An Introduction to the Standard Template Library (STL)

Carlos Moreno

Table of Contents

Introduction

The Standard Library is a fundamental part of the C++ Standard. It provides C++ programmers with a comprehensive set of efficiently implemented tools and facilities that can be used for most types of applications.

In this article, I present an introduction/tutorial on the Standard Template Library, which is the most important section of the Standard Library. I briefly present the fundamental concepts in the STL, showing code examples to help you understand these concepts. The article requires and assumes previous knowledge of the basic language features of the C++, in particular, templates (both function templates and class templates).

For a detailed description or listing of all the STL facilities, you should consult a reference material, such as Bjarne Stroustrup's *The C++ Programming Language, 3rd edition* , or Matthew Austern's *Generic Programming and the STL*.

Containers and Iterators

Pop quiz: What do arrays and linked lists have in common?

a) Nothing
b) Some features
c) Everything

If you answered a), please keep reading: you will be surprised! (if you answered b) or c), you may still want to keep reading, even if you won't be as surprised as those that answered a) :-)

In particular, let's take a look at a simple algorithm and try to implement it using arrays and using linked lists. So, given a group of elements (say, integer values, to keep this example as simple as possible), we want to find the highest element.

A way of doing this would be to have a variable (say, HIGH) where we store the first element, and then, for each other element in the group, if its value is higher than HIGH, then we assign HIGH with the value of that particular element. The pseudo-code for this algorithm could be as follows:

```
HIGH = first element
current_element = second element
while current_element is within the group of elements
  if current_element > HIGH, then HIGH = current_element
Advance to the next element
end while
```

Notice that this pseudo-coded algorithm is valid for a group of elements, regardless how exactly those elements are stored (of course, provided that we are able to perform the required tests). Let's try to implement it for both linked lists and arrays:

Linked lists:

```
struct Element // This is an extremely simplified definition,
{              // but enough for this example.
   int value;
   struct Element * next;
};

int high = list->value; // list points to the first element

struct Element * current = list->next;
                       // refers (points) to second element

while (current != NULL) // test if within the group of elements
{
   if (current->value > high)
   {
      high = current->value;
   }
   current = current->next; // Advance to next element
}
```

Arrays:

```
int high = *array;
int * one_past_end = array + size;
int * current = array + 1;   // starts at second element

while (current != one_past_end) // test if within the group of elements
{
   if (*current > high)
   {
      high = *current;
   }
   current++;     // Advance to the next element
}
```

Surprise! Both fragments of code are almost identical. (of course, I used pointer notation for the array to make them look even more similar) It's just the syntax that we use to manipulate and access the elements what changes. Notice that in both cases we have a pointer pointing to the current element. This pointer is compared to a particular value to test if we are within the group of values. Also, the pointer is dereferenced (in different concrete ways in both cases, but both are dereferencing operations) to obtain the particular value. This pointer allows us to advance to the next element (again, in different concrete ways, but still, in both cases we make the pointer point to the next element)

There is one important detail that makes the two examples conceptually identical: in this case, both data structures (array and linked list) are treated as a sequential group of elements; in both cases, the operations required are:

1. point to a particular element
2. access the element that is pointed
3. point to the next element
4. test if we are pointing within the group of elements

Notice that with these operations, we can implement any algorithm that requires sequential access to the elements of a group.

This example is based on the idea of the STL containers and iterators. A container is an object that represents a group of elements of a certain type, stored in a way that depends on the type of container (i.e., array, linked list, etc.). An iterator is a pointer-like object (that is, an object that supports pointer operations) that is able to "point" to a specific element in the container.

The interesting (and brilliant!) detail about the STL containers, is that the STL introduces the idea of a generic type of container, for which the operations and element manipulations are identical regardless the type of container that you are using (for example, you would (or could) code exactly the same algorithm to find the highest element in an array as you would to find the highest element in a linked list. This sounds like easy to say, but not-so-easy to implement.

The STL takes advantage of the operator overloading feature of the C++ language to provide class definitions for the iterators to represent pointer-like objects. In general, the basic operations required for pointers are: assigning a value (to make it point to a specific data item), dereferencing it (to access the data element - in read or write mode), pointer arithmetic (in particular, incrementing or decrementing the pointer with the **++** or **--** unary operators), and comparison of the values. (in particular, for equality or inequality)

If we are given class definitions that provide the assignment operator, comparison operators, the unary ***** operator and the (also unary) **++** and **--** operators, we can indeed write code that use these types of objects as if they were pointers (provided that each operator's definition perform an operation that corresponds to its intuitive equivalent for pointers).

At this point, you have seen the basic idea of the Standard Template Library. Now let's see the specifics, and a few examples:

First of all, containers are implemented via template class definitions. This allows us to define a group of elements of the required type. For example, if we need an array in which the elements are simply integers (as in the example above), we would declare it as:

```
vector<int> values;
```

(yes, the name of the container to represent arrays is vector - it follows the mathematical idea that a vector is a group of values in a multi-dimensional space)

The iterator is container-specific. That is, the iterator's operations depend on what type of container we are using. For that reason, the iterator class definition is inside the container class; this has a good side effect, which is that the syntax to declare an iterator suggests that the definition type belongs in the context of the container definition. Thus, we would declare an iterator for the values object as follows:

```
vector<int>::iterator current;
```

With this, we are almost ready to translate the example shown above to find the highest value in an array, using the STL tools. I will first present the implementation, and then explain the details:

```
vector<int>::iterator current = values.begin();
int high = *current++;

while (current != values.end())
{
   if (*current > high)
   {
      high = *current;
   }
   current++;
}
```

Surprisingly similar to the original version using arrays! Let's explain the differences: First, instead of declaring a pointer to int and initialize it pointing to the first element of the array, we declare an **`iterator`** for a **`vector`** of integers (i.e., an element that will "point" to integers contained in a **`vector`** object), and we initialize it "pointing" to the first element in values. Given that values is now an object and not a standard array, we need to ask that

object to give us a "pointer" to the first element (more exactly, to give us an **iterator** that is "pointing" to the first element).

This is done with the member-function **begin()** , which is provided by all the containers. The value of the **iterator** pointing to one past the end of the array is also provided by the container object, via the member-function **end()** .

Other than these two details, the rest is identical.

At this point, I guess you would not be surprised if I tell you that the algorithm to find the highest value in a linked list of integers is coded as follows using the STL tools:

```
list<int>::iterator current = values.begin();
int high = *current++;

while (current != values.end())
{
   if (*current > high)
   {
      high = *current;
   }
   current++;
}
```

The very only difference is the fact that I changed the word **vector** to **list** . Surprising? Well, it shouldn't be.

In this case, there are several important differences with respect to the version previously presented using a struct to define our own linked list:

In the original example, we get the value of the current element using:

current->value

In this case, we directly dereference current to obtain the value. Remember that **current** is an object of type **list<int>::iterator** . The unary ***** operator knows what to do, and surely will internally do something similar to **current->value** . But of course, the great thing is that such weird dereferencing operation is totally transparent for us, and we always use directly the ***** dereferencing operator.

Also, in the original example, we would advance to the next element (i.e., make **current** point to the next element) of the list in the obvious way:

current = current->next

In this case, we simply use the **++** operator. Again, this operator for the **list** container's iterator, will internally know how to advance ("point") to the next element, but the operation is transparent for the programmer.

The last difference is that instead of comparing with **NULL** to identify the end of the list, we compare with the value given by the member-function **end()** , which "points" to one element past the end of the list. This is implemented like that to be consistent and to provide a regular, standard way of manipulating the containers.

At this point, you understand all the basics of the containers section of the Standard Template Library. Of course, there are lots of details associated to this. In particular, what are the containers provided by the STL? These are some of the most frequently used containers: (for a complete list, refer to a complete documentation of the STL)

vector	Array
list	Doubly-linked list
slist	Singly-linked list

queue	FIFO (first-in, first-out) structure
deque	Array-like structure, with efficient insertion and removal at both ends
set	Set of unique elements
stack	LIFO (last in, first out) structure

The definition of these containers and their iterators are provided in the **std** namespace, and require to **#include** the appropriate file, which is always the same name of the container. The examples below show how to use specific containers:

```
#include <vector>

std::vector<double> values;

#include <queue>
using namespace std;

queue<Order> back_orders;

#include <stack>
using namespace std;

stack<Application_descriptor> undo_list;
```

Of course, don't be misled by the fact that all the containers are manipulated in the same manner: this only makes them look equivalent in the code, but their functionality and performance are obviously different.

Linked lists provide excellent performance for insertions and removals of elements - unlike vectors. However, linked lists require an important overhead for the storage, since they require extra pointers per element, besides the fact that each element requires a single memory allocation. Also, linked lists don't provide direct (random) access to arbitrary elements (i.e., subscripting). This disallows, for instance, the possibility of doing binary search in a linked list. (which we could do on a vector). On the other hand, a vector, which efficiently uses storage space to place elements in consecutive locations, is extremely inefficient if we need to do insertions or removals of elements in the sequence.

And speaking of insertions and removals of elements, I should mention that the idea of manipulating containers in a regular manner goes beyond the concept of the iterator and its related operators.

In particular, most containers provide member-functions to insert or remove elements from the ends. The "front" operations refer to operations performed at the beginning (first element), and the "back" operations at the end (last element). The following member-functions are provided for most containers:

push_front	Inserts element before the first (not available for vector)
pop_front	Removes the first element (not available for vector)
push_back	Inserts element after the last
pop_back	Removes the last element

Also, most containers provide the following member-functions:

empty	Boolean indicating if the container is empty
size	Returns the number of elements
insert	Inserts an element at a particular position
erase	Removes an element at a particular position
clear	Removes all the elements
resize	Resizes the container

front	Returns a reference to the first element
back	Returns a reference to the last element

The **vector** and **deque** containers provide subscripting access to elements, and subscripting access with bounds-checking:

[]	Subscripting access without bounds checking
at	Subscripting access with bounds checking

(there are many other operations provided in the STL, but I will omit a detailed list, given that this article is only an introduction to the STL. Consult a reference book for more details).

Again, don't be misled by the fact that these functions are supported by most containers and produce the same results: the performance will be totally different depending on the specific type of container used: for instance, inserting one element in an array is extremely expensive in terms of execution time (we need to move all the other elements to open up space for the element to insert). With a linked list, however, it takes a constant, small amount of time, regardless the number of elements in the list.

I will now present two examples to show the use of the **vector** and **list** containers. These examples indirectly illustrate the generic use of the STL containers.

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
   vector<string> names;

   while (more_data())
   {
      string temp = get_more_data();
      names.push_back(temp);
   }
     // Maybe a little inefficient because of the insertions.
     // If we know in advance how many elements we will have,
     // we could do:

   names.resize (num_elements);
   for (int i = 0; i < num_elements; i++)
   {
      names.at(i) = get_more_data();
   }
     // We could also use names[i] instead of names.at(i),
     // given that we know for sure that there won't be a
     // subscript overflow in that for loop, therefore, we
     // don't need checked access to the elements of the array

   // Sort the elements -- assuming that we provided a function to
   // do so, say,
   // template <class T>
   // void sort (vector<T> &);

   sort(names);

   // Now print the values
```

```
   for (int i = 0; i < names.size(); i++)
   {
      cout << names[i] << endl;
   }

   // We could also do it using an iterator

   vector<string>::iterator i;
   for (i = names.begin(); i != names.end(); ++i)
   {
      cout << *i << endl;
   }

   return 0;
}
```

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

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

private:
   string name, ID;
   int mark;
};

int main()
{
    list<Student> students;

    // Read from data base

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

    // Now print the students that failed (mark < 60%) - of
    // course, the particular Student object should provide a
    // member-function (say, passed()) that will determine that

   list<Student>::iterator i;
   for (i = students.begin(); i != students.end(); ++i)
   {
      if (! i->passed())    // iterators also provide operator ->
      {
         cout << "The student " << *i << " failed." << endl;
          // provided that class Student provides the overloaded
          // stream insertion operator <<
```

```
      }
   }

   // Now remove the failed students (of course, this could have
   // been done in the previous loop)

   i = students.begin()
   while (i != students.end())
   {
      if (! i->passed())
      {
         i = students.erase (i);
      }
      else
      {
         ++i;
      }
   }

   // ...

   return 0;
}
```

In the removal operation, we must be careful to assign **i** with the value returned by the member-function erase (it returns an iterator to the element after to the one that was removed). If not, **i** will be "pointing" to an element that no longer exists (its memory space was just released), therefore, the result of the next **i ++** operation would be undefined.

A Little More on Iterators

Iterators have, generally speaking, the same advantages and power as the pointers. But of course, they also have the same risks and inconveniences. With a pointer, we can accidentally modify data that we are not supposed to.

With a conventional pointer, we can partially solve that problem by having a pointer point to a data type with the **const** modifier.

In the first example with arrays, we could code the following:

```
const int * current;
// ...
```

And then, if we accidentally try to assign something to ***current** , or perform any operation that will or might modify the data pointed to by **current** , the compiler will report an error and stop compiling.

The equivalent idea is provided in the STL with the const iterators. As we have a class **iterator** defined inside the container class definition, we also have a definition for a class **const_iterator** , which provides basically the same functionality as a regular **iterator** , except that modifying the data "pointed to" by the **const_iterator** is disallowed.

Thus, we could use **const_iterators** to implement the examples previously shown, as illustrated below:

```
list<int>::const_iterator current = values.begin();
int high = *current++;

while (current != values.end())
{
   if (*current > high)
   {
      high = *current;
   }
   ++current;
}
```

In this example, if you accidentally write something like

```
*current = high;
```

the compiler will report an error, given that modifying the "dereferenced" value of a **const_iterator** is disallowed.

How is this implemented? It is indeed simpler than we might think. We already agreed that the **iterator** class definition provides the unary ***** operator to provide the "dereferencing" operation (access the value pointed to).

In the case of the iterator for a linked list, we could figure that the ***** operator's implementation is something along these lines:

```
T & list<T>::iterator::operator* () const
{
   return current->data;
}
```

Where **T** is the type in the template definition, and **data** is, of course, an item of type **T** .

Notice that the function must return a reference to the data item, such that the return value of the function can be used either as an Lvalue or as an Rvalue.

The equivalent operation, for a **const_iterator** , would be implemented as:

```
const T & list<T>::const_iterator::operator* () const
{
   return current->data;
}
```

The only difference is that the function now returns a reference to a constant value, which will prevent client code from modifying the returned value.

There is obviously more: the **const_iterator** class provides a constructor that takes an iterator of the same container as parameter. The converse is obviously not true: a **const_iterator** can not be used to create a regular **iterator** (that would compromise the constness of the object pointed to by the **const_iterator**).

As far as the containers are concerned, the functions **begin()** and **end()** (which return **iterator** s) have multiple versions (overloaded member-functions) in which the overloading resolution is done based on the constness of the object that calls the member-function. In particular, the prototypes for the **begin()** and **end()** functions would be something along these lines: (the example is shown for a list container)

```
// ... inside the class list<T> definition:

list<T>::iterator begin();
list<T>::iterator end();

list<T>::const_iterator begin() const;
list<T>::const_iterator end() const;
```

Thus, if a container object with the const attribute is used to call the functions **`begin()`** or **`end()`**, the compiler will use the **`const`** versions, which return a **`const_iterator`** ; if a non-const object is used, the non-const functions will be chosen by the compiler, which return regular **`iterator`**s.

These are other types of iterators, but I will omit here a detailed description of the others. Consult a reference book to further investigate about iterators.

Next: Algorithms and Function Objects

Main Page Tutorials