**C++ STL: CHEAT-SHEET**

|  | **array** | **vector** | **deque** | **set** | **multiset** | **map** | **multimap** | **unordered\_set** | **unordered\_multiset** | **unordered\_map** | **unordered\_multimap** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **begin()** | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| **end()** | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| **empty()** | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| **size()** | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| **insert()** | - | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| **erase()** | - | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| **count()** | - | - | - | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| **find()** | - | - | - | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| **push\_back()** | - | ✔ | ✔ | - | - | - | - | - | - | - | - |
| **pop\_back()** | - | ✔ | ✔ | - | - | - | - | - | - | - | - |
| **push\_front()** | - | - | ✔ | - | - | - | - | - | - | - | - |
| **pop\_front()** | - | - | ✔ | - | - | - | - | - | - | - | - |

|  | **stack** | **queue** | **priority\_queue** |
| --- | --- | --- | --- |
| **top()** | ✔ | - | ✔ |
| **front()** | - | ✔ | - |
| **back()** | - | ✔ | - |
| **empty()** | ✔ | ✔ | ✔ |
| **size()** | ✔ | ✔ | ✔ |
| **push()** | ✔ | ✔ | ✔ |
| **pop()** | ✔ | ✔ | ✔ |
| **insert(), erase(), begin(), end(), count(), find()** | - | - | - |

**POINTS TO REMEMBER:**

1. [**Array**](https://en.cppreference.com/w/cpp/container/array)**:**

* Typical array properties.

1. [**Vector**](https://en.cppreference.com/w/cpp/container/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.
* In vectors, data is inserted at the end. (push\_back())
* Inserting at the end takes different time, as sometimes there may be a need of extending the array.
* Removing the last element takes only constant time because no resizing happens(pop\_back()).
* Inserting and erasing at the beginning(push\_front(), pop\_front()) or in the middle is linear in time.
* Elements are indexed and hence random access of elements is possible using the operator[].
* reserve(n) will reserve n-sized vector or gives a vector with minimum n-size without reallocation.

| **#include <vector> // Include vector (std namespace)**  **#include <algorithm> // For sort(), reverse() etc...**  **vector<int> a(10, 0); // vector of size 10 all initialized with 0**  **vector<int> b{1,2,3}; // Create vector with values 1,2,3**  **a.size(); // Number of elements (10)**  **a.push\_back(3); // Increase size to 11, a[10]=3**  **a.pop\_back(); // Decrease size by 1**  **auto it = find(a.begin(), a.end(), 3) // returns an iterator pointing to 3 in the vector**  **int \_count = count(a.begin(),a.end(), 3); // count the number of 3’s in the vector**  **cout<< a.size(); // returns the total count of number of elements in the vector**  **a.erase(it); // delete the element pointed by the iterator**  **cout<<a.empty(); // returns true if a has 0 elements**  **a[20]=1; // Crash: not bounds checked**  **a.insert(a.begin()+2, 5); // inserts 5 at 2nd position from front**  **a.insert(a.begin()+3,{5,6,8}); // inserts 5,6,8 at 3rd position from front**  **vector<int> x = a; // copy - O(n)**  **for (int& p : a)**  **p=0; // C++11: Traversal**  **for (vector<int>::iterator p=a.begin(); p!=a.end(); ++p)**  **\*p=0; // Traversal. Can also use auto.**  **vector<int> b(a.begin(), a.end()); // b is copy of a**  **sort(a.begin(), a.end()); // sort the vector in ascending order**  **sort(a.begin(), a.end(), greater<int>); // sort the vector in descending order**  **vector<pair<int, int>> v;**  **v.push\_back(make\_pair(2,10));**  **v.push\_back(make\_pair(3,5));**  **v.push\_back(make\_pair(5,45));**  **static bool cmp(pair<int, int>& x, pair<int, int>& y)**  **{**  **return x.second > y.second; }**  **// custom sorting: sort the pairs in descending order of second element of the pair**  **sort(v.begin(), v.end(), cmp);**  **reverse(v.begin(), v.end()); // reverse the whole vector** |
| --- |

1. [**Deque**](https://en.cppreference.com/w/cpp/container/deque)**:**

* Double ended queues are a special case of queues where insertion and deletion operations are possible at both the ends.
* The functions for deque are same as vector, with an addition of push and pop operations for both front and back, which all take place in constant time.
* Implemented as a dynamic array.
* Elements are indexed and hence random access is possible using the operator[].

| **#include <deque> // Include deque (std namespace)**  **deque<int> a;**  **int x = 10**  **a.push\_front(x); // Puts x at a[0], shifts elements toward back**  **a.pop\_front(); // Removes a[0], shifts toward front**  **// use of rest all the interfaces remain consistent with that of the vector.** |
| --- |

1. [**Set**](https://en.cppreference.com/w/cpp/container/set)**:**

* Each element has to be unique, because the value of the element identifies it.
* The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.
* Elements are not indexed, but a set needs to be traversed sequentially using the iterators and referencing it to access the element.
* Implemented using a balanced binary search tree and hence the basic operations on them have logarithmic time complexity.
* Elements are inherently sorted in increasing order.

| **#include <set> // Include set (std namespace)**  **set<int> s; // Set of integers, default in ascending order**  **set<int, greater<int>> s; // Set of integers, arranged in descending order**  **set<int> x = s // copy - O(n)**  **s.insert(123); // Add element to set**  **cout << s.count(123); // returns 1, if element is present. Returns 0 if not present**  **if (s.find(123) != s.end()) // Search for an element: Returns s.end() if not present**  **s.erase(123); // Delete an element**  **if(!s.empty()) // if set is not empty**  **cout << s.size(); // Number of elements in set**  **for (auto itr = s.begin(); itr != s.end(); itr++)**  **cout << \*itr<<" "; // Traversal**  **for (auto& p:s)**  **cout << p << “ “; // Traversal** |
| --- |

1. [**Multiset**](https://en.cppreference.com/w/cpp/container/multiset)**:**

* Multisets have the same properties of sets, except the uniqueness property i.e. elements can have duplicates in the container.

| **#include <set> // Include set (std namespace)**  **multiset<int> s; // MultiSet of integers, default in ascending order**  **multiset<int, greater<int>> // MultiSet of integers, arranged in descending order**  **// use of rest all the interfaces remain consistent with that of the set.**  **// count here, will return 0 or count of the number of such elements instead of always // 1 if not 0.** |
| --- |

1. [**Map**](https://en.cppreference.com/w/cpp/container/map)**:**

* Contains key-value pairs in sorted order of keys.
* Keys have to be unique.
* Implemented using a balanced binary search tree and hence the basic operations on them have logarithmic time complexity.
* Random access of values is possible. Values can be accessed and modified using the key with the help of operator[] and operator= respectively.
* Can be traversed sequentially using a iterator and key can be obtained using ‘first’ member of the iterator(Eg: it.first or it->first) and value using the ‘second’ member of the iterator(Eg: it.second or it->second).

| **#include <map> // Include map (std namespace)**  #include <algorithm> // For sort(), reverse() etc...  **map<string, int> a; // Map from string to int**  **map<string, int, greater<int>> a1; // descending order**  **map<string, int> mp = a; // copy - O(n)**  **a["hello"] = 3; // Add or replace element a["hello"]**  **for (auto& p:a)**  **cout << p.first << p.second; // Traversal. Prints hello, 3**  **for (auto itr = a.begin(); itr != a.end(); itr++)**  **cout << itr->first<<" "<<itr->second; // Traversal**  **a.insert({“world”, 5}); // Insert a pair into the map**  **a.insert(make\_pair(“hey”, 6)); // Insert a pair into the map**  **cout << s.count(“world”); // returns 1, if the key is present. Returns 0 if not present**  **if (s.find(“hello”) != s.end()) // Search for an element: Returns s.end() if not present**  **s.erase(“hello”); // Delete an element**  **if(!s.empty()) // if set is not empty**  **cout << s.size(); // Number of elements in set**  **// Sorting the map mp, using vector of pairs**  **vector<pair<char, int>> v;**  **static bool cmp(pair<char, int>& x, pair<char, int>& y)**  **return x.second > y.second;**  **for(auto pair\_: mp)**  **v.push\_back(pair\_);**  **sort(v.begin(), v.end(), cmp);** |
| --- |

1. [**Multimap**](https://en.cppreference.com/w/cpp/container/multimap)**:**

* Multimaps have the same properties of maps, except the uniqueness property i.e. keys can have duplicates in the container.

| **#include <map> // Include map (std namespace)**  **multimap<string, int> s; // MultiSet of integers, default in ascending order**  **multimap<string, int, greater<int>> // Multimap of integers, arranged in descending //order.**  **// use of rest all the interfaces remain consistent with that of the map.**  **// however count() here, will return 0 or count of the number of such keys instead of //always 1 if not 0.** |
| --- |

1. [**Unordered\_set**](https://en.cppreference.com/w/cpp/container/unordered_set)**:**

* No internal ordering of values unlike a set, in an unordered\_set.
* Implemented using a hash table where elements are organized into buckets.
* Hence the average time complexity of basic operations is constant. But linear time complexity in the worst case.
* The above 3 points distinguish an unordered set when compared to a set. The remaining properties of sets hold true for unordered\_sets as well.

| **#include <unordered\_set> // Include unordered\_set (std namespace)**  **unordered\_set<int> s; // Unordered set of integers, no ordering.**  **// use of rest all the interfaces remain consistent with that of the set.** |
| --- |

1. [**Unordered\_multiset**](https://en.cppreference.com/w/cpp/container/unordered_multiset)**:**

* Unordered\_Multisets have the same properties of unordered\_sets, except the uniqueness property i.e. values can have duplicates in the container.

| **#include <unordered\_set> // Include unordered\_set (std namespace)**  **unordered\_multiset<int> s; // Unordered multiset of integers, no ordering.**  **// use of rest all the interfaces remain consistent with that of the multiset.** |
| --- |

1. [**Unordered\_map**](https://en.cppreference.com/w/cpp/container/unordered_map)**:**

* Stores key-value pairs but there is no internal ordering of keys unlike a set, in an unordered\_set.
* Implemented using a hash table where elements are organized into buckets, each identified by a unique key.
* Hence the average time complexity of basic operations is constant. But linear time complexity in the worst case.
* The above 3 points distinguish an unordered\_map when compared to a map. The remaining properties of maps hold true for unordered\_maps as well.

| **#include <unordered\_map> // Include unordered\_map (std namespace)**  **unordered\_map<string, int> s; // Unordered map of pair of string and integers, no ordering.**  **// use of rest all the interfaces remain consistent with that of the map.** |
| --- |

1. [**Unordered\_multimap**](https://en.cppreference.com/w/cpp/container/unordered_multimap)**:**

* **Unordered\_Multimaps have the same properties of unordered\_maps, except the uniqueness property i.e. keys can have duplicates in the container.**

| **#include <unordered\_map> // Include unordered\_map (std namespace)**  **// Unordered multimap of pair of string and integers, no order**  **unordered\_multimap<string, int> s;**  **// use of rest all the interfaces remain consistent with that of the multimap.** |
| --- |

1. [**Pair**](https://www.geeksforgeeks.org/pair-in-cpp-stl/)**:**

* The first element is referenced as ‘first’ and the second element as ‘second’ and the order is fixed (first, second).
* Pair can be assigned, copied and compared.
* The array of objects allocated in a map or hash\_map are of type ‘pair’ by default in which all the ‘first’ elements are unique keys associated with their ‘second’ value objects.

| **#include <utility> // Include utility (std namespace)**  **pair<string, int> a("hello", 3); // A 2-element struct**  **cout << a.first; // "hello"**  **cout << a.second; // 3** |
| --- |

1. [**Stack**](https://www.geeksforgeeks.org/stack-in-cpp-stl/)**:**

* Stack uses an encapsulated object of either vector or deque(by default) or list as its underlying container, but providing a specific set of member functions to access its elements..
* LIFO: We can insert an element only to the top of the stack(push) and remove an element from the same end(pop) one at a time. We can push, pop and access the top of the stack in constant time.

| **#include <stack> // Include stack (std namespace)**  **stack<int> st;**  **st.push(10);**  **st.push(20);**  **while (!st.empty())**  **{**  **cout<< st.size() << ” “;**  **cout << st.top() << endl;**  **st.pop();**  **}** |
| --- |

1. [**Queue**](https://www.geeksforgeeks.org/queue-cpp-stl/)**:**

* Queues use an encapsulated object of deque or list as its underlying container, but providing a specific set of member functions to access its elements.
* FIFO: Elements are inserted(enqueue) at the back (end) and are deleted(dequeue) from the front.
* Enqueue and Dequeue operations are named as push and pop itself like in stacks.

| **#include <queue> // Include queue (std namespace)**  **queue<int> q;**  **q.push(10);**  **q.push(20);**  **while (!q.empty())**  **{**  **cout<< q.size() << ” “;**  **// front() and back() instead of top() in stack.**  **cout << q.front() << endl;**  **cout << q.back() << endl;**  **q.pop();**  **}** |
| --- |

1. [**Priority\_queue**](https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/)**:**

* By default, it is implemented as a max heap where the first element of the queue is the greatest of all elements in the queue and elements are in non increasing order.
* Insertion and deletion takes logarithmic time complexity and we get max element in constant time.

| **#include <queue> // Include queue (std namespace)**  **priority\_queue<int> pq; // Max Heap**  **priority\_queue<int, vector<int>, greater<int>> pq; // Min Heap**  **pq.push(10);**  **pq.push(20);**  **while (!pq.empty())**  **{**  **cout<< pq.size() << ” “;**  **// top(), analogous to stacks.**  **cout << q.top() << endl; // returns the max/min element**  **q.pop();**  **}**  **// Max Heap for pairs, ordered by first element**  **priority\_queue<pair<int, int>> pq;**  **// Min Heap for pairs, ordered by first element**  **priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;**  **pair<int, int> top = pq.top();**  **cout << top.first << " " << top.second;**  **// Max heap for pairs, ordered by second element**  **struct myCmp**  **{**  **bool operator()(pair<int, int>& x, pair<int, int>& y)**  **{**  **return x.second < y.second;**  **}**  **};**  **priority\_queue<pair<int, int>, vector<pair<int,int>>, myCmp> pq;**  **// Min heap for pairs, ordered by second element: Change < in ‘x.second < x.second’ to > in // myCmp** |
| --- |

1. [**List**](https://www.geeksforgeeks.org/list-cpp-stl/)**:**

* As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick.
* Normally, when we say a List, we talk about doubly linked list. For implementing a singly linked list, we use forward list.
* The interferfaces mostly resemble that in a deque.

| **#include<list> // Include list (std namespace)**  **list<int> l;**  **// Insert head, index, tail**  **l.push\_front(value); // head**  **l.insert(l.begin() + index, value); // index**  **l.push\_back(value); // tail**  **// Access head, index, tail**  **int head = l.front(); // head**  **int value = next(l.begin(), index); // index**  **int tail = l.back(); // tail**  **// Size**  **unsigned int size = l.size();**  **// returns 1 if l is empty, 0 otherwise**  **bool res = l.empty();**  **// Iterate**  **for(auto it = l.begin(); it != l.end(); it++) {**  **cout << \*it << endl;**  **}**  **// Remove head, index, tail**  **l.pop\_front(); // head**  **l.erase(l.begin() + index); // index**  **l.pop\_back(); // tail**  **// Remove: Remove an element by value**  **l.remove(value);**  **// Unique: Remove duplicates**  **l.unique();**  **// Merge: Merge two sorted lists**  **l.merge(list2);**  **// Sort: Sort the list**  **l.sort();**  **// Reverse: Reverse the list order**  **l.reverse();** |
| --- |

1. [**String**](https://www.geeksforgeeks.org/stdstring-class-in-c/)**:**

* push\_back()
* pop\_back()
* insert()
* find()
* size()
* substr()
* reverse()
* swap()
* sort()
* append() or += shorthand operator
* copy() or = operator

| **#include <string> // Include string (std namespace)**  **string s1, s2="hello"; // Create strings**  **s1.size(), s2.size(); // Number of characters: 0, 5**  **s1 += s2 + ' ' + "world"; // Concatenation**  **cout << s1 == "hello world" // Comparison, also <, >, !=, etc.**  **cout << s1[0]; // 'h'**  **s1.substr(m, n); // Substring of size n starting at s1[m]**  **reverse(s1.begin(), s1.end()); // reverses the whole string in-place**  **sort(s1.begin(), s1.end()); // sorts the whole string in-place**  **s1.push\_back(‘k’); // Appends ‘k’ at the end of the string s1**  **s1.pop\_back(); // Removes a character from the end of the string**  **s1.append(s2); // appends s2 at the end of s1**  **s1.swap(s2); // swaps s1 and s2**  **s1 = to\_string(12.05); // Converts number to string**  **getline(cin, s); // Read line ending in '\n'** |
| --- |

**NOTE:**

* All of the above interfaces can be accessed by including just one header file #include<bits/stdc++.h>
* More detailed info: click [here](https://github.com/gibsjose/cpp-cheat-sheet/blob/master/Data%20Structures%20and%20Algorithms.md) or [here](https://www.cplusplus.com/reference/stl/).