Unit 6 Study Guide and Review: Useful Examples and C++ Glossary

6a. Describe and code the binary tree structure

1. How do you define a binary tree?
    a. A binary tree structure is essentially a series of nodes that each have two branches. The parent node (root of the tree) has two children and each of those children have up to two children. There are a number of characteristics of a binary tree, which include root of the tree, as well as depth, size and base type of the tree. The bottom of the tree is empty. When you define a tree, you can define it in sets of three nodes(the parent and two children) or as a vector of nodes.
2. How do you search a binary tree?

Given a binary tree, return true if a node with the target data is found in the tree. Recurs down the tree, chooses the left or right branch by comparing the target to each node.

1. Preorder traversal: visit the root, traverse the left subtree, traverse the right subtree
2. Inorder traversal: traverse the left subtree, visit the root, traverse the right subtree
3. Postorder traversal: traverse the left subtree traverse the right subtree, visit the root

For more information on binary trees, refer to this page.

6b. Code special data structures

1. What are the different data structures available and what are the benefits and drawbacks of each?

There are many different types of data structures. Each of these structures essentially has the same purpose: to store data. Which data structure you use will be based on which data structure behaves as you need it to behave. Do you need the size of the structure to change? Do you need to be able to add to the end of the structure? Maybe you need to add to the beginning of the structure. These behaviors will dictate which data structure you use. For more information about data structures, refer to this page.

1) Array class (std::array)
2) Vector class (std::array)
3) List class (std::list)
4) Deque class (std::deque)
5) Forward_list class (std::forward_list)
6) Set class (std::set)
7) Unordered_set class (std::unordered_set)
8) Multiset class and
9) unordered_multiset class (std::set & std::unordered_set)
10) Maps: map, multimap, unordered_map and unordered_multimap (std::map, std::multimap, std::unordered_map and std::unordered_multimap)
11)Stack class (std::stack)
12)Queue class (std::queue)
13) Priority_queue class (std::priority_queue)

1)**Array class (std::array)**

Header: <array>
Declaration: template < class T, size_t N > class array;

Arrays are fixed size sequence container. It takes the element type and number of elements in the array as template arguments. The size of the sequence is fixed at compile time. So no memory management and dynamic allocation involved. This class does not store any extra data other than the sequence of elements. It only adds layer of functions so that it works as a container. This is different than other containers whose size is dynamically changed. This has all the property of normal language defined array declared with bracket ([]) syntax. Zero size containers are valid but they should not be dereferenced.

2)**Vector class (std::array)**

Header: <vector>
Declaration: template < class T, class Alloc = allocator<T> > class vector;

The instance of vector template class is a sequence container of a given data type. The elements are stored contiguously in memory. It stores the elements similar to a normal array but it can dynamically re-size the array by allocating and deallocating memory. To optimize the reallocation it allocates extra memory so that it does not need to reallocate memory every time an element is added. If you know the minimum number of elements it is better to allocate that by calling the reserve() method of the vector so that no reallocation is needed until the number of elements exceed the minimum number. Like array container, vector elements can be accessed

directly using [] operator and offset on a regular pointer to its elements. Vector is very efficient in accessing its elements with array index. Adding elements at the end is efficient but inserting element in between is costly compared to other containers. This is because every time inserting an element require some elements to be moved. Also resizing has an overhead which is optimized by intelligent memory allocation.

3)**List class (std::list)**

Header: <list>
Declaration: template < class T, class Alloc = allocator<T> > class list;

List is a sequence container which stores the elements in a doubly linked list. The sequence of the elements is maintained by associating to each element of a link to the preceding element and another link to the following element. So each element size is increased by 2 pointers. But this has advantage of constant time to insert or remove elements when you have an iterator of the sequence. It also allows to iterate both direction with constant time. On the other hand you cannot access elements directly. You have to use iterator and move it to reach a desired position to access the element

4)**Deque class (std::deque)**

Header: <deque>
Declaration: template < class T, class Alloc = allocator<T> > class deque;

Template container class std::deque (double-ended-queue) is a sequence container and behaves similar to the std::vector class with the difference that deque can grow/shrink in both direction. You can add/remove elements to/from both the ends. Unlike vector it does not guarantee to store all the elements in contiguous memory location. That's why accessing an element by using offset to a pointer to another element is not safe. This also makes the internal implementation of deque more complex than vector but it helps it to grow more efficiently in some scenarios, specially when having very long sequence and reallocation is expensive.

6)**Forward_list class (std::forward_list)**

Header: <forward_list>
Declaration: template < class T, class Alloc = allocator<T> > class forward_list;

forward_list container class is a sequence container which stores the elements in a singly-linked list. Each element may be stored in different and unrelated memory locations. The sequence of the elements is maintained by associating to each element of a link to the following element. So

each element size is lesser by 1 pointers than list. It is similar to list with the difference that unlike list it does not allow traverse backwards. Each element hold the reference to the next element but not the previous element. Each element having only single link, insertion and removal of element is more efficient than list. Forward list is more efficient than other containers like array, vector and deque in terms of insertion, deletion, moving and the algorithm like sort which use those operations.

Unlike list, forward_list does not allow to access the elements directly. If you want to access to 7th elements you have to iterate from the beginning to the 7th element to access it. It is as efficient as a simple c-style singly-linked-list.

7)**Header:** <set>
Set is an associative container and store the unique elements in a specific order. Internally the elements are stored in a binary search tree. The value of an element also identifies itself, means the value is itself the key. The elements cannot be modified once they are in the container but they can be removed or inserted to the container. Set containers are slower than unordered_set to access individual elements by their key but it allows direct iteration on their elements based on their order. It supports bi-directional iterator

8)**Unordered_set class (std::unordered_set)**
unordered_set is an associative container and store the unique elements without any order. Internally the elements are stored in hash tables. The value of an element also identifies itself, means the value is itself the key. The elements cannot be modified once they are in the container but they can be removed or inserted in the container. unordered_set containers are faster than set to access individual elements by their key but slower in range iteration. It supports forward iterator.

9)**Multiset class and unordered_multiset class**
Multiset and unordered_multiset are similar to set and unordered_set respectively with the property that they can contain duplicate values. These classes are also declared in the same headers <set> and <unordered_set>. For unordered multi-set, duplicate valued elements are stored in the same bucket

10)**Maps:**
Maps are like set but the difference between sets and maps is that every element in the map is a key-value pair and the map elements are arranged and accessed on the basis of key and not the value. Key and value can be off different types. In case of sets value itself is treated as the key. Both sets and maps are implemented same way- binary search tree for ordered map and hash tables for unordered map.

11)**Stack class (std::stack)**
Container adapter class stack is designed for the LIFO(last-in first-out) context where elements are inserted and removed only at and from one end. The element last inserted will be removed

first. Inserting an element is called push operation and removing an element is called pop operation. This is a container adapter and use some other container to store the elements. The underlying container types should support following operations:

- empty
- size
- back
- push_back
- pop_back

vector, deque and list classes fulfills those requirement. Default container is deque. You can specify the container type while instantiation. Elements are inserted and removed from the back of the underlying container.

12)**Queue class (std::queue)**
Container adapter class queue is designed for the FIFO(first-in first-out) context where elements are inserted at one end and removed from the other end. The element first inserted will be removed first and inserted last will be remove last. Inserting an element is called enque operation and removing an element is called dequeue operation. This is a container adapter and use some other container to store the elements. The underlying container types should support following operations:

- empty
- size
- back
- front
- push_back
- pop_front

deque and list classes fulfills those requirement. Default container is deque. You can specify the container type while instantiation. Elements are inserted at the back of the container and remove from the front.

13)**Priority_queue class (std::priority_queue)**
Container adapter class priority_queue is designed such a way that it's greatest element is always the first element according to a strict weak ordering criterion. Internally it maintains a heap structure where an element can be inserted at any moment and only the max heap element can be retrieved which is at the top of the priority_queue. This is a container adapter and use some other container to store the elements. The underlying container types should support following operations:

- empty
- size
- front
- push_back
- pop_back

deque and vector classes fulfills those requirement. Default container is vector. You can specify the container type while instantiation. Elements are popped from the back of the specific container. Support of random access iterator is required to keep a heap structure internally all the time. This is done by the container adapter by automatically calling algorithm functions make_heap, push_heap and pop_heap when needed.

6c. Describe and code the container class objects

1. ## How do you declare an instance of each of the different container types?

1) Array class (std::array)

```
array<type, number_of_elements> arrayName ;
```

2) Vector class (std::array)

```
vector<type> vectorName;
```

3) List class (std::list)

```
list<type>ListName;
```

4) Deque class (std::deque)

```
deque<type>DequeName;
```

5) Forward_list class (std::forward_list)

```
forward_list<type>FlistName;
```

6) Set class (std::set)

```
set<type>SetName;
```

7) Unordered_set class (std::unordered_set)

unordered_set<type>unorderedSetName;

8) Multiset class and

multiset<type>multisetName;

9) unordered_multiset class (std::set & std::unordered_set)

unordered_multiset<type> unorderdedMultisetName;

10) Maps: map, multimap, unordered_map and unordered_multimap (std::map, std::multimap, std::unordered_map and std::unordered_multimap)

map<keyType, MappedType> MapName;

multmap<keyType, MappedType> multiMapName;

unordered_map<keyType, MappedType> unorderedMapName;

unordered_multimap<map<keyType, MappedType> unorderedMultiMapName;

11)Stack class (std::stack)

Stack modifies an underlying container. Underlying Containers can be of type Vector, Deque, or List. Default is deque.

stack< type, underlyingContainerType< uctType>> stackName(uctName);

```
 vector<int> v{1,2,3,4,5,6,7,8,9,10};
//Stack initialize with vector object
stack<int, vector<int>> mystack(v);
```

12)Queue class (std::queue)

Queue modifies an underlying container. Underlying Containers can be of type Vector, Deque, or List. Default is deque.

queue< type, underlyingContainerType< uctType>> queueName(uctName);

```
 list<int> lst{1,2,3,4,5,6,7,8,9,10};
//Queue initialize with list object
queue<int, list<int>> myqueue(lst);
cout << "Size of the initialized queue: " << myqueue.size()<<endl;
while (!myqueue.empty())
```

13) Priority_queue class (std::priority_queue)

Priority Queue modifies an underlying container. Underlying Containers can be of type Vector, Deque, or List. Default is vector.

priority_queue< type, underlyingContainerType< uctType>> priorityQueueName(uctName);

```
deque<int> dq{1,10,3,4,6,5,7,8,9,2};
   //Priority que will keep the smallest number at the top
   priority_queue<int, deque<int>, greater<int>> mypqueue(greater<int>(), dq);
```

2. What are the methods for each container type?

1) Array class (std::array)

Array class Member functions

Iterators

Begin

Return iterator to beginning (public member function)

end

Return iterator to end (public member function)

rbegin

Return reverse iterator to reverse beginning (public member function)

rend

Return reverse iterator to reverse end (public member function)

cbegin

Return const_iterator to beginning (public member function)

cend

Return const_iterator to end (public member function)

crbegin

Return const_reverse_iterator to reverse beginning (public member function)

crend

Return const_reverse_iterator to reverse end (public member function)

Capacity

size
Return size (public member function)
max_size
Return maximum size (public member function)
empty
Test whether array is empty (public member function)

Element access
operator[]
Access element (public member function)
at
Access element (public member function)
front
Access first element (public member function)
back
Access last element (public member function)
data
Get pointer to data (public member function)

Modifiers
Fill
Fill array with value (public member function)
swap
Swap content (public member function)

2) Vector class (std::array)

Vector Member functions
(constructor)
Construct vector (public member function)
(destructor)
Vector destructor (public member function)
operator=
Assign content (public member function)

Iterators:
begin
Return iterator to beginning (public member function)

end
Return iterator to end (public member function)
rbegin
Return reverse iterator to reverse beginning (public member function)
rend
Return reverse iterator to reverse end (public member function)
cbegin
Return const_iterator to beginning (public member function)
cend
Return const_iterator to end (public member function)
crbegin
Return const_reverse_iterator to reverse beginning (public member function)
crend
Return const_reverse_iterator to reverse end (public member function)

Capacity:
size
Return size (public member function)
max_size
Return maximum size (public member function)
resize
Change size (public member function)
capacity
Return size of allocated storage capacity (public member function)
empty
Test whether vector is empty (public member function)
reserve
Request a change in capacity (public member function)
shrink_to_fit
Shrink to fit (public member function)

Element access:
operator[]
Access element (public member function)
at
Access element (public member function)
front

Access first element (public member function)

back

Access last element (public member function)

data

Access data (public member function)

Modifiers:

assign

Assign vector content (public member function)

push_back

Add element at the end (public member function)

pop_back

Delete last element (public member function)

insert

Insert elements (public member function)

erase

Erase elements (public member function)

swap

Swap content (public member function)

clear

Clear content (public member function)

emplace

Construct and insert element (public member function)

emplace_back

Construct and insert element at the end (public member function)

3) List class (std::list)

List Member functions

(constructor)

Construct list (public member function)

(destructor)

List destructor (public member function)

operator=

Assign content (public member function)

Iterators:

begin

Return iterator to beginning (public member function)

end

Return iterator to end (public member function)

rbegin

Return reverse iterator to reverse beginning (public member function)

rend

Return reverse iterator to reverse end (public member function)

cbegin

Return const_iterator to beginning (public member function)

cend

Return const_iterator to end (public member function)

crbegin

Return const_reverse_iterator to reverse beginning (public member function)

crend

Return const_reverse_iterator to reverse end (public member function)

Capacity:

empty

Test whether container is empty (public member function)

size

Return size (public member function)

max_size

Return maximum size (public member function)

Element access:

front

Access first element (public member function)

back

Access last element (public member function)

Modifiers:

assign

Assign new content to container (public member function)

emplace_front

Construct and insert element at beginning (public member function)

push_front

Insert element at beginning (public member function)

pop_front
Delete first element (public member function)
emplace_back
Construct and insert element at the end (public member function)
push_back
Add element at the end (public member function)
pop_back
Delete last element (public member function)
emplace
Construct and insert element (public member function)
insert
Insert elements (public member function)
erase
Erase elements (public member function)
swap
Swap content (public member function)
resize
Change size (public member function)
clear
Clear content (public member function)

Operations:
splice
Transfer elements from list to list (public memberfunction)
remove
Remove elements with specific value (public member function)
remove_if
Remove elements fulfilling condition (public member function template)
unique
Remove duplicate values (public member function)
merge
Merge sorted lists (public member function)
sort
Sort elements in container (public member function)
reverse
Reverse the order of elements (public member function)

Observers:

get_allocator

Get allocator (public member function)

4) Deque class (std::deque)

Deque Member functions

(constructor)

Construct deque container (public member function)

(destructor)

Deque destructor (public member function)

operator=

Assign content (public member function)

Iterators:

begin

Return iterator to beginning (public member function)

end

Return iterator to end (public member function)

rbegin

Return reverse iterator to reverse beginning (public member function)

rend

Return reverse iterator to reverse end (public member function)

cbegin

Return const_iterator to beginning (public member function)

cend

Return const_iterator to end (public member function)

crbegin

Return const_reverse_iterator to reverse beginning (public member function)

crend

Return const_reverse_iterator to reverse end (public member function)

Capacity:

size

Return size (public member function)

max_size

Return maximum size (public member function)

resize
Change size (public member function)
empty
Test whether container is empty (public member function)
shrink_to_fit
Shrink to fit (public member function)

Element access:
operator[]
Access element (public member function)
at
Access element (public member function)
front
Access first element (public member function)
back
Access last element (public member function)

Modifiers:
assign
Assign container content (public member function)
push_back
Add element at the end (public member function)
push_front
Insert element at beginning (public member function)
pop_back
Delete last element (public member function)
pop_front
Delete first element (public member function)
insert
Insert elements (public member function)
erase
Erase elements (public member function)
swap
Swap content (public member function)
clear
Clear content (public member function)
emplace

Construct and insert element (public member function)

emplace_front

Construct and insert element at beginning (public member function)

emplace_back

Construct and insert element at the end (public member function)

5) Forward_list class (std::forward_list)

Forward_list Member functions

(constructor)

Construct forward_list object (public member function)

(destructor)

Destroy forward_list object (public member function)

operator=

Assign content (public member function)

Iterators

before_begin

Return iterator to before beginning (public member function)

begin

Return iterator to beginning (public member type)

end

Return iterator to end (public member function)

cbefore_begin

Return const_iterator to before beginning (public member function)

cbegin

Return const_iterator to beginning (public member function)

cend

Return const_iterator to end (public member function)

Capacity

empty

Test whether array is empty (public member function)

max_size

Return maximum size (public member function)

Element access

front

Access first element (public member function)

Modifiers

assign

Assign content (public member function)

emplace_front

Construct and insert element at beginning (public member function)

push_front

Insert element at beginning (public member function)

pop_front

Delete first element (public member function)

emplace_after

Construct and insert element (public member function)

insert_after

Insert elements (public member function)

erase_after

Erase elements (public member function)

swap

Swap content (public member function)

resize

Change size (public member function)

clear

Clear content (public member function)

Operations

splice_after

Transfer elements from another forward_list (public member function)

remove

Remove elements with specific value (public member function)

remove_if

Remove elements fulfilling condition (public member function template)

unique

Remove duplicate values (public member function)

merge

Merge sorted lists (public member function)

sort

Sort elements in container (public member function)

reverse
Reverse the order of elements (public member function)

Observers
get_allocator
Get allocator (public member function)

6) Set class (std::set)

Set Member functions
(constructor)
Construct set (public member function)
(destructor)
Set destructor (public member function)
operator=
Copy container content (public member function)

Iterators:
begin
Return iterator to beginning (public member function)
end
Return iterator to end (public member function)
rbegin
Return reverse iterator to reverse beginning (public member function)
rend
Return reverse iterator to reverse end (public member function)
cbegin
Return const_iterator to beginning (public member function)
cend
Return const_iterator to end (public member function)
crbegin
Return const_reverse_iterator to reverse beginning (public member function)
crend
Return const_reverse_iterator to reverse end (public member function)

Capacity:
empty

Test whether container is empty (public member function)

size

Return container size (public member function)

max_size

Return maximum size (public member function)

Modifiers:

insert

Insert element (public member function)

erase

Erase elements (public member function)

swap

Swap content (public member function)

clear

Clear content (public member function)

emplace

Construct and insert element (public member function)

emplace_hint

Construct and insert element with hint (public member function)

Observers:

key_comp

Return comparison object (public member function)

value_comp

Return comparison object (public member function)

Operations:

find

Get iterator to element (public member function)

count

Count elements with a specific value (public member function)

lower_bound

Return iterator to lower bound (public member function)

upper_bound

Return iterator to upper bound (public member function)

equal_range

Get range of equal elements (public member function)

7) Unordered_set class (std::unordered_set)

Unordered_set Member functions

(constructor)

Construct unordered_set (public member function)

(destructor)

Destroy unordered set (public member function)

operator=

Assign content (public member function)

Capacity

empty

Test whether container is empty (public member function)

size

Return container size (public member function)

max_size

Return maximum size (public member function)

Iterators

begin

Return iterator to beginning (public member type)

end

Return iterator to end (public member type)

cbegin

Return const_iterator to beginning (public member function)

cend

Return const_iterator to end (public member function)

Element lookup

find

Get iterator to element (public member function)

count

Count elements with a specific key (public member function)

equal_range

Get range of elements with a specific key (public member function)

Modifiers

emplace

Construct and insert element (public member function)

emplace_hint

Construct and insert element with hint (public member function)

insert

Insert elements (public member function)

erase

Erase elements (public member function)

clear

Clear content (public member function)

swap

Swap content (public member function)

Buckets

bucket_count

Return number of buckets (public member function)

max_bucket_count

Return maximum number of buckets (public member function)

bucket_size

Return bucket size (public member type)

bucket

Locate element's bucket (public member function)

Hash policy

load_factor

Return load factor (public member function)

max_load_factor

Get or set maximum load factor (public member function)

rehash

Set number of buckets (public member function)

reserve

Request a capacity change (public member function)

Observers

hash_function

Get hash function (public member type)

key_eq

Get key equivalence predicate (public member type)

get_allocator

Get allocator (public member function)

Non-member function overloads

operators (unordered_set)

Relational operators for unordered_set (function template)

swap (unordered_set)

Exchanges contents of two unordered_set containers (function template)

8) Multiset class (std::multiset) and

Multiset Member functions

(constructor)

Construct multiset (public member function)

(destructor)

Multiset destructor (public member function)

operator=

Copy container content (public member function)

Iterators:

begin

Return iterator to beginning (public member function)

end

Return iterator to end (public member function)

rbegin

Return reverse iterator to reverse beginning (public member function)

rend

Return reverse iterator to reverse end (public member function)

cbegin

Return const_iterator to beginning (public member function)

cend

Return const_iterator to end (public member function)

crbegin

Return const_reverse_iterator to reverse beginning (public member function)

crend
Return const_reverse_iterator to reverse end (public member function)

Capacity:

empty
Test whether container is empty (public member function)

size
Return container size (public member function)

max_size
Return maximum size (public member function)

Modifiers:

insert
Insert element (public member function)

erase
Erase elements (public member function)

swap
Swap content (public member function)

clear
Clear content (public member function)

emplace
Construct and insert element (public member function)

emplace_hint
Construct and insert element with hint (public member function)

Observers:

key_comp
Return comparison object (public member function)

value_comp
Return comparison object (public member function)

Operations:

find
Get iterator to element (public member function)

count
Count elements with a specific key (public member function)

lower_bound
Return iterator to lower bound (public member function)

upper_bound
Return iterator to upper bound (public member function)
equal_range
Get range of equal elements (public member function)

9) unordered_multiset class (std::set & std::unordered_set)

unordered_multiset **member functions**
(constructor)
Construct unordered_multiset (public member function)
(destructor)
Destroy unordered multiset (public member function)
operator=
Assign content (public member function)

Capacity
empty
Test whether container is empty (public member function)
size
Return container size (public member function)
max_size
Return maximum size (public member function)

Iterators
begin
Return iterator to beginning (public member type)
end
Return iterator to end (public member type)
cbegin
Return const_iterator to beginning (public member type)
cend
Return const_iterator to end (public member type)

Element lookup
find
Get iterator to element (public member function)

count
Count elements with a specific key (public member function)
equal_range
Get range of elements with specific key (public member function)

Modifiers
emplace
Construct and insert element (public member function)
emplace_hint
Construct and insert element with hint (public member function)
insert
Insert elements (public member function)
erase
Erase elements (public member function)
clear
Clear content (public member function)
swap
Swap content (public member function)

Buckets
bucket_count
Return number of buckets (public member function)
max_bucket_count
Return maximum number of buckets (public member function)
bucket_size
Return bucket size (public member type)
bucket
Locate element's bucket (public member function)

Hash policy
load_factor
Return load factor (public member function)
max_load_factor
Get or set maximum load factor (public member function)
rehash
Set number of buckets (public member function)
reserve

Request a capacity change (public member function)

Observers

hash_function

Get hash function (public member type)

key_eq

Get key equivalence predicate (public member type)

get_allocator

Get allocator (public member function)

Non-member function overloads

operators (unordered_multiset)

Relational operators for unordered_multiset(function template)

swap (unordered_multiset)

Exchanges contents of two unordered_multiset containers (function template)

10) Maps: map, multimap, unordered_map and unordered_multimap (std::map, std::multimap, std::unordered_map and std::unordered_multimap)

Map Member functions

(constructor)

Construct map (public member function)

(destructor)

Map destructor (public member function)

operator=

Copy container content (public member function)

Iterators:

begin

Return iterator to beginning (public member function)

end

Return iterator to end (public member function)

rbegin

Return reverse iterator to reverse beginning (public member function)

rend

Return reverse iterator to reverse end (public member function)

cbegin

Return const_iterator to beginning (public member function)

cend

Return const_iterator to end (public member function)

crbegin

Return const_reverse_iterator to reverse beginning (public member function)

crend

Return const_reverse_iterator to reverse end (public member function)

Capacity:

empty

Test whether container is empty (public member function)

size

Return container size (public member function)

max_size

Return maximum size (public member function)

Element access:

operator[]

Access element (public member function)

at

Access element (public member function)

Modifiers:

insert

Insert elements (public member function)

erase

Erase elements (public member function)

swap

Swap content (public member function)

clear

Clear content (public member function)

emplace

Construct and insert element (public member function)

emplace_hint

Construct and insert element with hint (public member function)

Observers:

key_comp

Return key comparison object (public member function)

value_comp

Return value comparison object (public member function)

Operations:

find

Get iterator to element (public member function)

count

Count elements with a specific key (public member function)

lower_bound

Return iterator to lower bound (public member function)

upper_bound

Return iterator to upper bound (public member function)

equal_range

Get range of equal elements (public member function)

Multimap Member functions

(constructor)

Construct multimap (public member function)

(destructor)

Multimap destructor (public member function)

operator=

Copy container content (public member function)

Iterators:

begin

Return iterator to beginning (public member function)

end

Return iterator to end (public member function)

rbegin

Return reverse iterator to reverse beginning (public member function)

rend

Return reverse iterator to reverse end (public member function)

cbegin

Return const_iterator to beginning (public member function)

cend

Return const_iterator to end (public member function)

crbegin

Return const_reverse_iterator to reverse beginning (public member function)

crend

Return const_reverse_iterator to reverse end (public member function)

Capacity:

empty

Test whether container is empty (public member function)

size

Return container size (public member function)

max_size

Return maximum size (public member function)

Modifiers:

insert

Insert element (public member function)

erase

Erase elements (public member function)

swap

Swap content (public member function)

clear

Clear content (public member function)

emplace

Construct and insert element (public member function)

emplace_hint

Construct and insert element with hint (public member function)

Observers:

key_comp

Return key comparison object (public member function)

value_comp

Return value comparison object (public member function)

Operations:

find

Get iterator to element (public member function)

count

Count elements with a specific key (public member function)

lower_bound

Return iterator to lower bound (public member function)

upper_bound

Return iterator to upper bound (public member function)

equal_range

Get range of equal elements (public member function)

Allocator:

get_allocator

Get allocator (public member function

Unordered_map Member functions

(constructor)

Construct unordered_map (public member function)

(destructor)

Destroy unordered map (public member function)

operator=

Assign content (public member function)

Capacity

empty

Test whether container is empty (public member function)

size

Return container size (public member function)

max_size

Return maximum size (public member function)

Iterators

begin

Return iterator to beginning (public member function)

end

Return iterator to end (public member function)

cbegin

Return const_iterator to beginning (public member function)

cend

Return const_iterator to end (public member function)

Element access

operator[]
Access element (public member function)
at
Access element (public member function)

Element lookup
find
Get iterator to element (public member function)
count
Count elements with a specific key (public member function)
equal_range
Get range of elements with specific key (public member function)

Modifiers
emplace
Construct and insert element (public member function)
emplace_hint
Construct and insert element with hint (public member function)
insert
Insert elements (public member function)
erase
Erase elements (public member function)
clear
Clear content (public member function)
swap
Swap content (public member function)

Buckets
bucket_count
Return number of buckets (public member function)
max_bucket_count
Return maximum number of buckets (public member function)
bucket_size
Return bucket size (public member type)
bucket
Locate element's bucket (public member function)

Hash policy

load_factor

Return load factor (public member function)

max_load_factor

Get or set maximum load factor (public member function)

rehash

Set number of buckets (public member function)

reserve

Request a capacity change (public member function)

Observers

hash_function

Get hash function (public member type)

key_eq

Get key equivalence predicate (public member type)

get_allocator

Get allocator (public member function)

Non-member function overloads

operators (unordered_map)

Relational operators for unordered_map (function template)

swap (unordered_map)

Exchanges contents of two unordered_map containers (function template)

Unordered_multipmap Member functions

(constructor)

Construct unordered_multimap (public member function)

(destructor)

Destroy unordered multimap (public member function)

operator=

Assign content (public member function)

Capacity

empty

Test whether container is empty (public member function)

size

Return container size (public member function)

max_size
Return maximum size (public member function)

Iterators
begin
Return iterator to beginning (public member type)
end
Return iterator to end (public member type)
cbegin
Return const_iterator to beginning (public member function)
cend
Return const_iterator to end (public member function)

Element lookup
find
Get iterator to element (public member function)
count
Count elements with a specific key (public member function)
equal_range
Get range of elements with specific key (public member function)

Modifiers
emplace
Construct and insert element (public member function)
emplace_hint
Construct and insert element with hint (public member function)
insert
Insert elements (public member function)
erase
Erase elements (public member function)
clear
Clear content (public member function)
swap
Swap content (public member function)

Buckets
bucket_count

Return number of buckets (public member function)

max_bucket_count

Return maximum number of buckets (public member function)

bucket_size

Return bucket_size (public member type)

bucket

Locate element's bucket (public member function)

Hash policy

load_factor

Return load factor (public member function)

max_load_factor

Get or set maximum load factor (public member function)

rehash

Set number of buckets (public member function)

reserve

Request a capacity change (public member function)

Observers

hash_function

Get hash function (public member type)

key_eq

Get key equivalence predicate (public member type)

get_allocator

Get allocator (public member function)

Non-member function overloads

operators (unordered_multimap)

Relational operators for unordered_multimap (function template)

swap (unordered_multimap)

Exchanges contents of two unordered_multimap containers (function template)

11)Stack class (std::stack)

Stack Member functions

(constructor)

Construct stack (public member function)

empty

Test whether container is empty (public member function)

size

Return size (public member function)

top

Access next element (public member function)

push

Insert element (public member function)

emplace

Construct and insert element (public member function)

pop

Remove top element (public member function)

swap

Swap contents (public member function)

Non-member function overloads

relational operators

Relational operators for stack (function)

swap (stack)

Exchange contents of stacks (public member function)

Non-member class specializations

uses_allocator<stack>

uses allocator for stack (class template)

12)Queue class (std::queue)

Queue Member functions

(constructor)

Construct queue (public member function)

empty

Test whether container is empty (public member function)

size

Return size (public member function)

front

Access next element (public member function)

back

Access last element (public member function)

push

Insert element (public member function)

emplace

Construct and insert element (public member function)

pop

Remove next element (public member function)

swap

Swap contents (public member function)

Non-member function overloads

relational operators

Relational operators for queue (function)

swap (queue)

Exchange contents of queues (public member function)

Non-member class specializations

uses_allocator<queue>

Uses allocator for queue (class template)

13) Priority_queue class (std::priority_queue)

Priority_queue Member functions

(constructor)

Construct priority queue (public member function)

empty

Test whether container is empty (public member function)

size

Return size (public member function)

top

Access top element (public member function)

push

Insert element (public member function)

emplace

Construct and insert element (public member function)

pop

Remove top element (public member function)

swap

Swap contents (public member function)

Non-member function overloads

swap (queue)

Exchange contents of priority queues (public member function)

Non-member class specializations

uses_allocator<queue>

Uses allocator for priority queue (class template)

Whether it is a vector or an array, or any of the other data structures, you need to know how to declare them. Do you need to set the size, as in an array, does it require a deque container, as in a stack, or do you allocate memory? For more information about data structures, refer to this page.

6d. Describe and code the vector container

1. ## How do you declare a vector?

   vector<type> vectorName;

   Or

   vector<type> vectorname(numofElem, InitialValue);

2. ## How do you add and remove items from a vector?

Modifiers:

assign

Assign vector content (public member function)

push_back

Add element at the end (public member function)

pop_back

Delete last element (public member function)

insert

Insert elements (public member function)

erase

Erase elements (public member function)

swap

Swap content (public member function)

clear

Clear content (public member function)

emplace

Construct and insert element (public member function)

emplace_back

Construct and insert element at the end (public member function)

3. ## How do you search a vector?

Use the std:: find iterator

- Find is an iterator to the first element in the range that compares equal to a specified value. If no elements match, the function returns the last element.

To use you will need to include the algorithm and vector headers.

Then use the following:

find(vectorName.begin(), vectorName.end(), elementToFind);

```
// find example
#include <iostream>     // std::cout
#include <algorithm>    // std::find
#include <vector>       // std::vector
```

```
  // using std::find with vector and iterator:
  std::vector<int> myvector (myints,myints+4);
  std::vector<int>::iterator it;

  it = find (myvector.begin(), myvector.end(), 30);
  if (it != myvector.end())
    std::cout << "Element found in myvector: " << *it << '\n';
  else
    std::cout << "Element not found in myvector\n";

  return 0;
}
```

The vector class is a container, but it is usually explored separately, because it is such a useful structure. The greatest benefit of a vector over an array is the ability to resize the array, improving efficiency and improved memory management. For more information about vectors review this page and this page.

Unit 6 Vocabulary

This vocabulary list includes terms that might help you with the review items above and some terms you should be familiar with to be successful in completing the final exam for the course. Try to think of the reason why each term is included.

- Array class
    - Arrays are fixed-size sequence containers: they hold a specific number of elements ordered in a strict linear sequence.
    - Internally, an array does not keep any data other than the elements it contains (not even its size, which is a template parameter, fixed on compile time). It is as efficient in terms of storage size as an ordinary array declared with the language's bracket syntax ([]). This class merely adds a layer of member and global functions to it, so that arrays can be used as standard containers.
    - Unlike the other standard containers, arrays have a fixed size and do not manage the allocation of its elements through an allocator: they are an aggregate type encapsulating a fixed-size array of elements. Therefore, they cannot be expanded or contracted dynamically (see vector for a similar container that can be expanded).
    - Zero-sized arrays are valid, but they should not be dereferenced (members front, back, and data).
    - Unlike with the other containers in the Standard Library, swapping two array containers is a linear operation that involves swapping all the elements in the ranges individually, which generally is a considerably less efficient operation. On the other side, this allows the iterators to elements in both containers to keep their original container association.
    - Another unique feature of array containers is that they can be treated as tuple objects: The <array> header overloads the get function to access the elements of the array as if it was a tuple, as well as specialized tuple_size and tuple_element types.
- Binary tree
    - a **tree** data structure in which each node has at most two children, which are referred to as the left child and the right child.
- Container class
    - 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.
- Data structures
    - data organization, management and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.
- Deque class

- (usually pronounced like *"deck"*) is an irregular acronym of **d**ouble-**e**nded **que**ue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

- Specific libraries may implement *deques* in different ways, generally as some form of dynamic array. But in any case, they allow for the individual elements to be accessed directly through random access iterators, with storage handled automatically by expanding and contracting the container as needed.

- Therefore, they provide a functionality similar to vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end. But, unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations: accessing elements in a deque by offsetting a pointer to another element causes *undefined behavior*.

- Both vectors and deques provide a very similar interface and can be used for similar purposes, but internally both work in quite different ways: While vectors use a single array that needs to be occasionally reallocated for growth, the elements of a deque can be scattered in different chunks of storage, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time and with a uniform sequential interface (through iterators). Therefore, deques are a little more complex internally than vectors, but this allows them to grow more efficiently under certain circumstances, especially with very long sequences, where reallocations become more expensive.

- For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, deques perform worse and have less consistent iterators and references than lists and forward lists.

- Forward_list class
  - Forward lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence.
  - Forward lists are implemented as singly-linked lists; Singly linked lists can store each of the elements they contain in different and unrelated storage locations.

The ordering is kept by the association to each element of a link to the next element in the sequence.

- The main design difference between a forward_list container and a list container is that the first keeps internally only a link to the next element, while the latter keeps two links per element: one pointing to the next element and one to the preceding one, allowing efficient iteration in both directions, but consuming additional storage per element and with a slight higher time overhead inserting and removing elements. forward_list objects are thus more efficient than list objects, although they can only be iterated forwards.
- Compared to other base standard sequence containers (array, vector and deque), forward_list perform generally better in inserting, extracting and moving elements in any position within the container, and therefore also in algorithms that make intensive use of these, like sorting algorithms.
- The main drawback of forward_lists and lists compared to these other sequence containers is that they lack direct access to the elements by their position; For example, to access the sixth element in a forward_list one has to iterate from the beginning to that position, which takes linear time in the distance between these. They also consume some extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements).

- The forward_list class template has been designed with efficiency in mind: By design, it is as efficient as a simple handwritten C-style singly-linked list, and in fact is the only standard container to deliberately lack a size member function for efficiency considerations: due to its nature as a linked list, having a size member that takes constant time would require it to keep an internal counter for its size (as list does). This would consume some extra storage and make insertion and removal operations slightly less efficient. To obtain the size of a forward_list object, you can use the distance algorithm with its begin and end, which is an operation that takes linear time.

- Heterogeneous tree
  - Heterogeneous Data Structures are those data structures that contains a variety or dissimilar type of data, for e.g. a data structure that can contain various data of different data types like integer, float and character. The examples of such data structures include structures, union etc.
- Left subtree - all nodes to left of the root node in a binary tree
- List class

  Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.

List containers are implemented as doubly-linked lists; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it.

They are very similar to forward_list: The main difference being that forward_list objects are single-linked lists, and thus they can only be iterated forwards, in exchange for being somewhat smaller and more efficient.

Compared to other base standard sequence containers (array, vector and deque), lists perform generally better in inserting, extracting and moving elements in any position within the container for which an iterator has already been obtained, and therefore also in algorithms that make intensive use of these, like sorting algorithms.

The main drawback of lists and forward_lists compared to these other sequence containers is that they lack direct access to the elements by their position; For example, to access the sixth element in a list, one has to iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these. They also consume some extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements).

- Maps
    - Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.
    - In a map, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type value_type, which is a pair type combining both:

```
typedef pair<const Key, T> value_type;
```

    - Internally, the elements in a map are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type Compare).

    - map containers are generally slower than unordered_map containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

    - The mapped values in a map can be accessed directly by their corresponding key using the *bracket operator* ((operator[]).
    - Maps are typically implemented as *binary search trees*.

- Multimap
    - Multimaps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order, and where multiple elements can have equivalent keys.
    - In a multimap, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type value_type, which is a pair type combining both:

```
typedef pair<const Key, T> value_type;
```

    - Internally, the elements in a multimap are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type Compare).
    - multimap containers are generally slower than unordered_multimap containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.
    - Multimaps are typically implemented as *binary search trees*.
- Multiset class
    - **Multiple-key set**
    - Multisets are containers that store elements following a specific order, and where multiple elements can have equivalent values.

    - In a multiset, the value of an element also identifies it (the value is itself the *key*, of type T). The value of the elements in a multisetcannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

- Internally, the elements in a multiset are always sorted following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type Compare).

- multiset containers are generally slower than unordered_multiset containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.
- Multisets are typically implemented as *binary search trees*.

- Priority_queue class

Priority queue

Priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains, according to some *strict weak ordering* criterion.

This context is similar to a *heap*, where elements can be inserted at any moment, and only the *max heap* element can be retrieved (the one at the top in the *priority queue*).

Priority queues are implemented as *container adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *popped* from the *"back"* of the specific container, which is known as the *top* of the priority queue.

The underlying container may be any of the standard container class templates or some other specifically designed container class. The container shall be accessible through *random access iterators* and support the following operations:

- empty()

- size()
- front()
- push_back()
- pop_back()

The standard container classes vector and deque fulfill these requirements. By default, if no container class is specified for a particular priority_queue class instantiation, the standard container vector is used.

Support of *random access iterators* is required to keep a heap structure internally at all times. This is done automatically by the container adaptor by automatically calling the algorithm functions make_heap, push_heap and pop_heap when needed.

- Queue class

FIFO queue

queues are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.

queues are implemented as *containers adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *pushed* into the *"back"* of the specific container and *popped* from its *"front"*.

The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

- empty
- size
- front

- back
- push_back
- pop_front

The standard container classes deque and list fulfill these requirements. By default, if no container class is specified for a particular queue class instantiation, the standard container deque is used.

- Recursion - Recursion is when a function calls itself. That is, in the course of the function definition there is a call to that very same function. Consists of a base case which can be readily returned, and a recursive case that gets closer and closer to the base case.

- Right subtree - all nodes to the right of the root node in a binary tree
- Set class

Set

Sets are containers that store unique elements following a specific order.

In a set, the value of an element also identifies it (the value is itself the *key*, of type T), and each value must be unique. The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

Internally, the elements in a set are always sorted following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type Compare).

set containers are generally slower than unordered_set containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

Sets are typically implemented as *binary search trees*.

- Stack class

LIFO stack

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.

stacks are implemented as *containers adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *pushed*/*popped* from the *"back"*of the specific container, which is known as the *top* of the stack.

The underlying container may be any of the standard container class templates or some other specifically designed container class. The container shall support the following operations:

- empty
- size
- back
- push_back
- pop_back

The standard container classes vector, deque and list fulfill these requirements. By default, if no container class is specified for a particular stack class instantiation, the standard container deque is used.

- Unordered_map

Unordered Map

Unordered maps are associative containers that store elements formed by the combination of a *key value* and a *mapped value*, and which allows for fast retrieval of individual elements based on their keys.

In an unordered_map, the *key value* is generally used to uniquely identify the element, while the *mapped value* is an object with the content associated to this *key*. Types of *key* and *mapped* value may differ.

Internally, the elements in the unordered_map are not sorted in any particular order with respect to either their *key* or *mapped* values, but organized into *buckets* depending on their hash values to allow for fast access to individual elements directly by their *key values* (with a constant average time complexity on average).

unordered_map containers are faster than map containers to access individual elements by their *key*, although they are generally less efficient for range iteration through a subset of their elements.

Unordered maps implement the direct access operator (operator[]) which allows for direct access of the *mapped value* using its *key value* as argument.

Iterators in the container are at least forward iterators.

- Unordered_multimap

  Unordered multimaps are associative containers that store elements formed by the combination of a *key value* and a *mapped value*, much like unordered_map containers, but allowing different elements to have equivalent keys.

  In an unordered_multimap, the *key value* is generally used to uniquely identify the element, while the *mapped value* is an object with the content associated to this *key*. Types of *key* and *mapped* value may differ.

  Internally, the elements in the unordered_multimap are not sorted in any particular order with respect to either their *key* or *mapped*values, but organized into *buckets* depending on their hash values to allow for fast access to individual

elements directly by their *key values* (with a constant average time complexity on average).

Elements with equivalent keys are grouped together in the same bucket and in such a way that an iterator (see equal_range) can iterate through all of them.

Iterators in the container are at least forward iterators.

Notice that this container is not defined in its own header, but shares header <unordered_map> with unordered_map.

- Unordered_multiset class

  Unordered multisets are containers that store elements in no particular order, allowing fast retrieval of individual elements based on their value, much like unordered_set containers, but allowing different elements to have equivalent values.

  In an unordered_multiset, the value of an element is at the same time its *key*, used to identify it. Keys are immutable, therefore, the elements in an unordered_multiset cannot be modified once in the container - they can be inserted and removed, though.

  Internally, the elements in the unordered_multiset are not sorted in any particular, but organized into *buckets* depending on their hash values to allow for fast access to individual elements directly by their *values* (with a constant average time complexity on average).

  Elements with equivalent values are grouped together in the same bucket and in such a way that an iterator (see equal_range) can iterate through all of them.

Iterators in the container are at least forward iterators.

Notice that this container is not defined in its own header, but shares header <unordered_set> with unordered_set.

- Unordered_set class

  Unordered sets are containers that store unique elements in no particular order, and which allow for fast retrieval of individual elements based on their value.

  In an unordered_set, the value of an element is at the same time its *key*, that identifies it uniquely. Keys are immutable, therefore, the elements in an unordered_set cannot be modified once in the container - they can be inserted and removed, though.

  Internally, the elements in the unordered_set are not sorted in any particular order, but organized into *buckets* depending on their hash values to allow for fast access to individual elements directly by their *values* (with a constant average time complexity on average).

  unordered_set containers are faster than set containers to access individual elements by their *key*, although they are generally less efficient for range iteration through a subset of their elements.

  Iterators in the container are at least forward iterators.

- Vector class

  Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just

as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time*complexity (see push_back).

Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

Compared to the other dynamic sequence containers (deques, lists and forward_lists), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end. For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than listsand forward_lists.