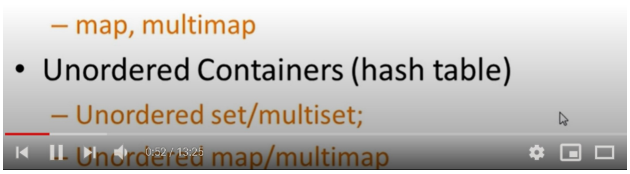
Containers

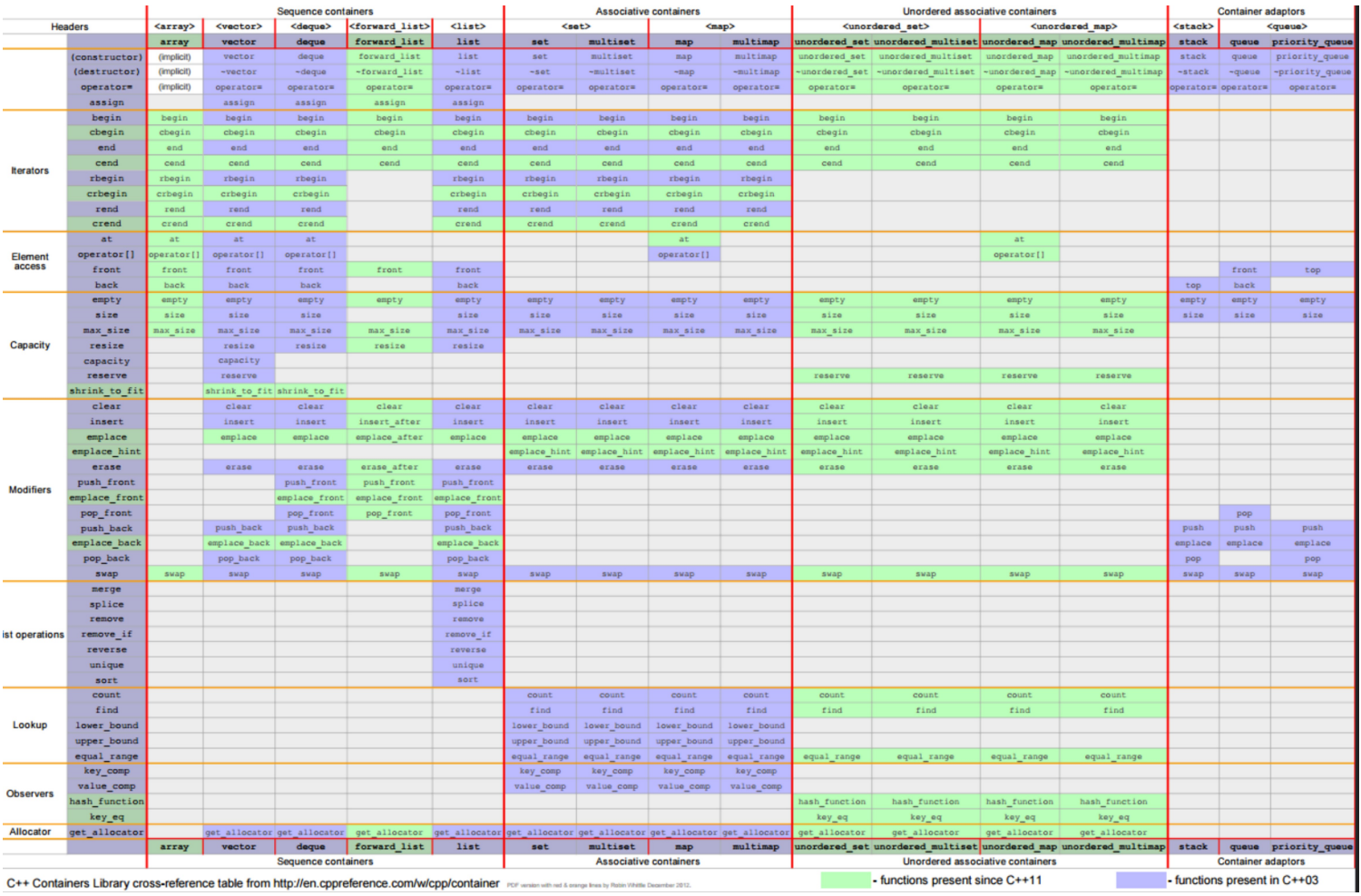
09 June 2021 07:07

Containers

- Sequence containers (array and linked list):
  - vector, deque, list, forward list, array
- Associative Containers (binary tree)
  - set, multiset,
  - map, multimap
- Unordered Containers (hash table)
  - Unordered set/multiset;
  - Unordered map/multimap



```
#include <vector>
#include <deque>
#include <list>
#include <set>    // set and multiset
#include <map>    // map and multimap
#include <unordered_set>  // unordered set/multiset
#include <unordered_map>  // unordered map/multimap
#include <iterator>
#include <algorithm>
#include <numeric>    // some numeric algorithm
#include <functional>
```



		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		
Headers		<array>	<vector>	<deque>	<forward_list>	<list>	<set>		<map>		<unordered_set>		<unordered_map>		<stack>	<queue>	
		array	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(constructor)	(implicit)	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(destructor)	(implicit)	~vector	~deque	~forward_list	~list	~set	~multiset	~map	~multimap	~unordered_set	~unordered_multiset	~unordered_map	~unordered_multimap	~stack	~queue	~priority_queue
	operator=	(implicit)	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=
	assign		assign	assign	assign	assign											
Iterators	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin			
	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin			
	end	end	end	end	end	end	end	end	end	end	end	end	end	end			
	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend			
	rbegin	rbegin	rbegin	rbegin		rbegin	rbegin	rbegin	rbegin	rbegin							
	crbegin	crbegin	crbegin	crbegin		crbegin	crbegin	crbegin	crbegin	crbegin							
	rend	rend	rend	rend		rend	rend	rend	rend	rend							
	crend	crend	crend	crend		crend	crend	crend	crend	crend							
Element access	at	at	at	at					at				at				
	operator[]	operator[]	operator[]	operator[]					operator[]				operator[]				
	front	front	front	front	front	front										front	top
	back	back	back	back		back									top	back	
Capacity	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty
	size	size	size	size		size	size	size	size	size	size	size	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size			
	resize		resize	resize	resize	resize											
	capacity		capacity														
	reserve		reserve								reserve	reserve	reserve	reserve			
	shrink_to_fit		shrink_to_fit	shrink_to_fit													
Modifiers	clear		clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear			
	insert		insert	insert	insert_after	insert	insert	insert	insert	insert	insert	insert	insert	insert			
	emplace		emplace	emplace	emplace_after	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace			
	emplace_hint						emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint			
	erase		erase	erase	erase_after	erase	erase	erase	erase	erase	erase	erase	erase	erase			
	push_front			push_front	push_front	push_front											
	emplace_front			emplace_front	emplace_front	emplace_front											
	pop_front			pop_front	pop_front	pop_front										pop	
	push_back		push_back	push_back		push_back									push	push	push
	emplace_back		emplace_back	emplace_back		emplace_back									emplace	emplace	emplace
	pop_back		pop_back	pop_back		pop_back									pop		pop
	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap
ist operations	merge					merge											
	splice					splice											
	remove					remove											
	remove_if					remove_if											
	reverse					reverse											
	unique					unique											
	sort					sort											
Lookup	count						count	count	count	count	count	count	count	count			
	find						find	find	find	find	find	find	find	find			
	lower_bound						lower_bound	lower_bound	lower_bound	lower_bound							
	upper_bound						upper_bound	upper_bound	upper_bound	upper_bound							
	equal_range						equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range			
Observers	key_comp						key_comp	key_comp	key_comp	key_comp							
	value_comp						value_comp	value_comp	value_comp	value_comp							
	hash_function										hash_function	hash_function	hash_function	hash_function			
	key_eq										key_eq	key_eq	key_eq	key_eq			
Allocator	get_allocator		get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator			
		array	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		

C++ Containers Library cross-reference table from http://en.cppreference.com/w/cpp/container PDF version with red & orange lines by Robin Whittle December 2012. - functions present since C++11 - functions present in C++03

1. Performance

	Sort		
vector		• Dynamic array – one direction • random access • Contiguous memory - easy access if u know location • Increment iterator/pointer to access elements	

		• SLOW search • SLOW Insertion in between O(n)	
deque		• Dynamic array - Grows both sides • Random access • Partially contiguous memory • Cannot increment iterator/pointer to access all elements • SLOW search • SLOW Insertion in between	
list		• Dynamic size in non-contiguous memory • bidirectional iterators – no random access • FAST Insert/Remove in any location • Cannot access via pointer/iterator increment or decrement • SLOW search • SPLICING - inserting another list, element or elements of another list at any position of first list • Dedicated SORT and other special functions – merge, sort, reverse, remove,	
forward_list		• Insert/Remove fast in any location • Forward SPLICING • SLOW search • No Random Access	
set	Auto	• FAST search - Find is very fast since sorted O(log n) - tree - Has dedicated find function • FAST insertion O(logn) ---> hint makes insertion O(1) • members cannot be modified.. *itr is const.. Members only erased	
map	Auto	• FAST search - Find very fast O(log n) - Has dedicated find function • FAST insertion O(logn) ----> hint can be given for O(1) • keys cannot be modified.. Only erased	
Unordered set		• Insert is very very fast O(1) • Find is very very very fast O(1) O(1) - hash collision - many items in same bucket O(n) • Rehash, Load_factor - ratio of total elements/total bucket, bucket - which bucket, bucket_count • Members cannot be modified • Forward iterator only • Has dedicated find function • Reserve(n) -> set/change number of buckets	
Unordered map		• Find is very very very fast O(1) - hash collision - many items in same bucket O(n) • Rehash, Load_factor - ratio of total elements/total bucket, bucket - which bucket, bucket_count • keys cannot be modified.. Only erased • Forward iterator only • Has dedicated find function • Reserve(n) -> set/change number of buckets	

Characteristic	std::array	std::vector	std::deque	std::list	std::forward_list
Size	static	dynamic	dynamic	dynamic	dynamic
Implementation	static array	dynamic array	sequence of arrays	double linked list	single linked list
Access	random access	random access	random access	forward and backward	forward
Optimized for		end O(1)	begin and end O(1)	▪ begin and end O(1) ▪ all over O(1)	▪ begin O(1) ▪ all over (1)
Memory Reservation		yes	no	yes	no
Memory Release		shrink_to_fit()	▪ sometimes ▪ shrink_to_fit()	always	always
Strength	▪ no memory allocation ▪ minimale memory requirements	95% solution	Insert and delete at the begin and end	Insert and delete at each position	▪ fast insertion and deletion ▪ minimalememory requirements
Weakness	no dynamic memory allocation	Insertion and deletion at arbitrary positions O(n)	Insertion and deletion at arbitrary positions O(n)	no random access	no random access

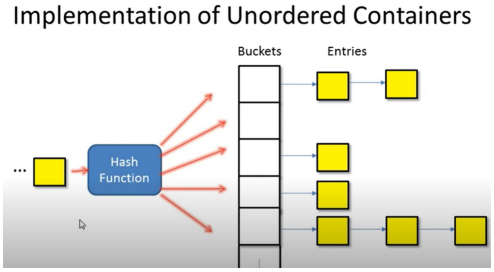
Container	Insertion	Access	Erase	Find	Persistent Iterators
vector / string	Back: O(1) or O(n) Other: O(n)	O(1)	Back: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	No
deque	Back/Front: O(1) Other: O(n)	O(1)	Back/Front: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	Pointers only
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)	Yes
set / map	O(log n)	-	O(log n)	O(log n)	Yes
unordered_set / unordered_map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	Pointers only
priority_queue	O(log n)	O(1)	O(log n)	-	-

590 × 346

Collection	Ordering	Contiguous Storage?	Direct Access?	Lookup Efficiency	Manipulate Efficiency	Notes
Dictionary	Unordered	Yes	Via Key	Key: O(1)	O(1)	Best for high performance lookups.
SortedDictionary	Sorted	No	Via Key	Key: O(log n)	O(log n)	Compromise of Dictionary speed and ordering, uses binary search tree.
SortedList	Sorted	Yes	Via Key	Key: O(log n)	O(n)	Very similar to SortedDictionary, except tree is implemented in an array, so has faster lookup on preloaded data, but slower loads.
List	User has precise control over element ordering	Yes	Via Index	Index: O(1) Value: O(n)	O(n)	Best for smaller lists where direct access required and no sorting.
LinkedList	User has precise control over element ordering	No	No	Value: O(n)	O(1)	Best for lists where inserting/deleting in middle is common and no direct access required.
HashSet	Unordered	Yes	Via Key	Key: O(1)	O(1)	Unique unordered collection, like a Dictionary except key and value are same object.
SortedSet	Sorted	No	Via Key	Key: O(log n)	O(log n)	Unique sorted collection, like SortedDictionary except key and value are same object.
Stack	LIFO	Yes	Only Top	Top: O(1)	O(1)*	Essentially same as List<T> except only process as LIFO
Queue	FIFO	Yes	Only Front	Front: O(1)	O(1)	Essentially same as List<T> except only process as FIFO

867 × 845

2. Implementation of unordered containers
   Array of buckets, linked list of entries
   Adv - if hash function is fast - finding elements is very fast



3. D
   I
4. F
   I
5. F
   I
6. F
   I
7. F
8. F
   I
9. F
   I
10. F
    I
11. F
    I
12. F
    I
13. F
    I
14. D
    I
15. F

I
16. F
I
17. F
I
18. F
19. F
I
20. F
I
21. F
I
22. F
I
23. F
I
24. F
I
25. D
I
26. F
I
27. F
I
28. F
I
29. F
30. F
I
31. F
I
32. F
I
33. F
I
34. F
I
35. F
I
36. D
I
37. F
I
38. F
I
39. F
I
40. F
41. F
I
42. F
I
43. F
I
44. F
I
45. F
I