Module 14 of 20

Algorithms and lambdas

The half-open range and predicate interface of <algorithm> and <numeric>, sorting with comparators and binary search, transform/accumulate/partition, lambdas as closure objects with std::function, and C++20 ranges with lazy views.

Module 3 catalogued the loops; this module retires most of them. `<algorithm>` and `<numeric>` hold over a hundred named functions — find, count, sort, transform, accumulate, partition — that take a half-open iterator range and, usually, a callable, and do what a hand loop would do with the edge cases already right and the library free to optimise. The lambda is the callable that made them usable: a function written where it is needed, carrying the values it needs. C++20 completes the picture with ranges, which take the container instead of two iterators, and views, which chain into lazy pipelines with no intermediate copies.

The five lessons move from interface to composition. The algorithm library fixes the conventions — `[first, last)`, predicates, iterators as results — through `find`, `count`, `minmax_element`, `fill` and `iota`. Sorting and searching covers comparators as strict weak orderings, tie-breaks, `std::stable_sort`, `partial_sort`, `nth_element` and `lower_bound` / `upper_bound` on sorted ranges. Transforming and reducing covers `transform`, `accumulate` and the `0LL` rule, `reduce`, the yes/no reductions, `copy_if`, the erase–remove idiom, `partition` and prefix sums. Lambdas in depth opens the closure object: capture defaults, init-captures, `mutable`, `this`, generic and immediately invoked lambdas, `std::function` and its cost, recursion, and comparators stored in containers. Ranges and views closes with projections, `filter` / `transform` / `take` / `drop` / `iota` / `reverse`, laziness, and how a pipeline is materialised on a C++20 runtime.

The exercises are whole programs, each built on its lesson's algorithm: `find_if` and `count_if` over invoice records, `minmax_element` with positions, a leaderboard sorted by score with a name tie-break, range counting with `lower_bound` / `upper_bound`, line totals through `transform` and `accumulate` with `0LL`, a ticket queue cleaned with `remove_if` and split with `partition`, a sequence counter that owns its state through an init-capture, a stack machine driven by a `std::function` dispatch table, a `filter | transform | take` pipeline copied into a vector, and a `std::ranges::sort` with a tuple projection. The checkpoint adds a `stable_sort` task list, a median by `nth_element`, and a nested-list walker written as a recursive lambda.

CONTENTS

1	The algorithm library — ranges, predicates and the first dozen calls	14 min
2	Sorting and searching — comparators, stability and binary search	15 min
3	Transforming and reducing — transform, accumulate and the erase–remove idiom	15 min
4	Lambdas in depth — captures, closures, std::function and recursion	15 min
5	Ranges and views — C++20 pipelines	15 min
✓	Checkpoint — Algorithms and lambdas	25 min

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

LESSON 1 · 14 MIN

The algorithm library — ranges, predicates and the first dozen calls

Every loop in Module 3's pattern catalogue — the search, the count, the minimum and maximum, the reversal — already exists in `<algorithm>` and `<numeric>` as a named function that takes a pair of iterators and, often, a callable. This lesson settles the interface those functions share — the half-open range, the predicate, the iterator that comes back — and works through the first dozen you will reach for daily: `find`, `find_if`, `count`, `count_if`, `min` / `max` and their `_element` cousins, `fill`, `iota`, `generate`, `reverse` and `copy`. The rest of the module builds on exactly this shape, and the last lesson shows how C++20 ranges shorten it.

Two headers, one convention

`<algorithm>` holds the functions that walk, search, rearrange and copy; `<numeric>` holds the arithmetic ones — `std::iota`, `std::accumulate`, `std::partial_sum`, `std::gcd`. Forgetting the second header is the commonest "`iota` is not a member of `std`" error. Every algorithm takes its input as a **half-open range** `[first, last)`: an iterator to the first element and one past the last. `last` is never dereferenced; it is a fence. That is why `v.end()` is a valid `last`, why an empty range is `first == last`, and why a sub-range is spelt with arithmetic on the iterators:

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

std::vector<int> v{5, 3, 8, 1, 9, 2};
std::sort(v.begin(), v.end());                 // the whole container
std::sort(v.begin() + 1, v.end() - 1);         // elements 1..4 only
int arr[] = {4, 2, 7};
std::reverse(std::begin(arr), std::end(arr));  // a built-in array, via <iter
```

Algorithms know nothing about containers — only iterators (Module 13, lesson 6) — which is why the same `std::find` searches a `std::vector`, a `std::deque`, a `std::string`, a built-in array or a `std::list`. The iterator category decides what is allowed: `std::sort` demands

random access, `std::find` accepts anything that can be incremented, dereferenced and compared.

Predicates and callables

Half the algorithms take a **callable** — anything that can be called with an element: a function, a function object (Module 11, lesson 4) or, almost always, a lambda (Module 4, lesson 5). A **unary predicate** takes one element and returns something convertible to `bool`; a **binary predicate** takes two. The algorithm calls it; you never do:

```
bool isNegative(int x) { return x < 0; }

std::vector<int> v{3, -1, 4, -1, 5};
auto a = std::find_if(v.begin(), v.end(), isNegative);                    // a
auto b = std::find_if(v.begin(), v.end(), [](int x) { return x > 3; });   // a
```

A predicate must not modify the elements it is given, and it should be a pure function of its argument: the algorithm may copy it, and the standard does not promise how many times or in what order it is called.

Finding things

```
std::vector<int> v{3, -1, 4, -1, 5};
auto it = std::find(v.begin(), v.end(), 4);            // first element == 4
if (it != v.end()) {
    std::cout << "found at index " << (it - v.begin()) << '\n';
}
```

`std::find` returns an **iterator** to the first match, or `last` when there is none — never `-1`, never a null pointer, never an exception. The `!= end()` test is the idiom; dereferencing the result without it is undefined behaviour. The index is `it - v.begin()` for random-access iterators and `std::distance(v.begin(), it)` for any category. `std::find_if` takes a predicate instead of a value and is the one you want for records:

```cpp
struct Account { std::string owner; int balance; };
std::vector<Account> accounts = loadAccounts();
auto overdrawn = std::find_if(accounts.begin(), accounts.end(),
                              [](const Account& a) { return a.balance < 0; });
if (overdrawn != accounts.end()) std::cout << overdrawn->owner << '\n';
```

`std::find_if_not` inverts the test. For "is this value present" `std::find(...) != end()` reads fine on a sequence, but for a `std::set` or `std::map` use the member `find` or `contains` — the free algorithm walks every node in O(n) where the member takes O(log n). `std::search` finds a sub-sequence, `std::adjacent_find` the first pair of equal neighbours, `std::mismatch` the point where two ranges first differ, and `std::equal` whether they differ at all.

Counting

```cpp
std::string s = "mississippi";
auto n  = std::count(s.begin(), s.end(), 's');
auto is = std::count_if(s.begin(), s.end(), [](char c) { return c == 'i'; });
```

Both return the iterator's `difference_type` — a signed integer, `std::ptrdiff_t` for every standard container — so `auto` or `long long` is the right variable: assigning to `int` narrows silently on a huge range, and `std::size_t` invites the signed/unsigned comparison warning. `std::count_if` over records with a lambda replaces the "loop, test, increment" pattern of Module 3, lesson 5.

Min, max and their positions

`std::min(a, b)` and `std::max(a, b)` compare two values of the **same** type — `std::max(3, 4.5)` fails to compile because `T` deduces to both `int` and `double`; write `std::max<double>(3, 4.5)` or `std::max(3.0, 4.5)`. The initialiser-list forms `std::min({a, b, c})` and `std::max({a, b, c})` take several values, `std::minmax(a, b)` returns a pair, and `std::clamp(x, lo, hi)` (C++17) pins a value into an interval. The two-argument forms and `clamp` return a reference to one of their arguments, so `const int& r = std::max(x, 5);` dangles when the temporary `5` wins — take the result by value.

Over a range, the `_element` versions return iterators:

```
std::vector<int> temps{21, 19, 25, 19, 25};
auto hottest = std::max_element(temps.begin(), temps.end());      // first 25
auto coldest = std::min_element(temps.begin(), temps.end());      // first 19
auto [lo, hi] = std::minmax_element(temps.begin(), temps.end());  // first 19
std::cout << *hottest << " at " << (hottest - temps.begin()) << '\n';
```

The asymmetry is deliberate and worth memorising: `max_element` returns the *first* of several equal maxima and `min_element` the first minimum, but `minmax_element` returns the first minimum and the **last** maximum — its pairwise scheme, about three comparisons per two elements, settles ties at opposite ends. On an empty range all three return `last`; test before dereferencing. A comparator as the third argument chooses the key: `std::max_element(accounts.begin(), accounts.end(), [](const Account& a, const Account& b) { return a.balance < b.balance; })` finds the richest account — the comparator is always "less than", even when you want the largest.

Filling, generating, reversing, copying

```
#include <numeric>

std::vector<int> v(5);
std::fill(v.begin(), v.end(), -1);                                  //
std::iota(v.begin(), v.end(), 10);                                  //
std::reverse(v.begin(), v.end());                                   //
std::generate(v.begin(), v.end(), [n = 0]() mutable { return n += 3; });  //
std::vector<int> w(v.size());
std::copy(v.begin(), v.end(), w.begin());                           //
```

`std::iota` writes `value, value + 1, value + 2, …` — the index vector `0..n-1` that a sort-by-key needs, or the "1 to n" a loop would build. `std::generate` calls a callable once per slot; a stateful lambda with a `mutable` init-capture (lesson 4) is the usual source. `std::copy` writes through an iterator into space that **must exist**: copying into an empty vector's `begin()` is undefined behaviour. Lesson 3 introduces `std::back_inserter`, the output iterator that grows the destination instead. `std::swap_ranges`, `std::rotate` and `std::shuffle` (which needs a seeded engine, and whose result is not something a judged program should print) round out the rearrangers.

Why algorithms beat hand loops

Three reasons, and an interviewer expects all three. **Names**: `std::count_if(…, isOverdue)` states the loop's purpose where `for (…) if (…) ++n;` makes the reader reconstruct it. **Correctness**: the empty range, the single element, the first-versus-last tie and the boundary are already handled — every bug in Module 3's "what goes wrong" tables lives in a loop someone wrote instead of calling. **Optimisation**: the library specialises where it can — `std::copy` and `std::fill` over trivially copyable elements become `memmove` and `memset`, `std::sort` is introsort with an insertion-sort tail, and the compiler inlines a lambda into the algorithm as if it were the loop body. A hand loop can match that; it rarely does. "No raw loops" is a goal rather than a law, but a loop that an algorithm could replace is a loop you should be able to justify.

Pitfalls

Mistake	What happens
`*std::find(…)` without the `!= end()` test	Dereferences `end()`: undefined behaviour
`std::max(3, 4.5)`	Compile error: `T` cannot be both `int` and `double`
`int n = std::count(…)`	Silent narrowing from `ptrdiff_t`; hold it in `auto`
`std::find` on a `std::map`	O(n) walk; the member `find` is O(log n)
`std::copy` into an empty destination	Writes past the end; size it first or use `std::back_inserter`
`std::sort(lst.begin(), lst.end())` on a `std::list`	Compile error: not random access; call `lst.sort()`
A range built from two containers (`a.begin(), b.end()`)	Undefined behaviour — the iterators never meet
Forgetting `<numeric>`	`std::iota` and `std::accumulate` "not a member of `std`"

Key takeaways

Every algorithm takes `[first, last)`; `last` is a fence that is never dereferenced, and an empty range is `first == last`.

`find` / `find_if` return an iterator or `last`; test `!= end()` before using it; the index is `it - begin()`.

`count` / `count_if` return a signed `difference_type` — hold it in `auto` or `long long`.

`std::min` / `max` need one type; `max_element` / `min_element` return the first extreme, `minmax_element` the first minimum and the last maximum.

`fill`, `iota`, `generate` write into existing space; `copy` needs a destination that already has room.

Algorithms win on intent, edge cases and library optimisation — reach for one before writing a loop.

EXERCISE 1

First overdue invoice

Read an integer `n`, then `n` lines each holding an invoice: an id (one word), an amount and the number of days it has been open (both integers). Then read a threshold `t`. An invoice is **overdue** when its days are strictly greater than `t`.

Store the invoices in a `std::vector` of a struct and answer with one predicate lambda and two algorithms: `std::count_if` for how many are overdue and `std::find_if` for the first overdue invoice in input order. No hand-written loop over the vector.

Input: `n`, then `n` lines of `id amount days`, then `t`. **Output:** `overdue: <count>`, then `first: <id> <amount>` for the first overdue invoice or `first: none` when there is none.

```
4
A1 120 45
B2 80 10
C3 200 31
D4 50 90
30
```

prints

```
overdue: 3
first: A1 120
```

INPUT 1

```
4
A1 120 45
B2 80 10
C3 200 31
D4 50 90
30
```

OUTPUT 1

```
overdue: 3
first: A1 120
```

INPUT 2

```
3
X 10 5
Y 20 7
Z 30 9
30
```

OUTPUT 2

```
overdue: 0
first: none
```

REFERENCE SOLUTION

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

struct Invoice {
    std::string id;
    int amount;
    int days;
};

int main() {
    int n;
    std::cin >> n;
    std::vector<Invoice> invoices(n);
    for (auto& inv : invoices) std::cin >> inv.id >> inv.amount >> inv.days;
    int threshold;
    std::cin >> threshold;

    const auto overdue = [threshold](const Invoice& inv) { return inv.days > thresho
    const auto count = std::count_if(invoices.begin(), invoices.end(), overdue);
    std::cout << "overdue: " << count << '\n';

    const auto first = std::find_if(invoices.begin(), invoices.end(), overdue);
    if (first == invoices.end()) {
        std::cout << "first: none\n";
    } else {
        std::cout << "first: " << first->id << ' ' << first->amount << '\n';
    }
    return 0;
}
```

EXERCISE 2

Readings at the extremes

Read an integer `n` (at least 1), then `n` integer readings. With **one** call to `std::minmax_element` find the smallest and the largest reading and print their values and 0-based positions, computed as the distance of each iterator from `begin()`. `std::minmax_element` returns the **first** occurrence of the minimum and the **last** occurrence of the maximum — print exactly those positions. Then print the spread, the maximum minus the minimum, computed in `long long`.

Input: `n`, then `n` integers. **Output:** three lines: `min <value> at <index>`, `max <value> at <index>`, `spread <value>`.

Example: `5` then `3 9 1 9 4` →

```
min 1 at 2
max 9 at 3
spread 8
```

INPUT 1

```
5
3 9 1 9 4
```

OUTPUT 1

```
min 1 at 2
max 9 at 3
spread 8
```

INPUT 2

```
4
-5 -2 -9 -2
```

OUTPUT 2

```
min -9 at 2
max -2 at 3
spread 7
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    std::vector<int> readings(n);
    for (int& r : readings) std::cin >> r;

    const auto [lo, hi] = std::minmax_element(readings.begin(), readings.end());
    std::cout << "min " << *lo << " at " << (lo - readings.begin()) << '\n';
    std::cout << "max " << *hi << " at " << (hi - readings.begin()) << '\n';
    std::cout << "spread " << (static_cast<long long>(*hi) - *lo) << '\n';
    return 0;
}
```

QUIZ

1. What does `std::find(v.begin(), v.end(), 7)` return when no element equals 7?

A. `-1`

B. `v.end()`

C. `nullptr`

D. It throws `std::out_of_range`

2. What does this print?

```
std::vector<int> v{4, 9, 2, 9, 1};
auto [lo, hi] = std::minmax_element(v.begin(), v.end());
std::cout << (lo - v.begin()) << ' ' << (hi - v.begin()) << '
';
```

A. `4 1`

B. `4 3`

C. `2 1`

D. `2 3`

3. What happens with `std::max(3, 4.5)`?

A. It returns `4.5`

B. It returns `4`

C. Compile error: `T` is deduced as both `int` and `double`

D. Undefined behaviour

4. Which header declares `std::iota` and `std::accumulate`?

A. `<algorithm>`

B. `<numeric>`

C. `<iterator>`

D. `<functional>`

5. What is the return type of `std::count_if` over a `std::vector<int>`?

A. `int`

B. `std::size_t`

C. The iterator's `difference_type` — a signed integer, `std::ptrdiff_t` here

D. `bool`

6. What does this print?

```
std::vector<int> v(5);
std::iota(v.begin(), v.end(), 3);
std::reverse(v.begin(), v.end());
std::cout << v.front() << ' ' << v.back() << '
';
```

A. `3 7`

B. `7 3`

C. `0 4`

D. `4 0`

7. Which statement about algorithms versus hand-written loops is correct?

A. Algorithms only work on sorted data

B. An algorithm call names the loop's intent, has the edge cases already right, and lets the library specialise — `std::copy` of trivially copyable elements becomes `memmove`

C. Hand loops are always faster because they avoid a function call

D. `std::find` on a `std::map` runs in O(log n)

ANSWER KEY · THE ALGORITHM LIBRARY — RANGES, PREDICATES AND THE FIRST DOZEN CALLS

1. B Algorithms report "not found" as the `last` iterator of the range they were given — never a sentinel integer or an exception. That is why the result must be compared with `end()` before it is dereferenced.

2. B `minmax_element` returns the first minimum (1, at index 4) and the **last** maximum (the second 9, at index 3). `max_element` on its own would have returned index 1 — the two functions settle ties differently.

3. C `std::max` is `template <class T> const T& max(const T&, const T&)`; deduction from `3` gives `int` and from `4.5` gives `double`, and the conflict is an error. `std::max<double>(3, 4.5)` or `std::max(3.0, 4.5)` fixes it.

4. B The arithmetic algorithms — `iota`, `accumulate`, `reduce`, `partial_sum`, `inner_product`, `gcd` — live in `<numeric>`. `<algorithm>` has the searching, counting and rearranging ones; forgetting `<numeric>` gives "`iota` is not a member of `std`".

5. C Counts are returned as the iterator's `difference_type`, which is signed. Holding it in `auto` or `long long` is right; `int` narrows on a huge range and `std::size_t` invites the signed/unsigned comparison warning.

6. B `iota` writes 3 4 5 6 7 into the five slots; `reverse` turns that into 7 6 5 4 3, so the front is 7 and the back is 3.

7. B Intent, correctness and optimisation are the three reasons. The lambda is inlined, so there is no call overhead to avoid; only the binary-search family needs sorted data; and `std::find` on a map is a linear walk — the member `find` is the O(log n) one.

LESSON 2 · 15 MIN

Sorting and searching — comparators, stability and binary search

`std::sort` is the algorithm everyone knows and the one most often called wrong: with a comparator that is not a strict weak ordering, with equal keys whose order the program then silently depends on, or followed by a binary search that uses a different rule from the sort. This lesson settles the comparator contract, sorting records by a key with a tie-break, when `std::stable_sort` is required, the cheaper partial orders `std::partial_sort` and `std::nth_element`, and the four binary-search functions that a sorted range unlocks.

`std::sort`

```
#include <algorithm>
#include <functional>

std::vector<int> v{5, 3, 8, 1};
std::sort(v.begin(), v.end());                      // 1 3 5 8
std::sort(v.begin(), v.end(), std::greater<>{});    // 8 5 3 1
```

`std::sort` is introsort — quicksort that switches to heapsort when a partition degenerates and to insertion sort on short runs — with a guaranteed O(n log n) worst case since C++11. It needs random-access iterators: `std::vector`, `std::deque`, `std::array`, `std::string` and built-in arrays qualify. A `std::list` does not, and has its own `lst.sort()` member that relinks nodes instead of moving values. The default order is `operator<`; the third argument replaces it. `std::greater<>{}` — the transparent function object from `<functional>` — is the idiom for descending; a lambda does the same at more length.

The comparator contract

A comparator `comp(a, b)` answers one question: must `a` come strictly before `b`? The standard requires a **strict weak ordering**: `comp(a, a)` is false (irreflexive), `comp(a, b)` and `comp(b, a)` are never both true (asymmetric), the relation is transitive, and "neither before the other" is itself transitive, so it partitions the elements into equivalence classes. `<=` breaks the first rule, and the result is undefined behaviour, not merely a wrong order: introsort trusts

the comparator to stop its inner scans, and a comparator that says an element comes before itself lets a scan run past the end of the array and crash. Rules of thumb: write `<` or `>`, never `⩽` or `⩾`; make the comparator a pure function of its two arguments; and never compare on something that changes while the sort runs.

Sorting records by a key

```
struct Player { std::string name; int score; };
std::vector<Player> players = readPlayers();
std::sort(players.begin(), players.end(), [](const Player& a, const Player& b)
    if (a.score ≠ b.score) return a.score > b.score;   // higher score first
    return a.name < b.name;                            // then alphabetical
});
```

Take the elements by `const&` — copying a `std::string` per comparison is the difference between a fast sort and a slow one. The tie-break matters twice over: it makes the order **total**, so the output is the same on every run and with every standard library, and it stops the order of equal scores from being an accident. `std::tie` writes the same comparator in one line — `return std::tie(b.score, a.name) < std::tie(a.score, b.name);` — because tuples compare lexicographically, and for the same reason a `std::vector<std::pair<int, std::string>>` sorts by `first` and then `second` with no comparator at all. A type with a defaulted `operator⟺` (Module 11, lesson 2) sorts by its members in declaration order.

Stability

`std::sort` makes no promise about the relative order of elements that compare equal: after sorting invoices by customer, one customer's invoices may come out in any order, and that order can change between library versions. When the input order carries meaning — "sort by priority, but within a priority keep arrival order" — use `std::stable_sort`, a merge sort that preserves it. Its cost is a temporary buffer: O(n log n) when the allocation succeeds, O(n log² n) in place when it does not. The alternative is the tie-break above, which turns the question into a total order. One of the two is mandatory whenever equal keys can occur and the output is compared exactly — a judged program that sorts by score alone and then prints names is printing an unspecified order.

Partial orders: `partial_sort` and `nth_element`

Sorting the whole range to read the top three is wasted work:

```
std::vector<int> v{9, 1, 8, 3, 7, 2};
std::partial_sort(v.begin(), v.begin() + 3, v.end());  // 1 2 3, then the rest
std::nth_element(v.begin(), v.begin() + 2, v.end());   // v[2] == 3; before it
```

`std::partial_sort(first, middle, last)` puts the smallest `middle - first` elements, sorted, at the front, in O(n log k). `std::nth_element(first, nth, last)` places at `nth` the element that a full sort would put there and partitions the rest around it — everything before is no greater, everything after no smaller, neither side sorted — in O(n) on average. That is the median in linear time: `nth_element` at `n / 2`, then read `v[n / 2]`; for an even `n` the lower middle is the largest element of the left part, `*std::max_element(v.begin(), v.begin() + n / 2)`, because the partition guarantees it is there. Both rearrange the range; copy first if the original order is still needed.

Binary search on a sorted range

A sorted range answers "is it there", "where would it go" and "how many" in O(log n):

```
std::vector<int> v{1, 3, 3, 3, 7, 9};
bool has = std::binary_search(v.begin(), v.end(), 3);       // true
auto lo  = std::lower_bound(v.begin(), v.end(), 3);         // first element
auto hi  = std::upper_bound(v.begin(), v.end(), 3);         // first element >
auto count = hi - lo;                                       // 3
auto [a, b] = std::equal_range(v.begin(), v.end(), 3);      // the same pair
auto slot = std::lower_bound(v.begin(), v.end(), 5);        // index 4: where
```

`lower_bound` returns the first position whose element is **not less than** the value — the insertion point that keeps the range sorted — and `upper_bound` the first whose element is **greater**. Their difference is the count of a value, and `upper_bound(hi) - lower_bound(lo)` counts everything in the closed interval `[lo, hi]` in two searches. Both return `last` when every element is smaller, so a `lower_bound` result must be tested against `end()` and then against the value before it is called "found"; `std::binary_search` does both tests and returns only a `bool`, which is why it cannot tell you *where*.

Two conditions are non-negotiable. The range must be sorted — or at least partitioned — by the **same** comparator you pass to the search; searching a descending range with the default ascending rule returns nonsense with no diagnostic. And the O(log n) counts comparisons: with random-access iterators the steps are O(log n) too, but on a `std::list` the algorithm still walks O(n) increments. For `std::set` and `std::map` use the member `lower_bound` (Module 13, lesson 4).

`std::unique` after a sort

```
std::vector<int> v{3, 1, 3, 2, 1};
std::sort(v.begin(), v.end());                       // 1 1 2 3 3
v.erase(std::unique(v.begin(), v.end()), v.end());   // 1 2 3
```

`std::unique` removes only *consecutive* duplicates — on an unsorted range it removes almost nothing — by shifting the survivors to the front and returning the new logical end; the container's size does not change until `erase` cuts the tail. Sort, `unique`, `erase` is the three-line deduplication that Module 20's idiom sheet lists first. `std::is_sorted` checks the precondition when you are not sure the data arrived in order.

The cost table

Call	Complexity	Requires
`std::sort`	O(n log n)	random access, a strict weak ordering
`std::stable_sort`	O(n log n) with a buffer, else O(n log² n)	random access
`std::partial_sort` (k of n)	O(n log k)	random access
`std::nth_element`	O(n) average	random access
`std::lower_bound` / `upper_bound` / `binary_search`	O(log n) comparisons	sorted by the same comparator
`std::unique`	O(n)	duplicates adjacent (sort first)

Pitfalls

Mistake	What happens
`return a ⩽ b;` in a comparator	Undefined behaviour — may crash or never finish
Sorting by score alone, then printing names	Equal scores come out in an unspecified order
`lower_bound` on an unsorted or differently sorted range	Wrong answer, no diagnostic
`*std::lower_bound(…)` without testing `end()`	Dereferences past the end
`std::unique` without `erase`	The duplicates still count in `size()`
A comparator taking `Player` by value	A string copy per comparison

Key takeaways

`std::sort` is O(n log n) introsort on random-access ranges; a `std::list` sorts itself with its member.

A comparator is a strict weak ordering: `<`, never `⩽`; tie-break on a second key so the order is total.

`std::stable_sort` keeps equal elements in input order; use it, or a tie-break, whenever equal keys can occur.

`std::partial_sort` for the top k, `std::nth_element` for the k-th element and the median in linear time.

`lower_bound` is the first element ≥, `upper_bound` the first >, and their difference is a count; the range must be sorted by the same comparator.

Sort, then `std::unique`, then `erase` — `unique` removes only neighbours.

EXERCISE 1

Leaderboard

Read an integer `n`, then `n` lines of `name score` (names are distinct single words, scores are integers). Sort the players with `std::sort` and a comparator lambda that orders by score **descending** and breaks ties by name **ascending**, so that the order is total and the output unique. Print the ranking as `<rank>. <name> <score>`, ranks starting at 1.

Take the elements by `const&` in the comparator, and write the tie-break so the comparator stays a strict weak ordering (never `<=`).

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

```
4
ada 90
bob 75
cy 90
dee 60
```

prints

```
1. ada 90
2. cy 90
3. bob 75
4. dee 60
```

INPUT 1

```
4
ada 90
bob 75
cy 90
dee 60
```

OUTPUT 1

```
1. ada 90
2. cy 90
3. bob 75
4. dee 60
```

INPUT 2

```
3
zed 10
amy 10
max 10
```

OUTPUT 2

```
1. amy 10
2. max 10
3. zed 10
```

REFERENCE SOLUTION

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

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

int main() {
    int n;
    std::cin >> n;
    std::vector<Player> players(n);
    for (auto& p : players) std::cin >> p.name >> p.score;

    std::sort(players.begin(), players.end(), [](const Player& a, const Player& b) {
        if (a.score != b.score) return a.score > b.score;
        return a.name < b.name;
    });

    for (std::size_t i = 0; i < players.size(); ++i) {
        std::cout << (i + 1) << ". " << players[i].name << ' ' << players[i].score <
    }
    return 0;
}
```

EXERCISE 2

How many in range

Read an integer `n`, then `n` integers in no particular order, then an integer `q`, then `q` queries `lo hi` (with `lo ⩽ hi`). Sort the values **once**; then answer each query with two binary searches — `std::lower_bound` for `lo` and `std::upper_bound` for `hi` — and print the number of values in the closed interval `[lo, hi]` as the distance between the two iterators. No loop over the data per query.

Input: `n`, the values, `q`, then `q` lines of `lo hi`. **Output:** `q` lines, one count each.

```
6
5 1 9 3 5 7
3
3 7
5 5
10 20
```

prints

```
4
2
0
```

INPUT 1

```
6
5 1 9 3 5 7
3
3 7
5 5
10 20
```

OUTPUT 1

```
4
2
0
```

INPUT 2

```
4
-3 0 2 8
2
-10 -1
0 8
```

OUTPUT 2

```
1
3
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    std::vector<int> values(n);
    for (int& v : values) std::cin >> v;
    std::sort(values.begin(), values.end());

    int q;
    std::cin >> q;
    for (int i = 0; i < q; ++i) {
        int lo, hi;
        std::cin >> lo >> hi;
        const auto first = std::lower_bound(values.begin(), values.end(), lo);
        const auto last = std::upper_bound(values.begin(), values.end(), hi);
        std::cout << (last - first) << '\n';
    }
    return 0;
}
```

QUIZ

1. `std::list<int> lst{3, 1, 2}; std::sort(lst.begin(), lst.end());` — what happens?

A. The list is sorted

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

C. It compiles but the order is unspecified

D. It sorts in O(n²)

2. A comparator passed to `std::sort` is written `[](int a, int b) { return a ⩽ b; }`. What does the standard say?

A. It sorts ascending, like `<`

B. It sorts descending

C. Undefined behaviour: `⩽` is not a strict weak ordering, and the sort may read past the end or never finish

D. Compile error

3. What does `std::lower_bound(v.begin(), v.end(), x)` return on a sorted vector?

A. An iterator to the last element less than `x`

B. An iterator to the first element that is **not less than** `x`, or `end()`

C. An iterator to the first element greater than `x`

D. `true` or `false`

4. `v` is sorted. Which expression is the number of elements equal to `x`?

A. `std::upper_bound(v.begin(), v.end(), x) - std::lower_bound(v.begin(), v.end(), x)`

B. `std::lower_bound(v.begin(), v.end(), x) - std::upper_bound(v.begin(), v.end(), x)`

C. `std::binary_search(v.begin(), v.end(), x)`

D. `std::find(v.begin(), v.end(), x) - v.begin()`

5. After this code, what is `v[2]`?

```
std::vector<int> v{9, 1, 8, 3, 7};
std::nth_element(v.begin(), v.begin() + 2, v.end());
```

A. `8`

B. `7`

C. `3`

D. Unspecified

6. You sort support tickets by priority and must keep tickets of equal priority in the order they arrived. Which is correct?

A. `std::sort` — it never reorders equal elements

B. `std::stable_sort`, or `std::sort` with a tie-break on the arrival index

C. `std::partial_sort` with the full range

D. `std::nth_element` at every position

7. What is `v.size()` after this?

```
std::vector<int> v{1, 1, 2, 1};
v.erase(std::unique(v.begin(), v.end()), v.end());
```

A. `2`

B. `3`

C. `4`

D. `1`

ANSWER KEY · SORTING AND SEARCHING — COMPARATORS, STABILITY AND BINARY SEARCH

1. B Introsort indexes into the range, so it requires random access; a list's bidirectional iterators do not satisfy the algorithm and the call fails to compile. `std::list` provides its own `sort()` member that relinks nodes.

2. C A comparator must be irreflexive — `comp(a, a)` false — and `<=` says an element comes before itself. Introsort relies on that guarantee to stop its scans, so the result is undefined behaviour, which in practice means crashes on inputs with equal elements.

3. B `lower_bound` is the first position where `x` could be inserted without breaking the order — the first element ≥ `x`. `upper_bound` is the first element > `x`; `binary_search` is the one that returns a `bool`.

4. A The elements equal to `x` occupy `[lower_bound, upper_bound)`, so the difference is their count — two O(log n) searches. `binary_search` returns only a `bool`, and `find` gives the index of the first match after a linear scan.

5. B `nth_element` puts at position 2 the element a full sort would put there: sorted, the vector is 1 3 7 8 9, so `v[2]` is 7. Only the other positions are unspecified — they are partitioned around 7 but not sorted.

6. B `std::sort` makes no promise about equal elements, and the order may differ between library versions. `std::stable_sort` preserves input order among equals; adding the arrival index as a second key makes the order total, which also works.

7. B `unique` removes only **consecutive** duplicates: 1 1 2 1 becomes 1 2 1, three elements. Sorting first would have made every duplicate adjacent and left two.

LESSON 3 · 15 MIN

Transforming and reducing — transform, accumulate and the erase–remove idiom

Most data processing is three verbs: *map* each element to something else, *filter* the ones that matter, *reduce* them to one result. The standard library spells them `std::transform`, `std::copy_if` / `std::remove_if` / `std::partition`, and `std::accumulate` / `std::reduce`, with `std::all_of` / `any_of` / `none_of` and `std::count_if` as the yes/no reductions. This lesson settles how each writes its output — into existing space, through `std::back_inserter`, or by shifting survivors to the front — and the one trap that fails more submissions than any other in this module: `std::accumulate` sums in the type of its *initial value*, so a bare `0` sums in `int` and overflows.

`std::transform`

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

std::vector<int> prices{120, 80, 200};
std::vector<int> withTax(prices.size());
std::transform(prices.begin(), prices.end(), withTax.begin(),
               [](int p) { return p + p / 10; });          // 132 88 220

std::transform(prices.begin(), prices.end(), prices.begin(),
               [](int p) { return p * 2; });               // in place: 240 1
std::vector<int> qty{2, 1, 4};
std::vector<long long> lineTotal(prices.size());
std::transform(prices.begin(), prices.end(), qty.begin(), lineTotal.begin(),
               [](int p, int q) { return static_cast<long long>(p) * q; });
```

The unary form applies a callable to each element and writes the results through an output iterator; the binary form walks two input ranges in step (the second must be at least as long as the first) and combines the pairs. The output iterator points at space that must already exist — hence `withTax(prices.size())` — or at the input itself for an in-place transform. A `std::string` is a range of `char`, so `std::transform(s.begin(), s.end(), s.begin(), [](unsigned char c) { return static_cast<char>(std::toupper(c)); })` upper-cases in place, with the `unsigned char` parameter that Module 5, lesson 3 requires for `<cctype>`.

`std::back_inserter`

```
#include <iterator>

std::vector<long long> squares;                    // empty: no size at all
std::transform(v.begin(), v.end(), std::back_inserter(squares),
               [](int x) { return 1LL * x * x; });
```

`std::back_inserter(c)` (in `<iterator>`) returns an output iterator whose every assignment calls `c.push_back` — the destination grows as the algorithm writes. It works with `transform`, `copy`, `copy_if`, `generate_n` and later with ranges. A `reserve` first keeps it from reallocating. Its siblings `std::front_inserter` (for `std::deque` and `std::list`) and `std::inserter(c, pos)` (for `std::set` and `std::map`) do the same through `push_front` and `insert`.

`std::accumulate` and the `0LL` rule

```
#include <numeric>

std::vector<long long> big{2'000'000'000, 2'000'000'000};
auto wrong = std::accumulate(big.begin(), big.end(), 0);      // sums in int: 
auto right = std::accumulate(big.begin(), big.end(), 0LL);    // 4000000000
```

`std::accumulate(first, last, init)` is a left fold: `acc = init; for each x: acc = acc + x; return acc;`. The type of `acc` is the type of `init`, deduced from the argument you pass — not from the elements. `0` is an `int`, so even a `std::vector<long long>` is summed in `int`, and the overflow is undefined behaviour the optimiser is free to exploit. Write `0LL` for integer sums, `0.0` for a `std::vector<double>` (with `0` every partial sum would be truncated to an integer), and `std::string{}` to join strings. The fourth argument replaces `+`:

```
auto product = std::accumulate(v.begin(), v.end(), 1LL, std::multiplies<>{});
auto csv = std::accumulate(words.begin(), words.end(), std::string{},
                           [](std::string acc, const std::string& w) {
                               return acc.empty() ? w : std::move(acc) + "," 
                           });
auto longest = std::accumulate(words.begin(), words.end(), std::size_t{0},
                               [](std::size_t best, const std::string& w) { re
```

`std::reduce` (C++17, also in `<numeric>`) is the same fold with one freedom removed and one gained: it may combine the elements in **any order and grouping**, which lets it run in parallel under an execution policy, so the operation must be associative and commutative — fine for `+` on integers, wrong for string concatenation, and for floating point it may change the last digits between runs. For a sequential integer sum `std::accumulate` with `0LL` is the habit; `std::reduce(v.begin(), v.end(), 0LL)` gives the same answer.

Yes/no reductions

```
bool allPositive = std::all_of(v.begin(), v.end(), [](int x) { return x > 0; }
bool anyZero     = std::any_of(v.begin(), v.end(), [](int x) { return x == 0;
bool noNegatives = std::none_of(v.begin(), v.end(), [](int x) { return x < 0;
auto evens       = std::count_if(v.begin(), v.end(), [](int x) { return x % 2
```

All three short-circuit like the loops they replace. On an empty range `all_of` and `none_of` are `true` and `any_of` is `false` — vacuous truth, the same answer a `for` loop with a flag would give, and the answer to know for the edge case.

Selecting: `copy_if`, `remove_if` and erase

```
std::vector<int> v{4, 7, 2, 9, 5};
std::vector<int> big;
std::copy_if(v.begin(), v.end(), std::back_inserter(big), [](int x) { return x

auto newEnd = std::remove_if(v.begin(), v.end(), [](int x) { return x > 4; });
// v is now 4 2 ? ? ? — size still 5; [newEnd, end) holds unspecified leftover
v.erase(newEnd, v.end());
```

`copy_if` writes the survivors elsewhere and leaves the source alone. `remove_if` cannot shrink a range — an algorithm sees only iterators — so it shifts the elements that stay to the front, returns the new logical end, and leaves whatever it likes beyond it. The container's `erase(newEnd, end())` finishes the job: the **erase–remove idiom**, two calls that are one operation. Forgetting the erase is the classic bug — `size()` is unchanged and the tail is garbage. C++20 wraps both in one function that also returns how many went: `std::erase_if(v, pred)` (Module 13, lesson 2). `std::remove` does the same for a value.

`std::partition`

```
std::vector<int> v{4, 7, 2, 9, 5};
auto mid = std::partition(v.begin(), v.end(), [](int x) { return x % 2 == 0; }
// [begin, mid) are the evens, [mid, end) the odds — each group in an unspecif
std::sort(v.begin(), mid);
std::sort(mid, v.end());
```

`std::partition` moves every element that satisfies the predicate before every element that does not and returns the boundary — an iterator that is itself a range end, which is what the two `sort` calls use. It is O(n) with no allocation but scrambles the order inside each group; `std::stable_partition` keeps the input order within both groups at the cost of a buffer, and `std::partition_point` finds the boundary of an already partitioned range in O(log n). `std::nth_element` and quicksort are this partition applied repeatedly.

Scans: `partial_sum` and `adjacent_difference`

```
std::vector<long long> v{3, 1, 4, 1, 5};
std::vector<long long> prefix(v.size());
std::partial_sum(v.begin(), v.end(), prefix.begin());                    // 
std::vector<long long> deltas(v.size());
std::adjacent_difference(v.begin(), v.end(), deltas.begin());            // 
long long dot = std::inner_product(v.begin(), v.end(), v.begin(), 0LL);  // 
```

Prefix sums turn every "sum of a sub-range" query into one subtraction: `prefix[j] - prefix[i - 1]`. Note the type: `std::partial_sum` accumulates in the **input's** value type, so a `std::vector<int>` of large values overflows even when the output is a `std::vector<long long>` — convert the input first, or use `std::inclusive_scan(first, last, out, std::plus<>{}, 0LL)`, whose initial value sets the accumulator type the way `accumulate` does. `std::adjacent_difference` copies the first element unchanged and then writes each difference from its predecessor; `std::inner_product` takes its accumulator type from its init, so `0LL` again.

The shape of a pipeline

```cpp
struct Line { int qty; int price; };
std::vector<Line> stock;                                          // 1. filter
std::copy_if(lines.begin(), lines.end(), std::back_inserter(stock),
             [](const Line& l) { return l.qty > 0; });
std::vector<long long> values;                                    // 2. map
std::transform(stock.begin(), stock.end(), std::back_inserter(values),
               [](const Line& l) { return 1LL * l.qty * l.price; });
long long total = std::accumulate(values.begin(), values.end(), 0LL);   // 3.
```

Filter, map, reduce — three named steps with two intermediate vectors. The intermediates are the price of the classic algorithms' iterator-pair interface, and they are exactly what lesson 5's views remove: `lines | filter | transform` computes the same total lazily, with no copies.

Pitfalls

Mistake	What happens
`std::accumulate(…, 0)` over `long long` or `double` data	Sums in `int`: overflow (undefined behaviour) or truncation
`std::transform` into an empty vector's `begin()`	Writes past the end — size it or use `std::back_inserter`
`std::remove_if` without `erase`	`size()` unchanged; the tail is unspecified
`std::reduce` with string concatenation	Elements combined in any order — use `accumulate`
Reading order out of `std::partition`'s groups	Unspecified — `stable_partition`, or sort each side
`std::partial_sum` over `int` into `long long`	The accumulator is `int`; it overflows anyway
`std::copy` whose destination overlaps the source's front	Use `std::copy_backward`

Key takeaways

`std::transform` maps into existing space, in place, or through `std::back_inserter`; the binary form zips two ranges.

`std::accumulate` folds in the type of its initial value: `0LL`, `0.0`, `std::string{}` — never a bare `0` over wide data.

`std::reduce` may reorder; use it only for associative, commutative operations.

`all_of` / `any_of` / `none_of` short-circuit; on an empty range they are true, false and true.

`remove_if` shifts and returns the new end — always `erase` after it, or call `std::erase_if`.

`partition` returns the boundary iterator; the groups are unordered unless you use `stable_partition`.

EXERCISE 1

Order total

Read an integer `n` (at least 1), then `n` lines of `qty price` — two integers; a negative quantity is a refund line. Build a `std::vector<long long>` of line totals (`qty * price`, multiplied in `long long` — cast one operand before multiplying) with `std::transform` writing through `std::back_inserter`. Then reduce: `std::accumulate` with the initial value `0LL` for the grand total, and `std::max_element` for the largest line total.

Input: `n`, then `n` lines. **Output:** `total <sum>` and `largest <max line total>`.

Example: `3` then `2 250`, `1 999`, `4 10` →

```
total 1539
largest 999
```

INPUT 1

```
3
2 250
1 999
4 10
```

OUTPUT 1

```
total 1539
largest 999
```

INPUT 2

```
2
100000 100000
1 1
```

OUTPUT 2

```
total 10000000001
largest 10000000000
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <numeric>
#include <utility>
#include <vector>

int main() {
    int n;
    std::cin >> n;
    std::vector<std::pair<int, int>> lines(n);
    for (auto& [qty, price] : lines) std::cin >> qty >> price;

    std::vector<long long> totals;
    totals.reserve(lines.size());
    std::transform(lines.begin(), lines.end(), std::back_inserter(totals),
                   [](const std::pair<int, int>& line) {
                       return static_cast<long long>(line.first) * line.second;
                   });

    const long long total = std::accumulate(totals.begin(), totals.end(), 0LL);
    const auto largest = std::max_element(totals.begin(), totals.end());
    std::cout << "total " << total << '\n';
    std::cout << "largest " << *largest << '\n';
    return 0;
}
```

EXERCISE 2

Triage the ticket queue

Read an integer `n`, then `n` lines of `name priority` (distinct single-word names; priority is an integer, 0 or more). Process the queue with three algorithms:

> Drop every **closed** ticket (priority `0`) with `std::remove_if` followed by `erase` (or `std::erase_if`), and print `removed <k>`.
>
> `std::partition` the rest so that **urgent** tickets (priority 5 or more) come first. The iterator it returns is the end of the urgent group.
>
> `std::sort` each group by name — `partition` scrambles the order inside a group, and sorting both halves makes the output unique.

Input: `n`, then `n` lines. **Output:** `removed <k>`, then `urgent <count>` followed by one `name priority` line per urgent ticket in name order, then `normal <count>` followed by the rest likewise.

```
6
ann 7
bob 3
cy 0
dan 9
eve 4
fay 0
```

prints

```
removed 2
urgent 2
ann 7
dan 9
normal 2
bob 3
eve 4
```

INPUT 1

```
6
ann 7
bob 3
cy 0
dan 9
eve 4
fay 0
```

OUTPUT 1

```
removed 2
urgent 2
ann 7
dan 9
normal 2
bob 3
eve 4
```

INPUT 2

```
3
x 5
y 5
z 5
```

OUTPUT 2

```
removed 0
urgent 3
x 5
y 5
z 5
normal 0
```

REFERENCE SOLUTION

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

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

int main() {
    int n;
    std::cin >> n;
    std::vector<Ticket> tickets(n);
    for (auto& t : tickets) std::cin >> t.name >> t.priority;

    const auto closed = std::remove_if(tickets.begin(), tickets.end(),
                                       [](const Ticket& t) { return t.priority == 0;
    const auto removed = tickets.end() - closed;
    tickets.erase(closed, tickets.end());
    std::cout << "removed " << removed << '\n';

    const auto boundary = std::partition(tickets.begin(), tickets.end(),
                                         [](const Ticket& t) { return t.priority ⩾
    const auto byName = [](const Ticket& a, const Ticket& b) { return a.name < b.nam
    std::sort(tickets.begin(), boundary, byName);
    std::sort(boundary, tickets.end(), byName);

    std::cout << "urgent " << (boundary - tickets.begin()) << '\n';
    for (auto it = tickets.begin(); it ≠ boundary; ++it) {
        std::cout << it→name << ' ' << it→priority << '\n';
    }
    std::cout << "normal " << (tickets.end() - boundary) << '\n';
    for (auto it = boundary; it ≠ tickets.end(); ++it) {
        std::cout << it→name << ' ' << it→priority << '\n';
    }
    return 0;
}
```

QUIZ

1. What does this print?

```
std::vector<long long> v{2000000000, 2000000000};
std::cout << std::accumulate(v.begin(), v.end(), 0) << '
';
```

A. `4000000000`

B. `-294967296`

C. Undefined behaviour: the sum is computed in `int` because the initial value is an `int`

D. Compile error: `0` is not a `long long`

2. What does this print?

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

A. `2`

B. `4`

C. `0`

D. Compile error

3. What does `std::all_of(v.begin(), v.end(), pred)` return when `v` is empty?

A. `false`

B. `true`

C. Undefined behaviour

D. It throws

4. `std::vector<int> out; std::transform(v.begin(), v.end(), out.begin(), f);` — what is wrong?

A. Nothing: `transform` grows `out`

B. It writes through `out.begin()` into space that does not exist — undefined behaviour; size `out` first or use `std::back_inserter(out)`

C. Compile error: `out` must be `const`

D. `f` is called twice per element

5. Which reduction is **unsafe** with `std::reduce`?

A. Summing a `std::vector<long long>` with `0LL`

B. Joining a `std::vector<std::string>` into one string with `+`

C. Multiplying a `std::vector<int>` with `1LL` and `std::multiplies<>{}`

D. Finding a maximum with a lambda that returns the larger of two values

6. What does `std::partition(first, last, pred)` return, and what is the order inside each group?

A. The number of elements satisfying `pred`; both groups keep input order

B. An iterator to the first element of the group that fails `pred`; the order inside each group is unspecified

C. `true` if any element satisfies `pred`

D. An iterator to the last element satisfying `pred`; both groups are sorted

7. `std::vector<int> big` holds values near two billion. `std::partial_sum(big.begin(), big.end(), out.begin())` writes into a `std::vector<long long> out`. What happens?

A. The running sum is computed in `long long` because `out` is `long long`

B. The running sum is computed in `int` — `partial_sum`'s accumulator has the input's value type — so it overflows

C. Compile error: the value types differ

D. The output is truncated to `int`

ANSWER KEY · TRANSFORMING AND REDUCING — TRANSFORM, ACCUMULATE AND THE ERASE–REMOVE IDIOM

1. C The accumulator has the type of the third argument, and `0` is an `int`, so the two `long long` elements are added into an `int` and overflow — undefined behaviour, whatever number happens to appear. `0LL` makes the accumulator `long long`.

2. B An algorithm cannot change a container's size — it only moves elements. `remove_if` shifts 1 and 3 to the front and returns the new logical end; without `v.erase(newEnd, v.end())` the size stays 4 and the tail holds unspecified values.

3. B On an empty range `all_of` and `none_of` are vacuously `true` and `any_of` is `false` — the same answers a `for` loop with a flag would give, because no element ever fails the test.

4. B `transform` assigns through whatever output iterator it is given, and an empty vector's `begin()` points at nothing. Either `out.resize(v.size())` first or pass `std::back_inserter(out)`, which turns each write into `push_back`.

5. B `std::reduce` may combine the elements in any order and grouping, so the operation must be associative and commutative. String concatenation is not commutative — `"a" + "b"` differs from `"b" + "a"` — so use `std::accumulate`, which folds left to right.

6. B `partition` returns the boundary between the two groups — an iterator usable as the end of the first and the start of the second. It makes no promise about order within a group; `std::stable_partition` does, at the cost of a buffer.

7. B Unlike `accumulate`, `partial_sum` takes no initial value: its accumulator is the input iterator's `value_type`, here `int`. Convert the input to `long long` first, or use `std::inclusive_scan` with an explicit `0LL` init.

LESSON 4 · 15 MIN

Lambdas in depth — captures, closures, std::function and recursion

Module 4, lesson 5 introduced the lambda as a function written in place. This lesson takes the lid off: a lambda is an object of a compiler-generated class, its captures are that object's data members, and once you see that, every rule follows — why `[&]` dangles, why `mutable` exists and what copying a lambda copies, why `std::function` costs something `auto` does not, and how a lambda calls itself. The exercises build a counter that owns its state through an init-capture and a command table of `std::function` values.

What a lambda really is

```
int limit = 10;
auto below = [limit](int x) { return x < limit; };
```

The compiler turns that into roughly:

```
struct Lambda7 {
    int limit;                                  // one member per capture
    bool operator()(int x) const { return x < limit; }
};
auto below = Lambda7{limit};
```

The **closure type** is unique to that lambda expression — two textually identical lambdas have two different types — which is why the variable must be `auto` (or a template parameter), why `decltype(cmp)` is how you name the type, and why a captureless lambda converts to a plain function pointer but a capturing one cannot. Calling the lambda calls `operator()`, which is `const` by default: a by-value capture is read-only inside the body unless the lambda is `mutable`. A captureless lambda has no members at all, and the compiler inlines its call into the algorithm as if you had written the loop body.

Capture defaults

Capture	Meaning
`[]`	nothing — only parameters and globals are visible
`[x]`, `[&x]`	`x` by copy, `x` by reference
`[=]`	everything the body uses, by copy
`[&]`	everything the body uses, by reference
`[=, &total]`	by copy, except `total` by reference
`[&, x]`	by reference, except `x` by copy
`[this]`, `[*this]`	the enclosing object by pointer, by copy (C++17)

A default captures only the variables the body actually names, so `[=]` in a function with twenty locals copies the two it uses. Capture by reference is right when the lambda is called before the scope ends — a comparator passed to `std::sort`, a predicate to `count_if`, a body handed to `std::for_each`. Capture by copy is mandatory when the lambda **outlives the scope**: it is returned, stored in a container or a member, or handed to a thread (Module 17). A `[&]` lambda returned from a function refers to locals that no longer exist, and calling it is undefined behaviour with no diagnostic:

```
auto makeGreeter(std::string name) {
    return [&]() { return "hello " + name; };   // dangling: name dies with th
}
```

The fix is `[name]`, or `[name = std::move(name)]` to avoid the copy.

Init-captures

```
auto counter = [n = 0]() mutable { return ++n; };
auto owner   = [buf = std::move(bigVector)]() { return buf.size(); };
auto scaled  = [k = factor * 2](int x) { return x * k; };
```

An init-capture `[name = expression]` declares a new member initialised from any expression: a fresh variable the closure owns (`n = 0`), a moved-in resource (a `std::unique_ptr` or a large vector that would otherwise be copied), or a precomputed value. It is how a lambda carries state without a surrounding class, and the only way to move

something into a closure. A closure holding a move-only member is itself move-only: it works with `auto` and templates but cannot be stored in a `std::function`, which requires a copyable callable (C++23 adds `std::move_only_function` — reading only on this runtime).

`mutable`, and what copying copies

```
auto next = [n = 0]() mutable { return ++n; };
auto other = next;                              // a copy of the closure, w
next(); next();
std::cout << next() << ' ' << other() << '\n';    // 3 1
```

`mutable` removes the `const` from `operator()` so by-value captures can change. The state lives in the object, and copying the object copies the state: `other` starts where `next` was at the moment of the copy and moves independently. Passing a `mutable` lambda by value to an algorithm passes a copy, so the caller's counter never advances — which is why `std::for_each` returns its function object. The captured variable itself is never touched: `int x = 1; auto f = [x]() mutable { return ++x; };` leaves the outer `x` at 1 however often `f` runs.

Capturing `this`

Inside a member function `[this]` captures the object's address, so the body can name members directly; `[*this]` copies the whole object into the closure. The pointer is cheap and dangles when the object is destroyed before the lambda runs — a callback registered by a member function and fired after the object died. C++20 deprecated the implicit `this` that `[=]` used to include: write `[=, this]` or `[=, *this]` and say which you mean.

Generic lambdas and immediately invoked lambdas

```
auto printAll = [](const auto& c) { for (const auto& x : c) std::cout << x <<
auto sum = []<typename T>(const std::vector<T>& v) { return std::accumulate(v.

const std::vector<int> table = [] {
    std::vector<int> t(10);
    std::iota(t.begin(), t.end(), 1);
    return t;
}();
```

An `auto` parameter makes `operator()` a template — one lambda for every element type, the same generic comparator for a `std::vector<int>` and a `std::vector<std::string>`. C++20 adds the explicit template form `[]<typename T>(…)` for when you need the type by name. An **immediately invoked** lambda — defined and called in one expression — initialises a `const` variable that takes several statements to compute, keeping it `const` where an `if` / `else` assignment could not.

`std::function`

```
#include <functional>

std::function<long long(long long, long long)> op = [](long long a, long long
op = std::multiplies<>{};                          // any callable with that
std::map<std::string, std::function<void()>> commands;
```

Every lambda is its own type, so a container cannot hold "some lambdas" — but it can hold `std::function<R(Args...)>`, a **type-erasing** wrapper that stores any callable with a compatible signature: a capturing lambda, a function pointer, a functor. That flexibility has a price: the wrapper is a few pointers wide, a callable too large for its small internal buffer is allocated on the heap, every call goes through an indirect jump the optimiser cannot inline, and copying it copies the stored callable. Calling an empty `std::function` throws `std::bad_function_call`. Use it for what it is for — dispatch tables, callbacks stored in objects, plug-in points chosen at run time — and use `auto` or a template parameter when the callable is passed straight down and called: `std::sort(…, cmp)` takes the comparator by template and pays nothing.

Recursive lambdas

```
std::function<long long(int)> fact = [&fact](int n) → long long {
    return n ⩽ 1 ? 1 : n * fact(n - 1);
};

auto fib = [](auto&& self, int n) → long long {
    return n < 2 ? n : self(self, n - 1) + self(self, n - 2);
};
std::cout << fib(fib, 30) << '\n';                 // 832040
```

`auto f = [&f](…)` fails to compile — `f` is used in its own initialiser before its type has been deduced. Two idioms work. Naming the variable `std::function` gives it a type before the body, at the cost of a `std::function` call per level. A generic lambda that receives *itself* as its first parameter recurses through `self(self, …)` with no wrapper and no overhead; the explicit return type is required because the body refers to the call before deduction could finish. C++23 makes the second idiom official as `[](this auto self, int n)` — "deducing `this`" — which this runtime does not compile.

Lambdas as comparators in containers

```
auto byLength = [](const std::string& a, const std::string& b) {
    return a.size() ≠ b.size() ? a.size() < b.size() : a < b;
};
std::set<std::string, decltype(byLength)> words(byLength);   // pass the obje
std::set<std::string, decltype(byLength)> more;              // C++20: a captu

auto cmp = [](int a, int b) { return a > b; };               // a min-heap: th
std::priority_queue<int, std::vector<int>, decltype(cmp)> pq(cmp);
```

A container's comparator is a template parameter, so a lambda's type is named with `decltype`. Before C++20 the object had to be handed to the constructor; since C++20 a captureless closure type is default-constructible, so the container can create its own. A lambda with captures still needs the constructor argument, and a `std::function` comparator would pay the indirect call on every comparison — the case where a small functor class (Module 11, lesson 4) is the better tool.

Pitfalls

Mistake	What happens
Returning or storing a `[&]` lambda	Dangling references: undefined behaviour when it is called
`++x` in a non-`mutable` lambda that captured `x` by value	Compile error: `operator()` is `const`
Expecting a `mutable` lambda's state to reach the caller	The algorithm got a copy; the original is untouched
`[=]` that uses members, in C++20	Deprecated implicit `this`; write `[=, this]`
`std::function` in a tight loop	Indirect call, no inlining — prefer `auto` or a template
`auto f = [&f]…`	`f` used before deduction; use `std::function` or the `self` idiom

Key takeaways

A lambda is an object of a unique closure type; captures are its members and the call is `operator()`, `const` unless `mutable`.

`[&]` only for lambdas called before the scope ends; `[=]` or init-captures for anything that outlives it.

Init-captures own state and move resources in; a copy of the closure is an independent copy of that state.

`std::function` erases the type for storage and dispatch and costs an indirect call; `auto`/templates when the callable is only passed and called.

Recursion: a `std::function` named before its body, or a generic lambda that receives itself.

Name a lambda's type with `decltype` for containers; captureless ones default-construct since C++20.

EXERCISE 1

A sequence counter

Write `makeCounter(int start, int step)`, a function that returns a lambda. The lambda must own its state through an **init-capture** — `[value = start, step]` — be `mutable`, and on each call return the current value and then advance it by `step`. No variable outside the lambda may hold the count.

Read `start step` from the first line, then lines of text until the input ends (ignore empty lines). Print each line prefixed with the counter's next value in square brackets, then a final line `next=<value>` — one more call to the same counter object.

Input: `start step`, then zero or more lines. **Output:** one line per input line, then `next=<value>`.

```
100 5
boot
load config
ready
```

prints

```
[100] boot
[105] load config
[110] ready
next=115
```

INPUT 1

```
100 5
boot
load config
ready
```

OUTPUT 1

```
[100] boot
[105] load config
[110] ready
next=115
```

INPUT 2

```
1 1
only
```

OUTPUT 2

```
[1] only
next=2
```

REFERENCE SOLUTION

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

auto makeCounter(int start, int step) {
    return [value = start, step]() mutable {
        const int current = value;
        value += step;
        return current;
    };
}

int main() {
    int start, step;
    std::cin >> start >> step;
    std::cin.ignore();
    auto next = makeCounter(start, step);
    std::string line;
    while (std::getline(std::cin, line)) {
        if (line.empty()) continue;
        std::cout << '[' << next() << "] " << line << '\n';
    }
    std::cout << "next=" << next() << '\n';
    return 0;
}
```

EXERCISE 2

A stack machine with a command table

Build a tiny stack machine whose commands live in a `std::map<std::string, std::function<void()>>`. The handlers are lambdas that capture the stack (a `std::vector<long long>`) by reference; the `push` handler reads its own integer argument from `std::cin`. Read words with `std::cin >>` until the input ends and look each one up in the table:

- `push <x>` — push the integer `x`
- `pop` — discard the top
- `add` / `mul` — pop two values and push their sum / product
- `neg` — negate the top
- `print` — print the top on its own line

A command that needs more values than the stack holds prints `error: stack underflow` and leaves the stack unchanged. A word that is not in the table prints `unknown: <word>` (such words carry no argument).

Input: commands until end of input. **Output:** whatever `print` and the errors produce.

```
push 3
push 4
add
print
push 5
mul
print
```

prints

```
7
35
```

INPUT 1

```
push 3
push 4
add
print
push 5
mul
print
```

OUTPUT 1

```
7
35
```

INPUT 2

```
print
pop
foo
push 2
neg
print
```

OUTPUT 2

```
error: stack underflow
error: stack underflow
unknown: foo
-2
```

REFERENCE SOLUTION

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

int main() {
    std::vector<long long> stack;
    const auto need = [&stack](std::size_t count) {
        if (stack.size() >= count) return true;
        std::cout << "error: stack underflow\n";
        return false;
    };
    const auto pop = [&stack]() {
        const long long top = stack.back();
        stack.pop_back();
        return top;
    };

    std::map<std::string, std::function<void()>> commands;
    commands["push"] = [&stack]() {
        long long x;
        std::cin >> x;
        stack.push_back(x);
    };
    commands["pop"] = [&]() {
        if (need(1)) pop();
    };
    commands["add"] = [&]() {
        if (need(2)) {
            const long long b = pop();
            const long long a = pop();
            stack.push_back(a + b);
        }
    };
    commands["mul"] = [&]() {
        if (need(2)) {
            const long long b = pop();
            const long long a = pop();
            stack.push_back(a * b);
        }
    };
    commands["neg"] = [&]() {
        if (need(1)) stack.back() = -stack.back();
    };
    commands["print"] = [&]() {
        if (need(1)) std::cout << stack.back() << '\n';
    };
```

```
    std::string word;
    while (std::cin >> word) {
        const auto it = commands.find(word);
        if (it == commands.end()) {
            std::cout << "unknown: " << word << '\n';
        } else {
            it->second();
        }
    }
    return 0;
}
```

QUIZ

1. What does this print?

```
int x = 1;
auto f = [x]() mutable { return ++x; };
f();
f();
std::cout << x << f() << '
';
```

A. `13`

B. `33`

C. `14`

D. Compile error

2. A function returns `[&]() { return total * 2; }` where `total` is one of its local variables. What happens when the returned lambda is called?

A. It returns twice the last value of `total`

B. Undefined behaviour: the closure holds a reference to a local that no longer exists

C. Compile error: a lambda cannot be returned

D. The lambda keeps `total` alive

3. `std::function<int(int)> f; std::cout << f(2);` — what happens?

A. Prints `0`

B. Prints `2`

C. Throws `std::bad_function_call`

D. Compile error

4. Which situation genuinely needs `std::function` rather than `auto` or a template parameter?

A. Passing a comparator to `std::sort`

B. Storing handlers with different capture lists in one `std::map<std::string, ...>` of callbacks

C. Calling a generic lambda with two different argument types

D. Capturing a variable by value

5. `auto fact = [&fact](int n) → int { return n ⩽ 1 ? 1 : n * fact(n - 1); };` — what happens?

A. It compiles and computes factorials

B. Compile error: `fact` is used in its own initialiser before its type is deduced

C. Stack overflow at run time

D. It compiles but always returns 1

6. A lambda is written `[p = std::make_unique<int>(5)]() { return *p; }`. Which statement is correct?

A. Compile error: a `std::unique_ptr` cannot be captured

B. It compiles; the closure is move-only, so it works with `auto` but cannot be stored in a `std::function`

C. It compiles and the closure is freely copyable

D. Undefined behaviour: the pointer dangles

7. Why write `const std::vector<int> table = [] { std::vector<int> t(10); std::iota(t.begin(), t.end(), 1); return t; }();`?

A. It is the only way to call `std::iota`

B. The immediately invoked lambda lets a variable that needs several statements to build be `const` from its first moment

C. It runs at compile time

D. It avoids allocating the vector

ANSWER KEY · LAMBDAS IN DEPTH — CAPTURES, CLOSURES, STD::FUNCTION AND RECURSION

1. C The closure holds its own copy of `x`, which `mutable` lets it change: the two calls take the copy to 2 and 3, and the third call returns 4. The outer `x` is never touched and prints 1 — since C++17 a chain of `<<` evaluates left to right, so the output is `14`. `33` would need the capture to be by reference.

2. B `[&]` stores references to the enclosing locals, and those locals are destroyed when the function returns. Nothing keeps them alive; calling the lambda reads a dead object. Capture by copy — `[total]` or `[t = total]` — for any lambda that outlives its scope.

3. C A default-constructed `std::function` is empty, and calling an empty one throws `std::bad_function_call`. Test with `if (f)` before calling a callback that may not have been set.

4. B Every lambda has its own type, so a container needs one common type to hold several of them — that is type erasure, and `std::function` provides it. A comparator passed straight into `std::sort` is deduced as a template parameter and pays no indirection.

5. B A variable declared `auto` has no type until its initialiser is complete, and the initialiser here names the variable. Declare it as `std::function<int(int)>`, or write a generic lambda that receives itself: `[](auto&& self, int n) → int { ... self(self, n - 1) ... }`.

6. B An init-capture moves the `unique_ptr` into the closure, which therefore has a move-only member and is itself move-only. `std::function` requires a copyable callable, so storing it there fails to compile; C++23's `std::move_only_function` exists for this and is reading only on this runtime.

7. B A `const` variable must be fully initialised in its declaration, and an `if` / `else` or a loop cannot appear there. Wrapping the steps in a lambda and calling it at once yields the finished value in one expression; the vector is still built at run time.

LESSON 5 · 15 MIN

Ranges and views — C++20 pipelines

Every algorithm so far took two iterators, and every pipeline needed an intermediate vector per step. C++20's `<ranges>` fixes both. The constrained algorithms in `std::ranges::` take the container itself and a **projection** that says which part of each element to look at; the **views** in `std::views::` are lazy, non-owning adaptors that compose with `|` into a pipeline that computes nothing until it is walked. This lesson settles what each brings, how a pipeline is materialised into a vector on a C++20 runtime, and the handful of rules — caching, `const`, re-evaluation, lifetime — that keep views honest.

Constrained algorithms and projections

```
#include <algorithm>
#include <ranges>

std::vector<int> v{5, 3, 8, 1};
std::ranges::sort(v);                                          // whole co
auto it = std::ranges::find(v, 8);                             // an itera
auto n  = std::ranges::count_if(v, [](int x) { return x > 2; });
```

`std::ranges::sort(v)` is `std::sort(v.begin(), v.end())` with the iterators taken for you and, crucially, with **concepts** (Module 12, lesson 5) on the parameters: passing a `std::list` fails with a constraint error that names `random_access_iterator`, not with forty lines of template noise. The iterator-pair forms still exist, and the end may be a *sentinel* of a different type from the iterator. Return values changed in a few places — `std::ranges::minmax(v)` returns a struct with `.min` and `.max` *values*, `std::ranges::minmax_element` the same with iterators, and `std::ranges::sort` returns the end of the range.

The extra parameter is the **projection**: a callable applied to each element before the comparator or predicate sees it.

```
struct Person { std::string name; int age; };
std::vector<Person> people = readPeople();
std::ranges::sort(people, {}, &Person::age);
std::ranges::sort(people, std::ranges::greater{}, &Person::age);
auto adults = std::ranges::count_if(people, [](int a) { return a ⩾ 18; }, &Pe
auto ann    = std::ranges::find(people, std::string{"ann"}, &Person::name);
std::ranges::sort(people, {}, [](const Person& p) { return std::tie(p.age, p.r
```

`{}` is the default comparator `std::ranges::less`; a pointer-to-member is the commonest projection, and a lambda returning a `std::tie` is how to sort by a key with a tie-break. The comparator sees `int`s, the predicate sees `int`s, `find` compares a `std::string` — each part of the call does one job, where the classic lambda comparator did three.

Views

```
std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
auto evens    = v | std::views::filter([](int x) { return x % 2 == 0; });
auto squares  = evens | std::views::transform([](int x) { return x * x; });
auto firstTwo = squares | std::views::take(2);
for (int x : firstTwo) std::cout << x << ' ';
```

A **view** is a lightweight object that refers to a range and presents it differently. Nothing is computed when the view is created: `filter` runs its predicate and `transform` its function only when an element is *asked for*, by the `for` loop or an algorithm — **laziness**. A view is non-owning (it holds the vector by reference, so the vector must outlive it), cheap to copy, and composed with `|`: the pipeline `v | filter | transform | take` reads left to right, with no intermediate containers, and stops after two elements because `take` never asks for a third.

Adaptor	Yields
`views::filter(pred)`	the elements for which `pred` is true
`views::transform(f)`	`f(element)` for each element
`views::take(n)` / `views::drop(n)`	the first `n` / everything after the first `n`
`views::take_while(pred)` / `views::drop_while(pred)`	a prefix / the rest, decided by `pred`
`views::reverse`	the range backwards (bidirectional or better)
`views::iota(a, b)` / `views::iota(a)`	the integers in `[a, b)` / from `a` without end
`views::keys` / `views::values`	the `.first` / `.second` of each pair — a map's keys

```
for (int i : std::views::iota(1, 6)) std::cout << i;
auto odds = std::views::iota(1) | std::views::filter([](int x) { return x % 2;
                                | std::views::take(3);
for (const auto& k : ages | std::views::keys) std::cout << k << ' ';
for (int x : v | std::views::reverse | std::views::drop(1)) std::cout << x;
```

`views::iota(1)` with no upper bound is infinite, and the pipeline still terminates because `take(3)` stops pulling — the clearest demonstration that a view is a *recipe*, not a result. `std::ranges::for_each`, `count_if`, `find`, `max_element` and the rest accept a view wherever they accept a container.

Materialising a view

A view is not a vector: it has no `size()` unless every adaptor preserves one (`filter` does not), cannot be indexed unless random access survives, and cannot be returned from a function that lets the source die. To keep the results, copy them out:

```cpp
#include <iterator>

std::vector<long long> out;
std::ranges::copy(v | std::views::filter(positive) | std::views::transform(squ
                  std::back_inserter(out));
```

`std::ranges::copy` into a `std::back_inserter` is the C++20 answer. The iterator-pair constructor `std::vector<int>(r.begin(), r.end())` does **not** compile for a pipeline like this: `filter` and `take` produce an `end()` that is a sentinel of a different type from `begin()`, and the constructor wants two of the same. `r | std::views::common` bridges the two when an old interface insists on an iterator pair. C++23 adds `std::ranges::to<std::vector>()` at the end of a pipeline, plus `std::views::enumerate`, `zip`, `chunk` and `adjacent` — worth recognising in modern code and in interviews, but this track's runtime is C++20 and none of them compile here: `ranges::copy` and `back_inserter` are the tools.

What to keep in mind

`filter` caches `begin()`. Finding the first passing element costs a scan, so `filter_view` remembers where it is. Two consequences: the view's `begin()` is not `const`, so a `const auto evens = v | std::views::filter(…)` cannot be iterated (compile error), and changing the underlying vector after the first pass leaves the cache stale. Build the view, walk it, and do not mutate the source in between.

`transform` recomputes. The function runs every time an element is read — walk a `transform` view twice and pay twice; index it in a loop and pay per access. Materialise when the function is expensive or the result is read more than once.

Views dangle like references. A view built from an lvalue holds a reference to it: return `v | std::views::filter(…)` from the function that owns `v`, or keep the view in a member after the vector is gone, and the view refers to a dead object — a bug of exactly Module 6's kind. (A pipeline started from an rvalue container, `getVector() | std::views::filter(…)`, moves the container into an `owning_view` and is safe.) When you must return the data, return a vector.

Adaptors lose capabilities. `filter` drops random access and `size()`; `reverse` needs a bidirectional source; `transform` keeps the category but, when its function returns by value, yields temporaries you cannot assign through. When an algorithm refuses a view, the concept named in the error is the missing capability.

What ranges fix

The classic `std::sort(v.begin(), v.end(), [](const Person& a, const Person& b) { return a.age < b.age; })` becomes `std::ranges::sort(people, {}, &Person::age)`; the filter–map–reduce pipeline of lesson 3, with its two intermediate vectors, becomes one expression with none; "the first k elements that match" stops being a loop with a counter; and an infinite sequence — every odd number, every power of two — is expressible at all, because `take` and `take_while` decide when to stop. What stays the same is the algorithm underneath: `std::ranges::sort` is introsort, `std::ranges::find` is a linear scan, and every complexity in lesson 2's table holds. Ranges change how a call is *written*, not what it costs.

Pitfalls

Mistake	What happens
Iterating a `const` `filter_view`	Compile error — `begin()` is non-`const` because it caches
`std::vector<int>(r.begin(), r.end())` on a filtered pipeline	Compile error — iterator and sentinel differ; use `ranges::copy` or `views::common`
Ending a pipeline in `std::ranges::to<std::vector>()`	C++23; does not compile on this runtime
A view of a local that goes out of scope	Dangling — an lvalue source must outlive the view
Walking an expensive `transform` twice	The function runs twice per element
Mutating the source after a `filter`'s first pass	Stale cached `begin()`
Passing a `std::list` to `std::ranges::sort`	Constraint failure: not `random_access_iterator`

Key takeaways

- `std::ranges::` algorithms take the container, check it with concepts, and accept a projection: `sort(v, {}, &T::member)`.

Views are lazy, non-owning recipes composed with `|`; `filter`, `transform`, `take`, `drop`, `reverse` and `iota` cover most pipelines.

Nothing runs until the view is walked, so `take` on an infinite `iota` terminates.

Materialise with `std::ranges::copy(view, std::back_inserter(out))` — `ranges::to` is C++23 and unavailable here.

`filter` caches and is not `const`-iterable; `transform` recomputes on every read; a view must not outlive its source.

Ranges change the spelling, not the cost: the same algorithms and complexities sit underneath.

EXERCISE 1

First k squares of the positives

Read an integer `n`, then `n` integers, then an integer `k`. Build **one** pipeline over the vector: `std::views::filter` keeps the strictly positive values, `std::views::transform` squares each as a `long long`, and `std::views::take` keeps the first `k`. Materialise it with `std::ranges::copy` into a `std::vector<long long>` through `std::back_inserter` (`std::ranges::to` is C++23 and not available here).

Print the vector's values on one line separated by single spaces, or `none` when it is empty; then `sum=<total>` from `std::accumulate` with `0LL`.

Input: `n`, the values, `k`. **Output:** two lines.

Example: `7` then `3 -1 4 0 5 -9 2` then `3` →

```
9 16 25
sum=50
```

INPUT 1

```
7
3 -1 4 0 5 -9 2
3
```

OUTPUT 1

```
9 16 25
sum=50
```

INPUT 2

```
4
-2 -3 -4 -5
2
```

OUTPUT 2

```
none
sum=0
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <numeric>
#include <ranges>
#include <vector>

int main() {
    int n;
    std::cin >> n;
    std::vector<int> values(n);
    for (int& v : values) std::cin >> v;
    int k;
    std::cin >> k;

    auto pipeline = values
        | std::views::filter([](int x) { return x > 0; })
        | std::views::transform([](int x) { return static_cast<long long>(x) * x; })
        | std::views::take(k);
    std::vector<long long> squares;
    std::ranges::copy(pipeline, std::back_inserter(squares));

    if (squares.empty()) {
        std::cout << "none\n";
    } else {
        for (std::size_t i = 0; i < squares.size(); ++i) {
            if (i > 0) std::cout << ' ';
            std::cout << squares[i];
        }
        std::cout << '\n';
    }
    std::cout << "sum=" << std::accumulate(squares.begin(), squares.end(), 0LL) << '
    return 0;
}
```

EXERCISE 2

Youngest and oldest by projection

Read an integer `n`, then `n` lines of `name age` (distinct single-word names, integer ages), then an integer `k`. Sort the people with `std::ranges::sort` and a **projection** — a lambda returning `std::tie(p.age, p.name)` so that equal ages are ordered by name — with the default comparator (`{}`), not a comparator lambda. Then print `youngest` followed by the first `k` people through `people | std::views::take(k)`, and `oldest` followed by the first `k` of `people | std::views::reverse`. When `k` exceeds `n`, each list is everyone.

Input: `n`, then `n` lines, then `k`. **Output:** `youngest`, up to `k` lines of `name age`, `oldest`, up to `k` lines.

```
5
ann 19
zed 71
bob 23
yan 65
max 40
2
```

prints

```
youngest
ann 19
bob 23
oldest
zed 71
yan 65
```

INPUT 1

```
5
ann 19
zed 71
bob 23
yan 65
max 40
2
```

OUTPUT 1

```
youngest
ann 19
bob 23
oldest
zed 71
yan 65
```

INPUT 2

```
3
c 30
a 30
b 30
2
```

OUTPUT 2

```
youngest
a 30
b 30
oldest
c 30
b 30
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <iostream>
#include <ranges>
#include <string>
#include <tuple>
#include <vector>

struct Person {
    std::string name;
    int age;
};

int main() {
    int n;
    std::cin >> n;
    std::vector<Person> people(n);
    for (auto& p : people) std::cin >> p.name >> p.age;
    int k;
    std::cin >> k;

    std::ranges::sort(people, {}, [](const Person& p) { return std::tie(p.age, p.nam

    std::cout << "youngest\n";
    for (const Person& p : people | std::views::take(k)) {
        std::cout << p.name << ' ' << p.age << '\n';
    }
    std::cout << "oldest\n";
    for (const Person& p : people | std::views::reverse | std::views::take(k)) {
        std::cout << p.name << ' ' << p.age << '\n';
    }
    return 0;
}
```

QUIZ

1. In `std::ranges::sort(people, {}, &Person::age)`, what are the second and third arguments?

A. The begin and end iterators

B. The default comparator (`std::ranges::less`) and a projection applied to each element before comparing

C. A comparator and an execution policy

D. An empty initialiser list and a member pointer that selects which members to sort

2. `v` has ten elements and `f` is expensive. After `auto r = v | std::views::transform(f);` how many times has `f` run?

A. 10

B. 0

C. 1

D. 20

3. What does this print?

```
auto r = std::views::iota(1) | std::views::filter([](int x) { return x % 2 == 1
for (int x : r) std::cout << x << ' ';
```

A. `1 3 5`

B. It never terminates: `iota(1)` is infinite

C. `1 2 3`

D. Compile error: `iota` needs an upper bound

4. `r` is `v | std::views::filter(pred) | std::views::take(k)`. Which line copies its elements into a `std::vector<int> out` on this C++20 runtime?

A. `auto out = r | std::ranges::to<std::vector>();`

B. `std::vector<int> out(r.begin(), r.end());`

C. `std::vector<int> out; std::ranges::copy(r, std::back_inserter(out));`

D. `std::vector<int> out = r;`

5. Does this compile?

```
const auto evens = v | std::views::filter([](int x) { return x % 2 == 0; });
for (int x : evens) std::cout << x;
```

A. Yes, it prints the even elements

B. No: a `filter_view`'s `begin()` is not `const`, because it caches the first match

C. No: views cannot be range-for iterated

D. Yes, but it prints nothing

6. Which of these is C++23 and therefore **unavailable** on this track's C++20 runtime?

A. `std::views::drop`

B. `std::views::enumerate`

C. `std::views::iota`

D. `std::views::take_while`

7. What does a ranges pipeline change about the cost of an algorithm?

A. `std::ranges::sort` is O(n) because it uses a projection

B. Nothing — the same algorithms and complexities sit underneath; the pipeline removes intermediate containers and changes the spelling

C. Views make every algorithm lazy, including `sort`

D. Ranges algorithms run in parallel by default

ANSWER KEY · RANGES AND VIEWS — C++20 PIPELINES

1. B Ranges algorithms take the range, then a comparator (`{}` means the default `std::ranges::less`), then a projection. `&Person::age` is invoked on each element, so the comparator compares ages while the whole `Person` objects are moved.

2. B Views are lazy: creating one stores the function and a reference to the source, nothing more. `f` runs only when an element is read — ten times for one full walk, twenty for two, and never if the view is discarded.

3. A `iota(1)` is unbounded, but `take(3)` stops asking after three elements, and the filter yields only the odd ones on the way. Laziness is what makes an infinite source usable.

4. C `ranges::to` is C++23 and does not compile here. The iterator-pair constructor fails because `take` over `filter` has a sentinel `end()` whose type differs from `begin()`; a view does not convert to a vector. `std::ranges::copy` into a `back_inserter` walks the view once and pushes each element.

5. B `filter_view` remembers where its first passing element is, so `begin()` mutates the view and is declared non-`const`. A `const` filter view has no usable `begin()`, and the loop fails to compile. Drop the `const`.

6. B `enumerate`, `zip`, `chunk`, `adjacent` and `ranges::to` arrived in C++23. `drop`, `iota` and `take_while` are C++20 and compile here.

7. B `std::ranges::sort` is still introsort and `std::ranges::find` is still a linear scan. What ranges save is the copies between steps and the boilerplate; a view is lazy, but an algorithm such as `sort` still does its whole job when called.

CHECKPOINT · 25 MIN

Checkpoint — Algorithms and lambdas

This checkpoint covers the whole module: the half-open range and predicate interface of `<algorithm>` and `<numeric>`, sorting with comparators and stability plus binary search on sorted ranges, `transform`, `accumulate`, `remove_if` and `partition`, lambdas as closure objects with captures, `mutable` and `std::function`, and C++20 ranges with projections and lazy views.

How it works. Fourteen 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:

- What does `std::find` return when nothing matches, and what must you test before dereferencing it?
- Why is a comparator written with `≤` undefined behaviour rather than a wrong order?
- When must you use `std::stable_sort` instead of `std::sort`?
- What does `std::accumulate(v.begin(), v.end(), 0)` compute over a `std::vector<long long>`?
- Why does `std::remove_if` need an `erase` after it?
- What does copying a `mutable` lambda copy?
- Why does `std::views::iota(1) | std::views::take(5)` terminate?

The three programs are a task list ordered by `std::stable_sort` that must keep arrival order within a priority, a median found with `std::nth_element` followed by k-th smallest queries, and a nested-list walker written as a recursive lambda. Read each input format before choosing the algorithm.

EXERCISE 1

Tasks by priority, arrival order kept

Read an integer `n`, then `n` lines of `task priority` (single-word task names, integer priorities). Order the tasks by priority **descending** with `std::stable_sort`, so that tasks of equal priority keep the order in which they were read — the comparator compares priority only. Print each task as `<task> (<priority>)`.

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

```
5
write 2
test 3
lint 2
deploy 3
docs 1
```

prints

```
test (3)
deploy (3)
write (2)
lint (2)
docs (1)
```

INPUT 1

```
5
write 2
test 3
lint 2
deploy 3
docs 1
```

OUTPUT 1

```
test (3)
deploy (3)
write (2)
lint (2)
docs (1)
```

INPUT 2

```
3
a 1
b 1
c 1
```

OUTPUT 2

```
a (1)
b (1)
c (1)
```

REFERENCE SOLUTION

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

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

int main() {
    int n;
    std::cin >> n;
    std::vector<Task> tasks(n);
    for (auto& t : tasks) std::cin >> t.name >> t.priority;

    std::stable_sort(tasks.begin(), tasks.end(),
                     [](const Task& a, const Task& b) { return a.priority > b.priori

    for (const Task& t : tasks) std::cout << t.name << " (" << t.priority << ")\n";
    return 0;
}
```

EXERCISE 2

Median without sorting

Read an integer `n` (at least 1), then `n` integers, then an integer `q`, then `q` integers `k` (each between 1 and `n`). Print the median with `std::nth_element` — no full sort: for odd `n` it is the element that lands at index `n / 2`; for even `n` it is the mean of that element and the largest element before it (`std::max_element` over the left part, which `nth_element` guarantees holds nothing greater). Print it with one decimal place (`std::fixed`, `std::setprecision(1)`). Then, for each `k`, print the `k`-th smallest value using `std::nth_element` again.

Input: `n`, the values, `q`, the queries. **Output:** `median=<value>`, then `q` lines.

Example: `5` then `7 1 9 3 5` then `2` then `1` and `4` →

```
median=5.0
1
7
```

INPUT 1

```
5
7 1 9 3 5
2
1
4
```

OUTPUT 1

```
median=5.0
1
7
```

INPUT 2

```
4
10 2 8 4
1
2
```

OUTPUT 2

```
median=6.0
4
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    std::vector<int> values(n);
    for (int& v : values) std::cin >> v;

    const auto middle = values.begin() + n / 2;
    std::nth_element(values.begin(), middle, values.end());
    double median = *middle;
    if (n % 2 == 0) {
        const int lower = *std::max_element(values.begin(), middle);
        median = (static_cast<double>(lower) + *middle) / 2.0;
    }
    std::cout << std::fixed << std::setprecision(1) << "median=" << median << '\n';

    int q;
    std::cin >> q;
    for (int i = 0; i < q; ++i) {
        int k;
        std::cin >> k;
        const auto kth = values.begin() + (k - 1);
        std::nth_element(values.begin(), kth, values.end());
        std::cout << *kth << '\n';
    }
    return 0;
}
```

EXERCISE 3

Walk a nested list with a recursive lambda

The input is one nested list written as space-separated tokens: `[` opens a list, `]` closes it, and every other token is an integer. Example: `[ 1 [ 2 3 ] [ 4 [ 5 ] ] ]`. Walk it with a **recursive lambda** — a generic lambda that receives itself as its first parameter (`auto&& self`) with an explicit `-> void` return type, or a `std::function` — reading tokens with `std::cin >>`: an integer is added to the running sum and counted, a `[` recurses one level deeper, a `]` returns. Report the sum (in `long long`), the number of integers, and the deepest nesting, where the outermost list is depth 1.

Input: one line. **Output:** `sum=<total>`, `count=<integers>`, `depth=<max depth>`.

Example: `[ 1 [ 2 3 ] [ 4 [ 5 ] ] ]` →

```
sum=15
count=5
depth=3
```

INPUT 1

```
[ 1 [ 2 3 ] [ 4 [ 5 ] ] ]
```

OUTPUT 1

```
sum=15
count=5
depth=3
```

INPUT 2

```
[ 10 20 30 ]
```

OUTPUT 2

```
sum=60
count=3
depth=1
```

REFERENCE SOLUTION

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

int main() {
    long long sum = 0;
    int count = 0;
    int maxDepth = 0;

    auto walk = [&](auto&& self, int depth) -> void {
        maxDepth = std::max(maxDepth, depth);
        std::string token;
        while (std::cin >> token && token != "]") {
            if (token == "[") {
                self(self, depth + 1);
            } else {
                sum += std::stoll(token);
                ++count;
            }
        }
    };

    std::string opening;
    std::cin >> opening;          // the outermost "["
    walk(walk, 1);

    std::cout << "sum=" << sum << '\n';
    std::cout << "count=" << count << '\n';
    std::cout << "depth=" << maxDepth << '\n';
    return 0;
}
```

CHECKPOINT QUESTIONS

1. `auto it = std::find_if(v.begin(), v.end(), pred);` — what must happen before `*it` is used?

A. Nothing; `find_if` always finds something

B. Compare `it` with `v.end()`; dereferencing `end()` is undefined behaviour

C. Check `it != nullptr`

D. Call `it.valid()`

2. `std::max_element` and `std::minmax_element` both run on `{2, 9, 9, 1}`. Which indices do they report for the maximum?

A. Both report 1

B. Both report 2

C. `max_element` reports 1 (the first 9); `minmax_element` reports 2 (the last 9)

D. `max_element` reports 2; `minmax_element` reports 1

3. Why is `std::max(1, 2.5)` a compile error?

A. `std::max` only accepts integers

B. Template argument deduction finds `T = int` from one argument and `T = double` from the other

C. `2.5` cannot be converted to `int`

D. `std::max` needs three arguments

4. A comparator for `std::sort` returns `a.score ⩾ b.score`. What is the consequence?

A. A correct descending sort

B. Undefined behaviour: `⩾` is not a strict weak ordering, and equal scores can make the sort read past the end

C. A compile error

D. The sort is stable

5. What is the average complexity of `std::nth_element` on `n` elements, and what does it guarantee afterwards?

A. O(n log n); the whole range is sorted

B. O(n); the nth position holds the sorted value, everything before is no greater, everything after no smaller

C. O(log n); it returns the value without moving anything

D. O(n²); only the nth position is correct

6. `v` is sorted ascending. `std::upper_bound(v.begin(), v.end(), 10) - std::lower_bound(v.begin(), v.end(), 5)` computes…

A. The number of elements in the closed interval `[5, 10]`

B. The number of elements strictly between 5 and 10

C. The index of the first 10

D. Undefined behaviour

7. `std::vector<long long> v{3000000000, 3000000000}; auto s = std::accumulate(v.begin(), v.end(), 0);` What is the type of the accumulator, and why?

A. `long long`, from the element type

B. `int`, from the initial value `0`

C. `double`, the widest available

D. `auto` — it is deduced from the largest element

8. Which single C++20 call removes every element matching `pred` from a `std::vector<int> v` — no erase needed afterwards?

A. `std::remove_if(v.begin(), v.end(), pred)`

B. `std::erase_if(v, pred)`

C. `v.remove_if(pred)`

D. `std::filter(v, pred)`

9. `std::any_of(v.begin(), v.end(), pred)` on an empty `v` returns…

A. `true`

B. `false`

C. Undefined behaviour

D. It throws

10. What does this print?

```
auto next = [n = 0]() mutable { return ++n; };
auto copy = next;
next(); next();
std::cout << next() << copy() << '
';
```

A. `33`

B. `31`

C. `34`

D. `11`

11. Calling a default-constructed `std::function<void()>` …

A. Does nothing

B. Throws `std::bad_function_call`

C. Is a compile error

D. Calls a default lambda

12. A member function stores `[&]() { use(counter); }` in a member for later; `counter` is a local of that member function. The stored lambda is called after the function returns. What is the result?

A. It uses the last value `counter` had

B. Undefined behaviour: `[&]` captured a reference to a local that has been destroyed

C. Compile error

D. The lambda keeps `counter` alive until it runs

13. After `auto r = std::views::iota(1) | std::views::transform(square) | std::views::take(5);` and before any loop, how much work has been done?

A. Five squares have been computed

B. None: views are lazy; `square` runs only as elements are read, and `take` stops the infinite `iota` after five

C. The whole infinite sequence is being computed in the background

D. Compile error: `iota` needs two arguments

14. Which line materialises a filtered pipeline `r` into a vector on this C++20 runtime?

A. `auto out = r | std::ranges::to<std::vector>();`

B. `std::vector<int> out(r.begin(), r.end());`

C. `std::vector<int> out; std::ranges::copy(r, std::back_inserter(out));`

D. `std::vector<int> out = std::vector(r);`

ANSWER KEY · CHECKPOINT — ALGORITHMS AND LAMBDAS

1. B An algorithm signals "no match" by returning the `last` iterator it was given. That iterator points past the end, so dereferencing it is undefined behaviour; the `≠ end()` test is the idiom.

2. C `max_element` returns the first of equal maxima; `minmax_element` returns the first minimum but the **last** maximum, a consequence of its pairwise comparison scheme.

3. B Both parameters are `const T&` with the same `T`, and the two arguments disagree on what `T` is. Supply it explicitly — `std::max<double>(1, 2.5)` — or make the literals agree.

4. B A strict weak ordering must be irreflexive, and `⩾` reports that an element precedes itself. Introsort's inner loops rely on the guarantee to stop; with equal elements they can run off the array. Use `>`.

5. B `nth_element` is a selection algorithm, linear on average, that partitions around the nth position rather than sorting. That partition property is what makes the median — and the lower middle for an even count — readable in linear time.

6. A `lower_bound(5)` is the first element ≥ 5 and `upper_bound(10)` the first element > 10; everything in between is in `[5, 10]`, and the iterator difference is the count.

7. B `accumulate` deduces its accumulator type from the third argument alone. `0` is an `int`, so both elements are added into an `int` and overflow; `0LL` is the one-character fix.

8. B `std::erase_if` (C++20, declared in `<vector>`) performs the erase–remove idiom in one call and returns the number of elements removed. `remove_if` alone only shifts survivors; `remove_if` as a member exists on `std::list`, not `std::vector`.

9. B No element satisfies the predicate because there is no element, so `any_of` is `false`; `all_of` and `none_of` are vacuously `true` on the same empty range.

10. B The closure's state is a data member, and copying the closure copies it: `copy` took a snapshot when `n` was 0. `next` advances to 3; `copy`'s first call returns 1.

11. B An empty `std::function` has no target; invoking it throws `std::bad_function_call`. Test `if (f)` first when a callback may be unset.

12. B A lambda stored past the enclosing scope must capture by copy. `[&]` leaves a reference to a dead stack variable, and nothing in C++ extends a local's lifetime for a closure.

13. B Building a pipeline stores adaptors, not results. Iterating `r` pulls five elements through `transform`, and `take` never asks the unbounded `iota` for a sixth.

14. C `ranges::to` is C++23. The iterator-pair constructor needs `begin()` and `end()` of the same type, and a filter's `end()` is a sentinel. Copying through `std::back_inserter` works on any input range.

CodeKairo · C++ study plan · Module 14: Algorithms and lambdas. Solve the exercises online to track your progress and earn XP.