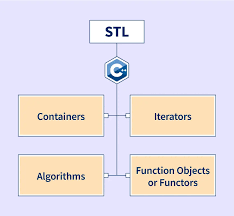
**Day 1: Introduction to STL and Containers**

**What is the Standard Template Library (STL)?**

The Standard Template Library (STL) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates. STL has four components:

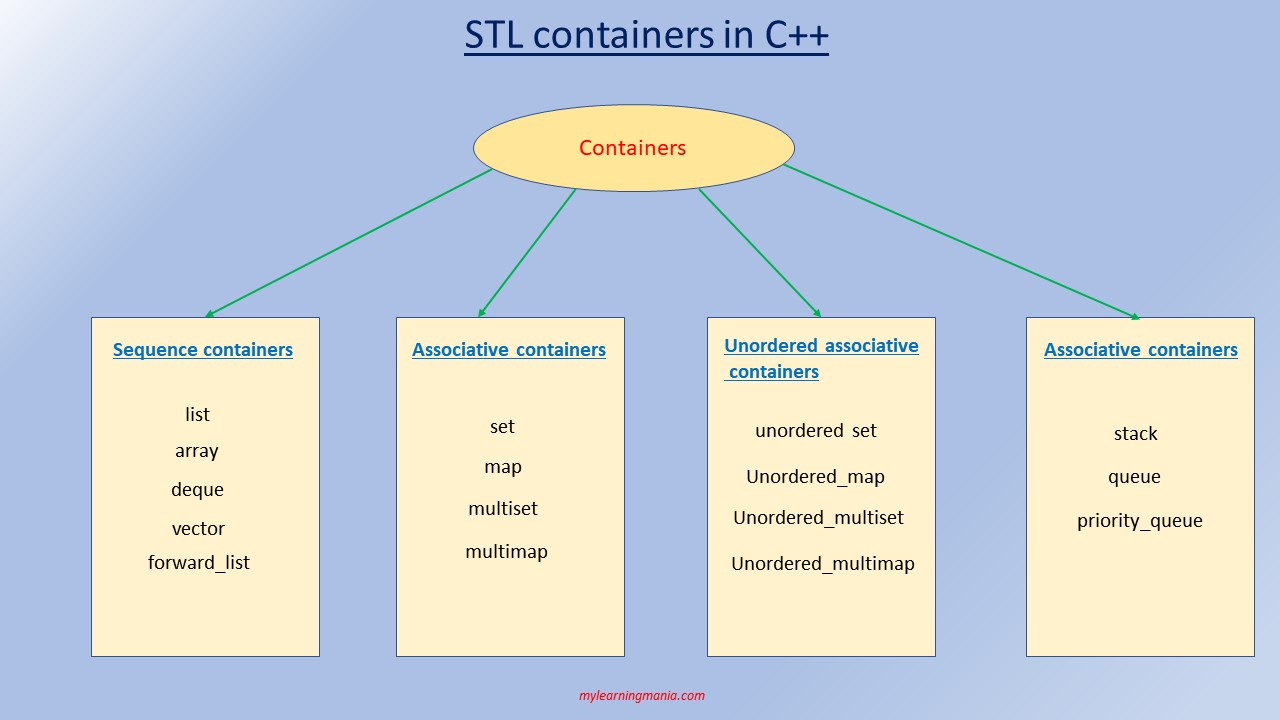
1. **Containers**: These are data structures like arrays, linked lists, stacks, queues, and others that store collections of objects.
2. **Iterators**: These are objects that point to elements within containers and allow you to traverse through them.
3. **Algorithms**: These are a set of functions for performing operations on the elements of containers, such as sorting, searching, counting, and manipulating data.
4. **Function Objects (Functors)**: These are objects that can be called as though they are functions, often used with algorithms.



**Overview of Containers**

Containers are the backbone of STL. **Containers** are data structures that hold a collection of elements. In STL, containers are generic, meaning they can store any data type (e.g., int, float, std::string, etc.). They store collections of data. Here's an overview of the most commonly used containers:

1. **Sequence Containers**: These store data in a linear sequence.
   * **std::vector**: A dynamic array that can grow in size. It's the most commonly used container because of its flexibility and ease of use.
   * **std::list**: A doubly linked list that allows for efficient insertion and deletion at both ends.
   * **std::deque**: A double-ended queue that allows for fast insertion and deletion at both ends.
   * **std::array**: A fixed-size array that is a wrapper around C-style arrays.
   * **std::forward\_list**: A singly linked list that is more memory-efficient than std::list.
2. **Associative Containers**: These store elements in sorted order and allow fast retrieval using keys.
   * **std::set**: Stores unique elements in sorted order.
   * **std::map**: Stores key-value pairs with unique keys in sorted order.
   * **std::multiset**: Similar to std::set, but allows duplicate elements.
   * **std::multimap**: Similar to std::map, but allows duplicate keys.
3. **Unordered Associative Containers**: These store elements in an unordered fashion, but retrieval is faster on average.
   * **std::unordered\_set**: Stores unique elements without any particular order.
   * **std::unordered\_map**: Stores key-value pairs with unique keys, with no particular order.
   * **std::unordered\_multiset**: Allows duplicate elements, with no particular order.
   * **std::unordered\_multimap**: Allows duplicate keys, with no particular order.
4. **Container Adapters**: These are wrappers around sequence containers that provide a different interface.
   * **std::stack**: Last-in, first-out (LIFO) data structure.
   * **std::queue**: First-in, first-out (FIFO) data structure.
   * **std::priority\_queue**: A queue where elements are ordered by priority.



**Key Points to Understand**

1. **Container Characteristics**:
   * **Efficiency**: Different containers have different strengths in terms of insertion, deletion, and access time.
   * **Flexibility**: Containers like std::vector are highly flexible with dynamic resizing, while containers like std::array have fixed sizes.
   * **Ordering**: Associative containers (std::set, std::map) maintain elements in a sorted order, while unordered containers (std::unordered\_map) do not.
2. **Choosing the Right Container**:
   * **Random Access**: Use std::vector or std::deque if you need fast random access.
   * **Frequent Insertion/Deletion**: Use std::list if you need frequent insertion or deletion in the middle of the sequence.
   * **Unique Elements**: Use std::set if you need to store unique elements in sorted order.
   * **Key-Value Pairs**: Use std::map if you need to associate keys with values.

**Day 2: Deep Dive into std::vector**

**Overview of std::vector**

std::vector is a dynamic array in C++ that provides a flexible way to manage collections of elements. Unlike a regular array, std::vector can change its size automatically when elements are added or removed. This makes it one of the most widely used containers in the STL.

**Key Features of std::vector**

* **Dynamic Size**: Automatically resizes when you add or remove elements.
* **Contiguous Memory**: Elements are stored in contiguous memory locations, which allows fast random access.
* **Efficient Access**: Provides constant time O(1) access to elements using an index.
* **Automatic Memory Management**: Manages memory allocation and deallocation internally, so you don’t have to worry about memory leaks.

**Important Methods of std::vector**

Here’s a list of commonly used methods provided by std::vector:

1. **push\_back(value)**: Adds a new element to the end of the vector.
2. **pop\_back()**: Removes the last element from the vector.
3. **size()**: Returns the number of elements in the vector.
4. **capacity()**: Returns the number of elements the vector can hold before it needs to allocate more memory.
5. **resize(new\_size)**: Changes the size of the vector. If the new size is smaller, elements at the end are removed; if larger, new elements are added with default values. numbers.resize(10, 200); // Resizes the vector to 10 elements, adding 200 to the new positions
6. **empty()**: Checks if the vector is empty.
7. **clear()**: Removes all elements from the vector.
8. **insert(iterator, value)**: Inserts an element at a specified position.
9. **erase(iterator)**: Removes an element at a specified position.
10. **at(index)**: Returns the element at the specified index, with bounds checking (throws an exception if out of range).
11. **operator[]**: Returns the element at the specified index without bounds checking.
12. **begin() and end()**: Returns iterators to the beginning and end of the vector, respectively
13. **rbegin() and rend()**: Returns iterators to the beginning and end of the vector, in orders order
14. **front() and back()**: Returns a reference to the first and last element in the vector.
15. **shrink\_to\_fit():** Request to shrink capacity to fit the size numbers.

#ifndef VECTOR\_H

#define VECTOR\_H

#include <stdexcept> // For std::out\_of\_range

class Vector **{**

private**:**

int**\*** arr**;** // Pointer to the array of integers

int currentSize**;** // Number of elements currently in the array

int currentCapacity**;** // Current capacity of the array

// Helper function to resize the array

void resize**()** **{**

int newCapacity **=** currentCapacity **\*** 2**;** // Double the capacity

int**\*** newArr **=** **new** int**[**newCapacity**];**

// Copy elements to the new array

**for** **(**int i **=** 0**;** i **<** currentSize**;** **++**i**)** **{**

newArr**[**i**]** **=** arr**[**i**];**

**}**

// Clean up old array

**delete[]** arr**;**

// Update pointers and capacity

arr **=** newArr**;**

currentCapacity **=** newCapacity**;**

**}**

public**:**

// Constructor

Vector**(**int initialCapacity **=** 10**)**

**:** currentSize**(**0**),** currentCapacity**(**initialCapacity**)** **{**

arr **=** **new** int**[**currentCapacity**];**

**}**

// Destructor

**~**Vector**()** **{**

**delete[]** arr**;**

**}**

// Add an element to the end of the vector

void push\_back**(**int value**)** **{**

**if** **(**currentSize **==** currentCapacity**)** **{**

resize**();** // Resize if needed

**}**

arr**[**currentSize**++]** **=** value**;**

**}**

// Remove the last element from the vector

void pop\_back**()** **{**

**if** **(**currentSize **>** 0**)** **{**

**--**currentSize**;**

**}**

**}**

// Get the number of elements

int size**()** const **{**

**return** currentSize**;**

**}**

// Get the capacity of the vector

int capacity**()** const **{**

**return** currentCapacity**;**

**}**

// Resize the vector

void resize**(**int newSize**,** int defaultValue **=** 0**)** **{**

**if** **(**newSize **>** currentCapacity**)** **{**

**while** **(**currentCapacity **<** newSize**)** **{**

resize**();** // Keep resizing until capacity is sufficient

**}**

**}**

**if** **(**newSize **>** currentSize**)** **{**

**for** **(**int i **=** currentSize**;** i **<** newSize**;** **++**i**)** **{**

arr**[**i**]** **=** defaultValue**;**

**}**

**}**

currentSize **=** newSize**;**

**}**

// Check if the vector is empty

bool empty**()** const **{**

**return** currentSize **==** 0**;**

**}**

// Remove all elements from the vector

void clear**()** **{**

currentSize **=** 0**;**

**}**

// Insert an element at a specific position

void insert**(**int index**,** int value**)** **{**

**if** **(**index **<** 0 **||** index **>** currentSize**)** **{**

**throw** std**::**out\_of\_range**(**"Index out of bounds"**);**

**}**

**if** **(**currentSize **==** currentCapacity**)** **{**

resize**();**

**}**

**for** **(**int i **=** currentSize**;** i **>** index**;** **--**i**)** **{**

arr**[**i**]** **=** arr**[**i **-** 1**];**

**}**

arr**[**index**]** **=** value**;**

**++**currentSize**;**

**}**

// Remove an element at a specific position

void erase**(**int index**)** **{**

**if** **(**index **<** 0 **||** index **>=** currentSize**)** **{**

**throw** std**::**out\_of\_range**(**"Index out of bounds"**);**

**}**

**for** **(**int i **=** index**;** i **<** currentSize **-** 1**;** **++**i**)** **{**

arr**[**i**]** **=** arr**[**i **+** 1**];**

**}**

**--**currentSize**;**

**}**

// Access element at a specific index with bounds checking

int at**(**int index**)** const **{**

**if** **(**index **<** 0 **||** index **>=** currentSize**)** **{**

**throw** std**::**out\_of\_range**(**"Index out of bounds"**);**

**}**

**return** arr**[**index**];**

**}**

// Access element at a specific index without bounds checking

int**&** **operator[](**int index**)** **{**

**return** arr**[**index**];**

**}**

// Get a pointer to the beginning of the vector

int**\*** begin**()** **{**

**return** arr**;**

**}**

// Get a pointer to the end of the vector

int**\*** end**()** **{**

**return** arr **+** currentSize**;**

**}**

// Get a pointer to the reverse beginning of the vector

int**\*** rbegin**()** **{**

**return** arr **+** currentSize **-** 1**;**

**}**

// Get a pointer to the reverse end of the vector

int**\*** rend**()** **{**

**return** arr **-** 1**;**

**}**

// Get a reference to the first element

int**&** front**()** **{**

**if** **(**empty**())** **throw** std**::**out\_of\_range**(**"Vector is empty"**);**

**return** arr**[**0**];**

**}**

// Get a reference to the last element

int**&** back**()** **{**

**if** **(**empty**())** **throw** std**::**out\_of\_range**(**"Vector is empty"**);**

**return** arr**[**currentSize **-** 1**];**

**}**

// Shrink capacity to fit size

void shrink\_to\_fit**()** **{**

**if** **(**currentSize **<** currentCapacity**)** **{**

int**\*** newArr **=** **new** int**[**currentSize**];**

**for** **(**int i **=** 0**;** i **<** currentSize**;** **++**i**)** **{**

newArr**[**i**]** **=** arr**[**i**];**

**}**

**delete[]** arr**;**

arr **=** newArr**;**

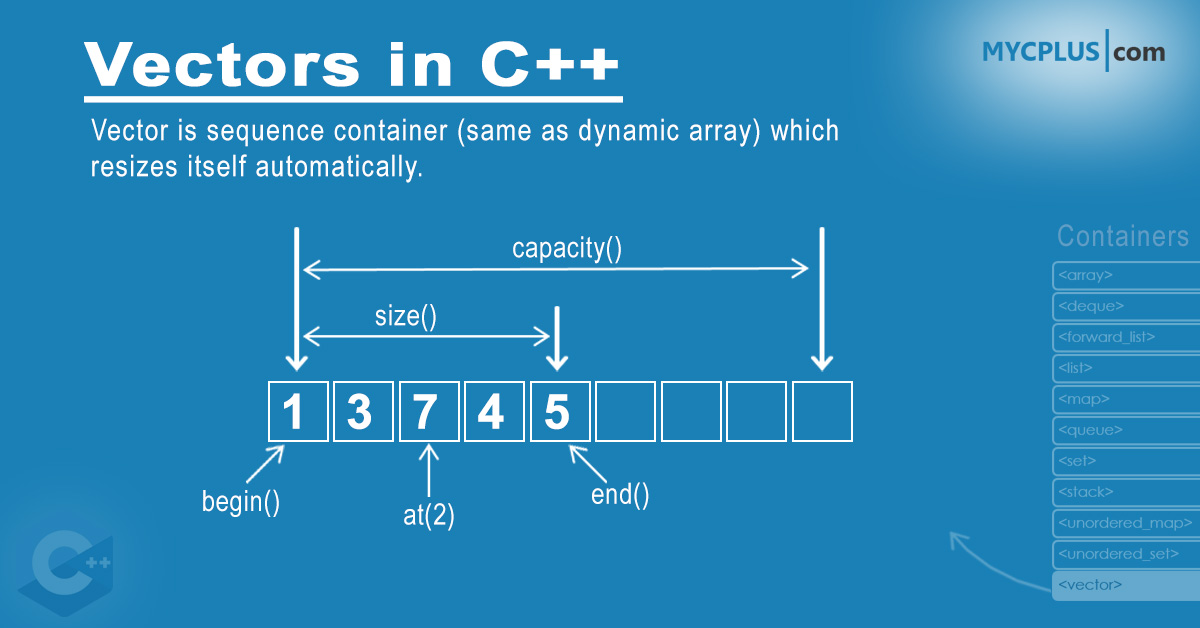
currentCapacity **=** currentSize**;**

**}**

**}**

**};**

#endif // VECTOR\_H



**Memory Management in std::vector**

* **Capacity vs. Size**: size() tells you how many elements are currently in the vector, while capacity() tells you how many elements it can hold before needing to allocate more memory.
* **Reallocation**: When you add elements that exceed the current capacity, the vector reallocates memory (usually doubling the capacity). This operation involves copying all elements to the new memory location, which can be expensive.

**Example Programs Using std::vector**

**Basic Operations**:

1. **#include <iostream>**
2. **#include <vector>**
3. **int main() {**
4. **std::vector<int> numbers;**
5. **// Adding elements**
6. **numbers.push\_back(10);**
7. **numbers.push\_back(20);**
8. **numbers.push\_back(30);**
9. **// Accessing elements**
10. **std::cout << "First element: " << numbers[0] << std::endl;**
11. **std::cout << "Second element: " << numbers.at(1) << std::endl;**
12. **// Removing the last element**
13. **numbers.pop\_back();**
14. **// Iterating through the vector**
15. **for (int i = 0; i < numbers.size(); ++i) {**
16. **std::cout << "Element " << i << ": " << numbers[i] << std::endl;**
17. **}**
18. **return 0;**
19. **}**

**Inserting and Erasing Elements**:

1. **#include <iostream>**
2. **#include <vector>**
3. **int main() {**
4. **std::vector<int> numbers = {1, 2, 3, 4, 5};**
5. **// Inserting an element at the second position**
6. **numbers.insert(numbers.begin() + 1, 10);**
7. **// Erasing the third element**
8. **numbers.erase(numbers.begin() + 2);**
9. **// Displaying the elements**
10. **for (auto it = numbers.begin(); it != numbers.end(); ++it) {**
11. **std::cout << \*it << " ";**
12. **}**
13. **std::cout << std::endl;**
14. **return 0;**
15. **}**

**Using resize and capacity**:

1. **#include <iostream>**
2. **#include <vector>**
3. **int main() {**
4. **std::vector<int> numbers(5, 100); // Create a vector with 5 elements, each initialized to 100**
5. **std::cout << "Initial size: " << numbers.size() << std::endl;**
6. **std::cout << "Initial capacity: " << numbers.capacity() << std::endl;**
7. **// Resizing the vector**
8. **numbers.resize(10, 200); // Resizes the vector to 10 elements, adding 200 to the new positions**
9. **std::cout << "Size after resizing: " << numbers.size() << std::endl;**
10. **std::cout << "Capacity after resizing: " << numbers.capacity() << std::endl;**
11. **// Displaying the elements**
12. **for (int num : numbers) {**
13. **std::cout << num << " ";**
14. **}**
15. **std::cout << std::endl;**
16. **return 0;**
17. **}**

**How auto Works**

When you declare a variable using auto, the compiler looks at the expression on the right-hand side of the assignment to determine the variable's type. The actual type is then substituted for auto by the compiler.

**Example Usage**

1. **#include <iostream>**
2. **#include <vector>**
3. **int main() {**
4. **std::vector<int> numbers = {10, 20, 30};**
5. **// Without auto**
6. **std::vector<int>::iterator it = numbers.begin();**
7. **// With auto**
8. **auto it\_auto = numbers.begin();**
9. **std::cout << "First element: " << \*it\_auto << std::endl; // Output: 10**
10. **return 0;**
11. **}**

**Iterators vs. Index-Based Loops**

* **Advantages of Using Iterators**:
  + **Generic**: Works with any container that supports iterators, not just std::vector.
  + **Safe**: Helps avoid out-of-bounds errors.
  + **Readability**: Provides a clear, concise way to iterate over elements.
* **Comparison with Index-Based Loops**:
  + **Index-Based**: Often used with arrays or vectors when you need random access to elements. E.g., for (int i = 0; i < numbers.size(); ++i) {}.
  + **Iterator-Based**: Preferred when working with containers like std::list, std::map, or even std::vector where you want to abstract away the underlying data structure.

**Further Considerations**

* **Const Iterators**: If you don’t need to modify the elements, consider using const\_iterator to ensure you don’t accidentally change the data.
  + Example: for (auto it = numbers.cbegin(); it != numbers.cend(); ++it) {}.
* **#include <iostream>**
* **#include <vector>**
* **int main() {**
* **std::vector<int> numbers = {10, 20, 30, 40, 50};**
* **// Using const\_iterator to ensure elements are not modified**
* **for (auto it = numbers.cbegin(); it != numbers.cend(); ++it) {**
* **std::cout << \*it << " "; // This will print the elements, but you can't modify them**
* **// \*it = 100; // This would cause a compilation error**
* **}**
* **return 0;**
* **}**
* **Reverse Iterators**: If you need to iterate backward, you can use reverse iterators with rbegin() and rend().

**Best Practices and Tips**

* **Reserve Capacity**: If you know the number of elements in advance, use reserve() to preallocate memory, reducing the number of reallocations.
* **Avoid Frequent Insertions/Deletions in the Middle**: These operations can be expensive in std::vector as they may require shifting elements.
* **Use at() for Safe Access**: While operator[] is faster, at() provides bounds checking and prevents out-of-range errors.

**Common Pitfalls**

* **Out-of-Bounds Access**: Using operator[] with an invalid index can lead to undefined behavior. Use at() if you’re unsure about the index validity.
* **Performance Impact of Reallocations**: Frequent insertions without reserving space can cause multiple memory reallocations, leading to performance hits.
* **Invalidating Iterators**: When the vector reallocates or when elements are inserted/erased, iterators may become invalid. Be cautious when modifying the vector while iterating.

**Understand Core Concepts and Methods of std::vector**

* **Overview**: std::vector is a dynamic array that can grow and shrink in size. It provides random access to elements and is one of the most commonly used STL containers.
* **Key Methods to Learn**:
  + **Construction and Initialization**:
    - std::vector<T> v; // Default constructor
    - std::vector<T> v(n); // Constructor with size
    - std::vector<T> v(n, value); // Constructor with size and initial value
    - std::vector<T> v(begin, end); // Constructor from another container’s range
* **#include <iostream>**
* **#include <vector>**
* **#include <list>**
* **int main() {**
* **// Initialize a list with some elements**
* **std::list<int> myList = {1, 2, 3, 4, 5};**
* **// Create a vector from the list using the range constructor**
* **std::vector<int> myVector(myList.begin(), myList.end());**
* **// Print the vector elements**
* **std::cout << "Vector elements: ";**
* **for (const auto& elem : myVector) {**
* **std::cout << elem << " ";**
* **}**
* **return 0;**
* **}**
  + **Element Access**:
    - v.at(index) // Safe access with bounds checking
    - v[index] // Direct access
    - v.front() // First element
    - v.back() // Last element
    - v.data() // Raw pointer to the underlying array
  + **Modifiers**:
    - v.push\_back(value) // Add element to the end
    - v.pop\_back() // Remove last element
    - v.insert(position, value) // Insert element at specified position
    - v.erase(position) // Remove element at specified position
    - v.resize(new\_size) // Change size of the vector
    - v.clear() // Remove all elements
  + **Capacity**:
    - v.size() // Number of elements
    - v.capacity() // Size of allocated storage
    - v.empty() // Check if the vector is empty
    - v.shrink\_to\_fit() // Reduce capacity to fit size
  + **Iterators**:
    - v.begin() // Iterator to the beginning
    - v.end() // Iterator to the end
    - v.rbegin() // Reverse iterator to the last element
    - v.rend() // Reverse iterator to the position before the first element

**2. Experiment with Various Operations**

* **Adding Elements**:
  + Use push\_back() to add elements.
  + Test adding elements with different data types.
* **Removing Elements**:
  + Use pop\_back() to remove the last element.
  + Use erase() and remove() for other removal operations.
* **Resizing**:
  + Use resize() to increase or decrease the size of the vector.
  + Observe how it affects the capacity and contents of the vector.
* **Accessing Elements**:
  + Practice accessing elements with operator[], at(), front(), and back().

**3. Write Practice Programs**

* **Basic Operations**:
  + Create a vector, add elements, and print them.
  + Remove elements and check the results.
* **Resize and Capacity**:
  + Initialize a vector with a specific size, resize it, and use shrink\_to\_fit() to observe the changes in capacity.
* **Iterators**:
  + Use iterators to traverse the vector and perform operations.
  + Practice with both forward and reverse iterators.

**4. Test Effects of Resizing, Capacity Management, and Iterator Usage**

* **Resize Testing**:
  + Test how resizing affects the vector's capacity and contents.
* **Capacity Management**:
  + Compare capacity before and after using shrink\_to\_fit().
* **Iterator Usage**:
  + Test iterator validity after operations like push\_back(), resize(), and erase().
  + Understand how operations affect iterator validity.

**5. Review Performance Considerations**

**Memory Management in std::vector**

Understanding how resizing affects memory allocation and deallocation in std::vector is crucial for managing performance and memory usage efficiently.

**1. Resizing and Memory Allocation**

* **Dynamic Array**: std::vector is implemented as a dynamic array that can grow or shrink in size. The memory for the elements is managed automatically by the vector.
* **Capacity vs. Size**:
  + **Size**: The number of elements currently in the vector.
  + **Capacity**: The number of elements that the vector can hold before it needs to allocate more memory.
* **Growth Strategy**:
  + When you add elements to a vector, if the size exceeds the capacity, the vector will allocate a new, larger block of memory, copy the existing elements to the new block, and then deallocate the old memory.
  + The capacity typically grows geometrically (e.g., doubling) to minimize the number of reallocations and to amortize the cost of resizing.
* **Deallocation**:
  + When the vector is resized smaller or cleared, the memory for excess elements is deallocated. The vector may also shrink its capacity to fit the new size if shrink\_to\_fit() is called.

**Iterator Invalidation**

Iterator invalidation refers to situations where iterators pointing to elements in a container become invalid due to modifications to the container.

**1. Operations that Invalidate Iterators**

* **push\_back()**:
  + May invalidate iterators if the vector needs to reallocate memory to accommodate new elements. After reallocation, all iterators (except those pointing to elements before the reallocation) become invalid.
* **resize()**:
  + If the vector is resized to a larger size, iterators to the original elements remain valid. However, if the size is reduced, iterators pointing to elements that are removed become invalid.
* **insert() and erase()**:
  + **insert()**: May invalidate iterators pointing to the position where elements are inserted or to the elements after the insertion point if reallocation occurs.
  + **erase()**: Invalidates iterators pointing to the element that is removed and potentially those to the elements after it.
* **clear()**:
  + Invalidates all iterators, as it removes all elements from the vector.

**Day 3: Deep Dive into std::list**

**Overview of std::list**

std::list is a doubly linked list in C++ that allows for efficient insertion and deletion of elements anywhere in the list. Unlike std::vector, which stores elements in contiguous memory, std::list stores elements as nodes, each linked to the next and previous nodes. This structure makes std::list an ideal choice when frequent insertions and deletions are needed at various positions in the list.

**Key Features of std::list**

* **Bidirectional**: Each element points to both the next and previous elements, allowing for forward and backward traversal.
* **Efficient Insertion/Deletion**: Adding or removing elements is done in constant time O(1) when working with iterators, regardless of position.
* **No Contiguous Memory**: Unlike arrays or vectors, elements are not stored in contiguous memory, making random access less efficient.
* **Dynamic Size**: Similar to std::vector, std::list can dynamically grow or shrink in size.

**Important Methods of std::list**

Here’s a list of commonly used methods provided by std::list:

1. **push\_back(value)**: Adds an element to the end of the list.
2. **push\_front(value)**: Adds an element to the front of the list.
3. **pop\_back()**: Removes the last element from the list.
4. **pop\_front()**: Removes the first element from the list.
5. **insert(iterator, value)**: Inserts an element at a specified position.
6. **erase(iterator)**: Removes the element at the specified position.
7. **size()**: Returns the number of elements in the list.
8. **clear()**: Removes all elements from the list.
9. **empty()**: Checks if the list is empty.
10. **front()**: Returns a reference to the first element.
11. **back()**: Returns a reference to the last element.
12. **begin() and end()**: Returns iterators to the beginning and end of the list, respectively.
13. **rbegin() and rend()**: Returns reverse iterators to the beginning and end of the list, respectively.

**Example Programs Using std::list**

**Basic Operations:**

#include <iostream>

#include <list>

int main() {

std::list<int> numbers;

// Adding elements to the front and back

numbers.push\_back(10);

numbers.push\_back(20);

numbers.push\_front(5);

// Accessing the front and back elements

std::cout << "Front: " << numbers.front() << std::endl;

std::cout << "Back: " << numbers.back() << std::endl;

// Removing elements from the front and back

numbers.pop\_back();

numbers.pop\_front();

// Printing remaining elements

for (int num : numbers) {

std::cout << num << " ";

}

std::cout << std::endl;

return 0;

}

**Inserting and Erasing Elements:**

#include <iostream>

#include <list>

int main() {

std::list<int> numbers = {1, 2, 3, 4, 5};

// Inserting an element at the second position

auto it = numbers.begin();

std::advance(it, 1); // Move the iterator to the second position

numbers.insert(it, 10);

// Erasing the third element

it = numbers.begin();

std::advance(it, 2);

numbers.erase(it);

// Displaying the elements

for (auto it = numbers.begin(); it != numbers.end(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

return 0;

}

**Using Reverse Iterators:**

#include <iostream>

#include <list>

int main() {

std::list<int> numbers = {1, 2, 3, 4, 5};

// Iterating in reverse order

for (auto it = numbers.rbegin(); it != numbers.rend(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

return 0;

}

**Memory Management in std::list**

* **No Capacity Function**: Unlike std::vector, std::list does not have a capacity() function because elements are not stored contiguously.
* **Dynamic Allocation**: Each node is dynamically allocated, which can be inefficient if memory is highly fragmented, but it allows for flexible insertion and deletion.

**Best Practices and Tips for std::list**

* **Efficient for Insertions and Deletions**: Use std::list when you need efficient insertions and deletions, especially at arbitrary positions.
* **Avoid Frequent Random Access**: Random access is O(n) in std::list, so prefer std::vector or std::deque for applications that require frequent random access.
* **Iterators are Stable**: Unlike std::vector, iterators remain valid after insertions or deletions, except for the iterators pointing to the erased elements.

**Common Pitfalls**

* **Performance for Random Access**: std::list does not provide constant-time random access. Operations like numbers[i] are inefficient compared to std::vector.
* **Memory Overhead**: Each element in a std::list requires additional memory for storing pointers to the previous and next elements.

**Day 2: Deep Dive into std::set**

**Overview of std::set**

std::set is an associative container in C++ that stores unique elements following a specific order. Unlike std::vector, std::set maintains its elements in a sorted order and automatically manages its size and memory.

**Key Features of std::set**

* **Sorted Order**: Elements are stored in a sorted order based on the specified comparison function.
* **Unique Elements**: Only one instance of each element is allowed.
* **Efficient Search**: Provides logarithmic time complexity (O(log n)) for search, insertion, and deletion operations.
* **Automatic Memory Management**: Handles memory allocation and deallocation internally.

**Important Methods of std::set**

1. **insert(value)**: Adds a new element to the set if it does not already exist.
2. **erase(value)**: Removes the specified element from the set.
3. **find(value)**: Searches for an element and returns an iterator to it if found.
4. **count(value)**: Returns the number of occurrences of an element (0 or 1 for std::set).
5. **size()**: Returns the number of elements in the set.
6. **empty()**: Checks if the set is empty.
7. **clear()**: Removes all elements from the set.
8. **lower\_bound(value)**: Returns an iterator to the first element that is not less than the given value.
9. **upper\_bound(value)**: Returns an iterator to the first element that is greater than the given value.
10. **equal\_range(value)**: Returns a pair of iterators representing the range of elements equivalent to the given value.
11. **begin() and end()**: Returns iterators to the beginning and end of the set, respectively.
12. **rbegin() and rend()**: Returns reverse iterators to the beginning and end of the set, respectively.
13. **size()**: Returns the number of elements currently in the set.

**Memory Management in std::set**

* **Memory Allocation**: std::set uses a self-balancing binary search tree (such as a Red-Black Tree) to store elements, which allows efficient insertion, deletion, and lookup operations.
* **Reallocation**: Unlike std::vector, std::set does not need to reallocate memory since it dynamically adjusts its structure based on element insertions and deletions.

**Example Programs Using std::set**

**Basic Operations:**

#include <iostream>

#include <set>

int main() {

std::set<int> numbers;

// Adding elements

numbers.insert(10);

numbers.insert(20);

numbers.insert(30);

// Accessing elements

std::cout << "Elements in set: ";

for (const auto& num : numbers) {

std::cout << num << " ";

}

std::cout << std::endl;

// Removing an element

numbers.erase(20);

// Displaying elements

std::cout << "Elements after erasing 20: ";

for (const auto& num : numbers) {

std::cout << num << " ";

}

std::cout << std::endl;

return 0;

}

**Inserting and Erasing Elements:**

cpp

Copy code

#include <iostream>

#include <set>

int main() {

std::set<int> numbers = {1, 2, 3, 4, 5};

// Inserting an element

numbers.insert(10);

// Erasing an element

numbers.erase(3);

// Displaying elements

std::cout << "Elements after insert and erase: ";

for (const auto& num : numbers) {

std::cout << num << " ";

}

std::cout << std::endl;

return 0;

}

**Using Iterators:**

cpp

Copy code

#include <iostream>

#include <set>

int main() {

std::set<int> numbers = {10, 20, 30, 40, 50};

// Using iterators to traverse the set

std::cout << "Elements using iterators: ";

for (auto it = numbers.begin(); it != numbers.end(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

// Using reverse iterators

std::cout << "Elements in reverse order: ";

for (auto it = numbers.rbegin(); it != numbers.rend(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

return 0;

}

**Understanding Core Concepts and Methods of std::set**

* **Overview**: std::set is an ordered associative container that stores unique elements.
* **Key Methods to Learn**:
  + **Construction and Initialization**:
    - std::set<T> s; // Default constructor
    - std::set<T> s(begin, end); // Constructor from another container’s range
  + **Element Access**:
    - s.find(value) // Search for an element
    - s.count(value) // Check if an element exists
  + **Modifiers**:
    - s.insert(value) // Add an element
    - s.erase(value) // Remove an element
    - s.clear() // Remove all elements
  + **Capacity**:
    - s.size() // Number of elements
    - s.empty() // Check if the set is empty
  + **Iterators**:
    - s.begin() // Iterator to the beginning
    - s.end() // Iterator to the end
    - s.rbegin() // Reverse iterator to the last element
    - s.rend() // Reverse iterator to the position before the first element

**Experiment with Various Operations**

* **Adding Elements**: Use insert() and test with different data types.
* **Removing Elements**: Use erase() and clear() to test removal operations.
* **Accessing Elements**: Practice searching with find(), checking counts with count().
* **Iterators**: Test both forward and reverse iterators to traverse and modify the set.

**Review Performance Considerations**

* **Memory Management**: Understand how std::set maintains balance and manages memory without needing to reallocate like std::vector.
* **Iterator Invalidation**: Modifications such as insert() and erase() can invalidate iterators. Be aware of iterator validity when performing these operations.

**=>**

* #include <iostream>
* #include <utility>
* class Set **{**
* private**:**
* int**\*** arr**;**
* int size**;**
* int capacity**;**
* bool contains**(**int value**)** const **{**
* **for** **(**int i **=** 0**;** i **<** size**;** **++**i**)** **{**
* **if** **(**arr**[**i**]** **==** value**)** **{**
* **return** **true;**
* **}**
* **}**
* **return** **false;**
* **}**
* public**:**
* Set**(**int cap**)** **:** size**(**0**),** capacity**(**cap**)** **{**
* arr **=** **new** int**[**capacity**];**
* **}**
* **~**Set**()** **{**
* **delete[]** arr**;**
* **}**
* void insert**(**int value**)** **{**
* **if** **(!**contains**(**value**)** **&&** size **<** capacity**)** **{**
* arr**[**size**++]** **=** value**;**
* **}**
* **}**
* void erase**(**int value**)** **{**
* int index **=** **-**1**;**
* **for** **(**int i **=** 0**;** i **<** size**;** **++**i**)** **{**
* **if** **(**arr**[**i**]** **==** value**)** **{**
* index **=** i**;**
* **break;**
* **}**
* **}**
* **if** **(**index **!=** **-**1**)** **{**
* **for** **(**int i **=** index**;** i **<** size **-** 1**;** **++**i**)** **{**
* arr**[**i**]** **=** arr**[**i **+** 1**];**
* **}**
* **--**size**;**
* **}**
* **}**
* int**\*** find**(**int value**)** **{**
* **for** **(**int i **=** 0**;** i **<** size**;** **++**i**)** **{**
* **if** **(**arr**[**i**]** **==** value**)** **{**
* **return** **&**arr**[**i**];**
* **}**
* **}**
* **return** **nullptr;**
* **}**
* int count**(**int value**)** const **{**
* **return** contains**(**value**)** **?** 1 **:** 0**;**
* **}**
* int size**()** const **{**
* **return** size**;**
* **}**
* bool empty**()** const **{**
* **return** size **==** 0**;**
* **}**
* void clear**()** **{**
* size **=** 0**;**
* **}**
* int**\*** lower\_bound**(**int value**)** **{**
* **for** **(**int i **=** 0**;** i **<** size**;** **++**i**)** **{**
* **if** **(**arr**[**i**]** **>=** value**)** **{**
* **return** **&**arr**[**i**];**
* **}**
* **}**
* **return** **nullptr;**
* **}**
* int**\*** upper\_bound**(**int value**)** **{**
* **for** **(**int i **=** 0**;** i **<** size**;** **++**i**)** **{**
* **if** **(**arr**[**i**]** **>** value**)** **{**
* **return** **&**arr**[**i**];**
* **}**
* **}**
* **return** **nullptr;**
* **}**
* std**::**pair**<**int**\*,** int**\*>** equal\_range**(**int value**)** **{**
* int**\*** first **=** **nullptr;**
* int**\*** last **=** **nullptr;**
* **for** **(**int i **=** 0**;** i **<** size**;** **++**i**)** **{**
* **if** **(**arr**[**i**]** **==** value**)** **{**
* **if** **(!**first**)** **{**
* first **=** **&**arr**[**i**];**
* **}**
* last **=** **&**arr**[**i**]** **+** 1**;**
* **}**
* **}**
* **return** std**::**make\_pair**(**first**,** last**);**
* **}**
* int**\*** begin**()** **{**
* **return** **&**arr**[**0**];**
* **}**
* int**\*** end**()** **{**
* **return** **&**arr**[**size**];**
* **}**
* int**\*** rbegin**()** **{**
* **return** size **>** 0 **?** **&**arr**[**size **-** 1**]** **:** **nullptr;**
* **}**
* int**\*** rend**()** **{**
* **return** **nullptr;** // Not implemented for raw arrays
* **}**
* **};**
* int main**()** **{**
* Set mySet**(**10**);**
* mySet**.**insert**(**5**);**
* mySet**.**insert**(**3**);**
* mySet**.**insert**(**8**);**
* mySet**.**insert**(**1**);**
* std**::**cout **<<** "Elements in the set: "**;**
* **for** **(**int**\*** it **=** mySet**.**begin**();** it **!=** mySet**.**end**();** **++**it**)** **{**
* std**::**cout **<<** **\***it **<<** " "**;**
* **}**
* std**::**cout **<<** std**::**endl**;**
* std**::**cout **<<** "Reverse elements in the set: "**;**
* **for** **(**int**\*** it **=** mySet**.**rbegin**();** it **!=** mySet**.**rend**();** **--**it**)** **{**
* std**::**cout **<<** **\***it **<<** " "**;**
* **}**
* std**::**cout **<<** std**::**endl**;**
* **return** 0**;**
* **}**