The C++ Master Companion — Syntax, Insight & Practice

ZephyrAmmor

October 2025

Contents

Module 4: The Standard Template Library (STL)

Overview

The Standard Template Library (STL) is C++'s backbone for data structures and algorithms — designed for performance, reusability, and type safety. It embodies the philosophy of zero-cost abstraction: high-level interfaces without runtime overhead.

Concept Overview

STL is composed of four main components:

1. Containers – Data structures that store collections of elements.

2. Iterators – Generalized pointers for navigating containers.
3. Algorithms – Reusable functions that operate on containers via iterators.
4. Functors & Lambdas – Callable objects for flexible behavior.

Each piece interacts seamlessly, giving C++ one of the most efficient and elegant standard libraries in existence.

Containers

Containers manage data storage and access. They are broadly categorized as:

Category	Description	Examples
Sequence Containers	Maintain elements in linear order	`vector`, `deque`, `list`
Associative Containers	Sorted key-value or unique-key sets	`map`, `set`, `multimap`, `multiset`
Unordered Containers	Hash-based for faster lookup	`unordered_map`, `unordered_set`
Container Adapters	Wrap other containers for special behavior	`stack`, `queue`, `priority_queue`

Example

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

int main() {
    std::vector<int> nums = {1, 2, 3, 4, 5};
    nums.push_back(6);

    for (auto n : nums) std::cout << n << " ";
}
```

Insight

`std::vector` provides contiguous storage — meaning it behaves much like a dynamic array but with automatic resizing.

Pitfall

Avoid calling `push_back` inside loops with unreserved capacity — it can trigger reallocations and iterator invalidation.

Under the Hood

`std::vector` doubles its capacity when reallocation is required. The old memory is copied/moved to a new location, which is why storing raw pointers to vector elements is unsafe across resizes.

Best Practices

- Prefer `std::vector` as your default container — it's cache-friendly and fast.
- Use `reserve()` if the final size is predictable.
- For associative lookups, prefer `unordered_map` unless order matters.

Iterators and Ranges

Iterators generalize pointers — they allow algorithms to work with any container.

 Syntax

```
auto it = container.begin();
while (it != container.end()) {
    std::cout << *it << " ";
    ++it;
}
```

 Insight

Iterators decouple data structure from algorithm logic — that's the essence of STL's design genius.

 Pitfall

Invalidating an iterator (e.g., after `erase()` or reallocation) leads to undefined behavior.

 Under the Hood

Each container provides specific iterator types (random-access, bidirectional, forward, etc.). Algorithms choose the optimal implementation based on iterator category.

 Best Practices

- Use range-based for loops or algorithms instead of manual iteration when possible.
- Use `cbegin()` and `cend()` for read-only iteration.

Algorithms

The `<algorithm>` header defines generic, optimized operations like sorting, searching, transforming, and accumulating.

 Syntax

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

std::sort(vec.begin(), vec.end());
```

 Example

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

int main() {
    std::vector<int> v = {5, 1, 4, 2, 3};
    std::sort(v.begin(), v.end());

    for (auto n : v) std::cout << n << " ";
}
```

 Insight

Algorithms don't know or care about containers — they only work on iterator pairs.

 Common Algorithms

Category	Examples
Sorting/Search	`sort`, `find`, `binary_search`
Modification	`remove`, `replace`, `transform`

Category	Examples
Aggregation	`accumulate`, `count`, `all_of`
Partitioning	`partition`, `stable_partition`

 Best Practices

- Use algorithms instead of manual loops whenever possible — they're expressive and optimized.
- Combine with lambdas for clarity:

```
std::for_each(vec.begin(), vec.end(), [](int &n){ n *= 2; });
```

Function Objects and Lambdas

Function objects (functors) and lambdas enable custom behavior injection into algorithms.

 Syntax

```
std::sort(vec.begin(), vec.end(), [](int a, int b){ return a > b; });
```

 Insight

Lambdas replaced functors in most modern C++ code due to conciseness and capture capabilities.

 Under the Hood

Lambdas are syntactic sugar for unnamed function objects with an `operator()` overload. Capture lists determine how external variables are stored.

 Pitfall

Avoid long or complex lambdas inside algorithms — move them to named functions or functors for clarity.

Performance and Complexity

- **`std::vector`**: amortized O(1) insert at end
- **`std::map`**: O(log n) for insert/search (balanced tree)
- **`std::unordered_map`**: average O(1), worst O(n) due to hash collisions

 Insight

Know your access pattern — STL is fast only when you pick the right container for the job.

Under the Hood Summary

- STL algorithms are header-only templates, optimized at compile time.
- Containers allocate on the heap, but small object optimizations and move semantics minimize cost.
- Iterator invalidation is a real-world bug magnet — learn each container's rules.

Best Practices & Optimization Insights

- Favor value semantics — avoid raw pointers inside STL containers.
- Combine STL algorithms with range-based programming (C++20's `<ranges>`).
- Profile and test — don't assume one container outperforms another universally.

□ Availability Tags

- `std::array`, `unordered_map`, `unordered_set` → C++11
- Range-based for, lambdas → C++11
- `std::make_unique` → C++14
- `<ranges>` → C++20

End of Module 4 — STL Mastery