C++ Standard Template Library (STL)

Last Updated : 11 Oct, 2024

The C++ Standard Template Library (STL) is a set of template classes and functions that provides the implementation of common data structures and algorithms such as lists, stacks, arrays, sorting, searching, etc. It also provides the iterators and functors which makes it easier to work with algorithms and containers.

STL was originally designed by Alexander Stepanov and was later accepted as the part of C++ standard in C++ 98. It is a generalized library so we can use it with almost every data type without repeating the implementation code.

Components of STL

The components of STL are the features provided by Standard Template Library (STL) in C++ that can be classified into 4 types:

1. **Containers**
2. **Algorithms**
3. **Iterators**
4. **Functors**

These components are designed to be efficient, flexible, and reusable, making them an integral part of modern C++ programming.

Containers

Containers are the data structures used to store objects and data according to the requirement. Each container is implemented as a template class that also contains the methods to perform basic operations on it. Every STL container is defined inside its own header file.

Containers can be further classified into 4 types:

1. **Sequence Containers**
2. **Container Adaptors**

3. **Associative Containers**
4. **Unordered Associated Containers**

If you want to dive deep into STL and understand its full potential, our **Complete C++ Course** offers a complete guide to mastering containers, iterators, and algorithms provided by STL.

Sequence Containers

Sequence containers store the data in the linear manner. They are also used to implement container adaptors.
There are 5 sequence containers in C++ STL:

1. **Arrays**: The STL array is an implementation of a compile time non-resizable array. It contains various method for common array operations.
2. **Vector**: An STL vector can be defined as the dynamic sized array which can be resized automatically when new elements are added or removed.
3. **Deque**: Deque or Double-Ended Queue is sequence containers with the feature of expansion and contraction on both ends. It means we can add and remove the data to and from both ends.
4. **Lists**: List container stores data in non-contiguous memory unlike vectors and only provide sequential access to the stored data. It basically implements the doubly linked list.
5. **Forward Lists:** Forward lists also store the data in a sequential manner like lists, but with the difference that forward list stores the location of only the next elements in the sequence. It implements the singly linked list.

Container Adaptors

The container adapters are the type of STL containers that adapt existing container classes to suit specific needs or requirements.
There are 3 container adaptors in C++ STL:

1. **Stack:** STL Stack follows the Last In First Out (LIFO) principle of element insertion and deletion. Also, these operations are performed only at one end of the stack.

2. **Queue**: STL Queue follows the First In First Out (FIFO) principle, means the element are inserted first are removed first and the elements inserted last are removed at last. It uses deque container by default.
3. **Priority Queue**: STL Priority Queue does not follow any of the FIFO or LIFO principle, but the deletion of elements is done on the basis of its priority. So, the element with the highest (by default) is always removed first. By default, it uses vector as underlying container.

Associative Containers

Associative containers are the type of containers that store the elements in a sorted order based on keys rather than their insertion order.
There are 4 associative containers in C++ STL:

1. **Sets**: STL Set is a type of associative container in which each element has to be unique because the value of the element identifies it. By default, the values are stored in ascending order.
2. **Maps**: STL Maps are associative containers that store elements in the form of a key-value pair. The keys have to be unique and the container is sorted on the basis of the values of the keys.
3. **Multisets**: STL Multiset is similar to the set container except that it can store duplicate values.
4. **Multimaps**: STL Multimap is similar to a map container but allows multiple mapped values to have same keys.

Unordered Associative Containers

Unordered associative containers store the data in no particular order, but they allow the fastest insertion, deletion and search operations among all the container types in STL.
There are 4 unordered associative containers in C++ STL:

1. **Unordered Set**: STL Unordered Set stores the unique keys in the form of hash table. The order is randomized but insertion, deletion and search are fast.
2. **Unordered Multiset**: STL Unordered Multiset works similarly to an unordered set but can store multiple copies of the same key.

3. **Unordered Map**: STL Unordered Map stores the key-value pair in a hash table, where key is hashed to find the storage place.
4. **Unordered Multimap**: STL Unordered Multimap container is similar to unordered map, but it allows multiple values mapped to the same key.

Algorithms

STL algorithms offer a wide range of functions to perform common operations on data (mainly containers). These functions implement the most efficient version of the algorithm for tasks such as sorting, searching, modifying and manipulating data in containers, etc. All STL algorithms are defined inside the **<algorithm>** and **<numeric>** header file.
There is no formal classification of STL algorithms, but we can group them into two types based on the type of operations they perform:

Manipulative Algorithms

Manipulative algorithms perform operations that modifies the elements of the given container or rearrange their order.
Some of the common manipulative algorithm includes:

- **copy**: Copies a specific number of elements from one range to another.
- **fill**: Assigns a specified value to all elements in a range.
- **transform**: Applies a function to each element in a range and stores the result in another range.
- **replace**: Replaces all occurrences of a specific value in a range with a new value.
- **swap**: Exchanges the values of two variables.
- **reverse**: Reverses the order of elements in a range.
- **rotate**: Rotates the elements in a range such that a specific element becomes the first.
- **remove**: Removes all elements with a specified value from a range but does not reduce the container size.
- **unique**: Removes consecutive duplicate elements from a range.

Non-Manipulative Algorithms

Non-manipulating algorithms are the type of algorithms provided by the Standard Template Library (STL) that operate on elements in a range without altering their values or the order of the elements.
The below are the few examples of the STL's non-manipulative algorithms:

- **max_element**: Find the maximum element in the given range.
- **min_element** : To find the minimum element in the given range.
- **accumulate**: Finds the sum of the elements of the given range.
- **count**: Counts the occurrences of given element in the range.
- **find**: Returns an iterator to the first occurrence of an element in the range.
- **is_permutation**: Checks if one range is a permutation of another.
- **is_sorted**: Checks if the elements in a range are sorted in non-decreasing order.
- **partial_sum**: Computes the cumulative sum of elements in a range.

Iterators

Iterators are the pointer like objects that are used to point to the memory addresses of STL containers. They are one of the most important components that contributes the most in connecting the STL algorithms with the containers. Iterators are defined inside the **<iterator>** header file.

In C++ STL, iterators are of 5 types:

1. **Input Iterators**: Input Iterators can be used to read values from a sequence once and only move forward.
2. **Output Iterators**: Output Iterators can be used to write values into a sequence once and only move forward.
3. **Forward Iterators**: Forward Iterators combine the features of both input and output iterators.
4. **Bidirectional Iterators**: Bidirectional Iterators support all operations of forward iterators and additionally can move backward.
5. **Random Access Iterators**: Random Access Iterators support all operations of bidirectional iterators and additionally provide efficient random access to elements.

Functors

Functors are objects that can be treated as though they are a function. Functors are most commonly used along with STL algorithms. It overloads the **function-call operator** `()` and allows us to use an object like a function. There are many predefined functors in C++ STL that are defined inside the **<functional>** header file.

Functors can be classified into multiple types based on the type of operator they perform:

1. **Arithmetic Functors**
2. **Relational Functors**
3. **Logical Functors**
4. **Bitwise Functors**

Arithmetic Functors

- **plus –** Returns the sum of two parameters.
- **minus** – Returns the difference of two parameters.
- **multiplies –** Returns the product of two parameters.
- **divides –** Returns the result after dividing two parameters.
- **modulus –** Returns the remainder after dividing two parameters.
- **negate –** Returns the negated value of a parameter.

Relational Functors

- **equal_to** – Returns true if the two parameters are equal.
- **not_equal_to** – Returns true if the two parameters are not equal.
- **greater** – Returns true if the first parameter is greater than the second.
- **greater_equal** – Returns true if the first parameter is greater than or equal to the second.
- **less** – Returns true if the first parameter is less than the second.
- **less_equal** – Returns true if the first parameter is less than or equal to the second.

Logical Functors

- **logical_and** – Returns the result of Logical AND operation of two parameters.
- **logical_or** – Returns the result of Logical OR operation of two parameters.
- **logical_not** – Returns the result of Logical NOT operation of the parameters.

Bitwise Functors

- **bit_and** – Returns the result of Bitwise AND operation of two parameters.
- **bit_or** – Returns the result of Bitwise OR operation of two parameters.
- **bit_xor** – Returns the result of Bitwise XOR operation of two parameters.

Utility and Memory Library

The Utility Library is a collection of utility components provided by the Standard Template Library (STL) that does not fall in the above categories. It offers various features such as pairs, tuples, etc.

The memory library contains the function that helps users to efficiently manage the memory such as std::move, smart pointers, etc.

1. **Pair**: Container to store and manipulate heterogeneous data.
2. **Move Semantics**: It allows the transfer of resources from one object to another without copying.
3. **Smart Pointers**: They are a wrapper over the raw pointers and helps in avoiding errors associated with pointers.
4. **Utility Functions**: Utility functions in C++ provide important operations like std::forward to facilitate efficient , generic and safe code manipulation.
5. **Integer Sequence**: Enable compile-time generation of integer sequences, useful in metaprogramming.

Benefits of C++ Standard Template Library (STL)

The key benefit of the STL is that it provides a way to **write generic, reusable code and tested code** that can be applied to different data types. This means you can write an algorithm once, and then use it with other types of data without having to write separate code for each type.
Other benefits include:

- STL provides flexibility through customizable templates, functors, and lambdas.
- Pre-implemented tools let you focus on problem-solving rather than low-level coding.
- STL handles memory management which reduces common errors like memory leaks.

Limitations of C++ Standard Template Library (STL)

The major limitation of the C++ Standard Template Library (STL) is **Performance Overheads**. While STL is highly optimized for general use cases, its generic nature can lead to less efficient memory usage and execution time compared to custom and specialized solutions.
Other limitations can be:

- Complexity of debugging.
- Lack of control over memory management.
- Limited support for concurrent programming.
- Limited flexibility with custom data structures integrating.

Despite these limitations, STL remains an invaluable part of C++ programming, offering a wide range of powerful and flexible tools.