The basis of comparison	Vector	Array
Creation	Sequential container to store elements	An original data structure, based on index concept
Memory	Occupy more memory than Array	Memory-efficient
Length	Length varies	Fixed-size length
Usage	Frequent insertion and deletion	Frequent element access
Resize	Resize Vector is dynamic in nature	Resizing arrays is expensive
Structure	Template class, C++ only construct	Contiguous memory location
Indexing	Non- index based structure	Index-based with the lowest address as first, and highest address as last
Access	Access element is time-consuming although based on the position of an element.	Access element is a constant time operation irrespective of element location.

Vector	List
It has contiguous memory.	While it has non-contiguous memory.
It is synchronized.	While it is not synchronized.
Vector may have a default size.	List does not have default size.
In vector, each element only requires the space for itself only.	In list, each element requires extra space for the node which holds the element, including pointers to the next and previous elements in the list.
Insertion at the end requires constant time but insertion elsewhere is costly.	Insertion is cheap no matter where in the list it occurs.
Vector is thread safe.	List is not thread safe.
Deletion at the end of the vector needs constant time but for the rest it is O(n).	Deletion is cheap no matter where in the list it occurs.
Random access of elements is possible.	Random access of elements is not possible.
Iterators become invalid if elements are added to or removed from the vector.	Iterators are valid if elements are added to or removed from the list.

- Lists -

List contains elements which have the address of a previous and next element stored in them. This means that you can INSERT or DELETE an element anywhere in the list with constant speed O(1) regardless of the list size. You also splice (insert another list) into the existing list anywhere with constant speed as well. The reason is that list only needs to change two pointers (the previous and next) for the element we are inserting into the list.

Lists are not good if you need random access. So if one plans to access nth element in the list - one has to traverse the list one by one - O(n) speed

VS.

- Vectors -

Vector contains elements in sequence, just like an array. This is very convenient for random access. Accessing the "nth" element in a vector is a simple pointer calculation (O(1) speed). Adding elements to a vector is, however, different. If one wants to add an element in the middle of a vector - all the elements that come after that element will have to be re allocated down to make room for the new entry. The speed will depend on the vector size and on the position of the new element. The worst case scenario is inserting an element at position 2 in a vector, the best one is appending a new element. Therefore - insert works with speed O(n), where "n" is the number of elements that need to be moved - not necessarily the size of a vector.

1. **array** - insert/delete takes linear time due to shifting; update takes constant time; no space is used for pointers; accessing an item using [] is faster.
2. stl **vector** - insert/delete takes linear time due to shifting; update takes constant time; no space is used for pointers; accessing an item is slower than an array since it is a call to operator[] and a linked list .
3. stl **list** - insert and delete takes linear time since you need to iterate to a specific position before applying the insert/delete; additional space is needed for pointers; accessing an item is slower than an array since it is a linked list linear traversal.