

Алгоритмизация и программирование

Лекция 3-4

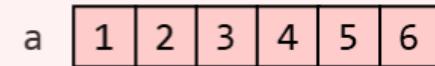
Последовательные контейнеры

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

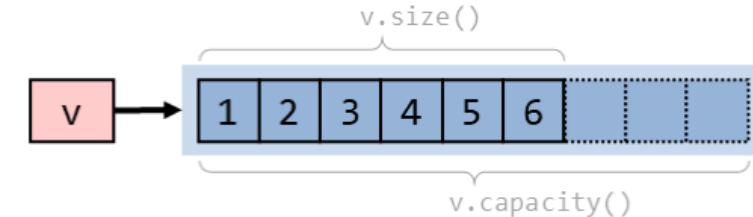
vector<T>

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
```



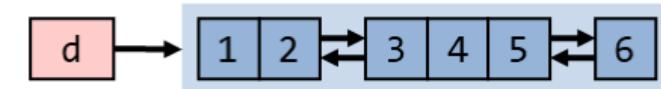
contiguous memory; random access; fast linear traversal; fast insertion/deletion at the ends

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
```



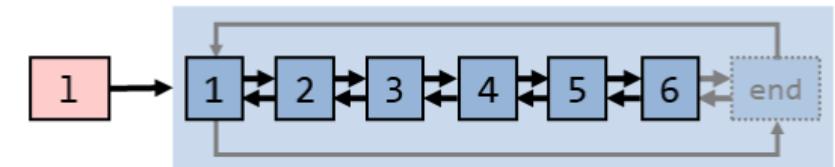
fast insertion/deletion at both ends

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();
```



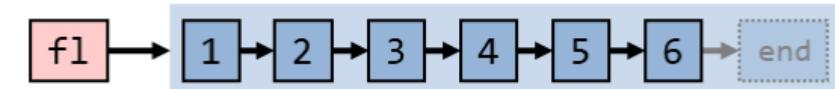
fast splicing; many operations without copy/move of elements

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);
```



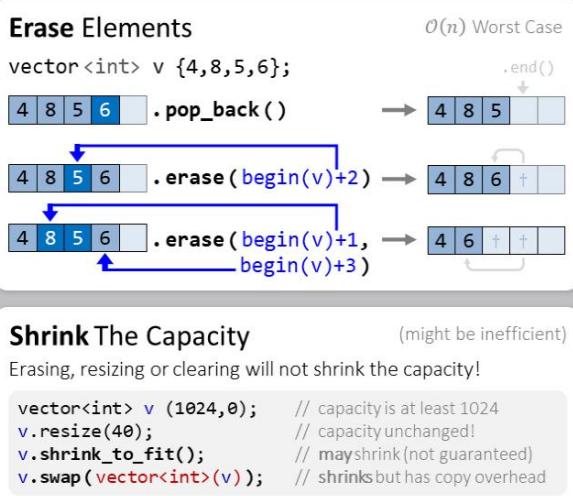
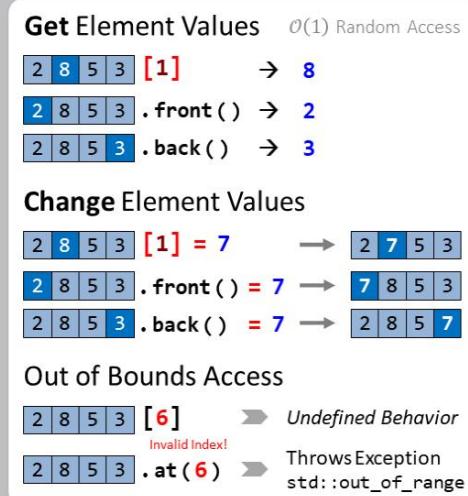
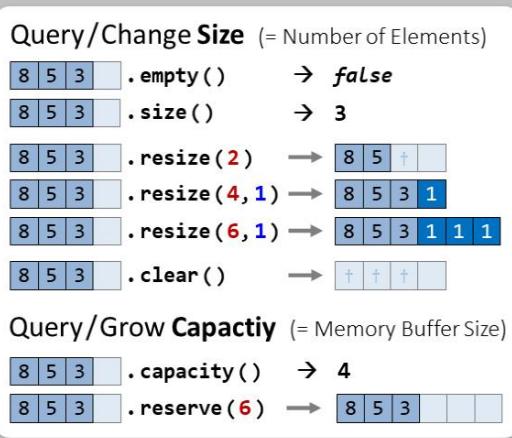
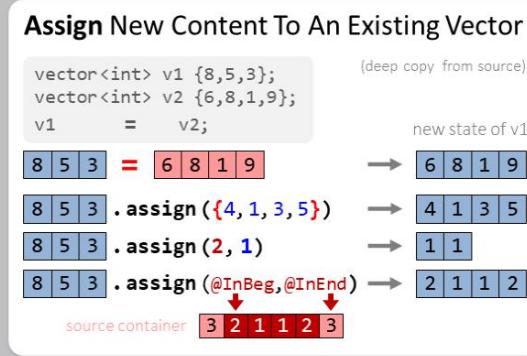
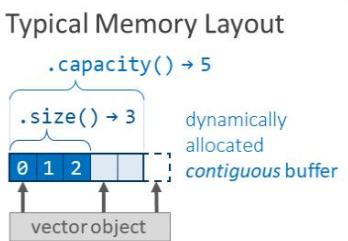
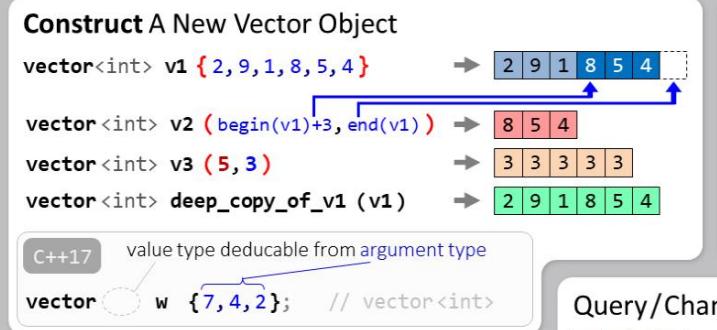
lower memory overhead than std::list; only forward traversal

std::array

std::vector

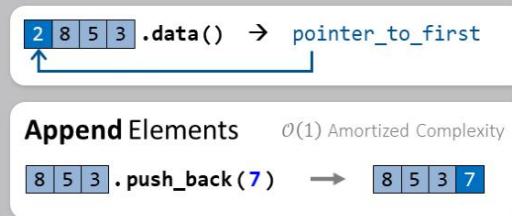
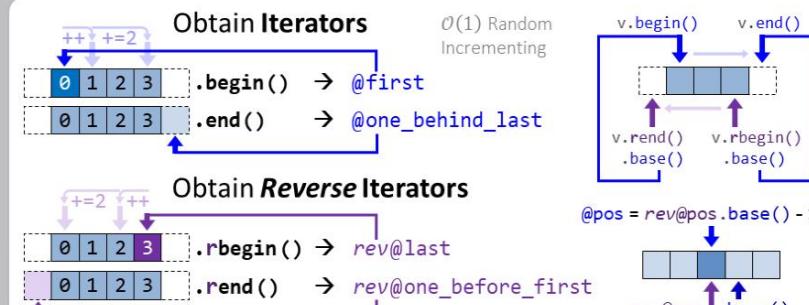
std::vector<ValueType>

C++'s "default" dynamic array

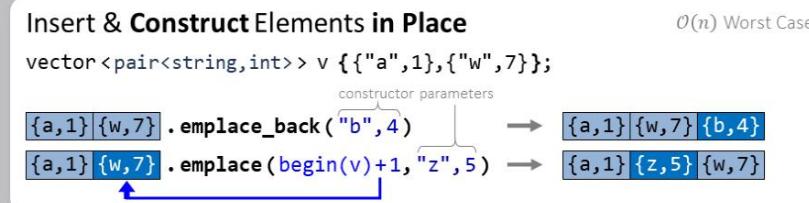
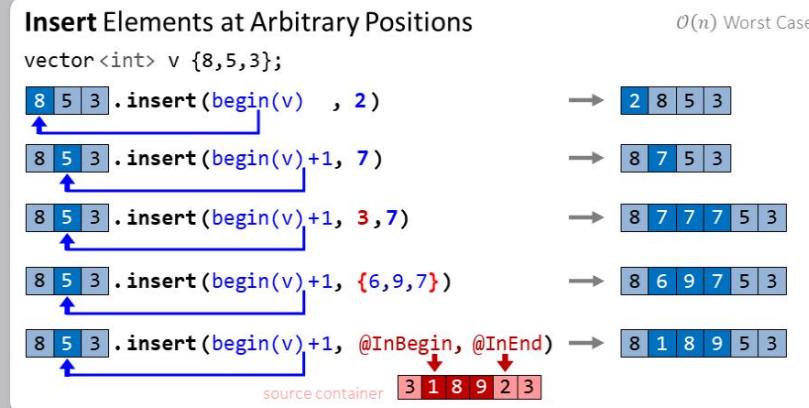


#include <vector>

h/cpp hackingcpp.com

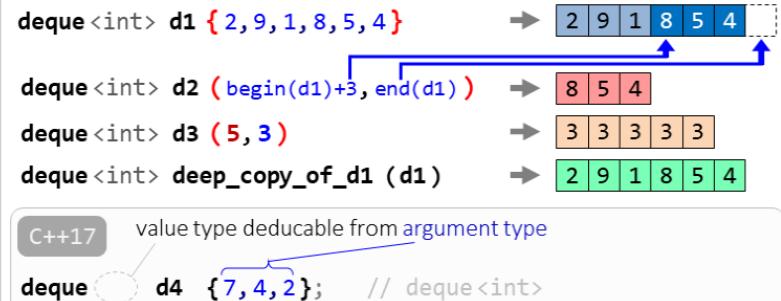


Avoid expensive memory allocations:
.reserve capacity before appending / inserting if you know the (approximate) number of elements to be stored in advance!



deque

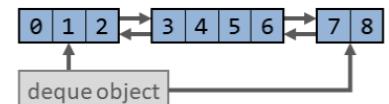
Construct A New Deque Object



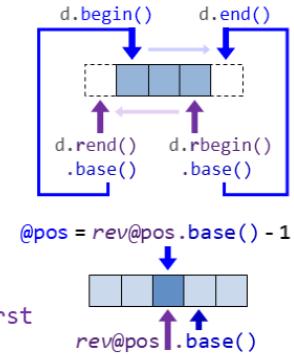
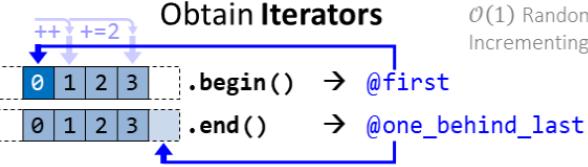
Typical Memory Layout

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 implemented.

dynamically allocated contiguous chunks

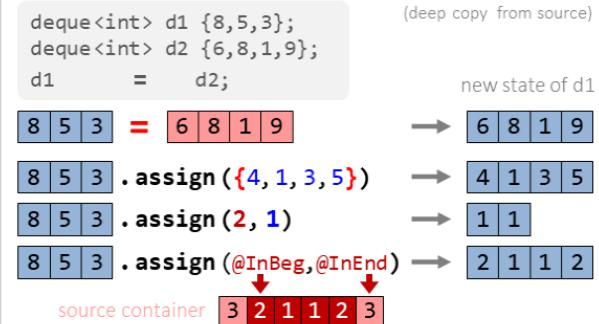


Obtain Iterators

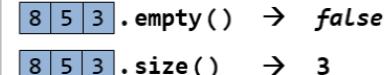


Obtain Reverse Iterators

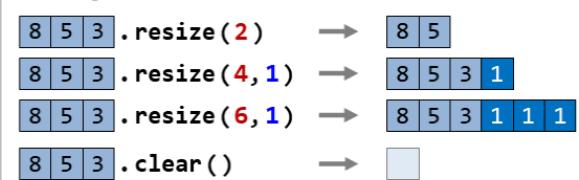
Assign New Content To An Existing Deque



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



Change Size

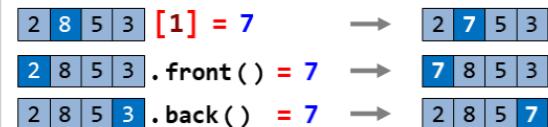


Get Element Values

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



Change Element Values

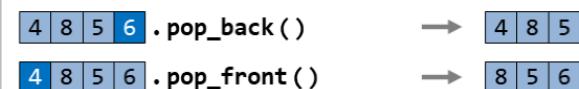


Out of Bounds Access



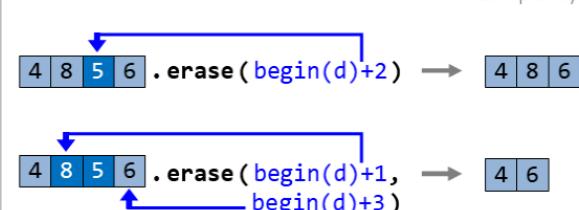
Erase Elements At The Ends

$\mathcal{O}(1)$



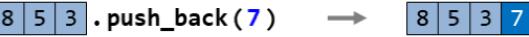
Erase Elements At Arbitrary Positions

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



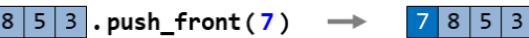
Append Elements

$\mathcal{O}(1)$



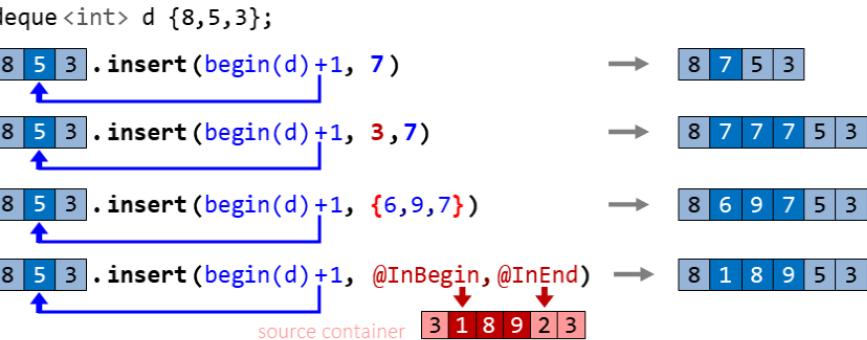
Prepend Elements

$\mathcal{O}(1)$



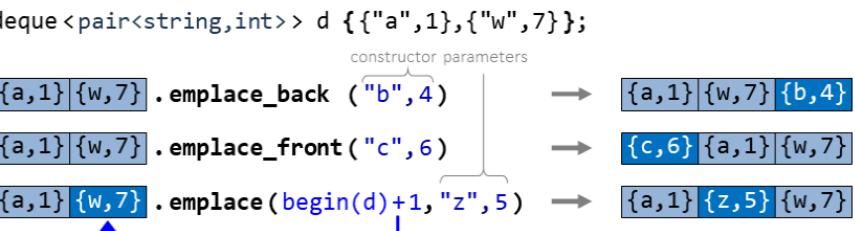
Insert Elements at Arbitrary Positions

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



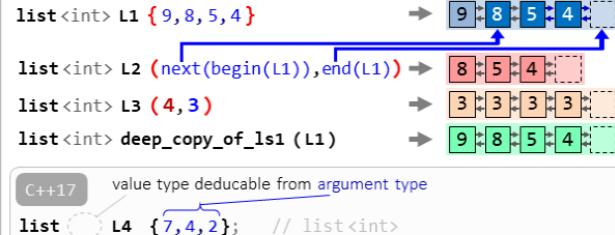
Insert & Construct Elements in Place

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

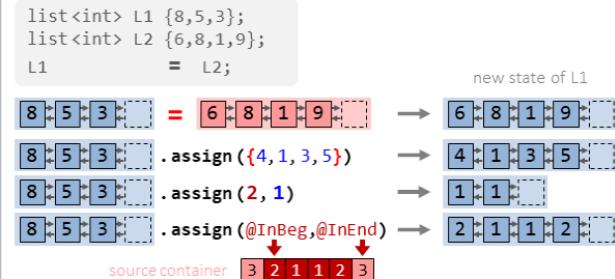


list

Construct A New List Object



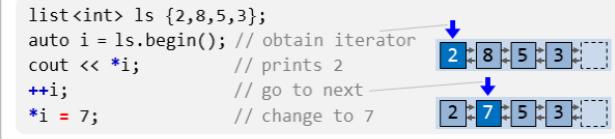
Assign New Content To An Existing List



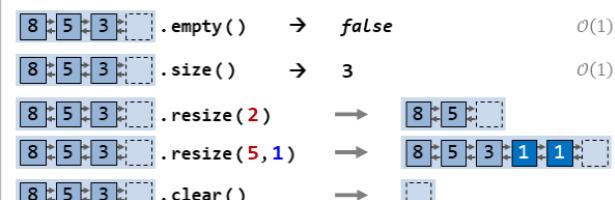
Access Element Values



Access Arbitrary Elements Using Iterators



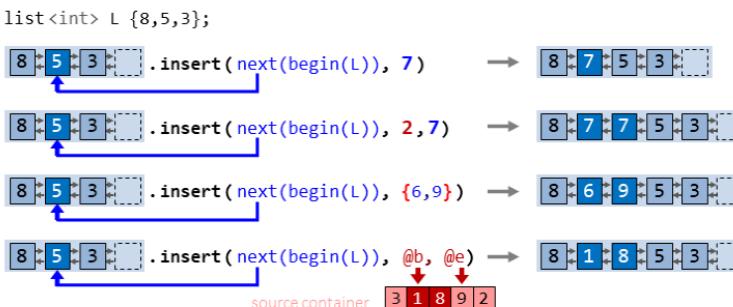
Query / Change Size (= Number of Elements)



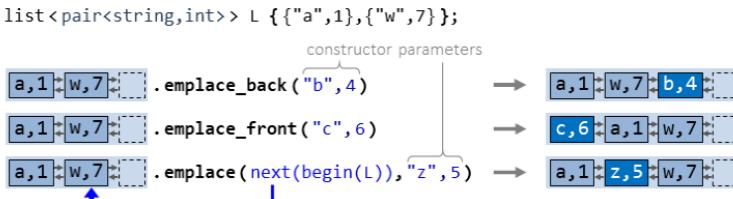
Append / Prepend Elements



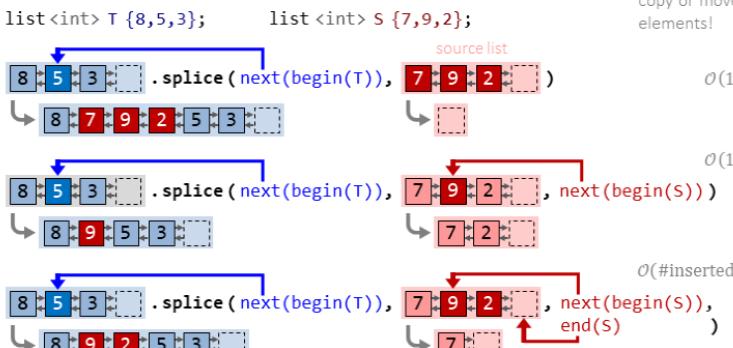
Insert Elements at Arbitrary Positions



Insert & Construct Elements Without Copy / Move



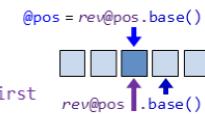
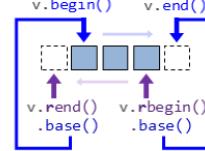
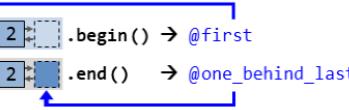
Splice (Elements From) One Lists Into Another One



Merge Already Sorted Lists



Obtain Iterators



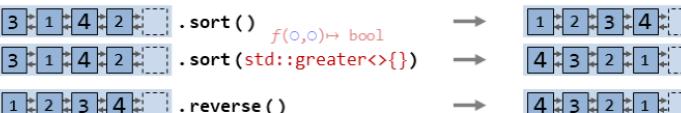
Obtain Reverse Iterators



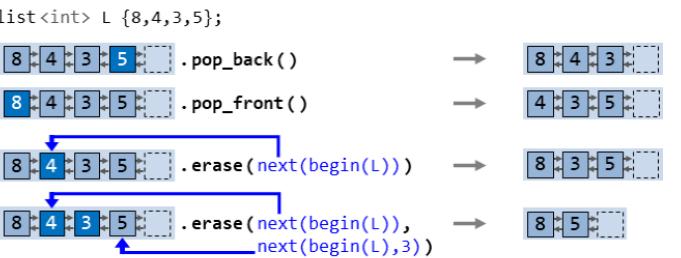
Increment Iterators



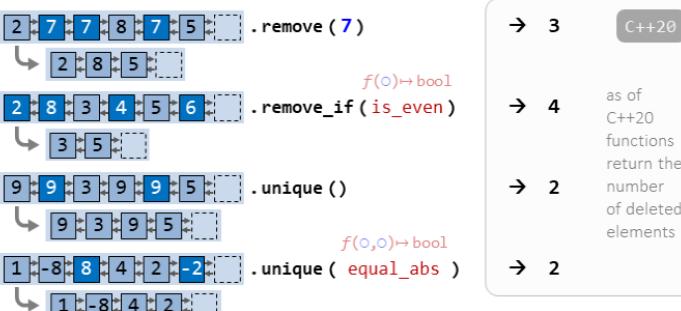
Reorder Elements



Erase Elements Based on Positions

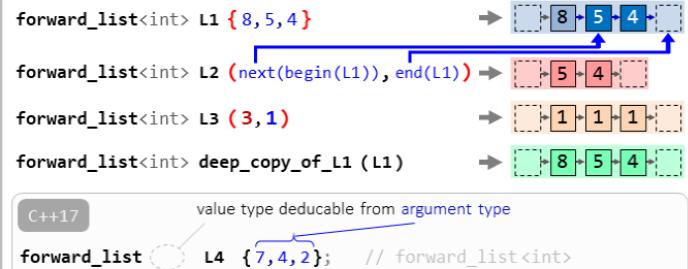


Erase Elements Based on Values

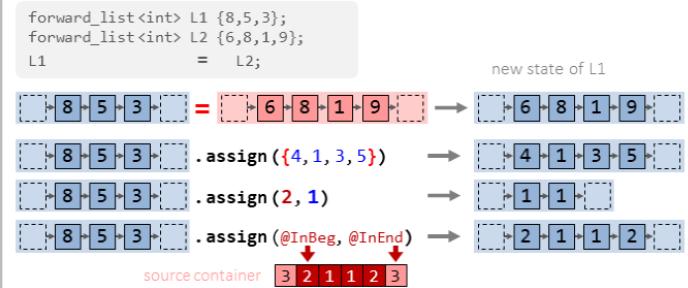


forward list

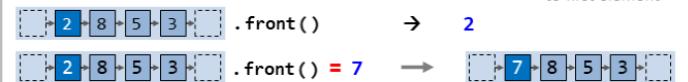
Construct A New List Object



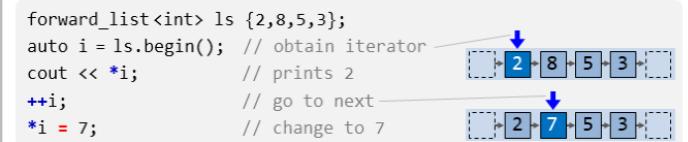
Assign New Content To An Existing List



Access Element Values

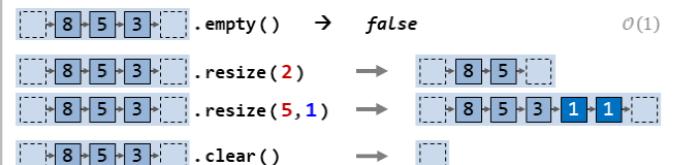


Access Arbitrary Elements Using Iterators

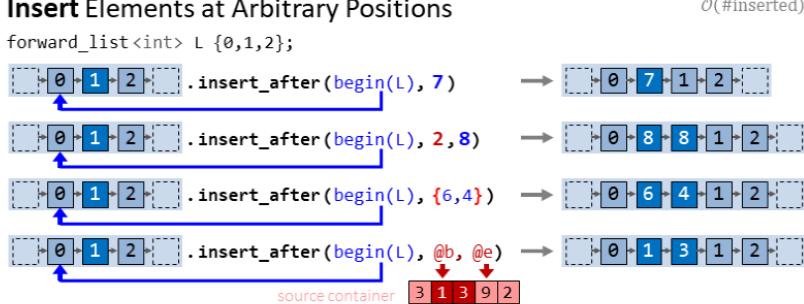


Check Emptiness / Change Size (= Number of Elements)

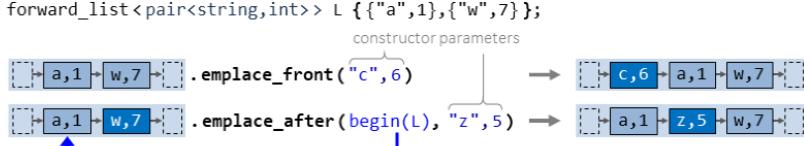
There's no member function available to determine the size!



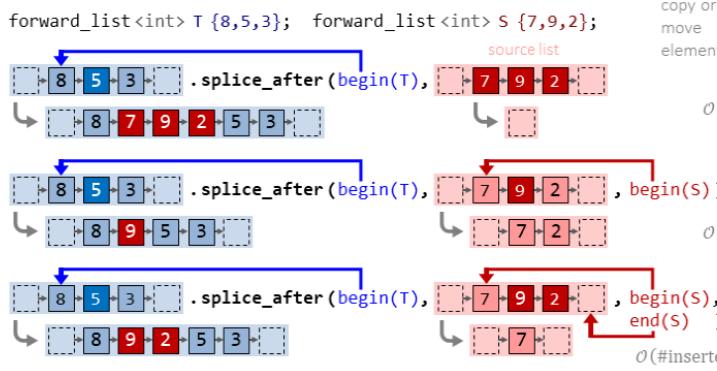
Insert Elements at Arbitrary Positions



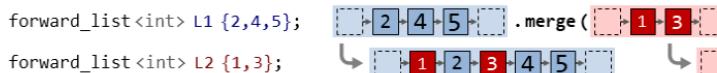
Insert & Construct Elements Without Copy / Move



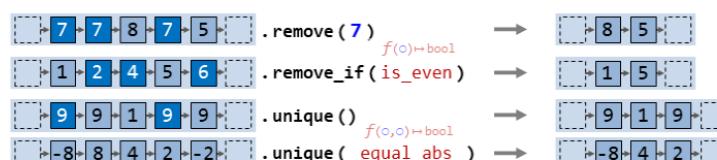
Splice (Elements From One List Into Another One)



Merge Already Sorted Lists



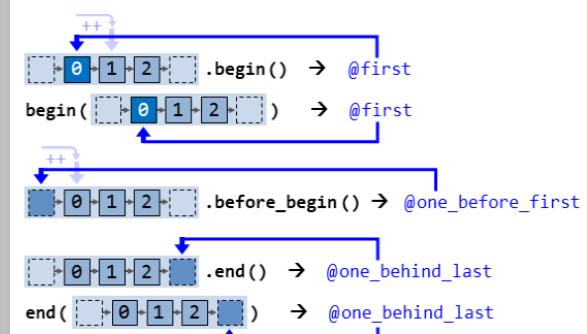
Erase Elements Based on Values



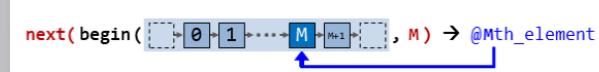
A singly-linked list of nodes each holding one value.

C++11

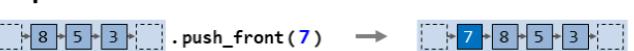
Obtain Iterators



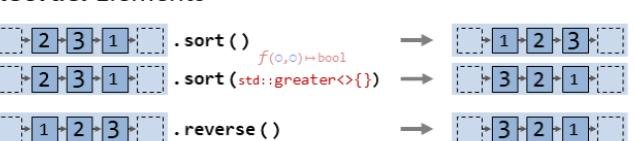
Increment Iterators



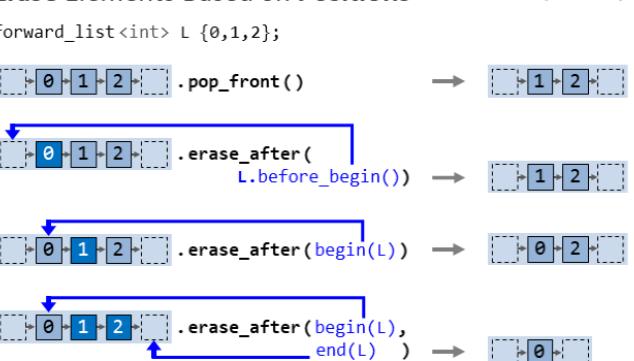
Prepend Elements



Reorder Elements



Erase Elements Based on Positions



`std::stack`

`std::queue`

`std::priority_queue`

Ассоциативные контейнеры

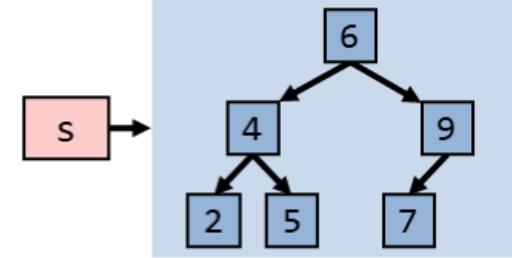
set<Key>

unique, ordered keys

`multiset<K>`

(non-unique) ordered keys

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



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

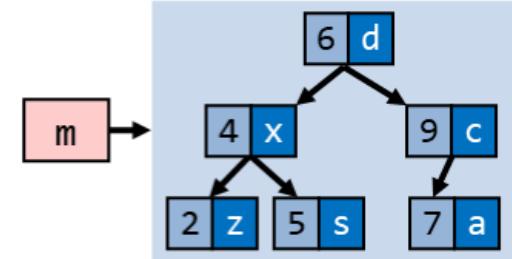
map<Key, Value>

unique key → value-pairs; ordered by keys

`multimap<K, V>`

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

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



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

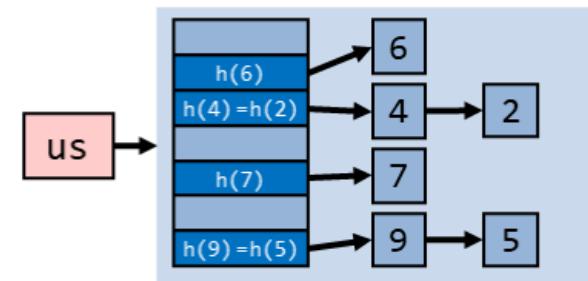
unordered_set<Key>

unique, hashable keys

`unordered_multiset<Key>`

(non-unique) hashable keys

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



hash table for key lookup, linked nodes for key storage

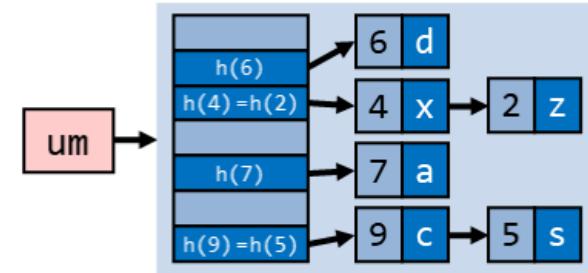
unordered_map<Key, Value>

unique key → value-pairs; hashed by keys

`unordered_multimap<Key, Value>`

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

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



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

std::set<KeyType, Compare>

(unique keys)

#include <set>

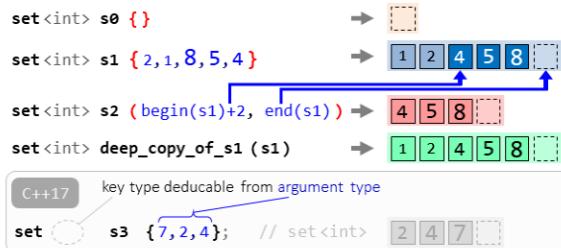
h/cpp hackingcpp.com

default: std::less<KeyType>

std::multiset<KeyType, Compare>

(multiple equivalent keys)

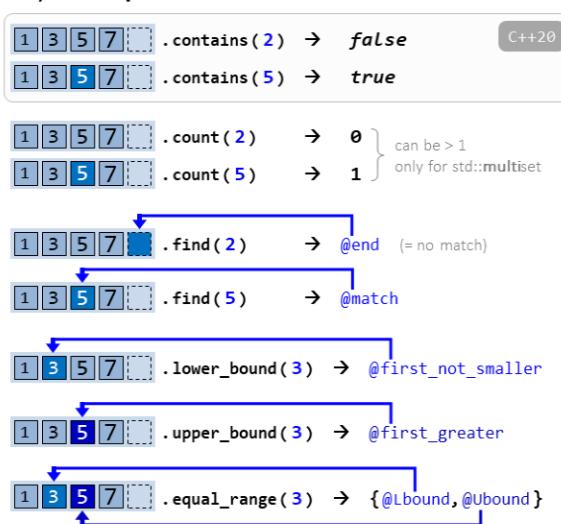
Construct A New Set Object



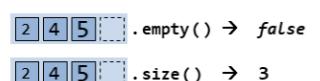
Assign New Content To An Existing Set



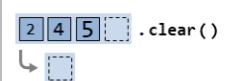
Key Lookup



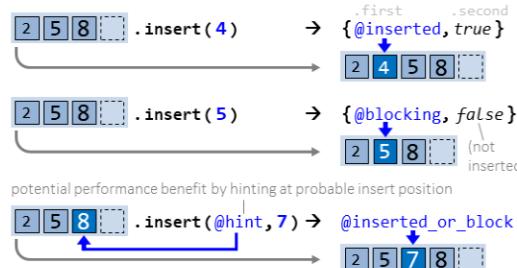
Query Size (= Number of Keys)



Erase All Keys



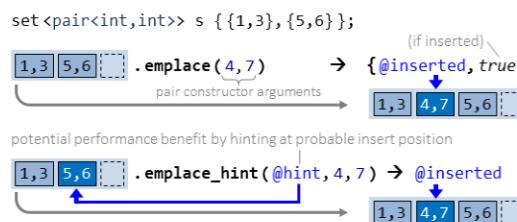
Insert A Single Key



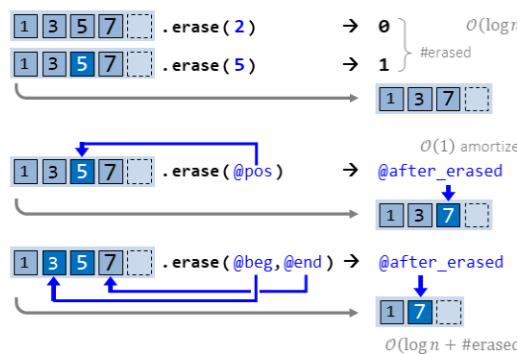
Insert Multiple Keys



Insert & Construct A Key in Place



Erase One Key or A Range of Keys



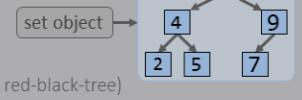
- keys are ordered according to their values

- keys are compared / matched based on equivalence:

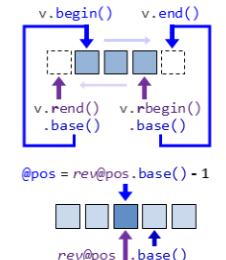
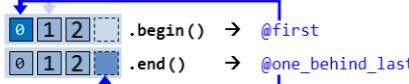
- a and b are equivalent if neither is ordered before the other, e.g., if not ($a < b$) and not ($b < a$)

- default ordering comparator is `std::less`

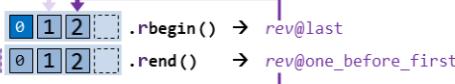
- sets are usually implemented as a balanced binary tree (e.g., as red-black-tree)



Obtain Iterators

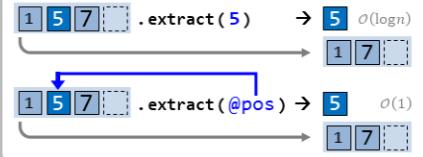


Obtain Reverse Iterators



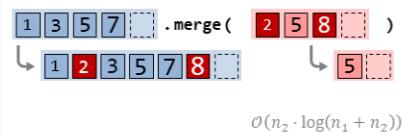
Extract Nodes

C++17
Allows efficient key modification and transfer of keys between different set objects.

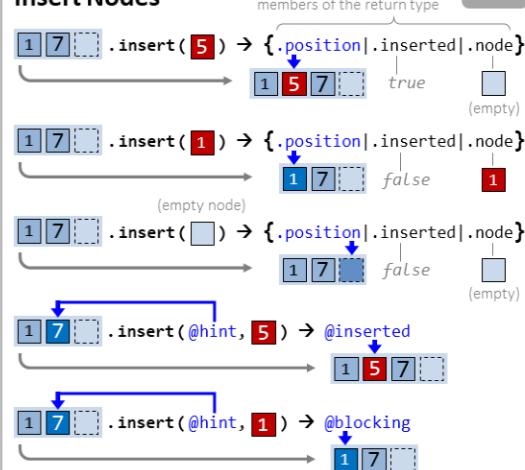


Merge Two Sets

C++17
set<int> s1 {1,3,5,7};
set<int> s2 {2,5,8};



Insert Nodes



Modify Key

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

```
set<int> s {1,5,7};
```

```
auto node = s.extract(5);
```

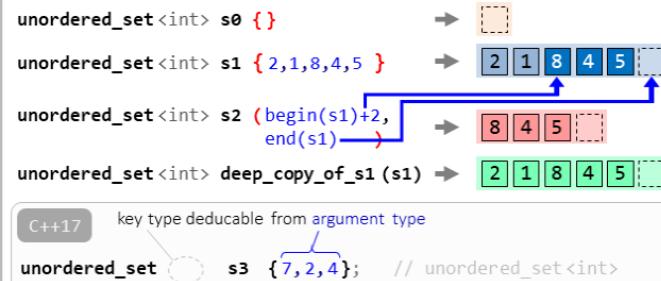
```
if (node) { // if key existed
```

```
node.value() = 8;
```

```
s.insert(std::move(node));
```

unordered set

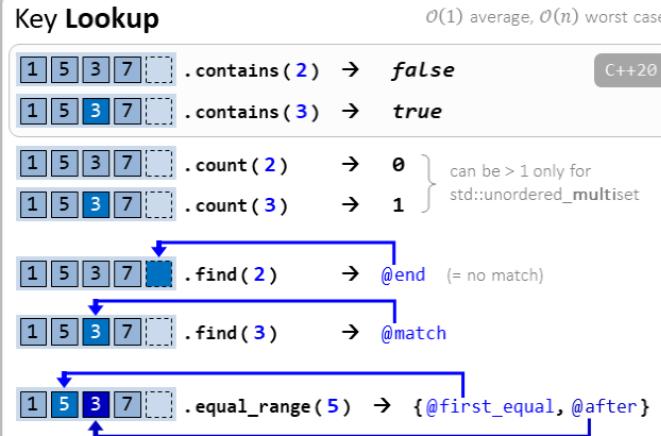
Construct A New Set Object



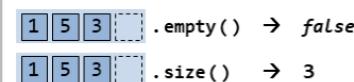
Assign New Content To An Existing Set



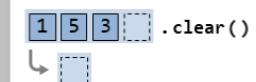
Key Lookup



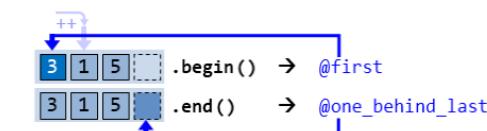
Query Size (=Number of Keys)



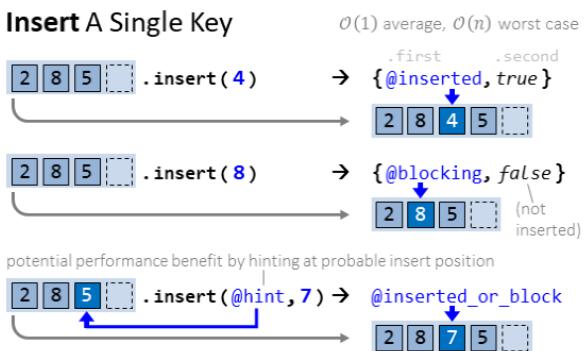
Erase All Keys



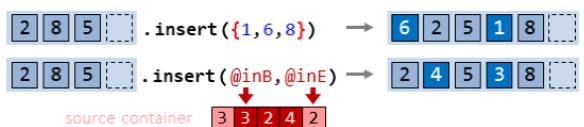
Obtain Iterators (to keys)



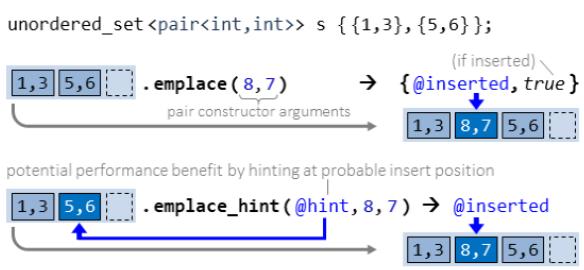
Insert A Single Key



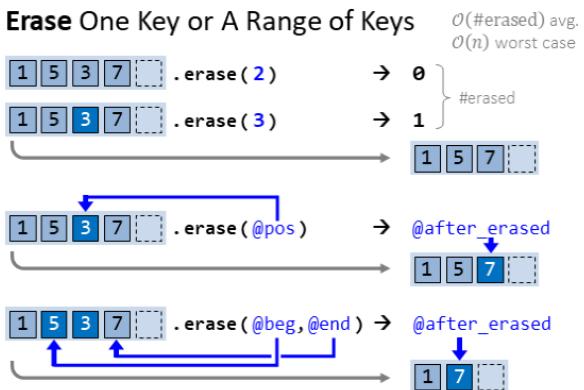
Insert Multiple Keys



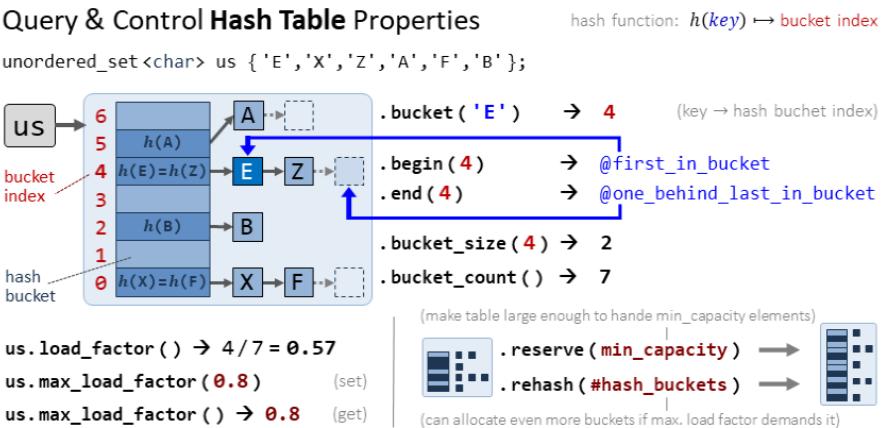
Insert & Construct A Key in Place



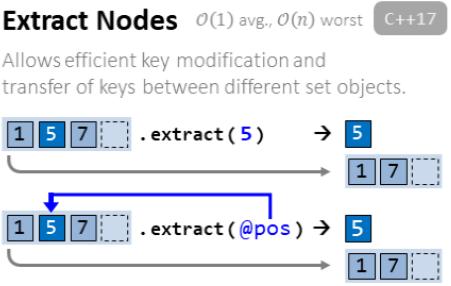
Erase One Key or A Range of Keys



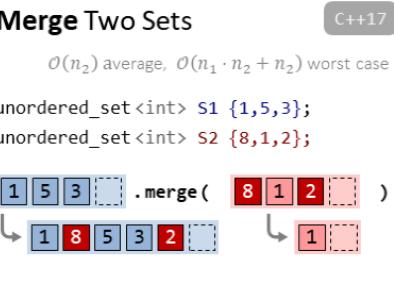
Query & Control Hash Table Properties



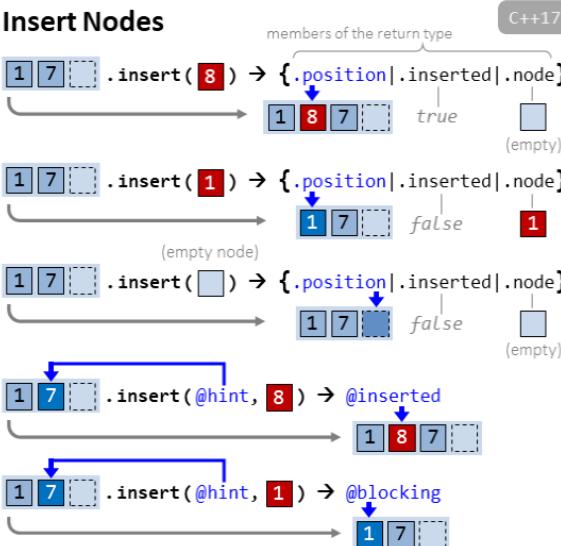
Extract Nodes



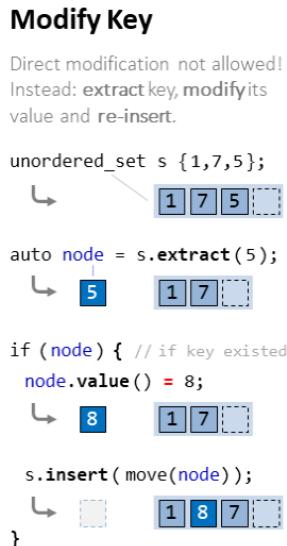
Merge Two Sets



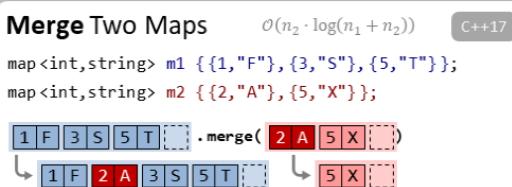
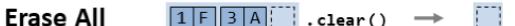
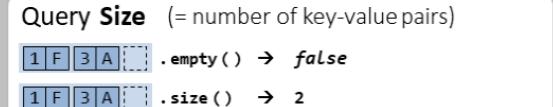
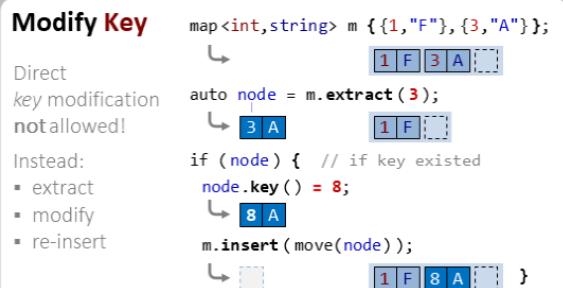
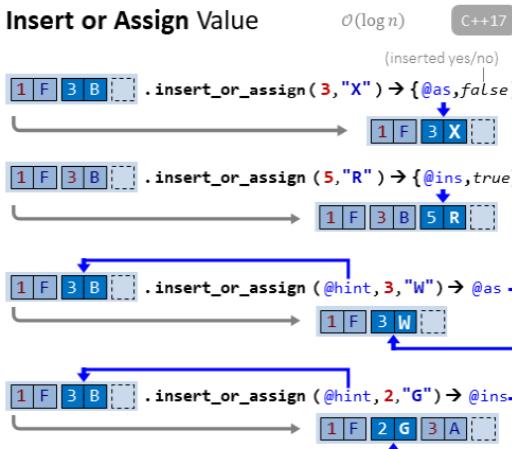
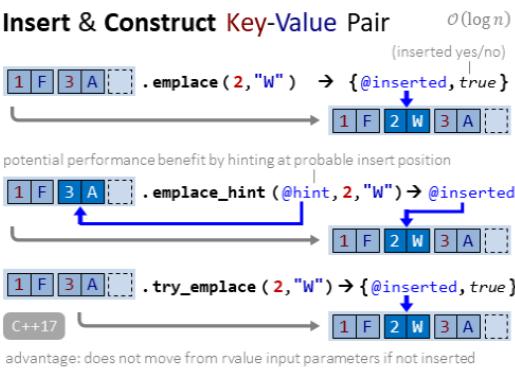
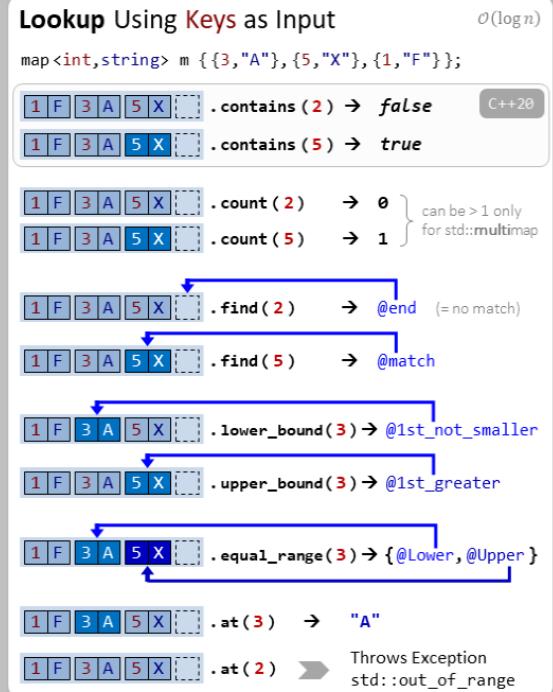
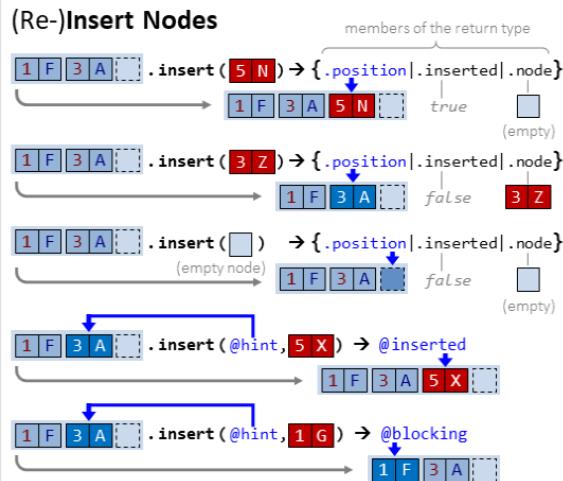
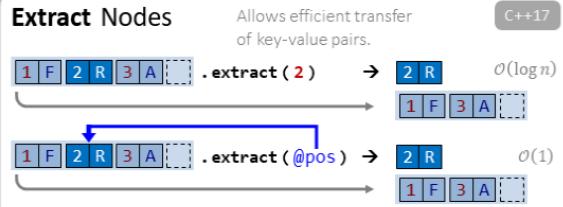
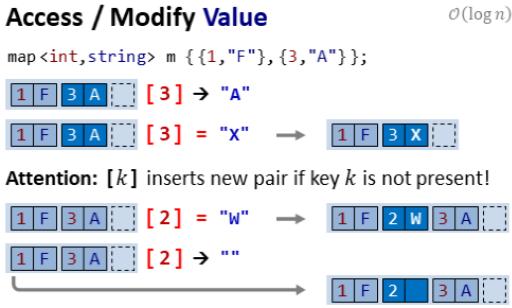
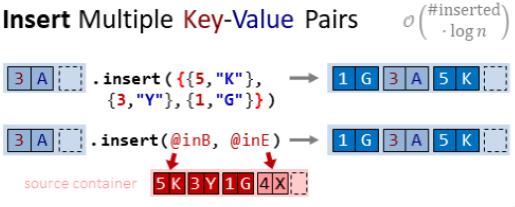
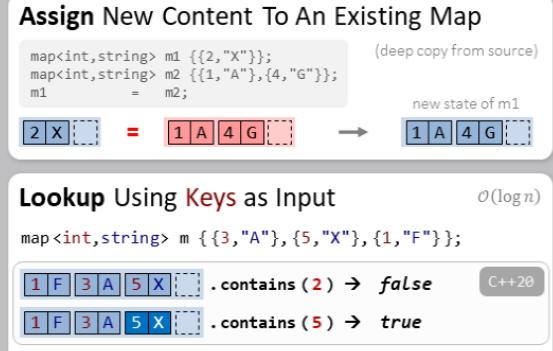
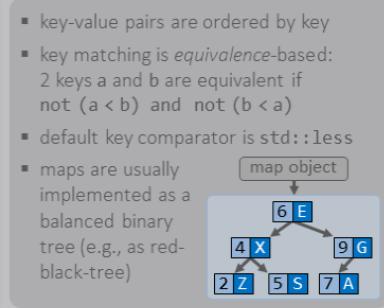
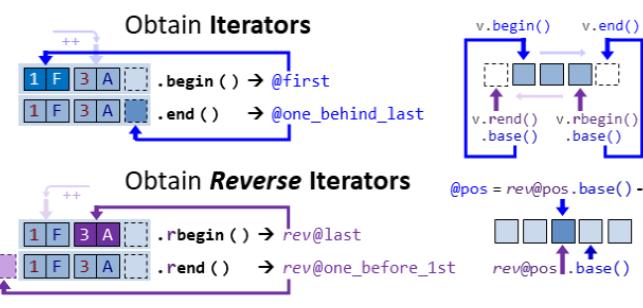
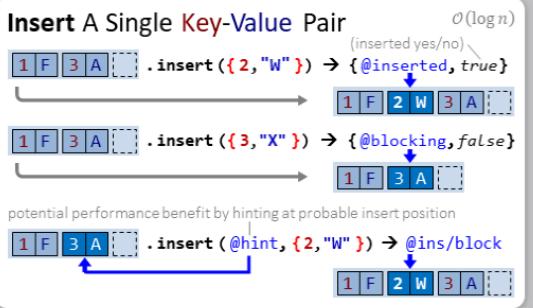
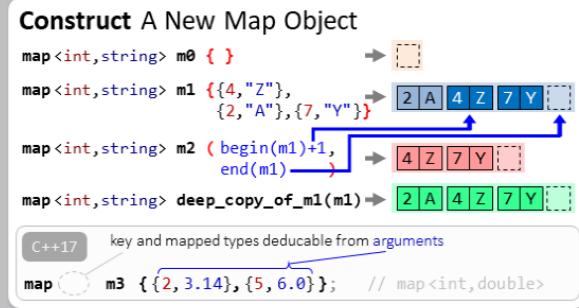
Insert Nodes



Modify Key



std::map



unordered_map

Construct A New Map Object

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

C++17 key and mapped types deducible from arguments
unordered_map m3 {2, 3.14}, {5, 6.0}; // unordered_map<int,double>
```

Assign New Content To An Existing Map

```
unordered_map<string,int> m1 {"X",2}; → [X 2]
unordered_map<string,int> m2 {"A",1}, {"G",4}; → [A 1] G 4
m1 = m2; → [A 1] G 4 [new state of m1]

[X 2] = [A 1] G 4 → [A 1] G 4 [deep copy from source]
```

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};
S 3 F 1 X 5 . contains("W") → false C++20
S 3 F 1 X 5 . contains("X") → true

S 3 F 1 X 5 . count("W") → 0 can be > 1 only for
S 3 F 1 X 5 . count("X") → 1 unorderd_multimap

S 3 F 1 X 5 . find("W") → @end (=no match)
S 3 F 1 X 5 . find("X") → @match

S 3 F 1 X 5 . equal_range("F") → {@1st_equal, @after}
S 3 F 1 X 5 . at("F") → 1
S 3 F 1 X 5 . at("B") → Throws std::out_of_range
```

Query Size

```
F 1 A 3 . empty() → false
F 1 A 3 . size() → 2
```

Erase All

```
F 1 A 3 . clear()
```

Obtain Iterators

```
F 1 A 3 . begin() → @first
F 1 A 3 . end() → @one_behind_last
```

Insert A Single Key-Value Pair

$\mathcal{O}(1)$ avg., $\mathcal{O}(n)$ worst (inserted yes/no)

```
F 1 A 3 . insert({"W",2}) → {@inserted, true}
F 1 A 3 . insert("A",9) → {@blocking, false}

potential performance benefit by hinting at probable insert position
F 1 A 3 . insert(@hint, "W",2) → @instd/blockng
```

Insert Multiple Key-Value Pairs

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

```
A 3 . insert({{"G",4}, {"K",9}, {"A",7}}) → [K 9] A 3 G 4
A 3 . insert(@inBegin, @inEnd) → [K 9] A 3 G 4

source container G 4 K 9 A 7 X 2
```

Construct Key-Value Pair

$\mathcal{O}(1)$ avg., $\mathcal{O}(n)$ worst (inserted yes/no)

```
F 1 A 3 . emplace("W",2) → {@inserted, true}
F 1 A 3 ["A"] → 3
F 1 A 3 ["A"] → 4 → [F 1] A 4

potential performance benefit by hinting at probable insert position
F 1 A 3 . emplace_hint(@hint, "W",2) → @inserted
F 1 A 3 ["W"] → 2 → [F 1] W 2 A 3
F 1 A 3 ["W"] → 0 → [F 1] W 0 A 3

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)

```
F 1 B 3 . insert_or_assign("B",5) → {@as, false}
F 1 B 3 . insert_or_assign("R",6) → {@ins, true}

potential performance benefit by hinting at probable insert position
F 1 B 3 . insert_or_assign(@hint, "B",5) → @as
F 1 B 3 . insert_or_assign(@hint, "G",2) → @ins
```

Query & Control Hash Table Properties

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

```
unordered_map<string,int> um {{"E",6}, {"X",4}, {"Z",1}, {"A",3}, {"F",2}, {"B",2}};
um.bucket("E") → 4 (key → hash bucket index)
um.begin(4) → @first_in_bucket
um.end(4) → @one_behind_last_in_bucket
um.bucket_size(4) → 2
um.bucket_count() → 7

(have table large enough to handle min_capacity elements)
.um.reserve(min_capacity) → [min_capacity]
.um.rehash(#hash_buckets) → [hash_buckets]
(can allocate even more buckets if max. load factor demands it)
```

Access / Modify Value

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

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

Attention: [k] inserts new pair if key k is not present!
F 1 A 3 ["W"] = 2 → [F 1] W 2 A 3
F 1 A 3 ["W"] → 0 → [F 1] W 0 A 3
```

Erase Key-Value-Pair(s)

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

```
F 1 A 3 X 5 . erase("W") → 0
F 1 A 3 X 5 . erase("A") → 1
F 1 A 3 X 5 . erase(@pos) → @after
F 1 A 3 X 5 . erase(@beg, @end) → @after
```

Modify Key

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

Extract Nodes

to efficiently transfer key-value pairs C++17

```
F 1 R 2 A 3 . extract("R") → R 2
F 1 A 3
F 1 R 2 A 3 . extract(@pos) → R 2 O(1) avg., O(n) worst
F 1 A 3
```

(Re-)Insert Nodes

members of the return type

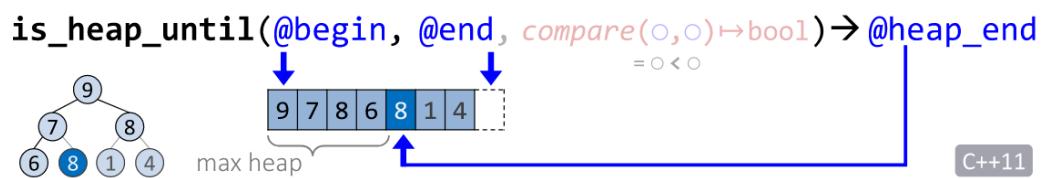
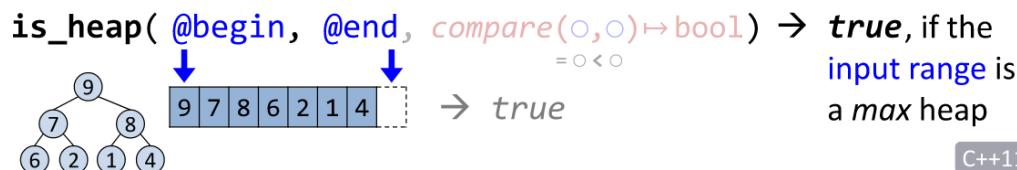
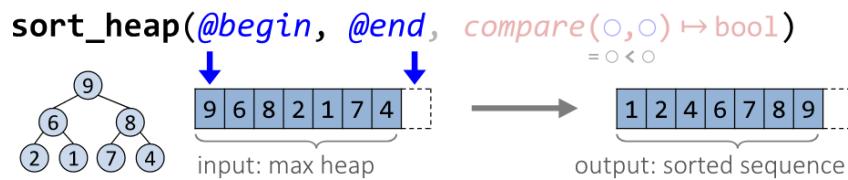
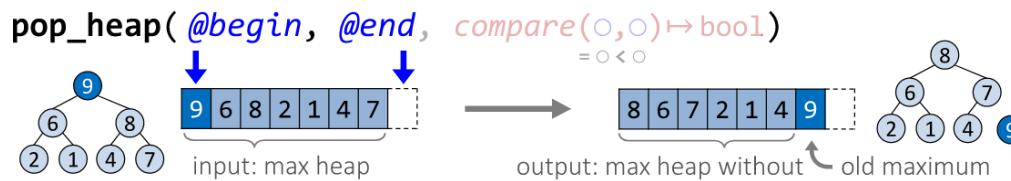
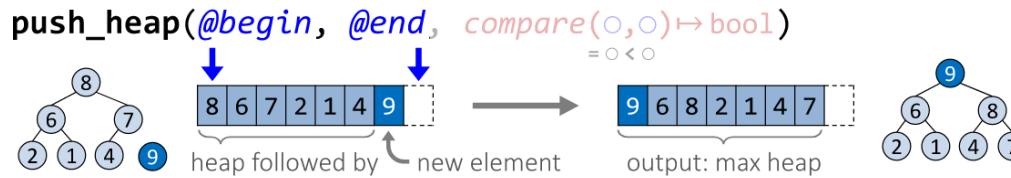
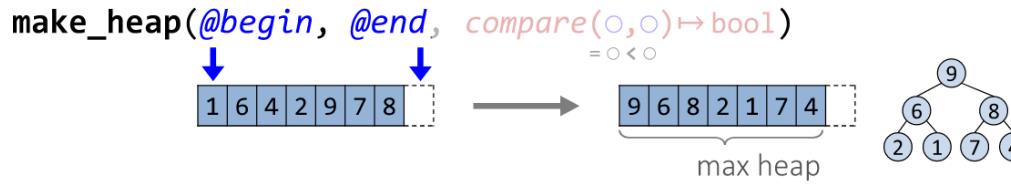
```
F 1 A 3 . insert(N 5) → {@position, inserted, node}
N 5
true (empty)
F 1 A 3 . insert(A 6) → {@position, inserted, node}
A 6
false Z 4
F 1 A 3 . insert(@pos) → {@position, inserted, node}
(empty node)
F 1 A 3 . insert(@hint, X 5) → @inserted
X 5
false (empty)
F 1 A 3 . insert(@hint, F 6) → @blocking
F 1 A 3
```

Merge Two Maps

C++17

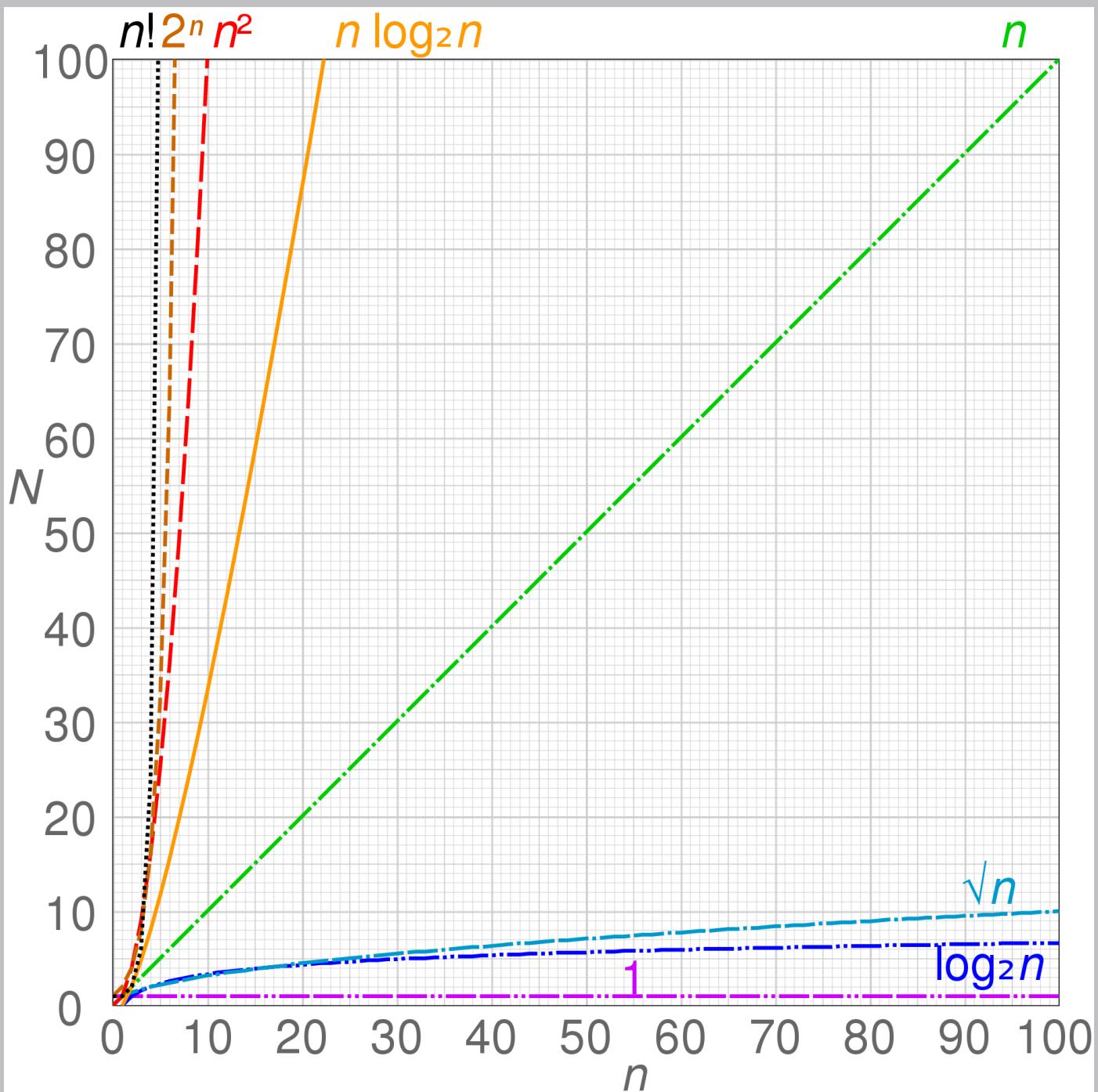
```
unordered_map<string,int> m1 {"F",1}, {"S",3}, {"X",5};
unordered_map<string,int> m2 {"A",2}, {"X",7}; F 1 S 3 X 5
F 1 S 3 X 5 . merge(A 2 X 7) → [F 1] A 2 S 3 X 5 O(n2) average, O(n1 * n2 + n2) worst case
```

Heap operations



Сложность алгоритмов

Оценка сложности Big-O



Оценка сложности для контейнеров STL

Container	Insert Head	Insert Tail	Insert	Remove Head	Remove Tail	Remove	Index Search	Find
vector	n/a	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(log n)
list	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	n/a	O(n)
deque	O(1)	O(1)	n/a	O(1)	O(1)	O(n)	n/a	n/a
queue	n/a	O(1)	n/a	O(1)	n/a	n/a	O(1)	O(log n)
stack	O(1)	n/a	n/a	O(1)	n/a	n/a	n/a	n/a
map	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)	O(log n)
multimap	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)*	O(log n)
set	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)	O(log n)
multiset	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)*	O(log n)

Лямбда выражения

Лямбда выражения

C++11

[CAPTURES] {BODY}
[CAPTURES] -> RETURN-TYPE {BODY}

C++23

[CAPTURES] SPECIFIERS {BODY}
[CAPTURES] SPECIFIERS -> RETURN-TYPE {BODY}

C++11

[CAPTURES](PARAMETERS) {BODY}
[CAPTURES](PARAMETERS) -> RETURN-TYPE {BODY}
[CAPTURES](PARAMETERS) SPECIFIERS {BODY}
[CAPTURES](PARAMETERS) SPECIFIERS -> RETURN-TYPE {BODY}

C++20

[CAPTURES]<TYPE-PARAMETERS>(PARAMETERS) {BODY}
[CAPTURES]<TYPE-PARAMETERS>(PARAMETERS) -> RETURN-TYPE {BODY}
[CAPTURES]<TYPE-PARAMETERS>(PARAMETERS) SPECIFIERS {BODY}
[CAPTURES]<TYPE-PARAMETERS>(PARAMETERS) SPECIFIERS -> RETURN-TYPE {BODY}

SPECIFIERS

C++11

mutable

noexcept

[[attribute]]

C++17

constexpr

C++20

consteval

requires...

```
auto f1 = [] { return 47; };
auto f2 = [] (int x, int y) { return 0.5*(x+y); };
auto f3 = [] (int x) -> float { return x*x; };

auto x = f1();           // ⇔ int x = 47;      C++11
auto y = f2(2,3);       // ⇔ double y = 2.5;
auto z = f3(2);         // ⇔ float z = 4.0f;
```

Capturing

of variables from the surrounding scope

by reference

```
int x = 1;
auto f = [&] (int y){ x+=2; return x*y; };
cout << f(2); // 6
cout << x;    // 3
```

by value

```
int x = 1;           lambda-local variable x=1
auto f = [=] (int y){ return x*y; };
cout << f(2); // 2
cout << x;    // 1
```

[=]	capture all by value
[&]	capture all by reference
[=,&x]	x by reference, all others by value
[&,x]	x by value, all others by reference
[x,&y]	only x by value and y by reference
[&x,y]	only x by reference and y by value
[&x,y,&z]	only x by ref, y by value and z by ref

init captures

define lambda-local variables

```
auto f = [x=2] (int y){ return x*y; };
std::vector<char> v (1000,'a');
auto g = [w=std::move(v)]() { /* use w */ };
```

Immediately Invoked Function Expressions (IIFE)

```
int z = [] (int x){ return x*x; }(); // create lambda and call it ⇔ int z = 4;
```

Mutable Lambdas

```
int y = 1;
auto f = [=] () {
    y += 2;
    return y;
};
```



COMPILER ERROR:
local variable
'y' is const!

```
int y = 1;
auto f = [=] () mutable {
    y += 2;
    return y;
};
cout << f(); // 3
cout << f(); // 5
cout << y;    // 1
```

```
auto f = [i=0] () mutable {
    ++i; return i;
};
cout << f(); // 1
cout << f(); // 2
```

Generic Lambdas

```
auto print = [](auto const& x){ std::cout << x; };
print(5); ✓
print(std::string("it works!")); ✓
```

```
[](auto value, auto const& cref, auto& ref){ ... }
[](auto... args){ return g(args...); }
```

Perfect Forwarding

preserving constness, l/r-valueness

```
[](auto&& x){ g( std::forward<decltype(x)>(x) ); }
[](auto&&... args){
    g( std::forward<decltype(args)>(args)... ); }
```

Constrained auto Parameters

```
#include <concepts>
[](<std::copyable> auto x){ ... }
```

Explicit Template Parameters

```
[<typename T> (T x, T y){ ... }
```

C++14

C++20

C++20

Итераторы

Итератор — это структура данных, предназначенная, для того чтобы перебирать элементы контейнера (последовательности), при этом не задумываясь, с каким именно контейнером происходит работа.

Получить итераторы у контейнера можно через:

- методы `begin()` и `end(): array.begin();`
- одноимённые функции: `begin(array);`

Чтобы ходить по контейнеру в обратном направлении используются реверс-итераторы: `rbegin()` и `rend()`.

Итераторы на примере `vector`

```
vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9, 9 };  
auto i = begin(v);  
int x = *i; // x: 1  
++i; // advance by 1  
auto j = begin(v) + 3;  
int y = *j; // y: 4  
auto e = end(v);  
*j = 47; // change element value: 4 → 47
```

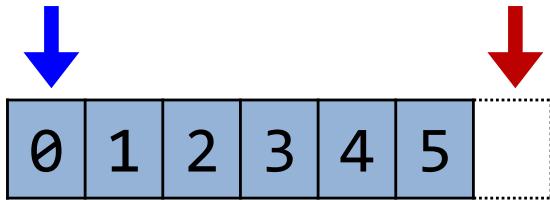
DO NOT ACCESS 'END' WITH '*'!
(does not refer to valid memory)
ONLY USE AS POSITION SPECIFIER!

Итераторы

```
std::vector<int> v {0, 1, 2, 3, 4, 5};
```

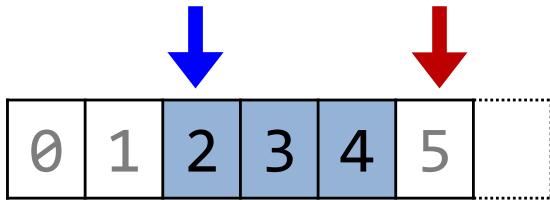
`begin(v), end(v)`

6 elements



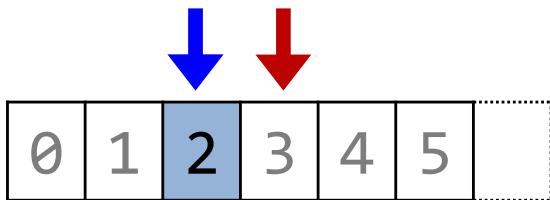
`begin(v) + 2, begin(v) + 5`

3 elements



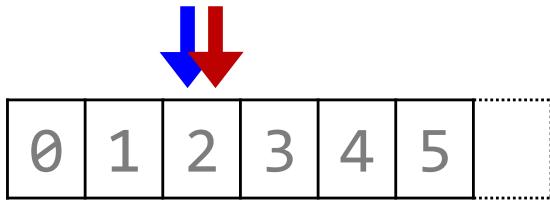
`begin(v) + 2, begin(v) + 3`

1 element

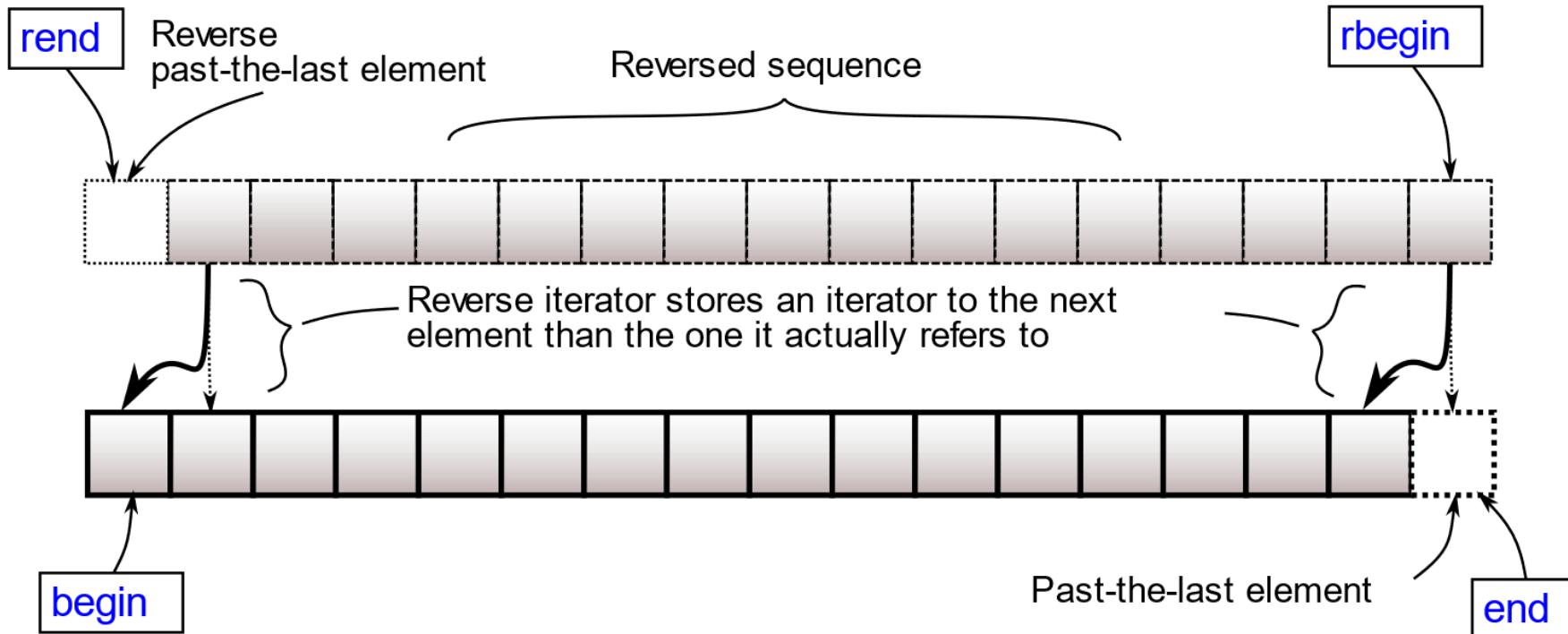


`begin(v) + 2, begin(v) + 2`

empty range



Реверс итераторы



Стандартные алгоритмы

Шпаргалки по стандартным алгоритмам: <https://hackingcpp.com/cpp/std/algorithms.html>

105 STL Algorithms in Less Than an Hour: <https://youtu.be/bXkWuUe9V2I>