### Standard Template Library (STL):

We have been waiting so long to start this, but creating a base is as important as any other phase. So, today we’ll be starting the most awaited topic, the STL( Standard Template Library).

There is a reason why I’ve been saying that this topic is a must for all the competitive programmers out there,so let’s deal with that first.

**Why is this important for competitive programmers?**

1. Competitive programming is a part of various environments, be it job interviews, coding contests and all, and if you’re in one of those environments, you’ll be given limited time to code your program.
2. So, suppose you want in your program, a resizable array, or sort an array or any other data structure. or search for some element in your container.
3. You will always try to code a function which will execute the above mentioned things, and end up losing a great amount of time. But here is when you will use STL.

An STL is a library of generic functions and classes which saves you time and energy which you would have spent constructing for your use. This helps you reuse these well tested classes and functions umpteenth number of times according to your own convenience.

To put this simply, STL is used because it is not a good idea to reinvent something which is already built and can be used to innovate things further. Suppose you go to a company who builds cars, they will not ask you to start from scratch, but to start from where it is left. This is the basic idea behind using STL.

**COMPONENTS OF STL:**

We have three components in STL:

1. Containers
2. Algorithm
3. Iterators

Let’s deal with them individually;

**Containers:**

Container is an object which stores data. We have different containers having their own benefits. These are the implemented template classes for our use, which can be used just by including this library. You can even customise these template classes.

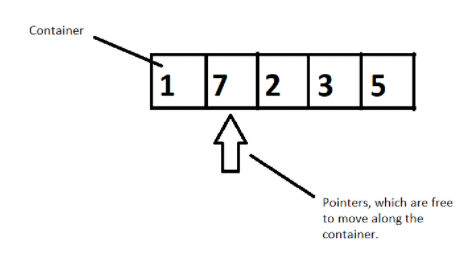
**Algorithms:**

Algorithms are a set of instructions which manipulates the input data to arrive at some desired result. In STL, we have already written algorithms, for example, to sort some data structure, or search some element in an array. These algorithms use template functions.

**Iterators:**

Iterators are objects which refer to an element in a container. And we handle them very much similarly to a pointer. Their basic job is to connect algorithms to the container and play a vital role in manipulation of the data.

I’ll give you a quick illustration of how they work combinedly.



**Figure 1: Illustration of how these three components work together**

Suppose we have a container of integers, and we want to sort them in ascending order. We will have pointers which will help moving elements to places by pointing to it, following a well-constructed algorithm. So, here a container gets sorted by following an algorithm by the use of pointers. This is how they work in accordance with each other.

So, this was the basics of STL and the motivation behind using it in your programs. I hope I was able to introduce it to you.

Thank you, for being with me throughout, hope you liked the tutorial. If you haven’t checked out the whole playlist yet, move on to [codewithharry.com](https://www.codewithharry.com/) or my YouTube channel to access it. I hope you enjoy them all. See you all in the next tutorial where we’ll dive deep in the containers and its different types. Till then keep coding.

### Containers in C++ STL:

**Containers**, objects which store data, **Algorithms,**set of procedures to process data, and **Iterators,**objects which point to some element in a container. Today, in this tutorial, we will be interested in discussing more about containers**.**

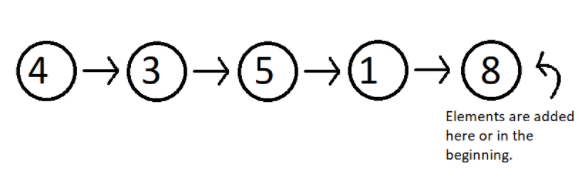
Containers are themselves of three types:

1. Sequence Containers
2. Associative Containers
3. Derived Containers

When we talked about containers, we said containers are objects which store data, but what are its three types all about? We’ll discuss that too.

* **Sequence Containers:**

A **sequence container**stores that data in a linear fashion. Refer to the illustration below to understand what storing something in a linear fashion means.

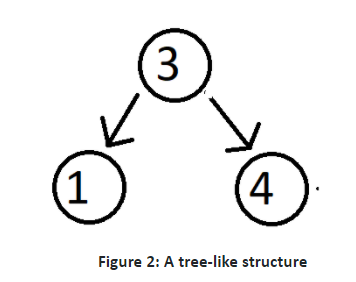


**Figure 1: Elements stored in a linear fashion**

Sequence containers include **Vector, List, Dequeue etc.**These are some of the most used sequence containers.

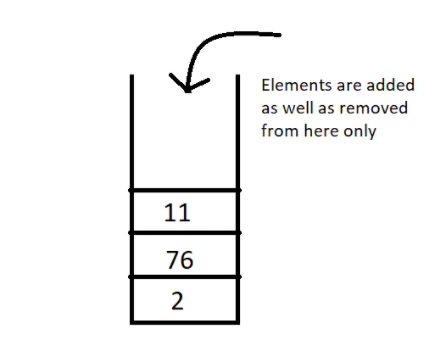
* **Associative Containers:**

An **associative container**is designed in such a way that enhances the accessing of some element in that container. It is very much used when the user wants to fastly reach some element. Some of these containers are, **Set, Multiset, Map, Multimap etc.**They store their data in a tree-like structure.



* **Derived Containers :**

As the name suggests, these containers are derived from either the sequence or the associative containers. They often provide you with some better methods to deal with your data. They deal with real life modelling. Some examples of derived containers are **Stack, Queue, Priority Queue, etc.**The following illustration give you the idea of how a stack works.



**Figure 3: A stack, works on the first in first out [FIFO] method**

Now since we have got the basic idea of all the three types of containers, a question which might arise is **when to use which**.  So, let’s deal with that,

In sequence containers, we have **Vectors**, which has following properties:

1. Faster random access to elements in comparison to array
2. Slower insertion and deletion at some random position, except at the end.
3. Faster insertion at the end.

In **Lists,**we have,

1. Random accessing elements is too slow, because every element is traversed using pointers.
2. Insertion and deletion at any position is relatively faster, because they only use pointers, which can easily be manipulated.

In associative containers, every operation except random access is faster in comparison to any other containers, be it inserting or deleting any element.

In associative containers, we cannot specifically tell which operation is faster or slower, we’ll have to inspect every data structure separately, and to get a clearer picture of all of these, you can access my Data Structure course : [Data Structures and Algorithms Course in Hindi](https://youtube.com/playlist?list=PLu0W_9lII9ahIappRPN0MCAgtOu3lQjQi)

For now, I'd like to hold on to our topic STL, and get you a strong hold on this too. In the coming videos, we’ll deal with our vectors, list, dequeues, set, multiset, maps, stack and much more. Just bear with me.

Thank you, for being with me throughout, hope you liked the tutorial. If you haven’t checked out the whole playlist yet, move on to [codewithharry.com](https://www.codewithharry.com/) or my YouTube channel to access it. I hope you enjoy them all. See you all in the next tutorial where we’ll talk about Vectors in C++ STL in detail. Till then keep coding.

**Vector In STL:**

Vectors are sequence containers representing arrays that can change in size.  
  
Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.  
  
Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.  
  
Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual [capacity](https://www.cplusplus.com/vector::capacity) greater than the storage strictly needed to contain its elements (i.e., its [size](https://www.cplusplus.com/vector::size)). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of [size](https://www.cplusplus.com/vector::size) so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see [push\_back](https://www.cplusplus.com/vector::push_back)).  
  
Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.  
  
Compared to the other dynamic sequence containers ([deques](https://www.cplusplus.com/deque), [lists](https://www.cplusplus.com/list) and [forward\_lists](https://www.cplusplus.com/forward_list)), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its [end](https://www.cplusplus.com/vector::end). For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than [lists](https://www.cplusplus.com/list) and [forward\_lists](https://www.cplusplus.com/forward_list).

### Member functions

[**(constructor)**](https://www.cplusplus.com/reference/vector/vector/vector/)

Construct vector (public member function )

[**(destructor)**](https://www.cplusplus.com/reference/vector/vector/~vector/)

Vector destructor (public member function )

[**operator=**](https://www.cplusplus.com/reference/vector/vector/operator=/)

Assign content (public member function )

**Iterators**:

[**begin**](https://www.cplusplus.com/reference/vector/vector/begin/)

Return iterator to beginning (public member function )

[**end**](https://www.cplusplus.com/reference/vector/vector/end/)

Return iterator to end (public member function )

[**rbegin**](https://www.cplusplus.com/reference/vector/vector/rbegin/)

Return reverse iterator to reverse beginning (public member function )

[**rend**](https://www.cplusplus.com/reference/vector/vector/rend/)

Return reverse iterator to reverse end (public member function )

[**cbegin**](https://www.cplusplus.com/reference/vector/vector/cbegin/)

Return const\_iterator to beginning (public member function )

[**cend**](https://www.cplusplus.com/reference/vector/vector/cend/)

Return const\_iterator to end (public member function )

[**crbegin**](https://www.cplusplus.com/reference/vector/vector/crbegin/)

Return const\_reverse\_iterator to reverse beginning (public member function )

[**crend**](https://www.cplusplus.com/reference/vector/vector/crend/)

Return const\_reverse\_iterator to reverse end (public member function )

**Capacity**:

[**size**](https://www.cplusplus.com/reference/vector/vector/size/)

Return size (public member function )

[**max\_size**](https://www.cplusplus.com/reference/vector/vector/max_size/)

Return maximum size (public member function )

[**resize**](https://www.cplusplus.com/reference/vector/vector/resize/)

Change size (public member function )

[**capacity**](https://www.cplusplus.com/reference/vector/vector/capacity/)

Return size of allocated storage capacity (public member function )

[**empty**](https://www.cplusplus.com/reference/vector/vector/empty/)

Test whether vector is empty (public member function )

[**reserve**](https://www.cplusplus.com/reference/vector/vector/reserve/)

Request a change in capacity (public member function )

[**shrink\_to\_fit**](https://www.cplusplus.com/reference/vector/vector/shrink_to_fit/)

Shrink to fit (public member function )

**Element access**:

[**operator[]**](https://www.cplusplus.com/reference/vector/vector/operator%5b%5d/)

Access element (public member function )

[**at**](https://www.cplusplus.com/reference/vector/vector/at/)

Access element (public member function )

[**front**](https://www.cplusplus.com/reference/vector/vector/front/)

Access first element (public member function )

[**back**](https://www.cplusplus.com/reference/vector/vector/back/)

Access last element (public member function )

[**data**](https://www.cplusplus.com/reference/vector/vector/data/)

Access data (public member function )

**Modifiers**:

[**assign**](https://www.cplusplus.com/reference/vector/vector/assign/)

Assign vector content (public member function )

[**push\_back**](https://www.cplusplus.com/reference/vector/vector/push_back/)

Add element at the end (public member function )

[**pop\_back**](https://www.cplusplus.com/reference/vector/vector/pop_back/)

Delete last element (public member function )

[**insert**](https://www.cplusplus.com/reference/vector/vector/insert/)

Insert elements (public member function )

[**erase**](https://www.cplusplus.com/reference/vector/vector/erase/)

Erase elements (public member function )

[**swap**](https://www.cplusplus.com/reference/vector/vector/swap/)

Swap content (public member function )

[**clear**](https://www.cplusplus.com/reference/vector/vector/clear/)

Clear content (public member function )

[**emplace**](https://www.cplusplus.com/reference/vector/vector/emplace/)

Construct and insert element (public member function )

[**emplace\_back**](https://www.cplusplus.com/reference/vector/vector/emplace_back/)

Construct and insert element at the end (public member function )

**Allocator**:

[**get\_allocator**](https://www.cplusplus.com/reference/vector/vector/get_allocator/)

Get allocator (public member function )

S