DAY 08

Templated containers, iterators, algorithms

Standard Template Library (STL):

Libraries: inside libraries (container/iterator/algorithm)

• The Standard Template Library (STL) is a collection of C++ template classes and functions that provide a set of reusable data structures and algorithms. The STL is part of the C++ Standard Library and is included in all modern C++ compilers.

• The STL provides a powerful and flexible set of tools for C++ programmers. By using the STL, you can write more concise and efficient code and reduce the amount of time you spend debugging and maintaining your code.

• The STL is organized into several components:

1. **Containers:** These are classes that hold objects of various types, such as arrays, vectors, lists, maps, and sets. Containers provide a way to manage groups of objects efficiently.(Data storage)

2. **Iterators:** These are objects that provide a way to traverse the elements of a container. Iterators act as a generalization of pointers and provide a way to access and modify the elements of a container.

3. **Algorithms:** These are functions that operate on sequences of elements provided by iterators. Algorithms include functions like sort, find, and count.

Containers :

- Containers are classes that are used to store and organize collections of objects or values , They are implemented as class templates, which allows great flexibility in the types supported as elements.

  - There are several types of containers, each designed for a specific purpose :

1. **Sequence Containers are containers that store elements in a specific order. They are implemented as templates and allow fast random access to elements.**

   - **Vector:** Vectors are dynamic arrays that can grow or shrink as elements are added or removed.
     - **Adding elements:** Vectors provide functions like push_back() and insert() to add elements to the container.
     - **Removing elements:** Vectors provide functions like pop_back() and erase() to remove elements from the container.
     - **Accessing elements:** Elements in a vector can be accessed using an index or an iterator.
     - **Modifying elements:** Elements in a vector can be modified using an index or an iterator.
     - **Resizing the container:** The size of a vector can be changed using the resize() function.

     - **Random Access:** Elements in a vector can be accessed in constant time using an index, similar to an array.
     - **Dynamic Size:** Vectors can grow or shrink dynamically as elements are added or removed. They automatically resize themselves when needed.
     - **Contiguous Memory:** Elements in a vector are stored in a contiguous block of memory, which allows for efficient memory access.
     - **Iterator Support:** Vectors provide iterator support for iterating over elements in the container.
     - **Standardized Interface:** Vectors provide a consistent interface with other sequence containers in the STL, such as list and deque.

   - **Declarations :**
     - **std::vector<int> vec;**
     - **std::vector<int> vec(5); // creates a vector with 5 elements, all initialized to 0**
     - **std::vector<int> vec(5, 10); // creates a vector with 5 elements, all initialized to 10**
     - **std::vector<int> vec = {1, 2, 3};**
     - **std::vector<int> vec {1, 2, 3};**
   - **Accessing Element :**
     - **vec[1] = 5;**
     - **vec.at(1) = 5;**

   - **List :** Sequence container that stores elements in a linked list. Unlike vector and deque, it does not provide constant-time random access to elements, but allows for fast insertion and deletion at any position in the list

     - **Fast Insertion and Deletion:** Elements can be added or removed from the list in constant time, regardless of their position in the list.
     - **No Random Access:** Elements cannot be accessed in constant time using an index, but must be traversed using iterators.
     - **Dynamic Size:** The size of the list can grow or shrink dynamically as elements are added or removed.
     - **Iterator Support:** Lists provide iterator support for iterating over elements in the container.
     - **Standardized Interface:** Lists provide a consistent interface with other sequence containers in the STL, such as vector and deque.

     - **Adding elements:** Elements can be added to the list using the push_front(), push_back(), or insert() functions.
     - **Removing elements:** Elements can be removed from the list using the pop_front(), pop_back(), or erase() functions.
     - **Accessing elements:** Elements in a list can be accessed using an iterator.
     - **Modifying elements**: Elements in a list can be modified using an iterator.
     - **Reversing the list:** The list can be reversed using the reverse() function.

   - **Declarations :**

     - **std::list<int> lst;**
     - **std::list<int> lst(5); // creates a list with 5 elements, all initialized to 0**
     - **std::list<int> lst(5, 10); // creates a list with 5 elements, all initialized to 10**
     - **std::list<int> lst = {1, 2, 3};**
     - **std::list<int> lst {1, 2, 3};**

   - **Accessing Element :**
     - **std::list<int> lst = {1, 2, 3};**
     - **int x = lst.front(); // assigns x the value 1**
     - **int y = lst.back(); // assigns y the value 3**

     - **std::list<int> lst = {1, 2, 3};**
     - **std::list<int>::iterator it = lst.begin(); // get an iterator to the beginning of the list**
     - **int x = *it; // assigns x the value 1**
     - **++it; // move the iterator to the next element**
     - **int y = *it; // assigns y the value 2**

2. **Associative containers are containers that store elements in a sorted order based on a key .They allow for efficient searching and retrieval of elements based on their key values .**

   - **Set** Associative container ithat stores a sorted set of unique elements. The elements are sorted in ascending order by default, but the sorting order can be customized by providing a comparison function. The elements in the set are always unique, meaning that duplicate elements are not allowed.

     `std::set<DataType> mySet;`

     - **Elements are stored in a sorted order .**
     - **Duplicate elements are not allowed .**
     - **Searching, insertion, and deletion operations are efficient .**
     - **.The size of the set can grow dynamically .**
     - **Iterating over the elements in a set is done in sorted order .**

   - **Declarations :**

     - **std::set<int> mySet;**
     - **std::set<std::string> mySet = {"apple", "banana", "cherry"};**
     - **std::set<Person> mySet = {{"Alice", 25}, {"Bob", 30}, {"Charlie", 20}};**

   - **Accessing Element :**

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

     ```
     auto res = mySet.insert(6);
     if (res.second) {
       std::cout << "Inserted: " << *res.first << std::endl;                    // Output: Inserted: 6
     } else {
     std::cout << "Already exists" << std::endl;}
     ```

   - **Map** Associative container that stores elements in key-value pairs. The keys are unique and stored in a sorted order, and each key is associated with a value.

     `std::map<KeyType, ValueType> mapName;`

     - **Elements are stored as key-value pairs.**
     - **Keys are unique and stored in a sorted order.**
     - **Values can be accessed and modified using the keys.**
     - **Insertion, deletion, and search operations have a time complexity of O(log n).**
     - **Iteration over the elements is done in a sorted order.**

   - **Declarations :**

     - **std::map<KeyType, ValueType> mapName;**
     - **std::map<KeyType, ValueType> mapName = std::map<KeyType, ValueType> { {key1, value1}, {key2, value2}, ... };**

   - **Accessing Element :**

     - **ValueType value = mapName[key];**
     - **mapName[key] = newValue;**

3. **Unordered Containers are a type of containers in STL that store elements in an unordered manner. They are implemented using Hash Tables. These containers provide faster search, insertion, and deletion operations .**

   - **Unordered_set** Unordered container that stores unique elements in no particular order. It is implemented using a hash table, which allows for fast search, insertion, and deletion operations.

     `std::unordered_set<int> mySet;`

     - **Stores unique elements in no particular order.**
     - **Implemented using a hash table, which allows for fast search, insertion, and deletion operations.**
     - **Operations (search, insertion, and deletion) have average constant time complexity O(1) under the assumption of uniform distribution of hash values.**
     - **Supports iterators for traversing the elements of the container.**
     - **Does not allow duplicate elements, as it only stores unique elements.**

   - **Declarations :**

     - **std::unordered_set<int> mySet;**
     - **std::unordered_set<int> mySet {3, 5, 1, 4, 2};**

   - **Accessing Element :**

     - **std::unordered_set<int> mySet {3, 5, 1, 4, 2};**
       mySet.insert(6);
     - **std::unordered_set<int> my_set;**
       my_set.emplace(10);

   - **Unordered_map** Container that is used to store key-value pairs in an unordered manner. It is implemented as a hash table, which allows for fast retrieval of elements based on their keys.

     `std::unordered_map<key_type, value_type> map_name;`

     - **Elements are stored as key-value pairs.**
     - **Keys are unique in the map, while values may be duplicated.**
     - **Elements are stored in an unordered manner, based on their hash values.**
     - **The complexity of operations such as insertion, deletion, and search is O(1) on average.**

   - **Declarations :**

     - **std::unordered_map<std::string, int> my_map;**
     - **std::unordered_map<std::string, int> my_map = { {"apple", 1}, {"banana", 2}, {"orange", 3} };**

   - **Accessing Element :**

     - **ValueType value = mapName[key];**
     - **mapName[key] = newValue;**

4. **Container Adapter are designed to be used in specific situations or scenarios , that provide a restricted interface to another container and are designed for specific purposes .These adapters are implemented using the sequence containers (vector, deque, and list) or other containers (such as the heap).**

   - **Stack** Container adapter that provides a Last-In-First-Out (LIFO) data structure. This means that the last element added to the stack will be the first one to be removed. Stacks are commonly used in many algorithms and data structures, such as expression evaluation, depth-first search, and backtracking.

     `std::stack<int> myStack;`

     - **push(element): Adds an element to the top of the stack.**
     - **pop(): Removes the top element from the stack.**
     - **top(): Returns a reference to the top element on the stack.**
     - **size(): Returns the number of elements on the stack.**
     - **empty(): Returns true if the stack is empty, false otherwise.**

   - **Queue** Container adapter provides an interface for working with a first-in, first-out (FIFO) data structure. It is implemented using a sequence container as its underlying storage, and it restricts the available operations to push elements onto the back of the container and pop elements from the front of the container.

     `std::queue<int> myStack;`

     - **enqueue(element) - adds an element to the back of the queue .**
     - **dequeue() - removes the front element from the queue .**
     - **front() - returns the front element of the queue .**
     - **empty() - returns true if the queue is empty, false otherwise .**
     - **size() - returns the number of elements in the queue .**

Iterator :

- Iterator is an object that provides a way to access the elements of a container (such as an array, vector, list, set, or map) sequentially. It acts as a pointer to the elements of the container, and provides a way to traverse the container without knowing its underlying implementation.

  - There are several types of iterators in C++, including:

1. **Input iterators:** These iterators can be used to read the elements of a container in a forward-only manner.

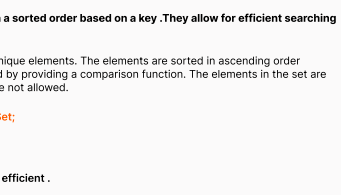
2. **Output iterators:** These iterators can be used to write the elements of a container in a forward-only manner.

3. **Forward iterators:** These iterators can be used to read and write the elements of a container in a forward-only manner.

4. **Bidirectional iterators:** These iterators can be used to read and write the elements of a container in both forward and backward directions.

5. **Random access iterators:** These iterators can be used to read and write the elements of a container in any direction, and can be used to access the elements of the container using indexing.

CONTAINER	TYPES OF ITERATOR SUPPORTED
Vector	Random-Access
List	Bidirectional
Deque	Random-Access
Map	Bidirectional
Multimap	Bidirectional
Set	Bidirectional
Multiset	Bidirectional
Stack	No iterator Supported
Queue	No iterator Supported
Priority-Queue	No iterator Supported



ITERATORS	PROPERTIES				
	ACCESS	READ	WRITE	ITERATE	COMPARE
Input	->	= *i		++	==, !=
Output			*i=	++	
Forward	->	= *i	*i=	++	==, !=
Bidirectional		= *i	*i=	++, --	==, !=,
Random-Access	->,[ ]	= *i	*i=	++, --, +=, -=, + ,-	==, !=, <,>,<=,>=

Algorithm :

- STL algorithms are template functions, which means that they can be used with any container type that satisfies certain equirements, such as providing the ability to access elements with iterators. The algorithms are implemented as generic functions that can work with any container type, such as vectors, lists, sets, maps, and arrays.

  Some common algorithms provided by the STL include:

  - **Sorting algorithms (e.g., sort(), stable_sort(), partial_sort())**
  - **Searching algorithms (e.g., find(), binary_search())**
  - **Numeric algorithms (e.g., accumulate(), inner_product(), partial_sum())**
  - **Mutating algorithms (e.g., fill(), copy(), transform())**
  - **Non-modifying sequence algorithms (e.g., all_of(), any_of(), none_of())**
  - **Set algorithms (e.g., set_union(), set_intersection(), set_difference())**

Containers hold data elements, iterators provide a way to access those elements, and algorithms operate on the data using iterators .