MODULE 37
--THE STL--
ALGORITHM PART V

My Training Period: hours

Note: Compiled using Microsoft Visual C++ .Net, win32 empty console mode application. **g++** compilation example is 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.

37.1 Continuation from previous Module…

set_symmetric_difference()

- Unites all of the elements that belong to one, but not both, of the 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_symmetric_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class
BinaryPredicate>
   OutputIterator set_symmetric_difference(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result,
      BinaryPredicate _Comp
   );
```

Parameters

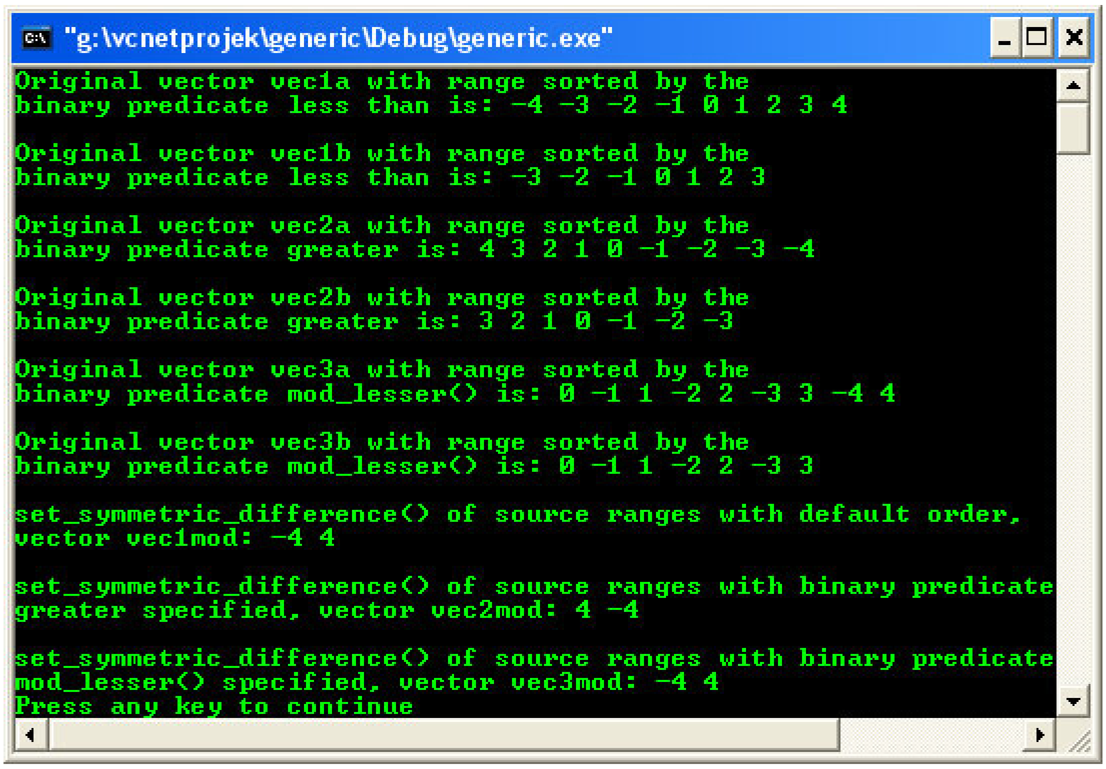
Parameter	Description
_First1	An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.
_Last1	An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the symmetric 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 symmetric 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 symmetric 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 symmetric 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 37.1

- An output iterator addressing the position one past the last element in the sorted destination range representing the symmetric 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 `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 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, then the destination range will contain the absolute value of the number by which the occurrences of those elements in the one of the source ranges exceeds 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_symmetric_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 = -4; i <= 4; i++)
     vec1a.push_back(i);

   int j;
   for(j =-3; j <= 3; 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 symmetric difference in ascending
   //order with the default binary predicate less <int>()
   Result1 = set_symmetric_difference(vec1a.begin(), vec1a.end(),
      vec1b.begin(), vec1b.end(), vec1.begin());
   cout<<"\nset_symmetric_difference() of source ranges with default order,"
   <<"\nvector vec1mod: ";
   for(Iter1 = vec1.begin(); Iter1 != Result1; Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //To combine into a symmetric difference in descending
   //order, specify binary predicate greater<int>()
   Result2 = set_symmetric_difference(vec2a.begin(), vec2a.end(),
      vec2b.begin(), vec2b.end(), vec2.begin(), greater<int>());
   cout<<"\nset_symmetric_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 symmetric difference applying a user
   //defined binary predicate mod_lesser
   Result3 = set_symmetric_difference(vec3a.begin(), vec3a.end(),
      vec3b.begin(), vec3b.end(), vec3.begin(), mod_lesser);
   cout<<"\nset_symmetric_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: -4 -3 -2 -1 0 1 2 3 4

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

Original vector vec2a with range sorted by the
binary predicate greater is: 4 3 2 1 0 -1 -2 -3 -4

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

Original vector vec3a with range sorted by the
binary predicate mod_lesser() is: 0 -1 1 -2 2 -3 3 -4 4

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

set_symmetric_difference() of source ranges with default order,
vector vec1mod: -4 4

set_symmetric_difference() of source ranges with binary predicate
greater specified, vector vec2mod: 4 -4

set_symmetric_difference() of source ranges with binary predicate
mod_lesser() specified, vector vec3mod: -4 4
Press any key to continue
```

set_union()

- Unites all of the elements that belong to at least one of two 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_union(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class
BinaryPredicate>
   OutputIterator set_union(
      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 union 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 union 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 union 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 union 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 union 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.

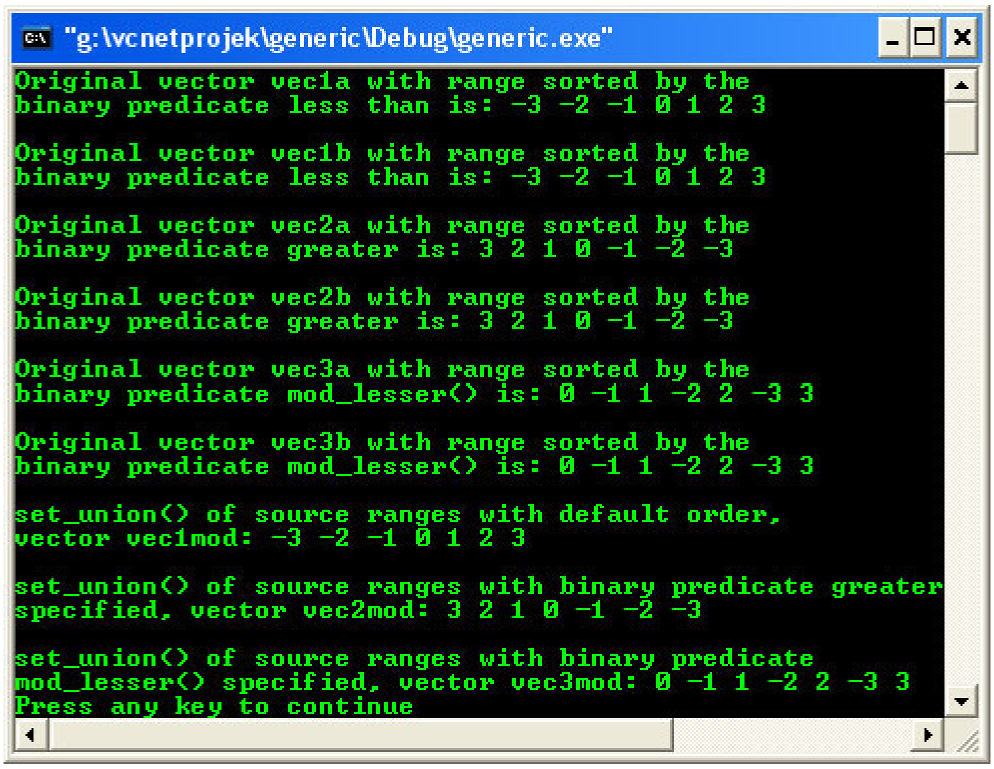
Table 37.2

- The return value is an output iterator addressing the position one past the last element in the sorted destination range representing the union 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 `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 `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, 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.

```
//algorithm, set_union()
#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 = -3; i <= 3; i++)
     vec1a.push_back(i);

   int j;
   for(j =-3; j <= 3; 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 union in ascending order with the default
   //binary predicate less <int>()
   Result1 = set_union(vec1a.begin(), vec1a.end(),
      vec1b.begin(), vec1b.end(), vec1.begin());
   cout<<"\nset_union() of source ranges with default order,"
   <<"\nvector vec1mod: ";
   for(Iter1 = vec1.begin(); Iter1 != Result1; Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

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

   //To combine into a union applying a user-defined
   //binary predicate mod_lesser
   Result3 = set_union(vec3a.begin(), vec3a.end(),
      vec3b.begin(), vec3b.end(), vec3.begin(), mod_lesser);
   cout<<"\nset_union() 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: -3 -2 -1 0 1 2 3

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

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

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

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

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

set_union() of source ranges with default order,
vector vec1mod: -3 -2 -1 0 1 2 3

set_union() of source ranges with binary predicate greater
specified, vector vec2mod: 3 2 1 0 -1 -2 -3

set_union() of source ranges with binary predicate
mod_lesser() specified, vector vec3mod: 0 -1 1 -2 2 -3 3
Press any key to continue
```

sort()

- Arranges the elements in a specified range into a non-descending order or according to an ordering criterion specified by a binary predicate.

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

Parameters

Parameter	Description
_First	A random-access iterator addressing the position of the first element in the range to be sorted.
_Last	A random-access iterator addressing the position one past the final element in the range to be sorted.
_Comp	User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns true when satisfied and false when not satisfied.

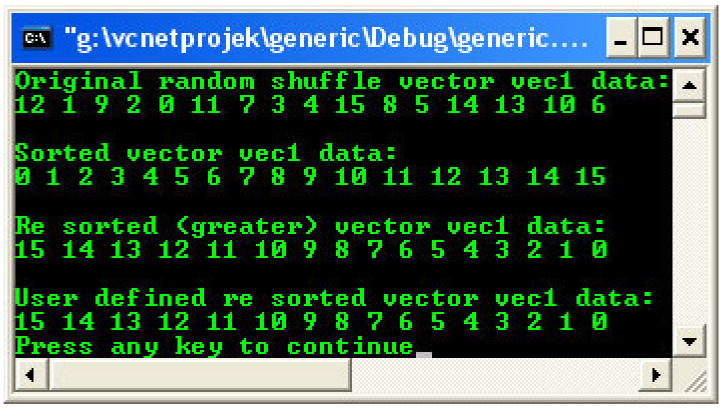
Table 37.3

- 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.
- Elements are equivalent, but not necessarily equal, if neither is less than the other. The `sort()` algorithm is not stable and so does not guarantee that the relative ordering of equivalent elements will be preserved. The algorithm `stable_sort()` does preserve this original ordering.
- The average of a sort complexity is *O*(*N* log *N*), where *N* = *_Last* – *_First*.

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

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

//Return whether first element is greater than the second
bool userdefgreater(int elem1, int elem2)
{return elem1 > elem2;}

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

   int k;
   for(k = 0; k <= 15; k++)
      vec1.push_back(k);

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

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

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

   //To sort in descending order, specify binary predicate
   sort(vec1.begin(), vec1.end(), greater<int>());
   cout<<"\nRe sorted (greater) vector vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //A user-defined binary predicate can also be used
   sort(vec1.begin(), vec1.end(), userdefgreater);
   cout<<"\nUser defined re sorted vector vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;
}
```

Output:



```
Original random shuffle vector vec1 data:
12 1 9 2 0 11 7 3 4 15 8 5 14 13 10 6

Sorted vector vec1 data:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Re sorted (greater) vector vec1 data:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

User defined re sorted vector vec1 data:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Press any key to continue
```

sort_heap()

- Converts a heap into a sorted range.

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

Parameters

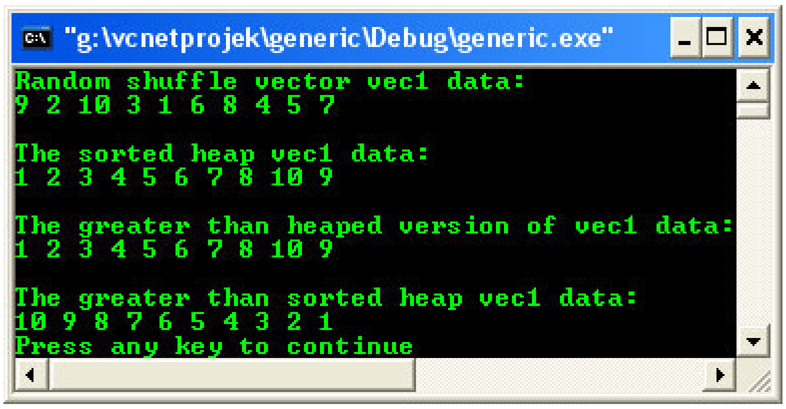
Parameter	Description
`_First`	A random-access iterator addressing the position of the first element in the target heap.
`_Last`	A random-access iterator addressing the position one past the final element in the target heap.
`_Comp`	User-defined predicate function object that defines sense in which one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

Table 37.4

- Heaps have two properties:
  - The first element is always the largest.
  - Elements may be added or removed in logarithmic time.
- After the application if this algorithm, the range it was applied to is no longer a heap.
- This is not a stable sort because the relative order of equivalent elements is not necessarily preserved.
- Heaps are an ideal way to implement priority queues and they are used in the implementation of the Standard Template Library 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 complexity is at most `N log N`, where `N = (_Last – _First)`.

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

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

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

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

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

   //Sort heap vec1 with default less-than ordering
   sort_heap(vec1.begin(), vec1.end());
   cout<<"\nThe sorted heap vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

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

   sort_heap(vec1.begin(), vec1.end(), greater<int>());
   cout<<"\nThe greater than sorted heap vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;
}
```

Output:



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

The sorted heap vec1 data:
1 2 3 4 5 6 7 8 10 9

The greater than heaped version of vec1 data:
1 2 3 4 5 6 7 8 10 9

The greater than sorted heap vec1 data:
10 9 8 7 6 5 4 3 2 1
Press any key to continue
```

stable_partition()

- Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it, preserving the relative order of equivalent elements.

```
template<class BidirectionalIterator, class Predicate>
   BidirectionalIterator stable_partition(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      Predicate _Pred
   );
```

Parameters

Parameter	Description
_First	A bidirectional iterator addressing the position of the first element in the range to be partitioned.
_Last	A bidirectional iterator addressing the position one past the final element in the range to be partitioned.
_Pred	User-defined predicate function object that defines the condition to be satisfied if an element is to be classified. A predicate takes single argument and returns true or false.

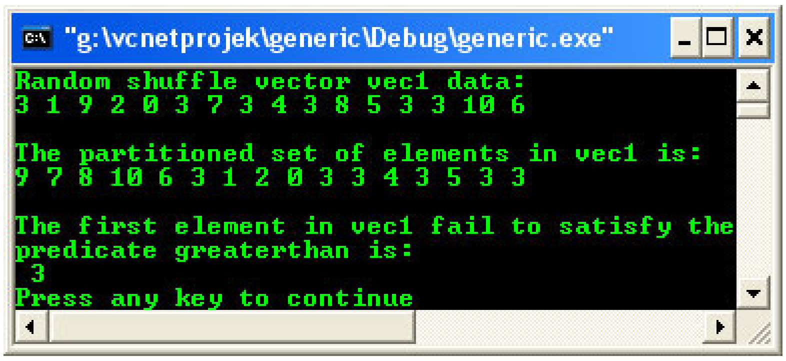
Table 37.5

- The return value is a bidirectional iterator addressing the position of the first element in the range to not satisfy the predicate condition.
- 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.
- Elements *a* and *b* are equivalent, but not necessarily equal, if both *Pr* (*a*, *b*) is false and *Pr* (*b*, *a*) if false, where *Pr* is the parameter-specified predicate. The `stable_ partition()` algorithm is stable and guarantees that the relative ordering of equivalent elements will be preserved.
- The algorithm `partition()` does not necessarily preserve this original ordering.

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

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

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

   int i;
   for(i = 0; i <= 10; i++)
```

```
      vec1.push_back(i);

    int j;
    for(j = 0; j <= 4; j++)
      vec1.push_back(3);

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

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

    //Partition the range with predicate greater than 5...
    result = stable_partition (vec1.begin(), vec1.end(), greaterthan);
    cout<<"\nThe partitioned set of elements in vec1 is:\n";
    for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
       cout<<*Iter1<<" ";
    cout<<endl;

    cout<<"\nThe first element in vec1 fail to satisfy the"
        <<"\npredicate greaterthan is:\n "<<*result<<endl;
}
```

Output:



```
Random shuffle vector vec1 data:
3 1 9 2 0 3 7 3 4 3 8 5 3 3 10 6

The partitioned set of elements in vec1 is:
9 7 8 10 6 3 1 2 0 3 3 4 3 5 3 3

The first element in vec1 fail to satisfy the
predicate greaterthan is:
 3
Press any key to continue
```

stable_sort()

- Arranges the elements in a specified range into a non-descending order or according to an ordering criterion specified by a binary predicate and preserves the relative ordering of equivalent elements.

```
template<class BidirectionalIterator>
   void stable_sort(
      BidirectionalIterator _First,
      BidirectionalIterator _Last
   );
template<class BidirectionalIterator, class BinaryPredicate>
   void stable_sort(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

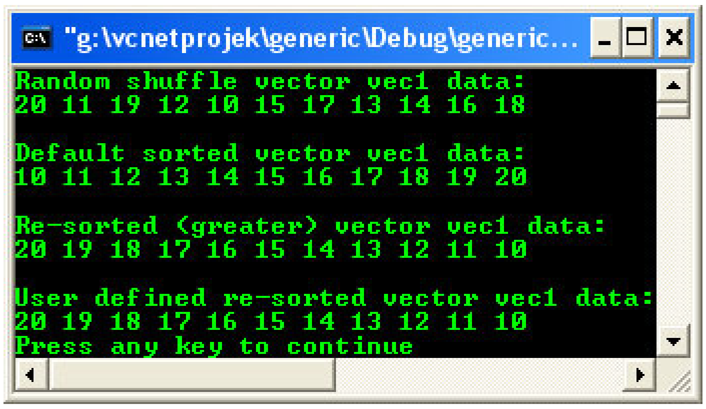
Parameter	Description
_First	A bidirectional iterator addressing the position of the first element in the range to be sorted.
_Last	A bidirectional iterator addressing the position one past the final element in the range to be sorted.
_Comp	User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns true when satisfied and false when not satisfied.

Table 37.6

- 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.
- Elements are equivalent, but not necessarily equal, if neither is less than the other. The `sort()` algorithm is stable and guarantees that the relative ordering of equivalent elements will be preserved.
- The run-time complexity of `stable_sort()` depends on the amount of memory available, but the best case (given sufficient memory) is $O(N \log N)$ and the worst case is $O(N(\log N)^2)$, where $N = \_Last-First$. Usually, the `sort()` algorithm is significantly faster than `stable_sort()`.

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

//Return whether first element is greater than the second
bool userdefgreater(int elem1, int elem2)
{ return elem1 > elem2;}

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

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

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

   sort(vec1.begin(), vec1.end());
   cout<<"\nDefault sorted vector vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //To sort in descending order, specify binary predicate
   sort(vec1.begin(), vec1.end(), greater<int>());
   cout<<"\nRe-sorted (greater) vector vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //A user-defined binary predicate can also be used
   sort(vec1.begin(), vec1.end(), userdefgreater);
   cout<<"\nUser defined re-sorted vector vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;
}
```

Output:



```
Random shuffle vector vec1 data:
20 11 19 12 10 15 17 13 14 16 18

Default sorted vector vec1 data:
10 11 12 13 14 15 16 17 18 19 20

Re-sorted (greater) vector vec1 data:
20 19 18 17 16 15 14 13 12 11 10

User defined re-sorted vector vec1 data:
20 19 18 17 16 15 14 13 12 11 10
Press any key to continue
```

swap()

- Exchanges the values of the elements between two types of objects, assigning the contents of the first object to the second object and the contents of the second to the first.

```
template<class Type>
   void swap(
      Type& _Left,
      Type& _Right
   );
```

Parameters

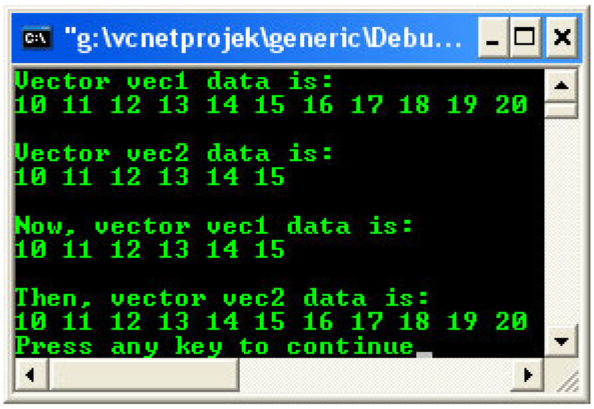
Parameter	Description
_Left	The first object to have its elements exchanged.
_Right	The second object to have its elements exchanged.

Table 37.7

- This algorithm is exceptional in the STL in being designed to operate on individual elements rather than on a range of elements.

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

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

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

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

   int j;
   for(j = 10; j <= 15; j++)
     vec2.push_back(j);

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

   cout<<"\nVector vec2 data is:\n";
   for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
      cout<<*Iter2<<" ";
   cout<<endl;

   swap(vec1, vec2);

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

   cout<<"\nThen, vector vec2 data is:\n";
   for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
      cout<<*Iter2<<" ";
   cout<<endl;
}
```

Output:



```
Vector vec1 data is:
10 11 12 13 14 15 16 17 18 19 20

Vector vec2 data is:
10 11 12 13 14 15

Now, vector vec1 data is:
10 11 12 13 14 15

Then, vector vec2 data is:
10 11 12 13 14 15 16 17 18 19 20
Press any key to continue
```

swap_ranges()

- Exchanges the elements of one range with the elements of another, equal sized range.

```
template<class ForwardIterator1, class ForwardIterator2>
   ForwardIterator2 swap_ranges(
      ForwardIterator1 _First1,
      ForwardIterator1 _Last1,
      ForwardIterator2 _First2
   );
```

Parameters

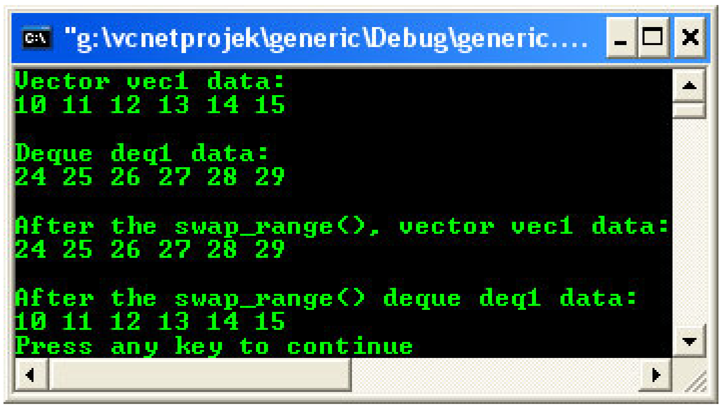
Parameter	Description
_First1	A forward iterator pointing to the first position of the first range whose elements are to be exchanged.
_Last1	A forward iterator pointing to one past the final position of the first range whose elements are to be exchanged.
_First2	A forward iterator pointing to the first position of the second range whose elements are to be exchanged.

Table 37.8

- The return value is a forward iterator pointing to one past the final position of the second range whose elements are to be exchanged.
- 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. The second range has to be as large as the first range.
- The complexity is linear with *_Last1 – _First1* swaps performed. If elements from containers of the same type are being swapped, them the `swap()` member function from that container should be used, because the member function typically has constant complexity.

```
//algorithm, swap_ranges()
#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 deq1Iter;

   int i;
   for(i = 10; i <= 15; i++)
     vec1.push_back(i);

   int j;
   for(j =24; j <= 29; j++)
     deq1.push_back(j);

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

   cout<<"\nDeque deq1 data:\n";
   for(deq1Iter = deq1.begin(); deq1Iter != deq1.end(); deq1Iter++)
      cout<<*deq1Iter<<" ";
   cout<<endl;

   swap_ranges(vec1.begin(), vec1.end(), deq1.begin());

   cout<<"\nAfter the swap_range(), vector vec1 data:\n";
   for(vec1Iter1 = vec1.begin(); vec1Iter1 != vec1.end(); vec1Iter1++)
      cout<<*vec1Iter1<<" ";
   cout<<endl;

   cout<<"\nAfter the swap_range() deque deq1 data:\n";
   for(deq1Iter = deq1.begin(); deq1Iter != deq1.end(); deq1Iter++)
      cout<<*deq1Iter<<" ";
   cout<<endl;
}
```

Output:



```
Vector vec1 data:
10 11 12 13 14 15

Deque deq1 data:
24 25 26 27 28 29

After the swap_range(), vector vec1 data:
24 25 26 27 28 29

After the swap_range() deque deq1 data:
10 11 12 13 14 15
Press any key to continue
```

transform()

- Applies a specified function object to each element in a source range or to a pair of elements from two source ranges and copies the return values of the function object into a destination range.

```
template<class InputIterator, class OutputIterator, class UnaryFunction>
   OutputIterator transform(
      InputIterator _First1,
      InputIterator _Last1,
      OutputIterator _Result,
      UnaryFunction _Func
   );
template<class InputIterator1, class InputIterator2, class OutputIterator,
   class BinaryFunction>
   OutputIterator transform(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      OutputIterator _Result,
      BinaryFunction _Func
   );
```

Parameters

Parameter	Description
_First1	An input iterator addressing the position of the first element in the first source range to be operated on.
_Last1	An input iterator addressing the position one past the final element in the first source range operated on.
_First2	An input iterator addressing the position of the first element in the second source range to be operated on.
_Result	An output iterator addressing the position of the first element in the

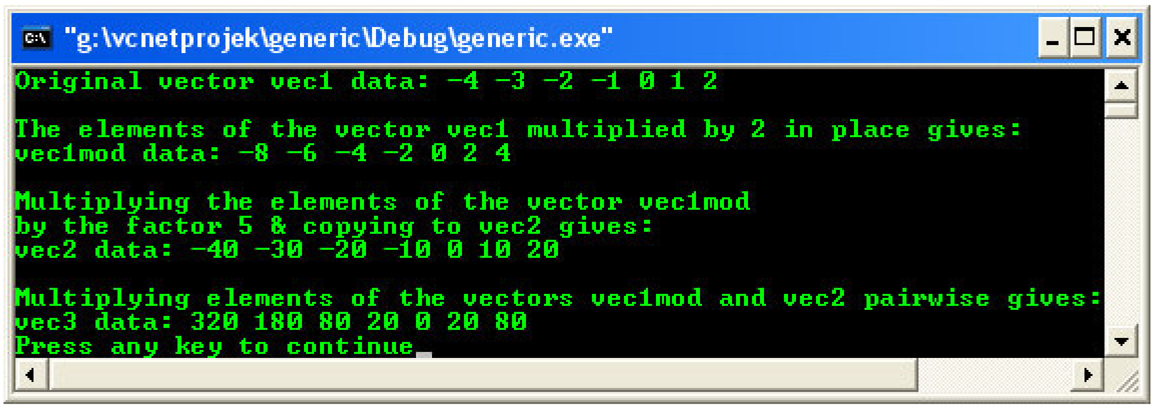
	destination range.
`_Func`	User-defined unary function object used in the first version of the algorithm that is applied to each element in the first source range or A user-defined binary function object used in the second version of the algorithm that is applied pairwise, in a forward order, to the two source ranges.

Table 37.9

- The return value is an output iterator addressing the position one past the final element in the destination range that is receiving the output elements transformed by the function object.
- The 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 must be large enough to contain the transformed source range.
- If `_Result` is set equal to `_First1` in the first version of the algorithm, then the source and destination ranges will be the same and the sequence will be modified in place. But the `_Result` may not address a position within the range `[_First1 +1, _Last1)`.
- The complexity is linear with at most `(_Last1 – _First1)` comparisons.

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

//The function object multiplies an element by a Factor
template <class Type>
class MultValue
{
   private:
      //The value to multiply by
      Type Factor;
   public:
      //Constructor initializes the value to multiply by
      MultValue(const Type& _Val) : Factor(_Val) {}

      //The function call for the element to be multiplied
      int operator()(Type& elem) const
      {return (elem * Factor);}
};

int main()
{
   vector <int> vec1, vec2(7), vec3(7);
   vector <int>::iterator Iter1, Iter2, Iter3;

   //Constructing vector vec1
   for(int i = -4; i <= 2; i++)
     vec1.push_back(i);

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

   //Modifying the vector vec1 in place
   transform(vec1.begin(), vec1.end(), vec1.begin(), MultValue<int>(2));
   cout<<"\nThe elements of the vector vec1 multiplied by 2 in place gives:"
   <<"\nvec1mod data: ";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Using transform() to multiply each element by a factor of 5
   transform(vec1.begin(), vec1.end(), vec2.begin(), MultValue<int>(5));

   cout<<"\nMultiplying the elements of the vector vec1mod\n"
   <<"by the factor 5 & copying to vec2 gives:\nvec2 data: ";
   for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
      cout<<*Iter2<<" ";
   cout<<endl;

   //The second version of transform used to multiply the
   //elements of the vectors vec1mod & vec2 pairwise
   transform(vec1.begin(), vec1.end(), vec2.begin(), vec3.begin(),
      multiplies<int>());

   cout<<"\nMultiplying elements of the vectors vec1mod and vec2 pairwise "
   <<"gives:\nvec3 data: ";
   for(Iter3 = vec3.begin(); Iter3 != vec3.end(); Iter3++)
      cout<<*Iter3<<" ";
   cout<<endl;
}
```

Output:



```
Original vector vec1 data: -4 -3 -2 -1 0 1 2

The elements of the vector vec1 multiplied by 2 in place gives:
vec1mod data: -8 -6 -4 -2 0 2 4

Multiplying the elements of the vector vec1mod
by the factor 5 & copying to vec2 gives:
vec2 data: -40 -30 -20 -10 0 10 20

Multiplying elements of the vectors vec1mod and vec2 pairwise gives:
vec3 data: 320 180 80 20 0 20 80
Press any key to continue
```

unique()

- Removes duplicate elements that are adjacent to each other in a specified range.

```
template<class ForwardIterator>
   ForwardIterator unique(
      ForwardIterator _First,
      ForwardIterator _Last
   );
template<class ForwardIterator, class Pr>
   ForwardIterator unique(
      ForwardIterator _First,
      ForwardIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

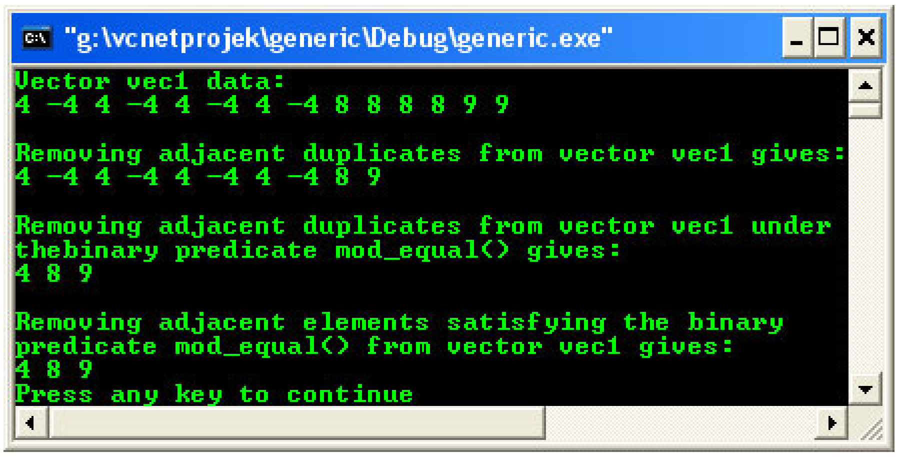
Parameter	Description
_First	A forward iterator addressing the position of the first element in the range to be scanned for duplicate removal.
_Last	A forward iterator addressing the position one past the final element in the range to be scanned for duplicate removal.
_Comp	User-defined predicate function 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 37.10

- The return value is a forward iterator to the new end of the modified sequence that contains no consecutive duplicates, addressing the position one past the last element not removed.
- Both forms of the algorithm remove the second duplicate of a consecutive pair of equal elements.
- The operation of the algorithm is stable so that the relative order of the undeleted elements is not changed.
- 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 number of elements in the sequence is not changed by the algorithm `unique()` and the elements beyond the end of the modified sequence are de-referenceable but not specified.
- The complexity is linear, requiring (*_Last* – *_First*)–1 comparisons.
- List provides a more efficient member function `unique()`, which may perform better.
- These algorithms cannot be used on an associative container.

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

//Return whether modulus of elem1 is equal to modulus of elem2
bool mod_equal(int elem1, int elem2)
{
   if(elem1 < 0)
      elem1 = - elem1;
   if(elem2 < 0)
      elem2 = - elem2;
   return (elem1 == elem2);
};

int main()
{
   vector <int> vec1;
   vector <int>::iterator vec1_Iter1, vec1_Iter2, vec1_Iter3,
         vec1_NewEnd1, vec1_NewEnd2, vec1_NewEnd3;

   int i;
   for(i = 0; i <= 3; i++)
   {
      vec1.push_back(4);
      vec1.push_back(-4);
   }

   int j;
   for(j = 1; j <= 4; j++)
     vec1.push_back(8);

   vec1.push_back(9);
   vec1.push_back(9);

   cout<<"Vector vec1 data:\n";
   for(vec1_Iter1 = vec1.begin(); vec1_Iter1 != vec1.end(); vec1_Iter1++)
      cout<<*vec1_Iter1<<" ";
   cout<<endl;

   //Remove consecutive duplicates
   vec1_NewEnd1 = unique(vec1.begin(), vec1.end());

   cout<<"\nRemoving adjacent duplicates from vector vec1 gives:\n";
   for(vec1_Iter1 = vec1.begin(); vec1_Iter1 != vec1_NewEnd1; vec1_Iter1++)
      cout<<*vec1_Iter1<<" ";
   cout<<endl;

   //Remove consecutive duplicates under the binary predicate mod_equal()
   vec1_NewEnd2 = unique(vec1.begin(), vec1_NewEnd1, mod_equal);

   cout<<"\nRemoving adjacent duplicates from vector vec1 under\nthe"
   <<"binary predicate mod_equal() gives:\n";
   for(vec1_Iter2 = vec1.begin(); vec1_Iter2 != vec1_NewEnd2; vec1_Iter2++)
      cout<<*vec1_Iter2<<" ";
   cout<<endl;

   //Remove elements if preceded by an element that was greater
   vec1_NewEnd3 = unique(vec1.begin(), vec1_NewEnd2, greater<int>());

   cout<<"\nRemoving adjacent elements satisfying the binary\n"
   <<"predicate mod_equal() from vector vec1 gives:\n";
   for(vec1_Iter3 = vec1.begin(); vec1_Iter3 != vec1_NewEnd3; vec1_Iter3++)
      cout<<*vec1_Iter3<<" ";
   cout<<endl;
}
```

Output:



```
Vector vec1 data:
4 -4 4 -4 4 -4 4 -4 8 8 8 8 8 9 9

Removing adjacent duplicates from vector vec1 gives:
4 -4 4 -4 4 -4 4 -4 8 9

Removing adjacent duplicates from vector vec1 under
thebinary predicate mod_equal() gives:
4 8 9

Removing adjacent elements satisfying the binary
predicate mod_equal() from vector vec1 gives:
4 8 9
Press any key to continue
```

unique_copy()

- Copies elements from a source range into a destination range except for the duplicate elements that are adjacent to each other.

```
template<class InputIterator, class OutputIterator>
   OutputIterator unique_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result
   );
template<class InputIterator, class OutputIterator, class BinaryPredicate>
   OutputIterator unique_copy(
      InputIterator _First,
      InputIterator _Last,
      OutputIterator _Result,
      BinaryPredicate _Comp,
   );
```

Parameters

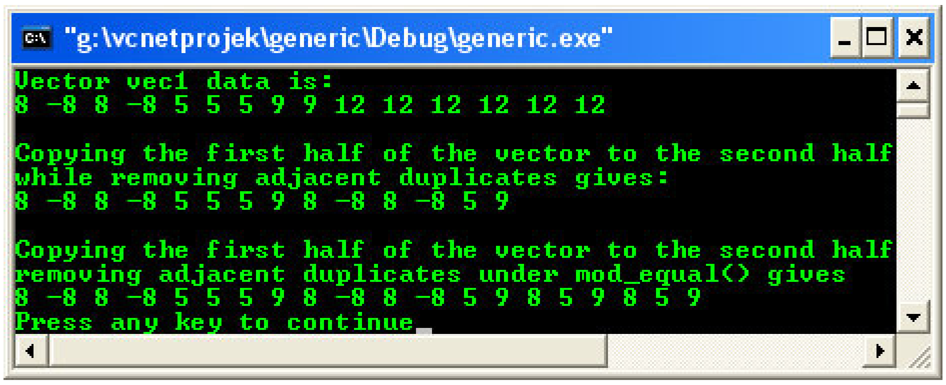
Parameter	Description
_First	A forward iterator addressing the position of the first element in the source range to be copied.
_Last	A forward iterator addressing the position one past the final element in the source range to be copied.
_Result	An output iterator addressing the position of the first element in the destination range that is receiving the copy with consecutive duplicates removed.
_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 37.11

- The return value is an output iterator addressing the position one past the final element in the destination range that is receiving the copy with consecutive duplicates removed.
- Both forms of the algorithm remove the second duplicate of a consecutive pair of equal elements.
- The operation of the algorithm is stable so that the relative order of the undeleted elements is not changed.
- The ranges referenced must be valid; all pointers must be de-referenceable and within a sequence the last position is reachable from the first by incrementation.
- The complexity is linear, requiring (*_Last - _First*) comparisons.

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

//Return whether modulus of elem1 is equal to modulus of elem2
bool mod_equal(int elem1, int elem2)
{
   if(elem1 < 0)
      elem1 = - elem1;
   if(elem2 < 0)
      elem2 = - elem2;
   return (elem1 == elem2);
};

int main()
{
   vector <int> vec1;
   vector <int>::iterator vec1_Iter1, vec1_Iter2,
         vec1_NewEnd1, vec1_NewEnd2;

   int i;
   for(i = 0; i <= 1; i++)
   {
      vec1.push_back(8);
      vec1.push_back(-8);
   }

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

       vec1.push_back(9);
       vec1.push_back(9);

   int k;
   for(k = 0; k <= 5; k++)
     vec1.push_back(12);

   cout<<"Vector vec1 data is:\n";
   for(vec1_Iter1 = vec1.begin(); vec1_Iter1 != vec1.end(); vec1_Iter1++)
      cout<<*vec1_Iter1<<" ";
   cout<<endl;

   //Copy first half to second, removing consecutive duplicates
   vec1_NewEnd1 = unique_copy(vec1.begin(), vec1.begin() + 8, vec1.begin() + 8);

   cout<<"\nCopying the first half of the vector to the second half\n"
          <<"while removing adjacent duplicates gives:\n";
   for(vec1_Iter1 = vec1.begin(); vec1_Iter1 != vec1_NewEnd1; vec1_Iter1++)
      cout<<*vec1_Iter1<<" ";
   cout<<endl;

   for(int l = 0; l <= 7; l++)
     vec1.push_back(10);

   //Remove consecutive duplicates under the binary predicate mod_equals
   vec1_NewEnd2 = unique_copy(vec1.begin(), vec1.begin() + 14, vec1.begin() + 14,
mod_equal);

   cout<<"\nCopying the first half of the vector to the second half\n"
       <<"removing adjacent duplicates under mod_equal() gives\n";
   for(vec1_Iter2 = vec1.begin(); vec1_Iter2 != vec1_NewEnd2; vec1_Iter2++)
      cout<<*vec1_Iter2<<" ";
   cout<<endl;
}
```

Output:



```
Vector vec1 data is:
8 -8 8 -8 5 5 5 9 9 12 12 12 12 12 12

Copying the first half of the vector to the second half
while removing adjacent duplicates gives:
8 -8 8 -8 5 5 5 9 8 -8 8 -8 5 9

Copying the first half of the vector to the second half
removing adjacent duplicates under mod_equal() gives
8 -8 8 -8 5 5 5 9 8 -8 8 -8 5 9 8 5 9 8 5 9
Press any key to continue
```

upper_bound()

- Finds the position of the first element in an ordered range that has a value that is greater than a specified value, where the ordering criterion may be specified by a binary predicate.

```
template<class ForwardIterator, class Type>
   ForwardIterator upper_bound(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val
   );
template<class ForwardIterator, class Type, class Pr>
   ForwardIterator upper_bound(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val,
      BinaryPredicate _Comp
   );
```

Parameters

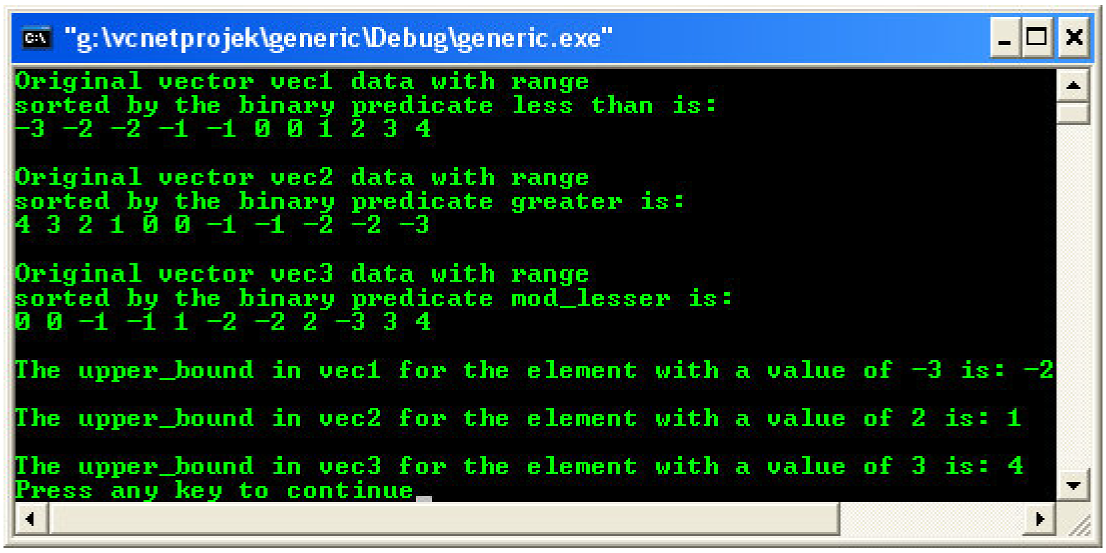
Parameter	Description
`_First`	The position of the first element in the range to be searched.
`_Last`	The position one past the final element in the range to be searched.
`_Val`	The value in the ordered range that needs to be exceeded by the value of the element addressed by the iterator returned.
`_Comp`	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 37.12

- The return value is a forward iterator addressing the position of the first element in an ordered range that has a value that is greater than a specified value, where the ordering criterion may be specified by a binary predicate.
- The sorted source range referenced must be valid; all pointers must be de-referenceable and within the sequence the last position must be reachable from the first by incrementation.
- The sorted range must each be arranged as a precondition to the application of the `upper_bound()` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.
- The range is not modified by the algorithm `merge()`.
- The value types of the forward 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
- The complexity of the algorithm is logarithmic for random-access iterators and linear otherwise, with the number of steps proportional to (`_Last1 – _First1`).

```
//algorithm, upper_bound()
#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> vec1;
   vector <int>::iterator Iter1, Result1;

   //Constructing vectors vec1a & vec1b with default less-than ordering
    for(int i = -2; i <= 4; i++)
     vec1.push_back(i);
   for(int j = -3; j <= 0; j++)
     vec1.push_back(j);

   sort(vec1.begin(), vec1.end());
   cout<<"Original vector vec1 data with range\nsorted by the"
   <<" binary predicate less than is:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //Constructing vectors vec2 with range sorted by greater
   vector <int> vec2(vec1);
   vector <int>::iterator Iter2, Result2;
   sort(vec2.begin(), vec2.end(), greater<int>());

   cout<<"\nOriginal vector vec2 data with range\nsorted by the"
   <<" binary predicate greater is:\n";
   for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
      cout<<*Iter2<<" ";
   cout<<endl;

   //Constructing vectors vec3 with range sorted by mod_lesser
   vector <int>vec3(vec1);
   vector <int>::iterator Iter3, Result3;
   sort(vec3.begin(), vec3.end(), mod_lesser);

   cout<<"\nOriginal vector vec3 data with range\nsorted by the"
   <<" binary predicate mod_lesser is:\n";
   for(Iter3 = vec3.begin(); Iter3 != vec3.end(); Iter3++)
      cout<<*Iter3<<" ";
   cout<<endl;

   //upper_bound of -3 in vec1 with default binary predicate less <int>()
   Result1 = upper_bound(vec1.begin(), vec1.end(), -3);
   cout<<"\nThe upper_bound in vec1 for the element with a value of -3 is: "
       <<*Result1<<endl;

   //upper_bound of 2 in vec2 with the binary predicate greater <int>()
   Result2 = upper_bound(vec2.begin(), vec2.end(), 2, greater<int>());
   cout<<"\nThe upper_bound in vec2 for the element with a value of 2 is: "
       <<*Result2<<endl;

   //upper_bound of 3 in vec3 with the binary predicate mod_lesser
   Result3 = upper_bound(vec3.begin(), vec3.end(), 3, mod_lesser);
   cout<<"\nThe upper_bound in vec3 for the element with a value of 3 is: "
       <<*Result3<<endl;
}
```

Output:



```
Original vector vec1 data with range
sorted by the binary predicate less than is:
-3 -2 -2 -1 -1 0 0 1 2 3 4

Original vector vec2 data with range
sorted by the binary predicate greater is:
4 3 2 1 0 0 -1 -1 -2 -2 -3

Original vector vec3 data with range
sorted by the binary predicate mod_lesser is:
0 0 -1 -1 1 -2 -2 2 -3 3 4

The upper_bound in vec1 for the element with a value of -3 is: -2

The upper_bound in vec2 for the element with a value of 2 is: 1

The upper_bound in vec3 for the element with a value of 3 is: 4
Press any key to continue
```

- Program example compiled using **g++**.

```
//******algosort.cpp*********
//algorithm, sort()
```

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

//Return whether first element is greater than the second
bool userdefgreater(int elem1, int elem2)
{return elem1 > elem2;}

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

   int k;
   for(k = 0; k <= 15; k++)
      vec1.push_back(k);

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

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

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

   //To sort in descending order, specify binary predicate
   sort(vec1.begin(), vec1.end(), greater<int>());
   cout<<"\nRe sorted (greater) vector vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;

   //A user-defined binary predicate can also be used
   sort(vec1.begin(), vec1.end(), userdefgreater);
   cout<<"\nUser defined re sorted vector vec1 data:\n";
   for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
      cout<<*Iter1<<" ";
   cout<<endl;
}
```

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

```
Original random shuffle vector vec1 data:
4 10 11 15 14 5 13 1 6 9 3 7 8 2 0 12

Sorted vector vec1 data:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Re sorted (greater) vector vec1 data:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

User defined re sorted vector vec1 data:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

-----------------------------------End of Algorithm Part V---------------------------

---www.tenouk.com---

Further reading and digging:

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