When you have a large collection of similar objects, you usually store them. You store your groceries in the fridge, your clothes in the closet, and your valuables in safes. The same concept applies to programming. In C++, when you have multiple variables or objects, you store them in what we call, **Containers**.

In C++, there are 3 main types of containers:

1. Sequence Containers
2. Associative Containers
3. Unordered Associative Containers

This blogpost will conduct a somewhat deep-dive on **Sequence Containers** and their applications in the C++ programming language.

What are Sequence Containers?

According to cppreference.com, "Sequence containers implement data structures which can be accessed sequentially.".

Sequence Containers, hence the name, have an emphasis on the fact that objects within this data structure can be accessed and stored in a sequential manner. This means that objects are stored one after another, and that when accessing these objects that you stored, you can do that in the same way in which you stored them.

Let's see how this looks in code, specifically with **vectors.**

```
std::vector<int> myVector;

myVector.push_back(1); // [1]
myVector.push_back(2); // [1, 2]
myVector.push_back(3); // [1, 2, 3]
myVector.push_back(4); // [1, 2, 3, 4]
```

Here, we create a vector called, myVector and store 4 elements **sequentially** in this vector.

★ The vector starts out as empty, []

★ When we .push_back **1** upon the vector, the vector is now [1]

★ When we .push_back **2** upon the vector, the vector is now [1, 2]

★ When we .push_back **3** upon the vector, the vector is now [1, 2, 3]

★ When we .push_back **4** upon the vector, the vector is now [1, 2, 3, 4]

This is what is meant by sequential storage. We can also access these elements sequentially.

```
std::cout << myVector[2];
Output: 3
```

By putting myVector[2], we sequentially access the third element in the vector. The reason that we also put [2] and not [3] for the third element is because C++ works on a zero-based index, meaning that indexing starts at 0.

What are the different types of Sequence Containers?

We have the following:

1. Std::vector
2. Std::deque
3. Std::list
4. Std::forward_list
5. Std::array

These are the 5 main sequence containers in C++. For this blog post, I want to conduct a deep dive into what exactly a **std::deque** and a **std::forward_list** are. In addition, I would like to compare and contrast both.

std::deque

std::deque, short for double-ended queue, is a container that allows for insertion and deletion at the beginning and the end of the queue. Usually, deques are a fixed size, so if you are trying to push elements onto a deque past its size then you'll have to create a new deque.

A deque performs as follows:

- Insertion & Deletion (at either end): O(1)
- Insertion & Deletion O(n)

Deques are a great data structure to use if you need really speedy insertion & deletions due to the nature of a queue. However, they can tend to use a lot of memory due the design of the deque.

std::forward_list

std::forward_list is a container that uses singly linked lists, but, compared to std::list forward_lists are more space efficent.

A forward list performs as follows:

- Insertion & Deletion (at the front): O(1)
- Insertion & Deletion: O(n)

Forward lists are very memory efficient, so the best use case for them is when you are most concerned about memory. Be careful, as forward lists will not be a good data structure to use if you need to directly access memory or do a lot of insertion/deletion.

Comparing & contrasting std::deque and std::forward_list

A forward list is much more memory efficient than a deque because deques manage multiple arrays. But, a deque can do much more than a forward list. It has the ability to do bidirectional traversal and random access unlike forward lists.

Both data structures do decent at insertion and deletion. While deques are really good at working from both ends, forward lists (hence the word **forward** at the beginning) are more efficient with insertion & deletion at the front.

To summarize, I would recommend using deques if you aren't worried about memory, want something similar to a vector. I would also recommend to use forward_lists if you are super concerned about space and don't need a lot of functionality with the data structure.