Module 13 of 20

STL containers

The four container families and their complexity table, std::vector's size–capacity model and the erase–remove idiom, deque, list and the adaptors, std::map and std::set with lower_bound, the unordered containers with custom hashes, and iterators with the invalidation table.

A C++ program keeps its data in the STL containers, and the difference between a program that is fast and one that is merely correct is usually which container it chose and how it used it. The containers were designed as one system: the same member names mean the same things everywhere, each container publishes what its operations cost, and each one promises which operations leave existing iterators pointing at garbage. Those three facts — the common interface, the complexity table and the invalidation rules — are what this module teaches, and they are also what interviewers ask.

The six lessons move from the map to the detail. The container zoo names the four families, the complexity table and the one-sentence reason to reach past `std::vector`. The vector lesson covers size versus capacity, geometric growth and amortised `push_back`, `reserve`, `emplace_back`, the O(n) cost of insertion in the middle, the erase–remove idiom and its C++20 successor `std::erase_if`, and the odd `std::vector<bool>`. Deque, list and the adaptors follow — the stable-ended deque, the node-based list with its `splice`, and `std::stack`, `std::queue` and `std::priority_queue` with the comparator convention that everyone inverts once. The ordered `std::map` and `std::set` bring `operator[]` that inserts, `insert` that reports, and `lower_bound` for "first key not less than". The unordered containers bring hashing, load factors, custom hashes for a struct, and the rule that an unordered container is sorted before it is printed. Iterators close the module: categories, `std::next` / `std::prev` / `std::distance`, the invalidation table per container, and the one correct shape of an erase-while-iterating loop.

Every exercise is a whole program judged on a real C++20 compiler and built on the container its lesson teaches: a template that reports three containers through their common interface, a bracket matcher on `std::stack`, a command-driven vector editor and an `std::erase_if` filter with a stateful predicate, a sliding-window maximum on a monotonic `std::deque`, a task scheduler on `std::priority_queue` with a tie-breaking comparator, word frequencies in `std::map` order and again through `std::unordered_map` sorted on output, "next departure" with `lower_bound` on a `std::set`, a custom-hash `std::unordered_set` of points, an account ledger pruned with `it =`

`m.erase(it)`, and a playlist cursor on a `std::list`. The checkpoint adds a breadth-first search with `std::queue`, an LRU cache from a `std::list` and an `std::unordered_map` of iterators, and a leaderboard on a `std::set` with a custom ordering.

CONTENTS

1	The container zoo — choosing a container	12 min
2	std::vector in depth — size, capacity and the erase–remove idiom	14 min
3	deque, list and the adaptors — stack, queue and priority_queue	14 min
4	std::map and std::set — the ordered containers	14 min
5	The unordered containers — hashing, buckets and custom keys	13 min
6	Iterators and invalidation — the contract behind every loop	14 min
✓	Checkpoint — STL containers	25 min

Exercises in this module compile on Clang 18 · C++20. Reference solutions and the answer key follow each lesson.

LESSON 1 · 12 MIN

The container zoo — choosing a container

The STL ships about a dozen containers and most programs use two of them, `std::vector` and a map. The rest earn their place in specific situations, and the skill this lesson settles is naming those situations: which family a container belongs to, what its operations cost, which member functions every container shares, and the one-sentence reason for reaching past `std::vector`. Every container is a class template (Module 12) that owns its elements by value, so the choice is about where the elements sit in memory and what that layout makes cheap.

Four families

Sequence containers keep elements in the order you put them. `std::vector` is one contiguous block that grows at the back; `std::deque` is a chain of fixed-size blocks that grows at both ends; `std::list` is a doubly linked chain of nodes; `std::forward_list` is singly linked; `std::array` (Module 6) never grows.

Associative containers keep elements sorted by key inside a balanced binary tree: `std::set` holds keys, `std::map` holds key–value pairs, and `std::multiset` / `std::multimap` allow duplicate keys.

Unordered containers keep elements in a hash table: `std::unordered_set`, `std::unordered_map` and their `multi` versions. Lookup is average O(1); the price is that iteration visits elements in no useful order.

Container adaptors wrap a sequence container and expose a narrower interface: `std::stack` (last in, first out), `std::queue` (first in, first out) and `std::priority_queue` (largest out first). They have no iterators.

Container	Header	Layout	Order
`std::vector<T>`	`<vector>`	one contiguous block	insertion
`std::deque<T>`	`<deque>`	blocks, both ends grow	insertion
`std::list<T>`	`<list>`	doubly linked nodes	insertion
`std::set<K>` / `std::map<K, V>`	`<set>` / `<map>`	red–black tree	sorted by key
`std::unordered_set<K>` / `std::unordered_map<K, V>`	`<unordered_set>` / `<unordered_map>`	hash buckets	unspecified
`std::stack`, `std::queue`, `std::priority_queue`	`<stack>`, `<queue>`	wrap a `deque` or `vector`	push/pop only

The complexity table

An interview question about containers is almost always a question about this table. Learn it as a picture of the layouts: contiguous memory makes indexing free and middle insertion expensive; linked nodes make the reverse true; a tree makes everything logarithmic; a hash table makes lookup constant on average.

Operation	`vector`	`deque`	`list`	`set` / `map`	`unordered_*`
index `c[i]`	O(1)	O(1)	—	—	—
push/pop at back	amortised O(1)	O(1)	O(1)	—	—
push/pop at front	O(n)	O(1)	O(1)	—	—
insert/erase in the middle	O(n)	O(n)	O(1) given an iterator	O(log n)	average O(1)
find by value or key	O(n)	O(n)	O(n)	O(log n)	average O(1), worst O(n)

"Given an iterator" carries the weight in the `std::list` column: the O(1) erase needs an iterator that already points at the element, and *finding* that iterator is O(n). Sorted data changes one cell — a sorted `std::vector` finds by key in O(log n) with `std::lower_bound` (Module 14, lesson 2) and is often the fastest associative structure of all for data that is built once and queried many times.

The common interface

The containers were designed together, so the same names mean the same things everywhere. Every container has `size()`, `empty()`, `clear()`, `begin()` / `end()` (and the `const` /reverse variants), `insert`, `erase`, `swap`, a copy and a move constructor, assignment and `==`. Any function written against that interface works for any container, which is exactly what the algorithms in `<algorithm>` rely on:

```cpp
#include <iostream>
#include <list>
#include <set>
#include <vector>

template <typename Container>
void report(const char* name, const Container& c) {
    std::cout << name << " size=" << c.size() << (c.empty() ? " (empty)" : "")
    for (const auto& x : c) std::cout << ' ' << x;
    std::cout << '\n';
}

int main() {
    std::vector<int> v{3, 1, 2, 3};
    std::list<int> l{3, 1, 2, 3};
    std::set<int> s{3, 1, 2, 3};
    report("vector", v);   // vector size=4: 3 1 2 3
    report("list", l);     // list size=4: 3 1 2 3
    report("set", s);      // set size=3: 1 2 3
    v.clear();
    report("vector", v);   // vector size=0 (empty):
    return 0;
}
```

The differences are where the layouts differ. `operator[]` exists on `std::vector`, `std::deque`, `std::array`, `std::map` and `std::unordered_map` — and on the two maps it *inserts* when the key is absent (lesson 4), the most common container surprise in the language. `push_back` exists on `vector`, `deque` and `list`; `push_front` only on `deque`, `list` and `forward_list`. A member `find` exists only on the associative and unordered containers, because for a sequence nothing beats the linear `std::find`. The adaptors expose `push`, `pop` and `top` (or `front` / `back` for a queue), and nothing else.

Value semantics

A container owns copies of what you put in it. `v.push_back(s)` copies the string `s`; `v.push_back(std::move(s))` moves it (Module 9); the container destroys its elements when it is destroyed. Copying a container copies every element; assigning one copies every element; moving one steals the buffer in O(1):

```
std::vector<std::string> a{"one", "two"};
std::vector<std::string> b = a;           // two strings copied
std::vector<std::string> c = std::move(a);   // a is now empty; c owns the buf
a.push_back("x");                         // a is valid and reusable
```

This is why `std::vector<std::unique_ptr<Shape>>` (Module 10) is the idiom for a polymorphic collection — the container owns the pointers, the pointers own the objects — and why a function that only reads a container takes it as `const std::vector<int>&` (Module 4, lesson 2).

How to choose

Start with `std::vector`. Contiguous memory means the CPU's cache works for you, iteration is the fastest of any container, indexing is free, and `push_back` is amortised constant. Measured against `std::list`, a vector wins at inserting into the middle of a small or medium collection because the linear search for the spot dominates and the shuffle of contiguous elements is cheap (Module 19, lesson 3). Reach for something else only with a named reason:

- pushes and pops at the *front* as well as the back — `std::deque`;
- elements must stay put in memory while the collection grows, or you need O(1) splicing — `std::list`;
- keys in sorted order, "smallest key not less than k", ranges — `std::set` / `std::map`;
- fastest lookup by key and order does not matter — `std::unordered_map` / `unordered_set`;
- a push/pop discipline and nothing else — `std::stack`, `std::queue`, `std::priority_queue`;
- a size fixed at compile time — `std::array`.

A first adaptor shows how narrow an interface can be and still be exactly right:

```cpp
#include <stack>
#include <string>

bool balanced(const std::string& text) {
    std::stack<char> open;                    // std::deque<char> underneath
    for (char c : text) {
        if (c == '(' || c == '[') {
            open.push(c);
        } else if (c == ')' || c == ']') {
            const char want = (c == ')') ? '(' : '[';
            if (open.empty() || open.top() != want) return false;
            open.pop();                       // returns void; read top() firs
        }
    }
    return open.empty();
}
```

`push`, `top`, `pop`, `empty`, `size` — nothing else, and no way to look below the top. The type documents the discipline.

Pitfalls

Mistake	What happens
"Insert in the middle is O(1), so `std::list`"	The O(n) search for the position dominates; the vector is usually faster and always smaller
`st.top()` or `st.pop()` on an empty adaptor	Undefined behaviour — the adaptors do not throw; check `empty()` first
Expecting `std::set` to keep insertion order	It keeps *key* order; use a vector, or a map from key to position
Printing an `std::unordered_map` and expecting a stable order	The order is unspecified and changes with library version and size; sort first
`std::vector<Base>` for a class hierarchy	Slices every derived object (Module 10); hold `std::unique_ptr<Base>`

Key takeaways

Four families: sequence (`vector` , `deque` , `list`), associative (`set` , `map`), unordered (`unordered_set` , `unordered_map`) and adaptors (`stack` , `queue` , `priority_queue`).

The complexity table follows from the layout: contiguous → O(1) index and O(n) middle insert; nodes → the reverse; tree → O(log n); hash → average O(1).

`size` , `empty` , `clear` , `begin` / `end` , `insert` , `erase` , `swap` and copy/move mean the same thing on every container; `operator[]` , `push_front` and a member `find` exist only where the layout makes them cheap.

Containers own their elements by value: copying copies everything, moving is O(1).

`std::vector` by default; every other choice needs a reason you can say in one sentence.

EXERCISE 1

Same numbers, three containers

Read an integer `n` , then `n` integers. Append each one to a `std::vector<int>` with `push_back` , push each one onto the *front* of a `std::deque<int>` with `push_front` , and insert each one into a `std::set<int>` . Then print the three containers through **one function template** that uses only the common interface (`size()` and a range-based `for`), in the format `<name>[<size>]: e1 e2 …` . Finally print `empty` when there were no numbers, otherwise `first=<vector.front()> last=<vector.back()> min=<smallest> max=<largest>` , taking the smallest and largest from the set.

Input: `n` (0 or more), then `n` integers. **Output:** three container lines, then one summary line.

```
5
3 1 2 3 9
```

prints

```
vector[5]: 3 1 2 3 9
deque[5]: 9 3 2 1 3
set[4]: 1 2 3 9
first=3 last=9 min=1 max=9
```

INPUT 1

```
5
3 1 2 3 9
```

OUTPUT 1

```
vector[5]: 3 1 2 3 9
deque[5]: 9 3 2 1 3
set[4]: 1 2 3 9
first=3 last=9 min=1 max=9
```

INPUT 2

```
3
7 7 7
```

OUTPUT 2

```
vector[3]: 7 7 7
deque[3]: 7 7 7
set[1]: 7
first=7 last=7 min=7 max=7
```

REFERENCE SOLUTION

```cpp
#include <deque>
#include <iostream>
#include <set>
#include <vector>

template <typename Container>
void print(const char* name, const Container& c) {
    std::cout << name << '[' << c.size() << "]:";
    for (const auto& x : c) std::cout << ' ' << x;
    std::cout << '\n';
}

int main() {
    int n;
    std::cin >> n;
    std::vector<int> v;
    std::deque<int> d;
    std::set<int> s;
    for (int i = 0; i < n; ++i) {
        int x;
        std::cin >> x;
        v.push_back(x);
        d.push_front(x);
        s.insert(x);
    }
    print("vector", v);
    print("deque", d);
    print("set", s);
    if (v.empty()) {
        std::cout << "empty\n";
    } else {
        std::cout << "first=" << v.front() << " last=" << v.back()
                  << " min=" << *s.begin() << " max=" << *s.rbegin() << '\n';
    }
    return 0;
}
```

EXERCISE 2

Bracket matcher on std::stack

Read an integer `n`, then `n` lines of text (a line may be empty or contain spaces and other characters). For each line decide whether its brackets — `()`, `[]` and `{}` — are balanced, using a `std::stack<char>`: push every opener; on a closer, if the stack is empty or its top is not the matching opener, print `mismatch at <i>` (the 0-based index of that closer) and stop scanning the line. If the whole line scans without a mismatch, print `balanced` when the stack is empty and `unclosed <k>` when `k` openers remain on it.

Read `n` with `std::cin >>` and call `std::cin.ignore()` before the first `std::getline`.

Input: `n`, then `n` lines. **Output:** `n` lines.

```
4
([]{})
([)]
((
)
```

prints

```
balanced
mismatch at 2
unclosed 2
mismatch at 0
```

INPUT 1

```
4
([]{})
([)]
((
)
```

OUTPUT 1

```
balanced
mismatch at 2
unclosed 2
mismatch at 0
```

INPUT 2

```
2
int main() { return v[0]; }
{[}
```

OUTPUT 2

```
balanced
mismatch at 2
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <stack>
#include <string>

int main() {
    int n;
    std::cin >> n;
    std::cin.ignore();
    for (int k = 0; k < n; ++k) {
        std::string line;
        std::getline(std::cin, line);
        std::stack<char> open;
        bool ok = true;
        for (std::size_t i = 0; i < line.size() && ok; ++i) {
            const char c = line[i];
            if (c == '(' || c == '[' || c == '{') {
                open.push(c);
            } else if (c == ')' || c == ']' || c == '}') {
                const char want = c == ')' ? '(' : c == ']' ? '[' : '{';
                if (open.empty() || open.top() != want) {
                    std::cout << "mismatch at " << i << '\n';
                    ok = false;
                } else {
                    open.pop();
                }
            }
        }
        if (!ok) continue;
        if (open.empty()) std::cout << "balanced\n";
        else std::cout << "unclosed " << open.size() << '\n';
    }
    return 0;
}
```

QUIZ

1. Which container should be the default choice for a sequence of elements, and why?

A. `std::list`, because insertion anywhere is O(1)

B. `std::vector`, because contiguous memory makes iteration and indexing cheapest and `push_back` is amortised O(1)

C. `std::deque`, because it is O(1) at both ends

D. `std::set`, because it keeps elements sorted

2. Which of these is a container *adaptor* rather than a container?

A. `std::deque`

B. `std::multiset`

C. `std::priority_queue`

D. `std::array`

3. What does this print?

```
std::set<int> s{4, 1, 4, 2, 1};
std::cout << s.size();
```

A. 5

B. 3

C. 2

D. compile error

4. Which operation is O(1) on a `std::list` but O(n) on a `std::vector`?

A. Finding an element by value

B. Reading the element at index `i`

C. Erasing an element you already hold an iterator to

D. Iterating over every element

5. What happens here?

```
std::stack<int> st;
st.pop();
```

A. Throws `std::out_of_range`

B. Does nothing

C. Undefined behaviour

D. Compile error

6. `std::vector<std::string> b = a;` — what does this line do?

A. Copies every string: `b` owns its own elements

B. Makes `b` share `a`'s elements until one of them is modified

C. Copies only the pointers to the strings

D. Does not compile: containers cannot be copied

ANSWER KEY · THE CONTAINER ZOO — CHOOSING A CONTAINER

1. B Contiguous storage is what the cache rewards, and it makes indexing free. A list's O(1) insert needs an iterator you must first find in O(n); a deque and a set each pay for a feature most sequences do not need.

2. C `std::stack`, `std::queue` and `std::priority_queue` wrap a sequence container and expose only push/pop/top — no iterators. The other three are containers in their own right.

3. B A `std::set` holds unique keys, so the duplicate 4 and 1 are dropped: {1, 2, 4}. A `std::multiset` would keep all five.

4. C A list relinks two neighbours; a vector shifts every element after the gap. Finding by value is O(n) on both, indexing is O(1) only on the vector, and iterating is O(n) on both.

5. C The adaptors do no checking: `top()` and `pop()` on an empty stack are undefined behaviour. Only `at()` on the indexable containers throws; test `empty()` first.

6. A Containers have value semantics: copying one copies every element. Sharing until modification is copy-on-write, which C++ containers do not do; only `std::move(a)` transfers the buffer without copying.

LESSON 2 · 14 MIN

std::vector in depth — size, capacity and the erase–remove idiom

`std::vector` is the container you will use in nine programs out of ten, and almost everything that goes wrong with it comes from one fact: its elements live in a single contiguous block that is occasionally thrown away and replaced by a bigger one. That fact explains why `push_back` is fast on average but not always, why a reference into a vector can silently die, why `reserve` exists, and why the idiomatic way to delete several elements is an algorithm rather than a loop. This lesson covers the size–capacity model, the growth rule, `emplace_back`, `insert` / `erase`, the erase–remove idiom and its C++20 replacement `std::erase_if`, the odd `std::vector<bool>`, `data()` and two-dimensional vectors.

Size and capacity

A vector holds three pointers: the start of its block, one past the last element (`size()`) and one past the block (`capacity()`). `size()` is how many elements exist; `capacity()` is how many fit before the block must be replaced.

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v;
    std::cout << v.size() << '\n';           // 0
    v.reserve(100);                          // one allocation, no elements
    std::cout << v.size() << ' ' << (v.capacity() >= 100) << '\n';   // 0 1
    for (int i = 0; i < 100; ++i) v.push_back(i);   // no reallocation happens
    v.resize(3);                             // destroys 97 elements; capacity
    v.clear();                               // destroys the rest; capacity is
    std::cout << v.size() << ' ' << v.empty() << '\n';   // 0 1
    v.shrink_to_fit();                       // a request to release the block
    return 0;
}
```

`resize(n)` changes the size — destroying elements or value-initialising new ones (`resize(n, x)` fills with `x`) — and `reserve(n)` changes only the capacity. Neither shrinks the block: `clear()` leaves the capacity in place so the vector can be refilled without reallocating. The

exact capacity after growth is unspecified by the standard, so a program must never print or compare it beyond `capacity() ⩾ n` right after a `reserve(n)`; a judged program prints `size()`.

The growth rule and amortised push_back

When `push_back` finds `size() == capacity()` it allocates a bigger block, moves (or copies, Module 9) every element across, destroys the old ones and frees the old block. libstdc++ doubles the capacity; MSVC multiplies by 1.5. Because the block grows geometrically, a sequence of n pushes performs O(n) element moves in total — each element is moved at most a constant number of times — so `push_back` is **amortised O(1)**: cheap on average, with an occasional expensive call.

`reserve(n)` before a loop of known length turns that occasional call into none. It avoids the moves, and — more importantly — the **invalidation**: a reallocation moves every element to a new address, so every pointer, reference and iterator into the old block is now dangling.

```
std::vector<int> v{1, 2, 3};
int& first = v.front();
v.push_back(4);            // may reallocate
std::cout << first;        // undefined behaviour if it did
```

The rule: never keep a reference or iterator into a vector across a `push_back`, `insert` or `resize` unless you reserved enough beforehand. Store an index instead — `v[0]` is recomputed from the current block every time.

---

push_back, emplace_back and the element type

`push_back(x)` copies or moves an existing object into the block. `emplace_back(args...)` constructs the element *in place* from constructor arguments, saving the temporary:

```
std::vector<std::string> words;
words.push_back("alpha");              // constructs a temporary std::string,
words.emplace_back("beta");            // constructs the std::string directly
words.emplace_back(3, 'x');            // std::string(3, 'x') → "xxx"
words.push_back(std::move(existing));  // moves; existing is left valid but u
```

Both are amortised O(1). `emplace_back` returns a reference to the new element since C++17, so `auto& w = words.emplace_back("gamma");` is legal — and dies at the next reallocation like any other reference.

---

insert and erase in the middle

`v.insert(pos, x)` shifts every element from `pos` to the end one place right, and `v.erase(pos)` shifts them one place left, so both are O(n) in the distance to the end. Both take an iterator, and `v.begin() + i` is how an index becomes one (random access makes that O(1)):

```
std::vector<int> v{10, 20, 30};
v.insert(v.begin() + 1, 15);        // 10 15 20 30
v.erase(v.begin());                 // 15 20 30
v.erase(v.begin() + 1, v.end());    // 15 — a half-open range
v.pop_back();                       // empty; pop_back on an empty vector is u
```

`erase` returns an iterator to the element after the one removed, which is the basis of the erase-while-iterating idiom in lesson 6. For inserting at the front repeatedly, a vector is the wrong container — every insert shifts everything — and `std::deque` is the answer (lesson 3).

Removing several elements: erase–remove and std::erase_if

Erasing one element at a time inside a loop is O(n²) — each `erase` shifts the tail — and the loop is easy to get wrong. The classic solution is two steps. `std::remove_if` (or `std::remove` for a value) walks the range once, copying the elements to keep towards the front, and returns an iterator to the new logical end; the elements from there on are left in a valid but unspecified state. `erase` then chops that tail:

```
#include <algorithm>
#include <vector>

std::vector<int> v{1, 2, 3, 4, 5, 6};
v.erase(std::remove_if(v.begin(), v.end(), [](int x) { return x % 2 == 0; }),
// v is 1 3 5
```

This is the **erase–remove idiom**, a single O(n) pass. It survives in a great deal of code and interviewers still ask for it, but since C++20 the standard spells it for you:

```
std::erase_if(v, [](int x) { return x % 2 == 0; });   // same effect, returns
std::erase(v, 3);                                       // remove every 3
```

`std::erase_if` exists for every standard container — `std::vector`, `std::deque`, `std::list`, `std::string`, `std::set`, `std::map` and the unordered ones — and returns how many elements went. Prefer it; the idiom is for code that predates C++20 or a range that is not a whole container. When the predicate needs state, such as a `std::set` of values already seen, capture it by reference so that a copy of the lambda shares it.

---

std::vector<bool> is not a vector of bool

`std::vector<bool>` is a specialisation that packs eight elements per byte. The saving is real, but it breaks the container contract: `v[i]` returns a proxy object rather than a `bool&`, so `auto b = v[0];` is not a `bool`, `bool& r = v[0];` does not compile, there is no `data()`, and threads writing neighbouring elements race. For flags with normal semantics use `std::vector<char>` or `std::deque<bool>`; for packed bits of a fixed size, `std::bitset<N>`.

---

data() and the contiguity guarantee

Because the elements are contiguous, `v.data()` is a `T*` to the first one and `v.data() + v.size()` is one past the last — the same `[begin, end)` pointer pair a C API or a built-in array uses (Module 6, lesson 3), and what `std::span<int>` (Module 16) wraps without owning. Contiguity is also why a plain index loop is as fast as it is: the next element is always in the same or the next cache line.

Two-dimensional vectors

A grid is a vector of vectors, sized in one expression: `std::vector<std::vector<int>> g(rows, std::vector<int>(cols, 0));`. Each row is its own block, so `g[r][c]` costs two indirections and rows may differ in length (a jagged array). For a grid indexed hot in a loop, a single `std::vector<int> flat(rows * cols)` indexed as `flat[r * cols + c]` keeps everything contiguous (Module 6, lesson 6). `std::vector<std::vector<int>> g(rows);` followed by `g[r][c] = x` is out of bounds — the inner vectors are empty until sized.

Pitfalls

Mistake	What happens
Keeping `int& r = v[i]` or an iterator across `push_back`	Dangling after a reallocation; undefined behaviour
`v[v.size()]` or `v[i]` with `i` past the end	No bounds check; undefined behaviour — `v.at(i)` throws `std::out_of_range`
`for (…) v.erase(v.begin() + i);` to remove several	O(n²) and skips the element after each erase
`std::remove_if(…)` without the `erase`	The vector keeps its old size; the tail holds leftovers
Printing `capacity()`	Implementation-specific numbers; print `size()`
Treating `std::vector<bool>` as a `std::vector`	Proxy elements, no `data()`, no `bool&`
`std::vector<int> v(10, 1)` vs `std::vector<int> v{10, 1}`	Ten ones vs the two elements 10 and 1 — braces prefer the initialiser list

Key takeaways

A vector is one contiguous block; `size()` is what is used, `capacity()` is what is allocated, and growth is geometric so `push_back` is amortised O(1).

Reallocation invalidates every pointer, reference and iterator into the vector; `reserve` ahead or hold indices.

`emplace_back` constructs in place from constructor arguments; `insert` / `erase` in the middle are O(n) and take iterators (`v.begin() + i`).

Delete several elements with `std::erase_if(v, pred)` (C++20) or the erase–remove idiom, never one `erase` per element in a loop.

`std::vector<bool>` is a packed proxy type; `data()` gives the `[begin, end)` pointer pair; a 2-D vector is sized as `g(rows, std::vector<int>(cols))` .

EXERCISE 1

Vector editor

Maintain a `std::vector<int>` and apply commands read until the end of input, one per line:

- `push x` — append `x`;
- `pop` — remove the last element, or print `error` if the vector is empty;
- `insert i x` — insert `x` before index `i` (`0 ≤ i ≤ size`), else print `error`;
- `erase i` — remove the element at index `i` (`0 ≤ i < size`), else print `error`;
- `size` — print `size=<n>`;
- `print` — print the elements as `[a, b, c]` (`[]` when empty).

Use `v.insert(v.begin() + i, x)` and `v.erase(v.begin() + i)` — the index becomes an iterator through random-access arithmetic. An index may be negative in the input; treat it as invalid.

Input: commands until end of input. **Output:** one line per `print`, `size` or failed command.

```
push 1
push 2
push 3
print
insert 1 9
print
erase 0
print
pop
print
```

prints

```
[1, 2, 3]
[1, 9, 2, 3]
[9, 2, 3]
[9, 2]
```

INPUT 1

```
push 1
push 2
push 3
print
insert 1 9
print
erase 0
print
pop
print
```

OUTPUT 1

```
[1, 2, 3]
[1, 9, 2, 3]
[9, 2, 3]
[9, 2]
```

INPUT 2

```
pop
print
push 5
insert 2 7
insert 1 7
print
erase 5
print
size
```

OUTPUT 2

```
error
[]
error
[5, 7]
error
[5, 7]
size=2
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <string>
#include <vector>

void print(const std::vector<int>& v) {
    std::cout << '[';
    for (std::size_t i = 0; i < v.size(); ++i) {
        if (i > 0) std::cout << ", ";
        std::cout << v[i];
    }
    std::cout << "]\n";
}

int main() {
    std::vector<int> v;
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "push") {
            int x;
            std::cin >> x;
            v.push_back(x);
        } else if (cmd == "pop") {
            if (v.empty()) std::cout << "error\n";
            else v.pop_back();
        } else if (cmd == "insert") {
            long long i;
            int x;
            std::cin >> i >> x;
            if (i < 0 || i > static_cast<long long>(v.size())) std::cout << "error\r
            else v.insert(v.begin() + i, x);
        } else if (cmd == "erase") {
            long long i;
            std::cin >> i;
            if (i < 0 || i ⩾ static_cast<long long>(v.size())) std::cout << "error\
            else v.erase(v.begin() + i);
        } else if (cmd == "size") {
            std::cout << "size=" << v.size() << '\n';
        } else if (cmd == "print") {
            print(v);
        }
    }
    return 0;
}
```

EXERCISE 2

Filter and dedupe with std::erase_if

Read an integer `n`, then `n` integers into a `std::vector<int>`, then a threshold `t`. In two passes, each a single call to `std::erase_if`:

- remove every value that is **less than** `t` and print `removed=<count>` — the count is the value `std::erase_if` returns;
- remove every value that has **already appeared** earlier in the vector (keep the first occurrence) and print `deduped=<count>` — the predicate keeps a `std::set<int>` of values seen so far, captured by reference.

Then print the remaining elements as `[a, b, c]` (`[]` when empty). Do not write an erase loop; the point is the algorithm.

Input: `n`, then `n` integers, then `t`. **Output:** three lines.

```
7
5 1 5 3 8 3 9
3
```

prints

```
removed=1
deduped=2
[5, 3, 8, 9]
```

INPUT 1

```
7
5 1 5 3 8 3 9
3
```

OUTPUT 1

```
removed=1
deduped=2
[5, 3, 8, 9]
```

INPUT 2

```
4
1 2 3 4
10
```

OUTPUT 2

```
removed=4
deduped=0
[]
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <iostream>
#include <set>
#include <vector>

int main() {
    int n;
    std::cin >> n;
    std::vector<int> v(n);
    for (int& x : v) std::cin >> x;
    int t;
    std::cin >> t;
    const auto removed = std::erase_if(v, [t](int x) { return x < t; });
    std::cout << "removed=" << removed << '\n';
    std::set<int> seen;
    const auto deduped = std::erase_if(v, [&seen](int x) { return !seen.insert(x).se
    std::cout << "deduped=" << deduped << '\n';
    std::cout << '[';
    for (std::size_t i = 0; i < v.size(); ++i) {
        if (i > 0) std::cout << ", ";
        std::cout << v[i];
    }
    std::cout << "]\n";
    return 0;
}
```

QUIZ

1. What does this print?

```cpp
std::vector<int> v;
v.reserve(10);
std::cout << v.size() << ' ' << (v.capacity() >= 10);
```

A. `10 1`

B. `0 1`

C. `0 0`

D. `10 0`

2. `v.push_back(x)` runs on a vector whose `size()` equalled its `capacity()`. Which is true afterwards?

A. Only `end()` is invalidated

B. Every iterator, pointer and reference into `v` is invalidated

C. Nothing is invalidated: `push_back` never moves elements

D. Only iterators before the insertion point stay valid

3. Which removes every `3` from `std::vector<int> v` in one O(n) pass?

A. `v.remove(3);`

B. `std::remove(v.begin(), v.end(), 3);` on its own

C. `std::erase(v, 3);`

D. `for (auto it = v.begin(); it ≠ v.end(); ++it) if (*it == 3) v.erase(it);`

4. What does this print?

```
std::vector<int> v{1, 2, 3, 4, 5};
v.erase(std::remove_if(v.begin(), v.end(), [](int x) { return x % 2 == 0; }), v
std::cout << v.size();
```

A. 5

B. 3

C. 2

D. undefined behaviour

5. `std::vector<bool> flags(8); auto f = flags[0];` — what is the type of `f`?

A. `bool`

B. `bool&`

C. A proxy object that stands for one bit

D. `char`

6. What does this print?

```
std::vector<std::string> v;
v.emplace_back(3, 'x');
std::cout << v[0];
```

A. `3x`

B. `xxx`

C. compile error

D. `x3`

7. What does this print?

```
std::vector<int> a(3, 7);
std::vector<int> b{3, 7};
std::cout << a.size() << b.size();
```

A. `22`

B. `32`

C. `23`

D. `33`

ANSWER KEY · STD::VECTOR IN DEPTH — SIZE, CAPACITY AND THE ERASE–REMOVE IDIOM

1. B `reserve` allocates room for at least 10 elements but creates none, so `size()` is still 0. `resize(10)` is the call that would make ten elements.

2. B A full vector reallocates: it moves every element to a new block and frees the old one, so anything that pointed into the old block dangles. Only when there is spare capacity is `end()` the sole casualty.

3. C `std::erase` (C++20) is the erase–remove idiom in one call. `std::vector` has no `remove` member (only `std::list` does); `std::remove` alone leaves the vector the same size with leftovers at the tail; the loop is O(n²) and skips elements after each erase.

4. B `remove_if` moves the odd elements to the front and returns the new logical end; `erase` chops the tail. Three odd numbers remain.

5. C `std::vector<bool>` packs bits, so `operator[]` cannot return a reference to a `bool`; it returns a small proxy class. `bool f = flags[0];` converts it; `auto` keeps the proxy.

6. B `emplace_back` forwards its arguments to the element's constructor: `std::string(3, 'x')` is three copies of `x`. `push_back(3, 'x')` would not compile.

7. B Parentheses call the (count, value) constructor: three sevens. Braces prefer the initialiser-list constructor: the two elements 3 and 7.

LESSON 3 · 14 MIN

deque, list and the adaptors — stack, queue and priority_queue

Two sequence containers and three adaptors cover the situations a vector does not. `std::deque` is for when both ends move; `std::list` is for when elements must stay put while the collection changes around them; and `std::stack`, `std::queue` and `std::priority_queue` are for when a discipline — last in first out, first in first out, largest first — is the whole point and a wider interface would only invite mistakes. This lesson covers the layout and cost of each, the `pop()` that returns nothing and why, the comparator convention of `std::priority_queue` that everyone gets backwards once, and the breadth-first search a queue was made for.

---

std::deque — stable at both ends

A deque (double-ended queue) stores its elements in fixed-size blocks and keeps a small index of block pointers. Pushing at either end adds to the current end block or allocates a new one; nothing already stored is moved. Indexing is O(1) — block number and offset — so `d[i]`, `front()`, `back()`, `push_front`, `push_back`, `pop_front` and `pop_back` are all constant time:

```
#include <deque>
#include <iostream>

int main() {
    std::deque<int> window;
    for (int x : {4, 8, 15, 16, 23, 42}) {
        window.push_back(x);
        if (window.size() > 3) window.pop_front();       // keep the last thre
    }
    for (int x : window) std::cout << x << ' ';           // 16 23 42
    std::cout << '\n' << window[1] << '\n';               // 23
    return 0;
}
```

The blocks mean a deque is not contiguous: there is no `data()`, iteration is a little slower than a vector's, and inserting in the middle is still O(n). Its invalidation rule is worth memorising: a push at either end invalidates every *iterator* but no *reference*, because the elements

themselves do not move; an insert in the middle invalidates everything. It is the right container for a sliding window, a work queue consumed from the front, and any buffer that grows at one end and shrinks at the other.

---

std::list — nodes that never move

A `std::list` is a doubly linked list: each element lives in its own heap node with pointers to its neighbours. That layout gives three guarantees no contiguous container can: inserting or erasing anywhere is O(1) *once you hold an iterator to the spot*; an iterator, pointer or reference to an element stays valid until that element is erased, whatever happens to the rest of the list; and `splice` moves elements between lists, or within one, by relinking nodes without copying anything.

```
#include <iostream>
#include <list>

int main() {
    std::list<int> l{1, 2, 3};
    auto two = std::next(l.begin());        // iterator to 2
    l.push_front(0);
    l.push_back(4);                         // two is still valid
    l.insert(two, 99);                      // 0 1 99 2 3 4 — O(1), two still
    l.erase(two);                           // 0 1 99 3 4
    std::list<int> other{7, 8};
    l.splice(l.end(), other);               // 0 1 99 3 4 7 8; other is now em
    l.sort();                               // a member: std::sort needs rando
    for (int x : l) std::cout << x << ' ';  // 0 1 3 4 7 8 99
    std::cout << '\n';
    return 0;
}
```

What a list gives up is everything positional: there is no `operator[]`, reaching the k-th element is O(k), every node is a separate allocation, and iteration chases pointers across the heap, which the cache hates. Measured, a vector beats a list at almost every workload until the elements are large and the collection is huge (Module 19). The honest reasons to use one are the guarantees above — stable addresses and O(1) splicing — and the classic use is an LRU cache, where a list keeps recency order and a hash map holds iterators into it (the checkpoint builds one). `std::forward_list` is the singly linked version: smaller nodes, forward iteration only.

The adaptors

An adaptor holds a sequence container and exposes only the operations of one discipline. `std::stack<T>` and `std::queue<T>` wrap a `std::deque<T>` by default; `std::priority_queue<T>` wraps a `std::vector<T>` kept as a binary heap. None of them has iterators or `operator[]` — if you need to look inside, you wanted the underlying container.

Adaptor	Discipline	Inspect	Push	Pop	Cost
`std::stack`	last in, first out	`top()`	`push` / `emplace`	`pop()`	O(1)
`std::queue`	first in, first out	`front()`, `back()`	`push` / `emplace`	`pop()`	O(1)
`std::priority_queue`	largest first	`top()`	`push` / `emplace`	`pop()`	O(log n) push and pop

`pop()` returns `void` on all three, and the reason is worth knowing for interviews. If `pop()` returned the element by value and the copy threw an exception (a `std::string` that cannot allocate, say), the element would already be gone from the container and lost for good. Separating `top()` (look) from `pop()` (remove) keeps the strong exception guarantee (Module 15, lesson 2) — so the idiom is always two lines: read `top()` or `front()`, then `pop()`. Calling `top()`, `front()` or `pop()` on an empty adaptor is undefined behaviour; check `empty()` first.

std::priority_queue and its comparator

The default `std::priority_queue<int>` is a max-heap: `top()` is the largest element. For the smallest first, supply the underlying container and `std::greater`:

```
#include <functional>
#include <queue>
#include <vector>

std::priority_queue<int> largestFirst;
std::priority_queue<int, std::vector<int>, std::greater<int>> smallestFirst;
```

For anything richer, a comparator. The convention trips everyone once: the comparator is called as `comp(a, b)` and must return `true` when `a` has **lower** priority than `b` — when `a` should come out *after* `b`. `std::less` puts the greatest on top because "a < b" means "a comes out after b". A scheduler that runs the highest priority first and breaks ties by the earlier deadline therefore says:

```
struct Task {
    std::string name;
    int priority;
    int deadline;
};

struct ByUrgency {
    bool operator()(const Task& a, const Task& b) const {
        if (a.priority != b.priority) return a.priority < b.priority;   // lo
        return a.deadline > b.deadline;                                  // l
    }
};

std::priority_queue<Task, std::vector<Task>, ByUrgency> tasks;
tasks.push({"deploy", 5, 30});
tasks.emplace("lint", 2, 10);
const Task& next = tasks.top();       // deploy
```

A lambda works too — `std::priority_queue<Task, std::vector<Task>, decltype(cmp)> pq(cmp);` — because the comparator type is part of the queue's type and a lambda object must be handed to the constructor. For quick work, `std::priority_queue<std::pair<int, std::string>>` orders pairs lexicographically, by the `int` then the string, with no comparator at all. The comparator must be a strict weak ordering (`false` for equal elements), and the queue offers no way to change an element's priority once it is in — the usual trick is to push a fresh entry and skip stale ones as they surface.

Breadth-first search with std::queue

A queue is the data structure of breadth-first search: process the frontier in the order it was discovered, and the first time the goal comes off the queue you have the shortest path. On a grid with four-neighbour moves:

```cpp
#include <queue>
#include <utility>
#include <vector>

int shortest(const std::vector<std::string>& grid, std::pair<int, int> start,
    const int rows = static_cast<int>(grid.size());
    const int cols = static_cast<int>(grid[0].size());
    std::vector<std::vector<int>> dist(rows, std::vector<int>(cols, -1));
    std::queue<std::pair<int, int>> frontier;
    dist[start.first][start.second] = 0;
    frontier.push(start);
    const int dr[] = {-1, 1, 0, 0};
    const int dc[] = {0, 0, -1, 1};
    while (!frontier.empty()) {
        auto [r, c] = frontier.front();
        frontier.pop();
        if (r == goal.first && c == goal.second) return dist[r][c];
        for (int k = 0; k < 4; ++k) {
            const int nr = r + dr[k], nc = c + dc[k];
            if (nr < 0 || nr >= rows || nc < 0 || nc >= cols) continue;
            if (grid[nr][nc] == '#' || dist[nr][nc] != -1) continue;
            dist[nr][nc] = dist[r][c] + 1;
            frontier.push({nr, nc});
        }
    }
    return -1;
}
```

The `dist` grid doubles as the visited set — mark a cell *when it is pushed*, not when it is popped, or the same cell enters the queue many times. Swap the `std::queue` for a `std::stack` and the same loop is depth-first search; swap it for a `std::priority_queue` keyed on distance and it is Dijkstra's algorithm.

Pitfalls

Mistake	What happens
`auto x = st.pop();`	Does not compile: `pop()` returns `void`; read `top()` first
`top()` / `front()` / `pop()` on an empty adaptor	Undefined behaviour; test `empty()`
Comparator returns `a.priority > b.priority` for "highest first"	Inverted — the *lowest* priority surfaces first
`std::sort(l.begin(), l.end())` on a `std::list`	Does not compile (needs random access); call `l.sort()`
Choosing `std::list` for "lots of inserts"	Usually slower than a vector; the reason for a list is stable iterators or `splice`
Marking BFS cells visited on pop instead of push	Cells enqueued repeatedly; still correct but far slower

Key takeaways

- `std::deque` is O(1) at both ends with O(1) indexing; pushes at the ends keep references valid but invalidate iterators.
- `std::list` gives stable iterators, O(1) insert/erase at a known position and O(1) `splice`; it pays with no indexing and cache-unfriendly nodes.
- `std::stack`, `std::queue` and `std::priority_queue` expose only their discipline; `pop()` returns `void` for exception safety, so read then pop.
- `std::priority_queue` is a max-heap by default; `std::greater` flips it; a comparator returns `true` when the first argument has lower priority.
- A queue is breadth-first search; mark visited on push.

EXERCISE 1

Sliding window maximum on a std::deque

Read `n` and `k` (`1 ≤ k ≤ n`), then `n` integers. Print the maximum of every window of `k` consecutive elements, in order, space-separated on one line.

Do it in O(n) with a **monotonic deque of indices**: before pushing index `i` at the back, pop from the back every index whose value is `≤ a[i]` (it can never be a maximum again); then pop the front if it has fallen out of the window (`front ≤ i - k`). The front of the deque is always the index of the current window's maximum. This is the exercise that uses all four ends of `std::deque`: `push_back`, `pop_back`, `front`, `pop_front`.

Input: `n k`, then `n` integers. **Output:** `n - k + 1` integers on one line.

Example: `8 3` / `1 3 -1 -3 5 3 6 7` → `3 3 5 5 6 7`

INPUT 1

```
8 3
1 3 -1 -3 5 3 6 7
```

OUTPUT 1

```
3 3 5 5 6 7
```

INPUT 2

```
5 1
4 2 9 1 7
```

OUTPUT 2

```
4 2 9 1 7
```

REFERENCE SOLUTION

```cpp
#include <deque>
#include <iostream>
#include <vector>

int main() {
    int n, k;
    std::cin >> n >> k;
    std::vector<int> a(n);
    for (int& x : a) std::cin >> x;
    std::deque<int> window;   // indices into a; values decrease from front to back
    bool first = true;
    for (int i = 0; i < n; ++i) {
        while (!window.empty() && a[window.back()] ⩽ a[i]) window.pop_back();
        window.push_back(i);
        if (window.front() ⩽ i - k) window.pop_front();
        if (i ⩾ k - 1) {
            if (!first) std::cout << ' ';
            std::cout << a[window.front()];
            first = false;
        }
    }
    std::cout << '\n';
    return 0;
}
```

EXERCISE 2

Task scheduler on std::priority_queue

Simulate a scheduler. Commands arrive one per line until the end of input: `add <name> <priority>` queues a task (the name is one word, the priority an integer, possibly negative), and `run` takes the task that should run next and prints `run <name> p=<priority>`, or `idle` when nothing is queued.

The task to run next is the one with the **highest priority**; among equal priorities, the **alphabetically smaller name** runs first (plain `std::string` comparison). Two tasks may have the same name and priority. Store the tasks in a `std::priority_queue` of a `Task` struct with a **custom comparator** — a struct whose `operator()(a, b)` returns `true` when `a` should run *after* `b`.

Input: commands until end of input. **Output:** one line per `run`.

```
add build 2
add test 5
add lint 2
run
run
run
run
```

prints

```
run test p=5
run build p=2
run lint p=2
idle
```

INPUT 1

```
add build 2
add test 5
add lint 2
run
run
run
run
```

OUTPUT 1

```
run test p=5
run build p=2
run lint p=2
idle
```

INPUT 2

```
run
add a 1
add b 1
add c 9
run
add d 9
run
run
run
```

OUTPUT 2

```
idle
run c p=9
run d p=9
run a p=1
run b p=1
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <queue>
#include <string>
#include <vector>

struct Task {
    std::string name;
    int priority;
};

struct RunsLater {
    // true when a should run AFTER b: lower priority, or same priority and a later
    bool operator()(const Task& a, const Task& b) const {
        if (a.priority != b.priority) return a.priority < b.priority;
        return a.name > b.name;
    }
};

int main() {
    std::priority_queue<Task, std::vector<Task>, RunsLater> tasks;
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "add") {
            Task t;
            std::cin >> t.name >> t.priority;
            tasks.push(t);
        } else if (cmd == "run") {
            if (tasks.empty()) {
                std::cout << "idle\n";
            } else {
                const Task& next = tasks.top();
                std::cout << "run " << next.name << " p=" << next.priority << '\n';
                tasks.pop();
            }
        }
    }
    return 0;
}
```

QUIZ

1. Which container gives O(1) `push_front`, O(1) `push_back` *and* O(1) `operator[]`?

A. `std::vector`

B. `std::list`

C. `std::deque`

D. `std::forward_list`

2. Why does `std::stack::pop()` return `void` instead of the removed element?

A. To make it faster: returning a value would copy

B. Exception safety: if returning the value threw, the element would already be gone

C. Because the underlying `std::deque` has no way to return the back

D. A historical accident kept for compatibility

3. What does this print?

```
std::priority_queue<int> pq;
for (int x : {3, 9, 1}) pq.push(x);
pq.pop();
std::cout << pq.top();
```

A. 9

B. 3

C. 1

D. undefined behaviour

4. How do you declare a priority queue of `int` whose `top()` is the *smallest* element?

A. `std::priority_queue<int, std::greater<int>>`

B. `std::priority_queue<int, std::vector<int>, std::greater<int>>`

C. `std::priority_queue<int>` then call `pq.reverse()`

D. `std::min_priority_queue<int>`

5. A comparator `cmp(a, b)` for `std::priority_queue` should return `true` when…

A. `a` should come out before `b`

B. `a` should come out after `b` — `a` has the lower priority

C. `a` and `b` are equal

D. `a` is greater than `b`, always

6. What does this print?

```
std::list<int> l{1, 2, 3};
auto it = std::next(l.begin());
l.push_front(0);
l.push_back(4);
std::cout << *it;
```

A. 2

B. 1

C. 0

D. undefined behaviour

7. Which pair of calls reads and removes the oldest element of a `std::queue<int> q`?

A. `q.top()` then `q.pop()`

B. `q.front()` then `q.pop()`

C. `q.pop()` returns it

D. `q.back()` then `q.pop_front()`

ANSWER KEY · DEQUE, LIST AND THE ADAPTORS — STACK, QUEUE AND PRIORITY_QUEUE

1. C A deque's block structure grows at both ends and still indexes in constant time. A vector's `push_front` would be O(n); a list has no `operator[]`.

2. B A `pop()` that returned by value could remove the element and then throw while copying it out, losing it. Separating `top()` from `pop()` keeps the strong exception guarantee.

3. B The default `std::priority_queue` is a max-heap: `pop()` removes 9, so `top()` is now 3.

4. B The comparator is the third template argument, so the underlying container must be named as the second. There is no `reverse()` and no `min_priority_queue`.

5. B The heap keeps the element for which nothing else 'is less' on top. With `std::less`, 'a < b' means a comes out after b, which is why the default is a max-heap.

6. A List nodes never move: inserting at either end leaves every existing iterator valid, so `it` still points at 2. On a `std::vector` the same code would be undefined behaviour after a reallocation.

7. B A queue exposes `front()` (oldest) and `back()` (newest); `top()` belongs to `std::stack` and `std::priority_queue`, and `pop()` returns nothing.

LESSON 4 · 14 MIN

std::map and std::set — the ordered containers

`std::map` and `std::set` are balanced binary trees (red–black trees in every mainstream implementation) that keep their elements sorted by key. Everything about them follows from that: lookup, insertion and erasure are O(log n); iteration visits the elements in key order for free; and "the first key not less than k" is a single call, which is the question a hash table cannot answer at all. This lesson covers the interface that matters — `operator[]` and the insertion it hides, `find` / `count` / `contains`, what `insert` returns, `emplace` and `try_emplace`, `lower_bound` / `upper_bound`, in-order iteration with structured bindings — then custom comparators, the `multi` variants, and `extract` / `merge` in brief.

The model

A `std::set<Key>` holds unique keys; a `std::map<Key, T>` holds unique keys, each with a value, stored as `std::pair<const Key, T>`. The key is `const` because changing it in place would break the tree's ordering. Ordering comes from a comparator, `std::less<Key>` by default, so the key type needs `operator<` (or a comparator you supply); `std::string`, the arithmetic types, `std::pair` and `std::tuple` all qualify. Each element is a separately allocated node, so — as with `std::list` — iterators, pointers and references to an element stay valid until that element is erased, no matter how many others are inserted.

```
#include <iostream>
#include <map>
#include <string>

int main() {
    std::map<std::string, int> stock{{"pear", 4}, {"apple", 7}};
    stock["fig"] = 12;                    // inserts fig
    stock["apple"] += 1;                  // updates apple
    for (const auto& [name, qty] : stock) {
        std::cout << name << ' ' << qty << '\n';   // apple 8, fig 12, pear 4
    }
    return 0;
}
```

operator[] inserts

`m[key]` returns a reference to the value for `key`, *inserting a value-initialised element first if the key is absent* — `0` for numbers, an empty string, an empty vector. That is exactly right for counting (`++counts[word]`) and grouping (`groups[key].push_back(x)`), and exactly wrong for looking up: `if (m["ghost"] == 0)` has just created `ghost`. It also means `operator[]` is not available on a `const std::map` (it cannot insert) and requires the value type to be default-constructible. For a lookup that must not insert, use `find`, `count`, `contains` (C++20) or `at` (throws `std::out_of_range` when absent):

```
if (auto it = stock.find("kiwi"); it ≠ stock.end()) std::cout << it→second;
if (stock.contains("fig")) std::cout << stock.at("fig");
std::cout << stock.count("pear");   // 0 or 1 in a map; the "how many" form fa
```

The `if` with initialiser (Module 3, lesson 1) keeps the iterator scoped to the test.

insert, emplace and what they return

`insert` and `emplace` never overwrite. They return a `std::pair<iterator, bool>`: the iterator points at the element with that key (new or pre-existing) and the `bool` says whether an insertion happened. Reading that `bool` is how "first time we have seen this key" is written without a separate lookup:

```
std::map<std::string, int> ids;
auto [where, inserted] = ids.insert({"ada", 1});     // inserted == true
auto [again, second] = ids.insert({"ada", 2});       // second == false; ada i
std::cout << again→second << '\n';                    // 1
ids.emplace("bob", 2);                                // constructs the pair i
ids.try_emplace("bob", 99);                           // C++17: does nothing,
ids.insert_or_assign("bob", 3);                       // C++17: overwrites; bo
```

`try_emplace` matters when constructing the value is expensive: unlike `emplace`, it builds nothing at all when the key exists. `insert_or_assign` is the spelling of "put" that overwrites and still reports, through its `bool`, whether the key was new. On a `std::set`, `s.insert(x).second` is the one-line membership-and-add.

lower_bound and upper_bound

`lower_bound(k)` returns an iterator to the first element whose key is **not less than** `k`; `upper_bound(k)` to the first whose key is **greater than** `k`; both return `end()` when there is none, and `equal_range(k)` gives both at once. They are the reason to choose an ordered container: "the next departure at or after 09:30", "the largest key below x", "every key in [lo, hi)" are all one or two calls:

```
std::set<int> departures{600, 645, 700, 900};
auto next = departures.lower_bound(650);          // → 700
auto prevIt = departures.lower_bound(650);
if (prevIt ≠ departures.begin()) --prevIt;        // → 645, the last key < 65
for (auto it = departures.lower_bound(640); it ≠ departures.upper_bound(700);
    std::cout << *it << ' ';                      // 645 700 — the keys in [
}
```

Use the **member** versions. The free `std::lower_bound(s.begin(), s.end(), k)` from `<algorithm>` compiles on a set but walks it linearly, because a tree iterator is bidirectional, not random access — O(n) instead of O(log n).

Erasing

`m.erase(key)` removes the element with that key and returns how many it removed (0 or 1 for a map or set); `m.erase(it)` removes the element at an iterator and returns the iterator after it; `m.erase(first, last)` removes a range. Erasing invalidates only iterators to the erased element, which is what makes the `it = m.erase(it)` loop of lesson 6 safe, and `std::erase_if(m, pred)` (C++20) removes every element matching a predicate on the pair.

Custom comparators

The third template argument decides the order. `std::greater<Key>` reverses it; a struct with a `const operator()` (a functor, Module 11, lesson 4) orders by anything you like. The comparator must be a **strict weak ordering**: irreflexive (`comp(a, a)` is false), so `⩽` is never correct — a comparator that returns `true` for equal keys corrupts the tree and the symptoms are lost elements and infinite loops.

```
struct ByLengthThenText {
    bool operator()(const std::string& a, const std::string& b) const {
        if (a.size() != b.size()) return a.size() < b.size();
        return a < b;
    }
};
std::set<std::string, ByLengthThenText> words{"pear", "fig", "apple", "kiwi"};
// fig kiwi pear apple

std::map<std::string, int, std::less<>> byName;    // transparent comparator
byName.find(std::string_view{"ada"});               // looks up without build
```

`std::less<>` — the transparent comparator, C++14 — lets `find`, `count`, `lower_bound` and `contains` take anything comparable with the key (a `std::string_view`, a `const char*`) without constructing a temporary `std::string`. It is worth writing by default for string-keyed maps.

---

multiset and multimap

The `multi` variants allow equal keys and keep them in insertion order among themselves. `insert` always succeeds and returns a plain iterator; `count(k)` is the number of matches; `equal_range(k)` yields the run of them; `erase(k)` removes **all** of them — erase one with `erase(find(k))`. A `std::multiset<int>` is a sorted bag that supports "remove one copy of x" in O(log n), which makes it the standard choice for a running median or a sliding-window minimum with deletions.

---

extract and merge, in brief

C++17 added node handles. `m.extract(key)` (or `extract(it)`) unlinks the node and returns a handle that owns it; the handle's `key()` is *mutable*, so a key can be changed and the node put back with `m.insert(std::move(handle))` — no copy of the value, no reallocation. `a.merge(b)` moves every node of `b` whose key is absent from `a` into `a`, leaving the duplicates in `b`. Both are rare in everyday code and common in interview follow-ups.

Pitfalls

Mistake	What happens
`if (m[key] == 0)` to test presence	Inserts `key` with `0`; use `find`, `count` or `contains`
`m[key]` on a `const std::map&` parameter	Does not compile; use `find` or `at`
Expecting `insert` to overwrite	It keeps the old value; use `m[k] = v` or `insert_or_assign`
`std::lower_bound(s.begin(), s.end(), k)` on a set	Correct but O(n); call `s.lower_bound(k)`
A comparator using `<=`	Not a strict weak ordering; the tree misbehaves
`for (auto [k, v] : m)`	Copies every pair; `const auto& [k, v]` reads, `auto& [k, v]` modifies `v`
`ms.erase(x)` on a multiset to remove one copy	Removes all copies; `ms.erase(ms.find(x))`

Key takeaways

`std::map` / `std::set` are balanced trees: O(log n) find, insert and erase; iteration is in key order; elements are `pair<const Key, T>` nodes whose iterators survive other insertions.

`operator[]` inserts a value-initialised element when the key is absent — right for counting, wrong for lookups; use `find`, `count`, `contains` or `at`.

`insert` / `emplace` never overwrite and return `{iterator, bool}`; `try_emplace` skips construction, `insert_or_assign` overwrites.

Member `lower_bound` / `upper_bound` answer "first key ≥ k" and "first key > k" in O(log n) — the reason to pick an ordered container.

A comparator is a strict weak ordering (never `<=`); `std::less<>` allows heterogeneous lookup; `multi` variants keep duplicates and `erase(key)` removes them all.

EXERCISE 1

Word frequencies in key order

Read all of standard input as whitespace-separated tokens. Normalise each token by keeping only its letters and digits (drop every other character) and lower-casing them — use `<cctype>` with the `unsigned char` cast — and skip a token that becomes empty. Count the words in a `std::map<std::string, int>` with `++freq[word]`, then print every `<word> <count>` on its own line **in the map's own order** (no sorting step), followed by `distinct=<number of words> total=<number of tokens counted>`.

Input: free text until end of input (possibly empty). **Output:** one line per distinct word, then the summary line.

```
The cat saw the other cat.
A cat!
```

prints

```
a 1
cat 3
other 1
saw 1
the 2
distinct=5 total=8
```

INPUT 1

```
The cat saw the other cat.
A cat!
```

OUTPUT 1

```
a 1
cat 3
other 1
saw 1
the 2
distinct=5 total=8
```

INPUT 2

```
b a B A b
```

OUTPUT 2

```
a 2
b 3
distinct=2 total=5
```

REFERENCE SOLUTION

```cpp
#include <cctype>
#include <iostream>
#include <map>
#include <string>

std::string normalise(const std::string& token) {
    std::string out;
    for (char c : token) {
        const unsigned char uc = static_cast<unsigned char>(c);
        if (std::isalnum(uc)) out += static_cast<char>(std::tolower(uc));
    }
    return out;
}

int main() {
    std::map<std::string, int> freq;
    long long total = 0;
    std::string token;
    while (std::cin >> token) {
        const std::string word = normalise(token);
        if (word.empty()) continue;
        ++freq[word];
        ++total;
    }
    for (const auto& [word, count] : freq) std::cout << word << ' ' << count << '\n'
    std::cout << "distinct=" << freq.size() << " total=" << total << '\n';
    return 0;
}
```

EXERCISE 2

Next departure with lower_bound

A timetable is a `std::set<int>` of departure times (minutes past midnight; the input may repeat a time — the set collapses it). Read `n`, then `n` times, then `q`, then `q` query times. For each query `t` print `<t>: next=<x> prev=<y>` where `x` is the first departure **at or after** `t` and `y` the last departure **strictly before** `t`; print `none` for either when it does not exist.

Use the set's **member** `lower_bound` for `next`, and `std::prev` of that iterator for `prev` — after checking it is not `begin()`.

Input: `n`, `n` integers, `q`, `q` integers. **Output:** `q` lines.

```
5
600 645 700 900 1200
4
630
700
1300
0
```

prints

```
630: next=645 prev=600
700: next=700 prev=645
1300: next=none prev=1200
0: next=600 prev=none
```

INPUT 1

```
5
600 645 700 900 1200
4
630
700
1300
0
```

OUTPUT 1

```
630: next=645 prev=600
700: next=700 prev=645
1300: next=none prev=1200
0: next=600 prev=none
```

INPUT 2

```
3
5 5 5
2
5
6
```

OUTPUT 2

```
5: next=5 prev=none
6: next=none prev=5
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <iterator>
#include <set>

int main() {
    int n;
    std::cin >> n;
    std::set<int> times;
    for (int i = 0; i < n; ++i) {
        int t;
        std::cin >> t;
        times.insert(t);
    }
    int q;
    std::cin >> q;
    for (int i = 0; i < q; ++i) {
        int t;
        std::cin >> t;
        const auto next = times.lower_bound(t);
        std::cout << t << ": next=";
        if (next == times.end()) std::cout << "none";
        else std::cout << *next;
        std::cout << " prev=";
        if (next == times.begin()) std::cout << "none";
        else std::cout << *std::prev(next);
        std::cout << '\n';
    }
    return 0;
}
```

QUIZ

1. What does this print?

```cpp
std::map<std::string, int> m;
if (m["x"] == 0) std::cout << m.size();
```

A. 0

B. 1

C. nothing is printed

D. compile error

2. What does this print?

```
std::map<std::string, int> m;
m.insert({"a", 1});
m.insert({"a", 2});
std::cout << m["a"];
```

A. 1

B. 2

C. 3

D. undefined behaviour

3. `std::set<int> s{1, 3, 5, 7};` — what do `*s.lower_bound(5)` and `*s.upper_bound(5)` give?

A. 5 and 5

B. 3 and 5

C. 5 and 7

D. 3 and 7

4. Which statement about iterating a `std::map` is correct?

A. Elements come in insertion order

B. Elements come in ascending key order and each is a `std::pair<const Key, T>`

C. Elements come in an unspecified order

D. Elements come in descending key order by default

5. A function takes `const std::map<std::string, int>& m`. Which lookup compiles?

A. `m["k"]`

B. `m.find("k")`

C. `m["k"] = 0`

D. `m.operator[]("k")`

6. Which call is O(log n) on a `std::map` with n elements?

A. `m.size()`

B. `m.find(k)`

C. `std::distance(m.begin(), it)`

D. `m.clear()`

7. What does this print?

```
std::set<int, std::greater<int>> s{1, 2, 3};
std::cout << *s.begin();
```

A. 1

B. 3

C. 2

D. compile error

ANSWER KEY · STD::MAP AND STD::SET — THE ORDERED CONTAINERS

1. B `m["x"]` inserts `x` with a value-initialised `0` before the comparison, so the condition is true and the map now has one element.

2. A `insert` never overwrites: the second call returns `{iterator, false}` and leaves the value at 1. `m["a"] = 2` or `insert_or_assign` would overwrite.

3. C `lower_bound` is the first element not less than 5 (5 itself); `upper_bound` is the first element greater than 5 (7).

4. B A map is a sorted tree; the key is `const` in the pair because changing it would break the ordering. Insertion order is what a vector keeps; unspecified order is the unordered containers.

5. B `operator[]` may insert, so it has no `const` overload and cannot be called on a const map in any spelling. `find`, `count`, `contains` and `at` are the const lookups.

6. B `find` walks one root-to-leaf path of the balanced tree. `size()` is O(1), `clear()` destroys every node in O(n), and `std::distance` steps a bidirectional iterator one element at a time.

7. B The comparator decides the order: with `std::greater` the set runs 3, 2, 1, so `begin()` is the largest element.

LESSON 5 · 13 MIN

The unordered containers — hashing, buckets and custom keys

`std::unordered_map` and `std::unordered_set` answer the same questions as their ordered cousins — is this key present, what is its value, add it, remove it — in average constant time instead of logarithmic, by hashing the key to a bucket instead of walking a tree. The price is order: iteration visits elements in whatever sequence the table happens to hold them, which changes with the number of elements, the library version and the hash function, so any program that *prints* an unordered container must sort first. This lesson covers how the table works (hash, buckets, load factor, rehashing), the average and worst cases, the interface, the rule for deterministic output, how to make a struct usable as a key, and when to choose ordered over unordered.

How the table works

An unordered container keeps an array of **buckets**. To store a key it computes `std::hash<Key>{}(key)`, a `std::size_t`, reduces it modulo the bucket count to pick a bucket, and links the element into that bucket's chain. Lookup repeats the hash, walks the one chain and compares with `operator==` (`std::equal_to<Key>`) until it finds the key or the chain ends. With a good hash and enough buckets the chains are short — one or two elements — and every operation is O(1) on average.

The **load factor** is `size() / bucket_count()`. When an insertion would push it past `max_load_factor()` (1.0 by default), the table **rehashes**: it allocates more buckets and re-links every element. Rehashing is O(n) but rare, so insertion stays amortised O(1); it also invalidates every iterator, while pointers and references to elements stay valid because elements are nodes that are re-linked rather than moved. `reserve(n)` sizes the table for `n` elements in advance and removes the rehashes from a loop of known length, exactly as `std::vector::reserve` removes reallocations.

```cpp
#include <iostream>
#include <string>
#include <unordered_map>

int main() {
    std::unordered_map<std::string, int> counts;
    counts.reserve(64);
    for (std::string word; std::cin >> word;) ++counts[word];   // operator[]
    std::cout << counts.size() << " distinct words\n";
    if (auto it = counts.find("the"); it != counts.end()) std::cout << "the: "
    std::cout << counts.count("zebra") << '\n';                 // 0 — no inse
    return 0;
}
```

The interface mirrors `std::map`: `operator[]` inserts a value-initialised element, `find` / `count` / `contains` look up without inserting, `insert` / `emplace` return `{iterator, bool}`, `try_emplace` and `insert_or_assign` behave as in lesson 4, `erase(key)` returns the count removed. What is missing is everything that needs order: no `lower_bound`, no `upper_bound`, no comparator, no reverse iterators.

Average O(1), worst O(n)

The constant-time claim depends on the keys spreading across buckets. If many keys hash to the same bucket — a poor hash function, or an adversary who chose the keys knowing the hash — one chain holds everything and every operation degrades to O(n). `std::hash` for integers is the identity in libstdc++, which is fine for typical data and terrible for keys that are all multiples of the bucket count; `std::hash<std::string>` is a proper mixing function. The practical rule: for untrusted input where a worst case is exploitable, or for a guaranteed bound, use `std::map`; for everything else the unordered container is faster and the constant matters more than the log.

Iteration order is unspecified

```cpp
std::unordered_set<int> seen{3, 1, 2};
for (int x : seen) std::cout << x << ' ';    // 3 1 2? 2 1 3? — the standard d
```

The order is a function of the hash values, the bucket count and the insertion history. It differs between libstdc++ and libc++, between releases, and between a table of ten elements and the same table after a rehash. A program whose output depends on it is not wrong, but it is not deterministic, and a judged exercise is compared byte for byte. The rule is mechanical: **copy the elements into a `std::vector` and sort them before printing**, with whatever order the output calls for:

```
#include <algorithm>
#include <vector>

std::vector<std::pair<std::string, int>> rows(counts.begin(), counts.end());
std::sort(rows.begin(), rows.end(), [](const auto& a, const auto& b) {
    if (a.second ≠ b.second) return a.second > b.second;   // most frequent j
    return a.first < b.first;                              // then alphabetic
});
for (const auto& [word, n] : rows) std::cout << word << ' ' << n << '\n';
```

The same applies to `bucket_count()`, `load_factor()` and hash values: they are real numbers about a real table, and they are not the same on another machine. Print sizes and sorted contents; never print those.

A struct as a key

Using your own type as a key needs two things the table cannot guess: a hash and an equality. Equality is `operator==`, which C++20 lets you default (Module 11, lesson 2). The hash is a callable that takes a `const Key&` and returns `std::size_t`, passed as the third template argument. Combining the members' hashes is the only design decision — `h1 ^ h2` alone is poor because `{1, 2}` and `{2, 1}` collide, so shift or mix one of them:

```cpp
#include <cstddef>
#include <functional>
#include <unordered_set>

struct Point {
    int x;
    int y;
    bool operator==(const Point&) const = default;   // C++20
};

struct PointHash {
    std::size_t operator()(const Point& p) const noexcept {
        const std::size_t hx = std::hash<int>{}(p.x);
        const std::size_t hy = std::hash<int>{}(p.y);
        return hx ^ (hy + 0x9e3779b9 + (hx << 6) + (hx >> 2));   // boost's ha
    }
};

std::unordered_set<Point, PointHash> visited;
visited.insert({0, 0});
bool again = !visited.insert({0, 0}).second;      // true: already there
```

The alternative is to specialise `std::hash<Point>` in namespace `std` (Module 11, lesson 4), after which `std::unordered_set<Point>` needs no third argument; the functor is simpler and keeps the choice local. The same recipe covers `std::pair`, which — surprisingly — has no standard hash: `std::unordered_map<std::pair<int, int>, int>` does not compile until you supply one, which is why grid problems often encode a cell as `r * cols + c` and hash the `int` instead.

Ordered or unordered?

Need	Choose
Membership, counting, lookup by key; order irrelevant	`std::unordered_map` / `unordered_set`
Output in key order; smallest/largest; "first key ≥ k"; ranges	`std::map` / `std::set`
A guaranteed worst case, or adversarial keys	`std::map` / `std::set`
A key type with `<` but no cheap hash	`std::map` / `std::set`
A few dozen elements, looked up rarely	a `std::vector` and `std::find` — no tree, no table
Keys are small dense integers	a `std::vector` indexed by the key

The unordered containers are usually the faster lookup, by a factor that grows with the size; the ordered ones are the faster answer when the question involves order. A sorted `std::vector` with `std::lower_bound` (Module 14, lesson 2) beats both for data that is built once and queried many times, because it has no nodes and no buckets — just contiguous memory.

Pitfalls

Mistake	What happens
Printing an unordered container directly	Unspecified order; copy to a vector and sort
Comparing output to another machine's bucket count or load factor	Implementation-specific; never print them
A custom key with a hash but no `operator==`	Does not compile — the table needs both
Holding an iterator across an `insert`	A rehash may have invalidated it; references and pointers survive
`std::unordered_map<std::pair<int, int>, V>`	No `std::hash` for `std::pair`; supply a functor or encode the pair
`um[key]` to test presence	Inserts, as in `std::map`; use `find` / `count` / `contains`
`max_load_factor(0.1)` to "make it faster"	Ten times the buckets, cache misses everywhere; the default is right

Key takeaways

A hash table stores each key in the bucket its hash selects; lookup, insertion and erasure are average O(1) and worst-case O(n) when keys collide.

The load factor is `size / buckets`; crossing `max_load_factor` (1.0) triggers a rehash that invalidates iterators but not references; `reserve(n)` avoids rehashes.

Iteration order is unspecified and unstable — copy to a `std::vector` and sort before printing; never print bucket counts or hash values.

A custom key needs `operator==` and a hash functor (or a `std::hash` specialisation); combine member hashes with a mixing step; `std::pair` has no standard hash.

Choose unordered for plain lookup, ordered for anything involving order or a guaranteed bound, and a sorted vector for build-once query-many data.

EXERCISE 1

Top words, hashed then sorted

Read an integer `k` on the first line, then free text until the end of input. Normalise each token as in the previous lesson (keep letters and digits, lower-case, skip empties) and count the words in a `std::unordered_map<std::string, int>`. Then print the `k` most frequent words as `<word> <count>`, most frequent first and **ties broken alphabetically** (fewer lines if there are fewer words), followed by `distinct=<number of distinct words>`.

An unordered container has no useful iteration order, so **copy its entries into a `std::vector` and sort that** with a comparator before printing. Never print the map directly.

Input: `k`, then text. **Output:** up to `k` lines, then the summary line.

```
3
the cat saw the other cat the end
```

prints

```
the 3
cat 2
end 1
distinct=5
```

INPUT 1

```
3
the cat saw the other cat the end
```

OUTPUT 1

```
the 3
cat 2
end 1
distinct=5
```

INPUT 2

```
10
b a b a c
```

OUTPUT 2

```
a 2
b 2
c 1
distinct=3
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
#include <unordered_map>
#include <utility>
#include <vector>

std::string normalise(const std::string& token) {
    std::string out;
    for (char c : token) {
        const unsigned char uc = static_cast<unsigned char>(c);
        if (std::isalnum(uc)) out += static_cast<char>(std::tolower(uc));
    }
    return out;
}

int main() {
    int k;
    std::cin >> k;
    std::unordered_map<std::string, int> freq;
    std::string token;
    while (std::cin >> token) {
        const std::string word = normalise(token);
        if (!word.empty()) ++freq[word];
    }
    std::vector<std::pair<std::string, int>> rows(freq.begin(), freq.end());
    std::sort(rows.begin(), rows.end(), [](const auto& a, const auto& b) {
        if (a.second != b.second) return a.second > b.second;
        return a.first < b.first;
    });
    const std::size_t shown = std::min(rows.size(), static_cast<std::size_t>(k));
    for (std::size_t i = 0; i < shown; ++i) std::cout << rows[i].first << ' ' << row
    std::cout << "distinct=" << freq.size() << '\n';
    return 0;
}
```

EXERCISE 2

Visited cells with a custom-hash std::unordered_set

A robot starts at `(0, 0)` and reads one line of moves: `U` (`y + 1`), `D` (`y - 1`), `L` (`x - 1`), `R` (`x + 1`). Track every cell it has stood on in an `std::unordered_set<Point, PointHash>`, where `Point` is a struct with `int x, y` and a defaulted `operator==`, and `PointHash` is a functor that combines the two members' hashes. The start cell counts as visited. Print `visited=<distinct cells>`, then

`first revisit=<step>` — the 1-based number of the first move that lands on a cell already visited — or `first revisit=none`, then `end=(<x>,<y>)`.

The hash values themselves must never appear in the output; `insert(...).second` is all the program needs.

Input: one line of `U` / `D` / `L` / `R` characters (possibly empty). **Output:** three lines.

Example: `RRUULLDD` →

```
visited=8
first revisit=8
end=(0,0)
```

INPUT 1

```
RRUULLDD
```

OUTPUT 1

```
visited=8
first revisit=8
end=(0,0)
```

INPUT 2

```
RLRL
```

OUTPUT 2

```
visited=2
first revisit=2
end=(0,0)
```

REFERENCE SOLUTION

```cpp
#include <cstddef>
#include <functional>
#include <iostream>
#include <string>
#include <unordered_set>

struct Point {
    int x;
    int y;
    bool operator==(const Point&) const = default;
};

struct PointHash {
    std::size_t operator()(const Point& p) const noexcept {
        const std::size_t hx = std::hash<int>{}(p.x);
        const std::size_t hy = std::hash<int>{}(p.y);
        return hx ^ (hy + 0x9e3779b9 + (hx << 6) + (hx >> 2));
    }
};

int main() {
    std::string moves;
    std::getline(std::cin, moves);
    std::unordered_set<Point, PointHash> visited;
    Point at{0, 0};
    visited.insert(at);
    int firstRevisit = 0;
    for (std::size_t i = 0; i < moves.size(); ++i) {
        switch (moves[i]) {
            case 'U': ++at.y; break;
            case 'D': --at.y; break;
            case 'L': --at.x; break;
            case 'R': ++at.x; break;
            default: continue;
        }
        const bool fresh = visited.insert(at).second;
        if (!fresh && firstRevisit == 0) firstRevisit = static_cast<int>(i) + 1;
    }
    std::cout << "visited=" << visited.size() << '\n';
    if (firstRevisit == 0) std::cout << "first revisit=none\n";
    else std::cout << "first revisit=" << firstRevisit << '\n';
    std::cout << "end=(" << at.x << ',' << at.y << ")\n";
    return 0;
}
```

QUIZ

1. What does this print?

```
std::unordered_set<int> s{1, 2, 3};
for (int x : s) std::cout << x;
```

A. `123`

B. `321`

C. Some permutation of 1, 2 and 3 — the order is unspecified

D. compile error

2. What is a hash table's load factor?

A. The number of buckets

B. `size() / bucket_count()`

C. The fraction of buckets that are empty

D. The length of the longest chain

3. What does a `struct Point` need before it can be a key in `std::unordered_set<Point, PointHash>`?

A. Only `operator<`

B. A hash functor and `operator==`

C. A hash functor and `operator<`

D. Nothing — the compiler derives a hash from the members

4. An insertion into an `std::unordered_map` triggers a rehash. Which stays valid?

A. Every iterator

B. Pointers and references to elements, but no iterators

C. Nothing

D. Iterators, but no references

5. What is the worst-case complexity of `find` on an `std::unordered_map` with n elements?

A. O(1)

B. O(log n)

C. O(n)

D. O(n log n)

6. Which task calls for `std::map` rather than `std::unordered_map`?

A. Counting how often each word occurs

B. Testing whether an id has been seen

C. Finding the smallest key that is not less than k

D. Caching results by an integer key

7. `std::unordered_map<std::pair<int, int>, int> m;` — what happens?

A. It works: pairs hash by combining their members

B. Compile error: there is no `std::hash` for `std::pair`

C. It compiles but every pair lands in one bucket

D. It works only if both `int`s are non-negative

ANSWER KEY · THE UNORDERED CONTAINERS — HASHING, BUCKETS AND CUSTOM KEYS

1. C Iteration follows the buckets, and the standard says nothing about which bucket holds which key. Any fixed answer is an assumption about one library version.

2. B The load factor is elements per bucket on average; when it would exceed `max_load_factor()` (1.0 by default) the table rehashes into more buckets.

3. B The table hashes to find the bucket and then compares with `==` inside it. `operator<` is what the ordered containers need; C++ never synthesises a hash.

4. B A rehash re-links the existing nodes into new buckets without moving the elements, so references and pointers survive while every iterator is invalidated.

5. C If every key hashes into the same bucket, `find` walks one chain of n elements. The O(1) figure is the average with a good hash.

6. C That is `lower_bound`, which only an ordered container can answer. The other three are plain lookups where the unordered container is faster.

7. B The standard provides `std::hash` for the arithmetic types, pointers, strings and a few library types, but not for `std::pair` or `std::tuple`. Supply a functor or encode the pair as one integer.

LESSON 6 · 14 MIN

Iterators and invalidation — the contract behind every loop

An iterator is a generalised pointer: something you can dereference to reach an element and increment to reach the next one. Every container hands them out through `begin()` and `end()`, every algorithm in `<algorithm>` takes a pair of them, and the range-based `for` of Module 3 is sugar over them. What makes them more than pointers is that each container promises a *category* — how far and in which directions its iterators can move — and an *invalidation rule* — which operations on the container leave existing iterators pointing at garbage. This lesson covers the categories and what they cost, the helper functions `std::next`, `std::prev`, `std::distance` and `std::advance`, `const_iterator` and reverse iterators, the invalidation table, the one correct way to erase while iterating, and iterator ranges as the interface algorithms share.

Categories

Category	Can do	Containers
input / output	read or write once, move forward	streams (`std::istream_iterator`)
forward	read/write, `++` , multi-pass	`std::forward_list` , the unordered containers
bidirectional	forward plus `--`	`std::list` , `std::set` , `std::map` and their `multi` forms
random access	bidirectional plus `it + n` , `it[n]` , `it - it2` , `<`	`std::deque`
contiguous (C++17)	random access, and elements are adjacent in memory	`std::vector` , `std::array` , `std::string` , built-in arrays

Each category includes the ones above it. The category is what decides the cost of an operation: `it + 5` is one addition on a vector and does not compile on a list; `std::sort` needs random access, which is why `std::list` has its own `sort()` member; `std::lower_bound` on a `std::set` compiles but walks linearly (lesson 4). C++20 spells these

categories as concepts — `std::random_access_iterator` and friends — and Module 14 uses them.

---

begin, end, const and reverse

`begin()` points at the first element and `end()` one past the last; the range is half-open, `[begin, end)`, so an empty container has `begin() == end()` and dereferencing `end()` is undefined behaviour. The iterator type is `Container::iterator` — spelt `auto` in practice — and its `const_iterator` cousin refuses writes. `cbegin()` / `cend()` always give the const form, and calling `begin()` on a `const` container (or through a `const&` parameter) gives it too. `rbegin()` / `rend()` walk backwards: `*rbegin()` is the last element, and `rit.base()` converts back to a forward iterator that points one past `*rit`.

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main() {
    std::vector<int> v{10, 20, 30, 40};
    for (auto it = v.cbegin(); it ≠ v.cend(); ++it) std::cout << *it << ' ';
    for (auto rit = v.rbegin(); rit ≠ v.rend(); ++rit) std::cout << *rit <<
    auto third = std::next(v.begin(), 2);                 // → 30, without cha
    auto last = std::prev(v.end());                       // → 40
    std::cout << '\n' << *third << ' ' << *last << ' '
              << std::distance(v.begin(), last) << '\n';  // 30 40 3
    std::advance(third, -1);                              // third now → 20 (n
    std::cout << *third << ' ' << *std::max_element(v.begin(), v.end()) << '\n
    return 0;
}
```

`std::next(it, n)` and `std::prev(it, n)` return a moved copy; `std::advance(it, n)` moves the iterator you pass; `std::distance(a, b)` counts the steps from `a` to `b`. On a random-access iterator all four are O(1) arithmetic; on a bidirectional or forward iterator they *step*, so `std::distance(s.begin(), s.find(k))` on a `std::set` is O(n) and `std::next(l.begin(), 1000)` on a list is a thousand hops. Prefer them to hand-written `it + 2` because they compile on every category and say what they mean.

The invalidation table

An operation **invalidates** an iterator when the element it pointed to has moved, been destroyed, or — for `end()` — is no longer where the end is. Using an invalidated iterator is undefined behaviour: it usually reads stale memory and sometimes appears to work, which is what makes the table worth learning rather than discovering.

Container	`push_back` / `insert`	`erase`
`std::vector`, `std::string`	if it reallocates: everything; otherwise every iterator at or after the insertion point (and `end()`)	every iterator at or after the erased element, and `end()`
`std::deque`	insert at an end: all iterators, no references; insert in the middle: everything	erase at an end: only the erased element (and `end()` for the back); in the middle: everything
`std::list`, `std::forward_list`	nothing	only the erased element
`std::set`, `std::map` (and `multi`)	nothing	only the erased element
`std::unordered_*`	if it rehashes: every iterator, no references; otherwise nothing	only the erased element

Two rules cover most of it. Node-based containers (list, set, map, unordered) keep an element's iterator valid until *that element* is erased — a rehash being the one exception for the unordered ones. Contiguous containers (vector, string, deque in the middle) invalidate everything from the point of change onwards, and a growing vector invalidates everything full stop. The practical habit: never hold an iterator or reference into a vector across a call that can change its size; hold an index, or `reserve` first.

Erasing while iterating

The classic bug:

```
for (auto it = v.begin(); it ≠ v.end(); ++it) {
    if (*it % 2 == 0) v.erase(it);        // it now points past the removed sl
}                                          // at the last element it steps pas
```

`erase` on every container returns an iterator to the element *after* the one removed, and that return value is the whole idiom:

```
for (auto it = v.begin(); it ≠ v.end(); ) {
    if (*it % 2 == 0) it = v.erase(it);   // erase advances for us
    else ++it;                             // only advance when we did not eras
}
```

The same loop is correct on a `std::map` (`it→second`), a `std::list`, a `std::set` and an unordered container, which is why it is the one to memorise. Two variants exist. `m.erase(it++)` — post-increment produces the old iterator for `erase` after stepping past it — works on node-based containers only, because on a vector the stepped iterator is itself invalidated by the erase; do not write it for a container you might later change. And since C++20, `std::erase_if(c, pred)` does the whole job for any standard container in one call (lesson 2); use it whenever the decision is a pure predicate on the element, and keep the explicit loop for when the body needs to do something else with each erased element, such as print it or move it somewhere.

The mirror image — inserting into a container while iterating it — has no safe idiom on a vector at all (each insert can reallocate). Collect the additions in a second container and append after the loop, or iterate by index up to a size captured before the loop.

Iterator ranges as the algorithm interface

Every algorithm in `<algorithm>` and `<numeric>` takes `[first, last)` rather than a container, and that decision is what lets one `std::sort` serve a vector, a deque, a string and a built-in array, and let it work on *part* of a container:

```cpp
#include <algorithm>
#include <numeric>

std::vector<int> v{5, 3, 9, 1, 7};
std::sort(v.begin(), v.begin() + 3);                          // 3 5 9 1 7 — onl
auto nine = std::find(v.begin(), v.end(), 9);                 // an iterator, or
long long sum = std::accumulate(v.begin(), v.end(), 0LL);     // 25 — 0LL so the
v.erase(std::unique(v.begin(), v.end()), v.end());            // adjacent duplic
std::ranges::sort(v);                                         // C++20: the cont
```

The range convention, the categories and the invalidation rules are the three things Module 14 assumes; the exercises there are written against iterator pairs and, in the last lesson, against C++20 ranges, which are the same idea with the pair packaged as one object.

Pitfalls

Mistake	What happens
`*v.end()` or `--v.begin()`	Undefined behaviour; the range is half-open
`v.erase(it)` inside a `for (…; ++it)`	Skips the next element; runs off the end on the last one
`it = v.erase(it)` *and* `++it` in the same iteration	Skips one element after every erase
`m.erase(it++)` on a vector	The incremented iterator was invalidated by the erase
`auto it = v.begin(); v.push_back(x); *it`	Dangling after a reallocation
`it + 2` on a `std::list` or `std::set` iterator	Does not compile; `std::next(it, 2)`
Comparing iterators from two different containers	Meaningless; undefined behaviour
`std::distance(s.begin(), it)` on a large set in a loop	O(n) per call; count as you go instead

Key takeaways

Iterators come in categories — forward (unordered), bidirectional (list, set, map), random access (deque), contiguous (vector, string, array) — and the category fixes what `+ n` and the algorithms may do.

`[begin, end)` is half-open; `cbegin` / `cend` are read-only; `rbegin` / `rend` walk backwards; `std::next` / `prev` / `distance` / `advance` are O(1) only on random access.

Node-based containers keep an iterator valid until its element is erased (a rehash aside); a vector invalidates everything from the change point and everything on reallocation.

Erase while iterating with `it = c.erase(it)` and advance only in the `else`; use `std::erase_if` when the decision is a pure predicate.

Algorithms take `[first, last)` so one implementation serves every container and any sub-range.

EXERCISE 1

Close the small accounts

Read `n`, then `n` lines of `<name> <balance>` into a `std::map<std::string, int>` (a repeated name keeps the last balance), then a threshold `t`. Walk the map **once with an explicit iterator**: for every account whose balance is below `t`, print `closed <name> <balance>` and erase it with `it = accounts.erase(it)`; for every other account print `kept <name> <balance>` and advance. Finish with `remaining=<size>`.

The point of the exercise is the erase-while-iterating shape — advance only in the branch that did not erase. Do not use `std::erase_if` here.

Input: `n`, `n` lines, then `t`. **Output:** one line per account in key order, then the summary line.

```
4
bob 50
ada 120
cy 5
dee 75
60
```

prints

```
kept ada 120
closed bob 50
closed cy 5
kept dee 75
remaining=2
```

INPUT 1

```
4
bob 50
ada 120
cy 5
dee 75
60
```

OUTPUT 1

```
kept ada 120
closed bob 50
closed cy 5
kept dee 75
remaining=2
```

INPUT 2

```
3
x 1
y 2
z 3
10
```

OUTPUT 2

```
closed x 1
closed y 2
closed z 3
remaining=0
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    int n;
    std::cin >> n;
    std::map<std::string, int> accounts;
    for (int i = 0; i < n; ++i) {
        std::string name;
        int balance;
        std::cin >> name >> balance;
        accounts[name] = balance;
    }
    int threshold;
    std::cin >> threshold;
    for (auto it = accounts.begin(); it != accounts.end(); ) {
        if (it->second < threshold) {
            std::cout << "closed " << it->first << ' ' << it->second << '\n';
            it = accounts.erase(it);
        } else {
            std::cout << "kept " << it->first << ' ' << it->second << '\n';
            ++it;
        }
    }
    std::cout << "remaining=" << accounts.size() << '\n';
    return 0;
}
```

EXERCISE 2

Playlist cursor on a std::list

Read `n` (at least 1), then `n` track names (one word each) into a `std::list<std::string>`, and keep a **cursor** — a list iterator — that starts on the first track. Then apply commands until the end of input:

- `next` — move the cursor forward, wrapping from the last track to the first;
- `prev` — move it backward, wrapping from the first to the last;
- `jump k` — move forward `k` steps with wrapping (`k ≥ 0`; reduce it modulo the size first);
- `insert <name>` — insert a track **before** the cursor; the cursor keeps pointing at the same track;
- `play` — print `playing <track> at <index>`, the index being `std::distance(playlist.begin(), cursor)`;
- `reverse` — print `reverse:` followed by every track from last to first, via `rbegin()` / `rend()`.

Use `std::next`, `std::prev` and `std::distance` rather than arithmetic — a list iterator has no `+`.

Input: `n`, the names, then commands. **Output:** one line per `play` or `reverse`.

```
3
alpha beta gamma
play
next
play
next
next
play
jump 4
play
reverse
```

prints

```
playing alpha at 0
playing beta at 1
playing alpha at 0
playing beta at 1
reverse: gamma beta alpha
```

INPUT 1

```
3
alpha beta gamma
play
next
play
next
next
play
prev
prev
play
jump 4
play
reverse
```

OUTPUT 1

```
playing alpha at 0
playing beta at 1
playing alpha at 0
playing beta at 1
playing gamma at 2
reverse: gamma beta alpha
```

INPUT 2

```
2
a b
insert z
play
prev
play
reverse
```

OUTPUT 2

```
playing a at 1
playing z at 0
reverse: b a z
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <iterator>
#include <list>
#include <string>

int main() {
    int n;
    std::cin >> n;
    std::list<std::string> playlist;
    for (int i = 0; i < n; ++i) {
        std::string track;
        std::cin >> track;
        playlist.push_back(track);
    }
    auto cursor = playlist.begin();
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "next") {
            ++cursor;
            if (cursor == playlist.end()) cursor = playlist.begin();
        } else if (cmd == "prev") {
            if (cursor == playlist.begin()) cursor = playlist.end();
            --cursor;
        } else if (cmd == "jump") {
            long long k;
            std::cin >> k;
            const long long size = static_cast<long long>(playlist.size());
            const long long steps = k % size;
            const long long ahead = std::distance(cursor, playlist.end());
            if (steps < ahead) cursor = std::next(cursor, steps);
            else cursor = std::next(playlist.begin(), steps - ahead);
        } else if (cmd == "insert") {
            std::string track;
            std::cin >> track;
            playlist.insert(cursor, track);   // before the cursor; list iterators s
        } else if (cmd == "play") {
            std::cout << "playing " << *cursor << " at " << std::distance(playlist.b
        } else if (cmd == "reverse") {
            std::cout << "reverse:";
            for (auto rit = playlist.rbegin(); rit != playlist.rend(); ++rit) std::c
            std::cout << '\n';
        }
    }
    return 0;
}
```

QUIZ

1. What iterator category does `std::list` provide, and what does `std::vector` provide?

A. Both random access

B. Bidirectional and random access (contiguous)

C. Forward and bidirectional

D. Random access and contiguous

2. `std::distance(s.begin(), s.find(7))` on a `std::set<int>` with a million elements costs…

A. O(1)

B. O(log n)

C. O(n)

D. It does not compile

3. What does this print?

```
std::vector<int> v{1, 2, 3, 4};
for (auto it = v.begin(); it ≠ v.end(); ) {
    if (*it % 2 == 0) it = v.erase(it);
    else ++it;
}
std::cout << v.size();
```

A. 4

B. 2

C. 3

D. undefined behaviour

4. `for (auto it = v.begin(); it ≠ v.end(); ++it) if (*it == 2) v.erase(it);` on a vector — what is wrong?

A. Nothing; `erase` handles the iterator

B. `it` is invalidated by `erase`; the `++it` then uses it — undefined behaviour

C. It compiles but removes the wrong element

D. `erase` needs a range, not a single iterator

5. `c.erase(it++);` is a correct erase-while-iterating step for which containers?

A. All standard containers

B. Only `std::vector` and `std::deque`

C. Node-based containers such as `std::map`, `std::set` and `std::list`

D. None — post-increment is always wrong here

6. What is the difference between `std::next(it, 3)` and `std::advance(it, 3)`?

A. No difference

B. `std::next` returns a moved copy and leaves `it` alone; `std::advance` moves `it` itself and returns nothing

C. `std::next` works only on random-access iterators

D. `std::advance` returns the new iterator, `std::next` does not

7. Which container's `insert` never invalidates existing iterators?

A. `std::vector`

B. `std::deque`

C. `std::unordered_map`

D. `std::list`

ANSWER KEY · ITERATORS AND INVALIDATION — THE CONTRACT BEHIND EVERY LOOP

1. B List nodes link both ways but cannot be indexed, so the iterator is bidirectional. A vector's iterator supports `+ n` and is contiguous, the strongest category.

2. C `find` is O(log n), but `std::distance` on a bidirectional iterator steps one element at a time, so the call is linear in the position of 7.

3. B This is the correct idiom: `erase` returns the iterator to the next element and the loop advances only when it did not erase. Both even numbers go.

4. B After `erase(it)` every iterator from `it` onwards is invalid, including `it` itself; incrementing it is undefined behaviour and, in practice, skips an element or runs past `end()`.

5. C The post-increment steps `it` before the erase and hands the old value to `erase`; that is fine when only the erased node is invalidated. On a vector the stepped iterator is invalidated too, so use `it = c.erase(it)` everywhere.

6. B `std::next` / `std::prev` are pure functions; `std::advance` mutates its argument. Both work on every category, stepping when the iterator cannot jump.

7. D List (and map/set) nodes never move, so other iterators are untouched. A vector may reallocate, a deque invalidates iterators even on an end insert, and an unordered map may rehash.

CHECKPOINT · 25 MIN

Checkpoint — STL containers

This checkpoint covers the whole module: the container families and the complexity table, `std::vector`'s size–capacity model and the erase–remove idiom, `std::deque`, `std::list` and the adaptors, `std::map` / `std::set` with `lower_bound`, the unordered containers and the sort-before-printing rule, and iterators with the invalidation table.

How it works. Thirteen questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- Which operation is O(1) on a `std::list` but O(n) on a `std::vector`, and what must you already hold for it to be O(1)?
- What does `push_back` do to every iterator into a vector when `size() == capacity()`?
- Why does `std::stack::pop()` return `void`?
- When does a `std::priority_queue` comparator return `true`?
- What does `m[key]` do when `key` is absent, and which calls look up without inserting?
- Why must an `std::unordered_map` be sorted before it is printed?
- What is the one correct shape of an erase-while-iterating loop?

The three programs are a breadth-first search over a grid with `std::queue`, an LRU cache built from a `std::list` and an `std::unordered_map` of iterators, and a leaderboard kept in a `std::map` and a `std::set` with a custom comparator.

EXERCISE 1

Shortest path on a grid with std::queue

Read `R C`, then `R` rows of `C` characters: `.` open, `#` wall, `S` the start and `E` the exit (each appears exactly once). Print the length of the shortest path from `S` to `E` moving up, down, left or right through open cells, or `-1` when the exit cannot be reached.

Use breadth-first search with a `std::queue` of cells and a `std::vector<std::vector<int>>` of distances initialised to `-1`, which doubles as the visited set — mark a cell when it is **pushed**.

Input: `R C`, then `R` rows. **Output:** one integer.

```
3 4
S..#
.#..
...E
```

prints `5`.

INPUT 1

```
3 4
S..#
.#..
...E
```

OUTPUT 1

```
5
```

INPUT 2

```
2 2
S#
#E
```

OUTPUT 2

```
-1
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <queue>
#include <string>
#include <utility>
#include <vector>

int main() {
    int rows, cols;
    std::cin >> rows >> cols;
    std::vector<std::string> grid(rows);
    std::pair<int, int> start{0, 0};
    std::pair<int, int> goal{0, 0};
    for (int r = 0; r < rows; ++r) {
        std::cin >> grid[r];
        for (int c = 0; c < cols; ++c) {
            if (grid[r][c] == 'S') start = {r, c};
            if (grid[r][c] == 'E') goal = {r, c};
        }
    }
    std::vector<std::vector<int>> dist(rows, std::vector<int>(cols, -1));
    std::queue<std::pair<int, int>> frontier;
    dist[start.first][start.second] = 0;
    frontier.push(start);
    const int dr[] = {-1, 1, 0, 0};
    const int dc[] = {0, 0, -1, 1};
    while (!frontier.empty()) {
        const auto [r, c] = frontier.front();
        frontier.pop();
        if (r == goal.first && c == goal.second) break;
        for (int k = 0; k < 4; ++k) {
            const int nr = r + dr[k];
            const int nc = c + dc[k];
            if (nr < 0 || nr >= rows || nc < 0 || nc >= cols) continue;
            if (grid[nr][nc] == '#' || dist[nr][nc] != -1) continue;
            dist[nr][nc] = dist[r][c] + 1;
            frontier.push({nr, nc});
        }
    }
    std::cout << dist[goal.first][goal.second] << '\n';
    return 0;
}
```

EXERCISE 2

LRU cache from a std::list and an std::unordered_map

Implement a least-recently-used cache. Read its capacity (at least 1), then commands until the end of input: `get <key>` prints `hit <value>` and marks the key most recently used, or `miss`; `put <key> <value>` inserts or updates the key and marks it most recently used, and when that pushes the size past the capacity it evicts the least recently used key and prints `evict <key>`. After the last command print `cache:` followed by every `<key>=<value>` from most to least recently used.

Keys and values are integers. The recency order lives in a `std::list<std::pair<int, int>>` (most recent at the front); a `std::unordered_map<int, std::list<…>::iterator>` finds a key's node in O(1). Moving a node to the front is one `splice`, and list iterators stay valid across it — which is why the map may hold them. The final line comes from the **list**, never from the map.

Input: the capacity, then commands. **Output:** one line per `get` and per eviction, then the `cache:` line.

```
2
put 1 10
put 2 20
get 1
put 3 30
get 2
get 3
put 1 11
```

prints

```
hit 10
evict 2
miss
hit 30
cache: 1=11 3=30
```

INPUT 1

```
2
put 1 10
put 2 20
get 1
put 3 30
get 2
get 3
put 1 11
```

OUTPUT 1

```
hit 10
evict 2
miss
hit 30
cache: 1=11 3=30
```

INPUT 2

```
1
get 5
put 5 1
put 6 2
get 5
get 6
```

OUTPUT 2

```
miss
evict 5
miss
hit 2
cache: 6=2
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <list>
#include <string>
#include <unordered_map>
#include <utility>

int main() {
    std::size_t capacity;
    std::cin >> capacity;
    std::list<std::pair<int, int>> order;                                   // most
    std::unordered_map<int, std::list<std::pair<int, int>>::iterator> where;
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "get") {
            int key;
            std::cin >> key;
            const auto it = where.find(key);
            if (it == where.end()) {
                std::cout << "miss\n";
            } else {
                order.splice(order.begin(), order, it->second);   // to the front; t
                std::cout << "hit " << it->second->second << '\n';
            }
        } else if (cmd == "put") {
            int key, value;
            std::cin >> key >> value;
            const auto it = where.find(key);
            if (it != where.end()) {
                it->second->second = value;
                order.splice(order.begin(), order, it->second);
            } else {
                order.push_front({key, value});
                where[key] = order.begin();
                if (order.size() > capacity) {
                    std::cout << "evict " << order.back().first << '\n';
                    where.erase(order.back().first);
                    order.pop_back();
                }
            }
        }
    }
    std::cout << "cache:";
    for (const auto& [key, value] : order) std::cout << ' ' << key << '=' << value;
    std::cout << '\n';
    return 0;
}
```

EXERCISE 3

Leaderboard on a std::set with a custom order

Keep a leaderboard. Commands arrive one per line until the end of input:

- `set <name> <score>` — add the player or replace their score;
- `remove <name>` — remove the player, or print `remove <name> unknown`;
- `rank <name>` — print `rank <name>=<r>`, the player's 1-based position, or `rank <name> unknown`;
- `top <k>` — print `top:` followed by the first `k` players as `<name>(<score>)`, space-separated (fewer if there are fewer players);
- `above <s>` — print `above <s>: <count>`, the number of players whose score is **strictly greater** than `s`.

The order is **highest score first, ties by name ascending**. Use two containers: a `std::map<std::string, int>` from name to score, and a `std::set<Entry, Comparator>` of `{score, name}` structs ordered by a comparator struct. Updating a score means erasing the old entry from the set and inserting the new one. `above` is `std::distance(board.begin(), board.lower_bound({s, ""}))` — with this ordering the key `{s, ""}` sorts before every real entry that has score `s`.

Input: commands until end of input. **Output:** one line per `rank`, `top`, `above` or failed `remove`.

```
set ada 50
set bob 70
set cy 50
top 2
rank cy
set ada 90
rank ada
above 50
remove bob
top 5
rank bob
```

prints

```
top: bob(70) ada(50)
rank cy=3
rank ada=1
above 50: 2
top: ada(90) cy(50)
rank bob=unknown
```

INPUT 1

```
set ada 50
set bob 70
set cy 50
top 2
rank cy
set ada 90
rank ada
above 50
remove bob
top 5
rank bob
```

OUTPUT 1

```
top: bob(70) ada(50)
rank cy=3
rank ada=1
above 50: 2
top: ada(90) cy(50)
rank bob=unknown
```

INPUT 2

```
top 3
above 0
set x 1
remove y
rank x
top 1
```

OUTPUT 2

```
top:
above 0: 0
remove y=unknown
rank x=1
top: x(1)
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <iterator>
#include <map>
#include <set>
#include <string>

struct Entry {
    int score;
    std::string name;
};

struct HigherScoreFirst {
    bool operator()(const Entry& a, const Entry& b) const {
        if (a.score != b.score) return a.score > b.score;
        return a.name < b.name;
    }
};

int main() {
    std::map<std::string, int> scores;
    std::set<Entry, HigherScoreFirst> board;
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "set") {
            std::string name;
            int score;
            std::cin >> name >> score;
            if (auto it = scores.find(name); it != scores.end()) {
                board.erase(Entry{it->second, name});
                it->second = score;
            } else {
                scores.emplace(name, score);
            }
            board.insert(Entry{score, name});
        } else if (cmd == "remove") {
            std::string name;
            std::cin >> name;
            const auto it = scores.find(name);
            if (it == scores.end()) {
                std::cout << "remove " << name << "=unknown\n";
            } else {
                board.erase(Entry{it->second, name});
                scores.erase(it);
            }
        } else if (cmd == "rank") {
            std::string name;
            std::cin >> name;
            const auto it = scores.find(name);
            if (it == scores.end()) {
```

```
                std::cout << "rank " << name << "=unknown\n";
            } else {
                const auto pos = board.find(Entry{it->second, name});
                std::cout << "rank " << name << '=' << std::distance(board.begin(),
            }
        } else if (cmd == "top") {
            int k;
            std::cin >> k;
            std::cout << "top:";
            int shown = 0;
            for (auto it = board.begin(); it != board.end() && shown < k; ++it, ++sh
                std::cout << ' ' << it->name << '(' << it->score << ')';
            }
            std::cout << '\n';
        } else if (cmd == "above") {
            int s;
            std::cin >> s;
            const auto firstNotAbove = board.lower_bound(Entry{s, ""});
            std::cout << "above " << s << ": " << std::distance(board.begin(), first
        }
    }
    return 0;
}
```

CHECKPOINT QUESTIONS

1. What is the complexity of `push_back` on a `std::vector`?

A. Always O(1)

B. Amortised O(1): usually constant, occasionally O(n) when it reallocates

C. O(log n)

D. O(n) every time

2. Which container keeps its elements sorted by key at all times?

A. `std::unordered_map`

B. `std::map`

C. `std::vector`

D. `std::deque`

3. What does this do?

```
std::vector<int> v{1, 2, 3};
int& r = v[0];
v.reserve(100);
std::cout << r;
```

A. Prints `1`

B. Undefined behaviour — `reserve` reallocated and `r` dangles

C. Prints `0`

D. Compile error

4. Which adaptor wraps a `std::vector` by default?

A. `std::stack`

B. `std::queue`

C. `std::priority_queue`

D. All three

5. What does this print?

```
std::priority_queue<int, std::vector<int>, std::greater<int>> pq;
for (int x : {5, 2, 8}) pq.push(x);
std::cout << pq.top();
```

A. 8

B. 5

C. 2

D. compile error

6. What does this print?

```
std::map<int, std::string> m;
m[3] = "c";
m[1] = "a";
m[2] = "b";
for (const auto& [k, v] : m) std::cout << v;
```

A. `cab`

B. `abc`

C. `cba`

D. unspecified order

7. `m.insert({k, v}).second` is `false`. What does that tell you?

A. The insertion failed because the map is full

B. The key already existed and the stored value was not changed

C. The value was replaced

D. `k` compared equal to `end()`

8. `s.lower_bound(k)` on a `std::set<int> s` returns…

A. An iterator to the largest element less than `k`

B. An iterator to the first element not less than `k`, or `end()` if there is none

C. `true` if `k` is present

D. An iterator to the first element greater than `k`

9. Before printing the contents of an `std::unordered_map`, a judged program should…

A. Call `rehash(0)` so the order becomes sorted

B. Copy the entries into a `std::vector` and sort them

C. Use `std::map` for printing only — the order is the same

D. Nothing: iteration order is insertion order

10. Which two things does a custom struct need to be a key in an unordered container?

A. `operator<` and `operator==`

B. A hash functor (or `std::hash` specialisation) and `operator==`

C. A hash functor and `operator<`

D. A copy constructor and a destructor

11. What does this print?

```
std::list<int> l{1, 2, 3, 4, 5};
for (auto it = l.begin(); it != l.end(); ) {
    if (*it % 2 == 1) it = l.erase(it);
    else ++it;
}
for (int x : l) std::cout << x;
```

A. `135`

B. `24`

C. `12345`

D. undefined behaviour

12. `std::sort(l.begin(), l.end());` where `l` is a `std::list<int>` — what happens?

A. Sorts the list

B. Compile error: `std::sort` needs random-access iterators; use `l.sort()`

C. Sorts it in O(n²)

D. Undefined behaviour

13. Which of these invalidates *only* iterators to the erased element?

A. `v.erase(it)` on a `std::vector`

B. `d.erase(it)` in the middle of a `std::deque`

C. `m.erase(it)` on a `std::map`

D. `s.erase(it)` on a `std::string`

14. What is the difference between `std::remove_if` and `std::erase_if`?

A. None; `erase_if` is the new name

B. `remove_if` only moves the kept elements forward and returns the new end; `erase_if` (C++20) actually erases and returns the count removed

C. `remove_if` works on any container, `erase_if` only on vectors

D. `erase_if` is O(n²), `remove_if` is O(n)

ANSWER KEY · CHECKPOINT — STL CONTAINERS

1. B Geometric growth means the O(n) reallocation happens rarely enough that n pushes cost O(n) in total.

2. B A map is a balanced tree ordered by its comparator. The unordered map is a hash table with no order; a vector and a deque keep insertion order.

3. B `reserve(100)` on a vector of capacity 3 allocates a new block and moves the elements; every reference into the old block, including `r`, is invalidated. That it might print 1 anyway is what makes the bug dangerous.

4. C A heap needs random access, so `std::priority_queue` defaults to `std::vector`; `std::stack` and `std::queue` default to `std::deque`.

5. C `std::greater` turns the heap into a min-heap: the smallest element, 2, is on top.

6. B A map iterates in key order regardless of insertion order: 1, 2, 3 → `abc`. Unspecified order is the unordered map.

7. B `insert` never overwrites; the `bool` is `false` precisely when the key was already present. The iterator in the pair points at the existing element.

8. B 'Not less than' is ≥. The first element strictly greater is `upper_bound`; the largest element below `k` is `std::prev(lower_bound(k))` when that is not `begin()`.

9. B The iteration order is unspecified and differs between library versions and table sizes. Sorting a copy is the only way to make the output deterministic.

10. B Hash to pick the bucket, equality to find the element inside it. `operator<` is what the ordered containers need instead.

11. B The idiom erases the odd elements and advances past the even ones; 2 and 4 remain.

12. B A list iterator is bidirectional, and `std::sort` requires random access, so the call does not compile. `std::list::sort()` is the member that relinks nodes.

13. C Node-based containers unlink one node and touch nothing else. A vector or string shifts everything after the gap; a deque erase in the middle invalidates all of its iterators.

14. B An algorithm cannot change a container's size — it only sees iterators — which is why the erase–remove idiom needs the `erase` call. `std::erase_if` takes the container itself and does both steps.
