A summary sheet for all the containers mentioned in **Advanced C++ for Robotics Unit 2: The STL Library** provided by ***theConstruct***. Snapshots from the course to go along with demo code.

- Composed by Donovan Gegg

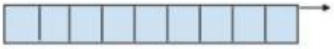
---

## Sequential Containers

### Array

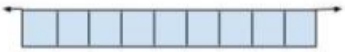
An array is a collection of similar data elements stored at contiguous memory locations.

### Vector



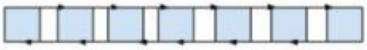
A vector is a sequence container representing arrays that can change their size during runtime

### Deque



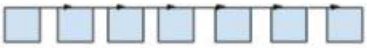
Double-ended queues are sequence containers with the feature of expansion and contraction on both ends

### List



Lists are sequence containers that allow non-contiguous memory allocation.

### Forward List

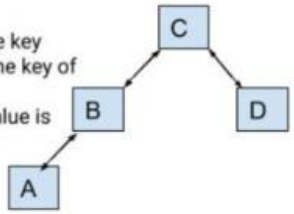


Forward list differs from the list by the fact that the forward list keeps track of the location of only the next element while the list keeps track of both the next and previous elements

## Associative Containers

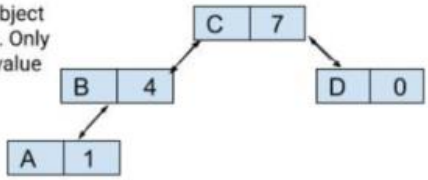
### Set

Stores only the key object. Only one key of each value allowed(the value is itself the *key*).



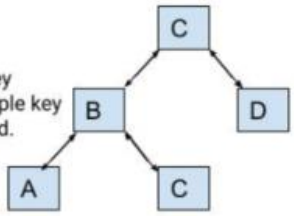

### Map

Associates key object with value object. Only one key of each value allowed.



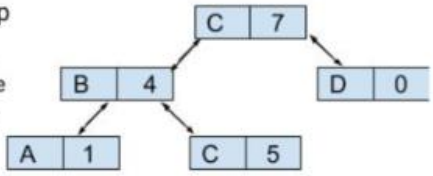

### Multi Set

Stores only key objects. Multiple key values allowed.




### Multi Map

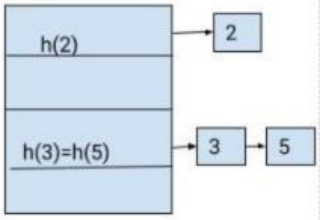
Associates key object with value object. Multiple key values allowed.




## Unordered Associative containers

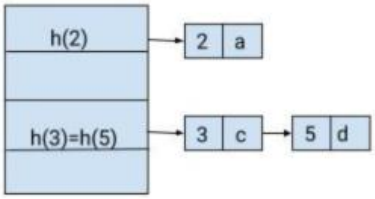
### Unordered Set

A container in which elements are unsorted and have distinct values



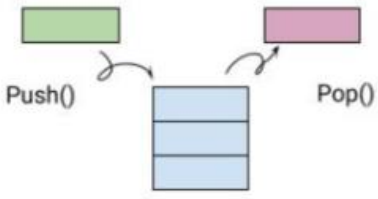

### Unordered Map

A container in which elements are unsorted key-value pair. Each key may occur only once.



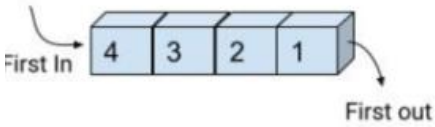
# Container Adapters

## Stack




A stack is a linear data structure in c++. It serves as a collection of data arranged serially in a specific order. It follows LIFO i.e. Last In First Out. You can think up of a stack of plates where you are allowed to add a plate or remove a plate only from the top. In order to add a plate you use push() and to remove you use pop().

## Queue




A queue is a type of container which operates in a FIFO i.e. First In First Out type of arrangement. You can understand and remember this container as a queue on the food counter. One who arrives first collects food first and leaves.

## Priority Queue

```
#include <queue>
using namespace std;
int main()
{
int sum = 0;
priority_queue<int> pque;
pque.push(10);
pque.push(20);
pque.push(345);
pque.push(312);
pque.push(309);
while(!pque.empty()){
cout<<"  "<<pque.top();
pque.pop();
}
return 0;
}
```

Input order : 10 20 345 312 309

Output order

```
345  312  309  20  10
```

A priority queue in c++ is a type of container adapter, which processes only the highest priority element, i.e. the first element will be the maximum of all elements in the queue, and elements are in decreasing order. When pop() operation is performed, element with highest value is popped first.

## Lists:

Important function supported by list:

- `swap()` - swaps elements of two list of same data type.
- `reverse()` - reverses a list completely
- `sort()` - arranges integers in the list in ascending order
- `sort(compare_function)` - Sorting a list as per user-defined compare_function().
- `merge()` - Merges two sorted lists in sorted order.

## Forward Lists:

Important functions supported by a forward list:

- `assign()` - Used to assign elements in the forward list.
- `push_front()` - Inserts the element at the first position of a forward list. The value from this function is copied to the space before the first element in the container. The size of the forward list increases by 1.
- `emplace_front()` - similar to the previous function but in this no copying operation occurs, the element is created directly at the memory before the first element of the forward list.
- `pop_front()` - This function is used to delete the first element of list.
- `insert_after()` - This function gives us a choice to insert elements at any position in the forward list. The arguments in this function are copied at the desired position.
- `emplace_after()` -This function also does the same operation as above function but the elements are directly made without any copy operation.
- `erase_after()` - This function is used to erase elements from a particular position in the forward list.
- `remove()` - This function removes the particular element from the forward list mentioned in its argument.
- `remove_if()` - This function removes according to the condition in its argument.
- `splice_after()` - This function transfers elements from one forward list to other.

## Sets:

Functions supported by a set:

- `insert(k)` : inserts k as per the ascending/descending order which is mentioned at the time of declaring the set.
- `begin()` : returns an iterator to the first element of the set.
- `end()` : returns an iterator to the *past-the-end* element of the set. The *past-the-end* element is the theoretical element that would follow the last element in the set container.
- `erase(k)` : erases element with value k.
- `lower_bound(k)` : returns iterator to the nth element which is equal to k(if present) or to the element which is right next to the kth element if existed in the set.
- `upper_bound(k)` : return the iterator to the element right next to the element(which is equal to k even if k is present in the set).

## Multisets:

Functions supported by multiset:

- `insert(k)` : inserts k as per the ascending/descending order which is mentioned at the time of declaring the multiset.
- `begin()` : returns an iterator to the first element of the multiset.
- `end()` : returns an iterator to the *past-the-end* element of the multiset. The *past-the-end* element is the theoretical element that would follow the last element in the set container.
- `erase(k)` : erases all the keys with value k(recall here key==value)
- `lower_bound(k)` : returns iterator to the nth element which is equal to k(if present) or to the element which is right next to the kth element if existed in the multiset.
- `upper_bound(k)` : return the iterator to the element right next to the element(which is equal to k even if k is present in the multiset).

## Maps:

Functions supported by the Map:

- `begin()` – Returns an iterator to the first element in the map.
- `end()` – Returns an iterator to the theoretical element that follows the last element in the map.
- `size()` – Returns the number of key-value pairs on the map.
- `max_size()` – Returns the maximum number of pairs that the map can hold.
- `empty()` – Returns 1 when map is empty.
- `pair insert(key-value, map value)` – Adds a new element to the map.
- `erase(iterator position)` – Removes the element at the position pointed by the iterator.
- `erase(const k)` – Removes the key k along with its value from the map.
- `clear()` – Removes all the elements from the map.

## Unordered Maps:

Function Supported:

- `at()` : Shows elements at that location.
- `begin()` : Returns an iterator pointing to the first element in the container.
- `end()` : Returns an iterator pointing to the position past the last element in the container in the unordered_map container
- `bucket()` : Returns the bucket number where the element with the key k is located in the map.
- `bucket_count` : bucket_count is used to count the total no. of buckets in the unordered_map.
- `bucket_size` : Returns the number of elements in each bucket of the unordered_map.
- `count()` : Count the number of elements present in an unordered_map with a given key.
- `equal_range` : Return the bounds of a range that includes all the elements in the container with a key that compares equal to k.

## Iterators:

Functions supported by iterators:

- `advance(k)` : advance iterator by k positions
- `distance()` : distance between two positions
- `next(iterator, n)` : returns the value of element n times on succession with iterator's current position
- `prev(iterator, n)` : returns the value of element n times on previous with iterator's current position
- `begin()` : return first element of the container
- `end()` : returns the last of the container

## Syntax:

### Array

```
data_type array_name[size_of_array]={element1,element2};
```

### Vector

```
std::vector<data_type> vector_name;
```

### Deque

```
deque<data_type> deque_name;
```

### List

```
list<data_type> list_name;
```

### Forward List

```
forward_list<data_type> name;
```

## Set

```
set<datatype> setname;
```

## MultiSet

```
multiset<datatype> name_of_multiset;
```

## Maps

```
map <data_type_of_key, data_type_of_value> name_of_map;
```

## Unordered Maps

```
unordered_map<datatype_key, datatype_value> name_of_unordered_map;
```

## Iterators

```
container_type <parameter_list>::iterator iterator_name;
```

## Terminal Outputs for Demos

### Demo 1

```
100 80
```

### Demo 2

```
vector1 = 1  2  3  4  5
vector2 = 6  7  8  9  10
vector3 = 12  12  12  12  12
Edited vector1 = 1  2  3  4  5  59
2nd element of vector 2: 7
Edited vector3 = 12  12  12  12
```

### Demo 3

```
The deque is :  d   b   a   c

M.at(2) : a
M.front() : d
M.back() : c
After removing front element using M.pop_front() :  b   a   c

After removing last element using M.pop_back() :    b   a
```

### Demo 4

```
2 3 5
```

**Demo 5**

```
user:~/stl_lib$ ./l1
1 2 3 4 8
5 7 9
9 7 5
5 7 9
1 2 3 4 5 7 8 9
```

**Demo 6**

```
user:~/stl_lib$ ./fl
The elements of first forward list are : 10 22 31
The elements of second forward list are : 10 10 10 10 10
```

**Demo 7**

```
The forward list : 10 20 30 40 50
The forward list after push_front operation : 60 10 20 30 40 50
The forward list after emplace_front operation : 70 60 10 20 30 40 50
The forward list after pop_front operation : 60 10 20 30 40 50
The forward list after insert_after operation : 60 1 2 3 10 20 30 40 50
The forward list after emplace_after operation : 60 1 2 3 2 10 20 30 40 50
The forward list after erase_after operation : 60 1 2 3 2 20 30 40 50
The forward list after remove operation : 60 1 2 3 2 20 30 50
The forward list after remove_if operation : 1 2 3 2 20
```

**Demo 8**

```
user:~/stl_lib$ ./set1
The element of set s are :
B O R T
The size of set :
 4
```

**Demo 9**

```
user:~/stl_lib$ ./setfunction

The set s1 is :
510 180 100 80 1 0

The set s2 after assign from s1 is :
0 1 80 100 180 510

s2 after removal of elements less than 100 :
100 180 510

After removal of 510 from s2
100 180

s3.lower_bound(80) :
80
s3.upper_bound(80) :
100
```

**Demo 10**

```
user:~/stl_lib$ ./ms
The element of set s are :
B O O R T
The size of set :
 5
```

**Demo 11**

```
user:~/stl_lib$ ./msf

The multiset s1 is :
510 510 180 100 100 80 1 0

The multiset s2 after assign from s1 is :
0 1 80 100 100 180 510 510

s2 after removal of elements less than 100 :
100 100 180 510 510

After removal of 510 from s2
100 100 180

s3.lower_bound(80) :
80
s3.upper_bound(80) :
100
```

**Demo 12**

```
Hot chocolate cake, Cheese Cake
```

**Demo 13**

```
The map function_test is :
        KEY     ELEMENT
        1       M
        2       O
        3       V
        4       E


The map copy_function after assign from function_test is :
        KEY     ELEMENT
        1       M
        2       O
        3       V
        4       E


copy_function after removal of elements less than key=2 :
        KEY     ELEMENT
        2       O
        3       V
        4       E

function_test.lower_bound(3) :  KEY = 3         ELEMENT = V
function_test.upper_bound(3) :  KEY = 4         ELEMENT = E
```

**Demo 14**

```
user:~/stl_lib$ g++ unmap.cpp -o umap
user:~/stl_lib$ ./umap
Implemention 0.2
Map 0.1
Unordered 0
```

**Demo 15**

```
user:~/stl_lib$ ./umf1
Found four

Found one

All Elements :
six 6.66
five 5.55555
four 4.4444
three 3.333
two 2.22
one 1.1
Total no. of buckets/bunches in which key are organised: 13
```

**Demo 16**

```
1 2 3 4 4 1 6 7
```

**Demo 17**

```
user:~/stl_lib$ ./it3
The vector we have : 1 2 3 4 4 6 7
Begin() : 1
Advance by 5 : 6
Move back by 1 : 4
Distance between begin and end : 7
4 elements next to i's current position(which is begin here) : 4
4 elements previous to i's current position(which is at 6 here) : 2
```