CS310

“Advanced C++: Templates and Generic Programming”

Introduction to Standard Template Library (STL)*

Hüseyin Akcan
huseyin.akcan@ieu.edu.tr

The heart of the C++ standard library, the part that influenced its overall architecture, is the standard template library (STL).

The STL is a generic library that provides solutions to managing collections of data with modern and efficient algorithms. It allows programmers to benefit from innovations in the area of data structures and algorithms.

From the programmer's point of view, the STL provides a bunch of collection classes that meet different needs, together with several algorithms that operate on them. All components of the STL are templates, so they can be used for arbitrary element types.

But the STL does even more: It provides a framework for supplying other collection classes or algorithms for which existing collection classes and algorithms work. All in all, the STL gives C++ a new level of abstraction.
Standard Template Library Components

Containers

Iterators

Algorithms
Do the Math!!!

Assume you have \( N \) containers and \( M \) algorithms to implement on these containers. How much work do you have to do?

In object oriented programming: \( N \times M \)

In generic programming: \( N + M \)

Key idea is to separate the containers and algorithms, so that an algorithm (such as SORT) works on any container.
Briefly: STL Containers

- Containers are used to manage collections of objects of a certain kind.
- Every kind of container has its own advantages and disadvantages, so having different container types reflects different requirements for collections in programs.
- The containers may be implemented as arrays or as linked lists, or they may have a special key for every element.
- All containers are template based, so they work with any type!
- Good knowledge of Data Structures helps in selecting the appropriate container!
Briefly: STL Iterators

- Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

- The major advantage of iterators is that they offer a small but common interface for any arbitrary container type. For example, one operation of this interface lets the iterator step to the next element in the collection. This is done independently of the internal structure of the collection. Regardless of whether the collection is an array or a tree, it works. This is because every container class provides its own iterator type that simply "does the right thing" because it knows the internal structure of its container.

- The interface for iterators is almost the same as for ordinary pointers. To increment an iterator you call operator ++. To access the value of an iterator you use operator *. So, you might consider an iterator a kind of a smart pointer that translates the call "go to the next element" into whatever is appropriate.
Briefly: STL Algorithms

- Algorithms are used to process the elements of collections. For example, they can search, sort, modify, or simply use the elements for different purposes.

- Algorithms use iterators. Thus, an algorithm has to be written only once to work with arbitrary containers because the iterator interface for iterators is common for all container types.

- To give algorithms more flexibility you can supply certain auxiliary functions called by the algorithms. Thus, you can use a general algorithm to suit your needs even if that need is very special or complex. For example, you can provide your own search criterion or a special operation to combine elements.
Briefly: STL Principles

- The concept of the STL is based on a separation of data and operations. The data is managed by container classes, and the operations are defined by configurable algorithms. Iterators are the glue between these two components. They let any algorithm interact with any container.

- In a way, the STL concept contradicts the original idea of object-oriented programming: The STL separates data and algorithms rather than combining them. However, the reason for doing so is very important. In principle, you can combine every kind of container with every kind of algorithm, so the result is a very flexible but still rather small framework.
STL Containers

- Container classes, or containers for short, manage a collection of elements. To meet different needs, the STL provides different kinds of containers.

- There are two general kinds of containers:
  - **Sequence containers** are ordered collections in which every element has a certain position. This position depends on the time and place of the insertion, but it is independent of the value of the element.
    - Vector, deque, lists
  - **Associative containers** are sorted collections in which the actual position of an element depends on its value due to a certain sorting criterion.
    - Set/Multiset, Map/Multimap
Common Container Abilities

- All containers provide value rather than reference semantics. Containers copy elements internally when they are inserted rather than managing references to it. Thus, each element of an STL container must be able to be copied. If objects you want to store don't have a public copy constructor, or copying is not useful, the container elements must be pointers or pointer objects that refer to these objects.

- In general, all elements have an order. Thus, you can iterate one or many times over all elements in the same order. Each container type provides operations that return iterators to iterate over the elements. This is the key interface of the STL algorithms.

- In general, operations are not safe. The caller must ensure that the parameters of the operations meet the requirements. Violating these requirements (such as using an invalid index) results in undefined behavior. Usually the STL does not throw exceptions by itself.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ContType c</code></td>
<td>Creates an empty container without any element</td>
</tr>
<tr>
<td><code>ContType c1(c2)</code></td>
<td>Copies a container of the same type</td>
</tr>
<tr>
<td><code>ContType c(beg,end)</code></td>
<td>Creates a container and initializes it with copies of all elements of [beg,end]</td>
</tr>
<tr>
<td>c.~<code>ContType()</code></td>
<td>Deletes all elements and frees the memory</td>
</tr>
<tr>
<td>c.size()</td>
<td>Returns the actual number of elements</td>
</tr>
<tr>
<td>c.empty()</td>
<td>Returns whether the container is empty (equivalent to <code>size()==0</code>, but might be faster)</td>
</tr>
<tr>
<td>c.max_size()</td>
<td>Returns the maximum number of elements possible</td>
</tr>
<tr>
<td>c1 == c2</td>
<td>Returns whether <code>c1</code> is equal to <code>c2</code></td>
</tr>
<tr>
<td>c1 != c2</td>
<td>Returns whether <code>c1</code> is not equal to <code>c2</code> (equivalent to !(c1==c2))</td>
</tr>
<tr>
<td>c1 &lt; c2</td>
<td>Returns whether <code>c1</code> is less than <code>c2</code></td>
</tr>
<tr>
<td>c1 &gt; c2</td>
<td>Returns whether <code>c1</code> is greater than <code>c2</code> (equivalent to c2&lt;c1)</td>
</tr>
<tr>
<td>c1 &lt;= c2</td>
<td>Returns whether <code>c1</code> is less than or equal to <code>c2</code> (equivalent to !(c2&lt;c1))</td>
</tr>
<tr>
<td>c1 &gt;= c2</td>
<td>Returns whether <code>c1</code> is greater than or equal to <code>c2</code> (equivalent to !(c1&lt;c2))</td>
</tr>
<tr>
<td>c1 = c2</td>
<td>Assigns all elements of <code>c1</code> to <code>c2</code></td>
</tr>
<tr>
<td>c1.swap(c2)</td>
<td>Swaps the data of <code>c1</code> and <code>c2</code></td>
</tr>
<tr>
<td>swap(c1,c2)</td>
<td>Same (as global function)</td>
</tr>
<tr>
<td>c.begin()</td>
<td>Returns an iterator for the first element</td>
</tr>
<tr>
<td>c.end()</td>
<td>Returns an iterator for the position after the last element</td>
</tr>
<tr>
<td>c.rbegin()</td>
<td>Returns a reverse iterator for the first element of a reverse iteration</td>
</tr>
<tr>
<td>c.rend()</td>
<td>Returns a reverse iterator for the position after the last element of a reverse iteration</td>
</tr>
<tr>
<td>c.insert(pos,elem)</td>
<td>Inserts a copy of <code>elem</code> (return value and the meaning of <code>pos</code> differ)</td>
</tr>
<tr>
<td>c.erase(beg,end)</td>
<td>Removes all elements of the range <code>[beg,end)</code> (some containers return next element not removed)</td>
</tr>
<tr>
<td>c.clear()</td>
<td>Removes all elements (makes the container empty)</td>
</tr>
<tr>
<td>c.get_allocator()</td>
<td>Returns the memory model of the container</td>
</tr>
</tbody>
</table>
Vectors

- A vector manages its elements in a dynamic array. It enables random access, which means you can access each element directly with the corresponding index.

- Appending and removing elements at the end of the array is very fast. However, inserting an element in the middle or at the beginning of the array takes time because all the following elements have to be moved to make room for it while maintaining the order.

- To use a vector, you must include the header file `<vector>`

- There, the type is defined as a template class inside namespace std:

```cpp
namespace std {

    template <class T, class Allocator = allocator<T> >
    class vector;

}
```
Vectors

- To use vectors effectively and correctly you should understand how size and capacity cooperate in a vector.

- Vectors provide the usual size operations size(), empty(), and max_size(). An additional "size" operation is the capacity() function. capacity() returns the number of characters a vector could contain in its actual memory. If you exceed the capacity(), the vector has to reallocate its internal memory.

- The capacity of a vector is important for two reasons:
  - Reallocation invalidates all references, pointers, and iterators for elements of the vector.
  - Reallocation takes time.
Vectors

• Thus, if a program manages pointers, references, or iterators into a vector, or if speed is a goal, it is important to take the capacity into account.

• To avoid reallocation, you can use reserve() to ensure a certain capacity before you really need it. In this way, you can ensure that references remain valid as long as the capacity is not exceeded.

```cpp
std::vector<int> v;  // create an empty vector
v.reserve (80);
```

• Another way to avoid reallocation is to initialize a vector with enough elements by passing additional arguments to the constructor.

```cpp
std::vector<T> v(5);  // creates a vector and initializes it with five
// values (calls five times the default constructor
// of type T)
```
Vectors

- Optimal performance regarding speed and memory usage is implementation defined.
- Thus, implementations might increase capacity in larger steps.
- In fact, to avoid internal fragmentation, many implementations allocate a whole block of memory (such as 2K) the first time you insert anything if you don't call reserve() first yourself. This can waste memory if you have many vectors with only a few small elements.
Vectors

- You can create vectors with and without elements for initialization. If you pass only the size, the elements are created with their default constructor. Note that an explicit call of the default constructor also initializes fundamental types such as int with zero

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vector&lt;Elem&gt; c</code></td>
<td>Creates an empty vector without any elements</td>
</tr>
<tr>
<td><code>vector&lt;Elem&gt; c1(c2)</code></td>
<td>Creates a copy of another vector of the same type (all elements are copied)</td>
</tr>
<tr>
<td><code>vector&lt;Elem&gt; c(n)</code></td>
<td>Creates a vector with n elements that are created by the default constructor</td>
</tr>
<tr>
<td><code>vector&lt;Elem&gt; c(n,elem)</code></td>
<td>Creates a vector initialized with n copies of element elem</td>
</tr>
<tr>
<td><code>vector&lt;Elem&gt; c(beg,end)</code></td>
<td>Creates a vector initialized with the elements of the range <code>[beg,end)</code></td>
</tr>
<tr>
<td><code>c~vector&lt;Elem&gt;()</code></td>
<td>Destroys all elements and frees the memory</td>
</tr>
</tbody>
</table>
Vectors

- See http://www.sgi.com/tech/stl/Vector.html for more of the interface and examples
Create an empty vector and keep adding elements into it.
Display the capacity and the size while doing the insertions and observe how the capacity changes!
**Deque**

- A deque (pronounced "deck") is very similar to a vector. It manages its elements with a dynamic array, provides random access, and has almost the same interface as a vector.

- The difference is that with a deque the dynamic array is open at both ends. Thus, a deque is fast for insertions and deletions at both the end and the beginning.

- To provide this ability, the deque is implemented typically as a bunch of individual blocks, with the first block growing in one direction and the last block growing in the opposite direction.
Abilities of Deques

Deques have the following differences compared with the abilities of vectors:

• Inserting and removing elements is fast both at the beginning and at the end (for vectors it is only fast at the end). These operations are done in amortized constant time.

• The internal structure has one more indirection to access the elements, so element access and iterator movement of deques are usually a bit slower.

• Iterators must be smart pointers of a special type rather than ordinary pointers because they must jump between different blocks.

• In systems that have size limitations for blocks of memory (for example, some PC systems), a deque might contain more elements because it uses more than one block of memory. Thus, max_size() might be larger for deques.
Abilities of Deques cont.

- Deques provide no support to control the capacity and the moment of reallocation. In particular, any insertion or deletion of elements other than at the beginning or end invalidates all pointers, references, and iterators that refer to elements of the deque. However, reallocation may perform better than for vectors, because according to their typical internal structure, deques don't have to copy all elements on reallocation.

- Blocks of memory might get freed when they are no longer used, so the memory size of a deque might shrink (however, whether and how this happens is implementation specific).
Similarities with Vector

The following features of vectors also apply to deques:

- Inserting and deleting elements in the middle is relatively slow because all elements up to either of both ends may be moved to make room or to fill a gap.
- Iterators are random access iterators.
When to use a Deque

In summary, you should prefer a deque if the following is true:

- You insert and remove elements at both ends (this is the classic case for a queue).
- You don't refer to elements of the container.
- It is important that the container frees memory when it is no longer used (however, the standard does not guarantee that this happens).
Deque

- The interface of vectors and deques is almost the same, so trying both is very easy when no special feature of a vector or a deque is necessary.

- Deque operations differ from vector operations only as follows:
  - Deques do not provide the functions for capacity (capacity() and reserve()).
  - Deques do provide direct functions to insert and to delete the first element (push_front() and pop_front()).

- No member functions for element access (except at()) check whether an index or an iterator is valid.

- An insertion or deletion of elements might cause a reallocation. Thus, any insertion or deletion invalidates all pointers, references, and iterators that refer to other elements of the deque. The exception is when elements are inserted at the front or the back. In this case, references and pointers to elements stay valid (but iterators don't).
Deque

- See http://www.sgi.com/tech/stl/Deque.html for more of the interface and examples
Lists

- A list manages its elements as a doubly linked list
- To use a list you must include the header file `<list>`
- There, the type is defined as a template class inside namespace std:
  
  ```cpp
  template <class T,
           class Allocator = allocator<T> >
  
  class Allocator = allocator<T> >
  
  class list;
  ```
Abilities of lists

- The internal structure of a list is totally different from a vector or a deque. Thus, a list differs in several major ways compared with vectors and deques:

- A list does not provide random access. For example, to access the fifth element, you must navigate the first four elements following the chain of links. Thus, accessing an arbitrary element using a list is slow.

- Inserting and removing elements is fast at each position, and not only at one or both ends. You can always insert and delete an element in constant time because no other elements have to be moved. Internally, only some pointer values are manipulated.

- Inserting and deleting elements does not invalidate pointers, references, and iterators to other elements.

- A list supports exception handling in such a way that almost every operation succeeds or is a no-op. Thus, you can't get into an intermediate state in which only half of the operation is complete.
Differences in member functions compared to vector and deque

- Lists provide neither a subscript operator nor at() because no random access is provided.

- Lists don't provide operations for capacity or reallocation because neither is needed. Each element has its own memory that stays valid until the element is deleted.

- Lists provide many special member functions for moving elements. These member functions are faster versions of general algorithms that have the same names. They are faster because they only redirect pointers rather than copy and move the values.
List operations

- The ability to create, copy, and destroy lists is the same as it is for every sequence container.
- Lists provide the usual operations for size and comparisons.
- Lists also provide the usual assignment operations for sequence containers.
- Because a list does not have random access, it provides only front() and back() for accessing elements directly.
- As usual, these operations do not check whether the container is empty. If the container is empty, calling them results in undefined behavior. Thus, the caller must ensure that the container contains at least one element.
- To access all elements of a list, you must use iterators. Lists provide the usual iterator functions. However, because a list has no random access, these iterators are only bidirectional. Thus, you can't call algorithms that require random access iterators. All algorithms that manipulate the order of elements a lot (especially sorting algorithms) fall under this category. However, for sorting the elements, lists provide the special member function sort().
Inserting and removing elements

- Lists provide all functions of deques, supplemented by special implementations of the remove() and remove_if() algorithms.

- As usual by using the STL, you must ensure that the arguments are valid. Iterators must refer to valid positions, the beginning of a range must have a position that is not behind the end, and you must not try to remove an element from an empty container.

- Inserting and removing happens faster if, when working with multiple elements, you use a single call for all elements rather than multiple calls.

- For removing elements, lists provide special implementations of the remove() algorithms

- These member functions are faster than the remove() algorithms because they manipulate only internal pointers rather than the elements.
Splice functions

- Linked lists have the advantage that you can remove and insert elements at any position in constant time. If you move elements from one container to another, this advantage doubles in that you only need to redirect some internal pointers.

- To support this ability, lists provide not only remove() but also additional modifying member functions to change the order of and relink elements and ranges. You can call these operations to move elements inside a single list or between two lists, provided the lists have the same type.
## Splice functions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.unique()</td>
<td>Removes duplicates of consecutive elements with the same value</td>
</tr>
<tr>
<td>c.unique(op)</td>
<td>Removes duplicates of consecutive elements, for which <code>op()</code> yields true</td>
</tr>
<tr>
<td>c1.splice(pos, c2)</td>
<td>Moves all elements of <code>c2</code> to <code>c1</code> in front of the iterator position <code>pos</code></td>
</tr>
<tr>
<td>c1.splice(pos, c2, c2pos)</td>
<td>Moves the element at <code>c2pos</code> in <code>c2</code> in front of <code>pos</code> of list <code>c1</code> (<code>c1</code> and <code>c2</code> may be identical)</td>
</tr>
<tr>
<td>c1.splice(pos, c2, c2beg, c2end)</td>
<td>Moves all elements of the range <code>[c2beg, c2end)</code> in <code>c2</code> in front of <code>pos</code> of list <code>c1</code> (<code>c1</code> and <code>c2</code> may be identical)</td>
</tr>
<tr>
<td>c.sort()</td>
<td>Sorts all elements with operator <code>&lt;</code></td>
</tr>
<tr>
<td>c.sort(op)</td>
<td>Sorts all elements with <code>op()</code></td>
</tr>
<tr>
<td>c1.merge(c2)</td>
<td>Assuming both containers contain the elements sorted, moves all elements of <code>c2</code> into <code>c1</code> so that all elements are merged and still sorted</td>
</tr>
<tr>
<td>c1.merge(c2, op)</td>
<td>Assuming both containers contain the elements sorted due to the sorting criterion <code>op()</code>, moves all elements of <code>c2</code> into <code>c1</code> so that all elements are merged and still sorted according to <code>op()</code></td>
</tr>
<tr>
<td>c.reverse()</td>
<td>Reverses the order of all elements</td>
</tr>
</tbody>
</table>
Lists

- See http://www.sgi.com/tech/stl/List.html for more of the interface and examples
C++ Code Jam

- Create a vector, a deque and a list
- Fill the containers by inserting elements from the front, middle and back
- Measure and report the time for each type of insertion
- Sort the containers and report the time