vec1(12);
   vector <int>::iterator Iter1a, Iter1b, Iter1, Result1;

   //Constructing vectors vec1a & vec1b with default less than ordering
   int i;
   for(i = -2; i <= 2; i++)
     vec1a.push_back(i);

   int j;
   for(j = -4; j <= 0; j++)
     vec1b.push_back(j);

   cout<<"Original vector vec1a with range sorted by the\n"
   <<"binary predicate less than is: ";
   for(Iter1a = vec1a.begin(); Iter1a != vec1a.end(); Iter1a++)
      cout<<*Iter1a<<" ";
   cout<<endl;

   cout<<"\nOriginal vector vec1b with range sorted by the\n"
   <<"binary predicate less than is: ";
   for(Iter1b = vec1b.begin(); Iter1b != vec1b.end(); Iter1b++)
      cout<<*Iter1b<<" ";
   cout<<endl;

   //Constructing vectors vec2a & vec2b with ranges sorted by greater
   vector <int> vec2a(vec1a), vec2b(vec1b), vec2(vec1);
   vector <int>::iterator Iter2a, Iter2b, Iter2, Result2;
   sort(vec2a.begin(), vec2a.end(), greater<int>());
   sort(vec2b.begin(), vec2b.end(), greater<int>());

   cout<<"\nOriginal vector vec2a with range sorted by the\n"
   <<"binary predicate greater is: ";
   for(Iter2a = vec2a.begin(); Iter2a != vec2a.end(); Iter2a++)
      cout<<*Iter2a<<" ";
   cout<<endl;

   cout<<"\nOriginal vector vec2b with range sorted by the\n"
   <<"binary predicate greater is: ";
   for(Iter2b = vec2b.begin(); Iter2b != vec2b.end(); Iter2b++)
      cout<<*Iter2b<<" ";
   cout<<endl;

   //Constructing vectors vec3a & vec3b with ranges sorted by mod_lesser
   vector<int>vec3a(vec1a), vec3b(vec1b), vec3(vec1);
   vector<int>::iterator Iter3a, Iter3b, Iter3, Result3;
   sort(vec3a.begin(), vec3a.end(), mod_lesser);
   sort(vec3b.begin(), vec3b.end(), mod_lesser);

   cout<<"\nOriginal vector vec3a with range sorted by the\n"
   <<"binary predicate mod_lesser() is: ";
   for(Iter3a = vec3a.begin(); Iter3a != vec3a.end(); Iter3a++)
      cout<<*Iter3a<<" ";
   cout<<endl;

   cout<<"\nOriginal vector vec3b with range sorted by the\n"
   <<"binary predicate mod_lesser() is: ";
   for(Iter3b = vec3b.begin(); Iter3b != vec3b.end(); Iter3b++)
      cout<<*Iter3b<<" ";
   cout<<endl;

   //To combine into an intersection in ascending order with the default
   //binary predicate less <int>()
   Result1 = set_intersection(vec1a.begin(), vec1a.end(),
      vec1b.begin(), vec1b.end(), vec1.begin());
   cout<<"\nset_intersection() of source ranges with default order,\nvector vec1mod: ";
   for(Iter1 = vec1.begin(); Iter1 != Result1; Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //To combine into an intersection in descending order, specify binary
   //predicate greater<int>()
   Result2 = set_intersection(vec2a.begin(), vec2a.end(),
      vec2b.begin(), vec2b.end(), vec2.begin(), greater<int>());
   cout<<"\nset_intersection() of source ranges with binary predicate\ngreater "
   <<"specified, vector vec2mod: ";
   for(Iter2 = vec2.begin(); Iter2 != Result2; Iter2++)
      cout<<*Iter2<<" ";
   cout<<endl;

   //To combine into an intersection applying a user-defined
   //binary predicate mod_lesser
   Result3 = set_intersection(vec3a.begin(), vec3a.end(),
      vec3b.begin(), vec3b.end(), vec3.begin(), mod_lesser);
   cout<<"\nset_intersection() of source ranges with binary predicate\n"
   <<"mod_lesser() specified, vector vec3mod: ";
   for(Iter3 = vec3.begin(); Iter3 != Result3; Iter3++)
      cout<<*Iter3<<" ";
   cout<<endl;
}
```

Output:



```
Original vector vec1a with range sorted by the
binary predicate less than is: -2 -1 0 1 2

Original vector vec1b with range sorted by the
binary predicate less than is: -4 -3 -2 -1 0

Original vector vec2a with range sorted by the
binary predicate greater is: 2 1 0 -1 -2

Original vector vec2b with range sorted by the
binary predicate greater is: 0 -1 -2 -3 -4

Original vector vec3a with range sorted by the
binary predicate mod_lesser() is: 0 -1 1 -2 2

Original vector vec3b with range sorted by the
binary predicate mod_lesser() is: 0 -1 -2 -3 -4

set_intersection() of source ranges with default order,
vector vec1mod: -2 -1 0

set_intersection() of source ranges with binary predicate
greater specified, vector vec2mod: 0 -1 -2

set_intersection() of source ranges with binary predicate
mod_lesser() specified, vector vec3mod: 0 -1 -2
Press any key to continue
```

- Program example compiled using g++.

```
//******algorandshuffle.cpp********
//algorithm, random_shuffle()
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
using namespace std;

int main()
{
   vector <int> vec1;
   vector <int>::iterator Iter1;

   int i;
   for(i = 1; i <= 10; i++)
     vec1.push_back(i);
   cout<<"The original of vector vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;

   //random shuffle…
   random_shuffle(vec1.begin(), vec1.end());
   cout<<"The original of vector vec1 random shuffle data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;

   //Shuffled once
   random_shuffle(vec1.begin(), vec1.end());
   push_heap(vec1.begin(), vec1.end());
   cout<<"Vector vec1 after reshuffle is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;

   //Shuffled again
   random_shuffle(vec1.begin(), vec1.end());
   push_heap(vec1.begin(), vec1.end());
   cout<<"Vector vec1 after another reshuffle is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
   cout<<*Iter1<<" ";
   cout<<endl;
}
```

```
[bodo@bakawali ~]$ g++ algorandshuffle.cpp -o algorandshuffle
[bodo@bakawali ~]$ ./algorandshuffle
```

```
The original of vector vec1 data:
1 2 3 4 5 6 7 8 9 10
The original of vector vec1 random shuffle data:
5 4 8 9 1 6 3 2 7 10
Vector vec1 after reshuffle is:
7 1 8 9 6 4 10 3 2 5
Vector vec1 after another reshuffle is:
10 1 5 4 8 6 3 7 2 9
```

```
//*******algosearchn.cpp*********
//algorithm, search_n()
//some type conversion warning
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

//Return whether second element is twice the first
bool twice(int elem1, int elem2)
{return 2 * elem1 == elem2;}

int main()
{
   vector <int> vec1, vec2;
   vector <int>::iterator Iter1;

   int i;
   for(i = 0; i <= 5; i++)
     vec1.push_back(5*i);

   for(i = 0; i <= 2; i++)
     vec1.push_back(5);

   for(i = 0; i <= 5; i++)
     vec1.push_back(5*i);


   for(i = 0; i <= 2; i++)
     vec1.push_back(5);

   cout<<"Vector vec1 data: ";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Searching vec1 for first match to (5 5 5) under identity
   vector <int>::iterator result1;
   result1 = search_n(vec1.begin(), vec1.end(), 3, 5);

   if(result1 == vec1.end())
      cout<<"\nThere is no match for a sequence (5 5 5) in vec1."<<endl;
   else
      cout<<"\nThere is at least one match of a sequence (5 5 5)"
          <<"\nin vec1 and the first one begins at "
          <<"position "<<result1 - vec1.begin()<<endl;

   //Searching vec1 for first match to (5 5 5) under twice
   vector <int>::iterator result2;
   result2 = search_n(vec1.begin(), vec1.end(), 3, 5);

   if(result2 == vec1.end())
      cout<<"\nThere is no match for a sequence (5 5 5) in vec1"
          <<" under the equivalence predicate twice."<<endl;
   else
      cout<<"\nThere is a match of a sequence (5 5 5) "
          <<"under the equivalence\npredicate twice"
          <<" in vec1 and the first one begins at "
          <<"position "<<result1 - vec1.begin()<<endl;
}
```

```
[bodo@bakawali ~]$ g++ algosearchn.cpp -o algosearchn
[bodo@bakawali ~]$ ./algosearchn
```

```
Vector vec1 data: 0 5 10 15 20 25 5 5 5 0 5 10 15 20 25 5 5 5

There is at least one match of a sequence (5 5 5)
in vec1 and the first one begins at position 6

There is a match of a sequence (5 5 5) under the equivalence
predicate twice in vec1 and the first one begins at position 6
```

---------------------------------------End of Algorithm Part IV-------------------------------

---www.tenouk.com---

Further reading and digging:

1. Check the best selling C / C++ and STL books at Amazon.com.