11.5. Nonmodifying Algorithms

The algorithms presented in this section enable you to access elements without modifying their values or changing their order.

11.5.1. Counting Elements

difference_type
count (InputIterator *beg*, InputIterator *end*, const T& *value*)

difference_type
count_if (InputIterator *beg*, InputIterator *end*, UnaryPredicate *op*)

- The first form counts the elements in the range [*beg* , *end*) that are equal to value *value*.
- The second form counts the elements in the range [*beg* , *end*) for which the unary predicate

  op(*elem*)

  yields `true` .
- The type of the return value, *difference_type*, is the difference type of the iterator:

  `typename iterator_traits<InputIterator>::difference_type`

  (Section 9.5, page 466, introduces iterator traits.)
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- Associative and unordered containers provide a similar member function, `count()` , to count the number of elements that have a certain value as key (see Section 8.3.3, page 404).
- Complexity: linear (*numElems* comparisons or calls of *op* `()` , respectively).

The following example counts elements according to various criteria:

Click here to view code image

```
// algo/count1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;
    int num;
    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    // count elements with value 4
    num = count (coll.cbegin(), coll.cend(),      // range
                 4);                              // value
    cout << "number of elements equal to 4:      " << num << endl;

    // count elements with even value
    num = count_if (coll.cbegin(), coll.cend(),   // range
                    [](int elem){                 // criterion
                        return elem%2==0;
                    });
    cout << "number of elements with even value: " << num << endl;

    // count elements that are greater than value 4
    num = count_if (coll.cbegin(), coll.cend(),   // range
                    [](int elem){                 // criterion
                        return elem>4;
                    });
    cout << "number of elements greater than 4:   " << num << endl;
}
```

The program has the following output:

```
coll: 1 2 3 4 5 6 7 8 9
number of elements equal to 4:       1
number of elements with even value:  4
number of elements greater than 4:   5
```

Instead of using a lambda, which checks whether the element is even, you could use binders like the following expression:

```
std::bind(std::logical_not<bool>(),
          std::bind(std::modulus<int>(),std::placeholders::_1,2)));
```

or even the deprecated expression:

```
std::not1(std::bind2nd(std::modulus<int>(),2))
```

See Section 10.2.4, page 497, for more details regarding these expressions.

11.5.2. Minimum and Maximum

Click here to view code image

```
ForwardIterator
min_element (ForwardIterator beg, ForwardIterator end)

ForwardIterator
min_element (ForwardIterator beg, ForwardIterator end, CompFunc op)

ForwardIterator
max_element (ForwardIterator beg, ForwardIterator end)

ForwardIterator
max_element (ForwardIterator beg, ForwardIterator end, CompFunc op)

pair<ForwardIterator,ForwardIterator>
minmax_element (ForwardIterator beg, ForwardIterator end)

pair<ForwardIterator,ForwardIterator>
minmax_element (ForwardIterator beg, ForwardIterator end, CompFunc op)
```

- These algorithms return the position of the minimum, the maximum element, or a pair of the minimum and the maximum element in the range [*beg* , *end*) .
- The versions without *op* compare the elements with operator < .
- *op* is used to compare two elements:

  op (*elem1*, *elem2*)

  It should return `true` when the first element is less than the second element.
- If more than one minimum or maximum element exists, `min_element()` and `max_element()` return the first found; `minmax_element()` returns the first minimum but the last maximum element, so `max_element()` and `minmax_element()` don't yield the same maximum element.
- If the range is empty, the algorithms return *beg* or a `pair<` *beg* , *beg* `>` .
- *op* should not modify the passed arguments.
- Complexity: linear (*numElems* -1 comparisons or calls of *op* () , respectively, for `min_element()` and `max_element()` and $\frac{3}{2}$ (*numElems* -1) comparisons or calls of *op* () , respectively, for `minmax_element()`).

The following program prints the minimum and the maximum of the elements in `coll` , using `min_element()` and `max_element()` , as well as `minmax_element()` , and, by using `absLess()` , prints the minimum and the maximum of the absolute values:

Click here to view code image

```
// algo/minmax1.cpp

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

```
bool absLess (int elem1, int elem2)
{
    return abs(elem1) < abs(elem2);
}

int main()
{
    deque<int> coll;

    INSERT_ELEMENTS(coll,2,6);
    INSERT_ELEMENTS(coll,-3,6);

    PRINT_ELEMENTS(coll);

    // process and print minimum and maximum
    cout << "minimum: "
         << *min_element(coll.cbegin(),coll.cend())
         << endl;
    cout << "maximum: "
         << *max_element(coll.cbegin(),coll.cend())
         << endl;

    // print min and max and their distance using minmax_element()
    auto mm = minmax_element(coll.cbegin(),coll.cend());
    cout << "min: " << *(mm.first) << endl;     // print minimum
    cout << "max: " << *(mm.second) << endl;    // print maximum
    cout << "distance: " << distance(mm.first,mm.second) << endl;

    // process and print minimum and maximum of absolute values
    cout << "minimum of absolute values: "
         << *min_element(coll.cbegin(),coll.cend(),
                         absLess)
         << endl;
    cout << "maximum of absolute values: "
         << *max_element(coll.cbegin(),coll.cend(),
                         absLess)
         << endl;
}
```

The program has the following output:

```
2 3 4 5 6 -3 -2 -1 0 1 2 3 4 5 6
minimum: -3
maximum: 6
min: -3
max: 6
distance: 9
minimum of absolute values: 0
maximum of absolute values: 6
```

Note that the algorithms return the *position* of the maximum or minimum element, respectively. Thus, you must use the unary operator `*` to print their values:

Click here to view code image

```
auto mm = minmax_element(coll.begin(),coll.end());
cout << "min: " << *(mm.first) << endl;
cout << "max: " << *(mm.second) << endl;
```

Note also that `minmax_element()` yields the last maximum, so the distance (see Section 9.3.3, page 445) is `9`. By using `max_element()`, the distance would be `-1`.

11.5.3. Searching Elements

Search First Matching Element

Click here to view code image

```
InputIterator
find (InputIterator beg, InputIterator end, const T& value)

InputIterator
find_if (InputIterator beg, InputIterator end, UnaryPredicate op)

InputIterator
find_if_not (InputIterator beg, InputIterator end, UnaryPredicate op)
```

- The first form returns the position of the first element in the range `[ beg , end )` that has a value equal to *value*.
- The second form returns the position of the first element in the range `[ beg , end )` for which the unary predicate

  op (*elem*)

  yields `true` .
- The third form (available since C++11) returns the position of the first element in the range `[ beg , end )` for which the unary predicate

  op (*elem*)

  yields `false` .
- All algorithms return *end* if no matching elements are found.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- If the range is sorted, you should use the `lower_bound()` , `upper_bound()` , `equal_range()` , or `binary_search()` algorithms (see Section 11.10, page 608).
- Associative and unordered containers provide an equivalent member function, `find()` (see Section 8.3.3, page 405), which has a better complexity (logarithmic for associative and even constant for unordered containers).
- Complexity: linear (at most, *numElems* comparisons or calls of *op* `()` , respectively).

The following example demonstrates how to use `find()` to find a subrange starting with the first element with value `4` and ending after the second `4` , if any:

Click here to view code image

```
// algo/find1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

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

    PRINT_ELEMENTS(coll,"coll: ");

    //find first element with value 4
    list<int>::iterator pos1;
    pos1 = find (coll.begin(), coll.end(),        //range
                 4);                               //value

    //find second element with value 4
    //- note: continue the search behind the first 4 (if any)
    list<int>::iterator pos2;
    if (pos1 != coll.end()) {
        pos2 = find (++pos1, coll.end(),          //range
                     4);                           //value
    }

    //print all elements from first to second 4 (both included)
    //- note: now we need the position of the first 4 again (if any)
    if (pos1!=coll.end() && pos2!=coll.end()) {
        copy (--pos1, ++pos2,
              ostream_iterator<int>(cout," "));
        cout << endl;
    }
}
```

To find the second `4` , you must increment the position of the first `4` . However, incrementing the `end()` of a collection results in undefined behavior. Thus, if you are not sure, you should check the return value of `find()` before you increment it. The program has the following output:

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

You can call `find()` twice for the same range but with two different values. However, you have to be careful to use the results as the beginning and the end of a subrange of elements; otherwise, the subrange might not be valid. See Section 6.4.1, page 203, for a discussion of possible problems and for an example.

The following example demonstrates how to use `find_if()` and `find_if_not()` to find elements according to very different search criteria:

Click here to view code image

```
// algo/find2.cpp

#include "algostuff.hpp"
using namespace std;
using namespace std::placeholders;

int main()
{
    vector<int> coll;
    vector<int>::iterator pos;

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

    // find first element greater than 3
    pos = find_if (coll.begin(), coll.end(),      // range
                   bind(greater<int>(),_1,3));    // criterion

    // print its position
    cout << "the "
         << distance(coll.begin(),pos) + 1
         << ". element is the first greater than 3" << endl;

    // find first element divisible by 3
    pos = find_if (coll.begin(), coll.end(),
                   [](int elem){
                       return elem%3==0;
                   });

    // print its position
    cout << "the "
         << distance(coll.begin(),pos) + 1
         << ". element is the first divisible by 3" << endl;

    // find first element not <5
    pos = find_if_not (coll.begin(), coll.end(),
                       bind(less<int>(),_1,5));
    cout << "first value >=5: " << *pos << endl;
}
```

The first call of `find_if()` uses a simple function object combined with the `bind` adapter (see Section 10.2.2, page 487) to search for the first element that is greater than `3`. The second call uses a lambda to find the first element that is divisible by `3` without remainder.

The program has the following output:

```
coll: 1 2 3 4 5 6 7 8 9
the 4. element is the first greater than 3
the 3. element is the first divisible by 3
first value >=5: 5
```

See Section 6.8.2, page 226, for an example that lets `find_if()` find the first prime number.

Search First *n* Matching Consecutive Elements

Click here to view code image

```
ForwardIterator
search_n (ForwardIterator beg, ForwardIterator end,
          Size count, const T& value)

ForwardIterator
search_n (ForwardIterator beg, ForwardIterator end,
          Size count, const T& value, BinaryPredicate op)
```

- The first form returns the position of the first of *count* consecutive elements in the range [*beg* , *end*) that all have a value equal to *value*.
- The second form returns the position of the first of *count* consecutive elements in the range [*beg* , *end*) for which the binary predicate

op(*elem*, *value*)

yields `true` (*value* is the passed fourth argument).

- Both forms return *end* if no matching elements are found.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- These algorithms were not part of the original STL and were not introduced very carefully. The fact that the second form uses a binary predicate instead of a unary predicate breaks the consistency of the original STL. See the remarks on page 532.
- Complexity: linear (at most, *numElems* * *count* comparisons or calls of *op*`()`, respectively).

The following example searches for consecutive elements that have a value equal to `7` or an odd value:

Click here to view code image

```
// algo/searchn1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   deque<int>  coll;

   coll = { 1, 2, 7, 7, 6, 3, 9, 5, 7, 7, 7, 3, 6 };
   PRINT_ELEMENTS(coll);

   // find three consecutive elements with value 7
   deque<int>::iterator pos;
   pos = search_n (coll.begin(), coll.end(),    // range
                   3,                           // count
                   7);                          // value

   // print result
   if (pos != coll.end()) {
       cout << "three consecutive elements with value 7 "
            << "start with " << distance(coll.begin(),pos) +1
            << ". element" << endl;
   }
   else {
       cout << "no four consecutive elements with value 7 found"
            << endl;
   }

   // find four consecutive odd elements
   pos = search_n (coll.begin(), coll.end(),    // range
                   4,                           // count
                   0,                           // value
                   [](int elem, int value){     // criterion
                       return elem%2==1;
                   });

   // print result
   if (pos != coll.end()) {
       cout << "first four consecutive odd elements are: ";
       for (int i=0; i<4; ++i, ++pos) {
            cout << *pos << ' ';
       }
   }
   else {
       cout << "no four consecutive elements with value > 3 found";
   }
   cout << endl;
}
```

The program has the following output:

```
1 2 7 7 6 3 9 5 7 7 7 3 6
three consecutive elements with value 7 start with 9. element
first four consecutive odd elements are: 3 9 5 7
```

There is a nasty problem with the second form of `search_n()`. Consider the second call of `search_n()`:

Click here to view code image

```
pos = search_n (coll.begin(), coll.end(),    // range
                4,                           // count
```

```
                   0,                                  // value

                   [](int elem, int value){            // criterion
                      return elem%2==1;
                   });
```

This kind of searching for elements that match a special criterion does not conform to the rest of the STL. Following the usual concepts of the STL, the call should be as follows:

Click here to view code image

```
pos = search_n_if (coll.begin(), coll.end(),       // range
                   4,                              // count
                   [](int elem){                   // criterion
                      return elem%2==1;
                   });
```

However, the algorithm requires a binary predicate, which gets the value passed as fourth argument to `search_n()` as second parameter.

Unfortunately, nobody noticed this inconsistency when these new algorithms were introduced to the C++98 standard (they were not part of the original STL). At first, it seemed that the version with four arguments is more convenient because you could implement something like:

Click here to view code image

```
// find four consecutive elements with value greater than 3
pos = search_n (coll.begin(), coll.end(),    // range
                4,                           // count
                3,                           // value
                greater<int>());             // criterion
```

However, as our example demonstrates, it requires a binary predicate even if you need only a unary predicate.

The consequence is that if you have an ordinary unary predicate, such as

```
bool isPrime (int elem);
```

you either have to change the signature of your function or write a simple wrapper:

Click here to view code image

```
bool binaryIsPrime (int elem1, int) {
    return isPrime(elem1);
}
...
pos = search_n (coll.begin(), coll.end(),    // range
                4,                           // count
                0,                           // required dummy value
                binaryIsPrime);              // binary criterion
```

Search First Subrange

Click here to view code image

```
ForwardIterator1
search (ForwardIterator1 beg, ForwardIterator1 end,
        ForwardIterator2 searchBeg, ForwardIterator2 searchEnd)

ForwardIterator1
search (ForwardIterator1 beg, ForwardIterator1 end,
        ForwardIterator2 searchBeg, ForwardIterator2 searchEnd,
        BinaryPredicate op)
```

- Both forms return the position of the first element of the first subrange matching the range [*searchBeg*, *searchEnd*) in the range [*beg*, *end*).
- In the first form, the elements of the subrange have to be equal to the elements of the whole range.
- In the second form, for every comparison between elements, the call of the binary predicate

  $$op(elem, searchElem)$$

  has to yield `true`.
- Both forms return *end* if no matching elements are found.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- See Section 6.4.1, page 203, for a discussion of how to find a subrange for which you know only the first and the last elements.

- Complexity: linear (at most, *numElems* * *numSearchElems* comparisons or calls of *op* () , respectively).

The following example demonstrates how to find a sequence as the first subrange of another sequence (compare with the example of find_end() on page 537):

Click here to view code image

```
// algo/search1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;
    list<int> subcoll;

    INSERT_ELEMENTS(coll,1,7);
    INSERT_ELEMENTS(coll,1,7);
    INSERT_ELEMENTS(subcoll,3,6);

    PRINT_ELEMENTS(coll,   "coll:    ");
    PRINT_ELEMENTS(subcoll,"subcoll: ");

    // search first occurrence of subcoll in coll
    deque<int>::iterator pos;
    pos = search (coll.begin(), coll.end(),          // range
                  subcoll.begin(), subcoll.end());   // subrange

    // loop while subcoll found as subrange of coll
    while (pos != coll.end()) {
        // print position of first element
        cout << "subcoll found starting with element "
             << distance(coll.begin(),pos) + 1
             << endl;

        // search next occurrence of subcoll
        ++pos;
        pos = search (pos, coll.end(),                  // range
                      subcoll.begin(), subcoll.end());  // subrange
    }
}
```

The program has the following output:

```
coll:    1 2 3 4 5 6 7 1 2 3 4 5 6 7
subcoll: 3 4 5 6
subcoll found starting with element 3
subcoll found starting with element 10
```

The next example demonstrates how to use the second form of the search() algorithm to find a subsequence that matches a more complicated criterion. Here, the subsequence *even, odd, and even value* is searched:

Click here to view code image

```
// algo/search2.cpp

#include "algostuff.hpp"
using namespace std;

// checks whether an element is even or odd
bool checkEven (int elem, bool even)
{
    if (even) {
        return elem % 2 == 0;
    }
    else {
        return elem % 2 == 1;
    }
}

int main()
{
    vector<int> coll;

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

    // arguments for checkEven()
    // - check for: "even odd even"
    bool checkEvenArgs[3] = { true, false, true };

    // search first subrange in coll
    vector<int>::iterator pos;
    pos = search (coll.begin(), coll.end(),       // range
                  checkEvenArgs, checkEvenArgs+3, // subrange values
                  checkEven);                     // subrange criterion

    // loop while subrange found
    while (pos != coll.end()) {
        // print position of first element
        cout << "subrange found starting with element "
             << distance(coll.begin(),pos) + 1
             << endl;

        // search next subrange in coll
        pos = search (++pos, coll.end(),              // range
                      checkEvenArgs, checkEvenArgs+3, // subr. values
                      checkEven);                     // subr. criterion
    }
}
```

The program has the following output:

```
coll: 1 2 3 4 5 6 7 8 9
subrange found starting with element 2
subrange found starting with element 4
subrange found starting with element 6
```

Search Last Subrange

Click here to view code image

```
ForwardIterator1
find_end (ForwardIterator1 beg, ForwardIterator1 end,
          ForwardIterator2 searchBeg, ForwardIterator2 searchEnd)

ForwardIterator1
find_end (ForwardIterator1 beg, ForwardIterator1 end,
          ForwardIterator2 searchBeg, ForwardIterator2 searchEnd,
          BinaryPredicate op)
```

- Both forms return the position of the first element of the last subrange matching the range `[` *searchBeg* `,` *searchEnd* `)` in the range `[` *beg* `,` *end* `)` .
- In the first form, the elements of the subrange have to be equal to the elements of the whole range.
- In the second form, for every comparison between elements, the call of the binary predicate

  op `(`*elem*`,`*searchElem*`)`

  has to yield `true` .
- Both forms return *end* if no matching elements are found.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- See Section 6.4.1, page 203, for a discussion of how to find a subrange for which you know only the first and the last elements.
- These algorithms were not part of the original STL. Unfortunately, they were called `find_end()` instead of `search_end()` , which would be more consistent, because the algorithm used to search the first subrange is called `search()` .
- Complexity: linear (at most, *numElems* `*` *numSearchElems* comparisons or calls of *op* `()` , respectively).

The following example demonstrates how to find a sequence as the last subrange of another sequence (compare with the example of `search()` on page 534):

Click here to view code image

```
// algo/findend1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;
    list<int> subcoll;

    INSERT_ELEMENTS(coll,1,7);
    INSERT_ELEMENTS(coll,1,7);
    INSERT_ELEMENTS(subcoll,3,6);

    PRINT_ELEMENTS(coll,    "coll:    ");
    PRINT_ELEMENTS(subcoll,"subcoll: ");

    // search last occurrence of subcoll in coll
    deque<int>::iterator pos;
    pos = find_end (coll.begin(), coll.end(),         // range
                    subcoll.begin(), subcoll.end());  // subrange

    // loop while subcoll found as subrange of coll
    deque<int>::iterator end(coll.end());
    while (pos != end) {
        // print position of first element
        cout << "subcoll found starting with element "
             << distance(coll.begin(),pos) + 1
             << endl;

        // search next occurrence of subcoll
        end = pos;
        pos = find_end (coll.begin(), end,                // range
                        subcoll.begin(), subcoll.end()); // subrange
    }
}
```

The program has the following output:

```
coll:    1 2 3 4 5 6 7 1 2 3 4 5 6 7
subcoll: 3 4 5 6
subcoll found starting with element 10
subcoll found starting with element 3
```

For the second form of this algorithm, see the second example of `search()` on page 535. You can use `find_end()` in a similar manner.

Search First of Several Possible Elements

Click here to view code image

```
InputIterator
find_first_of (InputIterator beg, InputIterator end,
               ForwardIterator searchBeg, ForwardIterator searchEnd)

InputIterator
find_first_of (InputIterator beg, InputIterator end,
               ForwardIterator searchBeg, ForwardIterator searchEnd,
               BinaryPredicate op)
```

- The first form returns the position of the first element in the range [*beg*, *end*) that is also in the range [*searchBeg*, *searchEnd*).
- The second form returns the position of the first element in the range [*beg*, *end*) for which any call

  op(*elem*,*searchElem*)

  with all elements of [*searchBeg*, *searchEnd*) yields `true`.
- Both forms return *end* if no matching elements are found.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- By using reverse iterators, you can find the last of several possible values.
- These algorithms were not part of the original STL.
- Before C++11, these algorithms required forward iterators instead of input iterators for the range [*beg*, *end*).
- Complexity: linear (at most, *numElems* * *numSearchElems* comparisons or calls of *op*(), respectively).

The following example demonstrates the use of `find_first_of()`:

Click here to view code image

```
// algo/findof1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;
    list<int> searchcoll;

    INSERT_ELEMENTS(coll,1,11);
    INSERT_ELEMENTS(searchcoll,3,5);

    PRINT_ELEMENTS(coll,       "coll:       ");
    PRINT_ELEMENTS(searchcoll,"searchcoll: ");

    // search first occurrence of an element of searchcoll in coll
    vector<int>::iterator pos;
    pos = find_first_of (coll.begin(), coll.end(),      // range
                         searchcoll.begin(),     // beginning of search set
                         searchcoll.end());      // end of search set
    cout << "first element of searchcoll in coll is element "
         << distance(coll.begin(),pos) + 1
         << endl;

    // search last occurrence of an element of searchcoll in coll
    vector<int>::reverse_iterator rpos;
    rpos = find_first_of (coll.rbegin(), coll.rend(),     // range
                          searchcoll.begin(),     // beginning of search set
                          searchcoll.end());      // end of search set
    cout << "last element of searchcoll in coll is element "
         << distance(coll.begin(),rpos.base())
         << endl;
}
```

The second call uses reverse iterators to find the last element that has a value equal to one element in `searchcoll`. To print the position of the element, `base()` is called to transform the reverse iterator into an iterator. Thus, you can process the distance from the beginning. Normally, you would have to add `1` to the result of `distance()` because the first element has distance `0` but actually is element `1`. However, because `base()` moves the position of the value to which it refers, you have the same effect (see Section 9.4.1, page 452, for the description of `base()`).

The program has the following output:

Click here to view code image

```
coll:       1 2 3 4 5 6 7 8 9 10 11
searchcoll: 3 4 5
first element of searchcoll in coll is element 3
last element of searchcoll in coll is element 5
```

Search Two Adjacent, Equal Elements

Click here to view code image

```
ForwardIterator
adjacent_find (ForwardIterator beg, ForwardIterator end)

ForwardIterator
adjacent_find (ForwardIterator beg, ForwardIterator end,
               BinaryPredicate op)
```

- The first form returns the first element in the range [*beg* , *end*) that has a value equal to the value of the following element.
- The second form returns the first element in the range [*beg* , *end*) for which the binary predicate

  op (*elem* , *nextElem*)

  yields `true`.
- Both forms return *end* if no matching elements are found.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- Complexity: linear (at most, *numElems* comparisons or calls of *op* (), respectively).

The following program demonstrates both forms of `adjacent_find()` :

Click here to view code image

```
// algo/adjacentfind1.cpp

#include "algostuff.hpp"
using namespace std;

// return whether the second object has double the value of the first
bool doubled (int elem1, int elem2)
{
   return elem1 * 2 == elem2;
}

int main()
{
   vector<int> coll;

   coll.push_back(1);
   coll.push_back(3);
   coll.push_back(2);
   coll.push_back(4);
   coll.push_back(5);
   coll.push_back(5);
   coll.push_back(0);

   PRINT_ELEMENTS(coll,"coll: ");

   // search first two elements with equal value
   vector<int>::iterator pos;
   pos = adjacent_find (coll.begin(), coll.end());

   if (pos != coll.end()) {
       cout << "first two elements with equal value have position "
            << distance(coll.begin(),pos) + 1
            << endl;
   }

   // search first two elements for which the second has double the value of the first
   pos = adjacent_find (coll.begin(), coll.end(),      // range
                        doubled);                      // criterion

   if (pos != coll.end()) {
       cout << "first two elements with second value twice the "
            << "first have pos. "
            << distance(coll.begin(),pos) + 1
            << endl;
   }
}
```

The first call of `adjacent_find()` searches for equal values. The second form uses `doubled()` to find the first element for which the successor has the double value. The program has the following output:

Click here to view code image

```
coll: 1 3 2 4 5 5 0
first two elements with equal value have position 5
first two elements with second value twice the first have pos. 3
```

11.5.4. Comparing Ranges

Testing Equality

Click here to view code image

```
bool
equal (InputIterator1 beg, InputIterator1 end,
       InputIterator2 cmpBeg)

bool
equal (InputIterator1 beg, InputIterator1 end,
       InputIterator2 cmpBeg,
       BinaryPredicate op)
```

- The first form returns whether the elements in the range [*beg* , *end*) are equal to the elements in the range starting with *cmpBeg*.
- The second form returns whether each call of the binary predicate

op (*elem* , *cmpElem*)

with the corresponding elements in the range [*beg* , *end*) and in the range starting with *cmpBeg* yields `true` .

- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- The caller must ensure that the range starting with *cmpBeg* contains enough elements.
- To determine the details of any differences, you should use the `mismatch()` algorithm (see page 546).
- To determine whether two sequences contain the same elements in different order, algorithm `is_permutation()` is provided since C++11 (see page 544).
- Complexity: linear (at most, *numElems* comparisons or calls of *op* () , respectively).

The following example demonstrates both forms of `equal()` . The first call checks whether the elements have values with equal elements. The second call uses an auxiliary predicate function to check whether the elements of both collections have corresponding even and odd elements:

Click here to view code image

```
// algo/equal1.cpp

#include "algostuff.hpp"
using namespace std;

bool bothEvenOrOdd (int elem1, int elem2)
{
    return elem1 % 2 == elem2 % 2;
}

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

    INSERT_ELEMENTS(coll1,1,7);
    INSERT_ELEMENTS(coll2,3,9);

    PRINT_ELEMENTS(coll1,"coll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");

    // check whether both collections are equal
    if (equal (coll1.begin(), coll1.end(),    // first range
               coll2.begin())) {              // second range
        cout << "coll1 == coll2" << endl;
    }
    else {
        cout << "coll1 != coll2" << endl;
    }

    // check for corresponding even and odd elements
    if (equal (coll1.begin(), coll1.end(),     // first range
               coll2.begin(),                  // second range
               bothEvenOrOdd)) {               // comparison criterion
        cout << "even and odd elements correspond" << endl;
    }
    else {
        cout << "even and odd elements do not correspond" << endl;
    }
}
```

The program has the following output:

```
coll1: 1 2 3 4 5 6 7
coll2: 3 4 5 6 7 8 9
coll1 != coll2
even and odd elements correspond
```

Testing for Unordered Equality

Click here to view code image

```
bool
is_permutation (ForwardIterator1 beg1, ForwardIterator1 end1,
                ForwardIterator2 beg2)

bool
is_permutation (ForwardIterator1 beg1, ForwardIterator1 end1,
                ForwardIterator2 beg2,
                    CompFunc op)
```

- Both forms return whether the elements in the range [*beg1* , *end1*) are a permutation of the elements in the range starting with *beg2*; that is, whether they return equal elements in whatever order.
- The first form compares the elements by using operator `==` .
- The second form compares the elements by using the binary predicate

  op (*elem1* , *elem2*)

  which should return `true` when *elem1* is equal to *elem2*.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- All Iterators must have the same value type.
- These algorithms are available since C++11.
- Complexity: at worst quadratic (*numElems1* comparisons or calls of *op* `()` , if all elements are equal and have the same order).

The following example demonstrates the use of an unordered comparison:

Click here to view code image

```
// algo/ispermutation1.cpp

#include "algostuff.hpp"
using namespace std;

bool bothEvenOrOdd (int elem1, int elem2)
{
    return elem1 % 2 == elem2 % 2;
}

int main()
{
    vector<int> coll1;
    list<int> coll2;
    deque<int> coll3;

    coll1 = { 1, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    coll2 = { 1, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
    coll3 = { 11, 12, 13, 19, 18, 17, 16, 15, 14, 11 };

    PRINT_ELEMENTS(coll1,"coll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");
    PRINT_ELEMENTS(coll3,"coll3: ");

    // check whether both collections have equal elements in any order
    if (is_permutation (coll1.cbegin(), coll1.cend(), // first range
                        coll2.cbegin())) {            // second range
        cout << "coll1 and coll2 have equal elements" << endl;
    }
    else {
        cout << "coll1 and coll2 don't have equal elements" << endl;
    }

    // check for corresponding number of even and odd elements
    if (is_permutation (coll1.cbegin(), coll1.cend(), // first range
                        coll3.cbegin(),               // second range
                        bothEvenOrOdd)) {             // comparison criterion
        cout << "numbers of even and odd elements match" << endl;
    }
    else {
        cout << "numbers of even and odd elements don't match" << endl;
    }
}
```

The program has the following output:

```
coll1: 1 1 2 3 4 5 6 7 8 9
coll2: 1 9 8 7 6 5 4 3 2 1
coll3: 11 12 13 19 18 17 16 15 14 11
coll1 and coll2 have equal elements
numbers of even and odd elements match
```

Search the First Difference

Click here to view code image

```
pair<InputIterator1,InputIterator2>
mismatch (InputIterator1 beg, InputIterator1 end,
          InputIterator2 cmpBeg)

pair<InputIterator1,InputIterator2>
mismatch (InputIterator1 beg, InputIterator1 end,
          InputIterator2 cmpBeg,
          BinaryPredicate op)
```

- The first form returns the first two corresponding elements of range [*beg* , *end*) and the range starting with *cmpBeg* that differ.
- The second form returns the first two corresponding elements of range [*beg* , *end*) and the range starting with *cmpBeg* for which the binary predicate

  op(*elem*,*cmpElem*)

  yields `false` .
- If no difference is found, a `pair<>` of *end* and the corresponding element of the second range is returned. Note that this does not mean that both sequences are equal, because the second sequence might contain more elements.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- The caller must ensure that the range starting with *cmpBeg* contains enough elements.
- To check whether two ranges are equal, you should use algorithm `equal()` (see Section 11.5.4, page 542).
- Complexity: linear (at most, *numElems* comparisons or calls of *op* `()` , respectively).

The following example demonstrates both forms of `mismatch()` :

Click here to view code image

```
// algo/mismatch1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll1 = { 1, 2, 3, 4, 5, 6 };
    list<int>   coll2 = { 1, 2, 4, 8, 16, 3 };

    PRINT_ELEMENTS(coll1,"coll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");

    //find first mismatch
    auto values = mismatch (coll1.cbegin(), coll1.cend(),   //first range
                            coll2.cbegin());                //second range
    if (values.first == coll1.end()) {
        cout << "no mismatch" << endl;
    }
    else {
        cout << "first mismatch: "
             << *values.first  << " and "
             << *values.second << endl;
    }

    //find first position where the element of coll1 is not
    //less than the corresponding element of coll2
    values = mismatch (coll1.cbegin(), coll1.cend(),    //first range
                       coll2.cbegin(),                  //second range
                       less_equal<int>());              //criterion
    if (values.first == coll1.end()) {
        cout << "always less-or-equal" << endl;
    }
    else {
        cout << "not less-or-equal: "
             << *values.first << " and "
             << *values.second << endl;
    }
}
```

The first call of `mismatch()` searches for the first corresponding elements that are not equal. The return type is:

```
pair<vector<int>::const_iterator,list<int>::const_iterator>
```

By checking whether the first element in the returned pair equals the end of the passed range, we check whether a mismatch exists. In that case, the values of the corresponding elements are written to standard output.

The second call searches for the first pair of elements in which the element of the first collection is greater than the corresponding element of the second collection and returns these elements. The program has the following output:

```
coll1: 1 2 3 4 5 6
coll2: 1 2 4 8 16 3
first mismatch: 3 and 4
not less-or-equal: 6 and 3
```

Testing for "Less Than"

Click here to view code image

```
bool
lexicographical_compare (InputIterator1 beg1, InputIterator1 end1,
                         InputIterator2 beg2, InputIterator2 end2)

bool
lexicographical_compare (InputIterator1 beg1, InputIterator1 end1,
                         InputIterator2 beg2, InputIterator2 end2,
                         CompFunc op)
```

- Both forms return whether the elements in the range `[` *beg1* `,` *end1* `)` are "lexicographically less than" the elements in the range `[` *beg2* `,` *end2* `)` .
- The first form compares the elements by using operator `<` .
- The second form compares the elements by using the binary predicate

  op (*elem1*, *elem2*)

  which should return `true` when *elem1* is less than *elem2*.
- *Lexicographical comparison* means that sequences are compared element-by-element until any of the following occurs:
  - When two elements are not equal, the result of their comparison is the result of the whole comparison.
  - When one sequence has no more elements, the sequence that has no more elements is less than the other. Thus, the comparison yields `true` if the first sequence is the one that has no more elements.
  - When both sequences have no more elements, both sequences are equal, and the result of the comparison is `false` .
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- Complexity: linear (at most, `min(` *numElems1* `,` *numElems2* `)` comparisons or calls of *op* `()` , respectively).

The following example demonstrates the use of a lexicographical sorting of collections:

Click here to view code image

```
// algo/lexicocmp1.cpp

#include "algostuff.hpp"
using namespace std;

void printCollection (const list<int>& l)
{
    PRINT_ELEMENTS(l);
}
bool lessForCollection (const list<int>& l1, const list<int>& l2)
{
    return lexicographical_compare
                (l1.cbegin(), l1.cend(),    // first range
                 l2.cbegin(), l2.cend());   // second range
}

int main()
{
    list<int> c1, c2, c3, c4;

    // fill all collections with the same starting values
    INSERT_ELEMENTS(c1,1,5);
    c4 = c3 = c2 = c1;

    // and now some differences
    c1.push_back(7);
    c3.push_back(2);
    c3.push_back(0);
```

```
        c4.push_back(2);

        // create collection of collections
        vector<list<int>> cc;
        cc.insert ( cc.begin(), { c1, c2, c3, c4, c3, c1, c4, c2 } );

        // print all collections
        for_each (cc.cbegin(), cc.cend(),
                  printCollection);
        cout << endl;

        // sort collection lexicographically
        sort (cc.begin(), cc.end(),      // range
              lessForCollection);        // sorting criterion

        // print all collections again
        for_each (cc.cbegin(), cc.cend(),
                  printCollection);
    }
```

The vector `cc` is initialized with several collections (all lists). The call of `sort()` uses the binary predicate `lessForCollection()` to compare two collections (see Section 11.9.1, page 596, for a description of `sort()`). In `lessForCollection()`, the `lexicographical_compare()` algorithm is used to compare the collections lexicographically.

The program has the following output:

```
 1 2 3 4 5 7
 1 2 3 4 5
 1 2 3 4 5 2 0
 1 2 3 4 5 2
 1 2 3 4 5 2 0
 1 2 3 4 5 7
 1 2 3 4 5 2
 1 2 3 4 5

 1 2 3 4 5
 1 2 3 4 5
 1 2 3 4 5 2
 1 2 3 4 5 2
 1 2 3 4 5 2 0
 1 2 3 4 5 2 0
 1 2 3 4 5 7
 1 2 3 4 5 7
```

11.5.5. Predicates for Ranges

The following algorithms were introduced with C++11 to check a specific condition for a given range.

Check for (Partial) Sorting

Click here to view code image

```
bool
is_sorted (ForwardIterator beg, ForwardIterator end)

bool
is_sorted (ForwardIterator beg, ForwardIterator end, BinaryPredicate op)

ForwardIterator
is_sorted_until (ForwardIterator beg, ForwardIterator end)

ForwardIterator
is_sorted_until (ForwardIterator beg, ForwardIterator end, BinaryPredicate
op)
```

- `is_sorted()` returns whether the elements in the range [*beg*, *end*) are sorted.
- `is_sorted()_until` returns the position of the first element in the range [*beg*, *end*), which breaks the sorting of this range, or *end* if none.
- The first and third forms use operator `<` to compare elements. The second and fourth forms use the binary predicate

  op(*elem1*, *elem2*)

  which should return `true` if *elem1* is "less than" *elem2*.

- If the range is empty or has only one element, the algorithms return `true` or *end*, respectively.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- These algorithms are available since C++11.
- Complexity: linear (at most *numElems* `-1` calls of `<` or *op* `()`).

The following program demonstrates the use of these algorithms:

Click here to view code image

```
// algo/issorted1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll1 = { 1, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    PRINT_ELEMENTS(coll1,"coll1: ");

    // check whether coll1 is sorted
    if (is_sorted (coll1.begin(), coll1.end())) {
        cout << "coll1 is sorted" << endl;
    }
    else {
        cout << "coll1 is not sorted" << endl;
    }

    map<int,string> coll2;
    coll2 = { {1,"Bill"}, {2,"Jim"}, {3,"Nico"}, {4,"Liu"}, {5,"Ai"} };
    PRINT_MAPPED_ELEMENTS(coll2,"coll2: ");

    // define predicate to compare names
    auto compareName = [](const pair<int,string>& e1,
                          const pair<int,string>& e2){
                            return e1.second<e2.second;
                       };

    // check whether the names in coll2 are sorted
    if (is_sorted (coll2.cbegin(), coll2.cend(),
                   compareName)) {
        cout << "names in coll2 are sorted" << endl;
    }

    else {
        cout << "names in coll2 are not sorted" << endl;
    }

    // print first unsorted name
    auto pos = is_sorted_until (coll2.cbegin(), coll2.cend(),
                                compareName);
    if (pos != coll2.end()) {
        cout << "first unsorted name: " << pos->second << endl;
    }
}
```

The program has the following output:

```
coll1: 1 1 2 3 4 5 6 7 8 9
coll1 is sorted
coll2: [1,Bill] [2,Jim] [3,Nico] [4,Liu] [5,Ai]
names in coll2 are not sorted
first unsorted name: Liu
```

Note that `is_sorted_until()` returns the position of the first unsorted element as an iterator, so we have to call `pos->second` to access the name (the value of the key/value pair).

Check for Being Partitioned

Click here to view code image

```
bool
is_partitioned (InputIterator beg, InputIterator end, UnaryPredicate op)

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

- `is_partitioned()` returns whether the elements in the range `[` *beg* `,` *end* `)` are partitions, so all the elements fulfilling the predicate *op* `()` are positioned before all elements that do not fulfill it.
- `partition_point()` returns the position of the first element in the *partitioned* range `[` *beg* `,` *end* `)`. Thus, for `[` *beg* `,` *end* `)`, `is_partitioned()` has to yield `true` on entry.
- The algorithms use the unary predicate

  op (*elem*)

  which should return `true` if *elem* is "less than" the "partition point" (that is, *elem* is "less than" all elements for which the predicate returns `false`).
- If the range is empty, `partition_point()` returns *end*.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- These algorithms are available since C++11.
- Complexity:
  - `is_partitioned()` : linear (at most *numElems* calls of *op* `()`).
  - `partition_point()` : logarithmic for random-access iterators and linear otherwise (in any case, at most `log(` *numElems* `)` calls of *op* `()`).

The following program demonstrates the use of these algorithms:

Click here to view code image

```
// algo/ispartitioned1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll = { 5, 3, 9, 1, 3, 4, 8, 2, 6 };
    PRINT_ELEMENTS(coll,"coll: ");

    // define predicate: check whether element is odd:
    auto isOdd = [](int elem) {
                     return elem%2==1;
                 };

    // check whether coll is partitioned in odd and even elements
    if (is_partitioned (coll.cbegin(), coll.cend(),     // range
                        isOdd)) {                        // predicate
        cout << "coll is partitioned" << endl;

        // find first even element:
        auto pos = partition_point (coll.cbegin(),coll.cend(),
                                    isOdd);
        cout << "first even element: " << *pos << endl;
    }
    else {
        cout << "coll is not partitioned" << endl;
    }
}
```

The program has the following output:

```
coll: 5 3 9 1 3 4 8 2 6
coll is partitioned
first even element: 4
```

Check for Being a Heap (Maximum Element First)

Click here to view code image

```
bool
is_heap (RandomAccessIterator beg, RandomAccessIterator end)

bool
is_heap (RandomAccessIterator beg, RandomAccessIterator end,
BinaryPredicate op)

RandomAccessIterator
is_heap_until (RandomAccessIterator beg, RandomAccessIterator end)
```

```
RandomAccessIterator
is_heap_until (RandomAccessIterator beg, RandomAccessIterator end,
               BinaryPredicate op)
```

- `is_heap()` returns whether the elements in the range `[` *beg* `,` *end* `)` are a heap (see Section 11.9.4, page 604), which means that *beg* is (one of) the maximum element(s).
- `is_heap()_until` returns the position of the first element in the range `[` *beg* `,` *end* `)` that breaks the sorting as a heap (is larger than the first element) or *end* if none.
- The first and third forms use operator `<` to compare elements. The second and fourth forms use the binary predicate

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

  which should return `true` if *elem1* is "less than" *elem2*.
- If the range is empty or has only one element, the algorithms return `true` or *end*, respectively.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- These algorithms are available since C++11.
- Complexity: linear (at most *numElems* `-1` calls of `<` or *op* `()`).

The following demonstrates the use of these algorithms:

Click here to view code image

```
// algo/isheap1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll1 = { 9, 8, 7, 7, 7, 5, 4, 2, 1 };
    vector<int> coll2 = { 5, 3, 2, 1, 4, 7, 9, 8, 6 };
    PRINT_ELEMENTS(coll1,"coll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");

    // check whether the collections are heaps
    cout << boolalpha << "coll1 is heap: "
         << is_heap (coll1.cbegin(), coll1.cend()) << endl;
    cout << "coll2 is heap: "
         << is_heap (coll2.cbegin(), coll2.cend()) << endl;

    // print the first element that is not a heap in coll2
    auto pos = is_heap_until (coll2.cbegin(), coll2.cend());
    if (pos != coll2.end()) {
        cout << "first non-heap element: " << *pos << endl;
    }
}
```

The program has the following output:

```
coll1: 9 8 7 7 7 5 4 2 1
coll2: 5 3 2 1 4 7 9 8 6
coll1 is heap: true
coll2 is heap: false
first non-heap element: 4
```

All, Any, or None

Click here to view code image

```
bool
all_of (InputIterator beg, InputIterator end, UnaryPredicate op)

bool
any_of (InputIterator beg, InputIterator end, UnaryPredicate op)

bool
none_of (InputIterator beg, InputIterator end, UnaryPredicate op)
```

- These algorithms return whether for all, any (at least one), or none of the elements in the range `[` *beg* `,` *end* `)` , the unary predicate *op* (*elem*)

yields `true` .

- If the range is empty, `all_of()` and `none_of()` return `true` , whereas `any_of()` returns `false` .
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- *op* should not modify the passed arguments.
- These algorithms are available since C++11.
- Complexity: linear (at most *numElems* calls of *op* `()`).

The following demonstrates the use of these algorithms:

Click here to view code image

```
// algo/allanynone1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;
    vector<int>::iterator pos;

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

    // define an object for the predicate (using a lambda)
    auto isEven = [](int elem) {
                      return elem%2==0;
                  };

    // print whether all, any, or none of the elements are/is even
    cout << boolalpha << "all even?:  "
         << all_of(coll.cbegin(),coll.cend(), isEven) << endl;
    cout << "any even?:   "
         << any_of(coll.cbegin(),coll.cend(), isEven) << endl;
    cout << "none even?: "
         << none_of(coll.cbegin(),coll.cend(), isEven) << endl;
}
```

The program has the following output:

```
coll: 1 2 3 4 5 6 7 8 9
all even?:  false
any even?:  true
none even?: false
```