https://simplifycpp.org

STL Containers in Modern C++

A Concise and Focused Guide to Achieving Professional Mastery

Prepared By Ayman Alheraki

STL Containers in Modern C++

A Concise and Focused Guide to Achieving Professional Mastery

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Author's Preface

Since its introduction in the mid-1990s, the Standard Template Library (STL) has been one of the defining achievements of C++. Its power, consistency, and performance made it the foundation of modern programming practices—especially after C++11, which improved containers dramatically through move semantics, allocator improvements, stronger iterator guarantees, and unified semantics.
This booklet provides a **concise but highly advanced guide** to mastering STL containers. It is written for:

- Intermediate developers seeking mastery.
- Professionals needing performance-critical container usage.
- Students and researchers studying STL design and internals.
- Engineers writing high-speed and large-scale software.

The goal is clarity and depth—covering memory models, iterator categories, performance characteristics, complexities, container selection strategies, and real-world patterns.
Each chapter includes **exercises and full solutions** to ensure complete understanding.

Chapter 1

STL Philosophy and Design Principles

The STL is built around three primary pillars:

1. **Containers** — data structures with well-defined complexity guarantees.
2. **Iterators** — a unifying abstraction for traversal.
3. **Algorithms** — generic operations working on iterator pairs.

1.1 Key Properties of STL Containers

- Value-based semantics.
- Strong exception guarantees.
- Predictable complexity.
- ABI-stable container interfaces.
- Seamless integration with modern features (lambdas, ranges, allocators).

1.2 Core Requirements of Containers

Each STL container must provide:

- Defined iterator types.
- Size, capacity, and access semantics.
- Proper allocator support.
- Strong invariants.

1.3 Iterator Categories (Extremely Important)

- **Input / Output**
- **Forward**
- **Bidirectional**
- **Random Access**
- **Contiguous** (C++20)

This determines which algorithms you can use efficiently.

1.4 Value Semantics and Move Semantics

STL containers strongly depend on move constructors for performance.

```
std::vector<std::string> v;
v.push_back("Hello");        // moves internally when reallocated
```

1.5 Allocator Awareness

All containers support custom allocators.

Exercises — Chapter 1

1. Explain the separation of containers, iterators, and algorithms.
2. What are the five iterator categories?
3. Why are move semantics important for STL containers?
4. What properties must every STL container provide?
5. Write a short example showing a container using move semantics.
6. Explain why STL uses value semantics instead of reference semantics.
7. Describe allocator-aware design.
8. What is the difference between random access and bidirectional iterators?
9. Explain why the STL is generic and composable.
10. Why is exception safety crucial in STL design?

Solutions — Chapter 1

1. Separation allows generic algorithms to work with any container using iterators.
2. Input, Output, Forward, Bidirectional, Random Access, and C++20 Contiguous.
3. Moves avoid expensive copies during reallocation.
4. Iterators, size/capacity APIs, allocator support, defined invariants.
5. Code:

```
std::vector<std::string> v;
v.push_back(std::string("move"));
```

6. Value semantics provide safety, predictability, and clear ownership.
7. Allocators abstract memory management.
8. Random access allows indexing; bidirectional does not.
9. The STL is built around templates and iterator abstraction.
10. Containers must remain valid under exceptions; basic/strong guarantees.

Chapter 2

Sequence Containers — vector, deque, list, forward_list, array

Sequence containers maintain an ordered collection of elements.

2.1 std::vector — The Workhorse Container

2.1.1 Key Properties

- Contiguous memory (C++20: contiguous iterator category)
- Fast random access
- Amortized O(1) push_back
- Reallocation invalidates pointers/references

2.1.2 Common Operations

```
std::vector<int> v{1,2,3};
v.push_back(4);
v.reserve(100);
v.shrink_to_fit();
```

2.1.3 When to Use

- Best general-purpose container.
- Use unless another container is strongly justified.

2.2 std::deque — Double-Ended Growth

2.2.1 Characteristics

- Fast push_front and push_back.
- Blocks of memory instead of contiguous buffer.
- Stable references except during reallocation of block map.

```
std::deque<int> dq;
dq.push_front(10);
dq.push_back(20);
```

2.3 std::list — Doubly Linked List

2.3.1 Characteristics

- Stable iterators.

- No contiguous memory.
- Slow random access.
- Splice operations: O(1)

```
std::list<int> lst{1,2,3};
lst.splice(lst.begin(), other_list);
```

2.4 std::forward_list — Singly Linked List

Minimalistic, low memory overhead, no size().

2.5 std::array — Stack-Based Fixed Container

```
std::array<int, 3> arr{1,2,3};
```

Fast, small, trivial layout.

Exercises — Chapter 2

1. Describe when vector is preferred over list.
2. What makes deque different from vector?
3. Write code to demonstrate list::splice.
4. Explain iterator invalidation rules for vector.
5. Create an example of forward_list usage.
6. Compare std::array and vector memory layout.
7. Why is list bad for cache locality?
8. Write a function that receives any sequence container using templates.
9. Show how vector reallocation happens.
10. When should forward_list be used?

Solutions — Chapter 2

1. Vector is better unless constant-time insertion in the middle is needed.
2. Deque uses many memory blocks; vector uses one contiguous block.
3.
```
lst.splice(lst.begin(), other);
```
4. Any push_back that triggers growth invalidates references.
5.
```
std::forward_list<int> fl{1,2,3};
```

6. Array is fixed on stack; vector allocates dynamically.

7. Linked lists scatter nodes in memory → poor locality.

8. ```
   template<typename Seq>
   void print(const Seq& s){
       for(auto& x:s) std::cout<<x;
   }
   ```

9. Capacity doubles, memory moves.

10. When memory usage must be minimal.
   ```

Chapter 3

Associative Containers — set, map, multiset, multimap

Associative containers maintain elements in sorted order using **balanced binary search trees** (typically Red–Black trees). They guarantee:

- O(log n) insertion
- O(log n) lookup
- O(log n) erase
- Ordered iteration

These containers are ideal when you need **sorted data**, **stable ordering**, or **predictability of operations**.

3.1 std::set — Unique Sorted Elements

3.1.1 Characteristics

- Unique keys.
- Strict weak ordering.
- Fast search (log n).

```
std::set<int> s;
s.insert(5);
s.insert(1);
s.insert(5); // ignored
```

3.1.2 When to Use

- Removing duplicates automatically.
- Maintaining a sorted sequence of unique values.

3.2 std::multiset — Sorted Elements with Duplicates

```
std::multiset<int> ms{1,2,2,3};
```

Allows repeated values.

3.3 std::map — Sorted Key–Value Pairs

3.3.1 Properties

- Unique keys.
- Keys sorted using comparison.

- Values may repeat.

```
std::map<std::string, int> phone;
phone["Ali"] = 123;
phone["Zara"] = 555;
```

3.3.2 insert vs operator[]

```
mp["John"] = 100;       // inserts default-constructed value if key doesn't
↪  exist
mp.insert({"John", 100}); // no accidental construction
```

3.4 std::multimap — Duplicate Keys

```
std::multimap<std::string, int> grades{
    {"Ali", 90},
    {"Ali", 85}
};
```

Useful when keys map to multiple values.

3.5 Lookup Functions

```
auto it = s.find(10);
auto count = ms.count(2);
auto range = mp.equal_range("Ali");
```

Exercises — Chapter 3

1. Insert unique values into a set and observe behavior.
2. Demonstrate duplicate handling with multiset.
3. Write code using map with operator[] and insert().
4. Find all values for a key in multimap.
5. Show O(log n) lookup with map.
6. Explain why sets store elements sorted.
7. Write a function template that prints any associative container.
8. Compare set vs vector for searching.
9. Insert elements in reverse order into set and note iteration order.
10. Explain red–black tree balancing.

Solutions — Chapter 3

1. ```
   std::set<int> s{1,2,2,3};
   ```
2. ```
   std::multiset<int> ms{1,2,2,2,3};
   ```
3. ```
   mp["John"] = 5;
   mp.insert({"John", 5});
   ```
4. ```
   auto range = mm.equal_range("Ali");
   ```
5. Red–black tree guarantees logarithmic search.

6. Sorted ordering preserves algorithmic complexity.

7. ```
   template<class Assoc>
   void print(const Assoc& a){
       for(auto& x:a) std::cout<<x.first<<" "<<x.second<<"\n";
   }
   ```

8. Vector needs O(n) search unless sorted + binary search.

9. Inserts reverse, but set iterates in sorted order.

10. Self-balancing ensures O(log n) invariants.
   ```

Chapter 4

Unordered Containers — unordered_set, unordered_map, multiversions

Unordered containers provide **average O(1)** lookup using hash tables.

4.1 Characteristics

- Hash-based storage.
- Buckets + chaining.
- Average O(1) insertion / lookup.
- Worst-case O(n).

4.2 unordered_set

```
std::unordered_set<int> us;
```

```
us.insert(10);
us.insert(10); // ignored
```

4.3 unordered_multiset

Allows duplicates.

4.4 unordered_map

```
std::unordered_map<std::string, int> freq;
freq["apple"]++;
freq["banana"]++;
```

4.5 unordered_multimap

Allows duplicate keys.

4.6 Load Factor Control

```
umap.reserve(1000);
umap.max_load_factor(0.7f);
```

4.7 When to Use Unordered Containers

- Best for fast lookup.
- When ordering is irrelevant.

4.8 Custom Hash Functions

```
struct Point { int x,y; };

struct PointHash {
    size_t operator()(const Point& p) const {
        return std::hash<int>()(p.x) ^ (std::hash<int>()(p.y) << 1);
    }
};

std::unordered_set<Point, PointHash> pts;
```

Exercises — Chapter 4

1. Create an unordered_set and test duplicate handling.
2. Build a frequency counter using unordered_map.
3. Demonstrate reserve() impact on performance.
4. Write a custom hash functor for a struct.
5. Compare set vs unordered_set in lookup performance.
6. Explain what load factor means.
7. Why is unordered_map average O(1)?
8. Show collision resolution conceptually.
9. Create a multimap version allowing duplicate keys.
10. State when ordered vs unordered containers should be used.

Solutions — Chapter 4

1. ```
   std::unordered_set<int> us{1,1,2};
   ```
2. ```
   for(auto& s:words) freq[s]++;
   ```
3. reserve reduces rehashing.
4. ```
   struct H{ size_t operator()(const P&p) const { ... } };
   ```
5. Unordered_set is hash-based → average constant time.
   ```

6. Load factor = size / bucket_count.
7. Hash bucket indexing gives constant expected time.
8. Chains store multiple elements in a bucket.
9. `std::unordered_multimap<int,int> mm;`
10. Ordered = sorted; unordered = fastest lookups.

Chapter 5

Container Adaptors — stack, queue, priority_queue

Container adaptors change the interface of an underlying container.

5.1 stack

LIFO structure.

```
std::stack<int> st;
st.push(5);
st.push(10);
st.pop();
```

Default container: `deque`

5.2 queue

FIFO structure.

```
std::queue<std::string> q;
q.push("Hi");
q.push("Bye");
q.pop();
```

5.3 priority_queue

Max-heap by default.

```
std::priority_queue<int> pq;
pq.push(5);
pq.push(10);
pq.push(1);
std::cout << pq.top(); // 10
```

5.4 Custom Comparator

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

5.5 When to Use

- stack — recursion replacement, parsing.
- queue — BFS, producer/consumer.
- priority_queue — scheduling, shortest path (Dijkstra).

Exercises — Chapter 5

1. Show LIFO behavior with stack.
2. Implement FIFO behavior with queue.
3. Create a priority_queue that behaves as a min-heap.
4. Write a BFS using queue.
5. Use priority_queue for scheduling tasks.
6. Explain why stack uses deque by default.
7. Create a stack using vector as the underlying container.
8. Show how to clear a queue.
9. Insert custom objects into priority_queue using lambda comparator.
10. When should you use an adaptor instead of a full container?

Solutions — Chapter 5

1. ```
   st.push(1); st.push(2); st.pop();
   ```
2. ```
   q.push(1); q.pop();
   ```
3. ```
   auto cmp=[](int a,int b){return a>b;};
   std::priority_queue<int, std::vector<int>, decltype(cmp)> pq(cmp);
   ```
4. BFS is a classic use of queue.
5. Priority_queue models max-heap or min-heap scheduling.
   ```

6. deque allows fast push_back and push_front.
7. `std::stack<int, std::vector<int>> st;`
8. Reassign a new queue: `q = std::queue<int>();`
9. `auto cmp = [](const Job&a, const Job&b){ return a.prio > b.prio; };`
10. Adaptors impose a simple interface without full container complexity.

Chapter 6

Performance, Memory Models, and Iterator Stability

Understanding the performance characteristics of STL containers is essential for high-quality, professional C++ development. Container choice directly influences cache locality, latency, throughput, and overall application complexity.

6.1 Cache Locality and Spatial Access

6.1.1 Why vector is so fast

- Contiguous memory layout.
- Sequential access fits CPU prefetchers.
- SIMD-friendly (C++20 `std::span` integration).

```
std::vector<int> v = {1,2,3,4};
for (int x : v) process(x); // contiguous, prefetch-friendly
```

6.2 Iterator Invalidation Summary

- **vector**: invalidates on reallocation or insertion in the middle.
- **deque**: inserting anywhere may invalidate most iterators.
- **list / forward_list**: never invalidates iterators unless erased.
- **set/map**: erase invalidates only the erased iterator.
- **unordered containers**: rehashing invalidates all iterators.

6.2.1 Rehash Triggers

- High load factor.
- reserve(n) can prevent rehashing.

6.3 Complexity Overview

Table 3-1: Complexity of STL Containers

Container	Insert	Find	Erase
vector	Amortized O(1) (push_back)	O(n)	O(n)
deque	O(1) ends	O(n)	O(n)
list	O(1) with iterator	O(n)	O(1) with iterator
set/map	O(log n)	O(log n)	O(log n)
unordered set/map	O(1) avg	O(1) avg	O(1) avg

6.4 Growth Factor and Allocations

6.4.1 vector Growth Factor

Typically 1.5x–2x.

```
v.reserve(1000);   // manual tuning
```

6.4.2 Effects of Growth

Fewer allocations → better performance.

Exercises — Chapter 6

1. Explain why vector is more cache-friendly than list.
2. Show how to prevent unordered_map from rehashing frequently.
3. Write code demonstrating iterator invalidation with vector.
4. Compare complexity of set::find vs unordered_set::find.
5. Explain effects of load factor on hash containers.
6. Demonstrate memory growth using reserve().
7. What makes list operations slow despite O(1) insertion?
8. Write code showing list splice advantage.
9. Why is rehashing expensive?
10. Explain contiguous iterators in C++20.

Solutions — Chapter 6

1. vector is contiguous → sequential memory access.
2. ```
   um.reserve(10000);
   ```
3. ```
   auto it = v.begin();
   v.push_back(10);   // may invalidate it
   ```
4. set = log n; unordered_set = O(1) average.
5. High load factor → collisions.

6. reserve reduces reallocations.
7. Poor cache locality $\rightarrow$ many CPU stalls.
8. `lst.splice(pos, other);`
9. Rehash moves every element to new buckets.
10. contiguous iterators guarantee pointer arithmetic.

Chapter 7

Choosing the Right STL Container — A Professional Decision Guide

Container selection impacts performance, maintainability, and memory use.

7.1 Quick Selection Summary

Need Fast Random Access?	vector, array
Need Fast Insert/Delete Anywhere?	list (rare), deque
Need Sorted Data?	set, map
Need Fast Lookup?	unordered_set, unordered_map
Need FIFO / LIFO?	queue, stack
Need Priority Scheduling?	priority_queue

7.2 Patterns and Use Cases

7.2.1 Vectors for Almost Everything

- Best default container.
- Optimal for iteration, sorting, storage.

7.2.2 Hash Tables for Key-Based Access

When order is irrelevant and performance matters.

7.2.3 Maps for Order Guarantees

Use when iteration order matters.

7.2.4 Deque for Double-Ended Queues

Great for BFS, event queues, and sliding windows.

7.2.5 List for Stable Pointers

Rare, but essential in scenarios like:

- Node-based allocators.
- Splicing between lists.

7.3 Container Misuse Examples

- Using list instead of vector for insertion performance (often slower).
- Using unordered_map without reserving $\rightarrow$ rehash storms.
- Using map when unordered_map suffices $\rightarrow$ log n penalty.

7.4 Memory Optimization Techniques

- reserve()
- shrink_to_fit()
- emplace vs insert
- custom allocators for high-frequency allocation

Exercises — Chapter 7

1. Choose the best container for unique sorted elements.
2. Choose the container for the fastest key lookup.
3. Select a container for BFS traversal.
4. Explain why vector is preferred in most cases.
5. Write a scenario where map must be used instead of unordered_map.
6. Optimize an unordered_map for heavy insertion workload.
7. Give an example where list's splice is uniquely advantageous.
8. Select the container for task scheduling.
9. Show a misuse of an STL container.
10. Explain the role of shrink_to_fit().

Solutions — Chapter 7

1. set
2. unordered_map
3. queue or deque
4. memory locality and prefetching
5. when order must be preserved lexicographically

6. `um.reserve(50000);`
7. joining two lists in O(1)
8. priority_queue
9. using list instead of vector for random access
10. reclaim unused memory (often non-binding)

Chapter 8

Advanced Patterns, Architectures, and Real-World Container Design

This chapter covers advanced, modern C++ container usage patterns applicable in:
- Game engines - Financial systems - High-frequency trading - Real-time systems - Data-intensive compute workloads

8.1 Struct of Arrays (SoA) vs Array of Structs (AoS)

8.1.1 AoS Example

```
struct Point { float x,y,z; };
std::vector<Point> pts;
```

8.1.2 SoA Example

```
struct Points {
    std::vector<float> x, y, z;
```

```
};
```

8.2 When SoA Wins

- SIMD vectorization.
- Better cache locality.

8.3 Small Buffer Optimization (SBO)

Some containers (string) store small data inline.

8.4 Custom Allocators for Performance

Used heavily in:

- Game engines.
- Low-latency applications.
- Embedded systems.

8.5 Memory Pools with Containers

```
std::pmr::vector<int> v{pmr_resource};
```

8.6 Container Fusion Hybrid Structures

Examples:

- flat_map (sorted vector)
- boost::container::vector

8.7 Views and Ranges Pipelines

Transforming data without copying.

```
auto r = v
     std::views::filter([](int x) return x>10; )
  std::views::transform([](int x){ return x*2; });
```

Exercises — Chapter 8

1. Explain when SoA layout outperforms AoS.
2. Build a small SoA structure using vectors.
3. Demonstrate a memory pool container using pmr::vector.
4. Compare flat_map with std::map.
5. Show a real-world use of ranges pipelines.
6. Write a simulation loop using vector for high performance.
7. Create a min-heap scheduler using priority_queue.
8. Explain the concept of hybrid containers.
9. Show code using SBO behavior.
10. Why do real-time systems avoid reallocation?

Solutions — Chapter 8

1. SoA benefits SIMD and contiguous access.
2. ```
   struct SoA{ std::vector<float>x,y,z; };
   ```
3. ```
   std::pmr::vector<int> v{pmr_resource};
   ```
4. flat_map uses sorted vector → faster traversal.
5. ```
   auto r = v  std::views::filter(...)  std::views::transform(...);
   ```
   ```

6. Use vector for game entities.

7. ```
   auto cmp = ...;
   priority_queue<Task, Vec, decltype(cmp)> pq(cmp);
   ```

8. Combining strengths of different structures.

9. Small strings stored inline.

10. Memory determinism is critical.
   ```

Final Conclusion

STL containers are one of the most powerful and elegant abstractions in the C++ language. Mastering them requires understanding their memory models, performance characteristics, iterator categories, and real-world engineering patterns.

This booklet has taken you from foundational container theory, through detailed study of sequence, associative, unordered, and adaptor containers, and finally into highly advanced performance and architecture-focused patterns.

With exercises and solutions, you should now be fully equipped to:

- Choose the right container for any professional scenario.
- Optimize containers for speed, memory, and stability.
- Build real-time, high-performance, and large-scale systems.
- Apply modern C++20/23 patterns across all container types.

Mastery of STL containers is mastery of Modern C++ itself.

References

1. ISO/IEC 14882:2020 — Programming Languages — C++.
2. ISO/IEC 14882:2023 — Programming Languages — C++.
3. Bjarne Stroustrup, “The C++ Programming Language”, 4th Edition.
4. Nicolai M. Josuttis, “C++20: The Complete Guide”, 2021.
5. David Vandevoorde, Nicolai Josuttis, Douglas Gregor, “C++ Templates: The Complete Guide”, 2nd Edition.
6. Anthony Williams, “C++ Concurrency in Action”, 2nd Edition.
7. Eric Niebler and Casey Carter, “C++ Ranges Design and Implementation”, C++ Committee Papers.
8. CppReference Documentation (2020–2024).
9. ISO C++ WG21 Papers (2020–2024).
10. LLVM and GCC Internals Manuals (2020–2024).