Gorgo Continuing C++

April 11, 2026

Contents

3. The Standard Template Library

In *Gorgo Starting C++* you learned `std::vector` and `std::array` — the two workhorses of sequential storage. But not every problem is a list. Sometimes you need to look up a value by key, enforce uniqueness, or process items in priority order. The **Standard Template Library** (STL) provides a rich set of containers, each optimized for different access patterns. In Chapter 2 you learned that these containers are class templates — they work with any type. In this chapter you will learn the associative containers, unordered containers, and container adaptors, and how to choose the right one for the job.

Associative Containers

Associative containers store elements in **sorted order** and provide efficient lookup, insertion, and deletion — typically O(log n). They are implemented as balanced binary search trees (usually red-black trees).

`std::map`

`std::map<Key, Value>` stores key-value pairs sorted by key. Each key is unique:

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

int main()
{
    std::map<std::string, int> album_years;
    album_years["Elephant"] = 2003;
    album_years["Hot Fuss"] = 2004;
    album_years["Funeral"] = 2004;

    for (const auto& [album, year] : album_years) {
        std::cout << album << ": " << year << "\n";
    }

    return 0;
}
```

```
Elephant: 2003
Funeral: 2004
Hot Fuss: 2004
```

The output is sorted alphabetically by key. The `const auto& [album, year]` syntax is a **structured binding** that unpacks each `std::pair<const std::string, int>` in the map.

Key operations:

```
// Insert or update
album_years["X&Y"] = 2005;
album_years.insert({"Absolution", 2003});

// Lookup
if (album_years.count("Funeral") > 0) {
    std::cout << "Found\n";
}

// C++20: contains()
//   bool contains(const Key& key) const;
if (album_years.contains("Funeral")) {
```

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

// Safe access with find()
//   iterator find(const Key& key);
auto it = album_years.find("Elephant");
if (it != album_years.end()) {
    std::cout << it->second << "\n";  // 2003
}
```

Trap: Using `operator[]` on a key that does not exist will **insert** a default-constructed value. If you just want to check whether a key exists, use `find()` or `contains()` instead.

std::set

`std::set<T>` stores unique values in sorted order. It is like a `map` with only keys:

```
#include <iostream>
#include <set>
#include <string>

int main()
{
    std::set<std::string> genres;
    genres.insert("Rock");
    genres.insert("Pop");
    genres.insert("Rock");  // duplicate — ignored
    genres.insert("Indie");

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

    std::cout << "Size: " << genres.size() << "\n";

    return 0;
}
```

```
Indie
Pop
Rock
Size: 3
```

`insert()` returns a `std::pair<iterator, bool>` — the `bool` tells you whether the insertion actually happened:

```
auto [it, inserted] = genres.insert("Pop");
if (!inserted) {
    std::cout << "Pop was already in the set\n";
}
```

std::multimap and std::multiset

`std::multimap` and `std::multiset` are like `map` and `set` but allow **duplicate keys**:

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

int main()
{
    std::multimap<std::string, std::string> songs_by_genre;
    songs_by_genre.insert({"Rock", "Seven Nation Army"});
    songs_by_genre.insert({"Rock", "Mr. Brightside"});
    songs_by_genre.insert({"Pop", "Crazy in Love"});
    songs_by_genre.insert({"Pop", "Toxic"});

    // equal_range returns a pair of iterators: [first match, past last match)
    //   pair<iterator, iterator> equal_range(const Key& key);
    auto [begin, end] = songs_by_genre.equal_range("Rock");
    std::cout << "Rock songs:\n";
    for (auto it = begin; it != end; ++it) {
        std::cout << "  " << it->second << "\n";
    }

    return 0;
}
```

```
Rock songs:
  Seven Nation Army
  Mr. Brightside
```

Tip: `multimap` does not have `operator[]` because a key can map to multiple values. Use `find()`, `equal_range()`, or a range-based loop to access elements.

Unordered Containers

Unordered containers use **hash tables** instead of trees. They provide O(1) average-case lookup, insertion, and deletion — faster than the O(log n) of ordered containers. The trade-off is that elements are not stored in any particular order.

`std::unordered_map`

`std::unordered_map<Key, Value>` is the hash-table equivalent of `std::map`:

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

int main()
{
    std::unordered_map<std::string, int> play_counts;
    play_counts["Rehab"] = 42;
    play_counts["Poker Face"] = 87;
    play_counts["Use Somebody"] = 55;

    play_counts["Rehab"] += 1;

    for (const auto& [song, count] : play_counts) {
```

```
        std::cout << song << ": " << count << " plays\n";
    }

    return 0;
}
```

The output order is not alphabetical — it depends on the hash function and internal bucket layout.

The API is nearly identical to `std::map`: `operator[]`, `find()`, `contains()`, `insert()`, `erase()` all work the same way.

`std::unordered_set`

`std::unordered_set<T>` is the hash-table equivalent of `std::set`:

```
#include <iostream>
#include <string>
#include <unordered_set>

int main()
{
    std::unordered_set<std::string> tags = {"chill", "summer", "dance", "chill"};

    std::cout << "Tags (" << tags.size() << "):\n";
    for (const auto& t : tags) {
        std::cout << "  " << t << "\n";
    }

    return 0;
}
```

Duplicates are ignored, just like `std::set`, but the elements are not sorted.

Wut: To use a custom type as a key in an unordered container, you must provide a hash function and an equality operator. Standard types like `std::string`, `int`, and `double` already have hash functions.

Ordered vs. Unordered

Feature	`map/set`	`unordered_map/unordered_set`
Order	Sorted by key	No guaranteed order
Lookup	O(log n)	O(1) average, O(n) worst
Insertion	O(log n)	O(1) average, O(n) worst
Requires	`operator<` or comparator	Hash function + `operator==`
Memory	Tree nodes (pointer overhead)	Hash buckets (may waste space)

Use ordered containers when you need sorted iteration or range queries. Use unordered containers when you only need fast lookup by key.

Container Adaptors

Container adaptors wrap an underlying container and restrict its interface to provide a specific behavior. They are not full containers — you cannot iterate through them.

std::stack

`std::stack<T>` provides LIFO (last in, first out) access:

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

int main()
{
    std::stack<std::string> history;
    history.push("Chasing Cars");
    history.push("How to Save a Life");
    history.push("Fix You");

    while (!history.empty()) {
        std::cout << history.top() << "\n";
        history.pop();
    }

    return 0;
}
```

```
Fix You
How to Save a Life
Chasing Cars
```

Key methods:

```
void push(const T& value);   // add to top
void pop();                   // remove from top (does not return it)
T& top();                     // access top element
bool empty() const;
size_t size() const;
```

Trap: `pop()` does not return the removed element. You must call `top()` first to get the value, then `pop()` to remove it.

std::queue

`std::queue<T>` provides FIFO (first in, first out) access:

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

int main()
{
    std::queue<std::string> playlist;
    playlist.push("Crazy");
    playlist.push("Gold Digger");
    playlist.push("Single Ladies");

    while (!playlist.empty()) {
        std::cout << "Now playing: " << playlist.front() << "\n";
        playlist.pop();
```

```
    }

    return 0;
}
```

```
Now playing: Crazy
Now playing: Gold Digger
Now playing: Single Ladies
```

Key methods:

```
void push(const T& value);   // add to back
void pop();                   // remove from front
T& front();                   // access front element
T& back();                    // access back element
bool empty() const;
size_t size() const;
```

std::priority_queue

`std::priority_queue<T>` is a queue where the highest-priority element is always at the top. By default, it is a max-heap — the largest element has the highest priority:

```
#include <iostream>
#include <queue>

int main()
{
    std::priority_queue<int> pq;
    pq.push(30);
    pq.push(10);
    pq.push(50);
    pq.push(20);

    while (!pq.empty()) {
        std::cout << pq.top() << " ";
        pq.pop();
    }
    std::cout << "\n";

    return 0;
}
```

```
50 30 20 10
```

To get a min-heap (smallest first), use `std::greater<T>` as the comparator:

```
std::priority_queue<int, std::vector<int>, std::greater<int>> min_pq;
```

Choosing the Right Container

Here is a quick decision guide:

Need	Container
Sequential access, dynamic size	`std::vector`
Fixed-size array	`std::array`
Fast lookup by key (sorted)	`std::map`

Need	Container
Fast lookup by key (unsorted)	`std::unordered_map`
Unique values (sorted)	`std::set`
Unique values (unsorted)	`std::unordered_set`
Duplicate keys allowed (sorted)	`std::multimap` / `std::multiset`
LIFO (stack behavior)	`std::stack`
FIFO (queue behavior)	`std::queue`
Priority-based access	`std::priority_queue`

When in doubt, start with `std::vector`. It has the best cache performance and works well for most tasks. Switch to a different container only when you have a specific need that `std::vector` does not serve well.

Tip: `std::vector` is almost always faster than you expect. Even linear search through a small vector often beats hash table or tree lookup because of CPU cache effects. Profile before switching containers.

Try It: Container Sampler

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

```cpp
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <stack>
#include <string>
#include <unordered_map>

int main()
{
    // map: album ratings
    std::map<std::string, double> ratings;
    ratings["Hot Fuss"] = 4.5;
    ratings["Elephant"] = 4.8;
    ratings["Funeral"] = 4.7;
    ratings["X&Y"] = 3.9;

    std::cout << "Albums (sorted):\n";
    for (const auto& [album, rating] : ratings) {
        std::cout << "  " << album << ": " << rating << "\n";
    }

    // set: unique genres
    std::set<std::string> genres = {"Rock", "Indie", "Pop", "Rock", "Indie"};
    std::cout << "\nGenres: ";
    for (const auto& g : genres) {
        std::cout << g << " ";
    }
    std::cout << "\n";

    // unordered_map: fast lookup
    std::unordered_map<std::string, int> track_num;
```

```
    track_num["Intro"] = 1;
    track_num["Verse"] = 2;
    track_num["Chorus"] = 3;

    if (track_num.contains("Chorus")) {
        std::cout << "\nChorus is track " << track_num["Chorus"] << "\n";
    }

    // stack: undo history
    std::stack<std::string> undo;
    undo.push("add track");
    undo.push("change volume");
    undo.push("delete track");

    std::cout << "\nUndo stack:\n";
    while (!undo.empty()) {
        std::cout << "  undo: " << undo.top() << "\n";
        undo.pop();
    }

    // priority_queue: highest rated first
    std::priority_queue<std::pair<double, std::string>> top_albums;
    for (const auto& [album, rating] : ratings) {
        top_albums.push({rating, album});
    }

    std::cout << "\nTop albums:\n";
    while (!top_albums.empty()) {
        auto [rating, album] = top_albums.top();
        std::cout << "  " << album << " (" << rating << ")\n";
        top_albums.pop();
    }

    return 0;
}
```

Try adding a `std::multimap` that maps genres to multiple songs. Experiment with `std::unordered_set` and see how the iteration order differs from `std::set`.

Key Points

- **Associative containers** (`std::map`, `std::set`, `std::multimap`, `std::multiset`) store elements in sorted order using balanced trees, with O(log n) operations.
- `std::map` stores unique key-value pairs; `std::set` stores unique values only.
- `std::multimap` and `std::multiset` allow duplicate keys.
- **Unordered containers** (`std::unordered_map`, `std::unordered_set`) use hash tables for O(1) average-case operations, but elements have no guaranteed order.
- Use `contains()` (C++20) or `find()` to check for existence without inserting. `operator[]` on a `map` inserts a default value if the key is missing.
- **Container adaptors** (`std::stack`, `std::queue`, `std::priority_queue`) wrap other containers to provide restricted interfaces.
- `stack` is LIFO, `queue` is FIFO, `priority_queue` is a max-heap by default.
- `pop()` on adaptors does not return the removed element — call `top()` or `front()` first.
- When choosing a container, start with `std::vector` and switch only when you have a specific need.

- Use structured bindings (`auto [key, value]`) to iterate over maps concisely.

Exercises

1. **Think about it:** Why does `std::map::operator[]` insert a default value when the key is missing, instead of throwing an exception? What would the implications be if it threw instead?
2. **What does this print?**

```cpp
std::map<std::string, int> m;
m["b"] = 2;
m["a"] = 1;
m["c"] = 3;

for (const auto& [k, v] : m) {
    std::cout << k << v;
}
std::cout << "\n";
```

3. **Where is the bug?**

```cpp
std::map<std::string, int> scores;
scores["Alice"] = 95;
scores["Bob"] = 87;

int charlie_score = scores["Charlie"];
std::cout << "Charlie: " << charlie_score << "\n";
```

4. **What does this print?**

```cpp
std::set<int> s = {5, 3, 1, 4, 1, 5, 3};
std::cout << s.size() << "\n";
```

5. **Calculation:** A `std::map<std::string, int>` has 1,000,000 entries. Approximately how many comparisons does a lookup require? (Hint: O(log n) with base 2.)
6. **Think about it:** When would you choose `std::map` over `std::unordered_map`? Give two concrete scenarios.
7. **What does this print?**

```cpp
std::stack<int> s;
s.push(10);
s.push(20);
s.push(30);
s.pop();
std::cout << s.top() << "\n";
s.pop();
std::cout << s.top() << "\n";
```

8. **Where is the bug?**

```cpp
std::priority_queue<int> pq;
pq.push(5);
pq.push(15);
pq.push(10);

int smallest = pq.top();
std::cout << "Smallest: " << smallest << "\n";
```

9. **What does this print?**

```
std::multiset<int> ms = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
std::cout << ms.count(5) << "\n";
```

10. **Write a program** that reads words from the user (one per line, until they type "done") and uses a `std::map<std::string, int>` to count how many times each word was entered. Print the word counts in alphabetical order.