Standard Containers

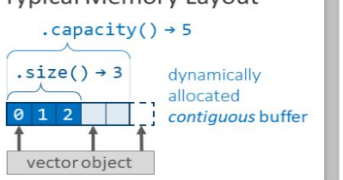
Construct A New Vector Object

```
vector<int> v1 {2,9,1,8,5,4}
vector<int> v2 (begin(v1)+3, end(v1))
vector<int> v3 (5,3)
vector<int> deep_copy_of_v1 (v1)
```

C++17 value type deducable from argument type

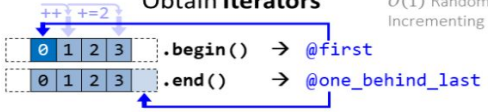
```
vector w {7,4,2};   // vector<int>
```

Typical Memory Layout

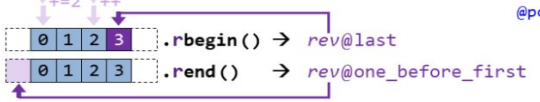


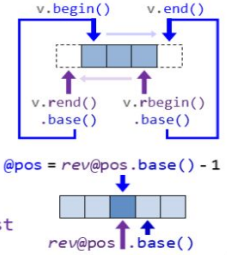
Obtain Iterators

$\mathcal{O}(1)$ Random Incrementing



Obtain *Reverse* Iterators



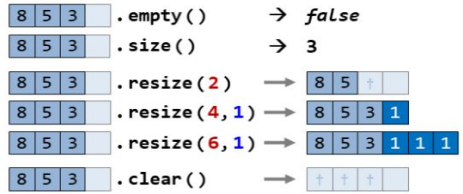


Assign New Content To An Existing Vector

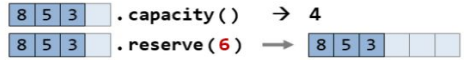
```
vector<int> v1 {8,5,3};
vector<int> v2 {6,8,1,9};
v1      =     v2;
```

new state of v1

Query/Change Size (= Number of Elements)



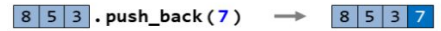
Query/Grow Capactiy (= Memory Buffer Size)



Avoid expensive memory allocations: `.reserve` capacity before appending / inserting if you know the (approximate) number of elements to be stored in advance!

Append Elements

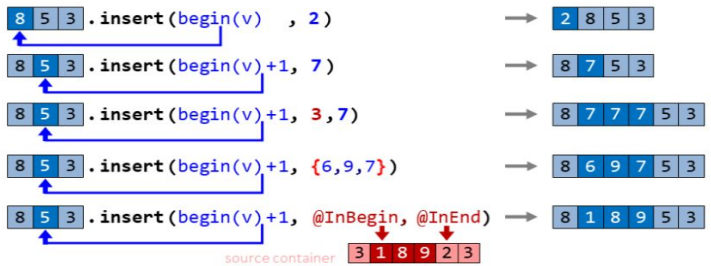
$\mathcal{O}(1)$ Amortized Complexity



Insert Elements at Arbitrary Positions

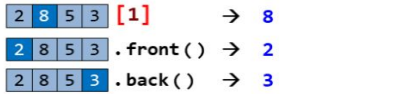
$\mathcal{O}(n)$ Worst Case

```
vector<int> v {8,5,3};
```

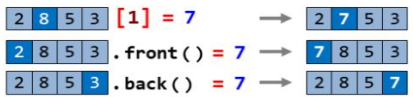


Get Element Values

$\mathcal{O}(1)$ Random Access



Change Element Values

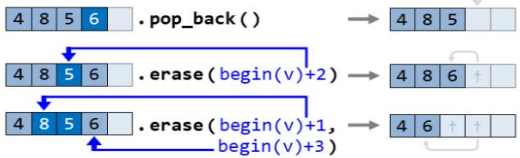


Out of Bounds Access

Erase Elements

$\mathcal{O}(n)$ Worst Case

```
vector<int> v {4,8,5,6};
```



Shrink The Capacity

(might be inefficient)

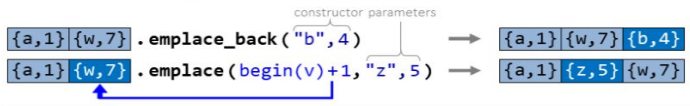
Erasing, resizing or clearing will not shrink the capacity!

```
vector<int> v (1024,0);   // capacity is at least 1024
v.resize(40);             // capacity unchanged!
v.shrink_to_fit();        // may shrink (not guaranteed)
v.swap(vector<int>(v));   // shrinks but has copy overhead
```

Insert & Construct Elements in Place

$\mathcal{O}(n)$ Worst Case

```
vector<pair<string,int>> v {{"a",1},{"w",7}};
```



Construct A New Deque Object

```
deque <int> d1 {2,9,1,8,5,4}
deque <int> d2 (begin(d1)+3, end(d1))
deque <int> d3 (5,3)
deque <int> deep_copy_of_d1 (d1)
```

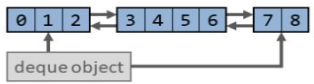
C++17 value type deducable from argument type

```
deque  d4 {7,4,2};   // deque <int>
```

Typical Memory Layout

Note that the ISO standard only specifies the properties of deque (e.g., constant-time insert at both ends) but not how that should be implementd.

dynamically allocated contiguous chunks



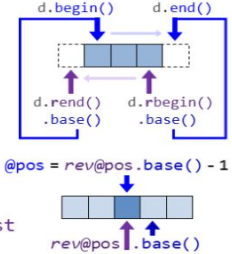
Obtain Iterators

$\mathcal{O}(1)$ Random Incrementing

```
.begin()  → @first
.end()    → @one_behind_last
```

Obtain *Reverse* Iterators

```
.rbegin() → rev@last
.rend()   → rev@one_before_first
```



Assign New Content To An Existing Deque

```
deque<int> d1 {8,5,3};
deque<int> d2 {6,8,1,9};
d1      =     d2;
```

new state of d1

```
.assign({4,1,3,5})
.assign(2, 1)
.assign(@InBeg,@InEnd)
```

source container

Query Size (= Number of Elements) $\mathcal{O}(1)$

```
.empty()  → false
.size()   → 3
```

Change Size $\mathcal{O}(|n - \text{newSize}|)$

```
.resize(2)
.resize(4,1)
.resize(6,1)
.clear()
```

Append Elements $\mathcal{O}(1)$

```
.push_back(7)
```

Prepend Elements $\mathcal{O}(1)$

```
.push_front(7)
```

Insert Elements at Arbitrary Positions $\mathcal{O}(n)$ Worst Case

```
deque <int> d {8,5,3};
.insert(begin(d)+1, 7)
.insert(begin(d)+1, 3,7)
.insert(begin(d)+1, {6,9,7})
.insert(begin(d)+1, @InBegin,@InEnd)
```

source container

Get Element Values $\mathcal{O}(1)$ Random Access

```
[1]   → 8
.front()  → 2
.back()   → 3
```

Change Element Values

```
[1] = 7
.front() = 7
.back()  = 7
```

Out of Bounds Access

```
[6]   → Undefined Behavior
```

Invalid Index!

```
.at(6)  → Throws Exception std::out_of_range
```

Erase Elements At The Ends $\mathcal{O}(1)$

```
.pop_back()
.pop_front()
```

Erase Elements At Arbitrary Positions

```
deque <int> d {4,8,5,6};
```

$\mathcal{O}(n)$ Worst Case Complexity

```
.erase(begin(d)+2)
.erase(begin(d)+1, begin(d)+3)
```

Insert & Construct Elements in Place $\mathcal{O}(n)$ Worst Case

```
deque <pair<string,int>> d {{"a",1},{"w",7}};
```

constructor parameters

```
.emplace_back ("b",4)
.emplace_front("c",6)
.emplace(begin(d)+1,"z",5)
```

Construct A New List Object

```
list<int> L1 {9,8,5,4}
list<int> L2 (next(begin(L1)),end(L1))
list<int> L3 (4,3)
list<int> deep_copy_of_ls1 (L1)
```

C++17 value type deducable from argument type

```
list    L4 {7,4,2};   // list<int>
```

Assign New Content To An Existing List

```
list<int> L1 {8,5,3};
list<int> L2 {6,8,1,9};
L1           =  L2;
```

new state of L1

```
.assign({4,1,3,5})
.assign(2, 1)
.assign(@InBeg,@InEnd)
```

source container

Access Element Values

$\mathcal{O}(1)$ access only to first and last element

```
.front()  →  2
.back()   →  3
.front() = 7
.back()  = 7
```

Access Arbitrary Elements Using Iterators

```
list<int> ls {2,8,5,3};
auto i = ls.begin(); // obtain iterator
cout << *i;          // prints 2
++i;                 // go to next
*i = 7;              // change to 7
```

Query / Change Size (= Number of Elements)

```
.empty()   →  false        O(1)
.size()    →  3            O(1)
.resize(2)
.resize(5,1)
.clear()
```

Append / Prepend Elements

$\mathcal{O}(1)$

```
.push_back(7)
.push_front(7)
```

Insert Elements at Arbitrary Positions

$\mathcal{O}(\#\text{inserted})$

```
list<int> L {8,5,3};
.insert(next(begin(L)), 7)
.insert(next(begin(L)), 2,7)
.insert(next(begin(L)), {6,9})
.insert(next(begin(L)), @b, @e)
```

source container

Insert & Construct Elements Without Copy / Move

$\mathcal{O}(1)$

```
list<pair<string,int>> L {{"a",1},{"w",7}};
```

constructor parameters

```
.emplace_back("b",4)
.emplace_front("c",6)
.emplace(next(begin(L)),"z",5)
```

Splice (Elements From) One Lists Into Another One

Does not copy or move elements!

```
list<int> T {8,5,3};      list<int> S {7,9,2};
.splice(next(begin(T)), S)                          O(1)
.splice(next(begin(T)), S, next(begin(S)))          O(1)
.splice(next(begin(T)), S, next(begin(S)), end(S))  O(#inserted)
```

source list

Merge Already Sorted Lists

(stable: if two elements are equivalent, that from L1 will precede that from L2)

$\mathcal{O}(n_1 + n_2)$

```
list<int> L1 {2,4,5,8};
list<int> L2 {1,2};
.merge( )
```

Obtain Iterators

```
.begin() → @first
.end()   → @one_behind_last
```

v.begin() v.end() v.rend().base() v.rbegin().base()

Obtain Reverse Iterators

```
.rbegin() → rev@last
.rend()   → rev@one_before_first
```

@pos = rev@pos.base() - 1

rev@pos .base()

Increment Iterators

```
next(begin( , M) → @Mth_element
```

$\mathcal{O}(M)$

Reorder Elements

```
.sort()
.sort(std::greater<>{})
.reverse()
```

$f(\circ,\circ) \mapsto$ bool

Erase Elements Based on Positions

$\mathcal{O}(\#\text{deleted})$

```
list<int> L {8,4,3,5};
.pop_back()
.pop_front()
.erase(next(begin(L)))
.erase(next(begin(L)), next(begin(L),3))
```

Erase Elements Based on Values

$\mathcal{O}(n)$

```
.remove(7)                  → 3   C++20
.remove_if(is_even)         → 4
.unique()                   → 2
.unique( equal_abs )        → 2
```

$f(\circ) \mapsto$ bool

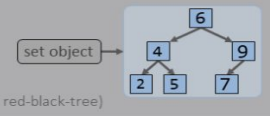
$f(\circ,\circ) \mapsto$ bool

as of C++20 functions return the number of deleted elements

Associative Containers

std::multiset<KeyType, Compare> (multiple equivalent keys)

- keys are ordered according to their values
- keys are compared / matched based on *equivalence*:
  a and b are equivalent if neither is ordered before the other,
  e.g., if `not (a < b) and not (b < a)`
- default ordering comparator is `std::less`
- sets are usually implemented as a balanced binary tree (e.g., as red-black-tree)



Construct A New Set Object

```
set<int> s0 {}
set<int> s1 {2,1,8,5,4}
set<int> s2 (begin(s1)+2, end(s1))
set<int> deep_copy_of_s1 (s1)
```

C++17 key type deducable from argument type

```
set s3 {7,2,4}; // set<int>
```

Assign New Content To An Existing Set

(deep copy from source)

```
set<int> s1 {1,3,5,7};
set<int> s2 {4,6,8};
s1 = s2;
```

new state of s1

Key Lookup

$\mathcal{O}(\log n)$

C++20

```
.contains(2) → false
.contains(5) → true

.count(2) → 0
.count(5) → 1
```

can be > 1 only for std::multiset

```
.find(2) → @end (= no match)
.find(5) → @match
.lower_bound(3) → @first_not_smaller
.upper_bound(3) → @first_greater
.equal_range(3) → {@Lbound,@Ubound}
```

Query Size (= Number of Keys)

```
.empty() → false
.size() → 3
```

Erase All Keys

```
.clear()
```

Insert A Single Key

$\mathcal{O}(\log n)$

```
.insert(4) → {.first @inserted, .second true}
.insert(5) → {.first @blocking, .second false} (not inserted)
```

potential performance benefit by hinting at probable insert position

```
.insert(@hint,7) → @inserted_or_block
```

Insert Multiple Keys

$\mathcal{O}(\#\text{inserted} \cdot \log n)$

```
.insert({1,6,8}) → 1 2 5 6 8
.insert(@inB,@inE) → 2 3 5 6 8
```

source container: 3 2 6 2

Insert & Construct A Key in Place

$\mathcal{O}(\log n)$

```
set<pair<int,int>> s {{1,3},{5,6}};
.emplace(4,7) → {.first @inserted, .second true} (if inserted)
```

pair constructor arguments

potential performance benefit by hinting at probable insert position

```
.emplace_hint(@hint,4,7) → @inserted
```

Erase One Key or A Range of Keys

```
.erase(2) → 0
.erase(5) → 1
```

$\mathcal{O}(\log n)$ #erased

$\mathcal{O}(1)$ amortized

```
.erase(@pos) → @after_erased
.erase(@beg,@end) → @after_erased
```

$\mathcal{O}(\log n + \#\text{erased})$

Obtain Iterators

```
.begin() → @first
.end() → @one_behind_last
```

Obtain *Reverse* Iterators

```
.rbegin() → rev@last
.rend() → rev@one_before_first
```

v.begin() v.end()
v.rend().base() v.rbegin().base()
@pos = rev@pos.base() - 1
rev@pos .base()

Extract Nodes

C++17

Allows efficient key modification and transfer of keys between different set objects.

```
.extract(5) → 5 O(logn)
.extract(@pos) → 5 O(1)
```

Merge Two Sets

$\mathcal{O}(n_2 \cdot \log(n_1 + n_2))$

```
set<int> S1 {1,3,5,7};
set<int> S2 {2,5,8};
.merge( 2 5 8 )
```

Insert Nodes

C++17

members of the return type

```
.insert(5) → {.position|.inserted|.node}
.insert(1) → {.position|.inserted|.node}
.insert( ) → {.position|.inserted|.node} (empty node)
.insert(@hint,5) → @inserted
.insert(@hint,1) → @blocking
```

Modify Key

No direct modification allowed!
Instead: **extract** key, **modify** its value and **re-insert**.

```
set<int> s {1,5,7};
auto node = s.extract(5);
if (node) {  // if key existed
  node.value() = 8;
  s.insert(std::move(node));
}
```

std::multimap<KeyType,MappedType,KeyCompare> (multiple equivalent keys allowed)

Construct A New Map Object

```
map<int,string> m0 { }
map<int,string> m1 {{4,"Z"},{2,"A"},{7,"Y"}}
map<int,string> m2 ( begin(m1)+1, end(m1) )
map<int,string> deep_copy_of_m1(m1)
```

C++17 key and mapped types deducable from arguments

```
map m3 {{2,3.14},{5,6.0}};   // map<int,double>
```

Insert A Single Key-Value Pair $\mathcal{O}(\log n)$

(inserted yes/no)

```
.insert({2,"W"}) → {@inserted,true}
.insert({3,"X"}) → {@blocking,false}
```

potential performance benefit by hinting at probable insert position

```
.insert(@hint,{2,"W"}) → @ins/block
```

Obtain Iterators

```
.begin() → @first
.end()   → @one_behind_last
```

Obtain *Reverse* Iterators

```
.rbegin() → rev@last
.rend()   → rev@one_before_1st
```

v.begin() v.end() v.rend() v.rbegin() .base() .base()

@pos = rev@pos.base() - 1 rev@pos.base()

- key-value pairs are ordered by key
- key matching is *equivalence*-based: 2 keys a and b are equivalent if not (a < b) and not (b < a)
- default key comparator is std::less
- maps are usually implemented as a balanced binary tree (e.g., as red-black-tree)

map object

Assign New Content To An Existing Map

```
map<int,string> m1 {{2,"X"}};
map<int,string> m2 {{1,"A"},{4,"G"}};
m1 = m2;
```

(deep copy from source)

new state of m1

Lookup Using Keys as Input $\mathcal{O}(\log n)$

```
map<int,string> m {{3,"A"},{5,"X"},{1,"F"}};
.contains(2) → false
.contains(5) → true
```

C++20

```
.count(2) → 0
.count(5) → 1
```

can be > 1 only for std::multimap

```
.find(2) → @end   (= no match)
.find(5) → @match
.lower_bound(3) → @1st_not_smaller
.upper_bound(3) → @1st_greater
.equal_range(3) → {@Lower,@Upper}
.at(3) → "A"
.at(2) → Throws Exception std::out_of_range
```

Query Size (= number of key-value pairs)

```
.empty() → false
.size()  → 2
```

Insert Multiple Key-Value Pairs $\mathcal{O}(\#\text{inserted} \cdot \log n)$

```
.insert({{5,"K"},{3,"Y"},{1,"G"}})
.insert(@inB, @inE)
```

source container

Insert & Construct Key-Value Pair $\mathcal{O}(\log n)$

(inserted yes/no)

```
.emplace(2,"W") → {@inserted,true}
```

potential performance benefit by hinting at probable insert position

```
.emplace_hint(@hint,2,"W") → @inserted
.try_emplace(2,"W") → {@inserted,true}
```

C++17

advantage: does not move from rvalue input parameters if not inserted

Erase Key-Value-Pair(s)

```
.erase(2) → 0
.erase(3) → 1
```

$\mathcal{O}(\log n)$ #erased

$\mathcal{O}(1)$ amortized

```
.erase(@pos) → @after_erased
```

$\mathcal{O}(\log n + \#\text{deleted})$

```
.erase(@b,@e) → @after_erased
```

Erase All

```
.clear()
```

Access / Modify Value $\mathcal{O}(\log n)$

```
map<int,string> m {{1,"F"},{3,"A"}};
[3] → "A"
[3] = "X"
```

Attention: [k] inserts new pair if key k is not present!

```
[2] = "W"
[2] → ""
```

Insert or Assign Value $\mathcal{O}(\log n)$ C++17

(inserted yes/no)

```
.insert_or_assign(3,"X") → {@as,false}
.insert_or_assign(5,"R") → {@ins,true}
.insert_or_assign(@hint,3,"W") → @as
.insert_or_assign(@hint,2,"G") → @ins
```

Merge Two Maps $\mathcal{O}(n_2 \cdot \log(n_1 + n_2))$ C++17

```
map<int,string> m1 {{1,"F"},{3,"S"},{5,"T"}};
map<int,string> m2 {{2,"A"},{5,"X"}};
.merge(2 A 5 X)
```

Extract Nodes

Allows efficient transfer of key-value pairs. C++17

```
.extract(2) → 2 R   O(log n)
.extract(@pos) → 2 R   O(1)
```

(Re-)Insert Nodes

members of the return type

```
.insert(5 N) → {.position|.inserted|.node}   true
.insert(3 Z) → {.position|.inserted|.node}   false
.insert( )   → {.position|.inserted|.node}   false
.insert(@hint,5 X) → @inserted
.insert(@hint,1 G) → @blocking
```

(empty node) (empty)

Modify Key

Direct *key* modification not allowed!

Instead:
- extract
- modify
- re-insert

```
map<int,string> m {{1,"F"},{3,"A"}};
auto node = m.extract(3);
if (node) {   // if key existed
  node.key() = 8;
  m.insert(move(node));
}
```

std::unordered_multiset<KeyT,Hash,KeyEqual>

(multiple equivalent keys allowed)

Construct A New Set Object

```
unordered_set<int> s0 {}
unordered_set<int> s1 {2,1,8,4,5}
unordered_set<int> s2 (begin(s1)+2, end(s1))
unordered_set<int> deep_copy_of_s1 (s1)
```

C++17 key type deducable from argument type

```
unordered_set s3 {7,2,4}; // unordered_set<int>
```

Assign New Content To An Existing Set

(deep copy from source)

```
unordered_set<int> s1 {1,5,3,7};
unordered_set<int> s2 {8,4,6};
s1 = s2;
```

new state of s1

Key Lookup

$\mathcal{O}(1)$ average, $\mathcal{O}(n)$ worst case

```
.contains(2) → false
.contains(3) → true
.count(2) → 0
.count(3) → 1
.find(2) → @end (= no match)
.find(3) → @match
.equal_range(5) → {@first_equal, @after}
```

C++20

can be > 1 only for std::unordered_multiset

Query Size (= Number of Keys)

```
.empty() → false
.size() → 3
```

Erase All Keys

```
.clear()
```

Obtain Iterators (to keys)

```
.begin() → @first
.end() → @one_behind_last
```

Insert A Single Key

$\mathcal{O}(1)$ average, $\mathcal{O}(n)$ worst case

```
.insert(4) → {@inserted, true}
.insert(8) → {@blocking, false}
```

.first .second

(not inserted)

potential performance benefit by hinting at probable insert position

```
.insert(@hint, 7) → @inserted_or_block
```

Insert Multiple Keys

$\mathcal{O}(\#ins)$ avg., $\mathcal{O}(n \cdot \#ins + \#ins)$ worst

```
.insert({1,6,8})
.insert(@inB, @inE)
```

source container

Insert & Construct A Key in Place

$\mathcal{O}(1)$ avg., $\mathcal{O}(n)$ worst

```
unordered_set<pair<int,int>> s {{1,3},{5,6}};
.emplace(8,7) → {@inserted, true}
```

(if inserted)

pair constructor arguments

potential performance benefit by hinting at probable insert position

```
.emplace_hint(@hint, 8, 7) → @inserted
```

Erase One Key or A Range of Keys

$\mathcal{O}(\#erased)$ avg. $\mathcal{O}(n)$ worst case

```
.erase(2) → 0
.erase(3) → 1
.erase(@pos) → @after_erased
.erase(@beg, @end) → @after_erased
```

#erased

Query & Control Hash Table Properties

hash function: $h(key) \mapsto$ bucket index

```
unordered_set<char> us {'E','X','Z','A','F','B'};
.bucket('E') → 4
.begin(4) → @first_in_bucket
.end(4) → @one_behind_last_in_bucket
.bucket_size(4) → 2
.bucket_count() → 7
us.load_factor() → 4/7 = 0.57
us.max_load_factor(0.8) (set)
us.max_load_factor() → 0.8 (get)
.reserve(min_capacity)
.rehash(#hash_buckets)
```

(key → hash buchet index)

(make table large enough to hande min_capacity elements)

(can allocate even more buckets if max. load factor demands it)

Extract Nodes

$\mathcal{O}(1)$ avg., $\mathcal{O}(n)$ worst

C++17

Allows efficient key modification and transfer of keys between different set objects.

```
.extract(5) → 5
.extract(@pos) → 5
```

Merge Two Sets

C++17

$\mathcal{O}(n_2)$ average, $\mathcal{O}(n_1 \cdot n_2 + n_2)$ worst case

```
unordered_set<int> S1 {1,5,3};
unordered_set<int> S2 {8,1,2};
.merge( )
```

Insert Nodes

C++17

members of the return type

```
.insert(8) → {.position|.inserted|.node}
.insert(1) → {.position|.inserted|.node}
.insert( ) → {.position|.inserted|.node}
.insert(@hint, 8) → @inserted
.insert(@hint, 1) → @blocking
```

true / false

(empty node)

(empty)

Modify Key

Direct modification not allowed! Instead: **extract** key, **modify** its value and **re-insert**.

```
unordered_set s {1,7,5};
auto node = s.extract(5);
if (node) { // if key existed
  node.value() = 8;
  s.insert(move(node));
}
```

std::unordered_multimap<KeyT,MappedT,Hash,KeyEq> (multiple equiv. keys allowed)

Construct A New Map Object

```
unordered_map<string,int> m0 { }
unordered_map<string,int> m1 {{"A",2},{"Z",4},{"Y",7}}
unordered_map<string,int> m2 (begin(m1)+1, end(m1))
unordered_map<string,int> deep_copy_of_m1(m1)
```

C++17 key and mapped types deducable from arguments

```
unordered_map m3 {{2, 3.14},{5, 6.0}}; // unordered_map<int,double>
```

Insert A Single Key-Value Pair

$\mathcal{O}(1)$ avg., $\mathcal{O}(n)$ worst

```
.insert({"W",2}) → {@inserted, true}   (inserted yes/no)
.insert({"A",9}) → {@blocking, false}
```

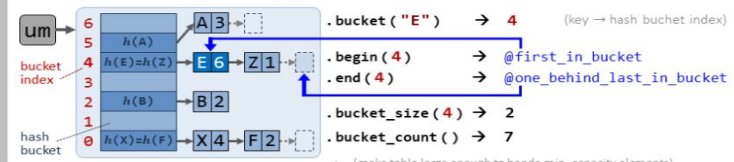
potential performance benefit by hinting at probable insert position

```
.insert (@hint, {"W",2}) → @instd/blockng
```

Query & Control Hash Table Properties

hash function: $h(key) \mapsto$ bucket index

```
unordered_map<string,int> um {{"E",6},{"X",4},{"Z",1},{"A",3},{"F",2},{"B",2}};
```




```
um.load_factor() → 4/7 = 0.57
um.max_load_factor(0.8)   (set)
um.max_load_factor() → 0.8   (get)
```

(make table large enough to hande min_capacity elements)

```
.reserve(min_capacity) →
.rehash(#hash_buckets) →
```

(can allocate even more buckets if max. load factor demands it)

Assign New Content To An Existing Map

```
unordered_map<string,int> m1 {{"X",2}};
unordered_map<string,int> m2 {{"A",1},{"G",4}};
m1 = m2;
```

(deep copy from source)

new state of m1

Insert Multiple Key-Value Pairs

$\mathcal{O}(\#ins)$ avg., $\mathcal{O}(n \cdot \#ins + \#ins)$ worst

```
.insert({{"G",4},{"K",9},{"A",7}}) →
.insert(@inBegin, @inEnd) →
```

source container

Lookup Using Keys as Input

$\mathcal{O}(1)$ average, $\mathcal{O}(n)$ worst case

```
unordered_map<string,int> m {{"S",3},{"X",5},{"F",1}};
.contains("W") → false   C++20
.contains("X") → true
.count("W") → 0
.count("X") → 1
```

can be > 1 only for unordered_multimap

```
.find("W") → @end (= no match)
.find("X") → @match
.equal_range("F") → {@1st_equal, @after}
.at("F") → 1
.at("B") → Throws std::out_of_range
```

Construct Key-Value Pair

$\mathcal{O}(1)$ avg., $\mathcal{O}(n)$ worst

(inserted yes/no)

```
.emplace("W",2) → {@inserted, true}
```

potential performance benefit by hinting at probable insert position

```
.emplace_hint(@hint,"W",2) → @inserted
.try_emplace("W",2) → {@inserted, true}
```

C++17

advantage: does not move from rvalue input parameters if not inserted

Insert or Assign Value

$\mathcal{O}(1)$ avg., $\mathcal{O}(n)$ worst C++17

(inserted yes/no)

```
.insert_or_assign("B",5) → {@as, false}
.insert_or_assign("R",6) → {@ins, true}
```

potential performance benefit by hinting at probable insert position

```
.insert_or_assign(@hint,"B",5) → @as
.insert_or_assign(@hint,"G",2) → @ins
```

Access / Modify Value

$\mathcal{O}(1)$ avg., $\mathcal{O}(n)$ worst

```
unordered_map<string,int> m {{"F",1},{"A",3}};
["A"] → 3
["A"] = 4 →
```

Attention: $[k]$ inserts new pair if key k is not present!

```
["W"] = 2 →
["W"] → 0
```

newly created mapped values are *value-initialized* (e.g. 0 for int)

Erase Key-Value-Pair(s)

$\mathcal{O}(\#erased)$ avg. $\mathcal{O}(n)$ worst case

```
.erase("W") → 0
.erase("A") → 1
.erase(@pos) → @after
.erase(@beg,@end) → @after
```

Modify Key

```
unordered_map<string,int> m {{"F",1},{"A",3}};
auto node = m.extract("A");
if (node) { node.key() = "X";
m.insert(move(node));
}
```

Extract Nodes

to efficiently transfer key-value pairs C++17

```
.extract("R") → R 2
.extract(@pos) → R 2
```

$\mathcal{O}(1)$ avg., $\mathcal{O}(n)$ worst

(Re-)Insert Nodes

members of the return type

```
.insert(N 5) → {.position|.inserted|.node}
.insert(A 6) → {.position|.inserted|.node}
.insert( ) → {.position|.inserted|.node}   (empty node)
.insert(@hint, X 5) → @inserted
.insert(@hint, F 6) → @blocking
```

(empty)

Merge Two Maps

C++17

```
unordered_map<string,int> m1 {{"F",1},{"S",3},{"X",5}};
unordered_map<string,int> m2 {{"A",2},{"X",7}};
.merge( A 2 X 7 )
```

$\mathcal{O}(n_2)$ average, $\mathcal{O}(n_1 \cdot n_2 + n_2)$ worst case

Query Size

```
.empty() → false
.size() → 2
```

Erase All

```
.clear()
```

Obtain Iterators

```
.begin() → @first
.end() → @one_behind_last
```

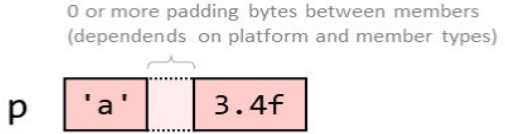
Special Containers

pair<A,B>

#include <utility>

contains *two* values
of same or different type

```
std::pair<char,float> p;
p.first  = 'a';
p.second = 3.4f;
char  x = std::get<0>(p);
float y = std::get<1>(p);
```

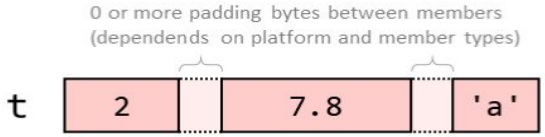


tuple<A,B,C,…>

#include <tuple>

contains *many* values
of same or different type

```
std::tuple<int,double,char> t { 2, 7.8, 'a' };
int    x = std::get<0>(t);
double y = std::get<1>(t);
char   z = std::get<2>(t);
auto [u,v,w] = t;  // u: 2  v: 7.8  w:  'a'
```

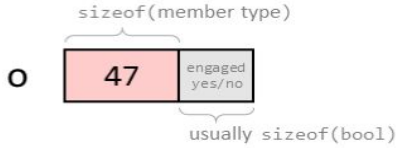


optional<*T*>

#include <optional>

either contains
one value of type *T* or *no* value

```
std::optional<int> o;
bool b = o.has_value(); // false
o = 47;          // has value '47' now
if(o) { std::cout << *o; }
o.reset();       // disengage => no value
```

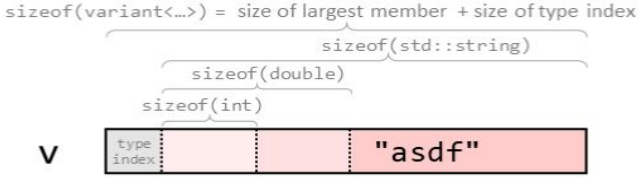


variant<*A*,*B*,*C*,…>

#include <variant>

contains *one* value of either
type *A* or type *B* or type *C*, etc.

```
std::variant<int,string,double> v { 47 };
bool b = std::holds_alternative<int>(v);
v = std::string("asdf");
auto i = v.index();       // 1 ⇔ string
```

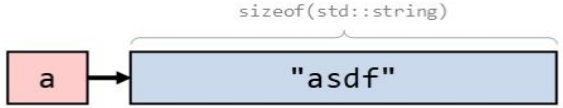


any

#include <any>

contains *one* value of any type

```
std::any a = 5;
a = std::string("abc");
a = std::set<int> {3,1,47,8,6};
try { int x = std::any_cast<int>(a); }
catch(std::bad_any_cast&) { /* … */ }
```



```cpp
int main() {
    std::stack<int> s;

    // Pushing elements onto the stack (O(1) for each push)
    s.push(10);  // Stack: [10]
    s.push(20);  // Stack: [10, 20]
    s.push(30);  // Stack: [10, 20, 30]

    // Accessing the top element (O(1))
    std::cout << "Top element: " << s.top() << std::endl;  // Output: 30

    // Popping the top element (O(1))
    s.pop();  // Stack: [10, 20]

    // Checking the size of the stack (O(1))
    std::cout << "Current size: " << s.size() << std::endl;  // Output: 2

    // Checking if the stack is empty (O(1))
    if (s.empty()) {
        std::cout << "The stack is empty." << std::endl;
    } else {
        std::cout << "The stack is not empty." << std::endl;
    }

    return 0;
}
```

stack<T>

```cpp
int main() {
    // Max-heap (default)
    std::priority_queue<int> maxHeap;

    // push() - O(log n) for each element
    maxHeap.push(10); maxHeap.push(20); maxHeap.push(5);

    // top() - O(1)
    std::cout << "Max-heap top: " << maxHeap.top() << std::endl;  // Output: 20

    // pop() - O(log n)
    maxHeap.pop();  // Removes 20

    // Min-heap (using std::greater)
    std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;

    // push() - O(log n) for each element
    minHeap.push(10); minHeap.push(20); minHeap.push(5);

    // top() - O(1)
    std::cout << "Min-heap top: " << minHeap.top() << std::endl;  // Output: 5

    // pop() - O(log n)
    minHeap.pop();  // Removes 5

    return 0;
}
```

priority_queue<T>