An Introduction to the Standard Template Library (STL)

Part II: Algorithms and Function Objects

Carlos Moreno

Table of Contents

Algorithms

The Standard Template Library provides a number of useful, generic algorithms to perform the most commonly used operations on groups/sequences of elements. These operations include traversals, searching, sorting and insertion/removal of elements.

The way that these algorithms are implemented is closely related to the idea of the containers and iterators, although, as we will see, they can be used with standard arrays and pointers. (in fact, we could use most of the STL algorithms in a program that uses exclusively C idioms)

Suppose that we want to find a particular value in a linked list (or in a vector, or any other container - it doesn't make any difference). We could do the following: (the example assumes that the values are integers)

```
list<int> values;
int search_value;

// ... code to fill values

list<int>::iterator i;
for (i = values.begin(); i != values.end(); ++i)
{
   if (*i == search_value)
   {
      break;
   }
}

if (i != values.end())
{
    // Found! Use now *i if necessary
}
else
{
   // Not found! Do whatever is required
}
```

We may want to provide a function **`find`** , defined as follows:

```
list<int>::iterator find (const list<int> & lst, int search_value)
{
   list<int>::iterator i;
   for (i = lst.begin(); i != lst.end(); ++i)
   {
      if (*i == search_value)
      {
         break;
      }
   }

   return i; // will return lst.end() if the value was not found
```

```
}
```

of course, this would be limited to search in the entire container; we could provide a more generic function (thus, better chances of reusability) that receives a search range within the container:

```
list<int>::iterator find (list<int>::iterator start,
list<int>::iterator end,
int search_value)
{
   list<int>::iterator i;
   for (i = start; i != end; ++i)
   {
      if (*i == search_value)
      {
         break;
      }
   }

   return i; // will return end if the value was not found
}
```

And this function can still be used to search through the entire container, as shown below:

```
list<int>::iterator i = find (values.begin(), values.end(), search_value);

if (i != values.end()) // ...
```

This example of the **find** function is one of the many algorithms provided in Standard Template Library. Of course, the previous example worked only on a linked list of integers, which makes it totally non-generic. The definitions of the Standard Library functions do not take arguments of specific container types. Instead, the functions are defined as template functions, which allows the programmer to use them directly with any type of container, and even with standard arrays and pointers!

The prototype for the function **find** , according to the standard, is the following:

```
template <class Iterator, class T>
Iterator find (Iterator first, Iterator last, const T & value);
```

The implementation could be something along these lines:

```
for (Iterator i = first; i != last && *i != value; ++i);
return i;
```

Notice that this function can be used passing a pointer as the first two arguments, or passing an **iterator** , or a **const_iterator** - in general, it can be any type that supports the unary ***** and **++** operators; also, the third parameter can be any value that supports comparison with the type of ***i** .

Algorithms requiring operations on the elements

The example shown above, though it illustrates an interesting approach, is rather limited. For instance, what if we want to find the first occurence of a positive value? Or the first prime number in the sequence? Or, working with a sequence of strings, what if we want to find the first string that contains no spaces?

The common denominator to these situations is that I am now describing an algorithm similar to find, only that we don't search for a particular value, but rather for an element that matches a predicate (a condition). This algorithm is part of the STL, and it is called **`find_if`**.

The implementation of this algorithm could be similar to the following:

```
template <class Iterator, class Function>
void find_if (Iterator first, Iterator last, Function predicate)
{
   for (Iterator i = first, i != last; ++i)
   {
      if (predicate(*i)) // found!
      {
         return i;
      }
   }

   return i; // returns last if not found
}
```

Now, the parameter **`predicate`** (which is templatized) can be anything that supports the expression **`predicate(x)`** and returns bool (or something convertible to bool). An obvious example is a pointer to a function that takes one argument of the same type as **`*i`** , and returns bool. But let's not go there. (I mean, let's face it: pointers to functions? eww!!).

Another -- less obvious, perhaps -- example is an object that supports the expression **`predicate(x)`**. That is, an object that provides the overloaded **`operator()`** (yes, a pair of parenthesis *is* an operator!). This object that acts as a predicate in this example is called a "function object" or "functor" -- it is an object whose purpose is limited to emulating a function. (Before you dismiss this idea as being insane and stop liking the STL, please keep reading.)

Below is an example of use of the **`find_if`** algorithm to check if a list of integers contains any negative values:

```
class is_negative
{
public:
   bool operator() (int value) const
   {
      return value < 0;
   }
};


list<int> values;

// ... fill the list

if (find_if (values.begin(), values.end(),
             is_negative()) != values.end())
{
   // yes, it contains at least one negative number
}
```

In this case, the compiler will instantiate a version of the **`find_if`** template with the first two parameters of type **`list<int>::iterator`** , and the last parameter of type **`is_negative`** . Notice the pair of brackets after the name is_negative. We are passing an object of class is_negative, which is instantiated on-the-fly to be passed to the function. That same object "lives" throughout the entire execution of the loop inside **`find_if`**, and the expression **`predicate(*i)`** inside find_if simply calls the member function **`operator()`** (passing an int as parameter -- each element of the list, since that's what we get when we dereference the iterator)

Of course, this function could also be used with a regular array of integers, as shown below:

```
int values[SIZE] = { ... };

if (find_if (values, values + SIZE, is_negative()) != values + SIZE)
```

At this point, I can hear you screaming "Oh my God, you and all the people that created the STL are insane" (them, for creating it, and me, for humoring them and actually write a tutorial on it).

There are several advantages with this approach, one of them is the extra flexibility that we get from the fact that we are using ***an object*** as the predicate, and that object can hold some extra information, if needed. Let's face it: this example shows an approach that is more flexible than the find example, but it is still limited: what if we want to search the first value that it is less than 5, or less than -5? Or less than a number specified by the user? The above example can be easily extended to provide that extra flexibility, since we can add data members to the function object, and hold some data that we pass it when instantiating the object.

The example below shows the use of `find_if` to find the first element that is greater than 5

```
class is_greater_than
{
public:
   is_greater_than (int n)
     : value(n)
   {}

   bool operator() (int element) const
   {
      return element > value;
   }

private:
   int value;
};


list<int> values;

// ... fill the list

if (find_if (values.begin(), values.end(),
            is_greater_than(5)) != values.end())
{
   // yes, it contains at least one number greater than 5
}
```

Bonus, for some extra fun: we could have provided a template class, and then we could use that function object with a sequence of doubles, or ints, or strings, etc:

```
template <typename T>
class is_greater_than
{
public:
   is_greater_than (const T & n)
      : value(n)
   {}

   bool operator() (const T & element) const
   {
      return element > value;
   }

private:
   T value;
};
```

And then we would call it like this:

```
if (find_if (values.begin(), values.end(),
             is_greater_than<int> (5)) != values.end())
```

The algorithms **`find`** and **`find_if`** are simply one example of the many available algorithms. The Standard Library algorithms cover most of the commonly used operations on sequences of elements, including traversal, searching, sorting and insertion/removal of elements. Some of the algorithms provided by the Standard Library are listed and briefly described below: (for a complete list and detailed description, consult a reference book)

Non modifying operations:

`for_each`	Do specified operation for each element in a sequence
`find`	Find the first occurence of a specified value in a sequence
`find_if`	Find the first match of a predicate in a sequence
`find_first_of`	Find the first occurence of a value from one sequence in another
`adjacent_find`	Find the first occurence of an adjacent pair of values
`count`	Count occurences of a value in a sequence
`count_if`	Count matches of a predicate in a sequence
`accumulate`	Accumulate (i.e., obtain the sum of) the elements of a sequence
`equal`	Compare two ranges
`max_element`	Find the highest element in a sequence
`min_element`	Find the lowest element in a sequence

Modifying operations:

`transform`	Apply an operation to each element in an input sequence and store the result in an output sequence (possibly the same input sequence)
`copy`	Copy a sequence
`replace`	Replace elements in a sequence with a specified value
`replace_if`	Replace elements matching a predicate
`remove`	Remove elements with a specified value
`remove_if`	Remove elements matching a predicate
`reverse`	Reverses a sequence
`random_shuffle`	Randomly reorganize elements using a uniform distribution
`fill`	Fill a sequence with a given value
`generate`	Fill a sequence with the result of a given operation

Sorting:

`sort`	Sort elements
`stable_sort`	Sort maintaining the order of equal elements
`nth_element`	Put n^{th} element in its place
`binary_search`	Find a value in a sequence, performing binary search

Making use of the standard algorithms, we could rewrite the example of the linked list of students as follows:

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

class Student
{
public:
    // ... various functions to perform the required operations

private:
    string name, ID;
```

```
    int mark;
};

class print_if_failed
{
public:
    void operator() (const Student & s) const
    {
        // ...
    }
};

class failed
{
public:
    bool operator() (const Student & s) const
    {
        // ...
    }
};


int main()
{
    list<Student> students;

    // Read from data base

    while (more_students())
    {
        Student temp;
        temp.read();
        students.push_back (temp);
    }

    // Print the students that failed

    for_each (students.begin(), students.end(), print_if_failed())

    // Now remove the failed students

    remove_if (students.begin(), students.end(), failed());
                                   // see note below

    // ...

    return 0;
}
```

Note: the function **`remove_if`** will not actually remove the elements from the list. It can not. This may sound strange, but remember that the algorithms are generic, and decoupled from the containers. They work with iterators, and not with containers (not directly, at least). Since this algorithm could be used with different types of containers, it can not know how to physically remove the elements. Even more: since it only receives iterators, it can not even figure out what container the elements are in. (still not convinced? It gets worse: what if you use **`remove`** with a C-style array of elements, where there is no such thing as physically eliminating elements?)

So, **`remove`** and **`remove_if`** only reorganize the elements, moving the "removed" elements to the end of the sequence, and returning an iterator that indicates the first element that was "removed" (i.e., the first element that is not part of the resulting sequence). If we must actually remove the elements, we should use the following:

```
students.erase (remove_if (students.begin(), students.end(), failed),
                students.end());
```

In other words, we have **`remove`**/**`remove_if`** determine the elements to be removed (and reorganize the sequence accordingly), and then we ask the container to get rid of the elements that we don't want.

Operations other than Unary Predicates

The examples of **find_if**, **count_if**, **remove_if** have one common detail: they work with operations that represent a unary predicate (a condition on one element). We use them with function objects for which the **operator()** returns bool. Function objects may represent operations that are not necessarily a predicate. An obvious example is the algorithm **transform**. This algorithm receives four parameters: two iterators to specify the input sequence, one to specify the output sequence (client code is responsible of making sure there is enough room in the output sequence), and the operation. In this case, the operation represents a function that returns an output value given an input value (well, all functions do that -- what I mean is that its output value represents the result of an operation, and not just a boolean indicating if the input value matches a predicate).

Below is an example of using **transform** to obtain the lowercase equivalent of a **string** (yes, a **string** can be used with STL algorithms -- it is a "quasi-container", in that it provides iterators, **begin()** and **end()** and other methods that make it compatible with STL containers):

```
class to_lower
{
public:
    char operator() (char c) const        // notice the return type
    {
        return tolower(c);
    }
};

string lower (const string & str)
{
    string lcase = str;
    transform (str.begin(), str.end(), lcase.begin(), to_lower());

    return lcase;
}
```

The **transform** line could have been:

```
transform (lcase.begin(), lcase.end(), lcase.begin(), to_lower())
```

(remember that the output sequence can be the same input sequence, if we want in-place transformations)

Some algorithms require a binary predicate, such as a user-provided comparison function. For instance, we may want to find the student with highest grade in the list of students. **max_element** seems to be the algorithm that would do that; except that **max_element** compares elements (as in, uses < to compare), and Student objects do not support comparison (i.e., class Student does not provide an overloaded operator<).

However, **max_element** (and **min_element**, and **sort**, etc.) come in two (overloaded) versions: one that only receives the sequence (i.e., two iterators), and one that receives the sequence, plus an operation representing the custom-defined comparison. All the algorithms that receive a comparison operation require an operation that emulates **operator<**; (i.e., they should be function objects acting as a binary predicate -- the **operator()** method should receive two parameters and return true if the first parameter is "less than" the second (whatever "less than" means). Let's see that "find the student with highest grade" example to illustrate the above:

```
class cmp_grades
{
public:
    bool operator() (const Student & s1, const Student & s2) const
    {
        return s1.grade() < s2.grade();        // cmp grades emulating "less than"
    }
};

    // Somewhere in the main program...
list<Student>::iterator best_grade = max_element (students.begin(), students.end(),
                                                  cmp_grades());

    // or sort them by grades:
sort (students.begin(), students.end(), cmp_grades());
```

Standard Library Function Objects

The STL provides a handful of ready-to-use function object classes, including predicates and arithmetic operations. These function objects are found in the `<functional>` library facility (i.e., we `#include <functional>` to use them).

The predicates include comparisons and logical operations, provided in the form of template classes, including the following: `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal`, `less_equal` (and a few others that I will omit in this tutorial).

These are binary predicates that can be used combined with algorithms that expect an operation. The implementation of these function objects is pretty straightforward. Except for one detail that is irrelevant for the purpose of this discussion, the implementation could be similar to this:

```
template <typename T>
class greater
{
public:
    bool operator() (const T & v1, const T & v2) const
    {
        return v1 > v2;
    }
};
```

For instance, we could use this function object greater to sort a sequence in descending order:

```
vector<int> values;
// ... add elements...

sort (values.begin(), values.end(), greater<int>());
```

The trick is that the third parameter is an operation that will be used instead of direct comparison, and that operation is supposed to emulate the "less-than" comparison. If we use "greater-than" instead, we are "lying" to the algorithm and always giving the opposite result -- the outcome is that the sequence ends up sorted in the exact opposite order.

The function objects representing arithmetic operations include `plus`, `minus`, `multiplies`, and `divides` (and a couple others that I will omit). These are binary operations that return the sum, difference, product, or division of the first argument and the second (in that order). You can imagine that their implementation is also straightforward.

We can use the `multiplies` function object to obtain the product of all the numbers in a sequence as shown below:

```
list<double> values;
// ... add elements ...

double product = accumulate (values.begin(), values.end(), 1.0,
                             multiplies<double>());
```

The trick here is that the user-provided operation is supposed to replace direct addition (e.g., we may want to accumulate the grades of all the students, or accumulate the lengths of a group of strings, etc.). We provide an operation that multiplies instead of adding.

Some More Insanity

Ok, now for the real fun!

Let's continue with those comparison and arithmetic function objects, and let's say that we want to transform a sequence to twice the original values (i.e., take an input sequence and transform it to an output sequence -- possibly in-place -- that contains the values multiplied by 2). Of course, we want to do it without having to provide our own function object class.

Seems trivial, right? After all, we know that `transform` expects an operation, and we also know that we have a function object `multiplies`, that performs precisely the operation that we need. (...or does it?)

Or, say that we want to count the elements greater than 5 -- we already did that, but again, we want to do it without having to provide our own function object class.

Another triviality, right? We have **count_if**, which expects a predicate, and we have the function object **greater**, that provides precisely the predicate that we need. (...or does it?)

In both cases, we'll hit the same wall: the function object required should be unary: in the first case, one input value (each element of the sequence), and the output value should be twice the input value; and in the second case, one input value, and the output should be the result of testing if the input value is greater than five.

The Standard Library function objects are binary -- they work with algorithms that require comparisons between elements of the sequence, or that perform operations on pairs of elements.

The solution in both cases is also in the STL: binders. A binder allows us to convert a binary function to a unary function, by binding one of the arguments to a given value (given at runtime).

For instance, to count the elements greater than five, we use the function object **greater**, and we bind its second argument to 5 (such that it now becomes a unary predicate that tests if an input value is greater than 5). In this example, we would use **bind2nd** (to bind the second argument to a given value):

```
int num_values = count_if (values.begin(), values.end(),
                           bind2nd (greater<int>(), 5));
```

(we read the rightmost expression as follows: bind the second parameter of **greater<int>** to 5)

In our other example, we wanted to transform a sequence to twice its values. In this case, too, we use **bind2nd**:

```
transform (values.begin(), values.end(), values.begin(),
           bind2nd (multiplies<int>(), 2));
```

(or, equivalently: **bind1st (multiplies<int>(), 2)** -- do you see why? Can you re-write the first example using **bind1st** instead of **bind2nd**?)

Other examples of use for **bind2nd** are: (hopefully, the examples will speak for themselves)

```
    // unless specified, the examples assume that values is a vector<int>

int num_positives = count_if (values.begin(), values.end(),
                              bind2nd (greater<int>(), 0));

bool has_negative_values =
     find_if (values.begin(), values.end(),
              bind2nd (less<int>(), 0)) != values.end();
```

And speaking of adapters (binders are one particular type of adapters -- auxiliary classes that "adapt" some particular function object to meet some slightly different requirement), wouldn't you just hate it if you had to provide your own function object class to simply call a member function of the element? Say that you want to print all the Student objects, and let's assume that class Student provides a member function print() to do that. You would certainly be tempted to use **for_each**; only that **for_each** expects a function object that will be called, passing each element as parameter! We'd have to do something like:

```
class print
{
public:
    void operator() (const Student & s) const
    {
        s.print();
    }
};
```

It just looks silly! Mainly when we learn that we already have a ready-to-use adapter that adapts the function call f(x) to a call to one of the methods of x, e.g., x.f(), or x.print(), etc. This adapter is **mem_fun_ref** (or **mem_fun**, if used with a container of pointers).

The example of printing all the students, using this technique, would be as follows:

```
for_each (students.begin(), students.end(),
          mem_fun_ref (Student::print));
```

Notice that **mem_fun_ref** receives a pointer-to-member-function, specifying which member function to call (actually, that could have been &Student::print). Even if you don't quite understand this pointer-to-member-functions concept, you have nothing to worry about, just remember the syntax: you specify which member function to call by its name, qualified with the class name using the scope resolution operator).

Another example: given a sequence of strings, you want to store the lengths of those strings in another sequence. The algorithm **transform** comes to mind, of course. The operation being the length of the string; but length is a member function of class string -- no problem, we adapt it using **mem_fun_ref**:

```
vector<string> words;
// ...
vector <int> lengths(words.size());    // make sure we have enough room

transform (words.begin(), words.end(), lengths.begin(),
           mem_fun_ref (&string::length));
```

The first example reminds me that I haven't mentioned stream iterators, a very useful tool from the STL. That example of **for_each** combined with an adapted version of **print** is a twisted view of something so simple as copying the contents of a vector to **stdout** -- copying from one sequence to another (because **stdout** is a stream, which after all is an output sequence).

Sequences are manipulated through iterators -- we iterate through the elements in the sequence. Not surprisingly, we can use **copy** instead of **for_each** in that example. Since **copy** (or **transform**) uses an output iterator to copy one element at a time to the destination sequence, we could use **copy** combined with a class that acts like an iterator, and that is able to copy one element at a time to a given output stream. Such class is part of the STL, and it is called **ostream_iterator**. We initialize an **ostream_iterator** with the output stream to which it should be attached (e.g., **cout**), and the separator string we want between the elements that are sent to that output stream. We must specify the type of the elements through which we iterate, by specifying the template parameter for **ostream_iterator**.

ostream_iterator relies on the **operator<<** for the elements' type; so, for built-in types it always works, and for user-defined types, it works if the particular type provides an **operator<<** compatible with the natural semantics of the stream insertion (<<) operator.

So, finally, let's see the example of printing all the students using copy and an **ostream_iterator**:

```
copy (students.begin(), students.end(),
      ostream_iterator<Student> (cout, "\n"));
```

Piece of cake, huh? :-)

And once again, speaking of this makes me think of the other example (storing the lengths of the strings in a vector of ints), and reminds me of how annoying it is that we always have to resize the output container so that there is enough room for the result. You know, annoying because it looks like we simply want to append every element to the output sequence. Then again, algorithms only know about iterators, and not about the container. And regular iterators don't know anything about the container: they just know how to do something with one element and advance to the next element.

The exception to this are the insert iterators -- they are special types of iterators that *do know* how to insert elements to a container. They are initialized with the container itself, so that they can keep track of it and ask it to insert the elements. There is a couple of auxiliary functions, **back_inserter** and **front_inserter** that create the **insert_iterator** of the right type (depending on the container) for us.

So, we use **back_inserter** whenever we have an algorithm that is copying to an output container that is initially empty (or when we just want to append the output sequence to the container). It's really simple (honest!). If you don't believe me, just take a look at this:

```
vector<string> words;
// ...
vector<int> lengths;

transform (words.begin(), words.end(),
```

```
             back_inserter (lengths),
             mem_fun_ref (&string::length));
```

Before I forget! How about **istream_iterators**? We saw how to copy from a container to an output stream (using **ostream_iterator**). But how about copying from an input stream (e.g., an input file, or **stdin**) to a container? The non-trivial part is that we would have to specify a begin and end for the input sequence -- but how do we specify the "end" of an input stream? We can "detect" the fact that we reached the end (when we exhaust the input stream -- e.g., we reach end-of-file, or we encouter an end-of-input character in stdin), but how do we obtain an iterator that indicates the "end of the input sequence"?

The trick is that the input stream iterator is set to a "magic value" when the associated input stream reached the end. This magic value can be indicated by instantiating the istream_iterator with no parameters.

So, an example: let's read words from a text file and store them in a vector<string> (initially empty):

```
vector<string> words;
ifstream file ("words.txt");
if (file)
{
    copy (istream_iterator<string> (file), istream_iterator<string>(),
          back_inserter (words));
}
```

How about taking an input text file that contains numbers, and create a file with the values corresponding to twice the values from the input file -- besides the lines for opening the files and validating, this is one-liner, using STL facilities:

```
transform (istream_iterator<int> (input_file), istream_iterator<int> (),
           ostream_iterator<int> (output_file, "\n"),
           bind2nd (multiplies<int>(), 2));
```

There is a lot more about the STL, but remember that this is only an introductory tutorial. I hope it helps you understand the basics (and some of the not-so-basics) and gets you curious enough to investigate further!

Bibliography

Bjarne Stroupstrup. The C++ Programming Language, 3rd Edition.
Addison-Wesley, 1997.

Matthew H. Austern. Generic Programming and the STL.
Addison-Wesley, 1999.

Recommended Reading:

Scott Meyers. Effective STL.
Addison-Wesley, 2001.

Main Page Tutorials