

MODULE 34 --THE STL-- ALGORITHM PART II

My Training Period: hours

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

Abilities

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

34.1 <algorithm> Header File

- Defines Standard Template Library (STL) container template functions that perform algorithms. The following header must be included in your program.

```
#include <algorithm>
```
- The STL algorithms are generic because they can operate on a variety of data structures. The data structures that they can operate on included not only the STL container classes such as vector and list, but also program-defined data structures and arrays of elements that satisfy the requirements of a particular algorithm.
- STL algorithms achieve this level of generality by accessing and traversing the elements of a container indirectly through **iterators**.
- STL algorithms process **iterator ranges** that are typically specified by their beginning or ending positions.
- The ranges referred to must be valid in the sense that all pointers in the ranges must be de-referenceable and within the sequences of each range, the last position must be reachable from the first by incrementation.
- The STL algorithms extend the actions supported by the operations and member functions of each STL container and allow working, for example, with different types of container objects at the same time. Two **suffixes** have been used to convey information about the purpose of the algorithms.
 - The **_if** suffix indicates that the algorithm is used with function objects operating on the values of the elements rather than on the values of the elements themselves. The `find_if()` algorithm looks for elements whose values satisfy the criterion specified by a function object, and the `find()` algorithm searches for a particular value.
 - The **copy** suffix indicates that the algorithm not only manipulates the values of the elements but also **copies** the modified values into a destination range. For example, the `reverse()` algorithm reverses the order of the elements within a range, and the `reverse_copy()` algorithm also copies the result into a destination range.
- STL algorithms are often classified into groups that indicate something about their purpose or requirements.
- These include **modifying** algorithms that change the value of elements as compared with **non-modifying** algorithms that do not. **Mutating** algorithms change **the order of elements**, but not the values of their elements.
- **Removing** algorithms can eliminate elements from a range or a copy of a range.
- **Sorting** algorithms reorder the elements in a range in various ways and sorted range algorithms only act on algorithms whose elements have been sorted in a particular way.
- The STL **numeric** algorithms that are provided for numerical processing have their own header file `<numeric>`, and function objects and adaptors are defined in the header `<functional>`.
- **Function objects** that return Boolean values are known as **predicates**. The default **binary predicate** is the comparison operator `<`.
- In general, the elements being ordered need to be **less than** comparable so that, given any two elements, it can 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 among the nonequivalent elements.

34.2 <algorithm> Member Functions

- The following table is a list of the member functions available in <algorithm>. So many huh!

Member functions

Member function	Description
adjacent_find()	Searches for two adjacent elements that are either equal or satisfy a specified condition.
binary_search()	Tests whether there is an element in a sorted range that is equal to a specified value or that is equivalent to it in a sense specified by a binary predicate.
copy()	Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a forward direction.
copy_backward()	Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a backward direction.
count()	Returns the number of elements in a range whose values match a specified value.
count_if()	Returns the number of elements in a range whose values match a specified condition.
equal()	Compares two ranges element by element either for equality or equivalence in a sense specified by a binary predicate.
equal_range()	Finds a pair of positions in an ordered range, the first less than or equivalent to the position of a specified element and the second greater than the element's position, where the sense of equivalence or ordering used to establish the positions in the sequence may be specified by a binary predicate.
fill()	Assigns the same new value to every element in a specified range.
fill_n()	Assigns a new value to a specified number of elements in a range beginning with a particular element.
find()	Locates the position of the first occurrence of an element in a range that has a specified value.
find_end()	Looks in a range for the last subsequence that is identical to a specified sequence or that is equivalent in a sense specified by a binary predicate.
find_first_of()	Searches for the first occurrence of any of several values within a target range or for the first occurrence of any of several elements that are equivalent in a sense specified by a binary predicate to a specified set of the elements.
find_if()	Locates the position of the first occurrence of an element in a range that satisfies a specified condition.
for_each()	Applies a specified function object to each element in a forward order within a range and returns the function object.
generate()	Assigns the values generated by a function object to each element in a range.
generate_n()	Assigns the values generated by a function object to a specified number of element in a range and returns to the position one past the last assigned value.
includes()	Tests whether one sorted range contains all the elements contained in a second sorted range, where the ordering or equivalence criterion between elements may be specified by a binary predicate.
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.
iter_swap()	Exchanges two values referred to by a pair of specified iterators.
lexicographical_compare()	Compares element by element between two sequences to determine which is lesser of the two.
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.
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.

<code>max()</code>	Compares two objects and returns the larger of the two, where the ordering criterion may be specified by a binary predicate.
<code>max_element()</code>	Finds the first occurrence of largest element in a specified range where the ordering criterion may be specified by a binary predicate.
<code>merge()</code>	Combines all the elements from two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
<code>min()</code>	Compares two objects and returns the lesser of the two, where the ordering criterion may be specified by a binary predicate.
<code>min_element()</code>	Finds the first occurrence of smallest element in a specified range where the ordering criterion may be specified by a binary predicate.
<code>mismatch()</code>	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.
<code>next_permutation()</code>	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.
<code>nth_element()</code>	Partitions a range of elements, correctly locating the <i>n</i> 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.
<code>partial_sort()</code>	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.
<code>partial_sort_copy()</code>	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.
<code>partition()</code>	Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it.
<code>pop_heap()</code>	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.
<code>prev_permutation()</code>	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.
<code>push_heap()</code>	Adds an element that is at the end of a range to an existing heap consisting of the prior elements in the range.
<code>random_shuffle()</code>	Rearranges a sequence of <i>N</i> elements in a range into one of <i>N</i> ! possible arrangements selected at random.
<code>remove()</code>	Eliminates a specified value from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.
<code>remove_copy()</code>	Copies elements from a source range to a destination range, except that elements of a specified value are not copied, without disturbing the order of the remaining elements and returning the end of a new destination range.
<code>remove_copy_if()</code>	Copies elements from a source range to a destination range, except that satisfying a predicate are not copied, without disturbing the order of the remaining elements and returning the end of a new destination range.
<code>remove_if()</code>	Eliminates elements that satisfy a predicate from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.
<code>replace()</code>	Examines each element in a range and replaces it if it matches a specified value.
<code>replace_copy()</code>	Examines each element in a source range and replaces it if it matches a specified value while copying the result into a new destination range.
<code>replace_copy_if()</code>	Examines each element in a source range and replaces it if it satisfies a specified predicate while copying the result into a new destination range.
<code>replace_if()</code>	Examines each element in a range and replaces it if it satisfies a specified predicate.
<code>reverse()</code>	Reverses the order of the elements within a range.
<code>reverse_copy()</code>	Reverses the order of the elements within a source range while copying them into a destination range

<code>rotate()</code>	Exchanges the elements in two adjacent ranges.
<code>rotate_copy()</code>	Exchanges the elements in two adjacent ranges within a source range and copy the result to a destination range.
<code>search()</code>	Searches for the first occurrence of a sequence within a target range whose elements are equal to those in a given sequence of elements or whose elements are equivalent in a sense specified by a binary predicate to the elements in the given sequence.
<code>search_n()</code>	Searches for the first subsequence in a range that of a specified number of elements having a particular value or a relation to that value as specified by a binary predicate.
<code>set_difference()</code>	Unites all of the elements that belong to one sorted source range, but not to a second sorted source range, into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
<code>set_intersection()</code>	Unites all of the elements that belong to both sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
<code>set_symmetric_difference()</code>	Unites all of the elements that belong to one, but not both, of the sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
<code>set_union()</code>	Unites all of the elements that belong to at least one of two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
<code>sort()</code>	Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate.
<code>sort_heap()</code>	Converts a heap into a sorted range.
<code>stable_partition()</code>	Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it, preserving the relative order of equivalent elements.
<code>stable_sort()</code>	Arranges the elements in a specified range into a non-descending order or according to an ordering criterion specified by a binary predicate and preserves the relative ordering of equivalent elements.
<code>swap()</code>	Exchanges the values of the elements between two types of objects, assigning the contents of the first object to the second object and the contents of the second to the first.
<code>swap_ranges()</code>	Exchanges the elements of one range with the elements of another, equal sized range.
<code>transform()</code>	Applies a specified function object to each element in a source range or to a pair of elements from two source ranges and copies the return values of the function object into a destination range.
<code>unique()</code>	Removes duplicate elements that are adjacent to each other in a specified range.
<code>unique_copy()</code>	Copies elements from a source range into a destination range except for the duplicate elements that are adjacent to each other.
<code>upper_bound()</code>	Finds the position of the first element in an ordered range that has a value that is greater than a specified value, where the ordering criterion may be specified by a binary predicate.

Table 34.1

- The following section presents some of the program examples using the member functions. Notice the using of the containers and iterators in the program examples and in the function and class templates.

`adjacent_find()`

- Searches for two adjacent elements that are either equal or satisfy a specified condition.

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

Parameters

Parameter	Description
<i>_First</i>	A forward iterator addressing the position of the first element in the range to be searched.
<i>_Last</i>	A forward iterator addressing the position one past the final element in the range to be searched
<i>_Comp</i>	The binary predicate giving the condition to be satisfied by the values of the adjacent elements in the range being searched.

Table 34.2

- The return value is a forward iterator to the first element of the adjacent pair that are either equal to each other (in the first version) or that satisfy the condition given by the binary predicate (in the second version), provided that such a pair of elements is found. Otherwise, an iterator pointing to *_Last* is returned.
- The `adjacent_find()` algorithm is a non-mutating sequence algorithm. 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 match between elements must impose an equivalence relation between its operands.

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

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

int main()
{
    list<int> lst;
    list<int>::iterator Iter;
    list<int>::iterator result1, result2;
    lst.push_back(14);
    lst.push_back(17);
    lst.push_back(31);
    lst.push_back(31);
    lst.push_back(10);
    lst.push_back(20);

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

    result1 = adjacent_find(lst.begin(), lst.end());
    if(result1 == lst.end())
        cout<<"There are not two adjacent elements that are equal."<<endl;
    else
        cout<<"There are two adjacent elements that are equal."
            <<"\nThey have a value of "<<*(result1)<<endl;

    result2 = adjacent_find(lst.begin(), lst.end(), twice);
    if(result2 == lst.end())
        cout<<"\nThere are no two adjacent elements where the "
            <<"second is twice the first."<<endl;
    else
        {cout<<"\nThere are two adjacent elements\nwhere "
            <<"the second is twice the first."
            <<"\nThey have values of "<<*(result2++)<<endl;
        }
    cout<<" & "<<*(result2)<<endl;
    return 0;
}
```

Output:

binary_search()

- Tests whether there is an element in a sorted range that is equal to a specified value or that is equivalent to it in a sense specified by a binary predicate.

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

Parameters

Parameter	Description
<i>_First</i>	A forward iterator addressing the position of the first element in the range to be searched.
<i>_Last</i>	A forward iterator addressing the position one past the final element in the range to be searched.
<i>_Val</i>	The value required to be matched by the value of the element or that must satisfy the condition with the element value specified by the binary predicate.
<i>_Comp</i>	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 34.3

- The return value is true if an element is found in the range that is equal or equivalent to the specified value; otherwise, false.
- The sorted source range referenced must be valid; all pointers must be dereferenceable 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 `binary_search()` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.
- The source ranges are not modified by `binary_search()`.
- The value types of the forward iterators need to 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, binary_search()
#include <list>
#include <vector>
#include <algorithm>
#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()
{
    list<int> lst;
    list<int>::iterator Iter;
    bool b1, b2;

    lst.push_back(13);
    lst.push_back(23);
    lst.push_back(10);
    lst.push_back(33);
    lst.push_back(35);
    lst.push_back(9);

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

    b1 = binary_search(lst.begin(), lst.end(), 10);
    if(b1)
        cout<<"\nThere is an element in list lst with\na value equal to 10."<<endl;
    else
        cout<<"\nThere is no element in list lst with\na value equal to 10."<<endl;

    b2 = binary_search(lst.begin(), lst.end(), 13, greater<int>());
    if(b2)
        cout<<"\nThere is an element in list lst with a\nvalue equivalent to 13 "
        <<"under greater than."<<endl;
    else
        cout<<"\nNo element in list lst with a\nvalue equivalent to 13 "
        <<"under greater than."<<endl;

    //a binary_search under the user-defined binary predicate mod_lesser
    vector<int> vec;
    vector<int>::iterator Iter1;
    int i;
    for(i = -3; i <= 5; i++)
        vec.push_back(i);

    sort(vec.begin(), vec.end(), mod_lesser);

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

    bool b3 = binary_search(vec.begin(), vec.end(), -2, mod_lesser);
    if(b3)
        cout<<"\nThere is an element with a value\nequivalent to -2 "
        <<"under mod_lesser()."<<endl;
    else
        cout<<"\nThere is no element with a value\nequivalent to -2 "
        <<"under mod_lesser()."<<endl;
    return 0;
}

```

Output:

copy()

- Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a forward direction.

```
template<class InputIterator, class OutputIterator>
OutputIterator copy
(
    InputIterator _First,
    InputIterator _Last,
    OutputIterator _DestBeg
);
```

Parameters

Parameter	Description
<i>_First</i>	An input iterator addressing the position of the first element in the source range.
<i>_Last</i>	An input iterator addressing the position that is one past the final element in the source range.
<i>_DestBeg</i>	An output iterator addressing the position of the first element in the destination range.

Table 34.4

- The return value is an output iterator addressing the position that is one past the final element in the destination range, that is, the iterator addresses $_Result + (_Last - _First)$.
- The source range must be valid and there must be sufficient space at the destination to hold all the elements being copied.
- Because the algorithm copies the source elements in order beginning with the first element, the destination range can overlap with the source range provided the *_First* position of the source range is not contained in the destination range.
- `copy()` can be used to shift elements to the left but not the right, unless there is no overlap between the source and destination ranges. To shift to the right any number of positions, use the `copy_backward()` algorithm.
- The `copy()` algorithm only modifies values pointed to by the iterators, assigning new values to elements in the destination range. It cannot be used to create new elements and cannot insert elements into an empty container directly.

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

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

    int i;
    for(i = 0; i <= 5; i++)
```



```

vec1.push_back(i);

int j;
for(j = 10; j <= 20; j++)
    vec2.push_back(j);

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

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

//To copy the first 4 elements of vec1 into the middle of vec2
copy(vec1.begin(), vec1.begin() + 4, vec2.begin() + 5);

cout<<"vec2 with vec1 insert data: ";
for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
    cout<<*Iter2<<" ";
cout<<endl;

//To shift the elements inserted into vec2 two positions
//to the left
copy(vec2.begin()+4, vec2.begin() + 7, vec2.begin() + 2);

cout<<"vec2 with shifted insert data: ";
for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
    cout<<*Iter2<<" ";
cout<<endl;
return 0;
}

```

Output :

```

c:\Documents and Settings\Administrator\My Documents\Visual St...
vec1 data: 0 1 2 3 4 5
vec2 data: 10 11 12 13 14 15 16 17 18 19 20
vec2 with vec1 insert data: 10 11 12 13 14 0 1 2 3 19 20
vec2 with shifted insert data: 10 11 14 0 1 0 1 2 3 19 20
Press any key to continue

```

copy_backward()

- Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a backward direction.

```

template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward
(
    BidirectionalIterator1 _First,
    BidirectionalIterator1 _Last,
    BidirectionalIterator2 _DestEnd
);

```

Parameters

Parameter	Description
<i>_First</i>	A bidirectional iterator addressing the position of the first element in the source range.
<i>_Last</i>	A bidirectional iterator addressing the position that is one past the final element in the source range.
<i>_DestEnd</i>	A bidirectional iterator addressing the position of the one past the final element in the destination range.

Table 34.5

- The return value is an output iterator addressing the position that is one past the final element in the destination range, that is, the iterator addresses $_DestEnd - (_Last - _First)$.

- The source range must be valid and there must be sufficient space at the destination to hold all the elements being copied.
- The `copy_backward()` algorithm imposes more stringent requirements than that the `copy` algorithm. Both its input and output iterators must be bidirectional.
- The `copy_backward()` algorithm is the only STL algorithm designating the output range with an iterator pointing to the end of the destination range.
- Because the algorithm copies the source elements in order beginning with the last element, the destination range can overlap with the source range provided the `_Last` position of the source range is not contained in the destination range.
- `copy()` can be used to shift elements to the right but not the left, unless there is no overlap between the source and destination ranges. To shift to the left any number of positions, use the `copy()` algorithm.
- The `copy_backward()` algorithm only modifies values pointed to by the iterators, assigning new values to elements in the destination range. It cannot be used to create new elements and cannot insert elements into an empty container directly.

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

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

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

    int j;
    for(j = 0; j <= 10; j++)
        vec2.push_back(j);

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

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

    //To copy_backward the first 4 elements of vec1 into the middle of vec2
    copy_backward(vec1.begin(), vec1.begin() + 4, vec2.begin() + 8);

    cout<<"vec2 with vec1 insert data: ";
    for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
        cout<<*Iter2<<" ";
    cout<<endl;

    //To shift the elements inserted into vec2 two positions
    //to the right
    copy_backward(vec2.begin()+4, vec2.begin()+7, vec2.begin()+9);

    cout<<"vec2 with shifted insert data: ";
    for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
        cout<<*Iter2<<" ";
    cout<<endl;
    return 0;
}
```

Output :

count()

- Returns the number of elements in a range whose values match a specified value.

```
template<class InputIterator, class Type>
typename iterator_traits<InputIterator>::difference_type count(
    InputIterator _First,
    InputIterator _Last,
    const Type& _Val
);
```

Parameters

Parameter	Description
<i>_First</i>	An input iterator addressing the position of the first element in the range to be traversed.
<i>_Last</i>	An input iterator addressing the position one past the final element in the range to be traversed.
<i>_Val</i>	The value of the elements to be counted.

Table 34.6

- The return value is the difference type of the `InputIterator` that counts the number of elements in the range `[_First, _Last)` that have value `_Val`.
- The operator `==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.
- This algorithm is generalized to count elements that satisfy any predicate with the template function `count_if()`.

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

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

    vec.push_back(12);
    vec.push_back(22);
    vec.push_back(12);
    vec.push_back(31);
    vec.push_back(12);
    vec.push_back(33);

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

    int result;
    cout<<"\nOperation: count(vec.begin(), vec.end(), 12)\n";
    result = count(vec.begin(), vec.end(), 12);
    cout<<"The number of 12s in vec is: "<<result<<endl;
    return 0;
}
```

Output:

count_if()

- Returns the number of elements in a range whose values satisfy a specified condition.

```
template<class InputIterator, class Predicate>
typename iterator_traits<InputIterator>::difference_type count_if(
    InputIterator _First,
    InputIterator _Last,
    Predicate _Pred
);
```

Parameters

Parameter	Description
<i>_First</i>	An input iterator addressing the position of the first element in the range to be searched.
<i>_Last</i>	An input iterator addressing the position one past the final element in the range to be searched.
<i>_Pred</i>	User-defined predicate function object that defines the condition to be satisfied if an element is to be counted. A predicate takes single argument and returns true or false.

Table 34.7

- The return value is the number of elements that satisfy the condition specified by the predicate.
- This template function is a generalization of the algorithm `count ()`, replacing the predicate "equals a specific value" with any predicate.

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

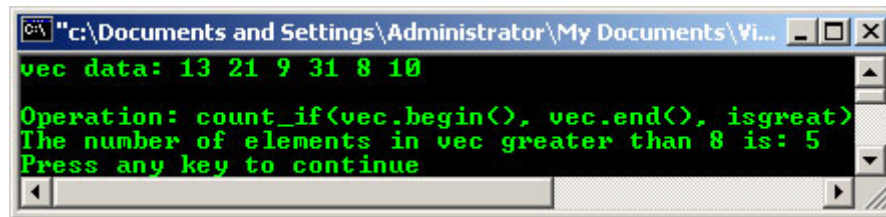
bool isgreat(int value)
{return value >8;}

int main()
{
    vector <int> vec;
    vector <int>::iterator Iter;
    vec.push_back(13);
    vec.push_back(21);
    vec.push_back(9);
    vec.push_back(31);
    vec.push_back(8);
    vec.push_back(10);

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

    int result1;
    cout<<"\nOperation: count_if(vec.begin(), vec.end(), isgreat)\n";
    result1 = count_if(vec.begin(), vec.end(), isgreat);
    cout<<"The number of elements in vec greater than 8 is: "<<result1<<endl;
    return 0;
}
```

Output:



```
C:\Documents and Settings\Administrator\My Documents\Vi...
vec data: 13 21 9 31 8 10
Operation: count_if(vec.begin(), vec.end(), isgreat)
The number of elements in vec greater than 8 is: 5
Press any key to continue
```

count_if()

- The following example is to show how to use the count_if() STL function in Microsoft Visual C++ as an implementation dependent.

```
template<class InputIterator, class Predicate> inline
size_t count_if(
    InputIterator First,
    InputIterator Last,
    Predicate P
)
```

- The class/parameter names in the prototype do not match the version in the header file. Some have been modified to improve readability.
- The count_if() algorithm counts the number of elements in the range [First, Last) that cause the predicate to return **true** and returns the number of elements for which the predicate was true.

```
// countif()
//
// Functions:
//   count_if - Count items in a range that satisfy a predicate.
//   begin    - Returns an iterator that points to the first element in
//              a sequence.
//   end      - Returns an iterator that points one past the end of a
//              sequence.
#include <iostream>
#include <algorithm>
#include <functional>
#include <string>
#include <vector>

using namespace std;

//Return true if string str starts with letter 'C'
int MatchFirstChar(const string& str)
{
    string s("C");
    return s == str.substr(0, 1);
}

int main()
{
    const int VECTOR_SIZE = 110;

    //Define a template class vector of strings
    typedef vector<string> StringVector;

    //Define an iterator for template class vector of strings
    typedef StringVector::iterator StringVectorIt;

    //vector containing names
    StringVector NamesVect(VECTOR_SIZE);
    StringVectorIt start, end, it;
    //stores count of elements that match value.
    ptrdiff_t result = 0;
    //Initialize vector NamesVect
    NamesVect[0] = "Learn";
    NamesVect[1] = "C";
    NamesVect[2] = "and";
    NamesVect[3] = "C++";
    NamesVect[4] = "also";
    NamesVect[5] = "Visual";
    NamesVect[6] = "C++";
    NamesVect[7] = "and";
    NamesVect[8] = "C++";
    NamesVect[9] = ".Net";
```

```

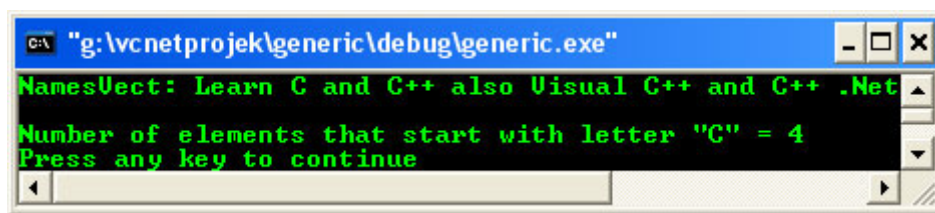
//location of first element of NamesVect
start = NamesVect.begin();
//one past the location last element of NamesVect
end = NamesVect.end();
//print content of NamesVect
cout<<"NamesVect: ";
for(it = start; it != end; it++)
    cout<<*it<<" ";
cout<<endl;

//Count the number of elements in the range [first, last +1)
//that start with letter 'C'
result = count_if(start, end, MatchFirstChar);

//print the count of elements that start with letter 'S'
cout<<"Number of elements that start with letter \"C\" = "<<result<<endl;
}

```

Output:



equal()

- Compares two ranges element by element either for equality or equivalence in a sense specified by a binary predicate.

```

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

```

Parameters

Parameter	Description
<i>_First1</i>	An input iterator addressing the position of the first element in the first range to be tested.
<i>_Last1</i>	An input iterator addressing the position one past the final element in the first range to be tested.
<i>_First2</i>	An input iterator addressing the position of the first element in the second range to be tested.
<i>_Comp</i>	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 34.8

- The return value is true if and only if the ranges are identical or equivalent under the binary predicate when compared element by element; otherwise, false.
- 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, equal()
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

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

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

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

    int j;
    for(j = 0; j <= 5; j++)
        vec2.push_back(j);

    int k;
    for(k = 10; k <= 15; k++)
        vec3.push_back(k);

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

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

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

    //Testing vec1 and vec2 for equality based on equality
    bool b;
    b = equal(vec1.begin(), vec1.end(), vec2.begin());

    if(b)
        cout<<"The vectors vec1 and vec2 are equal based on equality."<<endl;
    else
        cout<<"The vectors vec1 and vec2 are not equal based on equality."<<endl;

    //Testing vec1 and vec3 for equality based on equality
    bool c;
    c = equal(vec1.begin(), vec1.end(), vec3.begin());

    if(c)
        cout<<"The vectors vec1 and vec3 are equal based on equality."<<endl;
    else
        cout<<"The vectors vec1 and vec3 are not equal based on equality."<<endl;

    //Testing vec1 and vec3 for equality based on twice
    bool d;
    d = equal(vec1.begin(), vec1.end(), vec3.begin(), twice);

    if(d)
        cout<<"The vectors vec1 and vec3 are equal based on twice."<<endl;
    else
        cout<<"The vectors vec1 and vec3 are not equal based on twice."<<endl;
    return 0;
}

```

Output:

```

C:\Documents and Settings\Administrator\My Documents\Visual Stu...
vec1 data: 10 11 12 13 14 15
vec2 data: 0 1 2 3 4 5
vec3 data: 10 11 12 13 14 15
The vectors vec1 and vec2 are not equal based on equality.
The vectors vec1 and vec3 are equal based on equality.
The vectors vec1 and vec3 are not equal based on twice.
Press any key to continue

```

equal_range()

- Finds a pair of positions in an ordered range, the first less than or equivalent to the position of a specified element and the second greater than the element's position, where the sense of equivalence or ordering used to establish the positions in the sequence may be specified by a binary predicate.

```

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

```

Parameters

Parameter	Description
<i>_First</i>	A forward iterator addressing the position of the first element in the range to be searched.
<i>_Last</i>	A forward iterator addressing the position one past the final element in the range to be searched.
<i>_Val</i>	The value in the ordered range that needs to be equivalent to the value of the element addressed by the first component of the pair returned and that needs to be less than the value of the element addressed by the second component of that pair returns.
<i>_Comp</i>	User-defined predicate function object that is true when the left-hand argument is less than the right-hand argument. The user-defined predicate function should return false when its arguments are equivalent.

Table 34.9

- The return value is a pair of forward iterators addressing two positions in an ordered range in which the first component of the pair refers to the position where an element is or would be if it had a value that is less than or equivalent to a specified value and the second component of the pair refers to the first position where an element has a value that is greater than the value specified, where the sense of equivalence or ordering may be specified by a binary predicate.
- Alternatively, the pair of forward iterators may be described as specify a subrange, contained within the range searched, in which all of the elements are equivalent to the specified value in the sense defined by the binary predicate used.
- The first component of the pair of the algorithm returns `lower_bound()`, and the second component returns `upper_bound()`.
- The subrange defined by the pair of iterators returned by the `equal_range()` algorithm contains the equivalence class, in the standard set-theoretic sense, of the element whose value is specified as a parameter.
- 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 `equal_range()` 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, equal_range()
#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;
    pair< vector<int>::iterator, vector<int>::iterator > Result1, Result2, Result3;

    //Constructing vectors vec1 with default less than ordering
    int i;
    for(i = -2; i <= 4; i++)
        vec1.push_back(i);

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

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

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

    cout<<"\nvec2 data with range sorted by the binary predicate greater than is:\n";
    for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
        cout<<*Iter2<<" ";
    cout<<endl;

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

    cout<<"\nvec3 data with range sorted by the binary predicate mod_lesser is:\n";
    for(Iter3 = vec3.begin(); Iter3 != vec3.end(); Iter3++)
        cout<<*Iter3<<" ";
    cout<<"\n\n";

    //equal_range of 4 in vec1 with default binary predicate less <int>()
    Result1 = equal_range(vec1.begin(), vec1.end(), 4);
    cout<<"lower_bound in vec1 for the element with a value of 4 is:
    "<<*Result1.first<<endl;
    cout<<"upper_bound in vec1 for the element with a value of 4 is:
    "<<*Result1.second<<endl;
    cout<<"The equivalence class for the element with a value of 4 in \nvec1 includes
    the elements: ";
    for(Iter1 = Result1.first; Iter1 != Result1.second; Iter1++)
        cout<<*Iter1<<" ";
    cout<<endl<<endl;

    //equal_range of 4 in vec2 with the binary predicate greater <int>()
```

```

Result2 = equal_range(vec2.begin(), vec2.end(), 4, greater<int>());
cout<<"lower_bound in vec2 for the element with a value of 4 is:
"<<*Result2.first<<endl;
cout<<"upper_bound in vec2 for the element with a value of 4 is:
"<<*Result2.second<<endl;
cout<<"The equivalence class for the element with a value of 4 in"
        <<"\n vec2 includes the elements: ";
for(Iter2 = Result2.first; Iter2 != Result2.second; Iter2++)
cout<<*Iter2<<" ";
cout<<endl<<endl;

//equal_range of 4 in vec3 with the binary predicate mod_lesser
Result3 = equal_range(vec3.begin(), vec3.end(), 4, mod_lesser);
cout<<"lower_bound in vec3 for the element with a value of 4 is:
"<<*Result3.first<<endl;
cout<<"upper_bound in vec3 for the element with a value of 4 is:
"<<*Result3.second<<endl;
cout<<"equivalence class for the element with a value of 4 in \nvec3 includes the
elements: ";
for(Iter3 = Result3.first; Iter3 != Result3.second; Iter3++)
cout<<*Iter3<<" ";
cout<<endl<<endl;
return 0;
}

```

Output:

```

g:\vcnetprojek\generic\Debug\generic.exe
vec1 data with range sorted by the binary predicate less than is:
-2 -1 0 1 1 2 2 3 3 4 4 5
vec2 data with range sorted by the binary predicate greater than is:
5 4 4 3 3 2 2 1 1 0 -1 -2
vec3 data with range sorted by the binary predicate mod_lesser is:
0 -1 1 1 -2 2 2 3 3 4 4 5
lower_bound in vec1 for the element with a value of 4 is: 4
upper_bound in vec1 for the element with a value of 4 is: 5
The equivalence class for the element with a value of 4 in
vec1 includes the elements: 4 4
lower_bound in vec2 for the element with a value of 4 is: 4
upper_bound in vec2 for the element with a value of 4 is: 3
The equivalence class for the element with a value of 4 in
vec2 includes the elements: 4 4
lower_bound in vec3 for the element with a value of 4 is: 4
upper_bound in vec3 for the element with a value of 4 is: 5
equivalence class for the element with a value of 4 in
vec3 includes the elements: 4 4
Press any key to continue

```

fill()

- Assigns the same new value to every element in a specified range.

```

template<class ForwardIterator, class Type>
void fill(
    ForwardIterator _First,
    ForwardIterator _Last,
    const Type& _Val
);

```

Parameters

Parameter	Description
<i>_First</i>	A forward iterator addressing the position of the first element in the range to be traversed.
<i>_Last</i>	A forward iterator addressing the position one past the final element in the range to be traversed.

<i>_Val</i>	The value to be assigned to elements in the range [<i>_First</i> , <i>_Last</i>).
-------------	---

Table 34.10

- The destination range must be valid; all pointers must be de-referenceable, and the last position is reachable from the first by incrementation. The complexity is linear with the size of the range.

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

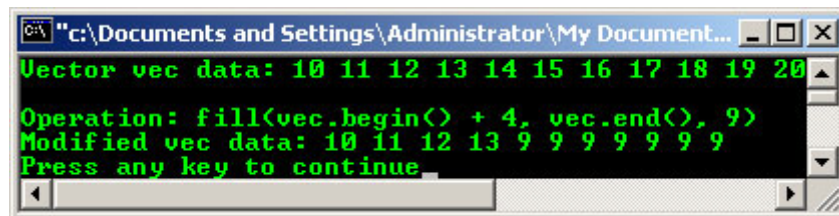
int main()
{
    vector <int> vec;
    vector <int>::iterator Iter1;

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

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

    //Fill the last 4 positions with a value of 9
    cout<<"\nOperation: fill(vec.begin() + 4, vec.end(), 9)\n";
    fill(vec.begin() + 4, vec.end(), 9);
    cout<<"Modified vec data: ";
    for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
        cout<<*Iter1<<" ";
    cout<<endl;
    return 0;
}
```

Output:



fill_n()

- Assigns a new value to a specified number of elements in a range beginning with a particular element.

```
template<class OutputIterator, class Size, class Type>
void fill_n(
    OutputIterator _First,
    Size _Count,
    const Type& _Val
);
```

Parameters

Parameter	Description
<i>_First</i>	An output iterator addressing the position of the first element in the range to be assigned the value <i>_Val</i> .
<i>_Count</i>	A signed or unsigned integer type specifying the number of elements to be assigned the value.
<i>_Val</i>	The value to be assigned to elements in the range [<i>_First</i> , <i>_First</i> + <i>_Count</i>).

Table 34.11

- The destination range must be valid; all pointers must be de-referenceable, and the last position is reachable from the first by incrementation. The complexity is linear with the size of the range.

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

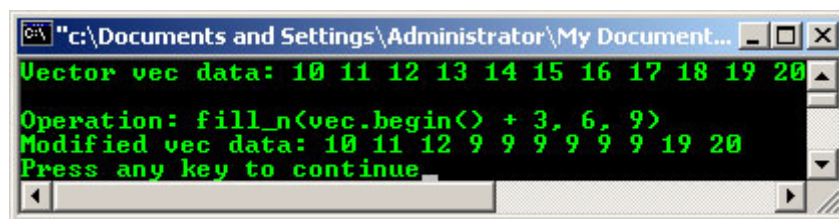
int main()
{
    vector <int> vec;
    vector <int>::iterator Iter1;

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

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

    //Fill the last 3 positions for 6 position with a value of 9
    cout<<"\nOperation: fill_n(vec.begin() + 3, 6, 9)\n";
    fill_n(vec.begin() + 3, 6, 9);
    cout<<"Modified vec data: ";
    for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
        cout<<*Iter1<<" ";
    cout<<endl;
    return 0;
}
```

Output:



find()

- Locates the position of the first occurrence of an element in a range that has a specified value.

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

Parameters

Parameter	Description
<i>_First</i>	An input iterator addressing the position of the first element in the range to be searched for the specified value.
<i>_Last</i>	An input iterator addressing the position one past the final element in the range to be searched for the specified value.
<i>_Val</i>	The value to be searched for.

Table 34.12

- The return value is an input iterator addressing the first occurrence of the specified value in the range being searched. If no such value exists in the range, the iterator returned addresses the last position of the range, one past the final element.
- The operator== used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

```

//algorithm, find()
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

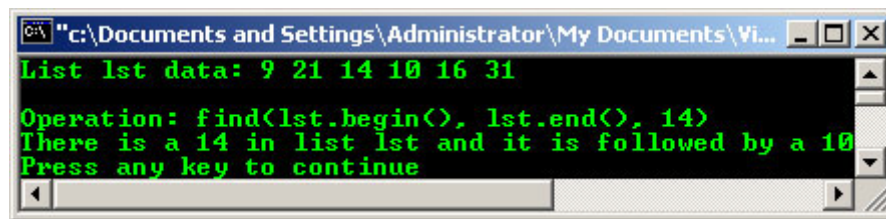
int main()
{
    list<int> lst;
    list<int>::iterator Iter;
    list<int>::iterator result;
    lst.push_back(9);
    lst.push_back(21);
    lst.push_back(14);
    lst.push_back(10);
    lst.push_back(16);
    lst.push_back(31);

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

    cout<<"\nOperation: find(lst.begin(), lst.end(), 14)\n";
    result = find(lst.begin(), lst.end(), 14);
    if(result == lst.end())
        cout<<"There is no 14 in list lst."<<endl;
    else
        result++;
    cout<<"There is a 14 in list lst and it is"<<" followed by a "<<*(result)<<endl;
    return 0;
}

```

Output:



find_end()

- Looks in a range for the last subsequence that is identical to a specified sequence or that is equivalent in a sense specified by a binary predicate.

```

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

```

Parameters

Parameter	Description
<code>_First1</code>	A forward iterator addressing the position of the first element in the range to be searched.
<code>_Last1</code>	A forward iterator addressing the position one past the final element in the range to be searched.
<code>_First2</code>	A forward iterator addressing the position of the first element in the range to be

	searched.
<code>_Last2</code>	A forward iterator addressing the position one past the final element in the range to be searched.
<code>_Comp</code>	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 34.13

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

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

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

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

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

    int j;
    for(j = 11; j <= 14; j++)
        lst.push_back(j);

    int k;
    for(k = 12; k <= 14; k++)
        vec2.push_back(2*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<<endl;

    //Searching vec1 for a match to lst under identity
    vector<int>::iterator result1;
    result1 = find_end(vec1.begin(), vec1.end(), lst.begin(), lst.end());

    if(result1 == vec1.end())
        cout<<"There is no match of lst in vec1."<<endl;
    else
        cout<<"There is a match of lst in vec1 that begins at "
            <<"position "<<result1 - vec1.begin()<<endl;

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

```

if(result2 == vec1.end())
cout<<"\nThere is no match of lst in vec1."<<endl;
else
cout<<"\nThere is a sequence of elements in vec1 that "
<<"are\nequivalent to those in vec2 under the binary "
<<"predicate\ntwice and that begins at position "
<<result2 - vec1.begin()<<endl;
return 0;
}

```

Output:

find_first_of()

- Searches for the first occurrence of any of several values within a target range or for the first occurrence of any of several elements that are equivalent in a sense specified by a binary predicate to a specified set of the elements.

```

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

```

Parameters

Parameter	Description
<i>_First1</i>	A forward iterator addressing the position of the first element in the range to be searched.
<i>_Last1</i>	A forward iterator addressing the position one past the final element in the range to be searched.
<i>_First2</i>	A forward iterator addressing the position of the first element in the range to be matched.
<i>_Last2</i>	A forward iterator addressing the position one past the final element in the range to be matched.
<i>_Comp</i>	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 34.14

- The return value is a forward iterator addressing the position of the first element of the first subsequence that matches the specified sequence or that is equivalent in a sense specified by a binary predicate.

- The operator== used to determine the match between an element and the specified value must impose an equivalence relation between its operands.
- The ranges referenced must be valid; all pointers must be de-referenceable and, within each sequence, the last position is reachable from the first by incrementation.

```
//algorithm, find_first_of()
#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 = 3; j <= 4; j++)
        lst.push_back(5*j);

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

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

    cout<<"List 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;

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

    if(result1 == vec1.end())
        cout<<"\nThere is no match of lst in vec1."<<endl;
    else
        cout<<"\nThere is at least one match of lst in vec1"
        <<"\nand the first one begins at "
        <<"position "<<result1 - vec1.begin()<<endl;

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

    if(result2 == vec1.end())
        cout<<"\nThere is no match of lst in vec1."<<endl;
    else
        cout<<"\nThere is a sequence of elements in vec1 that "
        <<"are\nequivalent to those in vec2 under the binary\n"
        <<"predicate twice and the first one begins at position "
        <<result2 - vec1.begin()<<endl;
    return 0;
}
```

Output:


```

C:\Documents and Settings\Administrator\My Documents\Visu...
Vector vec1 data: 0 5 10 15 20 25
List lst data: 15 20
Vector vec2 data: 20 30 40

There is at least one match of lst in vec1
and the first one begins at position 3

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

```

find_if()

- Locates the position of the first occurrence of an element in a range that satisfies a specified condition.

```

template<class InputIterator, class Predicate>
InputIterator find_if(
    InputIterator _First,
    InputIterator _Last,
    Predicate _Pred
);

```

Parameters

Parameter	Description
<i>_First</i>	An input iterator addressing the position of the first element in the range to be searched.
<i>_Last</i>	An input iterator addressing the position one past the final element in the range to be searched.
<i>_Pred</i>	User-defined predicate function object that defines the condition to be satisfied by the element being searched for. A predicate takes single argument and returns true or false.

Table 34.15

- The return value is an input iterator that addresses the first element in the range that satisfies the condition specified by the predicate.
- This template function is a generalization of the algorithm `find()`, replacing the predicate "equals a specific value" with any predicate.

```

//algorithm, find_if()
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

bool great(int value)
{return value>13;}

int main()
{
    list<int> lst;
    list<int>::iterator Iter;
    list<int>::iterator result;

    lst.push_back(13);
    lst.push_back(9);
    lst.push_back(10);
    lst.push_back(22);
    lst.push_back(31);
    lst.push_back(17);

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

    cout<<"\nOperation: find_if(lst.begin(), lst.end(), great)\n";
    result = find_if(lst.begin(), lst.end(), great);
    if(result == lst.end())
        cout<<"There is no element greater than 13 in list lst."<<endl;
}

```

```

else
result++;
cout<<"There is an element greater than 13\nin list lst,"
<<" and it is followed by a "
<<*(result)<<endl;
return 0;
}

```

Output :

```

c:\Documents and Settings\Administrator\My Document...
List lst data: 13 9 10 22 31 17
Operation: find_if(lst.begin(), lst.end(), great)
There is an element greater than 13
in list lst, and it is followed by a 31
Press any key to continue

```

for_each()

- Applies a specified function object to each element in a forward order within a range and returns the function object.

```

template<class InputIterator, class Function>
Function for_each(
    InputIterator _First,
    InputIterator _Last,
    Function _Func
);

```

Parameters

Parameter	Description
<i>_First</i>	An input iterator addressing the position of the first element in the range to be operated on.
<i>_Last</i>	An input iterator addressing the position one past the final element in the range operated on.
<i>_Func</i>	User-defined function object that is applied to each element in the range.

Table 34.16

- The return value is a copy of the function object after it has been applied to all of the elements in the range.
- The algorithm `for_each()` is very flexible, allowing the modification of each element within a range in different, user-specified ways.
- Templated functions may be reused in a modified form by passing different parameters. User-defined functions may accumulate information within an internal state that the algorithm may return after processing all of the elements in the range.
- The 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 complexity is linear with at most $(_Last - _First)$ comparisons.

```

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

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

```

```

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

//The function object to determine the average
class Average
{
private:
//The number of elements
long num;
//The sum of the elements
long sum;
public:
//Constructor initializes the value to multiply by
Average() : num(0), sum(0){}

//The function call to process the next element
void operator()( int elem ) \
{
//Increment the element count
num++;
//Add the value to the partial sum
sum += elem;
}
//return Average
operator double()
{
return (static_cast <double> (sum))/(static_cast <double> (num));
}
};

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

//Constructing vector vec
int i;
for(i = -3; i <= 4; i++)
vec.push_back(i);

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

//Using for_each to multiply each element by a Factor
for_each(vec.begin(), vec.end(), MultValue<int>(-2));

cout<<"\nMultiplying the elements of the vector vec\n"
<<"by the factor -2 gives:\nvecmult1 data: ";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

//The function object is templatized and so can be
//used again on the elements with a different Factor
for_each(vec.begin(), vec.end(), MultValue<int>(5));

cout<<"\nMultiplying the elements of the vector vecmult1\n"
<<"by the factor 5 gives:\nvecmult2 data: ";
for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
cout<<*Iter1<<" ";
cout<<endl;

//The local state of a function object can accumulate
//information about a sequence of actions that the
//return value can make available, here the Average
double avemod2 = for_each(vec.begin(), vec.end(), Average());
cout<<"\nThe average of the elements of vec is:\nAverage(vecmult2) = "<<avemod2<<endl;
return 0;
}

```

Output:

```

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

Multiplying the elements of the vector vec
by the factor -2 gives:
vecmult1 data: 6 4 2 0 -2 -4 -6 -8

Multiplying the elements of the vector vecmult1
by the factor 5 gives:
vecmult2 data: 30 20 10 0 -10 -20 -30 -40

The average of the elements of vec is:
Average(vecmult2) = -5
Press any key to continue

```

generate()

- Assigns the values generated by a function object to each element in a range.

```

template<class ForwardIterator, class Generator>
void generate(
    ForwardIterator _First,
    ForwardIterator _Last,
    Generator _Gen
);

```

Parameters

Parameter	Description
<i>_First</i>	A forward iterator addressing the position of the first element in the range to which values are to be assigned.
<i>_Last</i>	A forward iterator addressing the position one past the final element in the range to which values are to be assigned.
<i>_Gen</i>	A function object that is called with no arguments that is used to generate the values to be assigned to each of the elements in the range.

Table 34.17

- The function object is invoked for each element in the range and does not need to return the same value each time it is called. It may, for example, read from a file or refer to and modify a local state.
- The generator's result type must be convertible to the value type of the forward iterators for the range.
- The 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 complexity is linear, with exactly $(_Last - _First)$ calls to the generator being required.

```

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

int main()
{
    //Assigning random values to vector integer elements
    vector<int> vec(5);
    vector<int>::iterator Iter1;
    deque<int> deq(5);
    deque<int>::iterator deqIter;

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

    //Assigning random values to deque integer elements
    cout<<"\nOperation: generate(deq.begin(), deq.end(), rand)\n";
    generate(deq.begin(), deq.end(), rand);
}

```

```

cout<<"Deque deq data: ";
for(deqIter = deq.begin(); deqIter != deq.end(); deqIter++)
cout<<*deqIter<<" ";
cout<<endl;
return 0;
}

```

Output:

generate_n()

- Assigns the values generated by a function object to a specified number of element in a range and returns to the position one past the last assigned value.

```

template<class OutputIterator, class Size, class Generator>
void generate_n(
    OutputIterator _First,
    Size _Count,
    Generator _Gen
);

```

Parameters

Parameter	Description
<i>_First</i>	An output iterator addressing the position of first element in the range to which values are to be assigned.
<i>_Count</i>	A signed or unsigned integer type specifying the number of elements to be assigned a value by the generator function.
<i>_Gen</i>	A function object that is called with no arguments that is used to generate the values to be assigned to each of the elements in the range.

Table 34.18

- The function object is invoked for each element in the range and does not need to return the same value each time it is called. It may, for example, read from a file or refer to and modify a local state.
- The generator's result type must be convertible to the value type of the forward iterators for the range.
- The range referenced must be valid; all pointers must be dereferenceable and, within the sequence, the last position must be reachable from the first by incrementation.
- The complexity is linear, with exactly *_Count* calls to the generator being required.

```

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

int main()
{
    //Assigning random values to vector integer elements
    vector<int> vec(7);
    vector<int>::iterator Iter1;
    deque<int> deq(7);
    deque<int>::iterator deqIter;

    cout<<"\nOperation: generate_n(vec.begin(), 7, rand)\n";
    generate_n(vec.begin(), 7, rand);
    cout<<"Vector vec data: ";
    for(Iter1 = vec.begin(); Iter1 != vec.end(); Iter1++)
    cout<<*Iter1<<" ";
}

```

```

cout<<endl;

//Assigning random values to deque integer elements
cout<<"\nOperation: generate_n(deq.begin(), 4, rand)\n";
generate_n(deq.begin(), 4, rand);
cout<<"Deque deq data: ";
for(deqIter = deq.begin(); deqIter != deq.end(); deqIter++)
cout<<*deqIter<<" ";
cout<<endl;
return 0;
}

```

Output :

includes()

- Tests whether one sorted range contains all the elements contained in a second sorted range, where the ordering or equivalence criterion between elements may be specified by a binary predicate.

```

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

```

Parameters

Parameter	Description
<code>_First1</code>	An input iterator addressing the position of the first element in the first of two sorted source ranges to be tested for whether all the elements of the second are contained in the first.
<code>_Last1</code>	An input iterator addressing the position one past the last element in the first of two sorted source ranges to be tested for whether all the elements of the second are contained in the first.
<code>_First2</code>	An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be tested for whether all the elements of the second are contained in the first.
<code>_Last2</code>	An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be tested for whether all the elements of the second are contained in the first.
<code>_Comp</code>	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 34.19

- The return value is a true if the first sorted range contains all the elements in the second sorted range; otherwise, false.

- Another way to think of this test is that it determined whether the second source range is a subset of the first source 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 sorted source ranges must each be arranged as a precondition to the application of the algorithm includes in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.
- 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 non equivalent elements.
- More precisely, the algorithm tests whether all the elements in the first sorted range under a specified binary predicate have equivalent ordering to those in the second sorted range.
- The complexity of the algorithm is linear with at most $2 * ((_Last1 - _First1) - (_Last2 - _First2)) - 1$ comparisons for nonempty source ranges.

```
//algorithm, includes()
#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, vec2;
    vector<int>::iterator Iter1, Iter2;

    //Constructing vectors vec1 & vec2 with default less-than ordering
    int i;
    for(i = -2; i <= 4; i++)
        vec1.push_back(i);

    int j;
    for(j = -2; j <= 3; j++)
        vec2.push_back(j);

    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;

    cout<<"\nvector vec2 data with range sorted by the "
    <<"binary predicate\nless than is: ";
    for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
        cout<<*Iter2<<" ";
    cout<<endl;

    //Constructing vectors vec3 & vec4 with ranges sorted by greater
    vector<int> vec3(vec1), vec4(vec2);
    vector<int>::iterator Iter3, Iter4;
    sort(vec3.begin(), vec3.end(), greater<int>());
    sort(vec4.begin(), vec4.end(), greater<int>());
    vec3.pop_back();

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

    cout<<"\nvector vec4 data with range sorted by the "
    <<"binary predicate\ngreater is: ";
```

```

for(Iter4 = vec4.begin(); Iter4 != vec4.end(); Iter4++)
cout<<*Iter4<<" ";
cout<<endl;

//Constructing vectors vec5 & vec6 with ranges sorted by mod_lesser
vector<int> vec5(vec1), vec6(vec2);
vector<int>::iterator Iter5, Iter6;
reverse(vec5.begin(), vec5.end());
vec5.pop_back();
vec5.pop_back();
sort(vec5.begin(), vec5.end(), mod_lesser);
sort(vec6.begin(), vec6.end(), mod_lesser);

cout<<"\nvector vec5 data with range sorted by the "
<<"binary predicate\nmod_lesser is: ";
for(Iter5 = vec5.begin(); Iter5 != vec5.end(); Iter5++)
cout<<*Iter5<<" ";
cout<<endl;

cout<<"\nvector vec6 data with range sorted by the "
<<"binary predicate\nmod_lesser is: ";
for(Iter6 = vec6.begin(); Iter6 != vec6.end(); Iter6++)
cout<<*Iter6<<" ";
cout<<endl;

//To test for inclusion under an ascending order
//with the default binary predicate less<int>()
bool Result1;
Result1 = includes(vec1.begin(), vec1.end(), vec2.begin(), vec2.end());
if(Result1)
cout<<"\nAll the elements in vector vec2 are contained in vector vec1."<<endl;
else
cout<<"\nAt least one of the elements in vector vec2 is not contained in vector
vec1."<<endl;

//To test for inclusion under descending
//order specifies binary predicate greater<int>()
bool Result2;
Result2 = includes(vec3.begin(), vec3.end(), vec4.begin(), vec4.end(),
greater<int>());
if(Result2)
cout<<"\nAll the elements in vector vec4 are contained\nin vector vec3."<<endl;
else
cout<<"\nAt least one of the elements in vector vec4\nis not contained in vector
vec3."<<endl;

//To test for inclusion under a user
//defined binary predicate mod_lesser
bool Result3;
Result3 = includes(vec5.begin(), vec5.end(), vec6.begin(), vec6.end(),
mod_lesser);
if(Result3)
cout<<"\nAll the elements in vector vec6 are contained under\nmod_lesser in vector
vec5."<<endl;
else
cout<<"\nAt least one of the elements in vector vec6 is not\ncontained under
mod_lesser in vector vec5."<<endl;
return 0;
}

```

Output:

- Program example compiled using g++.

```

//*****algocopy.cpp*****
//algorithm, copy()
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

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

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

    int j;
    for(j = 10; j <= 20; j++)
        vec2.push_back(j);

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

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

    //To copy the first 4 elements of vec1 into the middle of vec2
    copy(vec1.begin(), vec1.begin() + 4, vec2.begin() + 5);

    cout<<"vec2 with vec1 insert data: ";
    for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
        cout<<*Iter2<<" ";
    cout<<endl;

    //To shift the elements inserted into vec2 two positions
    //to the left
    copy(vec2.begin()+4, vec2.begin() + 7, vec2.begin() + 2);

    cout<<"vec2 with shifted insert data: ";
    for(Iter2 = vec2.begin(); Iter2 != vec2.end(); Iter2++)
        cout<<*Iter2<<" ";
    cout<<endl;
    return 0;
}
```

```
}
```

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

```
[bodo@bakawali ~]$ ./algocopy
```

```
vec1 data: 0 1 2 3 4 5
```

```
vec2 data: 10 11 12 13 14 15 16 17 18 19 20
```

```
vec2 with vec1 insert data: 10 11 12 13 14 0 1 2 3 19 20
```

```
vec2 with shifted insert data: 10 11 14 0 1 0 1 2 3 19 20
```

```

//*****algofindfirstof.cpp*****
//algorithm, find_first_of()
#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 = 3; j <= 4; j++)
        lst.push_back(5*j);

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

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

    cout<<"List 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;

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

    if(result1 == vec1.end())
        cout<<"\nThere is no match of lst in vec1."<<endl;
    else
        cout<<"\nThere is at least one match of lst in vec1"
        <<"\nand the first one begins at "
        <<"position " <<result1 - vec1.begin()<<endl;

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

    if(result2 == vec1.end())
        cout<<"\nThere is no match of lst in vec1."<<endl;
    else
        cout<<"\nThere is a sequence of elements in vec1 that "
        <<"are\nequivalent to those in vec2 under the binary\n"
        <<"predicate twice and the first one begins at position "
        <<result2 - vec1.begin()<<endl;
    return 0;
}
```

```
}
```

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

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

```
There is at least one match of lst in vec1  
and the first one begins at position 3
```

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

```
-----End of Algorithm Part II-----  
---www.tenouk.com---
```

Further reading and digging:

1. Check the [best selling C / C++ and STL books at Amazon.com](#).