MODULE 35
--THE STL--
ALGORITHM PART III

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.

35.1 Continuation from the previous Module…

inplace_merge()

- Combines the elements from two consecutive sorted ranges into a single sorted range, where the ordering criterion may be specified by a binary predicate.

```
template<class BidirectionalIterator>
   void inplace_merge(
      BidirectionalIterator _First,
      BidirectionalIterator _Middle,
      BidirectionalIterator _Last
   );
template<class BidirectionalIterator, class Pr>
   void inplace_merge(
      BidirectionalIterator _First,
      BidirectionalIterator _Middle,
      BidirectionalIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

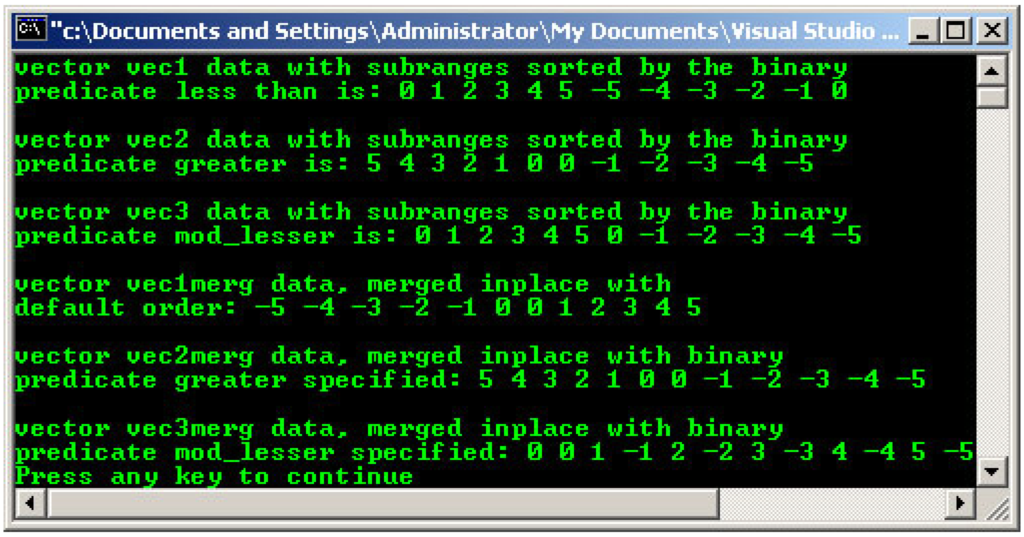
Parameter	Description
_First	A bidirectional iterator addressing the position of the first element in the first of two consecutive sorted ranges to be combined and sorted into a single range.
_Middle	A bidirectional iterator addressing the position of the first element in the second of two consecutive sorted ranges to be combined and sorted into a single range.
_Last	A bidirectional iterator addressing the position one past the last element in the second of two consecutive sorted ranges to be combined and sorted into a single range.
_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 35.1

- The sorted consecutive 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 sorted consecutive ranges must each be arranged as a precondition to the application of the `inplace_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. When there are equivalent elements in both source ranges, the element is the first range precedes the element from the second in the combined range.
- The complexity depends on the available memory as the algorithm allocates memory to a temporary buffer. If sufficient memory is available, the best case is linear with `(_Last – _First)–1` comparisons; if no auxiliary memory is available, the worst case is `N log(N), where N = (_Last–_First)`.

```
//algorithm, inplace_merge()
#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, Iter2, Iter3;

//Constructing vector vec1 with default less-than ordering
int i;
for(i = 0; i <= 5; i++)
vec1.push_back(i);

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

cout<<"vector vec1 data with subranges sorted by the "<<"binary\npredicate less
than is: ";
for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

//Constructing vector vec2 with ranges sorted by greater
vector <int> vec2(vec1);
vector <int>::iterator break2;
break2 = find(vec2.begin(), vec2.end(), -5);

sort(vec2.begin(), break2, greater<int>());
sort(break2, vec2.end(), greater<int>());

cout<<"\nvector vec2 data with subranges sorted by the "<<"binary\npredicate
greater is: ";
for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
cout<<*Iter2<<" ";
cout<<endl;

//Constructing vector vec3 with ranges sorted by mod_lesser
vector <int> vec3(vec1);
vector <int>::iterator break3;
break3 = find(vec3.begin(), vec3.end(), -5);

sort(vec3.begin(), break3, mod_lesser);
sort(break3, vec3.end(), mod_lesser);

cout<<"\nvector vec3 data with subranges sorted by the "<<"binary\npredicate
mod_lesser is: ";
for(Iter3 = vec3.begin(); Iter3 != vec3.end(); Iter3++)
cout<<*Iter3<<" ";
cout<<endl;

vector <int>::iterator break1;
break1 = find(vec1.begin(), vec1.end(), -5);
inplace_merge(vec1.begin(), break1, vec1.end());
cout<<"\nvector vec1merg data, merged inplace with\ndefault order: ";
for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

//To merge inplace in descending order, specify binary
//predicate greater<int>()
inplace_merge(vec2.begin(), break2, vec2.end(), greater<int>());
cout<<"\nvector vec2merg data, merged inplace with binary\npredicate greater
specified: ";
for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
cout<<*Iter2<<" ";
cout<<endl;

//Applying a user defined binary predicate mod_lesser
inplace_merge(vec3.begin(), break3, vec3.end(), mod_lesser);
cout<<"\nvector vec3merg data, merged inplace with binary\npredicate mod_lesser
specified: ";
for(Iter3 = vec3.begin(); Iter3 != vec3.end(); Iter3++)
cout<<*Iter3<<" ";
cout<<endl;
return 0;
}
```

Output:


"c:\Documents and Settings\Administrator\My Documents\Visual Studio ...

```
vector vec1 data with subranges sorted by the binary
predicate less than is: 0 1 2 3 4 5 -5 -4 -3 -2 -1 0

vector vec2 data with subranges sorted by the binary
predicate greater is: 5 4 3 2 1 0 0 -1 -2 -3 -4 -5

vector vec3 data with subranges sorted by the binary
predicate mod_lesser is: 0 1 2 3 4 5 0 -1 -2 -3 -4 -5

vector vec1merg data, merged inplace with
default order: -5 -4 -3 -2 -1 0 0 1 2 3 4 5

vector vec2merg data, merged inplace with binary
predicate greater specified: 5 4 3 2 1 0 0 -1 -2 -3 -4 -5

vector vec3merg data, merged inplace with binary
predicate mod_lesser specified: 0 0 1 -1 2 -2 3 -3 4 -4 5 -5
Press any key to continue
```

iter_swap()

- Exchanges two values referred to by a pair of specified iterators.

```
template<class ForwardIterator1, class ForwardIterator2>
   void iter_swap(
      ForwardIterator1 _Left,
      ForwardIterator2 _Right
   );
```

Parameters

Parameter	Description
_Left	One of the forward iterators whose value is to be exchanged.
_Right	The second of the forward iterators whose value is to be exchanged.

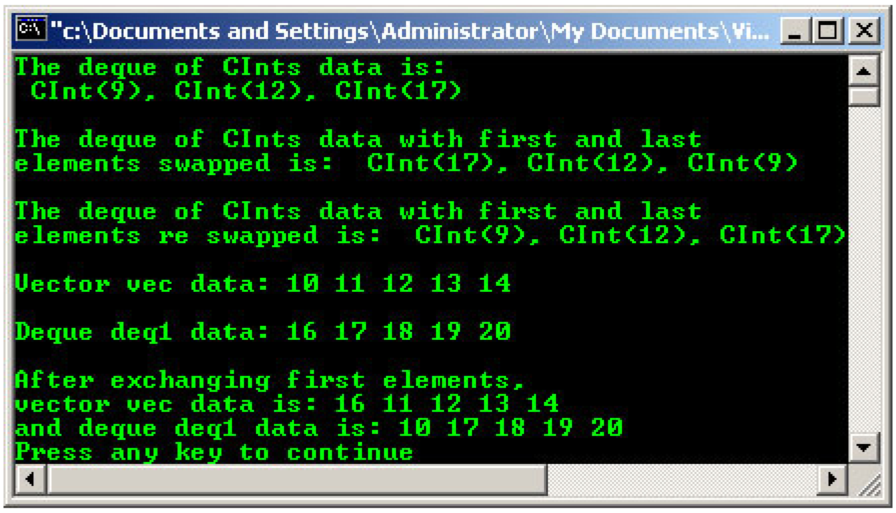
Table 35.2

- `swap()` should be used in preference to `iter_swap()`, which was included in the C++ Standard for backward compatibility. If *Fit1* and *Fit2* are forward iterators, then `iter_swap(`*Fit1*`, `*Fit2*`)`, is equivalent to `swap(*`*Fit1*`, *`*Fit2*`)`.
- The value types of the input forward iterators must have the same value.

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

class CInt;
ostream& operator<<(ostream& osIn, const CInt& rhs);

class CInt
{
public:
CInt(int n = 0) : m_nVal(n){}
CInt(const CInt& rhs) : m_nVal(rhs.m_nVal){}
CInt& operator=(const CInt& rhs)
{ m_nVal = rhs.m_nVal; return *this;}
bool operator<(const CInt& rhs) const
{ return (m_nVal < rhs.m_nVal);}
friend ostream& operator<<(ostream& osIn, const CInt& rhs);

private:
int m_nVal;
};

inline ostream& operator<<(ostream& osIn, const CInt& rhs)
{
osIn<<"CInt(" <<rhs.m_nVal<< ")";
return osIn;
}

//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()
{
CInt c1 = 9, c2 = 12, c3 = 17;
deque<CInt> deq;
deque<CInt>::iterator deqIter;

deq.push_back(c1);
deq.push_back(c2);
deq.push_back(c3);

cout<<"The deque of CInts data is:\n";
for(deqIter = deq.begin(); deqIter != --deq.end(); deqIter++)
cout<<" "<<*deqIter<<",";
deqIter = --deq.end();
cout<<" "<<*deqIter<<endl;

//Exchanging first and last elements with iter_swap
iter_swap(deq.begin(), --deq.end());
cout<<"\nThe deque of CInts data with first and last\nelements swapped is: ";
for(deqIter = deq.begin(); deqIter != --deq.end(); deqIter++)
cout<<" "<<*deqIter<<",";
deqIter = --deq.end();
cout<<" "<<*deqIter<<endl;

//Swapping back first and last elements with swap
swap(*deq.begin(), *(deq.end() -1));

cout<<"\nThe deque of CInts data with first and last\nelements re swapped is: ";
for(deqIter = deq.begin(); deqIter != --deq.end(); deqIter++)
cout<<" "<<*deqIter<<",";
deqIter = --deq.end();
cout<<" "<<*deqIter<<endl;

//Swapping a vector element with a deque element
vector <int> vec;
vector <int>::iterator Iter1;
deque <int> deq1;
deque <int>::iterator deq1Iter;

int i;
for(i = 10; i <= 14; i++)
vec.push_back(i);

int j;
for(j = 16; j <= 20; j++)
deq1.push_back(j);

cout<<"\nVector vec data: ";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

cout<<"\nDeque deq1 data: ";
```

```
for(deq1Iter = deq1.begin(); deq1Iter != deq1.end(); deq1Iter++)
cout<<*deq1Iter<<" ";
cout<<endl;

iter_swap(vec.begin(), deq1.begin());
cout<<"\nAfter exchanging first elements,\nvector vec data is: ";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl<<"and deque deq1 data is: ";
for(deq1Iter = deq1.begin(); deq1Iter != deq1.end(); deq1Iter++)
cout<<*deq1Iter<<" ";
cout<<endl;
return 0;
}
```

Output:



```
The deque of CInts data is:
 CInt(9), CInt(12), CInt(17)

The deque of CInts data with first and last
elements swapped is:  CInt(17), CInt(12), CInt(9)

The deque of CInts data with first and last
elements re swapped is:  CInt(9), CInt(12), CInt(17)

Vector vec data: 10 11 12 13 14

Deque deq1 data: 16 17 18 19 20

After exchanging first elements,
vector vec data is: 16 11 12 13 14
and deque deq1 data is: 10 17 18 19 20
Press any key to continue
```

lexicographical_compare()

- Compares element by element between two sequences to determine which is lesser of the two.

```
template<class InputIterator1, class InputIterator2>
   bool lexicographical_compare(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2
   );
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
   bool lexicographical_compare(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      BinaryPredicate _Comp
   );
```

Parameters

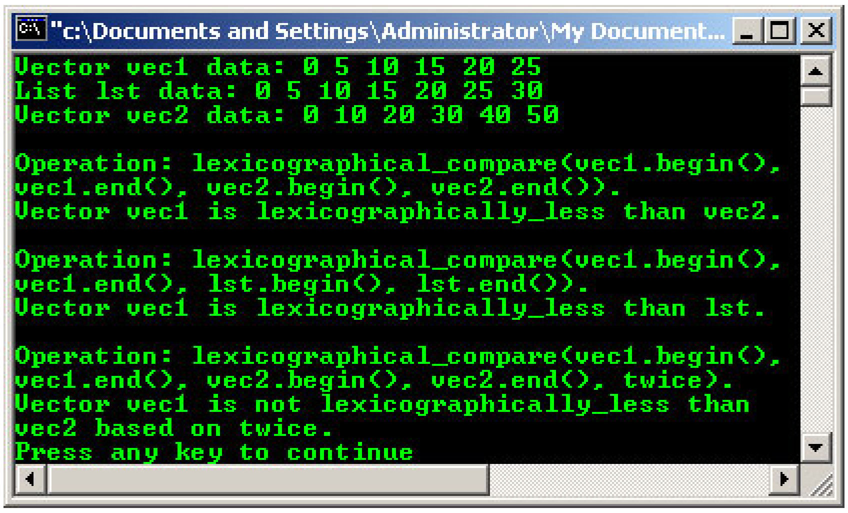
Parameter	Description
_First1	An input iterator addressing the position of the first element in the first range to be compared.
_Last1	An input iterator addressing the position one past the final element in the first range to be compared.
_First2	An input iterator addressing the position of the first element in the second range to be compared.
_Last2	An input iterator addressing the position one past the final element in the second range to be compared.
_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 35.3

- The return value is true if the first range is lexicographically less than the second range; otherwise false.
- A lexicographical comparison between sequences compares them element by element until:
    - It finds two corresponding elements unequal, and the result of their comparison is taken as the result of the comparison between sequences.
    - No inequalities are found, but one sequence has more elements than the other, and the shorter sequence is considered less than the longer sequence.
    - No inequalities are found and the sequences have the same number of elements, and so the sequences are equal and the result of the comparison is false.

```
//algorithm, lexicographical_compare()
#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> lst;
vector <int>::iterator Iter1, Iter2;
list <int>::iterator lst_Iter, lst_inIter;

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

int j;
for(j = 0; j <= 6; j++)
lst.push_back(5*j);

int k;
for(k = 0; k <= 5; k++)
vec2.push_back(10*k);

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

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

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

//Self lexicographical_comparison of vec1 under identity
cout<<"\nOperation: lexicographical_compare(vec1.begin(),\nvec1.end(),
vec2.begin(), vec2.end()).\n";
bool result1;
result1 = lexicographical_compare(vec1.begin(), vec1.end(), vec2.begin(),
vec2.end());
if(result1)
cout<<"Vector vec1 is lexicographically_less than vec2."<<endl;
else
cout<<"Vector vec1 is not lexicographically_less than vec2."<<endl;

//lexicographical_comparison of vec1 and lst under identity
cout<<"\nOperation: lexicographical_compare(vec1.begin(),\nvec1.end(),
lst.begin(), lst.end()).\n";
bool result2;
result2 = lexicographical_compare(vec1.begin(), vec1.end(), lst.begin(),
lst.end());
if(result2)
cout<<"Vector vec1 is lexicographically_less than lst."<<endl;
else
cout<<"Vector vec1 is lexicographically_less than lst."<<endl;

cout<<"\nOperation: lexicographical_compare(vec1.begin(),\nvec1.end(),
vec2.begin(), vec2.end(), twice).\n";
bool result3;
result3 = lexicographical_compare(vec1.begin(), vec1.end(), vec2.begin(),
vec2.end(), twice);
if(result3)
cout<<"Vector vec1 is lexicographically_less than\nvec2 based on twice."<<endl;
else
cout<<"Vector vec1 is not lexicographically_less than\nvec2 based on
twice."<<endl;
return 0;
}
```

Output:



```
Vector vec1 data: 0 5 10 15 20 25
List lst data: 0 5 10 15 20 25 30
Vector vec2 data: 0 10 20 30 40 50

Operation: lexicographical_compare(vec1.begin(),
vec1.end(), vec2.begin(), vec2.end()).
Vector vec1 is lexicographically_less than vec2.

Operation: lexicographical_compare(vec1.begin(),
vec1.end(), lst.begin(), lst.end()).
Vector vec1 is lexicographically_less than lst.

Operation: lexicographical_compare(vec1.begin(),
vec1.end(), vec2.begin(), vec2.end(), twice).
Vector vec1 is not lexicographically_less than
vec2 based on twice.
Press any key to continue
```

lower_bound()

- Finds the position where the first element in an ordered range is or would be if it had a value that is less than or equivalent to a specified value, where the sense of equivalence may be specified by a binary predicate.

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

Parameters

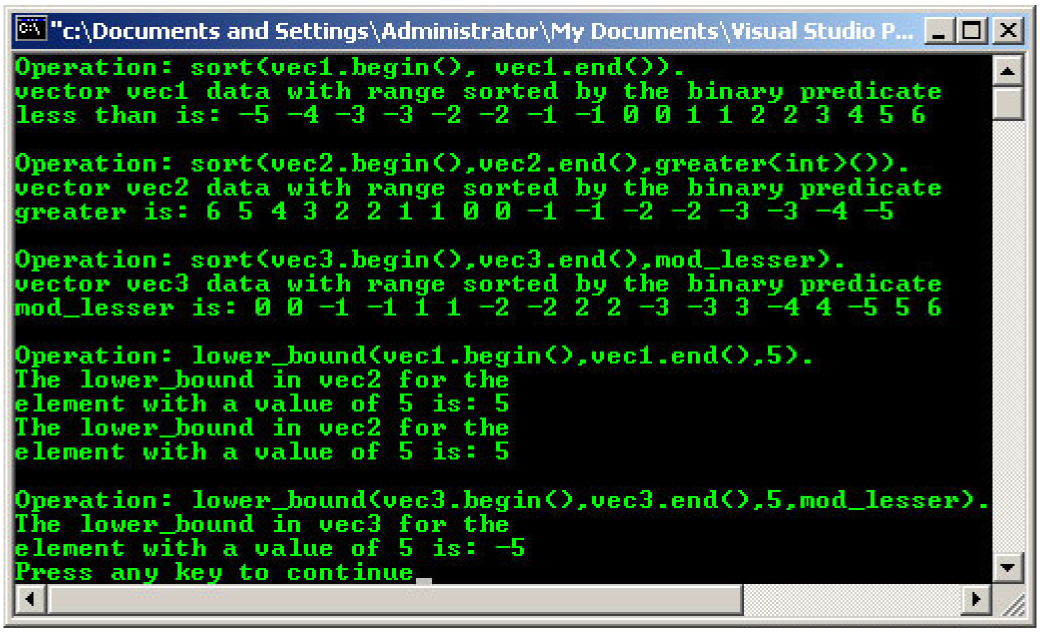
Parameter	Description
_First	A forward iterator addressing the position of the first element in the range to be searched.
_Last	A forward iterator addressing the position one past the final element in the range to be searched.
_Val	The value whose first position or possible first position is being searched for in the ordered range.
_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 35.4

- The return value is a forward iterator addressing the position where the first element in an ordered range is or would be if it had a value that is less than or equivalent to a specified value, where the sense of equivalence 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 `lower_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, lower_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
int i;
for(i = -3; i <= 6; i++)
vec1.push_back(i);

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

cout<<"Operation: sort(vec1.begin(), vec1.end()).\n";
sort(vec1.begin(), vec1.end());
cout<<"vector vec1 data with range sorted by the "
                              <<"binary predicate\nless than is: ";
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;

cout<<"\nOperation: sort(vec2.begin(), vec2.end(), greater<int>()).\n";
sort(vec2.begin(), vec2.end(), greater<int>());
cout<<"vector vec2 data with range sorted by the "
                       <<"binary predicate\ngreater is: ";
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;

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

//lower_bound of 5 in vec1 with default binary predicate less <int>()
cout<<"\nOperation: lower_bound(vec1.begin(), vec1.end(), 5).\n";
Result1 = lower_bound(vec1.begin(), vec1.end(), 5);
cout<<"The lower_bound in vec2 for the\nelement with a value of 5 is:
"<<*Result1<<endl;

//lower_bound of 5 in vec2 with the binary predicate greater<int>()
Result2 = lower_bound(vec2.begin(), vec2.end(), 5, greater<int>());
cout<<"The lower_bound in vec2 for the\nelement with a value of 5 is:
"<<*Result2<<endl;

//lower_bound of 5 in vec3 with the binary predicate mod_lesser
cout<<"\nOperation: lower_bound(vec3.begin(), vec3.end(), 5, mod_lesser).\n";
Result3 = lower_bound(vec3.begin(), vec3.end(), 5, mod_lesser);
cout<<"The lower_bound in vec3 for the\nelement with a value of 5 is:
"<<*Result3<<endl;
return 0;
}
```

Output



```
Operation: sort(vec1.begin(), vec1.end()).
vector vec1 data with range sorted by the binary predicate
less than is: -5 -4 -3 -3 -2 -2 -1 -1 0 0 1 1 2 2 3 4 5 6

Operation: sort(vec2.begin(),vec2.end(),greater<int>()).
vector vec2 data with range sorted by the binary predicate
greater is: 6 5 4 3 2 2 1 1 0 0 -1 -1 -2 -2 -3 -3 -4 -5

Operation: sort(vec3.begin(),vec3.end(),mod_lesser).
vector vec3 data with range sorted by the binary predicate
mod_lesser is: 0 0 -1 -1 1 1 -2 -2 2 2 -3 -3 3 -4 4 -5 5 6

Operation: lower_bound(vec1.begin(),vec1.end(),5).
The lower_bound in vec2 for the
element with a value of 5 is: 5
The lower_bound in vec2 for the
element with a value of 5 is: 5

Operation: lower_bound(vec3.begin(),vec3.end(),5,mod_lesser).
The lower_bound in vec3 for the
element with a value of 5 is: -5
Press any key to continue_
```

make_heap()

- Converts elements from a specified range into a heap in which the first element is the largest and for which a sorting criterion may be specified with a binary predicate.

```
template<class RandomAccessIterator>
   void make_heap(
      RandomAccessIterator _First,
      RandomAccessIterator _Last
   );
template<class RandomAccessIterator, class BinaryPredicate>
   void make_heap(
      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 converted into a 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.

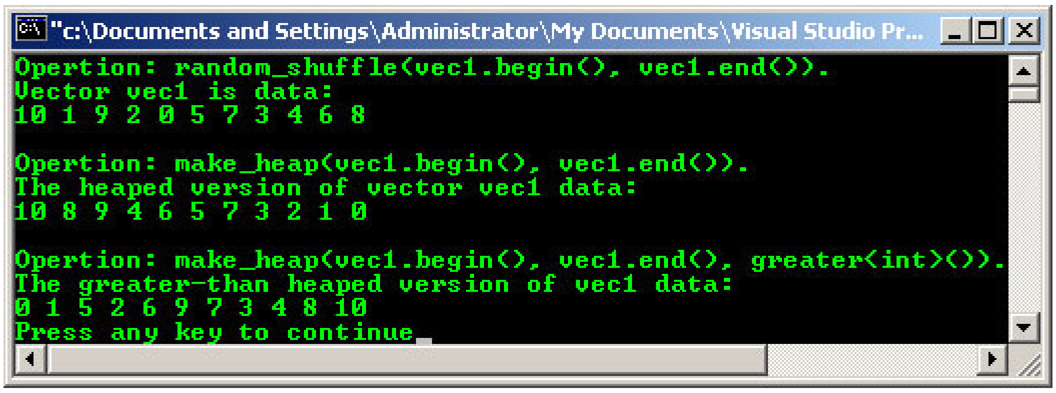
Table 35.5

- 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 Standard Template Library container adaptor `priority_queue` Class.
- The complexity is linear, requiring `3*(_Last – _First)` comparisons.

```
//algorithm, make_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 = 0; i <= 10; i++)
vec1.push_back(i);

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

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

//Make vec1 a heap with greater than ordering
cout<<"\nOperation: make_heap(vec1.begin(), vec1.end(), greater<int>()).\n";
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;
return 0;
}
```

Output



```
Opertion: random_shuffle(vec1.begin(), vec1.end()).
Vector vec1 is data:
10 1 9 2 0 5 7 3 4 6 8

Opertion: make_heap(vec1.begin(), vec1.end()).
The heaped version of vector vec1 data:
10 8 9 4 6 5 7 3 2 1 0

Opertion: make_heap(vec1.begin(), vec1.end(), greater<int>()).
The greater-than heaped version of vec1 data:
0 1 5 2 6 9 7 3 4 8 10
Press any key to continue_
```

"c:\Documents and Settings\Administrator\My Documents\Visual Studio Pr...

max()

- Compares two objects and returns the larger of the two, where the ordering criterion may be specified by a binary predicate.

```
template<class Type>
   const Type& max(
      const Type& _Left,
      const Type& _Right
   );
template<class Type, class Pr>
   const Type& max(
      const Type& _Left,
      const Type& _Right,
      BinaryPredicate _Comp
   );
```

Parameters

Parameter	Description
`_Left`	The first of the two objects being compared.
`_Right`	The second of the two objects being compared.
`_Comp`	A binary predicate used to compare the two objects.

Table 35.6

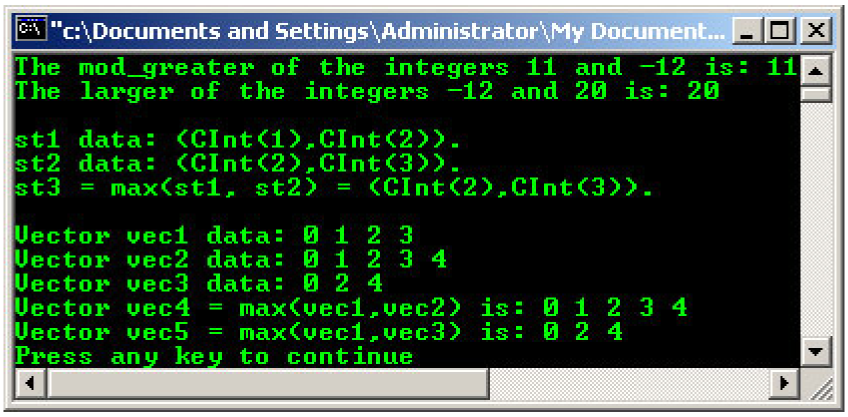
- The return value is the greater of the two objects unless neither is greater, in which case it returns the first of the two objects.
- The `max()` algorithm is unusual in having objects passed as parameters. Most STL algorithms operate on a range of elements whose position is specified by iterator passes as parameters.

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

class CInt;
ostream& operator<<(ostream& osIn, const CInt& rhs);

class CInt
{
public:
   CInt(int n = 0) : m_nVal(n){}
   CInt(const CInt& rhs) : m_nVal(rhs.m_nVal){}
   CInt& operator=(const CInt& rhs)
   {m_nVal = rhs.m_nVal; return *this;}
   bool operator<( const CInt& rhs ) const
   {return (m_nVal < rhs.m_nVal);}
   friend ostream& operator<<(ostream& osIn, const CInt& rhs);

private:
   int m_nVal;
};

inline ostream& operator<<(ostream& osIn, const CInt& rhs)
{
   osIn<<"CInt("<<rhs.m_nVal<<")";
   return osIn;
}

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

int main()
{
```

```
//Comparing integers directly using the max algorithm
int a = 11, b = -12, c = 20;
const int& result1 = max(a, b, mod_greater);
const int& result2 = max(b, c);

cout<<"The mod_greater of the integers 11 and -12 is: "<<result1<<endl;
cout<<"The larger of the integers -12 and 20 is: "<<result2<<endl;
cout<<endl;

//Comparing set containers with elements of type CInt
//using the max algorithm
CInt c1 = 1, c2 = 2, c3 = 3;
set<CInt> st1, st2, st3;
set<CInt>::iterator st1_Iter, st2_Iter, st3_Iter;

st1.insert(c1);
st1.insert(c2);
st2.insert(c2);
st2.insert(c3);

cout<<"st1 data: (";
for(st1_Iter = st1.begin(); st1_Iter != --st1.end(); st1_Iter++)
cout<<*st1_Iter<<",";
st1_Iter = --st1.end();
cout<<*st1_Iter<<")."<<endl;

cout<<"st2 data: (";
for(st2_Iter = st2.begin(); st2_Iter != --st2.end(); st2_Iter++)
cout<<*st2_Iter<<",";
st2_Iter = --st2.end();
cout<<*st2_Iter<<")."<<endl;

st3 = max(st1, st2);
cout<<"st3 = max(st1, st2) = (";
for(st3_Iter = st3.begin(); st3_Iter != --st3.end(); st3_Iter++)
cout<<*st3_Iter<<",";
st3_Iter = --st3.end();
cout<<*st3_Iter<<")."<<endl;

//Comparing vectors with integer elements using the max algorithm
vector <int> vec1, vec2, vec3, vec4, vec5;
vector <int>::iterator Iter1, Iter2, Iter3, Iter4, Iter5;

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

int j;
for(j = 0; j <= 4; j++)
vec2.push_back(j);

int k;
for(k = 0; k <= 2; k++)
vec3.push_back(2*k);

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

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

cout<<"Vector vec3 data: ";
for(Iter3 = vec3.begin(); Iter3 != vec3.end(); Iter3++)
cout<<*Iter3<<" ";
cout<<endl;

vec4 = max(vec1, vec2);
vec5 = max(vec1, vec3);

cout<<"Vector vec4 = max(vec1,vec2) is: ";
for(Iter4 = vec4.begin(); Iter4 != vec4.end(); Iter4++)
cout<<*Iter4<<" ";
cout<<endl;

cout<<"Vector vec5 = max(vec1,vec3) is: ";
for(Iter5 = vec5.begin(); Iter5 != vec5.end(); Iter5++)
```

```
cout<<*Iter5<<" ";
cout<<endl;
return 0;
}
```

Output



```
The mod_greater of the integers 11 and -12 is: 11
The larger of the integers -12 and 20 is: 20

st1 data: (CInt(1),CInt(2)).
st2 data: (CInt(2),CInt(3)).
st3 = max(st1, st2) = (CInt(2),CInt(3)).

Vector vec1 data: 0 1 2 3
Vector vec2 data: 0 1 2 3 4
Vector vec3 data: 0 2 4
Vector vec4 = max(vec1,vec2) is: 0 1 2 3 4
Vector vec5 = max(vec1,vec3) is: 0 2 4
Press any key to continue
```

max_element()

- Finds the first occurrence of largest element in a specified range where the ordering criterion may be specified by a binary predicate.

```
template<class ForwardIterator>
   ForwardIterator max_element(
      ForwardIterator _First,
      ForwardIterator _Last
   );
template<class ForwardIterator, class BinaryPredicate>
   ForwardIterator max_element(
      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 searched for the largest element.
_Last	A forward iterator addressing the position one past the final element in the range to be searched for the largest element.
_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.

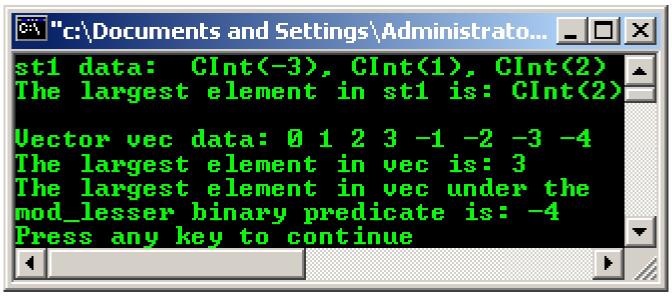
Table 35.7

- The return value is a forward iterator addressing the position of the first occurrence of the largest element in the range searched.
- The range referenced must be valid; all pointers must be dereferenceable and within each sequence the last position is reachable from the first by incrementation.
- The complexity is linear: (*_Last* – *_First*)–1 comparison is required for a nonempty range.

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

class CInt;
ostream& operator<<(ostream& osIn, const CInt& rhs);

class CInt
{
```

```
public:
   CInt(int n = 0) : m_nVal(n){}
   CInt(const CInt& rhs) : m_nVal(rhs.m_nVal){}
   CInt& operator=(const CInt& rhs)
   {m_nVal = rhs.m_nVal; return *this;}
   bool operator<(const CInt& rhs) const
   {return (m_nVal < rhs.m_nVal);}
   friend ostream& operator<<(ostream& osIn, const CInt& rhs);

private:
   int m_nVal;
};

inline ostream& operator<<(ostream& osIn, const CInt& rhs)
{
   osIn<<"CInt("<<rhs.m_nVal<<")";
   return osIn;
}

//Return whether modulus of elem1 is greater 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()
{
//Searching a set container with elements of type CInt
//for the maximum element
CInt c1 = 1, c2 = 2, c3 = -3;
set<CInt> st1;
set<CInt>::iterator st1_Iter, st2_Iter, st3_Iter;

st1.insert(c1);
st1.insert(c2);
st1.insert(c3);

cout<<"st1 data: ";
for(st1_Iter = st1.begin(); st1_Iter != --st1.end(); st1_Iter++)
cout<<" "<<*st1_Iter<<",";
st1_Iter = --st1.end();
cout<<" "<<*st1_Iter<<endl;

st2_Iter = max_element(st1.begin(), st1.end());

cout<<"The largest element in st1 is: "<<*st2_Iter<<endl;
cout<<endl;

//Searching a vector with elements of type int for the maximum
//element under default less than & mod_lesser binary predicates
vector <int> vec;
vector <int>::iterator vec_Iter, vec1_Iter, vec2_Iter;

int i;
for(i = 0; i <= 3; i++)
vec.push_back(i);

int j;
for(j = 1; j <= 4; j++)
vec.push_back(-j);

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

vec1_Iter = max_element(vec.begin(), vec.end());
vec2_Iter = max_element(vec.begin(), vec.end(), mod_lesser);

cout<<"The largest element in vec is: "<<*vec1_Iter<<endl;
cout<<"The largest element in vec under the\nmod_lesser"
<<" binary predicate is: "<<*vec2_Iter<<endl;
return 0;
}
```

Output



```
st1 data:  CInt(-3), CInt(1), CInt(2)
The largest element in st1 is: CInt(2)

Vector vec data: 0 1 2 3 -1 -2 -3 -4
The largest element in vec is: 3
The largest element in vec under the
mod_lesser binary predicate is: -4
Press any key to continue
```

merge()

- Combines all of the elements from 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 merge(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      InputIterator2 _Last2,
      OutputIterator _Result
   );
template<class InputIterator1, class InputIterator2, class OutputIterator, class
BinaryPredicate>
   OutputIterator merge(
      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 combined and sorted into a single range.
_Last1	An input iterator addressing the position one past the last element in the first of two sorted source ranges to be combined and sorted into a single range.
_First2	An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be combined and sorted into a single range.
_Last2	An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be combined and sorted into a single range.
_Result	An output iterator addressing the position of the first element in the destination range where the two source ranges are to be combined into a single sorted range.
_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.

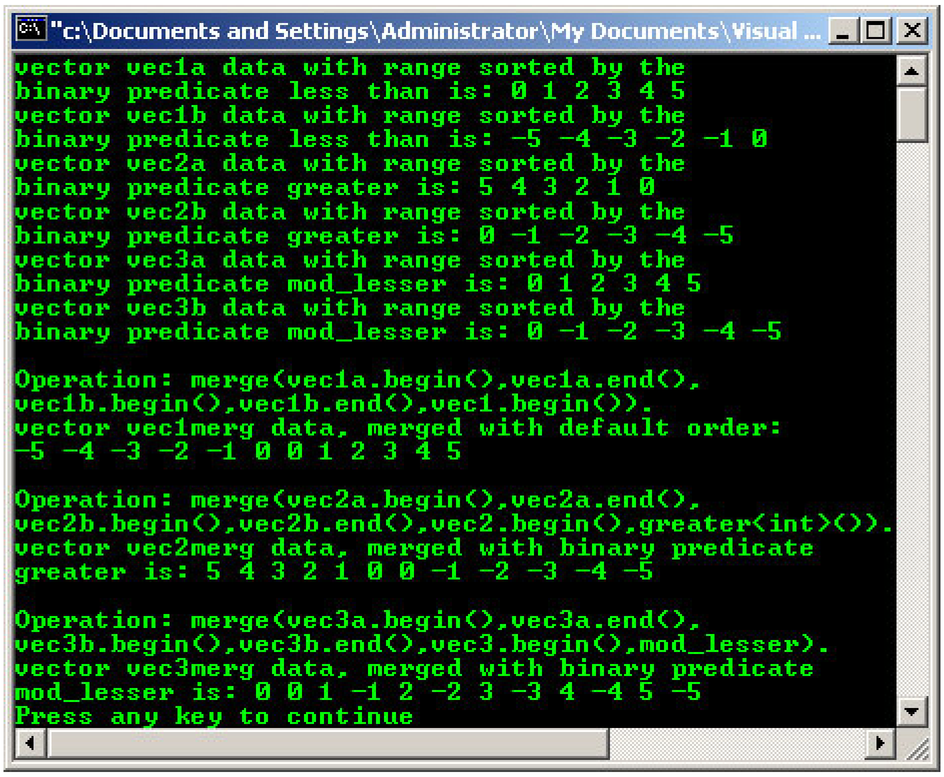
Table 35.8

- The return value is an output iterator addressing the position one past the last element in the sorted destination range.
- 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.
- The complexity of the algorithm is linear with at most `(_Last1 – _First1)–(_Last2 – _First2)–1` comparisons.
- The `list` class provides a member function `merge()` to merge the elements of two lists.

```
//algorithm, merge()
#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;

//Constructing vector vec1a and vec1b with default less than ordering
int i;
for(i = 0; i <= 5; i++)
vec1a.push_back(i);

int j;
for(j =-5; j <= 0; j++)
vec1b.push_back(j);

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

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

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

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

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

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

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

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

//To merge in ascending order with default binary
//predicate less <int>()
cout<<"\nOperation:
merge(vec1a.begin(),vec1a.end(),\nvec1b.begin(),vec1b.end(),vec1.begin()).\n";
merge(vec1a.begin(), vec1a.end(), vec1b.begin(), vec1b.end(), vec1.begin());
cout<<"vector vec1merg data, merged with default order:\n";
for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

//To merge in descending order, specify binary
//predicate greater<int>()
cout<<"\nOperation:
merge(vec2a.begin(),vec2a.end(),\nvec2b.begin(),vec2b.end(),vec2.begin(),greater<i
nt>()).\n";
merge(vec2a.begin(), vec2a.end(), vec2b.begin(), vec2b.end(), vec2.begin(),
greater<int>());
cout<<"vector vec2merg data, merged with binary predicate\ngreater is: ";
for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
cout<<*Iter2<<" ";
cout<<endl;

//Applying a user-defined binary predicate mod_lesser
cout<<"\nOperation:
merge(vec3a.begin(),vec3a.end(),\nvec3b.begin(),vec3b.end(),vec3.begin(),mod_lesse
r).\n";
merge(vec3a.begin(), vec3a.end(), vec3b.begin(), vec3b.end(), vec3.begin(),
mod_lesser);
cout<<"vector vec3merg data, merged with binary predicate\nmod_lesser is: ";
for(Iter3 = vec3.begin(); Iter3 != vec3.end(); Iter3++)
cout<<*Iter3<<" ";
cout<<endl;
 return 0;
}
```

Output



```
vector vec1a data with range sorted by the
binary predicate less than is: 0 1 2 3 4 5
vector vec1b data with range sorted by the
binary predicate less than is: -5 -4 -3 -2 -1 0
vector vec2a data with range sorted by the
binary predicate greater is: 5 4 3 2 1 0
vector vec2b data with range sorted by the
binary predicate greater is: 0 -1 -2 -3 -4 -5
vector vec3a data with range sorted by the
binary predicate mod_lesser is: 0 1 2 3 4 5
vector vec3b data with range sorted by the
binary predicate mod_lesser is: 0 -1 -2 -3 -4 -5

Operation: merge(vec1a.begin(),vec1a.end(),
vec1b.begin(),vec1b.end(),vec1.begin()).
vector vec1merg data, merged with default order:
-5 -4 -3 -2 -1 0 0 1 2 3 4 5

Operation: merge(vec2a.begin(),vec2a.end(),
vec2b.begin(),vec2b.end(),vec2.begin(),greater<int>()).
vector vec2merg data, merged with binary predicate
greater is: 5 4 3 2 1 0 0 -1 -2 -3 -4 -5

Operation: merge(vec3a.begin(),vec3a.end(),
vec3b.begin(),vec3b.end(),vec3.begin(),mod_lesser).
vector vec3merg data, merged with binary predicate
mod_lesser is: 0 0 1 -1 2 -2 3 -3 4 -4 5 -5
Press any key to continue
```

"c:\Documents and Settings\Administrator\My Documents\Visual ...

min()

- Compares two objects and returns the lesser of the two, where the ordering criterion may be specified by a binary predicate.

```
template<class Type>
   const Type& min(
      const Type& _Left,
      const Type& _Right
   );
template<class Type, class Pr>
   const Type& min(
      const Type& _Left,
      const Type& _Right,
      BinaryPredicate _Comp
   );
```

Parameters

Parameter	Description
_Left	The first of the two objects being compared.
_Right	The second of the two objects being compared.
_Comp	A binary predicate used to compare the two objects.

Table 35.9

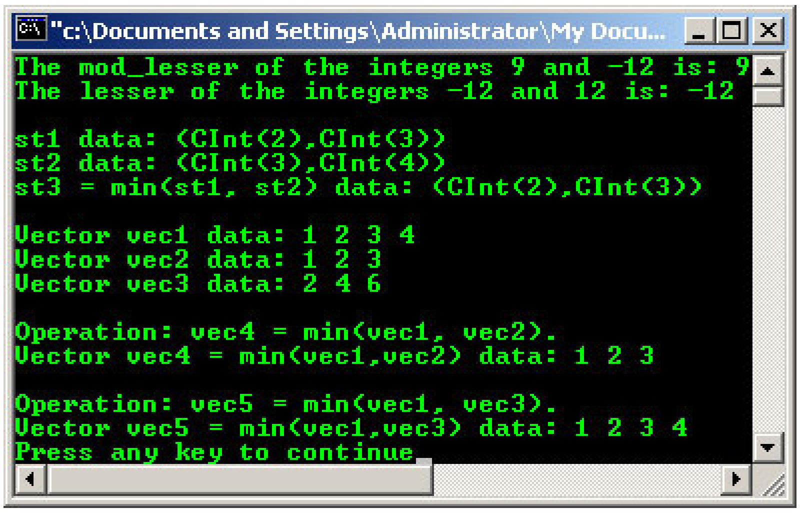
- The return value is the lesser of the two objects unless neither is lesser, in which case it returns the first of the two objects.
- The `min()` algorithm is unusual in having objects passed as parameters. Most STL algorithms operate on a range of elements whose position is specified by iterators passed as parameters.

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

class CInt;
ostream& operator<<(ostream& osIn, const CInt& rhs);

class CInt
{
public:
   CInt(int n = 0) : m_nVal(n){}
   CInt(const CInt& rhs) : m_nVal(rhs.m_nVal){}
   CInt& operator=(const CInt& rhs)
   {
m_nVal = rhs.m_nVal;
       return *this;
   }
   bool operator<(const CInt& rhs) const
      {return (m_nVal < rhs.m_nVal);}
   friend ostream& operator<<(ostream& osIn, const CInt& rhs);

private:
   int m_nVal;
};

inline ostream& operator<<(ostream& osIn, const CInt& rhs)
{
   osIn<<"CInt("<<rhs.m_nVal<<")";
   return osIn;
}

//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()
{
//Comparing integers directly using the min algorithm with
//binary predicate mod_lesser & with default less than
int a = 9, b = -12, c = 12;
const int& result1 = min(a, b, mod_lesser);
const int& result2 = min(b, c);

cout<<"The mod_lesser of the integers 9 and -12 is: "<<result1<<endl;
cout<<"The lesser of the integers -12 and 12 is: "<<result2<<endl;
cout<<endl;

//Comparing set containers with elements of type CInt
//using the min algorithm
CInt ci1 = 2, ci2 = 3, ci3 = 4;
set<CInt> st1, st2, st3;
set<CInt>::iterator st1Iter, st2Iter, st3Iter;

st1.insert(ci1);
st1.insert(ci2);
st2.insert(ci2);
st2.insert(ci3);

cout<<"st1 data: (";
for(st1Iter = st1.begin(); st1Iter != --st1.end(); st1Iter++)
cout<<*st1Iter<<",";
st1Iter = --st1.end();
cout<<*st1Iter<<")"<<endl;

cout<<"st2 data: (";
for(st2Iter = st2.begin(); st2Iter != --st2.end(); st2Iter++)
cout<<*st2Iter<<",";
st2Iter = --st2.end();
cout<<*st2Iter<<")"<<endl;

st3 = min(st1, st2);
cout<<"st3 = min(st1, st2) data: (";
for(st3Iter = st3.begin(); st3Iter != --st3.end(); st3Iter++)
cout<<*st3Iter<<",";
st3Iter = --st3.end();
cout<<*st3Iter<<")"<<endl;

//Comparing vectors with integer elements using min algorithm
vector <int> vec1, vec2, vec3, vec4, vec5;
vector <int>::iterator Iter1, Iter2, Iter3, Iter4, Iter5;

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

int j;
for(j = 1; j <= 3; j++)
vec2.push_back(j);

int k;
for(k = 1; k <= 3; k++)
vec3.push_back(2*k);

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

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

cout<<"Vector vec3 data: ";
for(Iter3 = vec3.begin(); Iter3 != vec3.end(); Iter3++)
cout<<*Iter3<<" ";
cout<<endl;

cout<<"\nOperation: vec4 = min(vec1, vec2).\n";
vec4 = min(vec1, vec2);
cout<<"Vector vec4 = min(vec1,vec2) data: ";
for(Iter4 = vec4.begin(); Iter4 != vec4.end(); Iter4++)
cout<<*Iter4<<" ";
```

```
cout<<endl;

cout<<"\nOperation: vec5 = min(vec1, vec3).\n";
vec5 = min(vec1, vec3);
cout<<"Vector vec5 = min(vec1,vec3) data: ";
for(Iter5 = vec5.begin(); Iter5 != vec5.end(); Iter5++)
cout<<*Iter5<<" ";
cout<<endl;
return 0;
}
```

Output:



```
The mod_lesser of the integers 9 and -12 is: 9
The lesser of the integers -12 and 12 is: -12

st1 data: (CInt(2),CInt(3))
st2 data: (CInt(3),CInt(4))
st3 = min(st1, st2) data: (CInt(2),CInt(3))

Vector vec1 data: 1 2 3 4
Vector vec2 data: 1 2 3
Vector vec3 data: 2 4 6

Operation: vec4 = min(vec1, vec2).
Vector vec4 = min(vec1,vec2) data: 1 2 3

Operation: vec5 = min(vec1, vec3).
Vector vec5 = min(vec1,vec3) data: 1 2 3 4
Press any key to continue
```

min_element()

- Finds the first occurrence of smallest element in a specified range where the ordering criterion may be specified by a binary predicate.

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

Parameters

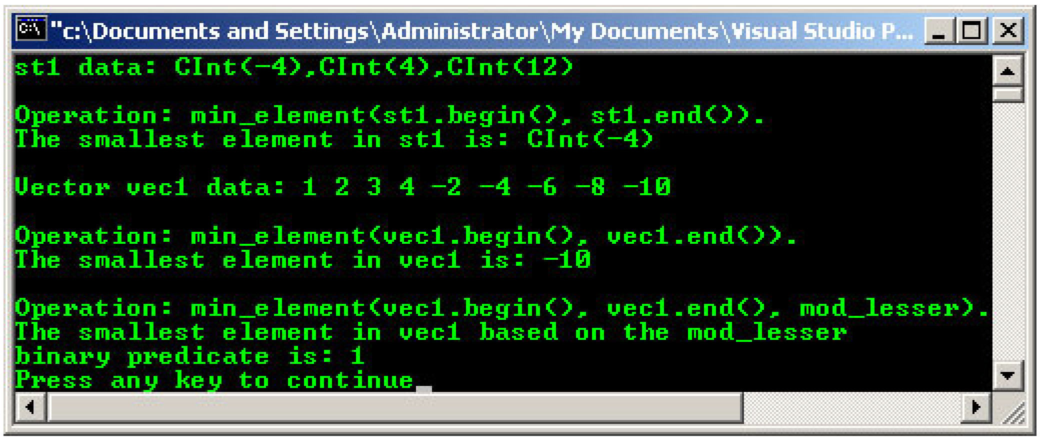
Parameter	Description
_First	A forward iterator addressing the position of the first element in the range to be searched for the largest element.
_Last	A forward iterator addressing the position one past the final element in the range to be searched for the largest element.
_Comp	User-defined predicate 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 35.10

- The return value is a forward iterator addressing the position of the first occurrence of the smallest element in the range searched.
- The range 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 complexity is linear: (*_Last* – *_First*)–1 comparison is required for a non-empty range.

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

class CInt;
ostream& operator<<(ostream& osIn, const CInt& rhs);

class CInt
{
public:
   CInt(int n = 0) : m_nVal(n){}
   CInt(const CInt& rhs) : m_nVal( rhs.m_nVal ){}
   CInt& operator=(const CInt& rhs)
   {
   m_nVal = rhs.m_nVal;
   return *this;
   }
   bool operator<(const CInt& rhs) const
      {return (m_nVal < rhs.m_nVal);}
   friend ostream& operator<<(ostream& osIn, const CInt& rhs);

private:
   int m_nVal;
};

inline ostream& operator<<(ostream& osIn, const CInt& rhs)
{
   osIn<<"CInt("<<rhs.m_nVal<<")";
   return osIn;
}

//Return whether modulus of elem1 is greater 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()
{
//Searching a set container with elements of type CInt
//for the minimum element
CInt ci1 = 4, ci2 = 12, ci3 = -4;
set<CInt> st1;
set<CInt>::iterator st1Iter, st2Iter, st3Iter;

st1.insert(ci1);
st1.insert(ci2);
st1.insert(ci3);

cout<<"st1 data: ";
for(st1Iter = st1.begin(); st1Iter != --st1.end(); st1Iter++)
cout<<*st1Iter<<",";
st1Iter = --st1.end();
cout<<*st1Iter<<endl;

cout<<"\nOperation: min_element(st1.begin(), st1.end()).\n";
st2Iter = min_element(st1.begin(), st1.end());
cout<<"The smallest element in st1 is: "<<*st2Iter<<endl;

//Searching a vector with elements of type int for the maximum
//element under default less than & mod_lesser binary predicates
vector <int> vec1;
vector <int>::iterator vec1Iter, vec2Iter, vec3Iter;

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

int j;
for(j = 1; j <= 5; j++)
vec1.push_back(-2*j);

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

cout<<"\nOperation: min_element(vec1.begin(), vec1.end()).\n";
vec2Iter = min_element(vec1.begin(), vec1.end());
cout<<"The smallest element in vec1 is: "<<*vec2Iter<<endl;

cout<<"\nOperation: min_element(vec1.begin(), vec1.end(), mod_lesser).\n";
vec3Iter = min_element(vec1.begin(), vec1.end(), mod_lesser);
cout<<"The smallest element in vec1 based on the mod_lesser"
<<"\nbinary predicate is: "<<*vec3Iter<<endl;
return 0;
}
```

Output:


"c:\Documents and Settings\Administrator\My Documents\Visual Studio P...

```
st1 data: CInt(-4),CInt(4),CInt(12)

Operation: min_element(st1.begin(), st1.end()).
The smallest element in st1 is: CInt(-4)

Vector vec1 data: 1 2 3 4 -2 -4 -6 -8 -10

Operation: min_element(vec1.begin(), vec1.end()).
The smallest element in vec1 is: -10

Operation: min_element(vec1.begin(), vec1.end(), mod_lesser).
The smallest element in vec1 based on the mod_lesser
binary predicate is: 1
Press any key to continue
```

mismatch()

- Compares two ranges element by element either for equality or equivalent in a sense specified by a binary predicate and locates the first position where a difference occurs.

```
template<class InputIterator1, class InputIterator2>
   pair<InputIterator1, InputIterator2> mismatch(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2
  );
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
   pair<InputIterator1, InputIterator2> mismatch(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2
      BinaryPredicate _Comp
   );
```

Parameters

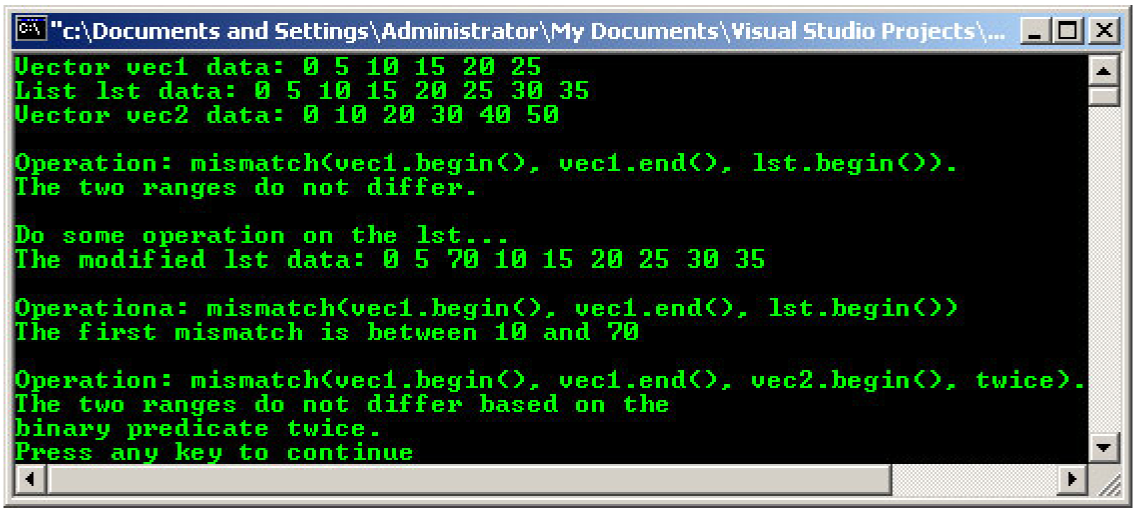
Parameter	Description
_First1	An input iterator addressing the position of the first element in the first range to be tested.
_Last1	An input iterator addressing the position one past the final element in the first range to be tested.
_First2	An input iterator addressing the position of the first element in the second range to be tested.
_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 35.11

- The return value is a pair of iterators addressing the positions of the mismatch in the two ranges, the first component iterator to the position in the first range and the second component iterator to the position in the second range.
- If there is no difference between the elements in the ranges compared or if the binary predicate in the second version is satisfied by all element pairs from the two ranges, then the first component iterator points to the position one past the final element in the first range and the second component iterator to position one past the final element tested in the second range.
- The range to be searched must be valid; all pointers must be de-referenceable and the last position is reachable from the first by incrementation.
- The time complexity of the algorithm is linear in the number of elements contained in the range.
- The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

```
//algorithm, mismatch()
#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 (elem1 * 2 == elem2);}

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

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

int j;
for(j = 0; j <= 7; j++)
lst.push_back(5*j);

int k;
for(k = 0; k <= 5; k++)
vec2.push_back(10*k);

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

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

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

//Testing vec1 and lst for mismatch under identity
pair<vector <int>::iterator, list <int>::iterator> result1;
result1 = mismatch(vec1.begin(), vec1.end(), lst.begin());

cout<<"\nOperation: mismatch(vec1.begin(), vec1.end(), lst.begin()).\n";
if(result1.first == vec1.end())
cout<<"The two ranges do not differ."<<endl;
else
cout<<"The fist mismatch is between "<<*result1.first<<" and
"<<*result1.second<<endl;

//Modifying the lst
cout<<"\nDo some operation on the lst...\n";
lst_inIter = lst.begin();
lst_inIter++;
lst_inIter++;
lst.insert(lst_inIter, 70);
cout<<"The modified lst data: ";
for(lst_Iter = lst.begin(); lst_Iter!= lst.end(); lst_Iter++)
cout<<*lst_Iter<<" ";
cout<<endl;

//Testing vec1 with modified lst for mismatch under identity
cout<<"\nOperationa: mismatch(vec1.begin(), vec1.end(), lst.begin())\n";
result1 = mismatch(vec1.begin(), vec1.end(), lst.begin());
if(result1.first == vec1.end())
cout<<"The two ranges do not differ."<<endl;
else
cout<<"The first mismatch is between "<<*result1.first<<" and "<<*result1.second<<
endl;

//Test vec1 and vec2 for mismatch under the binary predicate twice
pair<vector <int>::iterator, vector <int>::iterator> result2;
cout<<"\nOperation: mismatch(vec1.begin(), vec1.end(), vec2.begin(), twice).\n";
result2 = mismatch(vec1.begin(), vec1.end(), vec2.begin(), twice);
if(result2.first == vec1.end())
cout<<"The two ranges do not differ based on the\nbinary predicate twice."<<endl;
else
cout<<"The first mismatch is between "<<*result2.first<<" and
"<<*result2.second<<endl;
return 0;
}
```

Output



```
Vector vec1 data: 0 5 10 15 20 25
List lst data: 0 5 10 15 20 25 30 35
Vector vec2 data: 0 10 20 30 40 50

Operation: mismatch(vec1.begin(), vec1.end(), lst.begin()).
The two ranges do not differ.

Do some operation on the lst...
The modified lst data: 0 5 70 10 15 20 25 30 35

Operationa: mismatch(vec1.begin(), vec1.end(), lst.begin())
The first mismatch is between 10 and 70

Operation: mismatch(vec1.begin(), vec1.end(), vec2.begin(), twice).
The two ranges do not differ based on the
binary predicate twice.
Press any key to continue
```

next_permutation()

- Reorders the elements in a range so that the original ordering is replaced by the lexicographically next greater permutation if it exists, where the sense of next may be specified with a binary predicate.

```
template<class BidirectionalIterator>
   bool next_permutation(
      BidirectionalIterator _First,
      BidirectionalIterator _Last
   );
template<class BidirectionalIterator, class BinaryPredicate>
   bool next_permutation(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

Parameter	Description
_First	A bidirectional iterator pointing to the position of the first element in the range to be permuted.
_Last	A bidirectional iterator pointing to the position one past the final element in the range to be permuted.
_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

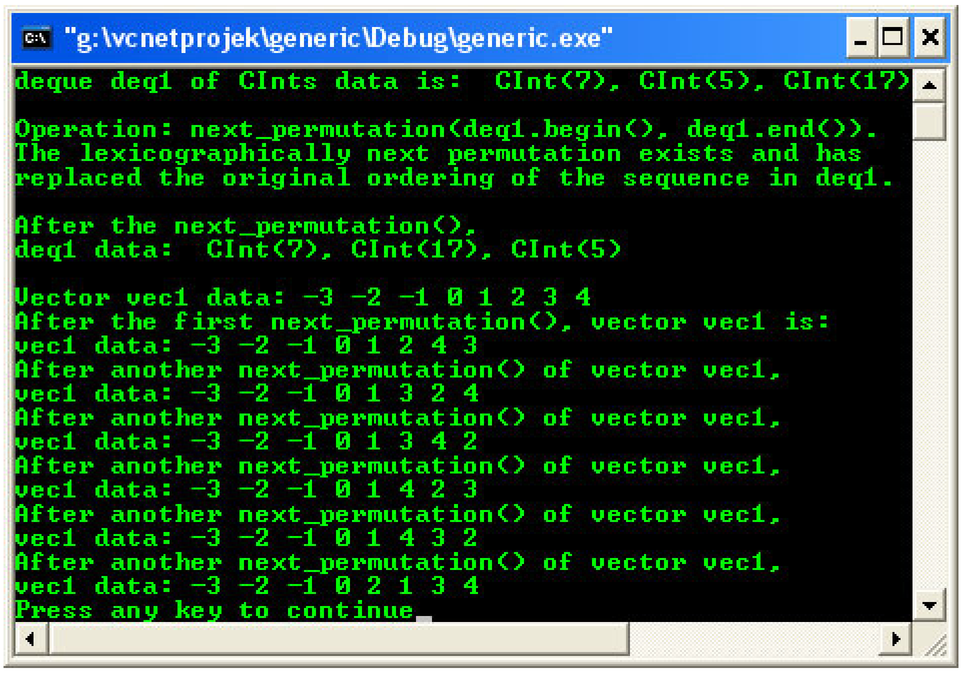
	arguments and returns true when satisfied and false when not satisfied.

Table 35.12

- The return value is true if the lexicographically next permutation exists and has replaced the original ordering of the range; otherwise false, in which case the ordering is transformed into the lexicographically smallest permutation.
- The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.
- The default binary predicate is less than and the elements in the range must be less than comparable to insure that the next permutation is well defined.
- The complexity is linear with at most `(_Last–_First)/2` swaps.

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

class CInt;
ostream& operator<<(ostream& osIn, const CInt& rhs);

class CInt
{
public:
   CInt(int n = 0) : m_nVal(n){}
   CInt(const CInt& rhs) : m_nVal(rhs.m_nVal){}
   CInt& operator=(const CInt& rhs)
   {m_nVal = rhs.m_nVal; return *this;}
   bool operator<(const CInt& rhs) const
      {return (m_nVal < rhs.m_nVal);}
   friend ostream& operator<<(ostream& osIn, const CInt& rhs);

private:
   int m_nVal;
};

inline ostream& operator<<(ostream& osIn, const CInt& rhs)
{
   osIn<<"CInt("<<rhs.m_nVal<<")";
   return osIn;
}

//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()
{
//Reordering the elements of type CInt in a deque
//using the prev_permutation algorithm
CInt ci1 = 7, ci2 = 5, ci3 = 17;
bool deq1Result;
deque<CInt> deq1, deq2, deq3;
deque<CInt>::iterator deq1Iter;

deq1.push_back(ci1);
deq1.push_back(ci2);
deq1.push_back(ci3);

cout<<"deque deq1 of CInts data is: ";
for(deq1Iter = deq1.begin(); deq1Iter != --deq1.end(); deq1Iter++)
cout<<" "<<*deq1Iter<<",";
deq1Iter = --deq1.end();
cout<<" "<<*deq1Iter<<endl;

cout<<"\nOperation: next_permutation(deq1.begin(), deq1.end()).\n";
deq1Result = next_permutation(deq1.begin(), deq1.end());
if(deq1Result)
cout<<"The lexicographically next permutation "
<<"exists and has\nreplaced the original "
<<"ordering of the sequence in deq1."<<endl;
else
cout<<"The lexicographically next permutation doesn't "
<<"exist\n and the lexicographically "
<<"smallest permutation\n has replaced the "
<<"ordering of the sequence in deq1."<<endl;

cout<<"\nAfter the next_permutation,\ndeq1 data: ";
for(deq1Iter = deq1.begin(); deq1Iter != --deq1.end(); deq1Iter++)
cout<<" "<<*deq1Iter<<",";
deq1Iter = --deq1.end();
cout<<" "<<*deq1Iter<<endl;

//Permuting vector elements with binary function mod_lesser
vector <int> vec1;
vector <int>::iterator Iter1;

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

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

next_permutation(vec1.begin(), vec1.end(), mod_lesser);
cout<<"After the first next_permutation(), vector vec1 is:\nvec1 data: ";
for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

int k = 1;
while (k <= 5)
{
next_permutation(vec1.begin(), vec1.end(), mod_lesser);
cout<<"After another next_permutation() of vector vec1,\nvec1 data: ";
for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1 ++)
cout<<*Iter1<<" ";
cout<<endl;
k++;
}
return 0;
}
```

Output:



```
deque deq1 of CInts data is:  CInt(7), CInt(5), CInt(17)

Operation: next_permutation(deq1.begin(), deq1.end()).
The lexicographically next permutation exists and has
replaced the original ordering of the sequence in deq1.

After the next_permutation(),
deq1 data:  CInt(7), CInt(17), CInt(5)

Vector vec1 data: -3 -2 -1 0 1 2 3 4
After the first next_permutation(), vector vec1 is:
vec1 data: -3 -2 -1 0 1 2 4 3
After another next_permutation() of vector vec1,
vec1 data: -3 -2 -1 0 1 3 2 4
After another next_permutation() of vector vec1,
vec1 data: -3 -2 -1 0 1 3 4 2
After another next_permutation() of vector vec1,
vec1 data: -3 -2 -1 0 1 4 2 3
After another next_permutation() of vector vec1,
vec1 data: -3 -2 -1 0 1 4 3 2
After another next_permutation() of vector vec1,
vec1 data: -3 -2 -1 0 2 1 3 4
Press any key to continue
```

nth_element()

- Partitions a range of elements, correctly locating the *n*th element of the sequence in the range so that all the elements in front of it are less than or equal to it and all the elements that follow it in the sequence are greater than or equal to it.

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

Parameters

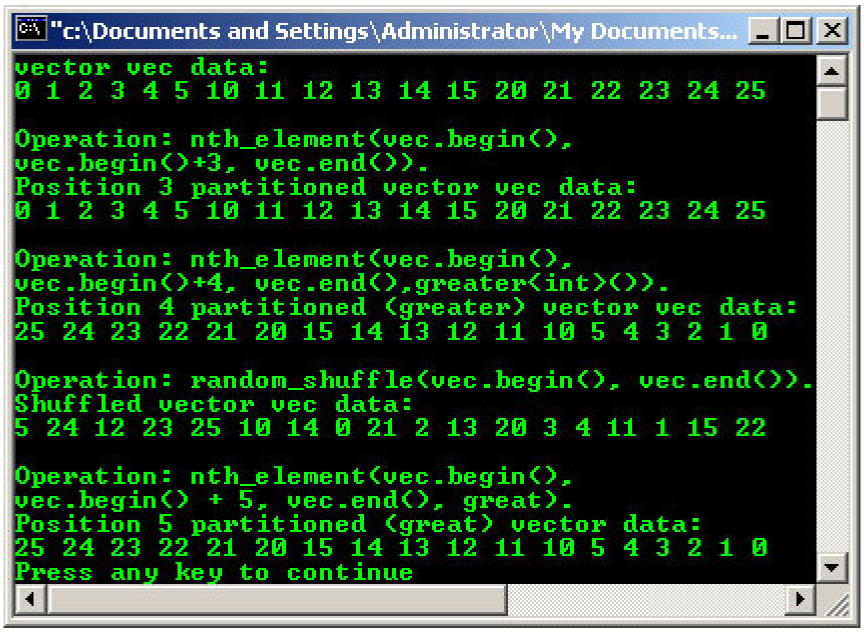
Parameter	Description
`_First`	A random-access iterator addressing the position of the first element in the range to be partitioned.
`_Nth`	A random-access iterator addressing the position of element to be correctly ordered on the boundary of the partition.
`_Last`	A random-access iterator addressing the position one past the final element in the range to be partitioned.
`_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 35.13

- 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 `nth_element()` algorithm does not guarantee that elements in the sub-ranges either side of the *n*th element are sorted. It thus makes fewer guarantees than `partial_sort()`, which orders the elements in the range below some chosen element, and may be used as a faster alternative to `partial_sort()` when the ordering of the lower range is not required.
- Elements are equivalent, but not necessarily equal, if neither is less than the other.
- The average of a sort complexity is linear with respect to `_Last – _First`.

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

//user defined function, return whether
//first element is greater than the second
bool great(int elem1, int elem2)
{return (elem1 > elem2);}

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

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

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

int k;
for(k = 20; k <= 25; k++)
vec.push_back(k);

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

cout<<"\nOperation: nth_element(vec.begin(),\nvec.begin()+3, vec.end()).\n";
nth_element(vec.begin(), vec.begin()+3, vec.end());
cout<<"Position 3 partitioned vector vec data:\n";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

//To sort in descending order, specify binary predicate
cout<<"\nOperation: nth_element(vec.begin(),\nvec.begin()+4,
vec.end(),greater<int>()).\n";
nth_element(vec.begin(), vec.begin()+4, vec.end(), greater<int>());
cout<<"Position 4 partitioned (greater) vector vec data:\n";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

cout<<"\nOperation: random_shuffle(vec.begin(), vec.end()).\n";
random_shuffle(vec.begin(), vec.end());
cout<<"Shuffled vector vec data:\n";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

//A user-defined binary predicate...
cout<<"\nOperation: nth_element(vec.begin(),\nvec.begin() + 5, vec.end(),
great).\n";
nth_element(vec.begin(), vec.begin() + 5, vec.end(), great);
cout<<"Position 5 partitioned (great) vector data:\n";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;
return 0;
}
```

Output:



```
vector vec data:
0 1 2 3 4 5 10 11 12 13 14 15 20 21 22 23 24 25

Operation: nth_element(vec.begin(),
vec.begin()+3, vec.end()).
Position 3 partitioned vector vec data:
0 1 2 3 4 5 10 11 12 13 14 15 20 21 22 23 24 25

Operation: nth_element(vec.begin(),
vec.begin()+4, vec.end(),greater<int>()).
Position 4 partitioned (greater) vector vec data:
25 24 23 22 21 20 15 14 13 12 11 10 5 4 3 2 1 0

Operation: random_shuffle(vec.begin(), vec.end()).
Shuffled vector vec data:
5 24 12 23 25 10 14 0 21 2 13 20 3 4 11 1 15 22

Operation: nth_element(vec.begin(),
vec.begin() + 5, vec.end(), great).
Position 5 partitioned (great) vector data:
25 24 23 22 21 20 15 14 13 12 11 10 5 4 3 2 1 0
Press any key to continue
```

partial_sort()

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

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

Parameters

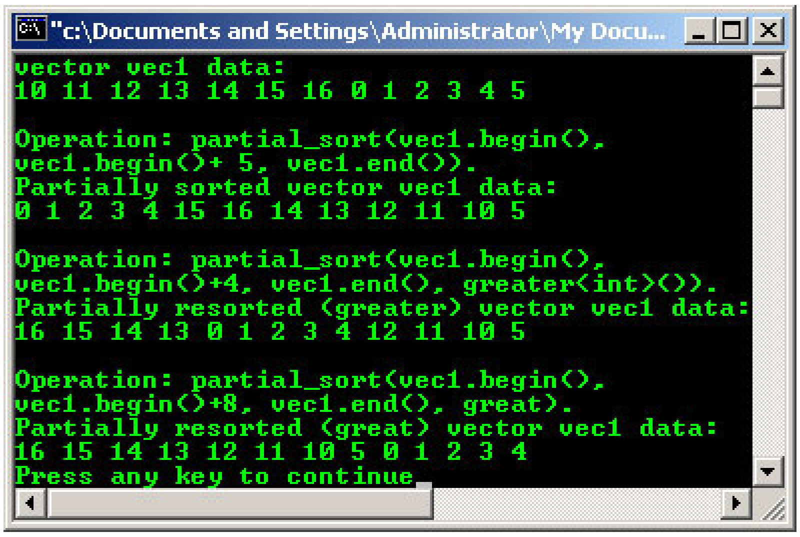
Parameter	Description
`_First`	A random-access iterator addressing the position of the first element in the range to be sorted.
`_SortEnd`	A random-access iterator addressing the position one past the final element in the sub-range to be sorted.
`_Last`	A random-access iterator addressing the position one past the final element in the range to be partially 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 35.14

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

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

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

//fill up the vector with data...
int i;
for(i = 10; i <= 16; i++)
vec1.push_back(i);

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

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

cout<<"\nOperation: partial_sort(vec1.begin(),\nvec1.begin()+ 5, vec1.end()).\n";
partial_sort(vec1.begin(), vec1.begin() + 5, vec1.end());
cout<<"Partially sorted vector vec1 data:\n";
for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

//To partially sort in descending order, specify binary predicate
cout<<"\nOperation: partial_sort(vec1.begin(),\nvec1.begin()+4, vec1.end(),
greater<int>()).\n";
partial_sort(vec1.begin(), vec1.begin()+4, vec1.end(), greater<int>());
cout<<"Partially resorted (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
cout<<"\nOperation: partial_sort(vec1.begin(),\nvec1.begin()+8, vec1.end(),
great).\n";
partial_sort(vec1.begin(), vec1.begin()+8, vec1.end(), great);
cout<<"Partially resorted (great) vector vec1 data:\n";
for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;
return 0;
}
```

Output:



```
vector vec1 data:
10 11 12 13 14 15 16 0 1 2 3 4 5

Operation: partial_sort(vec1.begin(),
vec1.begin()+ 5, vec1.end()).
Partially sorted vector vec1 data:
0 1 2 3 4 15 16 14 13 12 11 10 5

Operation: partial_sort(vec1.begin(),
vec1.begin()+4, vec1.end(), greater<int>()).
Partially resorted (greater) vector vec1 data:
16 15 14 13 0 1 2 3 4 12 11 10 5

Operation: partial_sort(vec1.begin(),
vec1.begin()+8, vec1.end(), great).
Partially resorted (great) vector vec1 data:
16 15 14 13 12 11 10 5 0 1 2 3 4
Press any key to continue
```

partial_sort_copy()

- Copies elements from a source range into a destination range where the source elements are ordered by either less than or another specified binary predicate.

```
template<class InputIterator, class RandomAccessIterator>
   RandomAccessIterator partial_sort_copy(
      InputIterator _First1,
      InputIterator _Last1,
      RandomAccessIterator _First2,
      RandomAccessIterator _Last2
   );
template<class InputIterator, class RandomAccessIterator, class BinaryPredicate>
   RandomAccessIterator partial_sort_copy(
      InputIterator _First1,
      InputIterator _Last1,
      RandomAccessIterator _First2,
      RandomAccessIterator _Last2,
      BinaryPredicate _Comp
   );
```

Parameters

Parameter	Description
_First1	An input iterator addressing the position of the first element in the source range.
_Last1	An input iterator addressing the position one past the final element in the source range.
_First2	A random-access iterator addressing the position of the first element in the sorted destination range.
_Last2	A random-access iterator addressing the position one past the final element in the sorted destination range.

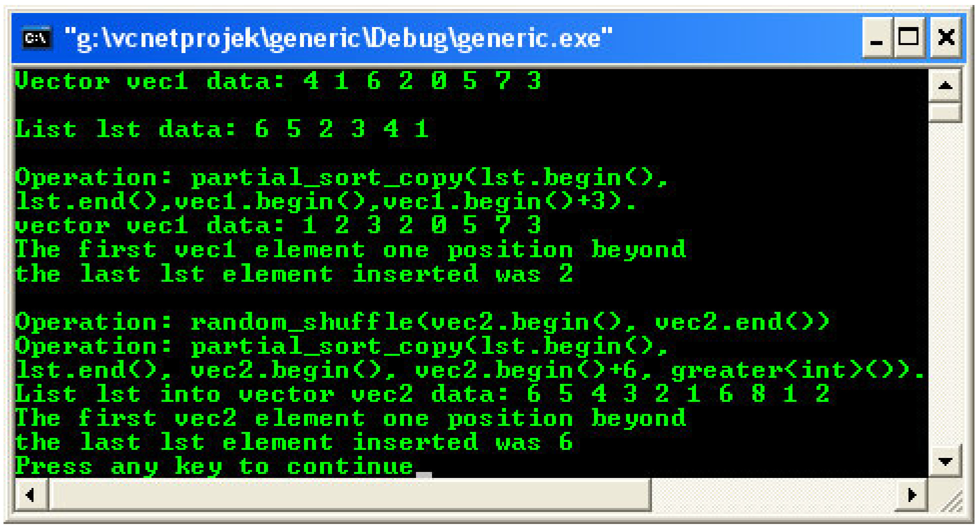
`_Comp`	User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns true when satisfied and false when not satisfied.

Table 35.15

- The return value is a random-access iterator addressing the element in the destination range one position beyond the last element inserted from the source range.
- The source and destination ranges must not overlap and must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.
- The binary predicate must provide a strict weak ordering so that elements that are not equivalent are ordered, but elements that are equivalent are not. Two elements are equivalent under less than, but not necessarily equal, if neither is less than the other.

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

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

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

random_shuffle(vec1.begin(), vec1.end());
lst.push_back(6);
lst.push_back(5);
lst.push_back(2);
lst.push_back(3);
lst.push_back(4);
lst.push_back(1);

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

cout<<"\nList lst data: ";
for(lst_Iter = lst.begin(); lst_Iter!= lst.end(); lst_Iter++)
cout<<*lst_Iter<<" ";
cout<<endl;

//Copying a partially sorted copy of lst into vec1
cout<<"\nOperation:
partial_sort_copy(lst.begin(),\nlst.end(),vec1.begin(),vec1.begin()+3).\n";
vector <int>::iterator result1;
result1 = partial_sort_copy(lst.begin(), lst.end(), vec1.begin(), vec1.begin()+3);
cout<<"vector vec1 data: ";
for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;
cout<<"The first vec1 element one position beyond"
<<"\nthe last lst element inserted was "<<*result1<<endl;

//Copying a partially sorted copy of lst into vec2
int j;
for(j = 0; j <= 9; j++)
vec2.push_back(j);

cout<<"\nOperation: random_shuffle(vec2.begin(), vec2.end())\n";
random_shuffle(vec2.begin(), vec2.end());

vector <int>::iterator result2;
cout<<"Operation: partial_sort_copy(lst.begin(),\nlst.end(), vec2.begin(),
vec2.begin()+6, greater<int>()).\n";
result2 = partial_sort_copy(lst.begin(), lst.end(), vec2.begin(), vec2.begin()+6,
greater<int>());
cout<<"List lst into vector vec2 data: ";
for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
cout<<*Iter2<<" ";
cout<<endl;
cout<<"The first vec2 element one position beyond"
<<"\nthe last lst element inserted was "<<*result2<<endl;
return 0;
}
```

Output:



```
Vector vec1 data: 4 1 6 2 0 5 7 3

List lst data: 6 5 2 3 4 1

Operation: partial_sort_copy(lst.begin(),
lst.end(),vec1.begin(),vec1.begin()+3).
vector vec1 data: 1 2 3 2 0 5 7 3
The first vec1 element one position beyond
the last lst element inserted was 2

Operation: random_shuffle(vec2.begin(), vec2.end())
Operation: partial_sort_copy(lst.begin(),
lst.end(), vec2.begin(), vec2.begin()+6, greater<int>()).
List lst into vector vec2 data: 6 5 4 3 2 1 6 8 1 2
The first vec2 element one position beyond
the last lst element inserted was 6
Press any key to continue
```

partition()

- Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it.

```
template<class BidirectionalIterator, class Predicate>
   BidirectionalIterator partition(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

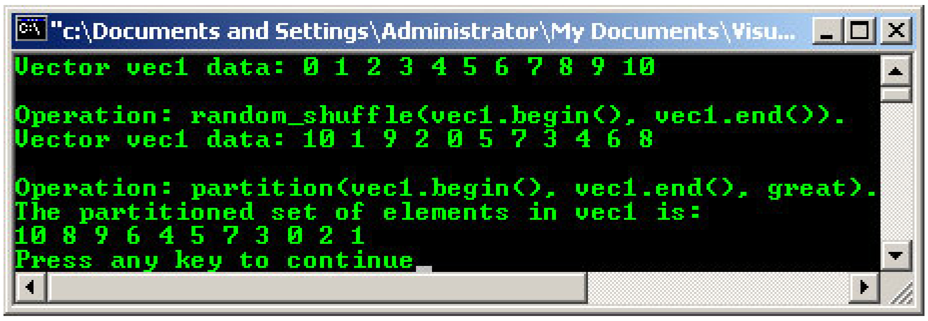
Parameter	Description
`_First`	A bidirectional iterator addressing the position of the first element in the range to be partitioned.
`_Last`	A bidirectional iterator addressing the position one past the final element in the range to be partitioned.
`_Comp`	User-defined predicate function object that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

Table 35.16

- 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 dereferenceable 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 `partition()` algorithm is not stable and does not guarantee that the relative ordering of equivalent elements will be preserved. The algorithm `stable_ partition()` does preserve this original ordering.
- The complexity is linear: there are `(_Last – _First)` applications of `_Comp` and at most `(_Last – _First)/2` swaps.

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

//user defined...
bool great(int value)
{return value >3;}

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

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

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

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

//Partition the range with predicate great
cout<<"\nOperation: partition(vec1.begin(), vec1.end(), great).\n";
partition(vec1.begin(), vec1.end(), great);
cout<<"The partitioned set of elements in vec1 is:\n";
for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;
return 0;
}
```

Output:



```
Vector vec1 data: 0 1 2 3 4 5 6 7 8 9 10

Operation: random_shuffle(vec1.begin(), vec1.end()).
Vector vec1 data: 10 1 9 2 0 5 7 3 4 6 8

Operation: partition(vec1.begin(), vec1.end(), great).
The partitioned set of elements in vec1 is:
10 8 9 6 4 5 7 3 0 2 1
Press any key to continue
```

pop_heap()

- Removes the largest element from the front of a heap to the next-to-last position in the range and then forms a new heap from the remaining elements.

```
template<class RandomAccessIterator>
   void pop_heap(
      RandomAccessIterator _First,
      RandomAccessIterator _Last
   );
template<class RandomAccessIterator, class BinaryPredicate>
   void pop_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 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 35.17

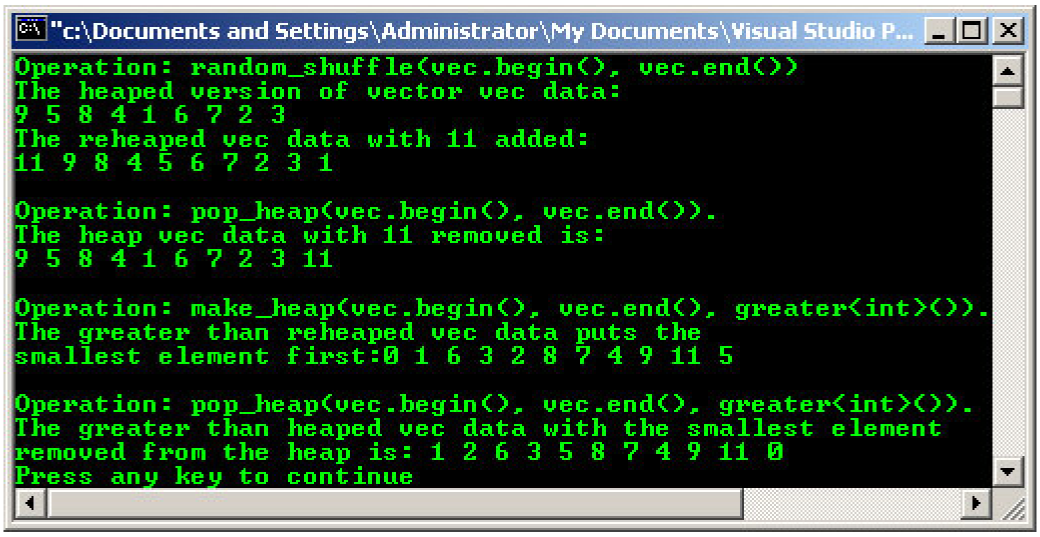
- The `pop_heap()` algorithm is the inverse of the operation performed by the `push_heap()` algorithm, in which an element at the next-to-last position of a range is added to a heap consisting of the prior elements in the range, in the case when the element being added to the heap is larger than any of the elements already in the heap.
- As mentioned before, 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 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 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, pop_heap()
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
using namespace std;

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

int i;
for(i = 1; i <= 9; i++)
vec.push_back(i);

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

//Add an element to the back of the heap
vec.push_back(11);
push_heap(vec.begin(), vec.end());
cout<<"The reheaped vec data with 11 added:\n";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

//Remove the largest element from the heap
cout<<"\nOperation: pop_heap(vec.begin(), vec.end()).\n";
pop_heap(vec.begin(), vec.end());
cout<<"The heap vec data with 11 removed is:\n";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

//Make vec a heap with greater-than ordering with a 0 element
cout<<"\nOperation: make_heap(vec.begin(), vec.end(), greater<int>()).\n";
make_heap(vec.begin(), vec.end(), greater<int>());
vec.push_back(0);
push_heap(vec.begin(), vec.end(), greater<int>());
cout<<"The greater than reheaped vec data puts the\nsmallest element first:";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

//Application of pop_heap to remove the smallest element
cout<<"\nOperation: pop_heap(vec.begin(), vec.end(), greater<int>()).\n";
pop_heap(vec.begin(), vec.end(), greater<int>());
cout<<"The greater than heaped vec data with the smallest element\nremoved from
the heap is: ";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;
return 0;
}
```

Output



"c:\Documents and Settings\Administrator\My Documents\Visual Studio P...

```
Operation: random_shuffle(vec.begin(), vec.end())
The heaped version of vector vec data:
9 5 8 4 1 6 7 2 3
The reheaped vec data with 11 added:
11 9 8 4 5 6 7 2 3 1

Operation: pop_heap(vec.begin(), vec.end()).
The heap vec data with 11 removed is:
9 5 8 4 1 6 7 2 3 11

Operation: make_heap(vec.begin(), vec.end(), greater<int>()).
The greater than reheaped vec data puts the
smallest element first:0 1 6 3 2 8 7 4 9 11 5

Operation: pop_heap(vec.begin(), vec.end(), greater<int>()).
The greater than heaped vec data with the smallest element
removed from the heap is: 1 2 6 3 5 8 7 4 9 11 0
Press any key to continue
```

prev_permutation()

- Reorders the elements in a range so that the original ordering is replaced by the lexicographically next greater permutation if it exists, where the sense of next may be specified with a binary predicate.

```
template<class BidirectionalIterator>
   bool prev_permutation(
      BidirectionalIterator _First,
      BidirectionalIterator _Last
   );
template<class BidirectionalIterator, class BinaryPredicate>
   bool prev_permutation(
      BidirectionalIterator _First,
      BidirectionalIterator _Last,
      BinaryPredicate _Comp
   );
```

Parameters

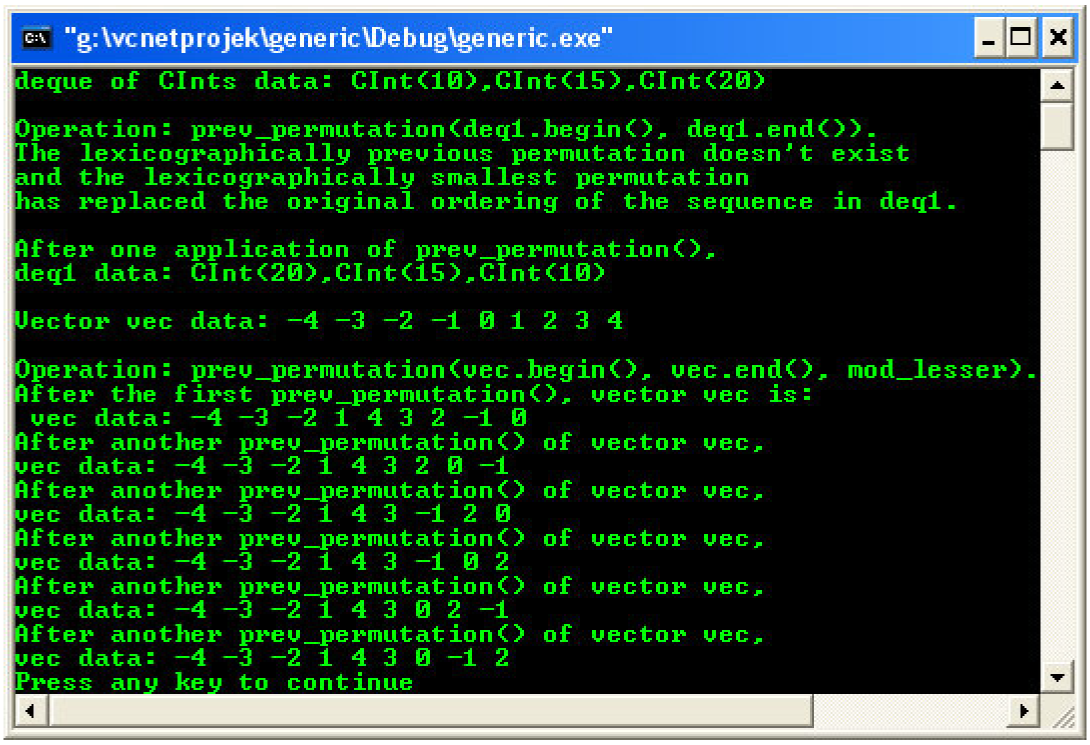
Parameter	Description
_First	A bidirectional iterator pointing to the position of the first element in the range to be permuted.
_Last	A bidirectional iterator pointing to the position one past the final element in the range to be permuted.
_Comp	User-defined predicate function object that defines the 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 35.18

- The return value is **true** if the lexicographically previous permutation exists and has replaced the original ordering of the range; otherwise **false**, in which case the ordering is transformed into the lexicographically largest permutation.
- The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.
- The default binary predicate is less than and the elements in the range must be less-than comparable to ensure that the next permutation is well defined.
- The complexity is linear, with at most `(_Last – _First)/2` swap.

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

class CInt;
ostream& operator<<(ostream& osIn, const CInt& rhs);

class CInt
{
public:
   CInt(int n = 0) : m_nVal(n){}
   CInt(const CInt& rhs) : m_nVal(rhs.m_nVal){}
   CInt& operator=(const CInt& rhs)
   {m_nVal = rhs.m_nVal; return *this;}
   bool operator<(const CInt& rhs) const
   {return (m_nVal < rhs.m_nVal);}
   friend ostream& operator<<(ostream& osIn, const CInt& rhs);

private:
   int m_nVal;
};

inline ostream& operator<<(ostream& osIn, const CInt& rhs)
{
   osIn<<"CInt("<<rhs.m_nVal<<")";
   return osIn;
}

//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()
{
//Reordering the elements of type CInt in a deque
//using the prev_permutation algorithm
CInt ci1 = 10, ci2 = 15, ci3 = 20;
bool deq1Result;
deque<CInt> deq1, deq2, deq3;
deque<CInt>::iterator d1_Iter;

deq1.push_back(ci1);
deq1.push_back(ci2);
deq1.push_back(ci3);

cout<<"deque of CInts data: ";
for(d1_Iter = deq1.begin(); d1_Iter != --deq1.end(); d1_Iter++)
cout<<*d1_Iter<<",";
d1_Iter = --deq1.end();
cout<<*d1_Iter<<endl;

cout<<"\nOperation: prev_permutation(deq1.begin(), deq1.end()).\n";
deq1Result = prev_permutation(deq1.begin(), deq1.end());
if(deq1Result)
cout<<"The lexicographically previous permutation "
<<"exists and has\nreplaced the original "
<<"ordering of the sequence in deq1."<<endl;
else
cout<<"The lexicographically previous permutation doesn't "
<<"exist\nand the lexicographically "
<<"smallest permutation\nhas replaced the "
<<"original ordering of the sequence in deq1."<<endl;

cout<<"\nAfter one application of prev_permutation(),\ndeq1 data: ";
for(d1_Iter = deq1.begin(); d1_Iter != --deq1.end(); d1_Iter++)
cout<<*d1_Iter<<",";
d1_Iter = --deq1.end();
cout<<*d1_Iter<<endl;

//Permutating vector elements with binary function mod_lesser
vector <int> vec;
vector <int>::iterator Iter1;

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

cout<<"\nVector vec data: ";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

cout<<"\nOperation: prev_permutation(vec.begin(), vec.end(), mod_lesser).\n";
prev_permutation(vec.begin(), vec.end(), mod_lesser);
cout<<"After the first prev_permutation(), vector vec is:\n vec data: ";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

int j = 1;
while (j <= 5)
{
prev_permutation(vec.begin(), vec.end(), mod_lesser);
cout<<"After another prev_permutation() of vector vec,\nvec data: ";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1 ++)
cout<<*Iter1<<" ";
cout<<endl;
j++;
}
return 0;
}
```

Output:



```
deque of CInts data: CInt(10),CInt(15),CInt(20)

Operation: prev_permutation(deq1.begin(), deq1.end()).
The lexicographically previous permutation doesn't exist
and the lexicographically smallest permutation
has replaced the original ordering of the sequence in deq1.

After one application of prev_permutation(),
deq1 data: CInt(20),CInt(15),CInt(10)

Vector vec data: -4 -3 -2 -1 0 1 2 3 4

Operation: prev_permutation(vec.begin(), vec.end(), mod_lesser).
After the first prev_permutation(), vector vec is:
 vec data: -4 -3 -2 1 4 3 2 -1 0
After another prev_permutation() of vector vec,
vec data: -4 -3 -2 1 4 3 2 0 -1
After another prev_permutation() of vector vec,
vec data: -4 -3 -2 1 4 3 -1 2 0
After another prev_permutation() of vector vec,
vec data: -4 -3 -2 1 4 3 -1 0 2
After another prev_permutation() of vector vec,
vec data: -4 -3 -2 1 4 3 0 2 -1
After another prev_permutation() of vector vec,
vec data: -4 -3 -2 1 4 3 0 -1 2
Press any key to continue
```

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

```
//*******algoiterswap.cpp*******
//algorithm, iter_swap()
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
using namespace std;

class CInt;
ostream& operator<<(ostream& osIn, const CInt& rhs);

class CInt
{
public:
CInt(int n = 0) : m_nVal(n){}
CInt(const CInt& rhs) : m_nVal(rhs.m_nVal){}
CInt& operator=(const CInt& rhs)
{ m_nVal = rhs.m_nVal; return *this;}
bool operator<(const CInt& rhs) const
{ return (m_nVal < rhs.m_nVal);}
friend ostream& operator<<(ostream& osIn, const CInt& rhs);

private:
int m_nVal;
};

inline ostream& operator<<(ostream& osIn, const CInt& rhs)
{
osIn<<"CInt(" <<rhs.m_nVal<< ")";
return osIn;
}

//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()
{
CInt c1 = 9, c2 = 12, c3 = 17;
deque<CInt> deq;
deque<CInt>::iterator deqIter;

deq.push_back(c1);
deq.push_back(c2);
deq.push_back(c3);

cout<<"The deque of CInts data is:\n";
for(deqIter = deq.begin(); deqIter != --deq.end(); deqIter++)
cout<<" "<<*deqIter<<",";
deqIter = --deq.end();
cout<<" "<<*deqIter<<endl;

//Exchanging first and last elements with iter_swap
iter_swap(deq.begin(), --deq.end());
cout<<"\nThe deque of CInts data with first and last\nelements swapped is: ";
for(deqIter = deq.begin(); deqIter != --deq.end(); deqIter++)
cout<<" "<<*deqIter<<",";
deqIter = --deq.end();
cout<<" "<<*deqIter<<endl;

//Swapping back first and last elements with swap
swap(*deq.begin(), *(deq.end() -1));

cout<<"\nThe deque of CInts data with first and last\nelements re swapped is: ";
for(deqIter = deq.begin(); deqIter != --deq.end(); deqIter++)
cout<<" "<<*deqIter<<",";
deqIter = --deq.end();
cout<<" "<<*deqIter<<endl;

//Swapping a vector element with a deque element
vector <int> vec;
vector <int>::iterator Iter1;
deque <int> deq1;
deque <int>::iterator deq1Iter;
int i;
for(i = 10; i <= 14; i++)
vec.push_back(i);

int j;
for(j = 16; j <= 20; j++)
deq1.push_back(j);

cout<<"\nVector vec data: ";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

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

iter_swap(vec.begin(), deq1.begin());
cout<<"\nAfter exchanging first elements,\nvector vec data is: ";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl<<"and deque deq1 data is: ";
for(deq1Iter = deq1.begin(); deq1Iter != deq1.end(); deq1Iter++)
cout<<*deq1Iter<<" ";
cout<<endl;
return 0;
}
```

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

```
The deque of CInts data is:
 CInt(9), CInt(12), CInt(17)

The deque of CInts data with first and last
elements swapped is:  CInt(17), CInt(12), CInt(9)

The deque of CInts data with first and last
elements re swapped is:  CInt(9), CInt(12), CInt(17)

Vector vec data: 10 11 12 13 14

Deque deq1 data: 16 17 18 19 20

After exchanging first elements,
vector vec data is: 16 11 12 13 14
and deque deq1 data is: 10 17 18 19 20
```

```
//******algopartition.cpp*******
//algorithm, partition()
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

//user defined...
bool great(int value)
{return value >3;}

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

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

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

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

//Partition the range with predicate great
cout<<"\nOperation: partition(vec1.begin(), vec1.end(), great).\n";
partition(vec1.begin(), vec1.end(), great);
cout<<"The partitioned set of elements in vec1 is:\n";
for(Iter1 = vec1.begin(); Iter1 != vec1.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;
return 0;
}
```

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

```
Vector vec1 data: 0 1 2 3 4 5 6 7 8 9 10

Operation: random_shuffle(vec1.begin(), vec1.end()).
Vector vec1 data: 4 10 7 8 0 5 2 1 6 9 3

Operation: partition(vec1.begin(), vec1.end(), great).
The partitioned set of elements in vec1 is:
4 10 7 8 9 5 6 1 2 0 3
```

---End of Algorithm part III---

---www.tenouk.com---

Further reading and digging:

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