C++ STL

STL Containers:

A container class is a data type that is capable of holding a collection of items. In C++, container classes can be implemented as a class, along with member functions to add, remove, and examine items.

Types:

1. **Sequence containers:** vector, deque, array, list, forward_list, basic_string
2. **Associative containers:** set, multiset, map, multimap
3. **Container adapters:** stack, queue, priority_queue

Sequence Containers:

Sequence containers are container classes that maintain the ordering of elements in the container. The most common example of a sequence container is the array: if you insert four elements into an array, the elements will be in the exact order you inserted them.

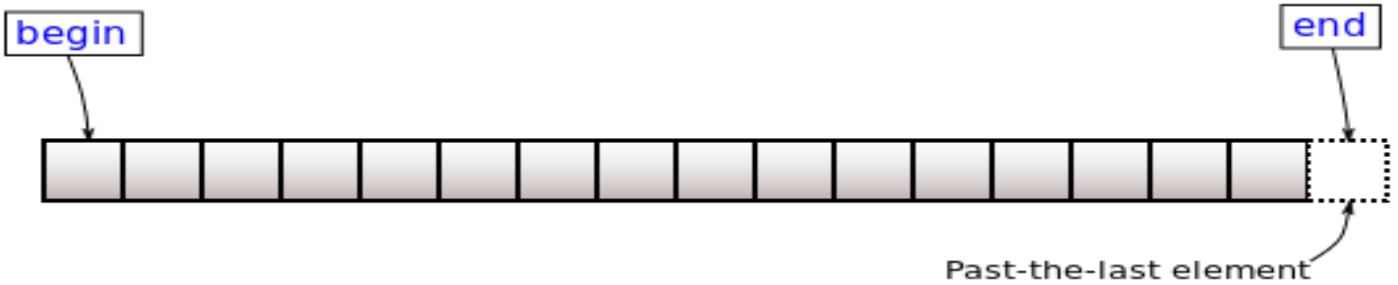
Vector

- Dynamic array
- Stored contiguously
- Elements are accessible using [ ] or iterators

Time complexity:

- Random access - constant O(1)
- Insertion or removal of elements at the end - amortized constant O(1)
- Insertion or removal of elements - linear in the distance to the end of the vector O(n)

Iterators



begin( ): Returns an iterator to the first element of the container. If the container is empty, the returned iterator will be equal to end( ).

end( ): Returns an iterator to the element following the last element of the container.

Some member functions:

- operator=
- operator[ ]
- front( )
- back( )
- begin( ), rbegin ()
- end( ), rend( )
- empty( )
- size( ), capacity( )
- clear( )
- erase( )
- push_back( ), pop_back( )
- resize( )

List

- A list is a special type of sequence container called a doubly linked list where each element in the container contains pointers that point at the next and previous elements in the list.
- Lists only provide access to the start and end of the list -- there is no random access provided
- The advantage of lists is that inserting elements into a list is very fast if you already know where you want to insert them.

deque

- The deque class (pronounced “deck”) is a double-ended queue class, implemented as a dynamic array that can grow from both ends.

Associative containers:

In standard template libraries they refer to the group of class templates used to implement **associative arrays.**

In associative containers, most complexities are in logarithmic terms

- Inserting an element is O(log n)
- Removing a element O(log n)
- Searching for an element O(log n)
- Incrementing or decrementing iterator O(1)(amortized)
- Insertion in middle is faster.

Associative containers:

1. Set:
2. Multiset:
3. Map:
4. Multimap:

Set

- Sets are a type of associative containers in which each element has to be unique, because the value of the element identifies it.
- The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

Some basic functions :

- begin() – Returns an iterator to the first element in the set.
- end() – Returns an iterator to the theoretical element that follows last element in the set.
- size() – Returns the number of elements in the set.
- max_size() – Returns the maximum number of elements that the set can hold.
- empty() – Returns whether the set is empty.
- erase(iterator position) – Removes the element at the position pointed by the iterator
- erase(const g)– Removes the key value ‘g’ from the map

Multiset

- Multisets are a type of associative containers similar to set, with an exception that multiple elements can have same values.

Some basic functions :

- begin() – Returns an iterator to the first element in the multiset
- end() – Returns an iterator to the theoretical element that follows last element in the multiset
- size() – Returns the number of elements in the multiset
- max_size() – Returns the maximum number of elements that the multiset can hold
- empty() – Returns whether the multiset is empty
- erase(iterator position) – Removes the element at the position pointed by the iterator
- erase(const g)– Removes the key value ‘g’ from the map

Map

- Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value.

Some basic functions associated with Map:

- begin() – Returns an iterator to the first element in the map
- end() – Returns an iterator to the theoretical element that follows last element in the map
- size() – Returns the number of elements in the map
- max_size() – Returns the maximum number of elements that the map can hold
- empty() – Returns whether the map is empty
- pair insert(keyvalue, mapvalue) – Adds a new element to the map
- erase(iterator position) – Removes the element at the position pointed by the iterator
- erase(const g)– Removes the key value ‘g’ from the map
- clear() – Removes all the elements from the map

Multimap

- Multimap is similar to map with an addition that multiple elements can have same keys. Rather than each element being unique, the key value and mapped value pair has to be unique in this case.

Some basic functions associated with Multimap:

- begin() – Returns an iterator to the first element in the multimap
- end() – Returns an iterator to the theoretical element that follows last element in the multimap
- size() – Returns the number of elements in the multimap
- max_size() – Returns the maximum number of elements that the multimap can hold
- empty() – Returns whether the multimap is empty
- pair<int,int> insert(keyvalue,multimapvalue) – Adds a new element to the multimap

Adaptive containers:

1. Queue
2. Stack
3. Priority_queue

Queue

- Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front.

Some basic functions associated with queue:

- empty() – Returns whether the queue is empty.
- size() – Returns the size of the queue.
- queue::front() – front() function returns a reference to the first element of the queue.
- queue::back()– back() function returns a reference to the last element of the queue.
- push(g) – push() function adds the element ‘g’ at the end of the queue.
- pop() – pop() function deletes the first element of the queue.
- queue::swap()- Exchange the contents of two queues but the queues must be of same type, although sizes may differ.

Stack

- Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end and (top) an element is removed from that end only.

Some basic functions associated with stack:

- empty() – Returns whether the stack is empty – Time Complexity : O(1)
- size() – Returns the size of the stack – Time Complexity : O(1)
- top() – Returns a reference to the top most element of the stack – Time Complexity : O(1)
- push(g) – Adds the element ‘g’ at the top of the stack – Time Complexity : O(1)
- pop() – Deletes the top most element of the stack – Time Complexity : O(1)

STL Algorithms

Some of the most used algorithms on vectors and most useful one's in Competitive Programming are mentioned as follows :

ALGORITHMS ON VECTORS

1. **sort(first_iterator, last_iterator)** – To sort the given vector.
2. **reverse(first_iterator, last_iterator)** – To reverse a vector.
3. ***max_element (first_iterator, last_iterator)** – To find the maximum element of a vector.
4. ***min_element (first_iterator, last_iterator)** – To find the minimum element of a vector.
5. **accumulate(first_iterator, last_iterator, initial value of sum)** – Does the summation of vector elements.

6. **count(first_iterator, last_iterator,x)** – To count the occurrences of x in vector.

7. **find(first_iterator, last_iterator, x)** – Points to last address of vector ((name_of_vector).end()) if element is not present in vector.

8. **binary_search(first_iterator, last_iterator, x)** – Tests whether x exists in sorted vector or not.

9. **lower_bound(first_iterator, last_iterator, x)** – returns an iterator pointing to the first element in the range [first,last) which has a value not less than ‘x’.

10. **upper_bound(first_iterator, last_iterator, x)** – returns an iterator pointing to the first element in the range [first,last) which has a value greater than ‘x’.

11. **vector_name.erase(position to be deleted)** – This erases selected element in vector and shifts and resizes the vector elements accordingly.

12. **vector_name.erase(unique(arr.begin(),arr.end()),arr.end())** – This erases the duplicate occurrences in sorted vector in a single line.

13. **next_permutation(first_iterator, last_iterator)** – This modified the vector to its next permutation.

14. **prev_permutation(first_iterator, last_iterator)** – This modified the vector to its previous permutation.

15. **distance(first_iterator,desired_position)** – It returns the distance of desired position from the first iterator.This function is very useful while finding the index.

MERGE OPERATIONS

1. **merge(beg1, end1, beg2, end2, beg3)** :- This function merges two sorted containers and stores in new container in sorted order (merge sort). It takes 5 arguments, first and last iterator of 1st container, first and last iterator of 2nd container and 1st iterator of resultant container.

1. **set_union(beg1, end1, beg2, end2, beg3)** :- This function computes the set union of two containers and stores in new container .It returns the iterator to the last element of resultant container. It takes 5 arguments, first and last iterator of 1st container, first and last iterator of 2nd container and 1st iterator of resultant container . The containers should be sorted and it is necessary that new container is resized to suitable size.

4. **set_intersection(beg1, end1, beg2, end2, beg3)** :- This function computes the set intersection of two containers and stores in new container .It returns the iterator to the last element of resultant container. It takes 5 arguments, first and last iterator of 1st container, first and last iterator of 2nd container and 1st iterator of resultant container . The containers should be sorted and it is necessary that new container is resized to suitable size.

5. **set_difference(beg1, end1, beg2, end2, beg3)** :- This function computes the set difference of two containers and stores in new container .It returns the iterator to the last element of resultant container. It takes 5 arguments, first and last iterator of 1st container, first and last iterator of 2nd container and 1st iterator of resultant container . The containers should be sorted and it is necessary that new container is resized to suitable size.

OTHER USEFUL ALGORITHMS

1. **fill(first_iter,last_iter,value) :** The ‘fill’ function assigns the value ‘val’ to all the elements in the range [begin, end), where ‘begin’ is the initial position and ‘end’ is the last position. **NOTE :** Notice carefully that ‘begin’ is included in the range but ‘end’ is NOT included.
2. **unique (first_iter,last_iter) :** It is used to remove duplicates of any element present consecutively in a range[first, last).
3. **is_partitioned(first_iter,last_iter,function): It** is used for finding whether the range[first, last) is partitioned or not. A range is said to be partitioned with respect to a condition if all the elements for which the condition evaluates to true precede those for which it is false.

References:

- https://en.cppreference.com/
- http://www.yolinux.com/
- https://www.learncpp.com/
- https://www.geeksforgeeks.org/c-magicians-stl-algorithms/
- https://www.geeksforgeeks.org/algorithms-library-c-stl/
- https://www.cs.helsinki.fi/u/tpkarkka/alglib/k06/lectures/algorithms.html