**C++ Cheat Sheet**

|  |  |
| --- | --- |
| CHAR\_BIT | 8 |
| SCHAR\_MIN | -128 |
| SCHAR\_MAX | +127 |
| UCHAR\_MAX | 255 |
| CHAR\_MIN | -128 |
| CHAR\_MAX | +127 |
| MB\_LEN\_MAX | 16 |
| SHRT\_MIN | -32768 |
| SHRT\_MAX | +32767 |
| USHRT\_MAX | 65535 **(because no bit reserved for sign)** |
| INT\_MIN | -2147483648 |
| INT\_MAX | +2147483647 |
| UINT\_MAX | 4294967295 **(because no bit reserved for sign)** |
| LONG\_MIN | -9223372036854775808 |
| LONG\_MAX | +9223372036854775807 |
| ULONG\_MAX | 18446744073709551615 **(because no bit reserved for sign)** |

MAX = 1e9+7 or **1000000007**

|  |
| --- |
| **MAX value of N Time complexity** |
| 10^9 O(log N) or Sqrt(N) |
| 10^8 O(N) Border case |
| 10^7 O(N) Might be accepted |
| 10^6 O(N) Perfect |
| 10^5 O(N \* log N) |
| 10^4 O(N ^ 2) |
| 10^2 O(N ^ 3) |
| <= 160 O(N ^ 4) |
| <= 18 O(2N\*N2) |
| <= 10 O(N!), O(2N) |

|  |
| --- |
| **NOTE: Short names include DT (Data Type)** |

**Containers**

1. **Vectors: Implementation by dynamic array.**

|  |  |  |
| --- | --- | --- |
| Member Functions | Return Values | Command Brief |
| vector<DT> v |  | Default, Empty vector |
| vector<DT>v{ } |  | Initializer List |
| vector<DT>v={ } |  | Initializer list |
| vector<DT> v(size) |  | Fill constructor |
| vector<DT> v(size, init\_value) |  | Filled constructor |
| vector<DT> v2(itr1, itr2) |  | Deep copy, range based |
| vector<DT> v1(v2) |  | Deep copy, copy constructor |
| vector<DT>() | Returns Empty vector |  |
| v.assign(size, value) |  | Same as filled constructor but a function |
| v.assign(itr1, itr2) |  | Range based filling |
| vector<DT> v = { \_\_\_\_\_\_\_\_ } |  | Initialization |
| v.begin() | Iterator pointing to first element |  |
| v.end() | Iterator pointing to the post last element |  |
| v.rbegin() | Reverse Iterator pointing to the last element and this iterator traverse in reverse. |  |
| v.rend() | Reverse Iterator pointing to the element preceding first element and this iterator traverse in reverse. |  |
| v.size() | Unsigned Integer |  |
| v.max\_size() |  | Maximum potential size of container |
| v.capacity() | Unsigned Integer | 2^n size |
| v.resize(size, value) |  | Makes and vector to the given size while filling empty spaces with value |
| v.shrink\_to\_fit() |  | Resizes to accommodate available elements. Capacity=Size. |
| v.at(index) | DT value | Returns value at index |
| Iterator loop, \*itr is the element | \*itr=value and itr is address location |  |
| v[index] |  |  |
| v.data() | Pointer to vector’s start, pointing to first element |  |
| v.front() | Returns leftmost element |  |
| v.back() | Returns rightmost element |  |
| v.clear() | Void | Removes all elements in the container. Size = 0. Capacity is intact. |
| v.emplace(itr, value) | Returns iterator to newly inserted element. | Inserts value before the location pointed by itr. |
| v.emplace\_back(value) | Void | Similar to push\_back(value) |
| v.empty() | Boolean | If container is empty or not |
| v.erase(itr1, itr2) | Returns iterator pointing to element after the one which was erased | If one argument is given, one element will be erased at location pointed by it. In case of 2, range is erased as [itr1, itr2). |
| Member Functions | **Return Values** | **Command Brief** |
| v.insert(itr, value) | Returns random access iterator pointing to first of newly inserted element(s). | Pushes before itr with value |
| v.insert(itr, count, value) | Returns random access iterator pointing to first of newly inserted element(s). | Pushes before itr with count number of values |
| v.insert(itr, itr\_1, itr\_2) | Returns random access iterator pointing to first of newly inserted element(s). | Pushes before itr with ellements between [itr\_1, itr\_2) |
| v.push\_back(value) | Void | Appends value on right |
| v.pop\_back() | Void | Pops value from right, size--; |

1. **List: Doubly Linked List**

|  |  |  |
| --- | --- | --- |
| Member Functions | Return values | Command Brief |
| list<DT> l |  | Empty list init. |
| list<DT> l(size) |  | Fill Constructor |
| list<DT> l(size, init\_value) |  | Filled Constructor |
| list<DT> l(ptr1, ptr2) |  | Deep Copy, range based |
| list<DT> l( list2 ) |  | Deep Copy, copy constructor |
| l.assign(size, value) |  | Overriding |
| l.assign(itr1, itr2) |  | Overriding |
| list<DT> l = { \_\_\_\_ } |  | Initialization |
| list<DT>() | Returns Empty list | Used for returning empyu list |
| l.pop\_back() | Void | Removes the last element in the list container, effectively reducing the container size by one |
| l.pop\_front() | Void | Removes the first element in the list container,effectively reducing the conatiner size by 1. |
| l.push\_back(value) | Void | Adds a new element at the end of the [list](https://cplusplus.com/list) container, after its current last element. Size++ |
| l.push\_front(value) | Void | Adds new element at the front of the continer list, before current first element. Size++. |
| l.sort( comparator ) | Void, modifies | Sorts elements (increasing by default) as per comparator |
| l.size() | Unsigned Integer, size\_type | Number of elements in the container list |
| l.unique( comparator ) | Void | **Notice** that an element is only removed from the list container if it compares equal to the element immediately preceding it. Thus, this function is especially useful for sorted lists. By default it will remove all duplicates in **sorted** list. A comparator function can be provided to reduce list via pair wise comparison |
| l.swap( list2 ) |  | Swaps all contents of container l with list2 container |
| l.begin() | Iteraor | Pointer to beginning element. |
| l.end() | Iterator | Iterator pointing to post last element. |
| l.empty() | Boolean | Returns true if there are no elements in the list conatiner. |
| l.front() | DT | Returns first element reference (alue but can be modified), not pointer/iterator. |
| l.back() | DT | Returns last element reference, not pointer/iterator. |
| l.clear() | Void | Deleted all nodes and reduces size to zero |
| l.insert( itr, times, value) | An iterator that points to the first of the newly inserted elements. | Inserts n (by default 1) times the value before location pointed by itr in the list conatiner l. |
| l.insert (itr, itr1, itr2) | An iterator that points to the first of the newly inserted elements. | Inserts elements in another container in range **[ itr1, itr2 )** before location pointed by **itr** in target list conatiner l. |
| l.erase(itr, itr2) | Returns iterator pointing to the elements past last removed element. | Removes element pointed by itr or all elements in **[ itr ,itr2 )** |
| l.max\_size() | Unsigned Integer | Maximum number of elements a list can hold. |
| l.remove(val) | Void, modify | Erase by value. All occurences. Removes from the container all the elements that compare equal to val. |
| l.remove\_if(comparator) | Void, modify | Removes from the container all the elements for which **Comparator** returns true. Comparator here has argument = &val (1 reference value of list) |
| l.reverse() | Void, modifies |  |
| l.splice(itr, list2) | Void, modifies | Transfer all from list2 in l before itr. |
| l.splice(itr, list2, itr2) | Void, modifies | Transfer one element at itr2 into l before itr. |
| l.splice(itr, list2, itr\_start, itr\_end) | Void, modifies | Transfer elements between [ itr\_start, itr\_end ) into l from list2 before itr. |

1. **Pair: Size 2 array**

|  |  |  |
| --- | --- | --- |
| Member Functions | Return Value | Command Brief |
| pair<DT, DT> p; | Reference to a pair, currently empty |  |
| pair<DT, DT> p(p2); OR = operator |  | Deep copy, copy constructor |
| pair<DT, DT> p(value1, value2); |  | Initialization |
| p = make\_pair(value1, value2); |  |  |
| p.first | First member |  |
| p.second | Second Member |  |
| == operator | Boolean | Returns if content of both pair are same or not. |
| != operator | Boolean |  |
| Comparison Operators (>, <, >=, <=) | Boolean | Compares first element first and then if false checks second element as per the operator. For example, for < comparison is (a.first<b.first || (!(a.first<b.first) && a.second<b.second))  For a<=b, it is a.first<=b.first || (!(a.first<=b.first) && a.second<=b.second), OR simply !(b<a). |

1. **Stack: Linked List**

|  |  |  |
| --- | --- | --- |
| Member Function | Return Value | Command Brief |
| stack<DT> st |  | Creates an empty stack container |
| stack<DT> st(another\_container) |  | Creates new stack from elements of another stl container with same DT |
| st.push(value) | Void | Pushes element on stack, size++ |
| st.pop() | Void | Removes top element, size-- |
| st.top() | Reference to top element | Element on top reference. Can be modified with st.top(). |
| st.empty() | Boolean | Is the stack container empty? |
| st.size() | Unsigned Integer | Number of elements in stack |
| st.swap(st2) | Void | Swap references to the containers |
| Comparison operator (==, !=, >, <, <=, >=) | Boolean | Returns on comparison with each element from bottom to top. If Camparison holds true throughout then returns true else false. |

1. **Queue: Linked List**

|  |  |  |
| --- | --- | --- |
| Member Function | Return Value | Command Brief |
| queue<DT> q |  | Creates an empty queue container |
| queue<DT> q(another\_container) |  | Creates new queue from elements of another stl container with same DT |
| q.push(value) | Void | Pushes element into the queue at last, size++ |
| q.pop() | Void | Removes front element, size-- |
| q.front() | Reference to front element | Element on front reference. Can be modified with q.front(). |
| q.back() | Reference to last element, last added | Element at last reference. Can be modified with q.back(). |
| q.empty() | Boolean | Is the queue container empty? |
| q.size() | Unsigned Integer | Number of elements in queue |
| q.swap(q2) | Void | Swap references to the queues |
| q.emplace(value) | Void | Same as push(value) |
| Comparison operator | Boolean | Returns on comparison with each element from front to back . If Camparison holds true throughout then returns true else false. |

1. **Deque: 2 Queue**

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| deque<DT> dq |  | Constructs an empty container (o size) |
| deque<DT> dq(<size>, <value=0>) |  | Fill type, by default value is zero |
| deque<DT> dq(itr1, itr2) |  | Range constructor from another container |
| deque<DT> dq(<dq2>) |  | Copy constructor |
| deque<DT>dq { } |  | Initializer list initialization |
| = operator, dq = d2 |  | Assigns new contents to the container, replacing its current contents, and modifying its size accordingly. |
| dq.size() |  | Unsigned int = size\_type |
| dq.begin() | Iterator pointing to the first element of the container |  |
| dq.end() | Iterator pointing to the 1 + last poistion of the conatiner |  |
| dq.assign(<size>,<const\_value>) |  |  |
| dq.assign(itr1,itr2) |  |  |
| dq.at(index) | Returns reference to value at position n in the container | This is different from [] operator because it throws out\_of\_range exception when index run out of bound |
| dq.front() | Reference to the first eleemnt in the queue |  |
| dq.back() | Reference to the last element |  |
| [<index>] operator |  | Accesses value at index. Mutable |
| dq.emplace(itr,<value>) | Returns iterator pointing to the newly inserted elements | Inserts element with <value> at the poirion pointed by itr. |
| dq.emplace\_back(<value>)  dq.push\_back(<value>) | Constant order complexity |  |
| dq.emplace\_front(<value>)  dq.push\_front(<value>) | Constant order complexity |  |
| dq.empty() | Boolean |  |
| dq.erase(itr) //one element  dq.erase(itr1,itr2) //range based | An iterator pointing to the new location of the element that followed the last element erased by the function call. |  |
| dq.insert(itr,value)  dq.insert(itr, size, value)  dq.insert(itr, itr1,itr2)  dq.insert(itr, { … } ) | An iterator that points to the first of the newly inserted elements. | The deque container is extended by inserting new elements before the element at the specified position. |
| Out of range elements holds value zero. |  |  |
| dq.swap(dq2) |  | Swaps refernces to the containers |

1. **Bitset**

**Size=bitset string size. It converst integer into binary and captures bit values from msb. Same goes for initialization from binary c string.**

**NOTE: Indexing for position is from right to left.**

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| Bitset<size> x; |  | Bitset string initialized with zeros |
| Bitset<size> x(<Integer value>) |  | Bitset string initailauzed with integer value given |
| Bitset<size> x(binary c string) |  | Bitset string initailauzed with given **binary** string |
| x.set()  x.set(<position>, <with value>)  x.set(<position>) |  | Set all to 1  Set <with value> at position  Set 1 at position |
| x.all() | Boolean | All values set |
| x.any() | Boolean | Any value set |
| x.none() | Boolean | No value set |
| x.count() | Boolean | Count set values |
| x.size() | Size\_t: unsigned integer | Size of Bitset |
| x.flip()  x.flip(<position>) |  | Flip the bit at position or flip all bits |
| x[<position>] |  | Access bit at position |
| x.reset()  x.reset(<position>) |  | Reset bit to zero at <position> or reset all bits to zero. |
| x.test(<position>) | Boolean | Checks if he bit at position is set |
| x.to\_string<char,std::string::traits\_type,std::string::allocator\_type>(); |  | Converst bitset to string type |
| x.to\_ullong()  x.to\_ulong() |  | Converts bitset to corresponding decimal representation. |

1. **Map -> Sorted by keys**

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| map<DT,DT> mp | Const time | Initialize an empty map |
| map<DT,DT> mp(mp2) |  | Copy constructor |
| map<DT,DT> mp(map\_itr1, map\_itr2) |  | Range based |
| map<DT,DT>mp{ { … } … } |  | Initializer list initialization  Values are inserted according to the map DTs. |
| mp.begin() | Iterator to first element in the map |  |
| mp.end() | Iterator to the position after the last element in the map. |  |
| mp.rend() | Reverse end pointing iterator |  |
| mp.rbegin() | Reverse beginning pointing iterator |  |
| mp.clear() |  | Emptys the map. |
| mp.count(<key>) | 1 present /0 | Searches for the key (unique & sorted hear) |
| mp.emplace(<key>,<value>) | Returns pair of iterator pointing to the newly inserted k-v pair and true else if the key was already present it resturns corresponding iterator and false. | Inserts if the key is unique |
| mp.emplace\_hint(<pos>,<key>,<value>) | Generally, logarithmic in the container size.  Amortized constant if the insertion point for the element is position.  Returns a bidirectional iterator to new/existing k-v pair. | Inserts a new element in the map if its key is unique, with a hint on the insertion position. |
| mp.empty() | Boolean | Checks if map is empty or not |
| mp.erase(itr)  mp.erase(<key>)  mp.erase(itr1, itr2) //remove within . //this range | Constant, log and linear time complexities.  Returns iterator to next k-v if removed via iterator. If removed via key, returns number of items erased. |  |
| mp.find(<key>) | Return iterator to found key or mp.end() if not found. |  |
| mp.size() |  |  |
| mp.swap(mp2) |  |  |
| mp.insert(pair)  mp.insert(map\_itr1, map\_itr2) | Returns pair of (iterator to newly inserted element/existing element and true/false). False if not inserted. |  |
| mp[<key>]=value  mp[<key>] | If key matches the key of an element in the container, the function returns a reference to its mapped value.  If k does not match the key of any element in the container, the function inserts a new element with that key and returns a reference to its mapped value. Notice that this always increases the container size by one, even if no mapped value is assigned to the element | Note: This always inserts and increase size by 1. |
| mp.at(<key>) | map::at, has the same behavior when an element with the key exists, but throws an exception when it does not. | Returns A reference to the mapped value of the element with a key value equivalent to <key>. |
| Mp.lower\_bound(<key>) | Iterator pointing to the k-v pair in ordered map in which k is just equal to or greater than <key> |  |
| Mp.upper\_bound(<key>) | Iterator pointing to the k-v pair in ordered map in which k is just equal to or greater than <key> |  |
| Mp.equal\_range(<key>) | Returns the bounds of a range that includes all the elements in the container which have a key equivalent to k. | Because the elements in a map container have unique keys, the range returned will contain a single element at most.  **If no matches are found, the range returned has a length of zero** |

1. **Set (associative/ordered/key=value/unique)**

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| set<DT> s  set<DT> s() | Const time | Initialize an empty set |
| set<DT> s(s2) |  | Copy construction. |
| set<DT> s(itr1 , itr2) |  | Range based initialization |
| set<DT> s{ … }  set<DT> s ={ … } |  | Initializer list initialization |
| set<DT, comp\_class> s  set<DT, comp\_func\_ptr> s |  | Ordering based on comp\_class only bool function or bool function pointed by the function pointer. |
| = operator (s1=s2) |  | Assigns new contents to the container, replacing its current content. |
| s.begin() | Iterator pointing to the first element |  |
| s.end() | Iterator pointing to the element after the last one in the set. |  |
| s.empty() | Bool |  |
| s.count() | Unsigned int (1/0) |  |
| s.find() | Iterator to the element or set::end/s.end(). |  |
| s.clear() |  | Emptys the set |
| s.emplace(<value>) | If successfully inserted, returns a pair <itr to new element, true> else return a pair <itr to existing element, false> |  |
| s.insert(<value>)  s.insert(itr , itr2)  s.insert({ … })  s.insert(<hint\_pos> , <value>) | For single valu insertion, If successfully inserted, returns pair<itr to new element in the container, true> else returns pair<itr to existing element, false>.  In case of insertion with hint it returns itr to new element in the set or existing element. |  |
| s.erase(<value>) | Log(n), returns number of elements erased. That is always 1 if present. |  |
| s.erase(<itr>) | Constant |  |
| s.erase(itr1, itr2) | Linear |  |
| s.upper\_bound(val) | Returns itr to a value in set which is just greater than “val” |  |
| s.lower\_bound(val) | Returns itr to a value in set which is equal to or graeter than val |  |

1. **Multiset (Set allowing multiple occurances of same values) (Uniqueness is compromised)**

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| multiset<DT> mset  multiset<DT> mset() | Const time | Initialize an empty multi set |
| Multiset<DT> mset(itr1 , it2) |  | Range based initialization |

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| multiset<DT> mset(mset2) |  | Copy construction-> Deep copy |
| multiset<DT, class\_comparator>  Struct class\_comp{  bool operator()(DT var1, DT var2) const { return var1 < var2; }  } | Returns mset with elements set according to user chosen order through class comparator |  |
| multiset<DT, bool(\*)(DT, DT)>mset (func\_name)  bool func\_name(DT lhs, DT rhs) {  return lhs < rhs;  } | Returns mset with elements set according to user chosen order through comparator function. |  |
| multiset<DT> mset{ … }  multiset<DT> mset = { … } |  | Initializer List initialization |
| mset.begin() | Iterator |  |
| mset.end() | Iterator |  |
| mset.rend() | Iterator |  |
| mset.rbegin() | Iterator |  |
| mset.clear() |  | Empty the multiset |
| mset.count( val ) | Returns the number of elements in the multiset who are equal to **val**. |  |
| mset.emplace( val ) | Iterator (Bidirectional) pointing to the new element |  |
| mset.empty() | Boolean | Tels if the multiset is empty or not |
| mset.equal\_range( val ) | Returns pair of 2 iterators. First pointing to the lower bound of val in mset and second pointing to the upper bound of val in the mset. |  |
| mset.find( val ) | Returns iterator to the val in mset else mset.end(). |  |
| mset.erase(val) //erases all occurances of val  mset.erase(itr) //erases value at itr, single  mset.erase( itr1, itr2 ) // erases from [ itr1, itr2 ) | Mset.erase(val) returns the number of elements deleted/erased. |  |
| mset.insert(val)  mset.insert(itr, val) //with hint  mset.insert(itr1,itr2) //range based insert from // another container | an iterator pointing to the newly inserted element in the multiset. |  |
| mset.upperbound( val ) | Returns iterator pointing to element (first) which is just greater than val. |  |
| mset.lowerbound( val ) | Returns iterator pointing to element in mset which is = or greater than val |  |
| mset.size() | Returns size\_t, unsigned int | Size of mutiset container |

1. **MultiMap (Associative container allowing 1 Key holding multiple values. One to Many mapping)**

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| multimap<DT, DT> mmp  multimap<DT, DT> mmp() | Const time | Initialize an empty multi set |
| multimap<DT, DT> mmp(mmp2) |  | Copy construction |
| multimap<DT, DT> mmp(mmpitr1, mmpit2) |  | Range based initialization |
| multimap<DT1, DT2, class\_comparator>  Struct class\_comp{  bool operator()(DT1 var1, DT1 var2) const { return var1 < var2; }  } | Returns mmp with elements set according to user chosen order through class comparator |  |
| multimap<DT1, DT2, bool(\*)(DT1, DT1)>mmp (func\_name)  bool func\_name(DT lhs, DT rhs) {  return lhs < rhs;  } | Returns mmp with elements set according to user chosen order through comparator function. |  |
| multimap<DT,DT> mmp{ {,}{,}{,}… } |  | Initializer List initialization |
| multimap<DT,DT> mmp = {{,},{,},{,}…} |  | Initializer List initialization |
| mmp.end() | Iterator |  |
| mmp.rend() | Reverse Iterator in beginning |  |
| mmp.begin() | Iterator |  |
| mmp.rbegin() | Reverse Iterator in the end+1. |  |
| mmp.clear() |  | Clears multimap contents |
| mmp.count(k) | number of elements in the container contains that have a key equivalent to k |  |
| mmp.emplace(val1, val2) | An iterator to the newly inserted element. | Logarithmic |
| mmp.erase(itr) // single erase by position (based on FIFO)  mmp.erase(key\_val) // erase by key\_val  mmp.erase(itr1, itr2) //range based erase | For the key-based version (2), the function returns the number of elements erased.  The other versions return an iterator to the element that follows the last element removed |  |
| mmp.find(key\_val) | Searches the container for an element with a key equivalent to k and returns an iterator to it if found, otherwise it returns an iterator to multimap::end. |  |
| mmp.empty() | Boolean | Checks if the multimap container is empty. |
| mmp.equal\_range(key\_val) | The function returns a pair, whose member pair::first is the lower bound of the range (the same as lower\_bound), and pair::second is the upper bound (the same as upper\_bound). |  |

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| mmp.insert(pair<DT,DT>(key,val))  mmp.insert(mmpitr1, mmpitr2)  mmp.insert( {{ , }, { , }, { , }……} ) | Only the first one retuens the iterator pointing to the newly insserted element. |  |
| mmp.size() | Unsigned Int | Size of the conatiner. No. of elements in the conatiner |
| mmp.upper\_bound(key\_val) | Returns an iterator pointing to the first element in the container whose key just greater than key\_val |  |
| mmp.lower\_bound(key\_val) | Returns an iterator pointing to the first element in the container whose key = or greater than key\_val. |  |

1. **Unordered\_set:** Equivalent to Hash table with V-V mapping.

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| unordered\_set<DT> uset  unordered\_set<DT> uset() | Const time  (For move const. as well) | Empty initialization |
| unordered\_set<DT> uset(uset2) | WC=quadratic |  |
| unordered\_set<DT> uset({ … })  unordered\_set<DT> uset={} | WC=quadratic | Initializer list |
| unordered\_set<DT> uset(itr1,itr2) | WC=quadratic | Range based Initialization |
| uset.begin() |  | Pointers pointing to the beginning element |
| uset.end() |  | Pointer pointing to the element after the end element. |
| uset.empty() | Boolean |  |
| uset.size() | Size\_t | Number of values in the unordered set container. |
| uset.insert(val) //single value insertion  uset.insert(itr1, itr2) //range based insertion  uset.insert({ … }) //insert via init list | First return a pair of <itr bool> bool is true if insertion is successful. If inserted itr points to the newly inserted element, else it points to the the existing element, with bool=false. | WC for single value = linear  WC for range based or init list insertion is size\*N+1 |
| uset.bucket(k) | Returns the bucket number where the element with value k is located.  A bucket is a slot in the container's internal hash table to which elements are assigned based on their hash value. Buckets are numbered from 0 to (bucket\_count-1). |  |
| uset.bucket\_count() | Returns the number of buckets in the unordered\_set container.  A bucket is a slot in the container's internal hash table to which elements are assigned based on their hash value. | Usigned Int |

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| uset.bucket\_size(<bucket\_number>) | Returns the number of elements in bucket <bucket\_number>. | The number of elements in a bucket influences the time it takes to access a particular element in the bucket. |
| uset.erase(itr)  uset.erase(<val>)  uset.erase(itr1, itr2) | Versions (1) and (3) return an iterator pointing to the position immediately following the last of the elements erased.  Version (2) returns the number of elements erased | WC=linear |
| uset.count(val) | Returns 1 or 0 |  |
| uset.find(val) | Iterator to the searched value | WC=linear or AC=constant |

1. **Unordered\_map: Assocative conatiner equivalent to a hashmap.**

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| unordered\_map<DT, DT> ump  unordered\_map<DT, DT> ump() |  | Empty initialization. TC: const |
| unordered\_map<DT, DT> ump(ump2) | WC: quadratic | Copy construction |
| unordered\_map<DT, DT> umap(itr1, it2) | Note: itr are pointing to the content in the for of k-v pair | Range based initialization |
| unordered\_map<DT, DT> ump={{k,v},{k,v} … }  unordered\_map<DT, DT> ump({{k,v},{k,v}, … }) |  | Initializer List Initialization |
| ump.begin() | Iterator pointing to the first k-v pair |  |
| ump.end() | Iterator pointing to the location after the last K-V pair in the container |  |
| ump.bucket(k) | Returns the number of bucket where the k is located. K is the key. Returns the unsigned int |  |
| ump.bucket\_count() | Unsigned int. Number of buckets in the container. It is not necessary that each bucket must contain something |  |
| ump.bucket\_size(<bucket\_number>) | Number of k-v pairs in the <bucket\_number> bucket |  |
| ump.clear() |  |  |
| ump.count(key) | 1 if an element with a key equivalent to k is found, or zero otherwise. |  |
| ump.empty() | Boolean |  |
| ump.erase(itr)  ump.erase(<key>)  ump.erase(itr1, itr2) | Versions (1) and (3) return an iterator pointing to the position immediately following the last of the elements erased.  Version (2) returns the number of elements erased |  |
| ump.find(<key>) | An iterator to the element, if the specified key value is found, or unordered\_map::end if the specified key is not found in the container. |  |
| ump.insert(<key>)  ump.insert({{k,v},{k,v}, ... })  ump.insert(itr1, itr2) | Version 1 Return pair <itr,bool> as expected and the other 2 versions returns nothing. | Inserts new elements in the unordered\_map. |

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| ump[<key>] | If <key> matches the key of an element in the container, the function returns a reference to its mapped value. | Accessing method with insertion |
| ump.at(<key>) | Returns a reference to the mapped value of the element with key k in the unordered\_map.  If k does not match the key of any element in the container, the function throws an out\_of\_range exception. | Accessing without insertion |
| ump.size() | Number of k-v pairs in the unordered map container. |  |
| ump.swap(ump2) | Exchange references to the 2 conatiners. That is ump points to ump2 and ump2 points to ump. |  |

1. **Unordered\_multiMap: Same as Unordered map but this allows non-bijective relations. 1 key can point to multiple values.**

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| unordered\_multimap<DT, DT> ummp  unordered\_map<DT, DT> ummp() | Constant time complexity | Empty initialization. TC: const |
| unordered\_multimap<DT, DT> ummp(ummp2) | TC: worst case is Quadratic construction | Copy construction. Deep copy |
| unordered\_multimap<DT,DT> ummp(itr1,itr2) | TC: worst case is Quadratic construction | Range based initialization |
| unordered\_multimap<DT,DT> ummp={{k,v},{k,v}…..}  unordered\_multimap<DT,DT> ummp({{k,v},{k,v}….}) | TC: worst case is Quadratic construction | Initializer List construction. |
| ummp.bucket(key) | A bucket is a slot in the container's internal hash table to which elements are assigned based on the hash value of their key. Elements with the same key are located in the same bucket. Buckets are numbered from 0 to (bucket\_count-1). | Constant complexity |
| ummp.bucket\_count() | Returns number of buckets in the container’s internal implementation. Unsigned int | Constant complexity |
| ummp.bucket\_size(<bucket\_index>) | Note that bucket index cab be from 0 to bucket\_count-1. It returns number of elements filled in the bucket with index = <bucket\_index> | Linear |
| ummp.clear() | Truncate the container | Linear |
| ummp.count(key) | Unsigned Int. Linear | Searches the container for elements whose key is k and returns the number of elements found. |
| ummp.size() | Unsigned Int | Number of elements.pairs in this container. |
| mmp.empty() | Boolean: is empty? | Constant |

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| ummp.emplace(key,value) | Inserts the value to the corresponding key. A forward iterator to the newly inserted element is returned. | This may trigger rehash |
| ummp.insert(pair<DT,DT>(key,value))  ummp.insert(make\_pair<DT,DT>{ key, value } )  ummp.insert({{key,value}, { } ….. })  ummp.insert(itr1, itr2) | Initializer list and ranges based insertion return no value. Else it returns the iterator to the newly inserted element. | Single element insertions:  Average case: constant.  Worst case: linear in container size.  Multiple elements insertion:  Average case: linear in the number of elements inserted.  Worst case: N\*(size+1): number of elements inserted times the container size plus one. |
| ummp.find(key)  //ummp.find(key)->second will return first value inserted corresponding to the given key. | An iterator to the element, if the specified key value is found, or unordered\_multimap::end if the specified key is not found in the container. |  |
| ummp.erase(itr)  ummp.erase(<key>)  ummp.erase(itr1, itr2) // range based removal. Itr must point to element in the conatiner. | Versions (1) and (3) return an iterator pointing to the position immediately following the last of the elements erased.  Version (2) returns the number of elements erased. | TC: Linear in the conatiner size |
| ummp.begin() | Bidirectional iterator |  |
| ummp.end() | Bidirectional iterator |  |

1. **Unordered\_multiset: Same as the unordered set associative container but this allows the insertion of duplicates.**

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| unordered\_multiset<DT> umset;  unordered\_multiset<DT> umset(); | TC: constant | Returns an empty unordered multi set conatiner |
| unordered\_multiset<DT> umset(itr1,itr2) | WC: Quadratic | Range based initialization |
| unordered\_multiset<DT> umset(umset2) | WC: Quadratic | Deep Copy construction |
| unordered\_multiset<DT> umset({k1,k2,k3….})  unordered\_multiset<DT> umset={k1,k2,k3…} | WC: Quadratic | Initializer list initialization. |
| umset.bucket\_count() | TC: constant , Unsigned Integer | Number buckets in the multi set conatiner. |
| umset.bucket(key) | TC: constant (based on hash). Unsigned Integer | Bucket number, the corresponding key belongs to. |
| umset.bucket\_size(index) | TC: Linear, Unsigned integer | Number of elements in the bucket on the index = index |
| umset.begin() | Bidirectional iterator to start |  |
| umset.end() | Bidirectional iterator to last element + 1 |  |
| umset.clear() | Trancates the conatiner. |  |
| umset.count(key) | Gives the count of the element = key | Linear complexity |

|  |  |  |
| --- | --- | --- |
| Member Funcion | Return Value | Command Brief |
| umset.emplace(key) | An iterator to the newly inserted element.  Member type iterator is a forward iterator type. | Linear complexity in the worst case if hash hits. |
| umset.insert(key)  umset.insert(itr1, itr2)  umset.insert( {k1,k2,k3 ….} ) | Range based and initializer list insertion do not return anything. Else will return the iterator (foraward) to the newly inserted element. | Single element insertions:  Average case: constant.  Worst case: linear in container size.  Multiple elements insertion:  Average case: linear in the number of elements inserted.  Worst case: N\*(size+1): number of elements inserted times the container size plus one. May trigger rehash. |
| umset.find(key) | Searches the container for an element with k as key and returns an iterator to it if found, otherwise it returns an iterator to unordered\_multiset::end |  |
| umset.erase(itr)  umset.erase(key)  umset.erase(itr1,itr2)  // range based itr must point to elements in the conatiner. | Versions (1) and (3) return an iterator pointing to the position immediately following the last of the elements erased.  Version (2) returns the number of elements erased. | WC: Linear. |
|  |  |  |

**C++ 11 Updates:**

* **Typedef <DT> <alias>**
* **Initializer list initialization:** All relevant STL containers have been accepted to use initializer list initialization. Apart from this can also be used to initialize public members of a class/struct. If we want to initialize private members from this kind of initialization, we can define our own contructor like this: \_\_init\_\_(const initializer\_list<DT> v){ … } //can access elements of initializer\_list via iterator.
* **Constructor Preference:** Initializer\_list > Normal Constructor > Aggregate initialization.
* **Initializer List initialization mimics python like initialization.**
* Use of for each loop with refefrence argument
* **Use of auto**
* **NULL can be replaced with nullptr (**dedicated to represent pointers**).**
* **enum class**
* **override and final keyword:** In C++11, overide keyword is return when a virtual function is overriden in the child class. Final keywork on the class or the function showcases this class cannot have child or this function cannot be overriden.

**Class class\_name final{ } / virtual void break() final{ }**

* **Constructor Delegation:** What if you want to call a constructor inside another constructor of same class.

**Class A { A(); A(int x){ A(); …. } } //This will give an error. To do this we can do like the folowing:**

**Class A { A(); A(int x):A(){ …. } } //Note A() will be called first, before A(int)**

* **delete keyword use for constructor, operator or polymorphic function.**
* **Computation of function in compile time using “constexpr” before function declaration/def.**
* **Lambda function:** Used for functional programming.

**[](<parameters>) { }**

**auto f = [](<parameters>){ }; f(<parameters>);**