
11.7. Removing Algorithms

The following algorithms remove elements from a range according to their value or to a criterion. These algorithms, however, *cannot* change the number of elements. The algorithms move logically only by overwriting "removed" elements with the following elements that were not removed. They return the new logical end of the range (the position after the last element not removed). See Section 6.7.1, page 218, for details.

11.7.1. Removing Certain Values

Removing Elements in a Sequence

Click here to view code image

```
ForwardIterator
remove (ForwardIterator beg, ForwardIterator end,
        const T& value)

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

- `remove()` removes each element in the range `[` *beg* `,` *end* `)` that is equal to *value*.
- `remove_if()` removes each element in the range `[` *beg* `,` *end* `)` for which the unary predicate

  op(*elem*)

  yields `true`.
- Both algorithms return the logical new end of the modified sequence (the position after the last element not removed).
- The algorithms overwrite "removed" elements by the following elements that were not removed.
- The order of elements that were not removed remains stable.
- It is up to the caller, after calling this algorithm, to use the returned new logical end instead of the original end *end* (see Section 6.7.1, page 218, for more details).
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- Note that `remove_if()` usually copies the unary predicate inside the algorithm and uses it twice. This may lead to problems if the predicate changes its state due to the function call. See Section 10.1.4, page 483, for details.
- Due to modifications, you can't use these algorithms for an associative or unordered container (see Section 6.7.2, page 221). However, these containers provide a similar member function, `erase()` (see Section 8.7.3, page 417).
- Lists provide an equivalent member function, `remove()`, which offers better performance because it relinks pointers instead of assigning element values (see Section 8.8.1, page 420).
- Complexity: linear (*numElems* comparisons or calls of *op* `()`, respectively).

The following program demonstrates how to use `remove()` and `remove_if()`:

Click here to view code image

```
// algo/remove1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,2,6);
    INSERT_ELEMENTS(coll,4,9);
    INSERT_ELEMENTS(coll,1,7);
    PRINT_ELEMENTS(coll,"coll:               ");

    // remove all elements with value 5
    vector<int>::iterator pos;
    pos = remove(coll.begin(), coll.end(),        // range
                 5);                              // value to remove
```

```
    PRINT_ELEMENTS(coll,"size not changed:   ");

    // erase the "removed" elements in the container
    coll.erase(pos, coll.end());
    PRINT_ELEMENTS(coll,"size changed:       ");

    // remove all elements less than 4
    coll.erase(remove_if(coll.begin(), coll.end(),   // range
                         [](int elem){               // remove criterion
                             return elem<4;
                         }),
               coll.end());
    PRINT_ELEMENTS(coll,"<4 removed:         ");
}
```

The program has the following output:

Click here to view code image

```
coll:              2 3 4 5 6 4 5 6 7 8 9 1 2 3 4 5 6 7
size not changed:  2 3 4 6 4 6 7 8 9 1 2 3 4 6 7 5 6 7
size changed:      2 3 4 6 4 6 7 8 9 1 2 3 4 6 7
<4 removed:        4 6 4 6 7 8 9 4 6 7
```

Removing Elements While Copying

Click here to view code image

```
OutputIterator
remove_copy (InputIterator sourceBeg, InputIterator sourceEnd,
             OutputIterator destBeg,
             const T& value)

OutputIterator
remove_copy_if (InputIterator sourceBeg, InputIterator sourceEnd,
                OutputIterator destBeg,
                UnaryPredicate op)
```

- `remove_copy()` is a combination of `copy()` and `remove()`. It copies each element in the source range `[` *sourceBeg,sourceEnd* `)` that is not equal to *value* into the destination range starting with *destBeg*.
- `remove_copy_if()` is a combination of `copy()` and `remove_if()`. It copies each element in the source range `[` *sourceBeg,sourceEnd* `)` for which the unary predicate

  op(*elem*)

  yields `false` into the destination range starting with *destBeg*.
- Both algorithms return the position after the last copied element in the destination range (the first element that is not overwritten).
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- Use `partition_copy()` (see Section 11.8.6, page 594), to copy elements into two destination ranges: one fulfilling and one non fulfilling a predicate (available since C++11).
- Complexity: linear (*numElems* comparisons or calls of *op* `()` and assignments, respectively).

The following program demonstrates how to use `remove_copy()` and `remove_copy_if()`:

Click here to view code image

```
//  algo/remove2.cpp

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

int main()
{
    list<int> coll1;

    INSERT_ELEMENTS(coll1,1,6);
    INSERT_ELEMENTS(coll1,1,9);
    PRINT_ELEMENTS(coll1);

    // print elements without those having the value 3
    remove_copy(coll1.cbegin(), coll1.cend(),        // source
```

```
                ostream_iterator<int>(cout," "),      // destination
                3);                                   // removed value
    cout << endl;

    // print elements without those having a value greater than 4
    remove_copy_if(coll1.cbegin(), coll1.cend(),      // source
                   ostream_iterator<int>(cout," "),   // destination
                   [](int elem){      // criterion for elements NOT copied
                       return elem>4;
                   });
    cout << endl;

    // copy all elements not less than 4 into a multiset
    multiset<int> coll2;
    remove_copy_if(coll1.cbegin(), coll1.cend(),      // source
                   inserter(coll2,coll2.end()),       // destination
                   bind(less<int>(),_1,4));           // elements NOT copied
    PRINT_ELEMENTS(coll2);
}
```

The program has the following output:

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

11.7.2. Removing Duplicates

Removing Consecutive Duplicates

```
ForwardIterator
unique (ForwardIterator beg, ForwardIterator end)

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

- Both forms collapse consecutive equal elements by removing the following duplicates.
- The first form removes from the range [*beg* , *end*) all elements that are equal to the previous elements. Thus, only when the elements in the sequence are sorted, or at least when all elements of the same value are adjacent, does it remove all duplicates.
- The second form removes all elements that follow an element *e* and for which the binary predicate

  $op(e, elem)$

  yields `true` . In other words, the predicate is not used to compare an element with its predecessor; the element is compared with the previous element that was not removed (see the following examples).
- Both forms return the logical new end of the modified sequence (the position after the last element not removed).
- The algorithms overwrite "removed" elements by the following elements that were not removed.
- The order of elements that were not removed remains stable.
- It is up to the caller, after calling this algorithm, to use the returned new logical end instead of the original end *end* (see Section 6.7.1, page 218, for more details).
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- Due to modifications, you can't use these algorithms for an associative or unordered container (see Section 6.7.2, page 221).
- Lists provide an equivalent member function, `unique()` , which offers better performance because it relinks pointers instead of assigning element values (see Section 8.8.1, page 421).
- Complexity: linear (*numElems* comparisons or calls of *op* `()` , respectively).

The following program demonstrates how to use `unique()` :

Click here to view code image

```
// algo/unique1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    // source data
```

```
    int source[] = { 1, 4, 4, 6, 1, 2, 2, 3, 1, 6, 6, 6, 5, 7,
                     5, 4, 4 };
    list<int> coll;

    // initialize coll with elements from source
    copy (begin(source), end(source),          // source
          back_inserter(coll));                // destination
    PRINT_ELEMENTS(coll);

    // remove consecutive duplicates
    auto pos = unique (coll.begin(), coll.end());

    // print elements not removed
    // - use new logical end
    copy (coll.begin(), pos,                   // source
          ostream_iterator<int>(cout," "));    // destination
    cout << "\n\n";

    // reinitialize coll with elements from source
    copy (begin(source), end(source),          // source
          coll.begin());                       // destination
    PRINT_ELEMENTS(coll);

    // remove elements if there was a previous greater element
    coll.erase (unique (coll.begin(), coll.end(),
                        greater<int>()),
                coll.end());
    PRINT_ELEMENTS(coll);
}
```

The program has the following output:

```
1 4 4 6 1 2 2 3 1 6 6 6 5 7 5 4 4
1 4 6 1 2 3 1 6 5 7 5 4

1 4 4 6 1 2 2 3 1 6 6 6 5 7 5 4 4
1 4 4 6 6 6 6 7
```

The first call of `unique()` removes consecutive duplicates. The second call shows the behavior of the second form and removes all the consecutive following elements of an element for which the comparison with `greater` yields `true`. For example, the first `6` is greater than the following `1`, `2`, `2`, `3`, and `1`, so all these elements are removed. In other words, the predicate is not used to compare an element with its predecessor; the element is compared with the previous element that was not removed (see the following description of `unique_copy()` for another example).

Removing Duplicates While Copying

Click here to view code image

```
OutputIterator
unique_copy (InputIterator sourceBeg, InputIterator sourceEnd,
             OutputIterator destBeg)

OutputIterator
unique_copy (InputIterator sourceBeg, InputIterator sourceEnd,
             OutputIterator destBeg,
             BinaryPredicate op)
```

- Both forms are a combination of `copy()` and `unique()`.
- They copy all elements of the source range `[` *sourceBeg,sourceEnd* `)` that are no duplicates of their previous elements into the destination range starting with *destBeg*.
- Both forms return the position after the last copied element in the destination range (the first element that is not overwritten).
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- Complexity: linear (*numElems* comparisons or calls of *op* `()` and assignments, respectively).

The following program demonstrates how to use `unique_copy()`:

Click here to view code image

```
// algo/unique2.cpp

#include "algostuff.hpp"
using namespace std;

bool differenceOne (int elem1, int elem2)
```

```
{
    return elem1 + 1 == elem2 || elem1 - 1 == elem2;
}

int main()
{
    // source data
    int source[] = { 1, 4, 4, 6, 1, 2, 2, 3, 1, 6, 6, 6, 5, 7,
                     5, 4, 4 };

    // initialize coll with elements from source
    list<int> coll;
    copy(begin(source), end(source),                    // source
         back_inserter(coll));                          // destination
    PRINT_ELEMENTS(coll);

    // print elements with consecutive duplicates removed
    unique_copy(coll.cbegin(), coll.cend(),             // source
                ostream_iterator<int>(cout," "));       // destination
    cout << endl;

    // print elements without consecutive entries that differ by one
    unique_copy(coll.cbegin(), coll.cend(),             // source
                ostream_iterator<int>(cout," "),        // destination
                differenceOne);                         // duplicates criterion
    cout << endl;
}
```

The program has the following output:

```
1 4 4 6 1 2 2 3 1 6 6 6 5 7 5 4 4
1 4 6 1 2 3 1 6 5 7 5 4
1 4 4 6 1 3 1 6 6 6 4 4
```

Note that the second call of `unique_copy()` does not remove the elements that differ by `1` from their predecessor by one. Instead, it removes all elements that differ by `1` from their previous element *that is not removed*. For example, after the three occurrences of `6`, the following `5`, `7`, and `5` differ by `1` compared with `6`, so they are removed. However, the following two occurrences of `4` remain in the sequence because compared with `6`, the difference is not `1`.

Another example compresses sequences of spaces:

Click here to view code image

```
// algo/unique3.cpp

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

bool bothSpaces (char elem1, char elem2)
{
    return elem1 == ' ' && elem2 == ' ';
}

int main()
{
    // don't skip leading whitespaces by default
    cin.unsetf(ios::skipws);

    // copy standard input to standard output
    // - while compressing spaces
    unique_copy(istream_iterator<char>(cin),    // beginning of source: cin
                istream_iterator<char>(),       // end of source: end-of-file
                ostream_iterator<char>(cout),   // destination: cout
                bothSpaces);                    // duplicate criterion
}
```

With the input of

```
Hello, here are   sometimes more  and sometimes fewer   spaces.
```

this example produces the following output:

```
Hello, here are sometimes more and sometimes fewer spaces.
```