STL Containers

Contents

1. Containers in C++ STL

Reasons to use C++ standard library:

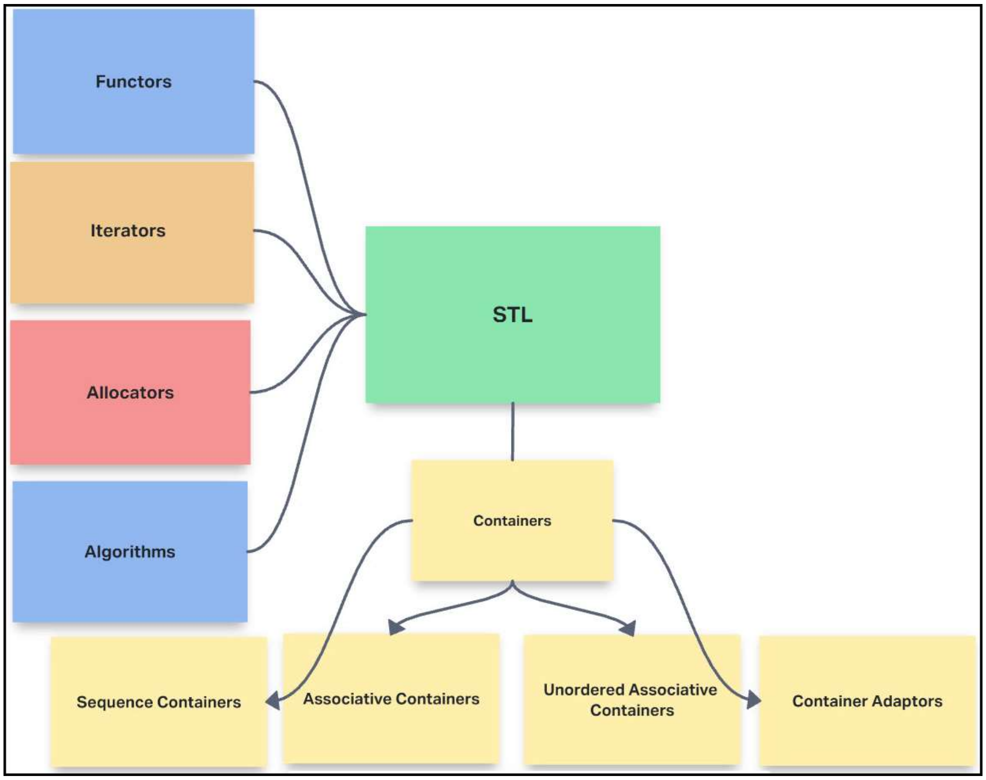
1. Code reuse, no need to re-invent the wheel.
2. Efficiency.
3. Accurate, less buggy.
4. Terse, readable code, reduced control flow.
5. standardization, guaranteed availability.
6. A role model of writing library.
7. Good knowledge of data structures and algorithm.

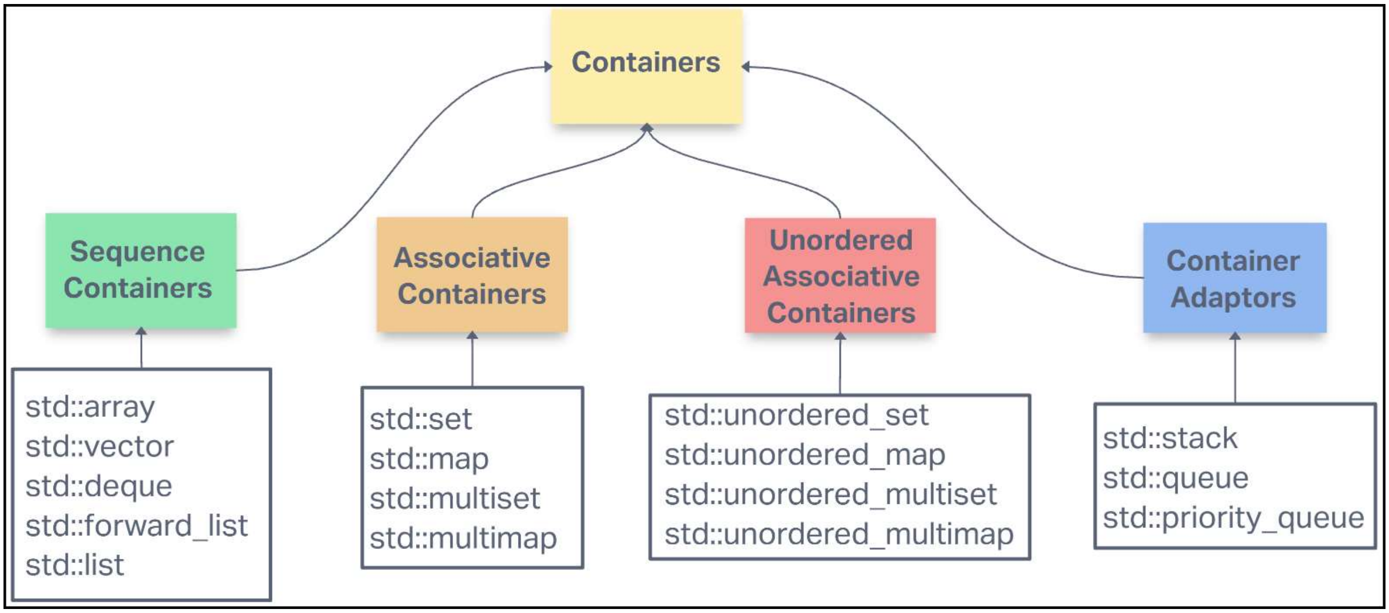
What is a container?

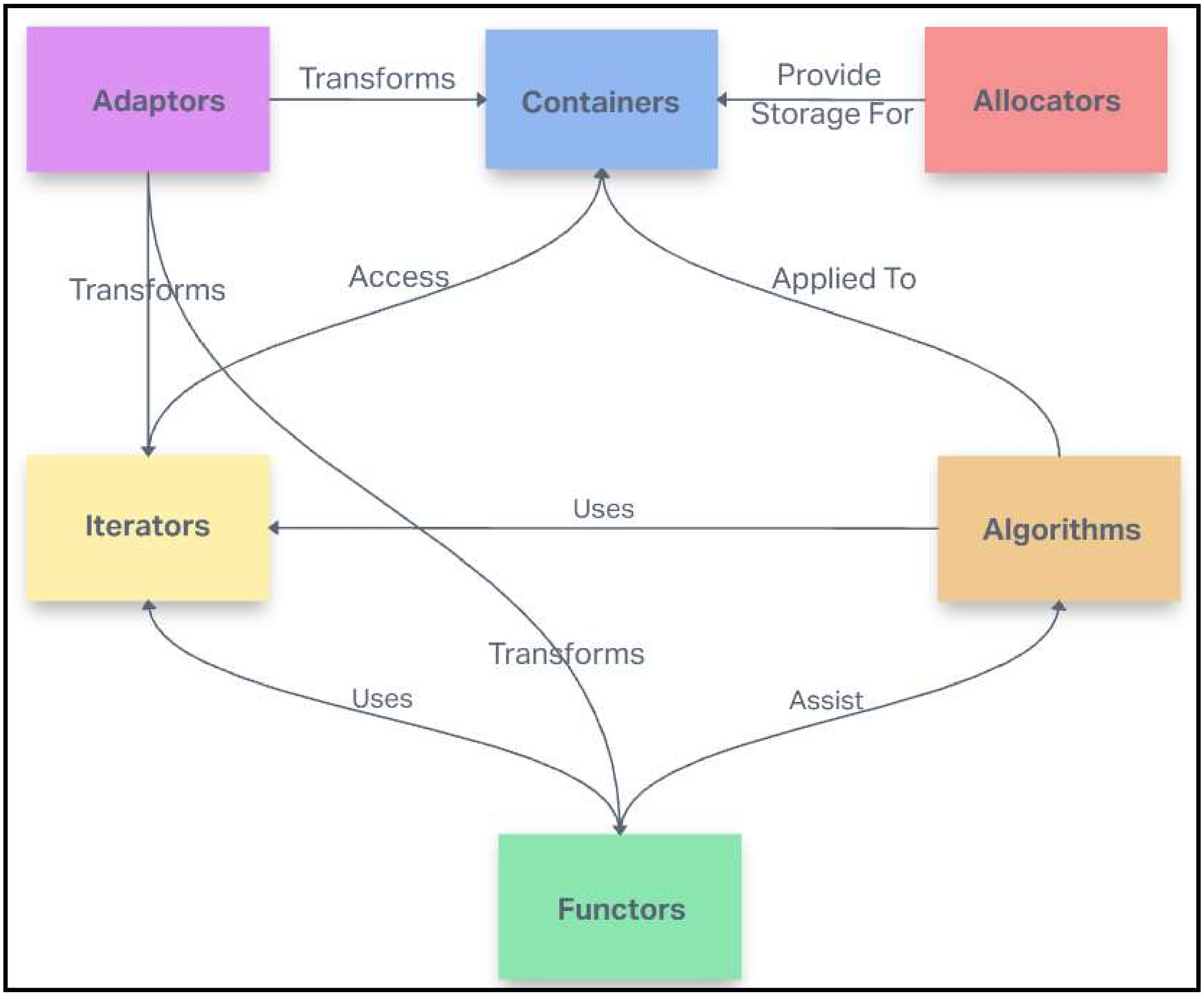
- A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows great flexibility in the types supported as elements.
- The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Types

- STL provides…
  I. Sequence containers.
  II. Associative containers.
  III. Unordered associative containers.
  IV. Container adaptors.




Containers
Sequence Containers
Associative Containers
Unordered Associative Containers
Container Adaptors
std::array
std::vector
std::deque
std::forward_list
std::list
std::set
std::map
std::multiset
std::multimap
std::unordered_set
std::unordered_map
std::unordered_multiset
std::unordered_multimap
std::stack
std::queue
std::priority_queue


Adaptors
Transforms
Containers
Provide Storage For
Allocators
Transforms
Access
Applied To
Iterators
Uses
Algorithms
Transforms
Uses
Assist
Functors

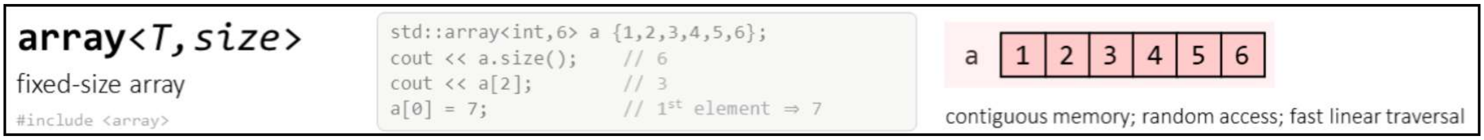
2. Sequence containers

- Sequence containers implement data structures that can be accessed sequentially.
- STL provides following sequence containers…

  I. `array`: Static contiguous array.
  II. `vector`: Dynamic contiguous array.
  III. `deque`: Double-ended queue.
  IV. `forward_list`: Singly-linked list.
  V. `list`: Doubly-linked list.

std::array

- Arrays are fixed-size sequence containers: they hold a specific number of elements ordered in a strict linear sequence.



array<*T*, *size*>

fixed-size array

#include <array>

```
std::array<int,6> a {1,2,3,4,5,6};
cout << a.size();    // 6
cout << a[2];        // 3
a[0] = 7;            // 1st element ⇒ 7
```

contiguous memory; random access; fast linear traversal

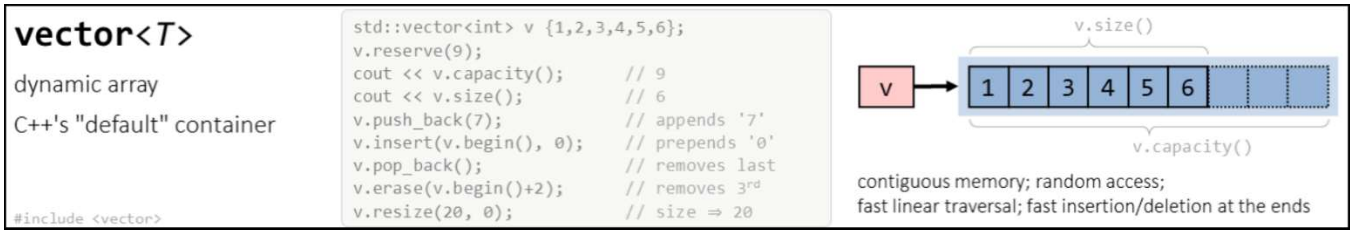
Element access		Complexity
at	Access specified element with bounds checking	O(1)
operator[]	Access specified element	O(1)
front	Access the first element	O(1)
back	Access the last element	O(1)
data	Direct access to the underlying array	O(1)
Capacity		
empty	Checks whether the container is empty	O(1)
size	Returns the number of elements	O(1)
max_size	Returns the maximum possible number of elements	O(1)
Operations		
fill	Fill the container with specified value	O(n)
swap	Swaps the contents	O(n)

Functionality	Static Array
Access	**O(1)**
Search	**O(n)**
Insertion	N/A
Appending	N/A
Deletion	N/A

```
int main(int argc, char* argv[]) {
        // int maxArray[1000000]; of Type 'int, double, char' inside a function.
        // Max size of 10^7 of type bool array in main().
        // Globally we can declare 10^7 max size array. 10^8 is max of bool type.
        array<int, 10> myArray;     // Garbage!
        myArray.fill(99);           // Fill all the elements with 99!.
        myArray.at(2) = 0;          // Access the 3rd element.

        // Iterators
        // begin(), end(), rbegin(), rend()
        /*
               begin                                           end
                 |                                              |
                 ▼                                              ▼
              ┌──────┬──────┬──────┬──────┬──────┬──────┬──────┐
              │      │      │      │      │      │      │      │
              └──────┴──────┴──────┴──────┴──────┴──────┴──────┘
           ▲                                              ▲
           |                                              |
          rend                                          rbegin
        */

        for (auto it = myArray.begin(); it != myArray.end(); it++) {
                cout << *it << " ";
        }
        cout << '\n';
        for (auto it = myArray.rbegin(); it != myArray.rend(); it++) {
                cout << *it << " ";
        }
        cout << '\n';

        cout << "Size: " << myArray.size() << endl;
        cout << "Front: " << myArray.front() << " Same as myArray[0]: "
                << myArray[0] << endl;
        cout << "Back: " << myArray.back() << " Same as myArray[myArray.size()-1]: "
                << myArray[myArray.size() - 1] << endl;

        return 0;
}
```

std::vector

- Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.
- Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.



vector<*T*>

dynamic array

C++'s "default" container

#include <vector>

```
std::vector<int> v {1,2,3,4,5,6};
v.reserve(9);
cout << v.capacity();      // 9
cout << v.size();          // 6
v.push_back(7);            // appends '7'
v.insert(v.begin(), 0);    // prepends '0'
v.pop_back();              // removes last
v.erase(v.begin()+2);      // removes 3rd
v.resize(20, 0);           // size ⇒ 20
```

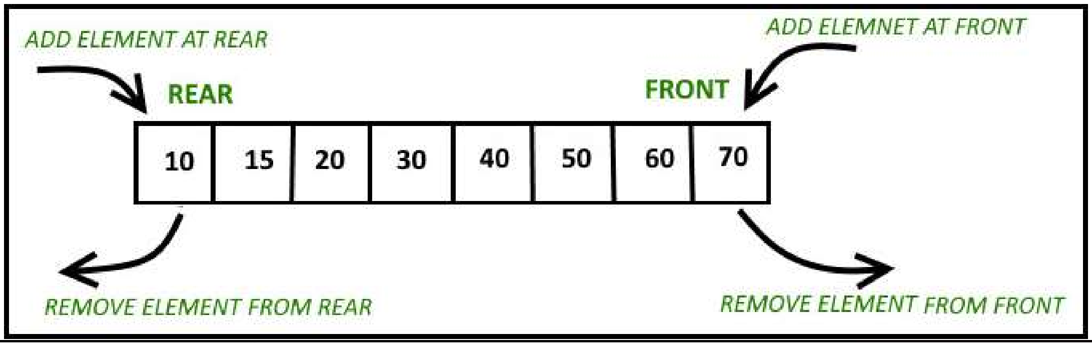
contiguous memory; random access;
fast linear traversal; fast insertion/deletion at the ends

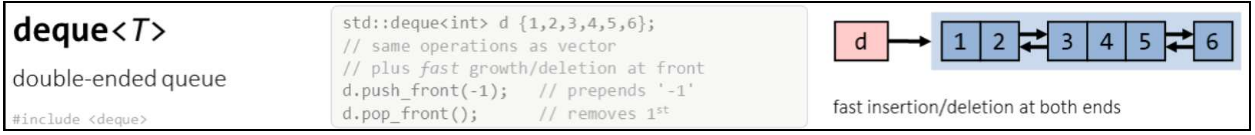
Vector		
Constructors		Complexity
vector<T> v;	Make an empty vector.	O(1)
vector<T> v(n);	Make a vector with N elements.	O(n)
vector<T> v(n, value);	Make a vector with N elements, initialized to value.	O(n)
vector<T> v(begin, end);	Make a vector and copy the elements from begin to end.	O(n)
Accessors		
v[i];	**Return (or set) the I'th element.**	**O(1)**
v.at(i);	**Return (or set) the I'th element, with bounds checking.**	**O(1)**
v.size();	**Return current number of elements.**	**O(1)**
v.empty();	**Return true if vector is empty.**	**O(1)**
v.begin();	**Return random access iterator to start.**	**O(1)**
v.end();	**Return random access iterator to end.**	**O(1)**
v.front();	**Return the first element.**	**O(1)**
v.back();	**Return the last element.**	**O(1)**
v.capacity();	**Return maximum number of elements.**	**O(1)**
Modifiers		
v.push_back(value);	**Add value to end.**	**O(1) (amortized)**
v.insert(iterator, value);	**Insert value at the position indexed by iterator.**	**O(n)**
v.pop_back();	**Remove value from end.**	**O(1)**
v.erase(iterator);	**Erase value indexed by iterator.**	**O(n)**
v.erase(begin, end);	Erase the elements from begin to end.	O(n)

```cpp
int main(int argc, char* argv[]) {
        vector<int> myVec;
        cout << "Empty Vector Size: " << myVec.size() << endl; // Prints 0

        myVec.push_back(0); // {0}
        myVec.push_back(2); // {0,2}
        cout << "After 2 elements, Size: " << myVec.size() << endl; // Prints 2

        myVec.pop_back(); // {0}
        cout << "After Pop, Size: " << myVec.size() << endl; // Prints 1

        myVec.push_back(3); // {0,3}
        myVec.push_back(4); // {0,3,4}
        myVec.push_back(5); // {0,3,4,5}
        myVec.clear();              // Erase all the elements and size is 0;
        cout << "After Clear, Size: " << myVec.size() << endl; // Prints 0

        vector<int> vec1(4, 2); // {2,2,2,2}
        vector<int> vec2(vec1.begin(), vec1.end()); // -> [)
        vector<int> vec3(vec2); // Copy construct.

        myVec.push_back(0); // {0}
        myVec.push_back(2); // {0,2}
        myVec.push_back(3); // {0,2,3}
        myVec.push_back(4); // {0,2,3,4}
        myVec.push_back(5); // {0,2,3,4,5}
        vector<int> vec4 (myVec.begin(), myVec.begin()+3); // vec4 -> {0,2,3}, cos [)
        for (auto i : vec4)
                cout << i << " ";
        cout << "\n";
        myVec.clear();

        // emplace_back will take less time than push_back
        myVec.emplace_back(1);
        myVec.emplace_back(2);
        myVec.emplace_back(3);
        myVec.emplace_back(4);
        myVec.emplace_back(5);
        myVec.emplace_back(6);

        cout << "myVec: ";   // myVec: 1 2 3 4 5 6
        for (auto i : myVec)
                cout << i << " ";
        cout << "\n";

        cout << "vec1: ";    // vec1: 2 2 2 2
        for (auto i : vec1)
                cout << i << " ";
        cout << "\n";
        cout << "After swap: \n";
        myVec.swap(vec1);

        cout << "myVec: ";   // myVec: 2 2 2 2
        for (auto i : myVec)
                cout << i << " ";
        cout << "\n";
        cout << "vec1: ";    // vec1 : 1 2 3 4 5 6
        for (auto i : vec1)
                cout << i << " ";
        cout << "\n";
        // 2D vector
        vector<vector<int>> my2DVec;
        my2DVec.push_back(myVec);
        my2DVec.push_back(vec1);
        my2DVec.push_back(vec2);
        my2DVec.push_back(vec3);

        // Defining 3X4 matrix. 3 Rows, 4 Columns, initialized with -1
        vector<vector<int>> my2DVec1(3, vector<int>(4, -1));
        for (auto i : my2DVec1) {
                for (auto j : i) {
                        cout << j << " ";
                }
                cout << "\n";
        }

        vector<int> arrOfVector[5]; // Array of 5 vector<int>

        return 0;
}
```

std::deque

- Double-ended queues are sequence containers with the feature of expansion and contraction on both ends.
- They are like vectors but are more efficient in case of insertion and deletion of elements. Unlike vectors, contiguous storage allocation may not be guaranteed.





deque<*T*>

double-ended queue

#include <deque>

```
std::deque<int> d {1,2,3,4,5,6};
// same operations as vector
// plus fast growth/deletion at front
d.push_front(-1);    // prepends '-1'
d.pop_front();       // removes 1st
```

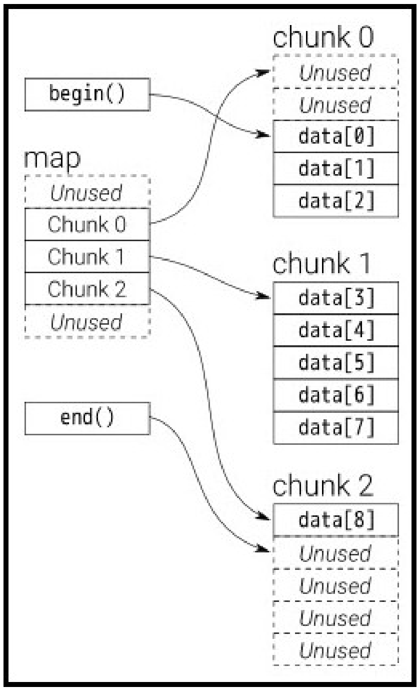
fast insertion/deletion at both ends

Deque		
Constructors		Complexity
`deque<T> d;`	Make an empty deque.	O(1)
`deque<T> d(n);`	Make a deque with N elements.	O(n)
`deque<T> d(n, value);`	Make a deque with N elements, initialized to value.	O(n)
`deque<T> d(begin, end);`	Make a deque and copy the values.	O(n)
Accessors		
`d[i];`	**Return (or set) the ith element.**	**O(1)**
`d.at(i);`	**Return (or set) the ith element, with bounds checking.**	**O(1)**
`d.size();`	**Return current number of elements.**	**O(1)**
`d.empty();`	**Return true if deque is empty.**	**O(1)**
`d.begin();`	Return random access iterator to start.	O(1)
`d.end();`	Return random access iterator to end.	O(1)
`d.front();`	Return the first element.	O(1)
`d.back();`	Return the last element.	O(1)
Modifiers		
`d.push_front(value);`	**Add value to front.**	**O(1) (amortized)**
`d.push_back(value);`	**Add value to end.**	**O(1) (amortized)**
`d.insert(iterator, value);`	**Insert value at the position indexed by iterator.**	**O(n)**
`d.pop_front();`	**Remove value from front.**	**O(1)**
`d.pop_back();`	**Remove value from end.**	**O(1)**
`d.erase(iterator);`	**Erase value indexed by iterator.**	**O(n)**
`d.erase(begin, end);`	**Erase the elements from begin to end.**	**O(n)**

How std::deque works internally?

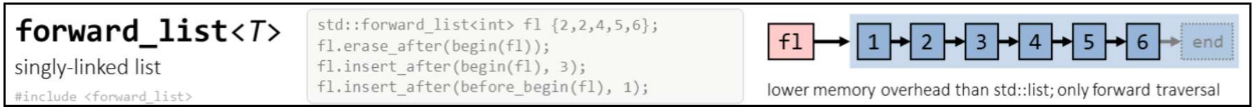
How deque is able to give good performance for insertion and deletion at both ends?

- A deque is generally implemented as a collection of memory blocks. These memory blocks contain the elements at contiguous locations.
- When we create a deque object it internally allocates a memory block to store the elements at contiguous location.
- When we insert an element at 'end' it stores that in allocated memory block until it gets filled and when this memory block gets filled with elements then it allocates a new memory block and links it with the 'end' of previous memory block. Now further inserted elements in the back are stored in this new memory block.
- When we insert an element in 'front' it allocates a new memory block and links it with the 'front' of previous memory block. Now further inserted elements in the 'front' are stored in this new memory block unless it gets filled.
- Consider deque as a linked list of vectors i.e. each node of this linked list is a memory block that store elements at contiguous memory location and each of the memory block node is linked with its previous and next memory block node.



std::forward_list

- Forward list implements singly linked list.
- Introduced from C++11, forward list is more useful than other containers in insertion, removal, and moving operations (like sort) and allow time constant insertion and removal of elements.
- It 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, thus increasing the storage space required to store each element.
- The drawback of a forward list is that it cannot be iterated backward and its individual elements cannot be accessed directly.



forward_list<*T*>

singly-linked list

#include <forward_list>

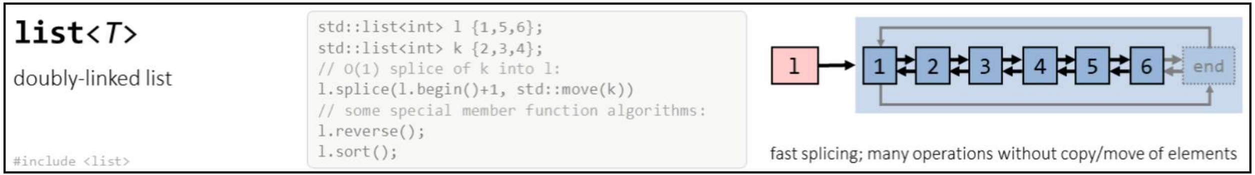
```
std::forward_list<int> fl {2,2,4,5,6};
fl.erase_after(begin(fl));
fl.insert_after(begin(fl), 3);
fl.insert_after(before_begin(fl), 1);
```

lower memory overhead than std::list; only forward traversal

Forward List		
Constructors		Complexity
`forward_list<T> l;`	Make an empty forward_list.	O(1)
`forward_list<T> l(begin, end);`	Make a forward_list and copy the values from begin to end.	O(n)
Accessors		
`l.size();`	No 'size()' member function.	N/A
`l.empty();`	Return true if forward_list is empty.	O(1)
`l.begin();`	Return iterator to beginning.	O(1)
`l.end();`	Return iterator to end.	O(1)
`l.front();`	Return the first element.	O(1)
Modifiers		
`l.push_front(value);`	Add value to front.	O(1)
`l.insert_after(iterator, value);`	Insert value after position indexed by iterator.	O(1)
`l.pop_front();`	Remove value from front.	O(1)
`l.pop_back();`	Remove value from end.	O(1)
`l.erase_after(iterator);`	Erase next value after the iterator.	O(1)
`l.erase_after(begin, end);`	Erase the elements from begin to end.	O(1)
`l.remove(value);`	Remove all occurrences of value.	O(n)
`l.remove_if(test);`	Remove all element that satisfy test.	O(n)
`l.reverse();`	Reverse the forward_list.	O(n)
`l.sort();`	Sort the forward_list.	O(n log n)
`l.sort(comparison);`	Sort with comparison function.	O(n logn)
`l.merge(l2);`	Merge sorted forward_lists.	O(n)

std::list

- `std::list` is a container that supports constant time insertion and removal of elements from anywhere in the container.
- Fast random access is not supported. It is usually implemented as a doubly linked list.
- Compared to `std::forward_list` this container provides bidirectional iteration capability while being less space efficient.



list<*T*>

doubly-linked list

```
std::list<int> l {1,5,6};
std::list<int> k {2,3,4};
// O(1) splice of k into l:
l.splice(l.begin()+1, std::move(k))
// some special member function algorithms:
l.reverse();
l.sort();
```

fast splicing; many operations without copy/move of elements

#include <list>

List		
Constructors		Complexity
`list<T> l;`	Make an empty list.	O(1)
`list<T> l(begin, end);`	Make a list and copy the values from begin to end.	O(n)
Accessors		
`l.size();`	Return current number of elements.	O(1)
`l.empty();`	Return true if list is empty.	O(1)
`l.begin();`	Return bidirectional iterator to start.	O(1)
`l.end();`	Return bidirectional iterator to end.	O(1)
`l.front();`	Return the first element.	O(1)
`l.back();`	Return the last element.	O(1)
Modifiers		
`l.push_front(value);`	**Add value to front.**	**O(1)**
`l.push_back(value);`	**Add value to end.**	**O(1)**
`l.insert(iterator, value);`	Insert value after position indexed by iterator.	O(1)
`l.pop_front();`	Remove value from front.	O(1)
`l.pop_back();`	Remove value from end.	O(1)
`l.erase(iterator);`	Erase value indexed by iterator.	O(1)
`l.erase(begin, end);`	Erase the elements from begin to end.	O(1)
`l.remove(value);`	Remove all occurrences of value.	O(n)
`l.remove_if(test);`	Remove all element that satisfy test.	O(n)
`l.reverse();`	Reverse the list.	O(n)
`l.sort();`	Sort the list.	O(n log n)
`l.sort(comparison);`	Sort with comparison function.	O(n log n)
`l.merge(l2);`	Merge sorted lists.	O(n)

Where are linked lists used?

- Used in many List, Queue & Stack implementations.
- Great for creating circular lists.
- Can easily model real world objects such as trains.
- Used in separate chaining, which is present certain Hash table implementations to deal with hashing collisions.
- Often used in the implementation of adjacency lists for graphs.

Terminology

- Head: The first node in a linked list
- Tail: The last node in a linked list
- Pointer: Reference to another node
- Node: An object containing data and pointer(s)

Complexity		
	Singly LL	Doubly LL
Access	**O(n)**	**O(n)**
Search	**O(n)**	**O(n)**
Insertion at Head	O(1)	O(1)
Insertion at Tail	O(1)	O(1)
Remove at Head	O(1)	O(1)
Remove at Tail	O(n)	O(1)
Remove in the Middle	**O(n)**	**O(n)**

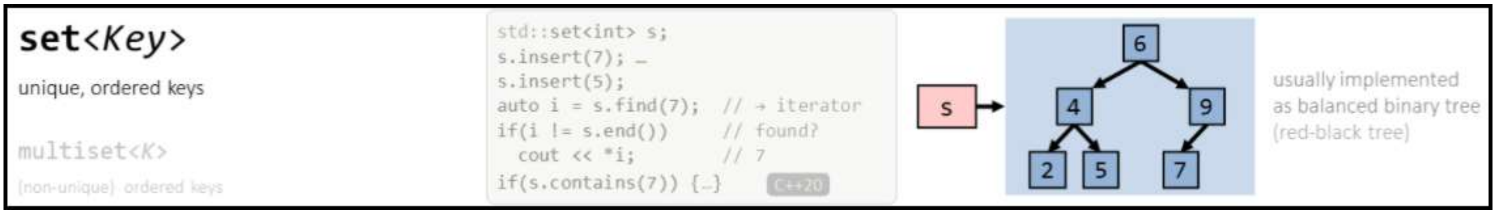
3. Associative containers

- Associative containers implement **sorted data structures** that can be quickly searched (O(log n) complexity).
- STL provides following associative containers…
  I. `set`: Collection of unique keys, sorted by keys.
  II. `map`: Collection of key-value pairs, sorted by keys, keys are unique.
  III. `multiset`: Collection of keys, sorted by keys.
  IV. `multimap`: Collection of key-value pairs, sorted by keys.

std::set and std::multiset

- std::set store objects and automatically keep them sorted and quick to find. In a set, there is only one copy of each object.
- std::multiset are declared and used the same as sets but allow duplicate elements.

set & multiset		
Constructors		Complexity
`set<type,compare> s;`	Make an empty set. compare should be a binary predicate for ordering the set. It's optional and will default to a function that uses operator<.	`O(1)`
`set<type,compare> s(begin, end);`	Make a set and copy the values from begin to end.	`O(n log n)`
Accessors		
`s.find(key)`	**Return an iterator pointing to an occurrence of key in s, or s.end() if key is not in s.**	**`O(log n)`**
`s.lower_bound(key)`	Return an iterator pointing to the first occurrence of an item in s not less than key, or s.end() if no such item is found.	`O(log n)`
`s.upper_bound(key)`	Return an iterator pointing to the first occurrence of an item greater than key in s, or s.end() if no such item is found.	`O(log n)`
`s.equal_range(key)`	Returns pair<lower_bound(key), upper_bound(key)>.	`O(log n)`
`s.count(key)`	Returns the number of items equal to key in s.	`O(log n)`
`s.size();`	Return current number of elements.	`O(1)`
`s.empty();`	Return true if set is empty.	`O(1)`
`s.begin()`	Return an iterator pointing to the first element.	`O(1)`
`s.end()`	Return an iterator pointing one past the last element.	`O(1)`
Modifiers		
`s.insert(iterator, key)`	Inserts key into s. iterator is taken as a "hint" but key will go in the correct position no matter what. Returns an iterator pointing to where key went.	`O(log n)`
`s.insert(key)`	**Inserts key into s and returns a pair<iterator, bool>, where iterator is where key went and bool is true if key was actually inserted, i.e., was not already in the set.**	**`O(log n)`**



set<*Key*>

unique, ordered keys

multiset<*K*>

(non-unique) ordered keys

```
std::set<int> s;
s.insert(7); …
s.insert(5);
auto i = s.find(7);  // → iterator
if(i != s.end())     // found?
  cout << *i;        // 7
if(s.contains(7)) {…}        C++20
```

usually implemented as balanced binary tree (red-black tree)

```
int main(int argc, char* argv[]) {
      // Given n elements, print only unique elements!
      set<int> s;
      cout << "Enter number of elements: ";
      int n; cin >> n;                            // 6

      cout << "Enter " << n << " elements: ";
      while (n--) {
            int ele; cin >> ele;                  // 1 1 2 2 4 4
            s.insert(ele);                        // Takes O(log n)
      }
      // Elements are unique and are also sorted!
      cout << "Unique elements are: "; //1 2 4
      for (auto i : s)
            cout << i << " ";
      cout << "\n";
      // Set doesn't provide the random access. s[2] is wrong.

      s.erase(s.begin()); // {2, 4} takes O(log n)
      s.insert(1);       // {1, 2, 4}
      s.insert(5);       // {1, 2, 4, 5}

      auto it = s.find(4);    // Takes O(log n)
      s.erase(s.begin(), it); // You cannot do s.erase(s.begin(), s.begin()+2)

      cout << "Elements are: "; // Elements are : 4 5 -> erases [)
      for (auto i : s)
            cout << i << " ";
      cout << "\n";

      s.erase(5); // Erase the key 5, takes O(log n)
      cout << "After deleting 5: "; // After deleting 5: 4
      for (auto i : s)
            cout << i << " ";
      cout << "\n";

      auto it = s.find(100);   // it -> s.end()
      s.emplace(9);            // Emplace is faster.
      return 0;
}
```

```
int main(int argc, char* argv[]) {
       multiset<int> ms;
       ms.insert(1); ms.insert(2); ms.insert(3); // Takes O(log n)
       ms.emplace(1); ms.emplace(3); // Same as insert. But faster!

       // 'multiset' stores duplicate elements in sorted order.
       cout << "Elements are: "; // Elements are: 1 1 2 3 3
       for (auto i : ms) cout << i << " ";

       cout << "\nErasing all 3's: ";
       ms.erase(3);  // Takes O(log n)
       cout << "\nElements are: "; // Elements are: 1 1 2
       for (auto i : ms) cout << i << " ";

       ms.emplace(3); ms.emplace(4);
       ms.emplace(3); ms.emplace(5);
       cout << "\nElements are: "; // Elements are: 1 1 2 3 3 4 5
       for (auto i : ms) cout << i << " ";

       // 'find' returns the iterator pointing to the first element.
       auto it = ms.find(3);       // Takes O(log n)
       ms.erase(it);               // Takes O(log n)
       cout << "\nElements are: "; // Elements are: 1 1 2 3 4 5
       for (auto i : ms) cout << i << " ";

       cout << "\nElement 1 occurs " << ms.count(1) << " times";

       return 0;
}
```

std::map and std::multimap

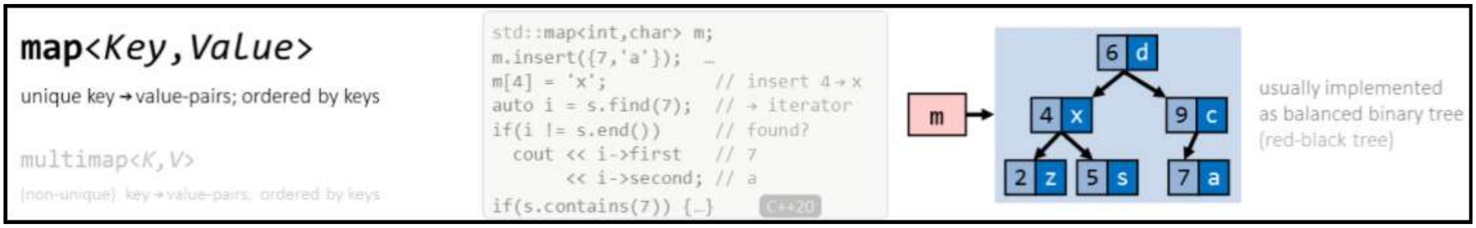
- Maps can be thought of as generalized vectors. They allow map[key] = value for any kind of key, not just integers.
- Maps are implemented with balanced binary search trees, typically red-black trees. Thus, they provide **logarithmic storage and retrieval times**. Because they use search trees, maps need a comparison predicate to sort the keys. `operator<()` will be used by default if none is specified a construction time.
- Maps store `<key, value>` pair's. That's what map iterators will return when dereferenced. To get the value pointed to by an iterator, you need to say `(*mapIter).second.` Usually though you can just use `map[key]` to get the value directly.

map & multimap		
Constructors		Complexity
`map<key_type,value_type,key_compare> m;`	Make an empty map. key_compare should be a binary predicate for ordering the keys. It's optional and will default to a function that uses operator<.	`O(1)`
`map<key_type,value_type,key_compare> m(begin, end);`	Make a map and copy the values from begin to end.	`O(n log n)`
Accessors		
`m[key]`	**Return the value stored for key. This adds a default value if key not in map.**	**`O(log n)`**
`m.find(key)`	**Return an iterator pointing to a key-value pair, or m.end() if key is not in map.**	**`O(log n)`**
`m.lower_bound(key)`	Return an iterator pointing to the first pair containing key, or m.end() if key is not in map.	`O(log n)`
`m.upper_bound(key)`	Return an iterator pointing one past the last pair containing key, or m.end() if key is not in map.	`O(log n)`
`m.equal_range(key)`	Return a pair containing the lower and upper bounds for key. This may be more efficient than calling those functions separately.	`O(log n)`
`m.size();`	**Return current number of elements.**	**`O(1)`**
`m.empty();`	**Return true if map is empty.**	**`O(1)`**
`m.begin()`	Return an iterator pointing to the first pair.	`O(1)`
`m.end()`	Return an iterator pointing one past the last pair.	`O(1)`
Modifiers		
`m[key] = value;`	**Store value under key in map.**	**`O(log n)`**
`m.insert(pair)`	**Inserts the <key, value> pair into the map. Equivalent to the above operation.**	**`O(log n)`**

Warning: `map[key]` creates a dummy entry for key if one wasn't in the map before. Use `map.find(key)` if you don't want this to happen.

- `multimaps` are like map except that they allow duplicate keys.

- **map[key] is not defined for multimaps.** Instead you use lower_bound() and upper_bound(), or equal_range(), to get the iterators for the beginning and end of the range of values stored for the key.



map<*Key,Value*>

unique key → value-pairs; ordered by keys

multimap<*K,V*>

(non-unique) key → value-pairs, ordered by keys

```
std::map<int,char> m;
m.insert({7,'a'}); …
m[4] = 'x';           // insert 4→x
auto i = s.find(7);   // → iterator
if(i != s.end())      // found?
  cout << i->first    // 7
       << i->second;  // a
if(s.contains(7)) {…}    C++20
```

usually implemented as balanced binary tree (red-black tree)

```
int main(int argc, char* argv[]) {
	map<string, int> mp;

	mp["Nishi"] = 39;    // Takes O(log n)
	mp["Rachhu"] = 35; mp["Gattu"] = 10;
	mp["Rajanna"] = 70; mp["Prabha"] = 65;
	mp["Rachhu"] = 36;   // Overwrites previous value.

	for (auto i : mp)    // Returns a pair!!!
		cout << "Name: " << i.first << " Age: " << i.second << endl;
	/* Map stores unique elements sorted order based on key.
		Name: Gattu Age: 10
		Name: Nishi Age: 39
		Name: Prabha Age: 65
		Name: Rachhu Age: 36
		Name: Rajanna Age: 70
	*/
	mp.emplace("Dummy", 99); // Takes O(log n), "Dummy" will be on top! (Sorted)
	mp.erase(mp.begin());    // "Dummy" will be erased. Takes O(log n)
	mp["Apple"] = 10;
	mp.erase("Apple");   // Takes O(log n)

	auto it = mp.find("Rajanna");      // Takes O(log n)
	if (it != mp.end())
		cout << "Found " << it->first << endl;

	if (!mp.empty()) cout << "Map is not empty!\n";

	mp.clear(); // Clears map.

	return 0;
}
```

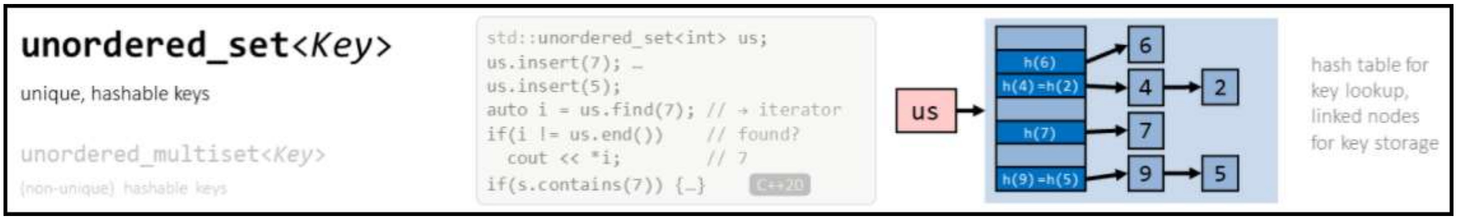
4. Unordered associative containers

- Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched (`O(1)` amortized, `O(n)` worst-case complexity).
- STL provides following unordered associative containers…
    1. `unordered_set`: Collection of unique keys, hashed by keys.
    2. `unordered_map`: Collection of key-value pairs, hashed by keys, keys are unique.
    3. `unordered_multiset`: Collection of keys, hashed by keys.
    4. `unordered_multimap`: Collection of key-value pairs, hashed by keys.

std::unordered_set

- An unordered_set is implemented using a hash table where keys are **hashed** into indices of a hash table so that the insertion is always **randomized**.
- All operations on the unordered_set takes constant time `O(1)` on an average which can go up to linear time `O(n)` in worst case which depends on the internally used hash function.

unordered_set		
Constructors		Complexity
`unordered_set<type,compare> s;`	Make an empty set. compare should be a binary predicate for ordering the set. It's optional and will default to a function that uses operator<.	`O(1)`
`unordered_set<type,compare> s(begin, end);`	Make a set and copy the values from begin to end.	`O(n)`
Accessors		
`s.find(key)`	Finds element with specific key	`O(1)`
`s.count(key)`	Returns the number of elements with key that compares equal to the specified argument key, which is either 1 or 0 since this container does not allow duplicates.	`O(1)`
`s.contains(key)`	Checks if the container contains element with specific key	`O(1)`
`s.size();`	Return current number of elements.	`O(1)`
`s.empty();`	Return true if set is empty.	`O(1)`
`s.begin()`	Return an iterator pointing to the first element.	`O(1)`
`s.end()`	Return an iterator pointing one past the last element.	`O(1)`
Modifiers		
`s.insert(iterator, key)`	Inserts value.	`O(1) (amortized)`
`s.erase(it);`	Removes specified elements from the container.	`O(1) (amortized)`
`s.emplace(key)`	Insert an element in an unordered_set container.	`O(1) (amortized)`



unordered_set<*Key*>

unique, hashable keys

unordered_multiset<*Key*>

(non-unique) hashable keys

```
std::unordered_set<int> us;
us.insert(7); …
us.insert(5);
auto i = us.find(7); // → iterator
if(i != us.end())    // found?
  cout << *i;        // 7
if(s.contains(7)) {…}    C++20
```

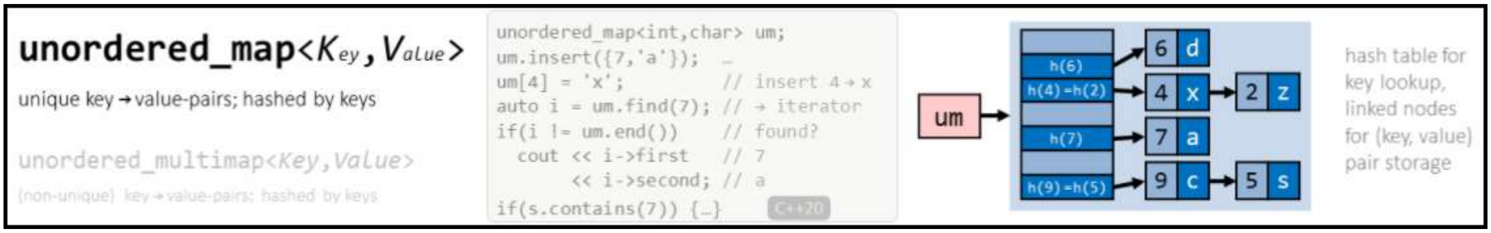
hash table for key lookup, linked nodes for key storage

```
int main(int argc, char* argv[]) {
        unordered_set<int> s;
        s.insert(4);  // Takes O(1)
        s.insert(2);
        s.insert(4);
        s.insert(1);

        // Stores unique elements but order is not guaranteed.
        // Whereas 'set' stores the elements in ascending order by default.
        cout << "Unique elements are: "; // 4 2 1
        for (auto i : s)
                cout << i << " ";
        cout << "\n";

        // All other operations such as 'find', 'erase' takes O(1)
        s.erase(s.begin()); // Takes O(1)
        auto it = s.find(1);// Takes O(1)
        if (it != s.end())
                cout << "1 is present in the set!\n";
        s.emplace(3);       // Takes O(1)

        cout << "Size: " << s.size() << endl; // Takes O(1)
        if (!s.empty())             // Takes O(1)
                cout << "Set is not empty!\n";

        return 0;
}
```

std::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.
- 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.
- The `unordered_map<int, int> M` is the implementation of Hash Table which makes the complexity of operations like `insert`, `delete` and `search` to amortized `O(1)`.

```
int main(int argc, char* argv[]) {
        unordered_map<string, int> ump;

        ump["Nishi"] = 39;   // Takes O(1)
        ump["Rachhu"] = 35; ump["Gattu"] = 10;
        ump["Rajanna"] = 70; ump["Prabha"] = 65;
        ump["Rachhu"] = 36;  // Overwrites previous value.

        for (auto i : ump)   // Returns a pair!!!
                cout << "Name: " << i.first << " Age: " << i.second << endl;
        /* Map stores unique elements sorted order based on key.
                Name: Nishi Age: 39
                Name: Rajanna Age: 70
                Name: Gattu Age: 10
                Name: Rachhu Age: 36
                Name: Prabha Age: 65
        */
        ump.emplace("Dummy", 99); // Takes O(1)
        ump.erase(ump.begin());   // Takes O(1), Don’t know which one is erased!
        ump["Apple"] = 10;
        ump.erase("Apple");        // Takes O(1)

        auto it = ump.find("Rajanna");     // Takes O(1)
        if (it != ump.end())
                cout << "Found " << it->first << endl;

        if (!ump.empty()) cout << "Map is not empty!\n";

        ump.clear(); // Clears map.

        return 0;
}
```



unordered_map<*Key*, *Value*>

unique key → value-pairs; hashed by keys

unordered_multimap<*Key*,*Value*>

(non-unique) key → value-pairs; hashed by keys

```
unordered_map<int,char> um;
um.insert({7,'a'});  …
um[4] = 'x';           // insert 4 → x
auto i = um.find(7);  // → iterator
if(i != um.end())     // found?
  cout << i->first     // 7
       << i->second;  // a
if(s.contains(7)) {…}     C++20
```

hash table for key lookup, linked nodes for (key, value) pair storage

std::unordered_multiset

- Similar to unordered_set, but allows duplicate elements.
- The internal implementation of unordered_multiset is same as that of unordered_set and also uses hash table for searching, just the count value is associated with each value in former one.
- All operation on unordered_multiset takes constant time O(1) on average but can go up to linear in worst case.

std::unordered_multimap

- Similar to unordered_map, but allows duplicate elements.
- The internal implementation of unordered_multimap is the same as that of unordered_map but for duplicate keys, another count value is maintained with each key-value pair.
- All operation on unordered_multimap takes a constant amount of time O(1) on an average but time can go to linear in the worst case depending on internally used hash function.

5. Container adaptors

- Container adaptors provide a different interface for sequential containers.
- STL provides following container adaptors…
  I. `stack`: Adapts a container to provide stack (LIFO data structure).
  II. `queue`: Adapts a container to provide queue (FIFO data structure).
  III. `priority_queue`: Adapts a container to provide priority queue.

std::stack

What is a Stack?

- A `stack` is a one—ended linear data structure which models a real-world stack by having two primary operations, namely push and pop.
- The `std::stack` class is a container adaptor that gives the programmer the functionality of a stack - specifically, a LIFO (last-in, first-out) data structure.
- The default underlying container is `deque`.

When and where is a stack used?

- Used by undo mechanisms in text editors.
- Used in compiler syntax checking for matching brackets and braces.
- Can be used to model a pile of books or plates.
- Used behind the scenes to support recursion by keeping track of previous function calls.
- Can be used to do a Depth First Search (DFS) on a graph.

Complexity	
Pushing	O(1)
Popping	O(1)
Peeking	O(1)
Searching	O(n)
Size	O(1)

```
int main(int argc, char* argv[]) {

       stack<int> dStack; // By default, underlying container is deque.
       dStack.push(1); dStack.push(2);
       dStack.push(3); dStack.push(4);

       while (!dStack.empty()) {
              cout << ' ' << dStack.top();
              dStack.pop();
       } // 4 3 2 1
       cout << '\n';

       stack<int, vector<int>> vStack; // Underlying container is now vector.
       vStack.push(1); vStack.push(2);
       vStack.push(3); vStack.push(4);

       while (!vStack.empty()) {
              cout << ' ' << vStack.top();
              vStack.pop();
       }
       cout << '\n';
       return 0;
}
```

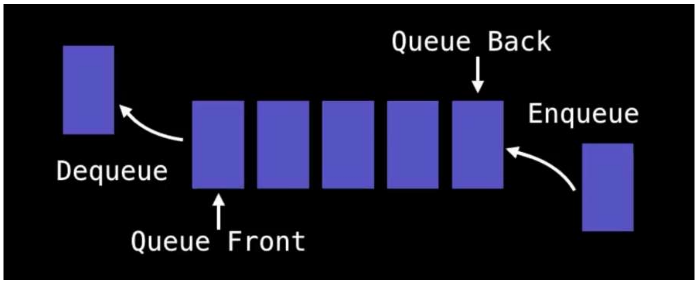
- Check for Balanced Brackets in an expression using stack.

```
char getRevrseBracket(char ch){
       if (ch == ']') return '[';
       if (ch == '}') return '{';
       if (ch == ')') return '(';
}

bool isLeftBracket(char ch) {
       return ch == '[' || ch == '{' || ch == '(';
}

bool areBracketsValid(string str){
       stack<char> s;
       for(auto ch: str)    {

              auto revCh = getRevrseBracket(ch);
              if (isLeftBracket(ch))
                     s.push(ch);
              else if (s.empty() || s.top() != getRevrseBracket(ch))
                     return false;
              else
                     s.pop();
       }
       return s.empty();
}

int main(int argc, char* argv[]) {
       string s1 = "{[]{}[](){({[]})}";
       cout << boolalpha << areBracketsValid(s1) << endl;
       return 0;
}
```

std::queue

What is a Queue?

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



When and where is a Queue used?

- Any waiting line models a queue, for example a line-up at a movie theatre.
- Can be used to efficiently keep track of the x most recently added elements.
- Web server request management where you want first come first serve.
- Breadth first search (BFS) graph traversal.

std::queue		
Operation		Complexity
q.push(x)	Add element to the back of the queue.	O(1)
q.pop( )	Remove element from the front of the queue.	O(1)
q.front( )	Look at the value at the front of the queue.	O(1)
q.back( )	Look at the value at the back of the queue.	O(1)
q.empty( )	Check if queue is empty or not.	O(1)
q.size( )	Return size	O(1)

```
int main(int argc, char* argv[]) {

        queue<int> dQueue; // By default, underlying container is deque.
        dQueue.push(1); dQueue.push(2);
        dQueue.push(3); dQueue.push(4);

        while (!dQueue.empty()) {
                cout << ' ' << dQueue.front();
                dQueue.pop();
        } // 1 2 3 4
        cout << '\n';

        queue<int, list<int>> vQueue; // Underlying container is now list.
        vQueue.push(1); vQueue.push(2);
        vQueue.push(3); vQueue.push(4);

        while (!vQueue.empty()) {
                cout << ' ' << vQueue.front();
                vQueue.pop();
        }
        cout << '\n';
        return 0;
}
```

std::priority_queue

What is a Priority Queue?

- A priority queue is a container adaptor that provides constant time lookup of the **largest** (by default) element, at the expense of **logarithmic insertion and extraction**.
- The priority of the elements in the priority queue determine the order in which elements are removed from the PQ.
- Priority Queue is used in many popular algorithms. Priority Queue is the implementation of **Max Heap** by default.

NOTE: Priority queues only supports comparable data, meaning the data inserted into the priority queue must be able to be ordered in some way either from least to greatest or greatest to least. This is so that we are able to assign relative priorities to each element.

std::priority_queue		
Operation		Complexity
`Q.top()`	accesses the top element	`O(1)`
`Q.push()`	inserts element and sorts the underlying container	`O(log n)`
`Q.pop()`	removes the top element	`O(log n)`
`Q.empty()`	checks whether the underlying container is empty	`O(1)`

```cpp
template<typename T>
void print_queue(T q) {
       while (!q.empty()) {
              cout << q.top() << ' ';
              q.pop();
       }
       cout << '\n';
}

struct Comparator {
       bool operator()(int a, int b) { return a > b; }
};
int main(int argc, char* argv[]) {
       const auto data = { 1,8,5,6,3,4,0,9,7,2 };
       priority_queue<int> q;
       for (int n : data)
              q.push(n);
       print_queue(q); // 9 8 7 6 5 4 3 2 1 0

       // Underlying container is vector and we have used custom comparator!.
       priority_queue<int, vector<int>, Comparator> q1(data.begin(),data.end());
       print_queue(q1); // 0 1 2 3 4 5 6 7 8 9

       return 0;
}
```

6. Time Complexity Table

	Insertion			Erase			Access	Find
Container	Front	Middle	Back	Front	Middle	Back		
std::vector	O(n)	O(n)	**O(1)**	O(n)	O(n)	**O(1)**	**O(1)**	O(n)
std::deque	**O(1)**	O(n)	**O(1)**	**O(1)**	O(n)	**O(1)**	**O(1)**	O(n)
std::forward_list	**O(1)**	**O(1)**	**O(1)**	**O(1)**	**O(1)**	**O(1)**	N/A	O(n)
std::list	**O(1)**	**O(1)**	**O(1)**	**O(1)**	**O(1)**	**O(1)**	N/A	O(n)
std::set	O(log n)			O(log n)			N/A	O(log n)
std::multiset	O(log n)			O(log n)			N/A	O(log n)
std::map	O(log n)			O(log n)			O(log n)	O(log n)
std::multimap	O(log n)			O(log n)			O(log n)	O(log n)
std::unordered_set	**O(1)**			**O(1)**			N/A	**O(1)**
std::unordered_multiset	**O(1)**			**O(1)**			N/A	**O(1)**
std::unordered_map	**O(1)**			**O(1)**			**O(1)**	**O(1)**
std::unordered_multimap	**O(1)**			**O(1)**			**O(1)**	**O(1)**