6.7. Manipulating Algorithms

So far, I have introduced the whole concept of the STL as a framework: Containers represent different concepts to manage collections of data. Algorithms are provided to perform read and write operations on these collections. Iterators are the glue between containers and algorithms. Provided by the containers, iterators allow you to iterate over all elements in different orders and in special modes, such as an appending mode.

Now, however, it's time for the "BUT" of the STL framework: In practice, there are some limits and workarounds you should know. Many of these have to do with modifications.

Several algorithms modify destination ranges. In particular, those algorithms may remove elements. If this happens, special aspects apply, which are explained in this section. These aspects are surprising and show the price of the STL concept that separates containers and algorithms with great flexibility.

6.7.1. "Removing" Elements

The `remove()` algorithm removes elements from a range. However, using this algorithm for all elements of a container operates in a surprising way. Consider the following example:

Click here to view code image

```
// stl/remove1.cpp

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

int main()
{
    list<int> coll;

    // insert elements from 6 to 1 and 1 to 6
    for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
    }

    // print all elements of the collection
    cout << "pre:   ";
    copy (coll.cbegin(), coll.cend(),          // source
          ostream_iterator<int>(cout," "));    // destination
    cout << endl;

    // remove all elements with value 3
    remove (coll.begin(), coll.end(),          // range
            3);                                // value

    // print all elements of the collection
    cout << "post: ";
    copy (coll.cbegin(), coll.cend(),          // source
          ostream_iterator<int>(cout," "));    // destination
    cout << endl;
}
```

Someone without deeper knowledge reading this program would expect that all elements with value `3` are removed from the collection. However, the output of the program is as follows:

```
pre:  6 5 4 3 2 1 1 2 3 4 5 6
post: 6 5 4 2 1 1 2 4 5 6 5 6
```

Thus, `remove()` did not change the number of elements in the collection for which it was called. The `cend()` member function returns the old end — as would `end()` — whereas `size()` returns the old number of elements. However, something has changed: The elements changed their order as if the elements had been removed. Each element with value `3` was overwritten by the following elements (see Figure 6.10). At the end of the collection, the old elements that were not overwritten by the algorithm remain unchanged. Logically, these elements no longer belong to the collection.

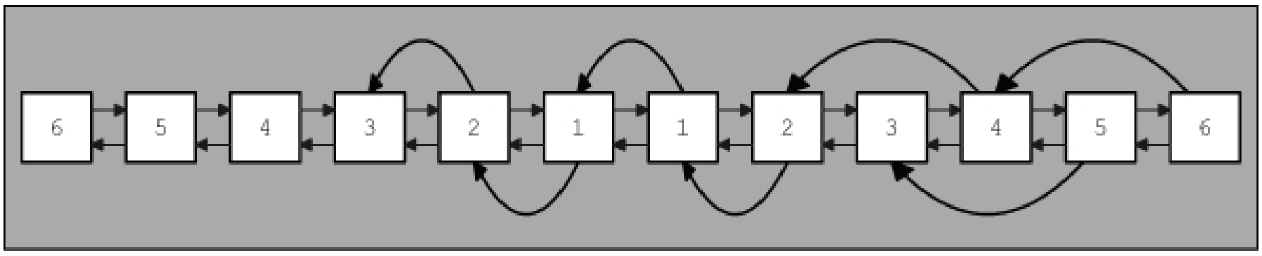

Figure 6.10. How `remove()` Operates

However, the algorithm does return the new logical end. By using the algorithm, you can access the resulting range, reduce the size of the collection, or process the number of removed elements. Consider the following modified version of the example:

```
// stl/remove2.cpp

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

int main()
{
    list<int> coll;

    //insert elements from 6 to 1 and 1 to 6
    for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
    }

    //print all elements of the collection
    copy (coll.cbegin(), coll.cend(),
          ostream_iterator<int>(cout," "));
    cout << endl;

    //remove all elements with value 3
    //- retain new end
    list<int>::iterator end = remove (coll.begin(), coll.end(),
                                      3);

    //print resulting elements of the collection
    copy (coll.begin(), end,
          ostream_iterator<int>(cout," "));
    cout << endl;

    //print number of removed elements
    cout << "number of removed elements: "
         << distance(end,coll.end()) << endl;

    //remove "removed" elements
    coll.erase (end, coll.end());

    //print all elements of the modified collection
    copy (coll.cbegin(), coll.cend(),
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

In this version, the return value of `remove()` is assigned to the iterator `end`:

Click here to view code image

```
list<int>::iterator end = remove (coll.begin(), coll.end(),
                                  3);
```

This is the new logical end of the modified collection after the elements are "removed." You can use this return value as the new end for further operations:

```
copy (coll.begin(), end,
      ostream_iterator<int>(cout," "));
```

Note that you have to use `begin()` here rather than `cbegin()` because `end` is defined a nonconstant iterator, and the begin and end of a range have to get specified by the same types.

Another possibility is to process the number of "removed" elements by processing the distance between the "logical" and the real end of the collection:

```
    cout << "number of removed elements: "
         << distance(end,coll.end()) << endl;
```

Here, a special auxiliary function for iterators, `distance()`, is used. It returns the distance between two iterators. If the iterators were random-access iterators, you could process the difference directly with operator `-`. However, the container is a list, so it provides only bidirectional iterators. See Section 9.3.3, page 445, for details about `distance()`.

If you really want to remove the "removed" elements, you must call an appropriate member function of the container. To do this, containers provide the `erase()` member function, which removes all elements of the range that is specified by its arguments:

```
    coll.erase (end, coll.end());
```

Here is the output of the whole program:

```
6 5 4 3 2 1 1 2 3 4 5 6
6 5 4 2 1 1 2 4 5 6
number of removed elements: 2
6 5 4 2 1 1 2 4 5 6
```

If you really want to remove elements in one statement, you can call the following statement:

```
    coll.erase (remove(coll.begin(),coll.end(),
                       3),
                coll.end());
```

Why don't algorithms call `erase()` by themselves? This question highlights the price of the flexibility of the STL. The STL separates data structures and algorithms by using iterators as the interface. However, iterators are an abstraction to represent a position in a container. In general, iterators do *not* know their containers. Thus, the algorithms, which use the iterators to access the elements of the container, can't call any member function for it.

This design has important consequences because it allows algorithms to operate on ranges that are different from "all elements of a container." For example, the range might be a subset of all elements of a collection. The range might even be a container that provides no `erase()` member function (an array is an example of such a container). So, to make algorithms as flexible as possible, there are good reasons not to require that iterators know their container.

Note that often it is not necessary to remove the "removed" elements. Often, it is no problem to use the returned new logical end instead of the real end of the container. In particular, you can call all algorithms with the new logical end.

6.7.2. Manipulating Associative and Unordered Containers

Manipulation algorithms — those that remove elements and those that reorder or modify elements — have another problem when you try to use them with associative or unordered containers: Associative and unordered containers can't be used as a destination. The reason is simple: If they would work for associative or unordered containers, modifying algorithms could change the value or position of elements, thereby violating the order maintained by the container (sorted for associative containers or according to the hash function for unordered containers). In order to avoid compromising the internal order, every iterator for an associative and unordered container is declared as an iterator for a constant value or key. Thus, manipulating elements of or in associative and unordered containers results in a failure at compile time.[11]

[11] Unfortunately, some systems provide really bad error handling. You see that something went wrong but have problems finding out why.

This problem also prevents you from calling removing algorithms for associative containers, because these algorithms manipulate elements implicitly. The values of "removed" elements are overwritten by the following elements that are not removed.

Now the question arises: How does one remove elements in associative containers? Well, the answer is simple: Call their member functions! Every associative and unordered container provides member functions to remove elements. For example, you can call the member function `erase()` to remove elements:

```
// stl/remove3.cpp

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

int main()
{
    // unordered set with elements from 1 to 9
    set<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // print all elements of the collection
    copy (coll.cbegin(), coll.cend(),
          ostream_iterator<int>(cout," "));
    cout << endl;

    // Remove all elements with value 3
    // - algorithm remove() does not work
```

```
        // - instead member function erase() works
        int num = coll.erase(3);

        // print number of removed elements
        cout << "number of removed elements: " << num << endl;

        // print all elements of the modified collection
        copy (coll.cbegin(), coll.cend(),
              ostream_iterator<int>(cout," "));
        cout << endl;
    }
```

Note that containers provide different `erase()` member functions. Only the form that gets the value of the element(s) to remove as a single argument returns the number of removed elements (see Section 8.7.3, page 418). Of course, when duplicates are not allowed, the return value can only be `0` or `1`, as is the case for `set`s, `map`s, `unordered_set`s, and `unordered_map`s.

The output of the program is as follows:

```
1 2 3 4 5 6 7 8 9
number of removed elements: 1
1 2 4 5 6 7 8 9
```

6.7.3. Algorithms versus Member Functions

Even if you are able to use an algorithm, it might be a bad idea to do so. A container might have member functions that provide much better performance.

Calling `remove()` for elements of a list is a good example of this. If you call `remove()` for elements of a list, the algorithm doesn't know that it is operating on a list and thus does what it does for any container: reorder the elements by changing their values. If, for example, the algorithm removes the first element, all the following elements are assigned to their previous elements. This behavior contradicts the main advantage of lists: the ability to insert, move, and remove elements by modifying the links instead of the values.

To avoid bad performance, lists provide special member functions for all manipulating algorithms. You should always prefer them. Furthermore, these member functions really remove "removed" elements, as this example shows:

```
// stl/remove4.cpp

#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // insert elements from 6 to 1 and 1 to 6
    for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
    }

    // remove all elements with value 3 (poor performance)
    coll.erase (remove(coll.begin(),coll.end(),
                       3),
                coll.end());

    // remove all elements with value 4 (good performance)
    coll.remove (4);
}
```

You should always prefer a member function over an algorithm if good performance is the goal. The problem is, you have to know that a member function exists that has significantly better performance for a certain container. No warning or error message appears if you use the `remove()` algorithm for a list. However, if you prefer a member function in these cases, you have to change the code when you switch to another container type. In the reference sections of algorithms (Chapter 11), I mention whether a member function exists that provides better performance than an algorithm.