MQF633 C++ FOR FINANCIAL ENGINEERING

Lecture 6 C++ Standard Template Library

C++ Standard Template Library

Definition

The C++ Standard Template Library (STL) is a powerful set of template classes and functions that provide common programming data structures and algorithms. It offers ready-to-use components like containers (e.g., vectors, lists, maps), algorithms (e.g., sorting, searching), and iterators (to traverse containers), significantly simplifying and speeding up C++ development by promoting code reusability and efficiency. It's designed to work seamlessly with various data types.

Key Components

- ### Containers:

Containers are objects that store data. Some commonly used containers include vectors, lists, queues, stacks, sets, maps, etc.

- ### Algorithms:

STL provides a set of generic algorithms that operate on these containers. These algorithms include sorting, searching, and manipulation functions, making it easy to perform common operations on different container types.

- ### Iterators

Iterators are used to traverse elements in a container. They act as a generalisation of pointers and provide a way to access elements in a container sequentially.

Standard Template Library Containers

- **Sequence Containers:**

Examples include vector, list, deque.

- **Associative Containers:**

Examples include set, map, multiset, multimap.

- **Container Adapters:**

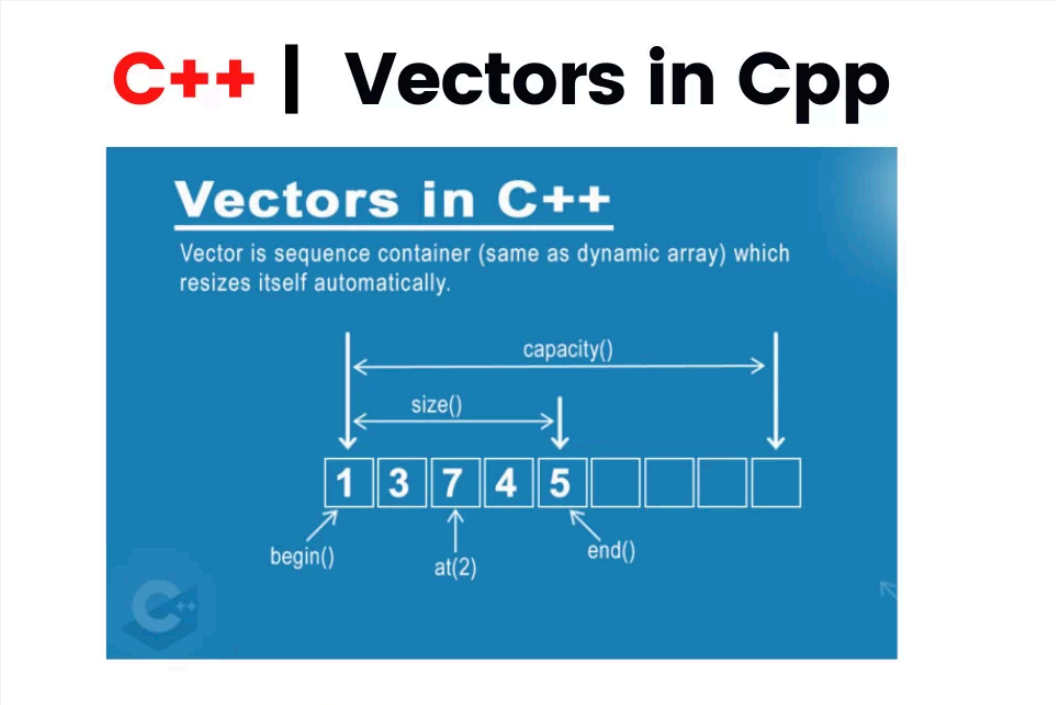
Examples include stack, queue, priority_queue.

Vector: std::vector

Vectors are the same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes the array may need to be extended. Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time. (why?)

Syntax:

```
#include<vector>
std::vector<Type> vector_name = {value1, value2, ...};
```



Refer to code example vector.cpp

Use std::vector when:

- You need **fast random access** to elements by their index. Accessing an element at a specific position (e.g., vec[5]) is a constant-time operation, O(1).
- You mostly add or remove elements at the **end** of the sequence. Adding or removing at the end is usually O(1) on average (due to amortised analysis).
- Memory contiguity is important. std::vector stores its elements in contiguous memory locations, which can lead to better cache locality and performance for many operations, especially iteration.

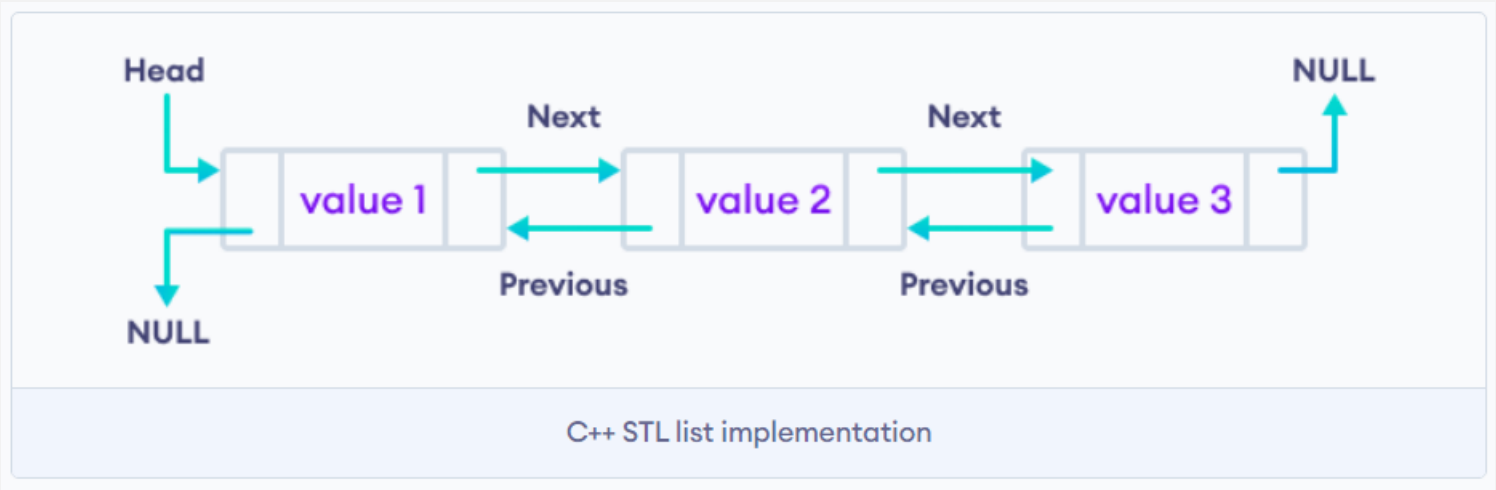
In summary, std::vector is generally the default choice when you need a dynamic array with efficient random access and appends/removals from the end.

Exercise: Check the element address within same vector

List: std::list

C++ List is a STL container that stores elements randomly in unrelated locations. To maintain sequential ordering, every list element includes two links:

- One that points to the previous element
- Another that points to the next element



C++ STL list implementation

In C++, the STL list implements the doubly-linked list data structure. As a result, we can iterate both forward and backward.

Syntax:
#include<list>
std::list<Type> list_name = {value1, value2, ...};

Refer to code example of list.cpp

Use std::list when:

- You need **frequent insertions and deletions** at arbitrary positions within the sequence. Inserting or deleting an element in the middle of a std::list is a constant-time operation, O(1), once you have an iterator to the position. This is because it only involves updating the pointers of the neighbouring nodes.
- You don't need or rarely need random access by index. Accessing an element at a specific index in a std::list requires traversing the list from the beginning, which is a linear-time operation, O(n).

In summary, std::list is preferred when you have many insertions and deletions in the middle of the sequence and random access is not a primary requirement.

Exercise: Check the element address within same list

Vector vs List

Operation	`std::vector`	`std::list`
Random Access	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Insert/Delete (end)	$\mathcal{O}(1)$ (average)	$\mathcal{O}(1)$
Insert/Delete (middle)	$\mathcal{O}(n)$	$\mathcal{O}(1)$ (with iterator)
Memory	Contiguous	Non-contiguous
Cache Locality	Generally better	Potentially worse

Example Scenarios:

- **std::vector:** Storing a collection of user profiles where you frequently access a user by their ID (if you maintain a separate mapping of ID to index) or simply iterate through them. Adding new users at the end is also common.
- **std::list:** Managing a playlist where you frequently insert or remove songs at various positions, and the order is important, but you don't often need to jump to the 50th song directly.

In most general-purpose scenarios where you need a dynamic array and frequent middle insertions/deletions are not the primary concern, **std::vector** is often the better choice due to its performance characteristics for random access and iteration.

Set: std::set

In C++, std::set is a part of the Standard Template Library (STL) and is a container that stores unique, ordered elements. It is implemented as a sorted binary tree, typically a Red-Black Tree, which ensures that the elements are always sorted in ascending order. As a result, the insertion, deletion, and search operations have a time complexity of O(log n).

Key features

- Uniqueness: Each element in a set must be unique. If an attempt is made to insert a duplicate element, it will not be added to the set.
- Ordered: The elements in a set are always ordered. This allows for efficient searching, insertion, and deletion operations.
- Associative Container: std::set is an associative container, meaning that it is based on key-value pairs. However, in a set, the key and value are the same, representing the element itself.
- Dynamic Sizing: The size of a set can grow or shrink dynamically as elements are inserted or removed.

Syntax:

```
#include<set>
std::set<Type> set_name;
```

Refer to example set.cpp

In C++, you should use std::set primarily when you need to store a collection of **unique** elements that are automatically **sorted**.
Here's a breakdown of when and why to use std::set:
When to use std::set:

1. **Maintaining unique elements:** If you need to ensure that your collection contains only distinct values, std::set automatically handles this. Any attempt to insert a duplicate element is simply ignored.
2. **Automatic sorting:** The elements in a std::set are always kept in a specific order (by default, using the < operator for the element type). This sorted order can be beneficial for various operations.
3. **Efficient searching, insertion, and deletion:** std::set is typically implemented using a self-balancing binary search tree (like a red-black tree). This data structure provides logarithmic time complexity (O(log N)) for searching, insertion, and deletion of elements.

Why use std::set?

- **Uniqueness:** It inherently enforces the uniqueness of its elements, simplifying code when you need to manage a set of distinct items.
- **Sorted order:** The automatic sorting can be useful when you need to iterate through elements in a specific order or perform operations that benefit from sorted data.
- **Performance for certain operations:** The logarithmic time complexity for key operations like searching, insertion, and deletion makes it efficient for dynamic collections where these operations are frequent.

In summary, use std::set when you need a collection of unique, sorted elements with efficient searching, insertion, and deletion.

std::unordered_set: Use when you need to store unique elements and want very fast average-case performance (O(1)) for insertion, deletion, and lookup, but you don't need the elements to be sorted.

Map: std::map

In C++, std::map is an associative container that stores key-value pairs, where each key is **unique**. Each element has a key value and a mapped value. No two mapped values can have the same key values. The elements in a std::map are **automatically sorted** based on their keys. std::map is the class template for map containers and it is defined inside the <map> header file.

Syntax:
#include<map>
std::set<map> map_name;

Refer to map.cpp example

When to use std::map:

1. **Need to associate values with unique keys:** When you have data that can be naturally represented as a mapping from one value (the key) to another (the value), std::map is an excellent choice. Think of dictionaries, configuration settings, or looking up information based on an ID.
2. **Require elements to be sorted by key:** The automatic sorting by key can be beneficial when you need to iterate through the key-value pairs in a specific order or when the sorted order simplifies other operations.
3. **Need efficient lookup by key:** std::map typically uses a self-balancing binary search tree (like a red-black tree) as its underlying implementation. This provides logarithmic time complexity (O(logn)) for searching for an element based on its key. Insertion and deletion also have logarithmic time complexity.

Why use std::map?

- **Key-value association:** It provides a clear and efficient way to manage relationships between keys and their corresponding values.
- **Automatic sorting by key:** This built-in sorting can simplify many tasks and provide a predictable order when iterating.
- **Efficient key-based access:** The logarithmic time complexity for lookup makes it suitable for scenarios where you frequently need to retrieve values based on their keys, even in large datasets.
- **Uniqueness of keys:** Each key in a std::map must be unique. If you try to insert a key that already exists, the existing element is either not modified (in the case of insert) or the value associated with the key is updated (using the [] operator).

In summary, use std::map when you need to store and efficiently retrieve values based on unique, sorted keys.

Example Scenarios:

- A dictionary where words (keys) are associated with their definitions (values).
- Configuration settings where setting names (keys) are associated with their values.
- Counting the occurrences of words in a text (where words could be keys and counts could be values).
- An index where page numbers (keys) map to the topics discussed on those pages (values).

In C++, std::unordered_map is an associative container that stores key-value pairs, similar to std::map. However, it differs significantly in how it organises its elements and the performance characteristics of its operations.

What is std::unordered_map?
std::unordered_map is implemented using a **hash** table. This means that elements are not stored in any particular order (hence "unordered"). Instead, the position of each key-value pair is determined by the hash value of the key.

std:unordered_map vs std::map

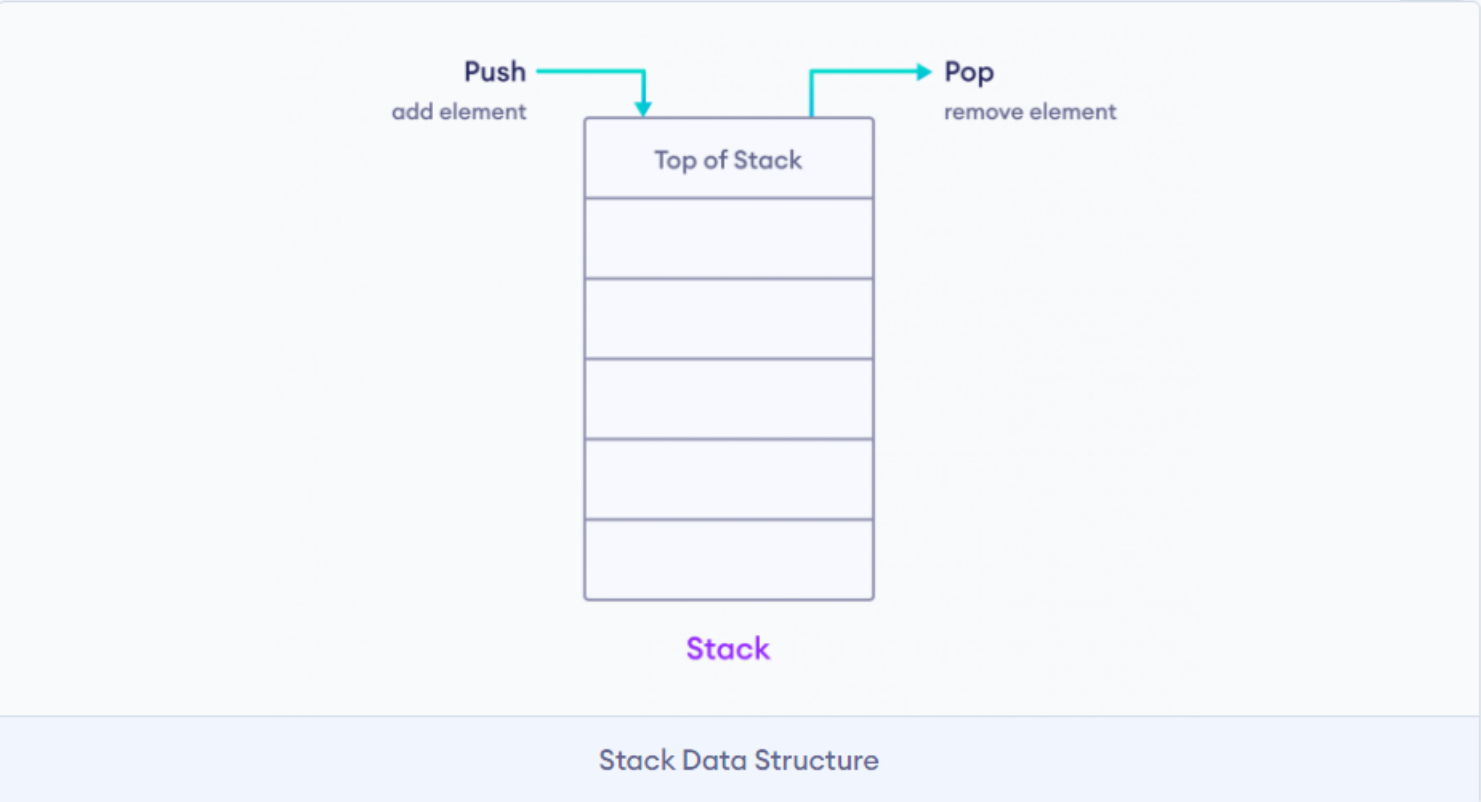
Feature	`std::unordered_map`	`std::map`
Underlying Data Structure	Hash table	Self-balancing binary search tree (e.g., red-black tree)
Ordering	No specific order (unordered)	Sorted by key
Lookup (by key)	Average $\mathcal{O}(1)$, worst $\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Insertion	Average $\mathcal{O}(1)$, worst $\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Deletion	Average $\mathcal{O}(1)$, worst $\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Memory Usage	Generally can be higher due to hash table overhead	Generally lower overhead per element

Use std::map when
- You need ordered data.
- You would have to print/access the data (in sorted order).
- You need predecessor/successor of elements.

Use std::unordered_map when
- You need to keep count of some data (Example – strings) and no ordering is required.
- You need single element access i.e. no traversal.

Stack: std::stack

The STL stack provides the functionality of a stack data structure in C++. The stack data structure follows the LIFO (Last In First Out) principle. That is, the element added last will be removed first.



Stack Data Structure

Operation	Description
`push()`	adds an element into the stack
`pop()`	removes an element from the stack
`top()`	returns the element at the top of the stack
`size()`	returns the number of elements in the stack
`empty()`	returns `true` if the stack is empty

Syntax

```
#include <stack>
std::stack<type_name> my_stack;
```

Refer to example of stack.cpp

When we shall use Stack?

- **Managing function calls:** Compilers often use a stack to keep track of active function calls. When a function is called, its information is "pushed" onto the stack. When the function returns, its information is "popped" off.
- **Undo/Redo functionality:** Many applications implement undo/redo features using one or more stacks. Each action is pushed onto the undo stack, and when you undo, the last action is popped. Redo operations can be managed with another stack.
- **Expression evaluation:** Stacks are crucial in evaluating arithmetic expressions, especially those involving parentheses, by handling operator precedence and operands.

- **Backtracking algorithms:** In algorithms that explore possibilities step-by-step and might need to backtrack (go back to a previous state), a stack can be used to keep track of the path taken. Examples include solving mazes or searching trees.
- **Reversing a sequence:** You can push all elements of a sequence onto a stack and then pop them off one by one to get the reversed sequence.

Duque: std::deque

In C++, std::deque (pronounced "deck," short for "double-ended queue") is a sequence container that, like std::vector, supports dynamic resizing. However, the key advantage of std::deque is that it allows for **efficient insertion and deletion at both the beginning and the end** of the sequence.

Syntax

```
#include <deque>
std::deque<type_name> my_deque;
```

Refer to deque.cpp

When to use std::deque:

- **Frequent insertions/deletions at both ends:** If your application requires adding or removing elements from both the front and the back of a sequence frequently, std::deque is a better choice than std::vector. While std::vector offers efficient operations at the end, inserting or deleting at the beginning is costly (O(n)) because it requires shifting all other elements. std::deque provides (amortized) O(1) time complexity for these operations at both ends.
- **Implementing queues or double-ended queues:** Naturally, std::deque is a suitable underlying container for implementing your own queue (where you insert at the back and remove from the front) or a double-ended queue data structure.
- **When you need some features of std::vector (like random access) but also efficient front-end operations:**std::deque provides O(1) random access (though potentially slightly slower constant factor than std::vector because the elements might not be in a single contiguous block of memory).

Why use std::deque?

- **Efficient front and back insertions/deletions:** This is the primary reason to choose std::deque over std::vector.
- **Random access:** It still provides O(1) random access to its elements, unlike std::list.
- **Dynamic size:** Like std::vector, it can grow or shrink as needed.

In summary, use std::deque when you need a dynamic array-like structure that supports efficient insertion and deletion at both its beginning and end, and you might also need random access.

Queue: std::queue

In C++, queue is a type of container adaptors that operate in a **first in first out (FIFO)** type of arrangement. Elements are inserted at the back (end) and are deleted from the front. Queues use an encapsulated object of deque or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

Method	Definition
queue::empty()	Returns whether the queue is empty. It return true if the queue is empty otherwise returns false.
queue::size()	Returns the size of the queue.
queue::swap()	Exchange the contents of two queues but the queues must be of the same data type, although sizes may differ.
queue::emplace()	Insert a new element into the queue container, the new element is added to the end of the queue.
queue::front()	Returns a reference to the first element of the queue.
queue::back()	Returns a reference to the last element of the queue.
queue::push(g)	Adds the element 'g' at the end of the queue.
queue::pop()	Deletes the first element of the queue.

Syntax

```
#include <queue>
std::queue<type_name> my_queue
```

Refer to queue.cpp.

print_queue function:
Time complexity: O(n), where n is the number of elements in the queue.
Space complexity: O(n), where n is the number of elements in the queue.

q1.push(1), q1.push(2), q1.push(3), q2.push(4), q2.push(5), q2.push(6):
Time complexity: O(1) for each push operation.
Space complexity: O(n), where n is the total number of elements in both queues.

q1.swap(q2):
Time complexity: O(1) for each swap operation.
Space complexity: O(1), as this operation only swaps the internal pointers of the two queues.

q1.empty():
Time complexity: O(1), as this operation simply checks if the queue is empty.
Space complexity: O(1), as no extra space is used for this operation.

Overall, the time and space complexities of this code are reasonable and efficient for typical use cases.

Standard Template Library Algorithm

For all those who aspire to excel in competitive programming, only having a knowledge about containers of STL is of less use till one is not aware what all STL has to offer. STL has an ocean of algorithms, for all < algorithm > library functions.Some of the most used algorithms on vectors and most useful one's in Competitive Programming are mentioned as follows:

- sort(first_iterator, last_iterator) – To sort the given vector.
- reverse(first_iterator, last_iterator) – To reverse a vector. (if ascending -> descending OR if descending -> ascending)
- *max_element (first_iterator, last_iterator) – To find the maximum element of a vector.
- *min_element (first_iterator, last_iterator) – To find the minimum element of a vector.
- accumulate(first_iterator, last_iterator, initial value of sum) – Does the summation of vector elements
- count(first_iterator, last_iterator, x) – To count the occurrences of x in vector.
- find(first_iterator, last_iterator, x) – Returns an iterator to the first occurrence of x in vector and points to last address of vector ((name_of_vector).end()) if element is not present in vector.

Sorting: std::sort()

It generally takes two parameters, the first one being the point of the array/vector from where the sorting needs to begin and the second parameter being the length up to which we want the array/vector to get sorted. The third parameter is optional and can be used in cases such as if we want to sort the elements lexicographically. By default, the sort() function sorts the elements in ascending order.

Refer to sort.cpp

Output
Array after sorting using default sort is :
0 1 2 3 4 5 6 7 8 9
Time Complexity: O(N log N)
Auxiliary Space: O(1)

What about sorting in descending order?

std::sort() takes a third parameter that is used to specify the order in which elements are to be sorted. We can pass the "greater()" function to sort in descending order. This function does a comparison in a way that puts greater elements before.

std::greater() defintion:

```
template <class T> struct greater {
        bool operator() (const T& x, const T& y) const {return x>y;}
        typedef T first_argument_type;
        typedef T second_argument_type;
        typedef bool result_type;
};
```

Find max element: std::max_element()

We have std::max to find maximum of 2 or more elements, but what if we want to find the largest element in an array or vector or list or in a sub-section. To serve this purpose, we have std::max_element in C++. std::max_element is defined inside the header file and it returns an iterator pointing to the element with the largest value in the range [first, last). std::max_element can be used in two ways. The comparisons can be performed either using operator < (first version), or using a pre-defined function (second version). If more than one element satisfies the condition of being the largest, the iterator returned points to the first of such elements.

Syntax:
Template ForwardIterator max_element (ForwardIterator first, ForwardIterator last, Compare comp);
Time Complexity: O(n) For comparison based on a pre-defined function:

Refer to max_element.cpp

Find element in container: std::find()

std::find is a function defined inside <algorithm> header file that finds the element in the given range. It returns an iterator to the first occurrence of the specified element in the given sequence. If the element is not found, an iterator to the end is returned.

Syntax:

```
input_iterator std::find(input_iterator first, input_iterator last, const T& value);
```

Parameters:
first: iterator to the initial position in the sequence.
last: iterator to position just after the final position in the sequence. (Note that vector.end() points to the next position to the last element of the sequence and not to the last position of the sequence).
value: value to be searched.

Return Value :
If the value is found in the sequence, the iterator to its position is returned.
If the value is not found, the iterator to the last position is returned.

Time Complexity: O(n)
Auxiliary Space: O(1)

Refer to find.cpp

std::reserse()

std::reverse() is a predefined function in header file algorithm. It is defined as a template in the above-mentioned header file. It reverses the order of the elements in the range [first, last) of any container. The time complexity is O(n).

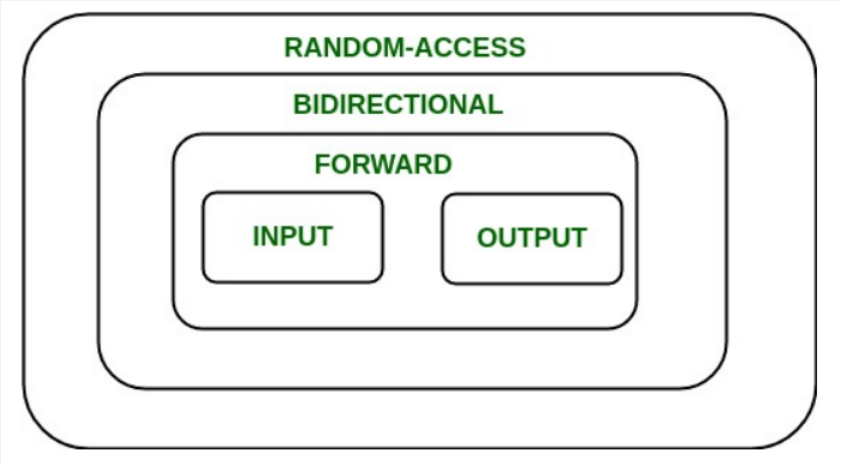
std::lower_bound()

The lower_bound() method in C++ is used to return an iterator pointing to the first element in the range [first, last) which has a value not less than val. This means that the function returns an iterator pointing to the next smallest number just greater than or equal to that number. If there are multiple values that are equal to val, lower_bound() returns the iterator of the first such value. The elements in the range shall already be sorted or at least partitioned with respect to val.

Standard Template Library Iterator

Iterators in C++

An iterator is an object (like a pointer) that points to an element inside the container. We can use iterators to move through the contents of the container. They can be visualised as something similar to a pointer pointing to some location and we can access the content at that particular location using them. Iterators play a critical role in connecting algorithm with containers along with the manipulation of data stored inside the containers. The most obvious form of an iterator is a pointer. A pointer can point to elements in an array and can iterate through them using the increment operator (++). But, all iterators do not have similar functionality as that of pointers. Depending upon the functionality of iterators they can be classified into five categories, as shown in the diagram below with the outer one being the most powerful one and consequently the inner one is the least powerful in terms of functionality.



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

Types of iterators

Based upon the functionality of the iterators, they can be classified into five major categories:

Input Iterators:

They are the weakest of all the iterators and have very limited functionality. They can only be used in a single-pass algorithms, i.e., those algorithms which process the container sequentially, such that no element is accessed more than once.

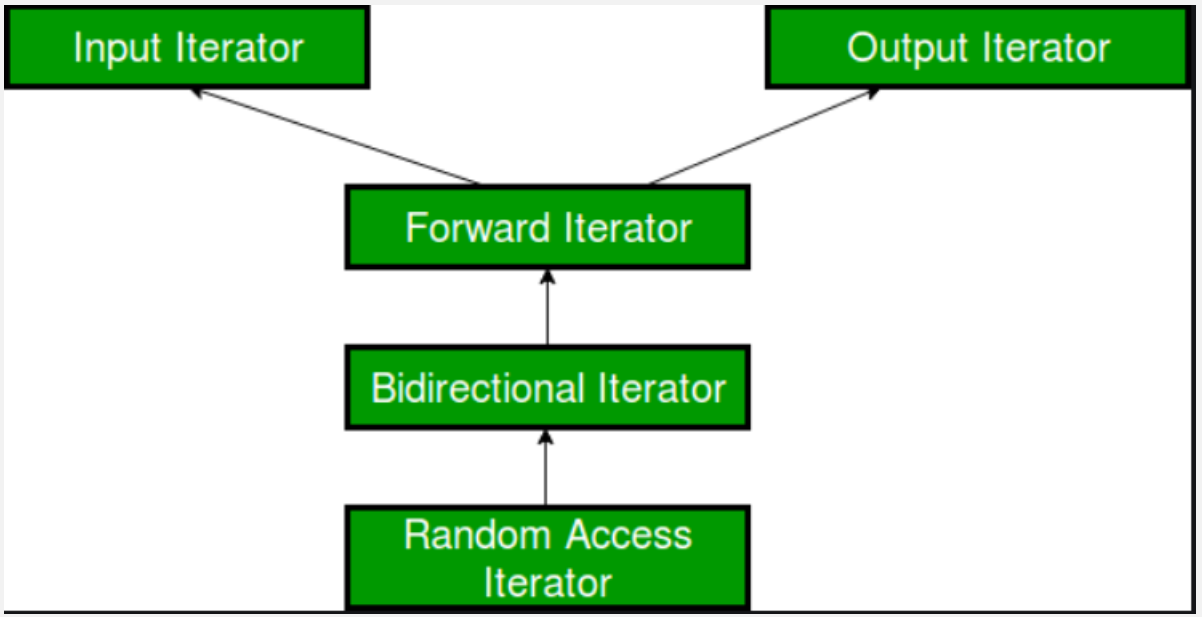
Output Iterators:

Just like input iterators, they are also very limited in their functionality and can only be used in single-pass algorithm, but not for accessing elements, but for being assigned elements.

Forward Iterator:

They are higher in the hierarchy than input and output iterators, and contain all the features present in these two iterators. But, as the name suggests, they also can only move in a forward direction and that too one step at a time.

Bidirectional Iterators:
They have all the features of forward iterators along with the fact that they overcome the drawback of forward iterators, as they can move in both the directions. This is why their name is bidirectional.
Random-Access Iterators:
They are the most powerful iterators. They are not limited to moving sequentially, as their name suggests, they can randomly access any element inside the container. They are the ones whose functionality are same as pointers.



Input Iterator

Input iterators are considered to be the weakest as well as the simplest among all the iterators available, based upon their functionality and what can be achieved using them. They are the iterators that can be used in sequential input operations, where each value pointed by the iterator is read-only once and then the iterator is incremented.

1. Usability: Input iterators can be used only with single-pass algorithms, i.e., algorithms in which we can go to all the locations in the range at most once, like when we have to search or find any element in the range, we go through the locations at most once.

2. Equality / Inequality Comparison: An input iterator can be compared for equality with another iterator. Since, iterators point to some location, so the two iterators will be equal only when they point to the same position, otherwise not. So, the following two expressions are valid if A and B are input iterators:
   ```
   A == B  // Checking for equality
   A != B  // Checking for inequality
   ```

3. Dereferencing: An input iterator can be dereferenced, using the operator * and -> as an rvalue to obtain the value stored at the position being pointed to by the iterator. So, the following two expressions are valid if A is an input iterator:
   ```
   *A      // Dereferencing using *
   A -> m  // Accessing a member element m
   ```

4. Incremental: An input iterator can be incremented, so that it refers to the next element in the sequence, using operator ++().
   ```
   A++   // Using post increment operator
   ++A   // Using pre increment operator
   ```

5. Swappable: The value pointed to by these iterators can be exchanged or swapped.

Refer to input_iterator.cpp

Limitations

1. Only accessing, no assigning: One of the biggest deficiency is that we cannot assign any value to the location pointed by this iterator, it can only be used to access elements and not assign elements.
   *i1 = 7;
   So, this is not allowed in the input iterator. However, if you try this for the above code, it will work, because vectors return iterators higher in the hierarchy than input iterators.That big deficiency is the reason why many algorithms like std::copy, which requires copying a range into another container cannot use input iterator for the resultant container, because we can't assign values to it with such iterators and instead make use of output iterators.

2. Cannot be decremented: Just like we can use operator ++() with input iterators for incrementing them, we cannot decrement them. If A is an input iterator, then
   A-- // Not allowed with input iterators

3. Use in multi-pass algorithms: Since it is unidirectional and can only move forward, therefore, such iterators cannot be used in multi-pass algorithms, in which we need to process the container multiple times.

4. Relational Operators: Although, input iterators can be used with equality operator (==), but it can not be used with other relational operators like <=. If A and B are input iterators, then
   A == B // Allowed
   A <= B // Not Allowed

5. Arithmetic Operators: Similar to relational operators, they also can't be used with arithmetic operators like +, – and so on. This means that input operators can only move in one direction that too forward and that too sequentially. If A and B are input iterators, then
   A + 1 // Not allowed
   B - 2 // Not allowed

Output Iterator

After going through the template definition of various STL algorithms like std::copy, std::move, std::transform, you must have found their template definition consisting of objects of type Output Iterator. Output iterators are considered to be the exact opposite of input iterators, as they perform the opposite function of input iterators. They can be assigned values in a sequence, but cannot be used to access values, unlike input iterators which do the reverse of accessing values and cannot be assigned values. So, we can say that input and output iterators are complementary to each other.

1. Usability: Just like input iterators, Output iterators can be used only with single-pass algorithms, i.e., algorithms in which we can go to all the locations in the range at most once, such that these locations can be dereferenced or assigned value only once.

2. Equality / Inequality Comparison: Unlike input iterators, output iterators cannot be compared for equality with another iterator. So, the following two expressions are invalid if A and B are output iterators:
   A == B // Invalid - Checking for equality
   A != B // Invalid - Checking for inequality

3. Dereferencing: An input iterator can be dereferenced as an rvalue, using operator * and ->, whereas an output iterator can be dereferenced as an lvalue to provide the location to store the value. So, the following two expressions are valid if A is an output iterator:
   *A = 1   // Dereferencing using *
   A -> m = 7   // Assigning a member element m

4. Incrementable: An output iterator can be incremented, so that it refers to the next element in sequence, using operator ++(). So, the following two expressions are valid if A is an output iterator:
   A++   // Using post increment operator
   ++A   // Using pre increment operator

5. Swappable: The value pointed to by these iterators can be exchanged or swapped.

Refer to Example of std::move()

std::move() is a utility function in C++ that indicates that the object should be "moved from" rather than copied from. It is often used in the context of move semantics, which allows for more efficient transfers of resources (like memory ownership) between objects

Template OutputIterator move (InputIterator first, InputIterator last, OutputIterator result);

Parameters :
the first, last Input iterators to the initial and final positions in a sequence to be moved. The range used is [first,last], which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.

Result output iterator to the initial position in the destination sequence. This shall not point to any element in the range [first,last].

Return type :
An iterator to the end of the destination range where elements have been moved

Forward Iterators in C++

Forward iterators are considered to be the combination of input as well as output iterators. It provides support to the functionality of both of them. It permits values to be both accessed and modified.

1. Usability: Performing operations on a forward iterator that is dereferenceable never makes its iterator value non-dereferenceable, as a result this enables algorithms that use this category of iterators to use multiple copies of an iterator to pass more than once by the same iterator values. So, it can be used in multi-pass algorithms.

2. Equality / Inequality Comparison: A forward iterator can be compared for equality with another iterator. Since, iterators point to some location, so the two iterators will be equal only when they point to the same position, otherwise not. So, the following two expressions are valid if A and B are forward iterators:
   A == B // Checking for equality
   A != B // Checking for inequality

3. Dereferencing: Because an input iterator can be dereferenced, using the operator * and -> as an rvalue and an output iterator can be dereferenced as an lvalue, so forward iterators can be used for both the purposes.

4. Incremental: A forward iterator can be incremented, so that it refers to the next element in sequence, using operator ++(). Note: The fact that we can use forward iterators with increment operator doesn't mean that operator – -() can also be used with them. Remember, that forward iterators are unidirectional and can only move in the forward direction. So, the following two expressions are valid if A is a forward iterator:
   A++   // Using post increment operator
   ++A   // Using pre increment operator

5. Swappable: The value pointed to by these iterators can be exchanged or swapped.

Limitations

After studying the salient features, one must also know its deficiencies as well although there are not as many as there are in input or output iterators as it is higher in the hierarchy.

1. Can not be decremented: Just like we can use operator ++() with forward iterators for incrementing them, we cannot decrement them. Although it is higher in the hierarchy than input and output iterators, still it can't overcome this deficiency. That is why, its name is forward, which shows that it can move only in forward direction.

   ```
   If A is a forward iterator, then
   A--    // Not allowed with forward iterators
   ```

2. Relational Operators: Although, forward iterators can be used with equality operator (==), it can not be used with other relational operators like =.

   ```
   If A and B are forward iterators, then
   A == B     // Allowed
   A <= B     // Not Allowed
   ```

3. Arithmetic Operators: Similar to relational operators, they also can't be used with arithmetic operators like +, – and so on. This means that forward operators can only move in one direction that too forward and that too sequentially.

   ```
   If A and B are forward iterators, then
   A + 1     // Not allowed
   B - 2     // Not allowed
   ```

4. Use of offset dereference operator ([ ]): Forward iterators do not support offset dereference operator ([ ]), which is used for random-access.

   ```
   If A is a forward iterator, then
   A[3]    // Not allowed
   ```

Refer to std::lower_bound() example of forward iterator

The lower_bound() method in C++ is used to return an iterator pointing to the first element in the range [first, last) which has a value not less than val. This means that the function returns an iterator pointing to the next smallest number just greater than or equal to that number. If there are multiple values that are equal to val, lower_bound() returns the iterator of the first such value. The elements in the range shall already be sorted or at least partitioned with respect to val.
Templates:
Syntax 1:
ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last, const T& val);
Syntax 2:
ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last, const T& val, Compare comp);

Parameters:
The above methods accept the following parameters.
first, last: The range used is [first, last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.
val: Value of the lower bound to be searched for in the range.
comp: Binary function that accepts two arguments (the first of the type pointed by ForwardIterator, and the second, always val), and returns a value convertible to bool. The function shall not modify any of its arguments. This can either be a function pointer or a function object.

Return Value:
An iterator to the lower bound of val in the range. If all the elements in the range compare less than val, the function returns last. If all the elements in the range are larger than val, the function returns a pointer to the first element.

Why Use Iterators?

Here are some reasons you might want to use iterators in C++:

- Support for algorithms: The C++ Standard Library provides a wide range of algorithms that can be used with iterators, such as std::find(), std::sort(), and std::accumulate(). These algorithms allow us to perform common operations on containers.
- Memory efficiency: Iterators allow us to process large datasets one element at a time without loading the entire dataset at once, resulting in memory efficiency.
- Consistency: Iterators provide a way to access and manipulate data of different types of containers in a consistent way. We can use the same iterator syntax for vectors, lists, sets, or any other container.
- Simplification of code: Iterators can simplify the code by hiding the details of iterating over the container, thus making the code more readable.

Quiz

Question 1:
Which STL container is implemented as a doubly-linked list?

a) std::vector
b) std::list
c) std::deque
d) std::set

Question 2:
What is the purpose of the std::algorithm library in C++ STL?

a) String manipulation
b) Input and output operations
c) Container algorithms
d) Memory management

Question 3:
What is the difference between std::map and std::unordered_map?

a) std::map is sorted; std::unordered_map is not
b) std::map uses a hash function; std::unordered_map uses a comparator
c) std::map allows duplicate keys; std::unordered_map does not
d) There is no difference; they are the same container

Question 4:

Which STL algorithm is used to sort elements in a range?

a) std::sort
b) std::find
c) std::search
d) std::merge

Question 5:
What is the purpose of the std::pair template class?

a) Representing a sequence of elements
b) Associating two values together
c) Managing dynamic memory allocation
d) Sorting elements in a container

Question 6:
Which STL container provides constant time complexity for insertion and deletion of elements at the beginning and end?

a) std::vector
b) std::list
c) std::deque
d) std::queue

Question 7:
What is the purpose of the std::unique algorithm?

a) Removing consecutive duplicate elements in a range

b) Finding the first occurrence of a value in a range
c) Merging two sorted ranges
d) Reversing the order of elements in a range

Question 8:
Which STL container is typically implemented as a binary heap?

a) std::vector
b) std::list
c) std::queue
d) std::stack