string s = "I'm sorry, Dave.";

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	indices

non-mutating

s.**size**()	→	16	
s[0]	→	'I'	(character at position 0)
s.**find**("r")	→	6	(first occurrence from start)
s.**rfind**("r")	→	7	(first occurrence from end)
s.**find**("X")	→	string::npos	(not found, invalid index)
s.**find**(' ', 5)	→	10	(first occurr. starts at 5)
s.**substr**(4, 6)	→	string{"sorry,"}	
s.**contains**("sorry")	→	*true*	(C++23)
s.**starts_with**('I')	→	*true*	(C++20)
s.**ends_with**("Dave.")	→	*true*	(C++20)
s.**compare**("I'm sorry, Dave.")	→	0	(identical)
s.**compare**("I'm sorry, Anna.")	→	> 0	(same length, but 'D' > 'A')
s.**compare**("I'm sorry, Saul.")	→	< 0	(same length, but 'D' < 'S')

mutating

s **+=** " I'm afraid I can't do that."	⇒	s = "I'm sorry, Dave. **I'm afraid I can't do that.**"	
s.**append**("..")	⇒	s = "I'm sorry, Dave..."	
size			
s.**clear**()	⇒	s = ""	
s.**resize**(3)	⇒	s = "I'm"	
s.**resize**(20, '?')	⇒	s = "I'm sorry, Dave.????";	
index based			
s.**insert**(4, "very ")	⇒	s = "I'm **very** sorry, Dave."	
s.**erase**(5, 2)	⇒	s = "I'm **sry**, Dave."	
s[15] = '!'	⇒	s = "I'm sorry, Dave**!**"	
s.**replace**(11, 5, "Frank")	⇒	s = "I'm sorry, **Frank**"	
iterator based			
s.**insert**(s.begin(), "HAL: ")	⇒	s = "**HAL:** I'm sorry, Dave."	
s.**insert**(s.begin()+4, "very ")	⇒	s = "I'm **very** sorry, Dave."	
s.**erase**(s.begin()+5)	⇒	s = "I'm **srry**, Dave."	
s.**erase**(s.begin(), s.begin()+4)	⇒	s = "sorry, Dave."	

Constructors

string{'a','b','c'} ➡ a b c

string(4, '$') ➡ $ $ $ $

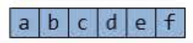
string(@firstIn, @lastIn) ➡ e f g h

string(a b c d **)** copy/move ➡ a b c d

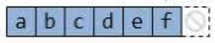
source string object

Obtain Iterators or Reverse Iterators

.begin() → *@first*

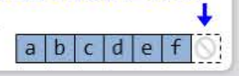


.rbegin() → *reverse@last* .base()



.end() → *@one_behind_last*

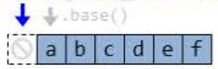
don't use to access elements!



.rend() → *reverse@one_before_first*

don't use to access elements!

.base()



String → Number Conversion

```
          int       stoi (●,●,●);
          long      stol (●,●,●);
          long long stoll(●,●,●);

unsigned long       stoul (●,●,●);
unsigned long long  stoull(●,●,●);

        float       stof (●,●,●);
        double      stod (●,●,●);
        long double stold(●,●,●);
```

- const string& — input string
- std::size_t* p = nullptr — output for number of processed characters
- int base = 10 — base of target system; default: decimal

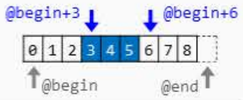
Number → String Conversion

```
string to_string( ● );
```

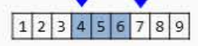
int | long | long long | unsigned | unsinged long | unsigned long long | float | double | long double

C++ Standard Library Algorithms

Iterator Ranges



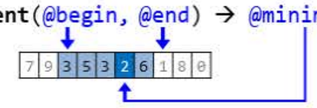
distance(@begin, @end) → element_count



<iterator>

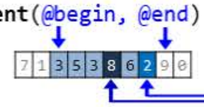
Non-Modifying Sequence Operations

min_element(@begin, @end) → @minimum



minmax_element(@begin, @end) → {@minimum, @maximum}

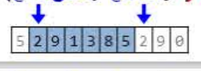
C++11



any_of (@begin, @end, *f*(○)↦bool) →
all_of (@begin, @end, *f*(○)↦bool) →
none_of(@begin, @end, *f*(○)↦bool) →

- true, if *f* yields true for *any*, *all* or *none* elements in the input range
- false otherwise

C++11

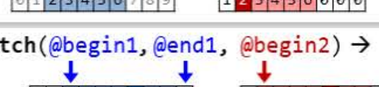


find_if(@begin, @end , *f*(○)↦bool)→
find (@begin, @end , value)→

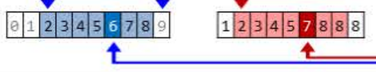
- @1st match
- @end if no match

count_if(@begin, @end , *f*(○)↦bool) →
count (@begin, @end , value) → number of occurrences

equal(@begin1, @end1, @begin2) → true if all elements in both ranges are equal

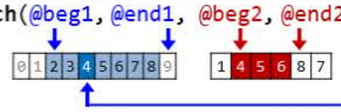


mismatch(@begin1, @end1, @begin2) → {@mismatch_in1, @mismatch_in2}



search(@beg1, @end1, @beg2, @end2) →

- @1st occurrence of sequence 2 inside sequence 1
- @end1 otherwise



Binary Search On *Sorted* Sequences ⇒ $\mathcal{O}(\log n)$

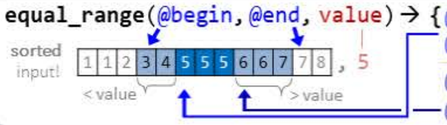
lower_bound(@begin, @end, value) →

- @1st element **not** < value
- @end if no such element exists

upper_bound(@begin, @end, value) →

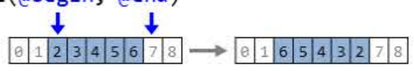
- @1st element > value
- @end if no such element exists

equal_range(@begin, @end, value) → {@1st item **not** < value or @end if none such found), (@1st item > value or @end if none such found)**}**



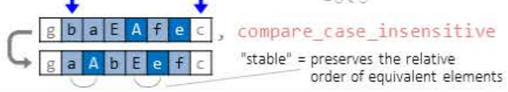
Reordering Elements

reverse(@begin, @end)

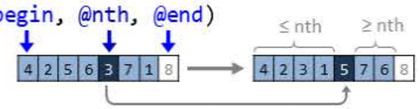


sort(@begin, @end, *f*(○,○)↦bool)
sort(@begin, @end, std::less)

stable_sort(@begin, @end, compare(○,○)↦bool)
=○<○



nth_element(@begin, @nth, @end)



partition(@begin, @end, *f*(○)↦bool) → @part2

rotate(@begin, @newfst, @end) → @old_begin

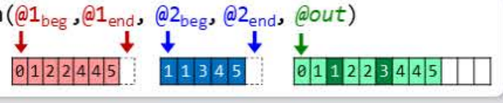
next_permutation(@begin, @end) → *true* if new permutation is lexicographically greater

shuffle(@begin, @end, random_engine)

Manipulate *Sorted* Sequences ⇒ $\mathcal{O}(n)$

merge(@1beg, @1end, @2beg, @2end, @out)

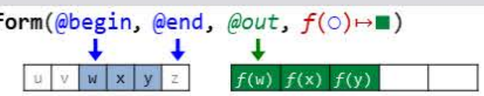
set_union(@1beg, @1end, @2beg, @2end, @out)



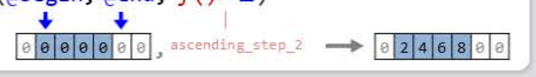
Changing Values

copy(@begin, @end, @out)

transform(@begin, @end, @out, *f*(○)↦■)



generate(@begin, @end, *f*()↦■)



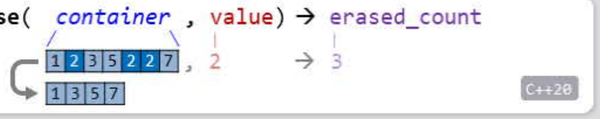
replace(@begin, @end, old, new)

replace_if(@begin, @end, *f*(○)↦bool, new)

remove (@begin, @end, value) → @end_of_
remove_if(@begin, @end, *f*(○)↦bool) → remaining

unique(@begin, @end) → @end_of_remaining

erase(*container* , value) → erased_count



C++20

Numeric Algorithms

<numeric>

reduce(@begin, @end, ω = ●{}, ⊕ = ○+○)
reduce(@begin, @end, ω, ⊕(□,○)↦■) → ω+●2+●0+●n +⋯+●1+●3

C++17

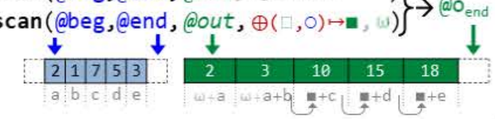
arbitrary evaluation order!

transform_reduce(@●,@●, @♦, ω, ⊕ = □+○ , ⊗ = ○×◊)
transform_reduce(@●,@●, @♦, ω, ⊕(□,Δ)↦■, ⊗(○,◊)↦▲) → $R_{\otimes}$
transform_reduce(@●,@●, ω, ⊕(□,Δ)↦■, *f*(○)↦▲) → R_f

arbitrary evaluation order!

C++17 $R_f = \omega + f(\bullet_0) + f(\bullet_1) + \cdots$ $R_{\otimes} = \omega + (\bullet_0 \times \blacklozenge_0) + (\bullet_1 \times \blacklozenge_1) + \cdots$

inclusive_scan(@beg,@end, @out, ⊕ = □+○)
inclusive_scan(@beg,@end, @out, ⊕(□,○)↦■, ω) → @o_end



C++17

h/cpp hackingcpp.com

v3.1

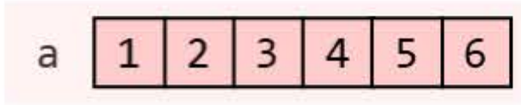
C++ Standard Library Sequence Containers

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

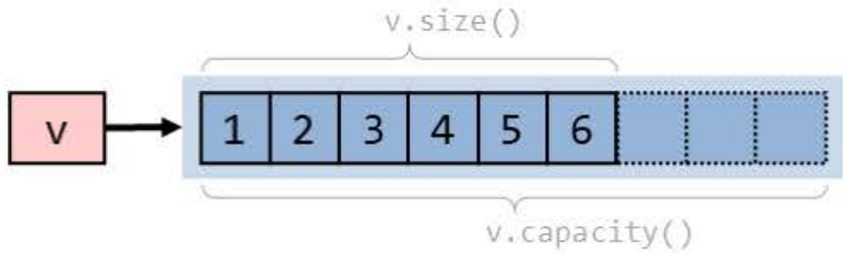
vector<*T*>

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
```



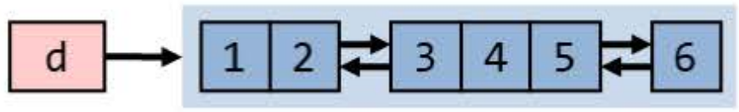
contiguous memory; random access;
fast linear traversal; fast insertion/deletion at the ends

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
```



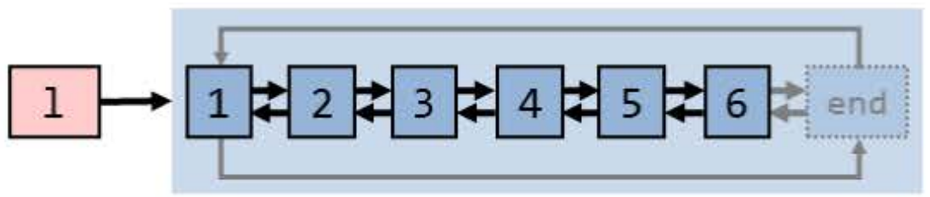
fast insertion/deletion at both ends

list<*T*>

doubly-linked list

```
#include <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();
```



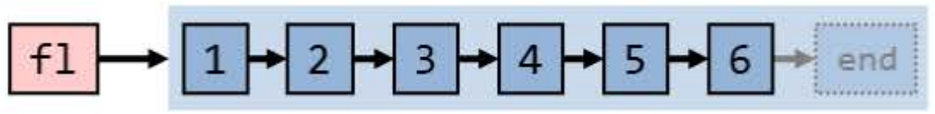
fast splicing; many operations without copy/move of elements

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

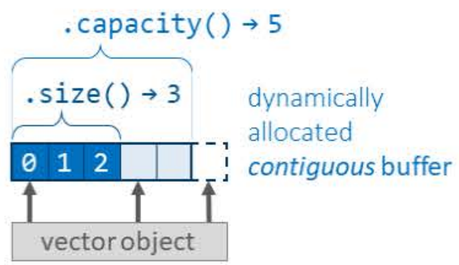
Construct A New Vector Object

```
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

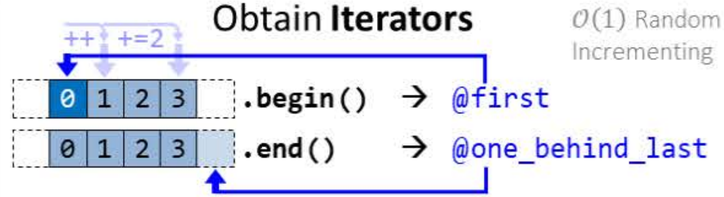
```
vector w {7,4,2};   // vector<int>
```

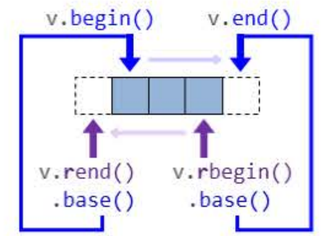
Typical Memory Layout



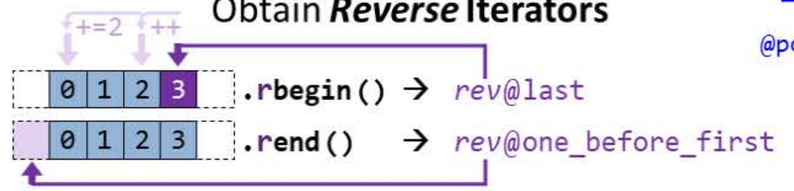
Obtain **Iterators**

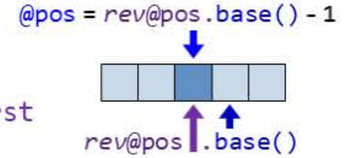
$\mathcal{O}(1)$ Random Incrementing





Obtain ***Reverse*** Iterators



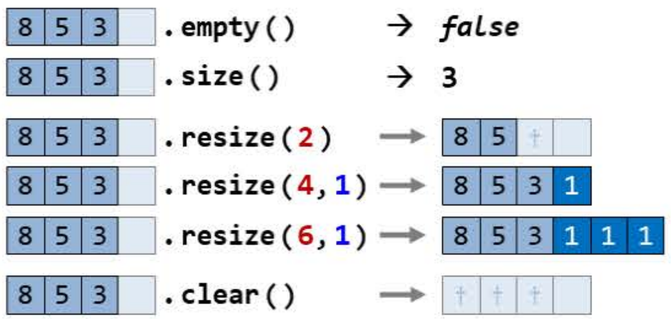


Assign New Content To An Existing Vector

```
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

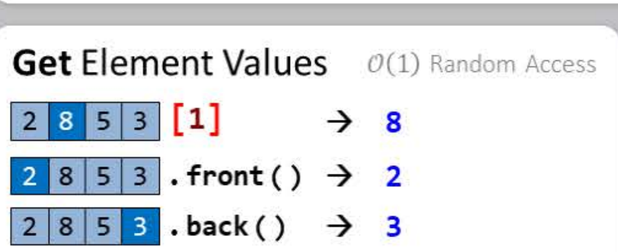
Insert Elements at Arbitrary Positions

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

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

Get Element Values

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



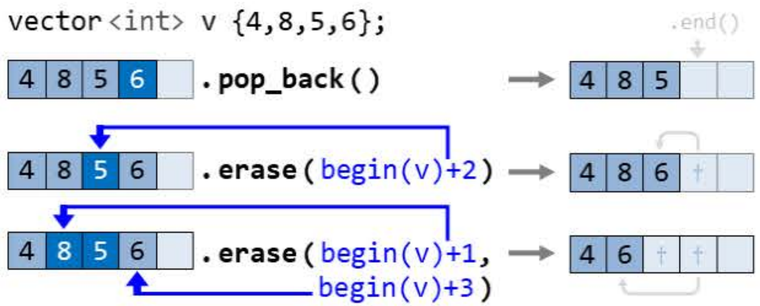
Change Element Values

Out of Bounds Access

Erase Elements

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

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



Shrink The Capacity

(might be inefficient)

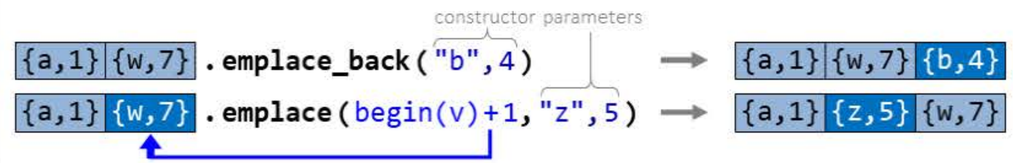
Erasing, resizing or clearing will not shrink the capacity!

```
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 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: 3 2 1 1 2 3

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

$\mathcal{O}$(#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: 3 1 8 9 2

Insert & **Construct** Elements **Without Copy / Move**

$\mathcal{O}(1)$

```
list<pair<string,int>> L {{"a",1},{"w",7}};
.emplace_back("b",4)
.emplace_front("c",6)
.emplace(next(begin(L)),"z",5)
```

constructor parameters

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)), )                          O(1)
.splice(next(begin(T)), , next(begin(S)))          O(1)
.splice(next(begin(T)), , 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( 0 1 ··· M M+1 , M)  →  @Mth_element
```

$\mathcal{O}(M)$

Reorder Elements

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

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

Erase Elements Based on **Positions**

$\mathcal{O}$(#deleted)

```
list<int> L {8,4,3,5,3};
.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)
.remove_if(is_even)
.unique()
.unique(equal_abs)
```

$f(\circ) \mapsto$ bool

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

```
→ 3
→ 4
→ 2
→ 2
```

C++20

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

A singly-linked list of nodes each holding one value.

C++11

Construct A New List Object

```
forward_list<int> L1 {8,5,4}
forward_list<int> L2 (next(begin(L1)),end(L1))
forward_list<int> L3 (3,1)
forward_list<int> deep_copy_of_L1 (L1)
```

C++17

value type deducable from argument type

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

Insert Elements at Arbitrary Positions

$\mathcal{O}$(#inserted)

```
forward_list<int> L {0,1,2};
```

```
.insert_after(begin(L), 7)
.insert_after(begin(L), 2,8)
.insert_after(begin(L), {6,4})
.insert_after(begin(L), @b, @e)
```

source container

Obtain **Iterators**

$\mathcal{O}(1)$

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

Increment Iterators

$\mathcal{O}(M)$

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

Assign New Content To An Existing List

```
forward_list<int> L1 {8,5,3};
forward_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

Insert & **Construct** Elements **Without Copy / Move**

$\mathcal{O}(1)$

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

constructor parameters

```
.emplace_front("c",6)
.emplace_after(begin(L), "z",5)
```

Prepend Elements

$\mathcal{O}(1)$

```
.push_front(7)
```

Splice (Elements From) One List Into Another One

does not copy or move elements!

```
forward_list<int> T {8,5,3};   forward_list<int> S {7,9,2};
```

source list

$\mathcal{O}(1)$

```
.splice_after(begin(T),
.splice_after(begin(T),
.splice_after(begin(T),
```

$\mathcal{O}$(#inserted)

Reorder Elements

```
.sort()
.sort(std::greater<>{})      f(○,○)↦bool
.reverse()
```

Access Element Values

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

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

Access Arbitrary Elements Using Iterators

```
forward_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
```

Merge Already **Sorted** Lists

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

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

```
forward_list<int> L1 {2,4,5};
forward_list<int> L2 {1,3};
.merge( )
```

Erase Elements Based on **Positions**

$\mathcal{O}$(#deleted)

```
forward_list<int> L {0,1,2};
.pop_front()
.erase_after( L.before_begin())
.erase_after(begin(L))
.erase_after(begin(L), end(L) )
```

Check **Emptiness** / Change **Size** (= Number of Elements)

There's no member function available to determine the size!

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

Erase Elements Based on **Values**

$\mathcal{O}(n)$

```
.remove(7)          f(○)↦bool
.remove_if(is_even)
.unique()           f(○,○)↦bool
.unique( equal_abs )
```

Construct A New Deque Object

```
deque <int> d1 {2,9,1,8,5,4}
```
→ 2 9 1 8 5 4

```
deque <int> d2 (begin(d1)+3, end(d1))
```
→ 8 5 4

```
deque <int> d3 (5, 3)
```
→ 3 3 3 3 3

```
deque <int> deep_copy_of_d1 (d1)
```
→ 2 9 1 8 5 4

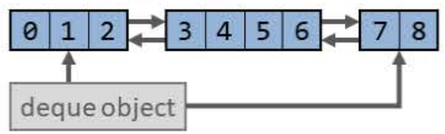
C++17 value type deducable from argument type

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

Typical Memory Layout

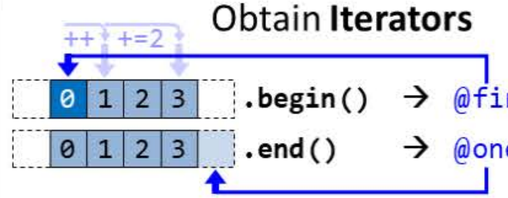
Note that the ISO standard only specifies the properties of deque (e.g., constant-time insert at both ends) but not how that should be implementd.

dynamically allocated contiguous chunks

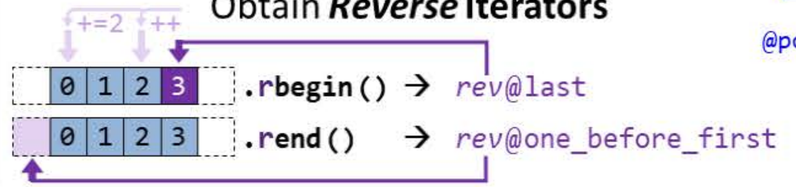


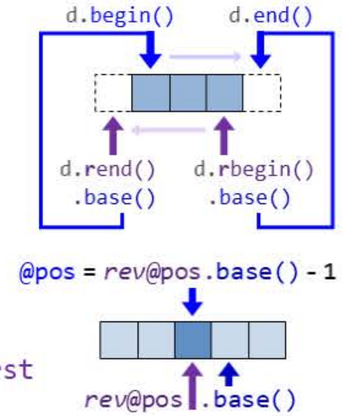
Obtain **Iterators**

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



Obtain ***Reverse*** Iterators





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

8 5 3 = 6 8 1 9 → 6 8 1 9

8 5 3 .assign({4,1,3,5}) → 4 1 3 5

8 5 3 .assign(2, 1) → 1 1

8 5 3 .assign(@InBeg,@InEnd) → 2 1 1 2

source container: 3 2 1 1 2 3

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

8 5 3 .empty() → *false*

8 5 3 .size() → 3

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

8 5 3 .resize(2) → 8 5

8 5 3 .resize(4,1) → 8 5 3 1

8 5 3 .resize(6,1) → 8 5 3 1 1 1

8 5 3 .clear() →

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

8 5 3 .push_back(7) → 8 5 3 7

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

8 5 3 .push_front(7) → 7 8 5 3

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

```
deque <int> d {8,5,3};
```

8 5 3 .insert(begin(d)+1, 7) → 8 7 5 3

8 5 3 .insert(begin(d)+1, 3,7) → 8 7 7 7 5 3

8 5 3 .insert(begin(d)+1, {6,9,7}) → 8 6 9 7 5 3

8 5 3 .insert(begin(d)+1, @InBegin,@InEnd) → 8 1 8 9 5 3

source container: 3 1 8 9 2 3

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

2 8 5 3 [1] → 8

2 8 5 3 .front() → 2

2 8 5 3 .back() → 3

Change Element Values

2 8 5 3 [1] = 7 → 2 7 5 3

2 8 5 3 .front() = 7 → 7 8 5 3

2 8 5 3 .back() = 7 → 2 8 5 7

Out of Bounds Access

2 8 5 3 [6] → *Undefined Behavior*

Invalid Index!

2 8 5 3 .at(6) → Throws Exception `std::out_of_range`

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

4 8 5 6 .pop_back() → 4 8 5

4 8 5 6 .pop_front() → 8 5 6

Erase Elements At Arbitrary Positions

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

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

4 8 5 6 .erase(begin(d)+2) → 4 8 6

4 8 5 6 .erase(begin(d)+1, begin(d)+3) → 4 6

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

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

constructor parameters

{a,1} {w,7} .emplace_back("b",4) → {a,1} {w,7} {b,4}

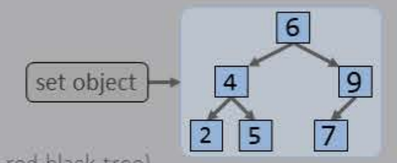
{a,1} {w,7} .emplace_front("c",6) → {c,6} {a,1} {w,7}

{a,1} {w,7} .emplace(begin(d)+1,"z",5) → {a,1} {z,5} {w,7}

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**

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

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

C++20

```
.count(2) → 0
.count(5) → 1
```

can be > 1 only for std::**multi**set

```
.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) → {@inserted,true}   // .first .second
.insert(5) → {@blocking,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})
.insert(@inB,@inE)
```

source container

Insert & **Construct** A Key **in Place**

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

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

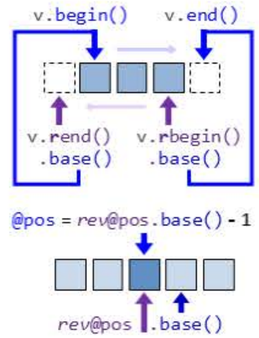
$\mathcal{O}(\log n + \#\text{erased})$

Obtain **Iterators**

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

Obtain ***Reverse*** **Iterators**

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



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}   true   (empty)
.insert(1) → {.position|.inserted|.node}   false   1
```

(empty node)

```
.insert( ) → {.position|.inserted|.node}   false   (empty)
.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().base() v.rbegin().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

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

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

source container

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] → ""
```

Extract Nodes

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

```
.extract(2)
.extract(@pos)
```

(Re-)Insert Nodes

members of the return type

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

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
```

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

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)
```

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));
}
```

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()
```

Query Size (= number of key-value pairs)

```
.empty() → false
.size()  → 2
```

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

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;
```

[1 5 3 7] = [8 4 6] → new state of s1: [8 4 6]

Key Lookup

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

```
[1 5 3 7] .contains(2)  →  false        C++20
[1 5 3 7] .contains(3)  →  true

[1 5 3 7] .count(2)  →  0   } can be > 1 only for
[1 5 3 7] .count(3)  →  1   } std::unordered_multiset

[1 5 3 7] .find(2)  →  @end    (= no match)
[1 5 3 7] .find(3)  →  @match

[1 5 3 7] .equal_range(5)  →  {@first_equal, @after}
```

Query Size (= Number of Keys)

```
[1 5 3] .empty()  →  false
[1 5 3] .size()   →  3
```

Erase All Keys

```
[1 5 3] .clear()
```

Obtain Iterators (to keys)

```
[3 1 5] .begin()  →  @first
[3 1 5] .end()    →  @one_behind_last
```

Insert A Single Key

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

```
[2 8 5] .insert(4)  →  {@inserted, true}      (.first, .second)
                       [2 8 4 5]

[2 8 5] .insert(8)  →  {@blocking, false}     (not inserted)
                       [2 8 5]
```

potential performance benefit by hinting at probable insert position

```
[2 8 5] .insert(@hint, 7)  →  @inserted_or_block
                              [2 8 7 5]
```

Insert Multiple Keys

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

```
[2 8 5] .insert({1,6,8})      →  [6 2 5 1 8]
[2 8 5] .insert(@inB,@inE)    →  [2 4 5 3 8]
```

source container: [3 3 2 4 2]

Insert & Construct A Key in Place

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

```
unordered_set<pair<int,int>> s {{1,3},{5,6}};

[1,3 5,6] .emplace(8,7)  →  {@inserted, true}     (if inserted)
   pair constructor arguments   [1,3 8,7 5,6]
```

potential performance benefit by hinting at probable insert position

```
[1,3 5,6] .emplace_hint(@hint,8,7)  →  @inserted
                                       [1,3 8,7 5,6]
```

Erase One Key or A Range of Keys

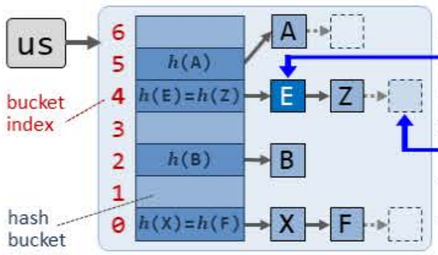
$\mathcal{O}(\#erased)$ avg. $\mathcal{O}(n)$ worst case

```
[1 5 3 7] .erase(2)  →  0   } #erased
[1 5 3 7] .erase(3)  →  1   }
                        [1 5 7]

[1 5 3 7] .erase(@pos)  →  @after_erased
                           [1 5 7]

[1 5 3 7] .erase(@beg,@end)  →  @after_erased
                                [1 7]
```

Query & Control Hash Table Properties

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

```
unordered_set<char> us {'E','X','Z','A','F','B'};
```




```
us.bucket('E')  →  4        (key → hash buchet index)
us.begin(4)     →  @first_in_bucket
us.end(4)       →  @one_behind_last_in_bucket
us.bucket_size(4)  →  2
us.bucket_count()  →  7
```

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

Extract Nodes

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

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

```
[1 5 7] .extract(5)  →  5
                        [1 7]

[1 5 7] .extract(@pos)  →  5
                           [1 7]
```

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};

[1 5 3] .merge( [8 1 2] )
   [1 8 5 3 2]    [1]
```

Insert Nodes

C++17

members of the return type: {.position | .inserted | .node}

```
[1 7] .insert(8)  →  {.position: [1 8 7], .inserted: true, .node: (empty)}

[1 7] .insert(1)  →  {.position: [1 7], .inserted: false, .node: 1}

(empty node)
[1 7] .insert( )  →  {.position: [1 7], .inserted: false, .node: (empty)}

[1 7] .insert(@hint, 8)  →  @inserted
                            [1 8 7]

[1 7] .insert(@hint, 1)  →  @blocking
                            [1 7]
```

Modify Key

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

```
unordered_set s {1,7,5};
   └→ [1 7 5]

auto node = s.extract(5);
   └→ 5        [1 7]

if (node) {  // if key existed
  node.value() = 8;
   └→ 8        [1 7]

  s.insert(move(node));
   └→          [1 8 7]
}
```

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}};
```

```
.bucket("E") → 4        (key → hash buchet index)
.begin(4) → @first_in_bucket
.end(4) → @one_behind_last_in_bucket
.bucket_size(4) → 2
.bucket_count() → 7
```

bucket index, hash bucket

```
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

```
.emplace("W",2) → {@inserted, true}   (inserted yes/no)
```

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

```
.insert_or_assign("B",5) → {@as, false}   (inserted yes/no)
.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
```

true / false / (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
```