Gorgo Continuing C++

April 11, 2026

Contents

4. Ranges, Algorithms, and Lambdas **2**
The <algorithm> Header . . . 2
Sorting . . . 2
Finding Elements . . . 2
Counting . . . 3
`for_each` . . . 3
Lambdas . . . 4
Transform . . . 5
Accumulate . . . 6
Min and Max . . . 6
Ranges (C++20) . . . 7
Views . . . 8
Try It: Algorithm Starter . . . 10
Key Points . . . 12
Exercises . . . 12

4. Ranges, Algorithms, and Lambdas

In *Gorgo Starting C++* you learned to store data in `std::vector` and `std::array` and to iterate through them with range-based for loops. But when you need to sort, search, or filter that data, you end up writing loops by hand. Hand-written loops are repetitive, easy to get wrong (off-by-one errors, forgotten edge cases), and they bury your intent behind boilerplate. Every time you write a loop to find the maximum element, you are re-implementing logic that has already been written and tested. The standard library provides **algorithms** — pre-built, well-tested functions for the operations you need most. Combined with **lambdas** (inline functions you pass as arguments) and C++20 **ranges**, they let you say *what* you want done instead of spelling out *how* to do it. In this chapter you will learn the most common algorithms, how to write lambdas to customize their behavior, and how ranges and views make composing operations cleaner.

The <algorithm> Header

The `<algorithm>` header provides dozens of functions that operate on containers. Most of them take a pair of iterators (remember `.begin()` and `.end()` from *Gorgo Starting C++*) to define the range of elements to work on.

Let's start with the most commonly used algorithms.

Sorting

`std::sort` arranges elements in ascending order:

```
void sort(Iterator first, Iterator last);
void sort(Iterator first, Iterator last, Compare comp);
```

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

int main()
{
    std::vector<int> scores = {88, 42, 95, 67, 73};

    std::sort(scores.begin(), scores.end());

    for (const auto& s : scores) {
        std::cout << s << " ";
    }
    std::cout << "\n";

    return 0;
}
```

```
42 67 73 88 95
```

It works with strings too — they sort alphabetically:

```
std::vector<std::string> songs = {"Hey Ya!", "Mr. Brightside", "Hips Don't Lie"};
std::sort(songs.begin(), songs.end());
// songs is now {"Hey Ya!", "Hips Don't Lie", "Mr. Brightside"}
```

Finding Elements

`std::find` searches for a value and returns an iterator to the first match, or `.end()` if not found:

```
Iterator find(Iterator first, Iterator last, const T& value);
```

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

int main()
{
    std::vector<std::string> playlist = {"Hey Ya!", "Mr. Brightside", "Hips Don't Lie"};

    auto it = std::find(playlist.begin(), playlist.end(), "Mr. Brightside");
    if (it != playlist.end()) {
        std::cout << "Found: " << *it << "\n";
    } else {
        std::cout << "Not found\n";
    }

    return 0;
}
```

```
Found: Mr. Brightside
```

Tip: Always check the result of `std::find` against `.end()` before dereferencing the iterator. Dereferencing `.end()` is undefined behavior.

Counting

`std::count` counts how many times a value appears:

```cpp
int count(Iterator first, Iterator last, const T& value);
```

```cpp
std::vector<int> votes = {1, 2, 1, 3, 1, 2, 1};
int ones = std::count(votes.begin(), votes.end(), 1);
std::cout << "Number of 1s: " << ones << "\n";  // 4
```

`for_each`

`std::for_each` applies a function to every element in a range. It is similar to a range-based for loop, but is useful when you want to pass a function directly:

```cpp
Function for_each(Iterator first, Iterator last, Function f);
```

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

void print_song(const std::string& song)
{
    std::cout << "  " << song << "\n";
}

int main()
{
    std::vector<std::string> songs = {"Hey Ya!", "Mr. Brightside"};
```

```
    std::cout << "Playlist:\n";
    std::for_each(songs.begin(), songs.end(), print_song);

    return 0;
}
```

```
Playlist:
  Hey Ya!
  Mr. Brightside
```

That third argument is a function — `std::for_each` calls it once for every element. But writing a separate named function for something this simple is tedious. This is where lambdas come in.

Lambdas

A **lambda** is an anonymous function that you define right where you need it. Instead of writing a separate function like `print_song` above, you can write:

```
std::for_each(songs.begin(), songs.end(), [](const std::string& song) {
    std::cout << "  " << song << "\n";
});
```

The lambda syntax is:

```
[captures](parameters) { body }
```

- **`[captures]`** — what variables from the surrounding scope the lambda can access
- **`(parameters)`** — just like function parameters
- **`{ body }`** — the code to execute

Here is a simple example:

```
auto greet = [](const std::string& name) {
    std::cout << "Hola, " << name << "!\n";
};

greet("Shakira");  // Hola, Shakira!
greet("OutKast");  // Hola, OutKast!
```

You can store a lambda in a variable using `auto` and call it like a regular function.

Captures

Lambdas can access variables from the surrounding scope through **captures**:

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

int main()
{
    std::vector<int> nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int threshold = 5;

    // std::count_if: like std::count, but with a predicate
    //   int count_if(Iterator first, Iterator last, Predicate pred);
    auto count = std::count_if(nums.begin(), nums.end(),
        [threshold](int n) { return n > threshold; });
```

```
    std::cout << count << " numbers above " << threshold << "\n";

    return 0;
}
```

```
5 numbers above 5
```

The `[threshold]` capture makes a copy of `threshold` available inside the lambda.

Here are the capture options:

Syntax	Meaning
`[]`	Capture nothing
`[x]`	Capture x by value (copy)
`[&x]`	Capture x by reference
`[=]`	Capture everything by value
`[&]`	Capture everything by reference
`[=, &x]`	Capture everything by value, but x by reference

```
int total = 0;
std::vector<int> prices = {10, 20, 30};

std::for_each(prices.begin(), prices.end(), [&total](int p) {
    total += p;
});

std::cout << "Total: " << total << "\n";  // Total: 60
```

Here `[&total]` captures `total` by reference, so the lambda can modify it.

Tip: Prefer specific captures like `[x]` or `[&x]` over blanket captures like `[=]` or `[&]`. Explicit captures make it clear what the lambda depends on and help prevent accidental captures.

Transform

`std::transform` applies a function to each element and stores the result. It can write the results back into the same container or into a different one:

```
OutputIterator transform(Iterator first, Iterator last, OutputIterator result, Function f);
```

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

int main()
{
    std::vector<int> nums = {1, 2, 3, 4, 5};
    std::vector<int> doubled(nums.size());

    std::transform(nums.begin(), nums.end(), doubled.begin(),
        [](int n) { return n * 2; });

    for (const auto& d : doubled) {
        std::cout << d << " ";
```

```
    }
    std::cout << "\n";

    return 0;
}
```

```
2 4 6 8 10
```

Trap: When writing results to a different container, that container must already have enough space. In the example above, `doubled` is created with `nums.size()` elements. If you use an empty vector, you will write past its end — undefined behavior.

Accumulate

`std::accumulate` reduces a range to a single value by applying an operation. It lives in `<numeric>`, not `<algorithm>`:

```
T accumulate(Iterator first, Iterator last, T init);
T accumulate(Iterator first, Iterator last, T init, BinaryOp op);
```

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

int main()
{
    std::vector<int> scores = {90, 85, 92, 88};

    int sum = std::accumulate(scores.begin(), scores.end(), 0);
    std::cout << "Sum: " << sum << "\n";
    std::cout << "Average: " << sum / static_cast<int>(scores.size()) << "\n";

    return 0;
}
```

```
Sum: 355
Average: 88
```

The third argument (`0`) is the initial value. You can also pass a custom operation as a fourth argument:

```
int product = std::accumulate(scores.begin(), scores.end(), 1,
    [](int a, int b) { return a * b; });
```

Min and Max

`std::min_element` and `std::max_element` return iterators to the smallest and largest elements:

```
Iterator min_element(Iterator first, Iterator last);
Iterator max_element(Iterator first, Iterator last);
```

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

int main()
{
    std::vector<int> temps = {72, 68, 85, 61, 79};
```

```
    auto coldest = std::min_element(temps.begin(), temps.end());
    auto hottest = std::max_element(temps.begin(), temps.end());

    std::cout << "Coldest: " << *coldest << "\n";
    std::cout << "Hottest: " << *hottest << "\n";

    return 0;
}
```

```
Coldest: 61
Hottest: 85
```

Ranges (C++20)

The algorithms above all take pairs of iterators: `container.begin(), container.end()`. C++20 introduced the `std::ranges` namespace, which lets you pass the container directly:

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

int main()
{
    std::vector<std::string> songs = {"Hey Ya!", "Mr. Brightside", "Hips Don't Lie"};

    // void ranges::sort(Range& r);
    std::ranges::sort(songs);

    for (const auto& s : songs) {
        std::cout << s << "\n";
    }

    return 0;
}
```

```
Hey Ya!
Hips Don't Lie
Mr. Brightside
```

Compare `std::sort(songs.begin(), songs.end())` with `std::ranges::sort(songs)`. The ranges version is simpler and less error-prone — you cannot accidentally pass mismatched iterators.

`std::ranges::find` works the same way:

```
// Iterator ranges::find(Range& r, const T& value);
auto it = std::ranges::find(songs, "Mr. Brightside");
if (it != songs.end()) {
    std::cout << "Encontrado: " << *it << "\n";
}
```

Tip: When your compiler supports C++20, prefer `std::ranges::` versions of algorithms. They are simpler, safer, and often provide better error messages when something goes wrong.

Views

Views are one of the most powerful features added in C++20. A **view** is a lightweight wrapper that transforms or filters elements **lazily** — it does not create a new container or copy data. Instead, it computes each element on demand as you iterate.

Think of it like looking through a filter: the original data does not change, you just see it differently.

Pipe Syntax

Views use the pipe operator `|` to chain operations together, much like Unix pipes:

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

int main()
{
    std::vector<int> nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // auto views::filter(Predicate pred);   // keeps elements where pred is true
    for (int n : nums | std::views::filter([](int n) { return n % 2 == 0; })) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    return 0;
}
```

```
2 4 6 8 10
```

The expression `nums | std::views::filter(...)` creates a view that only yields even numbers. No new vector is created — the filter is applied on the fly as you iterate.

Common Views

Here are the views you will use most often:

```cpp
auto views::filter(Predicate pred);      // keeps matching elements
auto views::transform(Function f);       // applies f to each element
auto views::take(int n);                 // first n elements
auto views::drop(int n);                 // skip first n elements
auto views::reverse;                     // iterate in reverse
```

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

int main()
{
    std::vector<int> nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // filter: keep only elements matching a condition
    std::cout << "Even: ";
    for (int n : nums | std::views::filter([](int n) { return n % 2 == 0; })) {
        std::cout << n << " ";
    }
    std::cout << "\n";
```

```
    // transform: apply a function to each element
    std::cout << "Tripled: ";
    for (int n : nums | std::views::transform([](int n) { return n * 3; })) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    // take: keep only the first N elements
    std::cout << "First 3: ";
    for (int n : nums | std::views::take(3)) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    // drop: skip the first N elements
    std::cout << "Skip 7: ";
    for (int n : nums | std::views::drop(7)) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    // reverse: iterate in reverse order
    std::cout << "Reversed: ";
    for (int n : nums | std::views::reverse) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    return 0;
}
```

```
Even: 2 4 6 8 10
Tripled: 3 6 9 12 15 18 21 24 27 30
First 3: 1 2 3
Skip 7: 8 9 10
Reversed: 10 9 8 7 6 5 4 3 2 1
```

Chaining Views

The real power of views shows when you chain them together:

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

int main()
{
    std::vector<int> nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Get the first 3 even numbers, doubled
    std::cout << "Result: ";
    for (int n : nums
            | std::views::filter([](int n) { return n % 2 == 0; })
            | std::views::transform([](int n) { return n * 2; })
```

```
            | std::views::take(3)) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    return 0;
}
```

```
Result: 4 8 12
```

This reads almost like English: take `nums`, filter the even ones, double them, and take the first 3. Each `|` passes the result of the left side as input to the right side.

Wut: Views are lazy. In the chained example above, the `transform` and `filter` are not applied to all 10 elements. Once `take(3)` has yielded 3 elements, the pipeline stops — elements 8, 10, and beyond are never even looked at. This makes views very efficient when you only need a subset of results.

Views with Strings

Views work well with any container, including vectors of strings:

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

int main()
{
    std::vector<std::string> songs = {
        "Hey Ya!", "Mr. Brightside", "Hips Don't Lie"
    };

    std::cout << "Long titles:\n";
    for (const auto& s : songs
            | std::views::filter([](const std::string& s) {
                return s.size() > 10;
            })) {
        std::cout << "  " << s << "\n";
    }

    return 0;
}
```

```
Long titles:
  Mr. Brightside
  Hips Don't Lie
```

Try It: Algorithm Starter

Here is a program that exercises several algorithms and views. Type it in, compile with `g++ -std=c++23`, and experiment:

```
#include <algorithm>
#include <iostream>
#include <numeric>
```

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

int main()
{
    std::vector<int> scores = {72, 95, 88, 61, 84, 90, 77};

    // Sort
    std::ranges::sort(scores);
    std::cout << "Sorted: ";
    for (int s : scores) {
        std::cout << s << " ";
    }
    std::cout << "\n";

    // Find
    // std::distance returns the number of steps between two iterators:
    //   int distance(Iterator first, Iterator last);
    auto it = std::ranges::find(scores, 88);
    if (it != scores.end()) {
        std::cout << "Found 88 at position "
                  << std::distance(scores.begin(), it) << "\n";
    }

    // Accumulate (still needs begin/end)
    int sum = std::accumulate(scores.begin(), scores.end(), 0);
    std::cout << "Sum: " << sum << ", Average: "
              << sum / static_cast<int>(scores.size()) << "\n";

    // Min and max — ranges::minmax returns the smallest and largest elements:
    //   auto ranges::minmax(Range& r);   // returns {min, max}
    auto [lo, hi] = std::ranges::minmax(scores);
    std::cout << "Min: " << lo << ", Max: " << hi << "\n";

    // Lambda with count_if
    //   int ranges::count_if(Range& r, Predicate pred);
    int above_80 = std::ranges::count_if(scores,
        [](int s) { return s > 80; });
    std::cout << "Scores above 80: " << above_80 << "\n";

    // Views pipeline
    std::cout << "Top 3 scores doubled: ";
    for (int s : scores
            | std::views::reverse
            | std::views::take(3)
            | std::views::transform([](int s) { return s * 2; })) {
        std::cout << s << " ";
    }
    std::cout << "\n";

    return 0;
}
```

Key Points

- The `<algorithm>` header provides reusable functions like `std::sort`, `std::find`, `std::count`, `std::for_each`, and `std::transform`.
- `std::accumulate` (from `<numeric>`) reduces a range to a single value.
- Lambdas are anonymous functions written as `[captures](params) { body }`. They are the primary way to customize algorithm behavior.
- Captures control what a lambda can access from its surrounding scope: `[x]` by value, `[&x]` by reference, `[=]` all by value, `[&]` all by reference.
- C++20 `std::ranges::` algorithms accept containers directly instead of iterator pairs.
- Views (`std::views::filter`, `transform`, `take`, `drop`, `reverse`) apply transformations lazily without copying data.
- Views chain together with the `|` pipe operator, creating readable data pipelines.
- Lazy evaluation means views only compute elements as they are consumed, making them efficient for large data sets.

Exercises

1. **Think about it:** Why do you think the standard library provides both `std::sort(v.begin(), v.end())` and `std::ranges::sort(v)`? If the ranges version is simpler, why keep the iterator version?

2. **What does this print?**

```cpp
std::vector<int> v = {5, 3, 8, 1, 9, 2};
std::sort(v.begin(), v.end());
auto it = std::find(v.begin(), v.end(), 8);
std::cout << *it << " " << *(it - 1) << "\n";
```

3. **What does this print?**

```cpp
std::vector<int> v = {1, 2, 3, 4, 5};
auto result = std::count_if(v.begin(), v.end(),
    [](int n) { return n % 2 != 0; });
std::cout << result << "\n";
```

4. **Where is the bug?**

```cpp
std::vector<int> nums = {10, 20, 30};
std::vector<int> doubled;

std::transform(nums.begin(), nums.end(), doubled.begin(),
    [](int n) { return n * 2; });
```

5. **Calculation:** Given this code:

```cpp
std::vector<int> v = {4, 7, 2, 9, 1};
int x = std::accumulate(v.begin(), v.end(), 10);
```

   What is the value of x?

6. **What does this print?**

```cpp
int factor = 3;
auto multiply = [factor](int n) { return n * factor; };
std::cout << multiply(5) << " " << multiply(10) << "\n";
```

7. **Where is the bug?**

```cpp
std::vector<int> nums = {1, 2, 3, 4, 5};
int total = 0;
```

```cpp
std::for_each(nums.begin(), nums.end(), [total](int n) {
    total += n;
});

std::cout << "Total: " << total << "\n";
```

8. **Think about it:** Views are "lazy" — they do not process elements until you iterate. Why is this an advantage? Can you think of a situation where processing all elements upfront would be better?

9. **What does this print?** (Assume C++20)

```cpp
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8};
for (int n : v
        | std::views::filter([](int n) { return n > 3; })
        | std::views::take(3)) {
    std::cout << n << " ";
}
std::cout << "\n";
```

10. **Write a program** that stores a list of test scores in a `std::vector<int>`, then uses algorithms and/or views to:

    - Sort the scores
    - Print only scores above 70
    - Print the average score
    - Print the highest and lowest scores