Vectors

Introduction: Vectors are a versatile and widely used data structure in the C++ Standard Template Library (STL). They provide a dynamic array-like container that can grow or shrink as needed, making them highly efficient and flexible. In this guide, we will explore the features, operations, and performance aspects of vectors, empowering you to confidently utilize them in your C++ code.

Definition and Syntax:

- Vectors are part of the <vector> header in the C++ STL and can store elements of the same data type.
- To use vectors, include the <vector> header file and declare a vector object with the desired data type. For example, **std::vector<int> numbers;** creates an empty vector of integers.

Dynamic Size and Memory Management:

- Vectors can dynamically grow or shrink as elements are added or removed, thanks to their automatic memory management.
- The **push_back()** function appends an element to the end of the vector, automatically resizing it if needed.
- Vectors handle memory allocation and deallocation internally. When a vector exceeds its current capacity, it reallocates memory to accommodate additional elements.

Accessing Elements:

- Elements in a vector can be accessed using the subscript operator **[]** or the **at()** member function.
- For example, **numbers[0]** or **numbers.at(0)** retrieves the first element of the vector.

Size and Capacity:

- The **size()** function returns the number of elements currently stored in the vector.
- The **capacity()** function returns the current allocated storage size. The capacity may be larger than the size to accommodate future growth.

Modifying Elements:

- Elements in a vector can be modified using the assignment operator (**=**) or by directly accessing them using the subscript operator **[]**.
- For example, **numbers[1] = 42;** assigns the value 42 to the second element of the vector.

Removing Elements:

- Vectors provide functions to remove elements, such as **pop_back()** (removes the last element) and **erase()** (removes a specific element or a range of elements).

Sorting:

- Vectors can be sorted using the **std::sort()** algorithm from the <algorithm> header.
- It allows you to sort the elements of a vector in ascending or descending order.

Performance:

- Vectors offer constant-time access to individual elements (**O(1)**).
- They efficiently handle element insertion and removal at the end of the vector (**O(1)** amortized).
- However, inserting or removing elements in the middle of the vector is relatively slow (**O(n)**, where n is the number of elements).

Iterators:

- Vectors support iterators, which are used to traverse the elements of the vector.
- Common iterator functions include **begin()** (returns an iterator to the first element) and **end()** (returns an iterator to one position past the last element).

Common Operations:

- Checking if the vector is empty: Use **empty()** function.
- Finding the first and last element: Use **front()** and **back()** functions.
- Resizing the vector: Use **resize()** function.
- Clearing all elements: Use **clear()** function.
- Swapping the contents of two vectors: Use **swap()** function.

Space and Time Complexities of the Functions used in Vector.

1. **empty()**: Time complexity is O(1). Space complexity is O(1).
2. **push_back()**: Time complexity is amortized O(1). Space complexity is O(1) (ignoring the potential reallocation of memory).
3. **operator[]**: Time complexity is O(1). Space complexity is O(1).
4. **size()**: Time complexity is O(1). Space complexity is O(1).
5. **capacity()**: Time complexity is O(1). Space complexity is O(1).
6. Range-based for loop: Time complexity is O(n), where n is the number of elements in the vector. Space complexity is O(1).
7. **insert()**: Time complexity is O(n), where n is the number of elements to be inserted, plus the number of elements shifted. Space complexity is O(1) (ignoring the potential reallocation of memory).
8. **front()**: Time complexity is O(1). Space complexity is O(1).
9. **back()**: Time complexity is O(1). Space complexity is O(1).
10. **sort()**: Time complexity is O(n log n), where n is the number of elements in the vector. Space complexity is O(log n) due to the recursive nature of the sorting algorithm.
11. **erase()**: Time complexity is O(n), where n is the number of elements to be erased, plus the number of elements shifted. Space complexity is O(1).
12. **pop_back()**: Time complexity is O(1). Space complexity is O(1).
13. **clear()**: Time complexity is O(n), where n is the number of elements in the vector. Space complexity is O(1).
14. **reserve()**: Time complexity is O(1). Space complexity is O(1).
15. **resize()**: Time complexity is O(n), where n is the new size of the vector. Space complexity is O(1) (ignoring the potential reallocation of memory).
16. **swap()**: Time complexity is O(1). Space complexity is O(1).
17. **shrink_to_fit():** The time complexity is linear to the size of the vector O(n), The space complexity of **shrink_to_fit()** is generally considered to be constant **O(1)**.