Username: Pralay Patoria **Book:** The C++ Standard Library: A Tutorial and Reference, Second Edition. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

11.6. Modifying Algorithms

This section describes algorithms that modify the elements of a range. There are two ways to modify elements:

1. Modify them directly while iterating through a sequence.
2. Modify them while copying them from a source range to a destination range.

Several modifying algorithms provide both ways of modifying the elements of a range. In this case, the name of the latter uses the `_copy` suffix.

You can't use an associative or unordered container as a destination range, because the elements in these containers are constant. If you could, it would be possible to compromise the automatic sorting or the hash based position, respectively.

All algorithms that have a separate destination range return the position after the last copied element of that range.

11.6.1. Copying Elements

Click here to view code image

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

OutputIterator
copy_if (InputIterator sourceBeg, InputIterator sourceEnd,
         OutputIterator destBeg,
         UnaryPredicate op)

OutputIterator
copy_n (InputIterator sourceBeg,
        Size num,
        OutputIterator destBeg)

BidirectionalIterator2
copy_backward (BidirectionalIterator1 sourceBeg,
               BidirectionalIterator1 sourceEnd,
               BidirectionalIterator2 destEnd)
```

- All algorithms copy all elements of a source range ([*sourceBeg,sourceEnd*) or *num* elements starting with *sourceBeg*) into the destination range starting with *destBeg* or ending with *destEnd*, respectively.
- They return the position after the last copied element in the destination range (the first element that is not overwritten).
- For `copy()` , *destBeg* should not be part of [*sourceBeg,sourceEnd*) . For `copy_if()` , source and destination ranges should not overlap. For `copy_backward()` , *destEnd* should not be part of (*sourceBeg,sourceEnd*] .
- `copy()` iterates forward through the sequence, whereas `copy_backward()` iterates backward. This difference matters only if the source and destination ranges overlap.
  - – To copy a subrange to the front, use `copy()` . Thus, for `copy()` , *destBeg* should have a position in front of *sourceBeg*.
  - – To copy a subrange to the back, use `copy_backward()` . Thus, for `copy_backward()` , *destEnd* should have a position after *sourceEnd*.

  So, whenever the third argument is an element of the source range specified by the first two arguments, use the other algorithm. Note that switching to the other algorithm means that you switch from passing the beginning of the destination range to passing the end. See page 559 for an example that demonstrates the differences.
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- See Section 9.4.2, page 454, for the implementation of the `copy()` algorithm.
- Since C++11, if the source elements are no longer used, you should prefer `move()` over `copy()` and `move_backward()` over and `copy_backward()` (see Section 11.6.2, page 561).
- Before C++11, no `copy_if()` and `copy_n()` algorithms were provided. To copy only those elements meeting a certain criterion, you had to use `remove_copy_if()` (see Section 11.7.1, page 577) with a negated predicate.
- Use `reverse_copy()` to reverse the order of the elements during the copy (see Section 11.8.1, page 583). Note that

`reverse_copy()` may be slightly more efficient than using `copy()` with reverse iterators.

- To assign all elements of a container, use the assignment operator if the containers have the same type (see Section 8.4, page 406) or the `assign()` member function if the containers have different types (see Section 8.4, page 407).
- To remove elements while they are being copied, use `remove_copy()` and `remove_copy_if()` (see Section 11.7.1, page 577).
- To modify elements while they are being copied, use `transform()` (see Section 11.6.3, page 563) or `replace_copy()` (see Section 11.6.6, page 573).
- Use `partition_copy()` (see Section 11.8.6, page 594) to copy elements into two destination ranges: one fulfilling and one not fulfilling a predicate.
- Complexity: linear (*numElems* assignments).

The following example shows some simple calls of `copy()` (see Section 11.6.2, page 562, for a corresponding version using `move()` when possible):

Click here to view code image

```
// algo/copy1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<string>  coll1 = { "Hello", "this", "is", "an", "example" };
    list<string> coll2;

    // copy elements of coll1 into coll2
    // - use back inserter to insert instead of overwrite
    copy (coll1.cbegin(), coll1.cend(),           // source range
          back_inserter(coll2));                  // destination range

    // print elements of coll2
    // - copy elements to cout using an ostream iterator
    copy (coll2.cbegin(), coll2.cend(),           // source range
          ostream_iterator<string>(cout," "));    // destination range
    cout << endl;

    // copy elements of coll1 into coll2 in reverse order
    // - now overwriting
    copy (coll1.crbegin(), coll1.crend(),         // source range
          coll2.begin());                         // destination range

    // print elements of coll2 again
    copy (coll2.cbegin(), coll2.cend(),           // source range
          ostream_iterator<string>(cout," "));    // destination range
    cout << endl;
}
```

In this example, back inserters (see Section 9.4.2, page 455) are used to insert the elements into the destination range. Without using inserters, `copy()` would overwrite the empty collection `coll2`, resulting in undefined behavior. Similarly, the example uses ostream iterators (see Section 9.4.3, page 460) to use standard output as the destination. The program has the following output:

```
Hello this is an example
example an is this Hello
```

The following example demonstrates the difference between `copy()` and `copy_backward()`:

Click here to view code image

```
// algo/copy2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    // initialize source collection with ''..........abcdef..........''
    vector<char> source(10,'.');
    for (int c='a'; c<='f'; c++) {
        source.push_back(c);
    }
    source.insert(source.end(),10,'.');
    PRINT_ELEMENTS(source,"source: ");
```

```
         // copy all letters three elements in front of the 'a'
         vector<char> c1(source.cbegin(),source.cend());
         copy (c1.cbegin()+10, c1.cbegin()+16,           // source range
               c1.begin()+7);                            // destination range
         PRINT_ELEMENTS(c1,"c1:     ");

         // copy all letters three elements behind the 'f'
         vector<char> c2(source.cbegin(),source.cend());
         copy_backward (c2.cbegin()+10, c2.cbegin()+16,  // source range
                        c2.begin()+19);                  // destination range
         PRINT_ELEMENTS(c2,"c2:     ");
     }
```

Note that in both calls of `copy()` and `copy_backward()`, the third argument is not part of the source range. The program has the following output:

Click here to view code image

```
source: . . . . . . . . . . a b c d e f . . . . . . . . . .
c1:     . . . . . . . a b c d e f d e f . . . . . . . . . .
c2:     . . . . . . . . . . a b c a b c d e f . . . . . . .
```

A third example demonstrates how to use `copy()` as a data filter between standard input and standard output. The program reads strings and prints them, each on one line:

Click here to view code image

```
// algo/copy3.cpp

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

int main()
{
    copy (istream_iterator<string>(cin),          // beginning of source
          istream_iterator<string>(),             // end of source
          ostream_iterator<string>(cout,"\n"));   // destination
}
```

11.6.2. Moving Elements

Click here to view code image

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

BidirectionalIterator2
move_backward (BidirectionalIterator1 sourceBeg,
               BidirectionalIterator1 sourceEnd,
               BidirectionalIterator2 destEnd)
```

- Both algorithms move all elements of the source range [*sourceBeg,sourceEnd*) into the destination range starting with *destBeg* or ending with *destEnd*, respectively.
- Call for each element:

  destElem*=std::move(sourceElem*)

  Thus, if the element type provides move semantics, the value of the source elements becomes undefined, so the source element should no longer be used except to reinitialize or assign a new value to it. Otherwise, the elements are copied as with `copy()` or `copy_backward()` (see Section 11.6.1, page 557).
- They return the position after the last copied element in the destination range (the first element that is not overwritten).
- For `move()`, *destBeg* should not be part of [*sourceBeg,sourceEnd*). For `move_backward()`, *destEnd* should not be part of (*sourceBeg,sourceEnd*].
- `move()` iterates forward through the sequence, whereas `move_backward()` iterates backward. This difference matters only if the source and destination ranges overlap.
  - To move a subrange to the front, use `move()`. Thus, for `move()`, *destBeg* should have a position in front of *sourceBeg*.

- To move a subrange to the back, use `move_backward()`. Thus, for `move_backward()`, *destEnd* should have a position after *sourceEnd*.

So, whenever the third argument is an element of the source range specified by the first two arguments, use the other algorithm. Note that switching to the other algorithm means that you switch from passing the beginning of the destination range to passing the end. See Section 11.6.1, page 559, for an example that demonstrates the differences for the corresponding copy algorithms.

- The caller must ensure that the destination range is big enough or that insert iterators are used.
- These algorithms are available since C++11.
- Complexity: linear (*numElems* move assignments).

The following example demonstrates some simple calls of `move()`. It is the improved example of *algo/copy1.cpp* (see Section 11.6.1, page 558), using `move()` instead of `copy()` whenever possible:

Click here to view code image

```
// algo/move1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<string> coll1 = { "Hello", "this", "is", "an", "example" };
    list<string> coll2;

    // copy elements of coll1 into coll2
    // - use back inserter to insert instead of overwrite
    // - use copy() because the elements in coll1 are used again
    copy (coll1.cbegin(), coll1.cend(),           // source range
          back_inserter(coll2));                  // destination range

    // print elements of coll2
    // - copy elements to cout using an ostream iterator
    // - use move() because these elements in coll2 are not used again
    move (coll2.cbegin(), coll2.cend(),           // source range
          ostream_iterator<string>(cout," "));    // destination range
    cout << endl;

    // copy elements of coll1 into coll2 in reverse order
    // - now overwriting (coll2.size() still fits)
    // - use move() because the elements in coll1 are not used again
    move (coll1.crbegin(), coll1.crend(),         // source range
          coll2.begin());                         // destination range

    // print elements of coll2 again
    // - use move() because the elements in coll2 are not used again
    move (coll2.cbegin(), coll2.cend(),           // source range
          ostream_iterator<string>(cout," "));    // destination range
    cout << endl;
}
```

Note that the elements in `coll2` have an undefined state after their first output because `move()` is used. However, `coll2` still has the size of `5` elements, so we can overwrite these elements with the second call of `move()`. The program has the following output:

```
Hello this is an example
example an is this Hello
```

11.6.3. Transforming and Combining Elements

The `transform()` algorithms provide two abilities:

1. The first form has four arguments. It transforms elements from a source to a destination range. Thus, this form copies and modifies elements in one step.
2. The second form has five arguments. It combines elements from two source sequences and writes the results to a destination range.

Transforming Elements

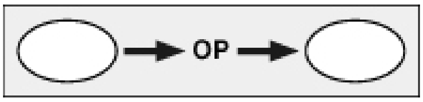
Click here to view code image

```
OutputIterator
transform (InputIterator sourceBeg, InputIterator sourceEnd,
           OutputIterator destBeg,
           UnaryFunc op)
```

- Calls

  op (*elem*)

  for each element in the source range [*sourceBeg,sourceEnd*) and writes each result of *op* to the destination range starting with *destBeg*:



- Returns the position after the last transformed element in the destination range (the first element that is not overwritten with a result).
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- *sourceBeg* and *destBeg* may be identical. Thus, as with `for_each()` , you can use this algorithm to modify elements inside a sequence. See the comparison with the `for_each()` algorithm (Section 11.2.2, page 509) for this kind of use.
- To replace elements matching a criterion with a particular value, use the `replace()` algorithms (see Section 11.6.6, page 571).
- Complexity: linear (*numElems* calls of *op* ()).

The following program demonstrates how to use this kind of `transform()` :

Click here to view code image

```
// algo/transform1.cpp

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

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

    INSERT_ELEMENTS(coll1,1,9);
    PRINT_ELEMENTS(coll1,"coll1:   ");

    // negate all elements in coll1
    transform (coll1.cbegin(), coll1.cend(),       // source range
               coll1.begin(),                      // destination range
               negate<int>());                     // operation
    PRINT_ELEMENTS(coll1,"negated: ");

    // transform elements of coll1 into coll2 with ten times their value
    transform (coll1.cbegin(), coll1.cend(),       // source range
               back_inserter(coll2),               // destination range
               bind(multiplies<int>(),_1,10));     // operation
    PRINT_ELEMENTS(coll2,"coll2:   ");

    // print coll2 negatively and in reverse order
    transform (coll2.crbegin(), coll2.crend(),     // source range
               ostream_iterator<int>(cout," "),    // destination range
               [](int elem){                       // operation
                   return -elem;
               });
    cout << endl;
}
```

The program has the following output:

```
coll1:   1 2 3 4 5 6 7 8 9
negated: -1 -2 -3 -4 -5 -6 -7 -8 -9
coll2:   -10 -20 -30 -40 -50 -60 -70 -80 -90
90 80 70 60 50 40 30 20 10
```

Combining Elements of Two Sequences

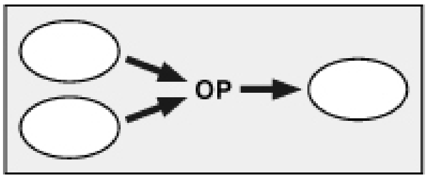
Click here to view code image

```
OutputIterator
transform (InputIterator1 source1Beg, InputIterator1 source1End,
           InputIterator2 source2Beg,
           OutputIterator destBeg,
           BinaryFunc op)
```

- Calls

  op (*source1Elem* , *source2Elem*)

  for all corresponding elements from the first source range [*source1Beg* , *source1End*) and the second source range starting with *source2Beg* and writes each result to the destination range starting with *destBeg*:



- Returns the position after the last transformed element in the destination range (the first element that is not overwritten with a result).
- The caller must ensure that the second source range is big enough (has at least as many elements as the source range).
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- *source1Beg*, *source2Beg*, and *destBeg* may be identical. Thus, you can process the results of elements that are combined with themselves, and you can overwrite the elements of a source with the results.
- Complexity: linear (*numElems* calls of *op* ()).

The following program demonstrates how to use this form of `transform()` :

Click here to view code image

```
// algo/transform2.cpp

#include "algostuff.hpp"
using namespace std;

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

    INSERT_ELEMENTS(coll1,1,9);
    PRINT_ELEMENTS(coll1,"coll1:   ");

    // square each element
    transform (coll1.cbegin(), coll1.cend(),      // first source range
               coll1.cbegin(),                    // second source range
               coll1.begin(),                     // destination range
               multiplies<int>());                // operation
    PRINT_ELEMENTS(coll1,"squared: ");

    // add each element traversed forward with each element traversed backward
    // and insert result into coll2
    transform (coll1.cbegin(), coll1.cend(),      // first source range
               coll1.crbegin(),                   // second source range
               back_inserter(coll2),              // destination range
               plus<int>());                      // operation
    PRINT_ELEMENTS(coll2,"coll2:   ");

    // print differences of two corresponding elements
    cout << "diff:      ";
    transform (coll1.cbegin(), coll1.cend(),      // first source range
               coll2.cbegin(),                    // second source range
               ostream_iterator<int>(cout, " "), // destination range
               minus<int>());                     // operation
    cout << endl;
}
```

The program has the following output:

```
coll1:   1 2 3 4 5 6 7 8 9
squared: 1 4 9 16 25 36 49 64 81
coll2:   82 68 58 52 50 52 58 68 82
diff:    -81 -64 -49 -36 -25 -16 -9 -4 -1
```

11.6.4. Swapping Elements

Click here to view code image

```
ForwardIterator2
```

```
swap_ranges (ForwardIterator1 beg1, ForwardIterator1 end1,
             ForwardIterator2 beg2)
```

- Swaps the elements in the range [*beg1* , *end1*) with the corresponding elements starting with *beg2*.
- Returns the position after the last swapped element in the second range.
- The caller must ensure that the second range is big enough.
- Both ranges must not overlap.
- To swap all elements of a container of the same type, use its `swap()` member function because the member function usually has constant complexity (see Section 8.4, page 407).
- Complexity: linear (*numElems* swap operations).

The following example demonstrates how to use `swap_ranges()` :

Click here to view code image

```
// algo/swapranges1.cpp

#include "algostuff.hpp"
using namespace std;

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

    INSERT_ELEMENTS(coll1,1,9);
    INSERT_ELEMENTS(coll2,11,23);

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

    // swap elements of coll1 with corresponding elements of coll2
    deque<int>::iterator pos;
    pos = swap_ranges (coll1.begin(), coll1.end(),   // first range
                       coll2.begin());               // second range

    PRINT_ELEMENTS(coll1,"\ncoll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");
    if (pos != coll2.end()) {
        cout << "first element not modified: "
             << *pos << endl;
    }

    // mirror first three with last three elements in coll2
    swap_ranges (coll2.begin(), coll2.begin()+3,     // first range
                 coll2.rbegin());                    // second range

    PRINT_ELEMENTS(coll2,"\ncoll2: ");
}
```

The first call of `swap_ranges()` swaps the elements of `coll1` with the corresponding elements of `coll2` . The remaining elements of `coll2` are not modified. The `swap_ranges()` algorithm returns the position of the first element not modified. The second call swaps the first and the last three elements of `coll2` . One of the iterators is a reverse iterator, so the elements are mirrored (swapped from outside to inside). The program has the following output:

Click here to view code image

```
coll1: 1 2 3 4 5 6 7 8 9
coll2: 11 12 13 14 15 16 17 18 19 20 21 22 23

coll1: 11 12 13 14 15 16 17 18 19
coll2: 1 2 3 4 5 6 7 8 9 20 21 22 23
first element not modified: 20

coll2: 23 22 21 4 5 6 7 8 9 20 3 2 1
```

11.6.5. Assigning New Values

Assigning the Same Value

Click here to view code image

```
void
fill (ForwardIterator beg, ForwardIterator end,
     const T& newValue)

OutputIterator
fill_n (OutputIterator beg, Size num,
       const T& newValue)
```

- `fill()` assigns *newValue* to each element in the range [*beg* , *end*) .
- `fill_n()` assigns *newValue* to the first *num* elements in the range starting with *beg*. If `num` is negative, `fill_n()` does nothing (specified only since C++11).
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- Since C++11, `fill_n()` returns the position after the last modified element (*beg* + *num*)or *beg* if *num* is negative (before C++11, `fill_n()` had return type `void`).
- Complexity: linear (*numElems*, *num*, or `0` assignments).

The following program demonstrates the use of `fill()` and `fill_n()` :

Click here to view code image

```
// algo/fill1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    // print ten times 7.7
    fill_n(ostream_iterator<float>(cout, " "),   // beginning of destination
           10,                                   // count
           7.7);                                 // new value
    cout << endl;

    list<string> coll;

    // insert "hello" nine times
    fill_n(back_inserter(coll),       // beginning of destination
           9,                         // count
           "hello");                  // new value
    PRINT_ELEMENTS(coll,"coll: ");

    // overwrite all elements with "again"
    fill(coll.begin(), coll.end(),    // destination
         "again");                    // new value
    PRINT_ELEMENTS(coll,"coll: ");

    // replace all but two elements with "hi"
    fill_n(coll.begin(),               // beginning of destination
           coll.size()-2,              // count
           "hi");                      // new value
    PRINT_ELEMENTS(coll,"coll: ");

    // replace the second and up to the last element but one with "hmmm"
    list<string>::iterator pos1, pos2;
    pos1 = coll.begin();
    pos2 = coll.end();
    fill (++pos1, --pos2,             // destination
          "hmmm");                    // new value
    PRINT_ELEMENTS(coll,"coll: ");
}
```

The first call shows how to use `fill_n()` to print a certain number of values. The other calls of `fill()` and `fill_n()` insert and replace values in a list of strings. The program has the following output:

Click here to view code image

```
7.7 7.7 7.7 7.7 7.7 7.7 7.7 7.7 7.7 7.7
coll: hello hello hello hello hello hello hello hello hello
coll: again again again again again again again again again
coll: hi hi hi hi hi hi hi again again
coll: hi hmmm hmmm hmmm hmmm hmmm hmmm hmmm again
```

Assigning Generated Values

Click here to view code image

```
void
generate (ForwardIterator beg, ForwardIterator end,
          Func op)

OutputIterator
generate_n (OutputIterator beg, Size num,
            Func op)
```

- `generate()` assigns the values that are generated by a call of

  op ()

  to each element in the range [*beg* , *end*) .
- `generate_n()` assigns the values that are generated by a call of

  op ()

  to the first *num* elements in the range starting with *beg*. If `num` is negative, `generate_n()` does nothing (specified only since C++11).
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- Since C++11, `generate_n()` returns the position after the last modified element (*beg* + *num*) or *beg* if *num* is negative (before C++11, `generate_n()` had return type `void`).
- Complexity: linear (*numElems*, *num*, or `0` calls of *op* `()` and assignments).

The following program demonstrates how to use `generate()` and `generate_n()` to insert or assign some random numbers:

Click here to view code image

```
// algo/generate1.cpp

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

int main()
{
    list<int> coll;

    // insert five random numbers
    generate_n (back_inserter(coll),      // beginning of destination range
                5,                        // count
                rand);                    // new value generator
    PRINT_ELEMENTS(coll);

    // overwrite with five new random numbers
    generate (coll.begin(), coll.end(),  // destination range
              rand);                      // new value generator
    PRINT_ELEMENTS(coll);
}
```

The `rand()` function is described in Section 17.3, page 942. The program might have the following output:

Click here to view code image

```
1481765933 1085377743 1270216262 1191391529 812669700
553475508 445349752 1344887256 730417256 1812158119
```

The output is platform dependent because the random-number sequence that `rand()` generates is not standardized.

See Section 10.1.2, page 478, for an example that demonstrates how to use `generate()` with function objects so that it generates a sequence of numbers.

Assigning Sequence of Increments Values

Click here to view code image

```
void
iota (ForwardIterator beg, ForwardIterator end,
      T startValue)
```

- assigns *startValue*, *startValue* `+1` , *startValue* `+2` , and so on.
- Provided since C++11.

- Complexity: linear (*numElems* assignments and increments).

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

Click here to view code image

```
// algo/iota1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    array<int,10> coll;

    iota (coll.begin(), coll.end(),   // destination range
          42);                        // start value

    PRINT_ELEMENTS(coll,"coll: ");
}
```

The program has the following output:

```
coll: 42 43 44 45 46 47 48 49 50 51
```

11.6.6. Replacing Elements

Replacing Values Inside a Sequence

Click here to view code image

```
void
replace (ForwardIterator beg, ForwardIterator end,
         const T& oldValue, const T& newValue)

void
replace_if (ForwardIterator beg, ForwardIterator end,
            UnaryPredicate op, const T& newValue)
```

- `replace()` replaces each element in the range `[` *beg* `,` *end* `)` that is equal to *oldValue* with *newValue*.
- `replace_if()` replaces each element in the range `[` *beg* `,` *end* `)` for which the unary predicate

  op(*elem*)

  yields `true` with *newValue*.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- Complexity: linear (*numElems* comparisons or calls of *op* `()` , respectively).

The following program demonstrates some examples of the use of `replace()` and `replace_if()` :

Click here to view code image

```
// algo/replace1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

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

    // replace all elements with value 6 with 42
    replace (coll.begin(), coll.end(),      // range
             6,                             // old value
             42);                           // new value
    PRINT_ELEMENTS(coll,"coll: ");

    // replace all elements with value less than 5 with 0
    replace_if (coll.begin(), coll.end(),   // range
                [](int elem){               // criterion for replacement
                  return elem<5;
```

```
                 },
                 0);                           // new value
    PRINT_ELEMENTS(coll,"coll: ");
}
```

The program has the following output:

```
coll: 2 3 4 5 6 7 4 5 6 7 8 9
coll: 2 3 4 5 42 7 4 5 42 7 8 9
coll: 0 0 0 5 42 7 0 5 42 7 8 9
```

Copying and Replacing Elements

Click here to view code image

```
OutputIterator
replace_copy (InputIterator sourceBeg, InputIterator sourceEnd,
              OutputIterator destBeg,
              const T& oldValue, const T& newValue)

OutputIterator
replace_copy_if (InputIterator sourceBeg, InputIterator sourceEnd,
                 OutputIterator destBeg,
                 UnaryPredicate op, const T& newValue)
```

- `replace_copy()` is a combination of `copy()` and `replace()` . It replaces each element in the source range [*sourceBeg,sourceEnd*) that is equal to *oldValue* with *newValue* while the elements are copied into the destination range starting with *destBeg*.
- `replace_copy_if()` is a combination of `copy()` and `replace_if()` . It replaces each element in the source range [*sourceBeg,sourceEnd*) for which the unary predicate

  op(*elem*)

- yields `true` with *newValue* while the elements are copied 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.
- Complexity: linear (*numElems* comparisons or calls of *op* () and assignments, respectively).

The following program demonstrates how to use `replace_copy()` and `replace_copy_if()` :

Click here to view code image

```
// algo/replace2.cpp

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

int main()
{
    list<int> coll;

    INSERT_ELEMENTS(coll,2,6);
    INSERT_ELEMENTS(coll,4,9);
    PRINT_ELEMENTS(coll);

    //print all elements with value 5 replaced with 55
    replace_copy(coll.cbegin(), coll.cend(),        //source
                 ostream_iterator<int>(cout," "),   //destination
                 5,                                 //old value
                 55);                               //new value
    cout << endl;

    //print all elements with a value less than 5 replaced with 42
    replace_copy_if(coll.cbegin(), coll.cend(),     //source
                    ostream_iterator<int>(cout," "), //destination
                    bind(less<int>(),_1,5),          //replacement criterion
                    42);                             //new value
    cout << endl;

    //print each element while each odd element is replaced with 0
    replace_copy_if(coll.cbegin(), coll.cend(),     //source
                    ostream_iterator<int>(cout," "), // destination
                    [](int elem){                    // replacement criterion
                        return elem%2==1;
                    },
                    0);                              // new value
    cout << endl;
}
```

The program has the following output:

```
2 3 4 5 6 4 5 6 7 8 9
2 3 4 55 6 4 55 6 7 8 9
42 42 42 5 6 42 5 6 7 8 9
2 0 4 0 6 4 0 6 0 8 0
```