6.4. Algorithms

The STL provides several standard algorithms for processing elements of collections. These algorithms offer general fundamental services, such as searching, sorting, copying, reordering, modifying, and numeric processing.

Algorithms are not member functions of the container classes but instead are global functions that operate with iterators. This has an important advantage: Instead of each algorithm being implemented for each container type, all are implemented only once for any container type. The algorithm might even operate on elements of different container types. You can also use the algorithms for user-defined container types. All in all, this concept reduces the amount of code and increases the power and the flexibility of the library.

Note that this is not an object-oriented programming paradigm; it is a generic functional programming paradigm. Instead of data and operations being unified, as in object-oriented programming, they are separated into distinct parts that can interact via a certain interface. However, this concept also has its price: First, the usage is not intuitive. Second, some combinations of data structures and algorithms might not work. Even worse, a combination of a container type and an algorithm might be possible but not useful (for example, it may lead to bad performance). Thus, it is important to learn the concepts and the pitfalls of the STL to benefit from it without abusing it. I provide examples and more details about this throughout the rest of this chapter.

Let's start with a simple example of the use of STL algorithms. The following program shows some algorithms and their usage:

Click here to view code image

```
// stl/algo1.cpp

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

int main()
{
    // create vector with elements from 1 to 6 in arbitrary order
    vector<int> coll = { 2, 5, 4, 1, 6, 3 };

    // find and print minimum and maximum elements
    auto minpos = min_element(coll.cbegin(),coll.cend());
    cout << "min: "  << *minpos << endl;
    auto maxpos = max_element(coll.cbegin(),coll.cend());
    cout << "max: "  << *maxpos << endl;

    // sort all elements
    sort (coll.begin(), coll.end());

    // find the first element with value 3
    // - no cbegin()/cend() because later we modify the elements pos3 refers to
    auto pos3 = find (coll.begin(), coll.end(),  // range
                      3);                        // value

    // reverse the order of the found element with value 3 and all following elements
    reverse (pos3, coll.end());

    // print all elements
    for (auto elem : coll) {
        cout << elem << ' ';
    }
    cout << endl;
}
```

To be able to call the algorithms, you must include the header file `<algorithm>` (some algorithms need special header files, see Section 11.1, page 505):

```
#include <algorithm>
```

The first two algorithms, `min_element()` and `max_element()`, are called with two parameters that define the range of the processed elements. To process all elements of a container, you simply use `cbegin()` and `cend()` or `begin()` and `end()`, respectively. Both algorithms return an iterator for the position of the element found. Thus, in the statement

```
auto minpos = min_element(coll.cbegin(),coll.cend());
```

the `min_element()` algorithm returns the position of the minimum element. (If there is more than one minimum element, the algorithm returns the first.) The next statement prints the element the iterator refers to:

```
cout << "min: " << *minpos << endl;
```

Of course, you could do both in one statement:

```
cout << *min_element(coll.cbegin(),coll.cend()) << endl;
```

The next algorithm, `sort()`, as the name indicates, sorts the elements of the range defined by the two arguments. As usual, you could pass an optional sorting criterion. The default sorting criterion is operator `<`. Thus, in this example, all elements of the container are sorted in ascending order:

```
sort (coll.begin(), coll.end());
```

Thus, afterward, the vector contains the elements in this order:

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

Note that you can't use `cbegin()` and `cend()` here, because `sort()` modifies the values of the elements, which is not possible for `const_iterator` s.

The `find()` algorithm searches for a value inside the given range. In this example, this algorithm searches for the first element that is equal to the value `3` in the whole container:

```
auto pos3 = find (coll.begin(), coll.end(),  // range
                  3);                        // value
```

If the `find()` algorithm is successful, it returns the iterator position of the element found. If the algorithm fails, it returns the end of the range passed as second argument, which is the past-the-end iterator of `coll` here. In this example, the value `3` is found as the third element, so afterward, `pos3` refers to the third element of `coll`.

The last algorithm called in the example is `reverse()`, which reverses the elements of the passed range. Here, the third element that was found by the `find()` algorithm and the past-the-end iterator are passed as arguments:

```
reverse (pos3, coll.end());
```

This call reverses the order of the third element up to the last one. Because this is a modification, we have to use a nonconstant iterator here, which explains why we called `find()` with `begin()` and `end()` instead of `cbegin()` and `cend()`. Otherwise, `pos3` would be a `const_iterator`, which would result in an error when passing it to `reverse()`.

The output of the program is as follows:

```
min: 1
max: 6
1 2 6 5 4 3
```

Note that a couple of features of C++11 are used in this example. If you have a platform that doesn't support all features of C++11, the same program might look as follows:

Click here to view code image

```
// stl/algo1old.cpp

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

int main()
{
    // create vector with elements from 1 to 6 in arbitrary order
    vector<int> coll;
    coll.push_back(2);
    coll.push_back(5);
    coll.push_back(4);
    coll.push_back(1);
    coll.push_back(6);
    coll.push_back(3);

    // find and print minimum and maximum elements
    vector<int>::const_iterator minpos = min_element(coll.begin(),
                                                     coll.end());
    cout << "min: "  << *minpos << endl;
    vector<int>::const_iterator maxpos = max_element(coll.begin(),
                                                     coll.end());
    cout << "max: "  << *maxpos << endl;
```

```
        // sort all elements
        sort (coll.begin(), coll.end());

        // find the first element with value 3
        vector<int>::iterator pos3;
        pos3 = find (coll.begin(), coll.end(),    // range
                     3);                          // value

        // reverse the order of the found element with value 3 and all following elements
        reverse (pos3, coll.end());

        // print all elements
        vector<int>::const_iterator pos;
        for (pos=coll.begin(); pos!=coll.end(); ++pos) {
            cout << *pos << ' ';
        }
        cout << endl;
    }
```

The differences are as follows:

- No initializer list can be used to initialize the vector.
- Members `cbegin()` and `cend()` are not provided, so you have to use `begin()` and `end()` instead. But you can still use `const_iterator` s.
- Instead of using `auto` , you always have to declare the iterators explicitly.
- Instead of range-based `for` loops, you have to use iterators to output each element.

6.4.1. Ranges

All algorithms process one or more *ranges* of elements. Such a range might, but is not required to, embrace all elements of a container. Therefore, to be able to handle subsets of container elements, you pass the beginning and the end of the range as two separate arguments rather than the whole collection as one argument.

This interface is flexible but dangerous. The caller must ensure that the first and second arguments define a *valid* range. A range is valid if the end of the range is *reachable* from the beginning by iterating through the elements. This means that it is up to the programmer to ensure that both iterators belong to the same container and that the beginning is not behind the end. Otherwise, the behavior is undefined, and endless loops or forbidden memory access may result. In this respect, iterators are just as unsafe as ordinary pointers. But undefined behavior also means that an implementation of the STL is free to find such kinds of errors and handle them accordingly. The following paragraphs show that ensuring that ranges are valid is not always as easy as it sounds. See Section 6.12, page 245, for more details about the pitfalls and safe versions of the STL.

Every algorithm processes *half-open* ranges. Thus, a range is defined so that it includes the position used as the beginning of the range but excludes the position used as the end. This concept is often described by using the traditional mathematical notations for half-open ranges:

$$[\,begin, end\,)$$

or

$$[\,begin, end\,[$$

In this book, I use the first alternative.

The half-open-range concept has the advantages that it is simple and avoids special handling for empty collections (see Section 6.3, page 189). However, it also has some disadvantages. Consider the following example:

Click here to view code image

```
    // stl/find1.cpp

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

    int main()
    {
        list<int> coll;

        // insert elements from 20 to 40
        for (int i=20; i<=40; ++i) {
            coll.push_back(i);
        }

        // find position of element with value 3
        // - there is none, so pos3 gets coll.end()
        auto pos3 = find (coll.begin(), coll.end(),      // range
                                  3);                              // value

        // reverse the order of elements between found element and the end
        // - because pos3 is coll.end() it reverses an empty range
        reverse (pos3, coll.end());

        // find positions of values 25 and 35
        list<int>::iterator pos25, pos35;
        pos25 = find (coll.begin(), coll.end(),     // range
                      25);                          // value
        pos35 = find (coll.begin(), coll.end(),     // range
                      35);                          // value

        // print the maximum of the corresponding range
        // - note: including pos25 but excluding pos35
        cout << "max: " << *max_element (pos25, pos35) << endl;

        // process the elements including the last position
        cout << "max: " << *max_element (pos25, ++pos35) << endl;
    }
```

In this example, the collection is initialized with integral values from `20` to `40`. When the search for an element with the value `3` fails, `find()` returns the end of the processed range (`coll.end()` in this example) and assigns it to `pos3`. Using that return value as the beginning of the range in the following call of `reverse()` poses no problem, because it results in the following call:

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

This is simply a call to reverse an empty range. Thus, it is an operation that has no effect (a so-called “no-op”).

However, if `find()` is used to find the first and the last elements of a subset, you should consider that passing these iterator positions as a range will exclude the last element. So, the first call of

```
max_element()
   max_element (pos25, pos35)
```

finds `34` and not `35`:

```
max: 34
```

To process the last element, you have to pass the position that is one past the last element:

```
max_element (pos25, ++pos35)
```

Doing this yields the correct result:

```
max: 35
```

Note that this example uses a list as the container. Thus, you must use operator `++` to get the position that is behind `pos35`. If you have random-access iterators, as with vectors and deques, you also could use the expression `pos35 + 1` because random-access iterators allow *iterator arithmetic* (see Section 6.3.2, page 198, and Section 9.2.5, page 438, for details).

Of course, you could use `pos25` and `pos35` to find something in that subrange. Again, to search including `pos35`, you have to pass the position after `pos35`. For example:

```
// increment pos35 to search with its value included
++pos35;
pos30 = find(pos25,pos35,       // range
             30);               // value
if (pos30 == pos35) {
    cout << "30 is NOT in the subrange" << endl;
}
else {
    cout << "30 is in the subrange" << endl;
}
```

All the examples in this section work only because you know that `pos25` is in front of `pos35`. Otherwise, [`pos25`, `pos35`) would not be a valid range. If you are not sure which element is in front, things get more complicated, and undefined behavior may easily occur.

Suppose that you don’t know whether the element having value `25` is in front of the element having value `35`. It might even be possible that one or both values are not present. By using random-access iterators, you can call operator `<` to check this:

```
if (pos25 < pos35) {
        // only [pos25,pos35) is valid
        ...
    }
    else if (pos35 < pos25) {
        // only [pos35,pos25) is valid
        ...
    }
    else {
        // both are equal, so both must be end()
        ...
    }
```

However, without random-access iterators, you have no simple, fast way to find out which iterator is in front. You can only search for one iterator in the range of the beginning to the other iterator or in the range of the other iterator to the end. In this case, you should change your algorithm as follows: Instead of searching for both values in the whole source range, you should try to find out, while searching for them, which value comes first. For example:

```
    pos25 = find (coll.begin(), coll.end(),     // range
                  25);                          // value
    pos35 = find (coll.begin(), pos25,          // range
                  35);                          // value
    if (pos25 != coll.end() && pos35 != pos25) {
        // pos35 is in front of pos25
        // so, only [pos35,pos25) is valid
        ...
    }
    else {
        pos35 = find (pos25, coll.end(),        // range
                      35);                      // value
        if (pos35 != coll.end()) {
            // pos25 is in front of pos35
            // so, only [pos25,pos35) is valid
            ...
        }
        else {
            // 25 and/or 35 not found
            ...
        }
    }
```

In contrast to the previous version, you don't search for `35` in the full range of all elements of `coll`. Instead, you first search for it from the beginning to `pos25`. Then, if it's not found, you search for it in the part that contains the remaining elements after `pos25`. As a result, you know which iterator position comes first and which subrange is valid.

This implementation is not very efficient. A more efficient way to find the first element that has either value `25` or value `35` is to search exactly for that. You could do this by using `find_if()` and passing a lambda (see Section 3.1.10, page 28), defining the criterion that is evaluated with each element in `coll`:

```
    pos = find_if (coll.begin(), coll.end(),   // range
                   [] (int i) {                // criterion
                       return i == 25 || i == 35;
                   });
    if (pos == coll.end()) {
        // no element with value 25 or 35 found
        ...
    }
    else if (*pos == 25) {

        // element with value 25 comes first
        pos25 = pos;
        pos35 = find (++pos, coll.end(),         // range
                      35);                       // value
        ...
    }
    else {
        // element with value 35 comes first
        pos35 = pos;
        pos25 = find (++pos, coll.end(),         // range
                      25);                       // value
        ...
    }
```

Here, the special lambda expression

```
    [] (int i) {
        return i == 25 || i == 35;
    }
```

is used as a criterion that allows a search of the first element that has either value `25` or value `35`. The use of lambdas in the STL is introduced in Section 6.9, page 229, and discussed in detail in Section 10.3, page 499.

6.4.2. Handling Multiple Ranges

Several algorithms process more than one range. In this case, you usually must define both the beginning and the end only for the first range. For all other ranges, you need to pass only their beginnings. The ends of the other ranges follow from the number of elements in the first range. For example, the following call of `equal()` compares all elements of the collection `coll1` element-by-element with the elements of `coll2`, beginning with its first element:

```
    if (equal (coll1.begin(), coll1.end(),      //first range
               coll2.begin())) {                //second range
        ...
    }
```

Thus, the number of elements of `coll2` that are compared with the elements of `coll1` is specified indirectly by the number of elements in `coll1` (see Figure 6.9).

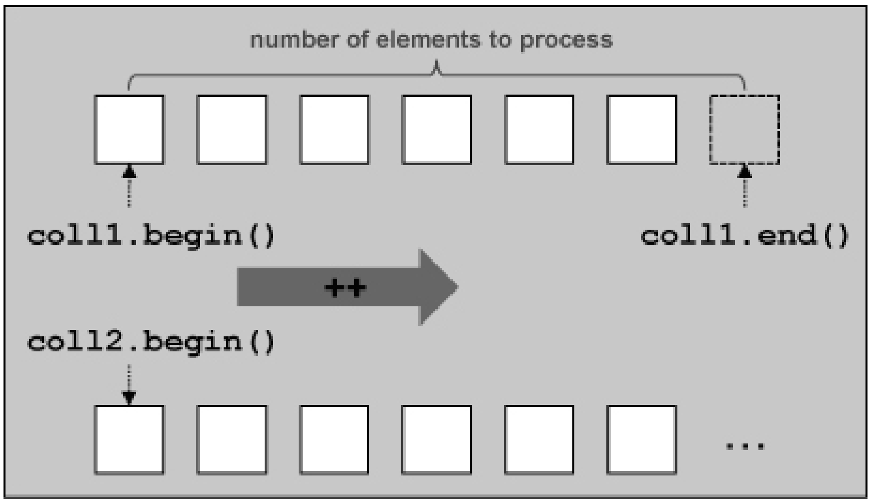

Figure 6.9. Algorithm Iterating over Two Ranges

This leads to an important consequence: **When you call algorithms for multiple ranges, make sure that the second and additional ranges have at least as many elements as the first range.** In particular, make sure that destination ranges are big enough for algorithms that write to collections.

Consider the following program:

```
    // stl/copybug.cpp

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

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

        //RUNTIME ERROR:
        //- overwrites nonexisting elements in the destination
        copy (coll1.cbegin(), coll1.cend(),     //source
              coll2.begin());                   //destination
        ...
    }
```

Here, the `copy()` algorithm is called. It simply copies all elements of the first range into the destination range. As usual, the beginning and the end are defined for the first range, whereas only the beginning is specified for the second range. However, the algorithm overwrites rather than inserts. So, the algorithm *requires* that the destination has enough elements to be overwritten. If there is not enough room, as in this case, the result is undefined behavior. In practice, this often means that you overwrite whatever comes after the `coll2.end()`. If you're in luck, you'll get a crash, so at least you'll know that you did something wrong. However, you can force your luck by using a safe version of the STL for which the undefined behavior is defined as leading to a certain error-handling procedure (see Section 6.12.1, page 247).

To avoid these errors, you can (1) ensure that the destination has enough elements on entry, or (2) use *insert iterators*. Insert iterators are covered in Section 6.5.1, page 210. I'll first explain how to modify the destination so that it is big enough on entry.

To make the destination big enough, you must either create it with the correct size or change its size explicitly. Both alternatives apply only to some sequence containers (`vector` , `deque` , `list` , and `forward_list`). However, this is not really a problem for other containers, because associative and unordered containers cannot be used as a destination for overwriting algorithms (Section 6.7.2, page 221, explains why). The following program shows how to increase the size of containers:

Click here to view code image

```
// stl/copy1.cpp

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

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

    // resize destination to have enough room for the overwriting algorithm
    coll2.resize (coll1.size());

    // copy elements from first into second collection
    // - overwrites existing elements in destination
    copy (coll1.cbegin(), coll1.cend(),      // source
          coll2.begin());                    // destination

    // create third collection with enough room
    // - initial size is passed as parameter
    deque<int> coll3(coll1.size());

    // copy elements from first into third collection
    copy (coll1.cbegin(), coll1.cend(),      // source
          coll3.begin());                    // destination
}
```

Here, `resize()` is used to change the number of elements in the existing container `coll2` :

```
coll2.resize (coll1.size());
```

Later, `coll3` is initialized with a special initial size so that it has enough room for all elements of `coll1` :

```
deque<int> coll3(coll1.size());
```

Note that both resizing and initializing the size create new elements. These elements are initialized by their default constructor because no arguments are passed to them. You can pass an additional argument both for the constructor and for `resize()` to initialize the new elements.