# C++ STL Masterclass (Modern C++17/20)

This is a practical, example‑heavy guide to the C++ Standard Template Library (STL): containers, iterators (and a peek at ranges), algorithms, function objects, utilities, and common patterns. Code is modern C++ (C++17+), but you can compile most examples with C++14.

## 0) What is the STL?

STL is the core of C++’s Standard Library built around **generic programming**: - **Containers** store elements (e.g., vector, map). - **Iterators** are generalized pointers to traverse containers. - **Algorithms** operate over iterator ranges (e.g., sort, accumulate). - **Function objects (functors)**, lambdas, and **comparators** customize behavior. - **Allocators** manage memory (rarely customized; know they exist).

Philosophy: *Separate data structures (containers) from operations (algorithms) connected by iterators.*

## 1) Containers at a Glance

### Sequence Containers

* array<T, N> – fixed size, stack allocation semantics, contiguous.
* vector<T> – dynamic array, contiguous; **default workhorse**.
* deque<T> – double-ended queue, fast push/pop at both ends, not contiguous.
* list<T> – doubly linked list, stable iterators, slow random access.
* forward\_list<T> – singly linked list, minimal overhead, forward iterators only.

### Container Adapters

* stack<T, Container=deque<T>>
* queue<T, Container=deque<T>>
* priority\_queue<T, Container=vector<T>, Compare=less<T>>

### Associative Containers (ordered, tree-based: typically Red‑Black Tree)

* set<T> / multiset<T> – unique / duplicate keys.
* map<Key, T> / multimap<Key, T> – key→value, unique / duplicate keys.

### Unordered Containers (hash table)

* unordered\_set<T> / unordered\_multiset<T>
* unordered\_map<Key, T> / unordered\_multimap<Key, T>

### Utility/String-ish

* basic\_string (std::string, std::u16string, …), string\_view (non‑owning), bitset, span (C++20, non‑owning view of contiguous memory).

## 2) Big‑O Cheat Sheet & When to Choose What

| Container | Access | Insert (end) | Insert (middle) | Erase (middle) | Find | Memory | Notes |
| --- | --- | --- | --- | --- | --- | --- | --- |
| array | O(1) | – | – | – | – | low | fixed size |
| vector | O(1) amortized | O(1) amortized (end) | O(n) | O(n) | O(n) | low | contiguous; best cache; invalidates on reallocation |
| deque | O(1) | O(1) ends | O(n) | O(n) | O(n) | moderate | fast push\_front/back |
| list | O(n) | O(1) with iterator | O(1) with iterator | O(1) with iterator | O(n) | high | stable iterators; no random access |
| forward\_list | O(n) | O(1) after pos | O(1) after pos | O(1) after pos | O(n) | low | singly‑linked |
| set/map | O(log n) | O(log n) | O(log n) | O(log n) | O(log n) | moderate | ordered; iterates in order |
| unordered\_set/map | average O(1) | avg O(1) | avg O(1) | avg O(1) | avg O(1) | can be high | hash quality & load factor matter |
| priority\_queue | top O(1) | push O(log n) | – | pop O(log n) | – | low | heap‑based |

**Quick picks** - Need random access & speed? → vector. - Need order by key? → map/set. - Need constant‑time average lookup by key? → unordered\_map/unordered\_set. - Need frequent push/pop at both ends? → deque. - Need stable references/iterators across inserts? → list (or node‑based maps/sets).

## 3) Iterators (and a peek at Ranges)

Iterator categories (in increasing power): - **Input/Output**: one‑pass read/write. - **Forward**: multi‑pass single‑direction. - **Bidirectional**: -- allowed (e.g., list, set). - **Random Access**: +, -, [] (e.g., vector, deque). - **Contiguous** (C++20): memory contiguous (vector, string, array).

std::vector<int> v{1,2,3};  
for (auto it = v.begin(); it != v.end(); ++it) {  
 \*it += 10; // mutate through iterator  
}

**Range‑based for** (C++11+):

for (int &x : v) x \*= 2;

**C++20 ranges** (<ranges>): algorithm + pipe syntax on ranges

#include <ranges>  
#include <vector>  
#include <algorithm>  
  
auto odds\_doubled = std::vector{1,2,3,4,5}  
 | std::views::filter([](int x){return x%2;})  
 | std::views::transform([](int x){return x\*2;});  
  
std::vector<int> out;  
std::ranges::copy(odds\_doubled, std::back\_inserter(out));

## 4) Algorithm Essentials (Header: <algorithm>, <numeric>, <ranges>)

**Non‑modifying**: all\_of, any\_of, none\_of, for\_each, count, find, equal, mismatch.

**Modifying**: copy, move, swap\_ranges, fill, transform, generate, replace, remove, unique.

**Partitioning**: partition, stable\_partition, partition\_point.

**Sorting**: sort, stable\_sort, partial\_sort, nth\_element, is\_sorted, inplace\_merge.

**Binary search family** (require sorted range): lower\_bound, upper\_bound, equal\_range, binary\_search.

**Set algorithms** (sorted ranges): set\_union, set\_intersection, set\_difference, set\_symmetric\_difference.

**Heap**: make\_heap, push\_heap, pop\_heap, sort\_heap.

**Permutations**: next\_permutation, prev\_permutation.

**Numeric** (<numeric>): accumulate, reduce (C++17), inner\_product, partial\_sum, exclusive\_scan (C++17), iota.

**Example: Top‑K with nth\_element**

std::vector<int> v{7,1,4,9,3,8,2,6,5};  
size\_t k = 3; // smallest 3  
std::nth\_element(v.begin(), v.begin()+k, v.end());  
v.resize(k); // first k elements are the k smallest (unordered within k)

**Erase‑remove idiom**

std::vector<int> v{1,2,3,2,4};  
v.erase(std::remove(v.begin(), v.end(), 2), v.end()); // remove all 2s

**Transform**

std::transform(v.begin(), v.end(), v.begin(), [](int x){ return x\*x; });

## 5) vector Deep Dive

std::vector<int> v; // empty  
v.reserve(100); // avoid reallocation (performance)  
for (int i = 0; i < 10; ++i) v.push\_back(i);  
  
v.emplace\_back(42); // constructs in place  
  
// iteration  
for (const auto &x : v) {/\*...\*/}  
  
// insert/erase mid (O(n))  
v.insert(v.begin()+3, 999);  
v.erase(v.begin()+5);  
  
// shrink  
v.shrink\_to\_fit();  
  
// data pointer (contiguous)  
int \*p = v.data(); p[0] = 123;

**Iterator invalidation (vector)** - Inserting may reallocate → **all** iterators/references invalidated. - Erasing a single element invalidates iterators **from erased pos to end**.

**Tip:** Use indices or std::size\_t when you need stability across reallocations.

## 6) deque, list, forward\_list

* **deque**: Fast push\_front and push\_back; random access allowed, but memory is segmented.
* **list**: Splice without copying elements:

std::list<int> a{1,2,3}, b{4,5};  
a.splice(a.begin(), b); // moves all from b to front of a, O(1)

* **forward\_list**: Singly linked; use insert\_after, erase\_after.

**When to use list?** Rarely. Prefer vector unless you *truly* need stable iterators, frequent middle insert/erase with iterator positions, or splicing between lists.

## 7) Ordered Maps/Sets (map, set)

std::map<std::string, int> freq;  
freq["apple"]++; // inserts default 0, then ++  
  
// Preferred: avoid default insert when not needed  
freq.insert({"banana", 3});  
  
// Find  
if (auto it = freq.find("apple"); it != freq.end()) {  
 it->second += 10;  
}  
  
// Iterate in key order  
for (auto &[k,v] : freq) {  
 // structured binding, C++17  
}  
  
// Bounds  
auto it = freq.lower\_bound("b"); // first key not less than "b"

* set/map keep keys ordered by comparator (std::less by default). Supply custom comparator for custom order or case‑insensitive strings.

struct CaseLess {  
 bool operator()(const std::string& a, const std::string& b) const {  
 return std::lexicographical\_compare(  
 a.begin(), a.end(), b.begin(), b.end(),  
 [](char x, char y){ return std::tolower(x) < std::tolower(y); });  
 }  
};  
std::map<std::string, int, CaseLess> m;

**Duplicates?** Use multimap / multiset. Retrieve all with equal\_range(key).

## 8) Unordered Maps/Sets (Hash Tables)

#include <unordered\_map>  
  
std::unordered\_map<std::string, int> freq;  
freq.reserve(1024); // reduce rehashes  
  
freq["cat"]++;  
freq.insert({"dog", 2});  
  
if (auto it = freq.find("cat"); it != freq.end()) {  
 // avg O(1)  
}  
  
// Custom hash for a struct  
struct Point {int x,y;};  
struct Hash {  
 size\_t operator()(const Point& p) const noexcept {  
 return std::hash<int>()(p.x) \* 1315423911u ^ std::hash<int>()(p.y);  
 }  
};  
struct Eq {  
 bool operator==(const Point& a, const Point& b) const noexcept {  
 return a.x==b.x && a.y==b.y;  
 }  
};  
std::unordered\_set<Point, Hash, Eq> S;

**Load factor & rehashing**: bucket\_count, load\_factor(), rehash(n), reserve(n) influence performance.

## 9) Adapters: stack, queue, priority\_queue

std::stack<int> st; st.push(1); st.top(); st.pop();  
std::queue<int> q; q.push(1); q.front(); q.pop();  
  
// Max‑heap by default (largest on top)  
std::priority\_queue<int> pq;  
pq.push(5); pq.push(1); pq.push(10);  
// Min‑heap  
std::priority\_queue<int, std::vector<int>, std::greater<int>> minpq;

## 10) Strings & string\_view

std::string s = "hello";  
s += " world";  
  
// find/replace  
auto pos = s.find("lo");  
if (pos != std::string::npos) s.replace(pos, 2, "LO");  
  
// string\_view: non‑owning, cheap slice  
std::string text = "abcdef";  
std::string\_view sv(text);  
auto sub = sv.substr(2,3); // "cde"

**Warning:** string\_view doesn’t own data → dangling if source dies.

## 11) Smart Iteration Utilities

* std::begin(c), std::end(c) work for arrays too.
* Insert iterators: back\_inserter(v), front\_inserter(dq), inserter(c, it).
* Stream iterators for quick IO glue:

std::istream\_iterator<int> in(std::cin), eof;  
std::vector<int> v(in, eof);  
std::ostream\_iterator<int> out(std::cout, " ");  
std::copy(v.begin(), v.end(), out);

## 12) Common Patterns You’ll Reuse

### 12.1 Counting/Frequency Map

std::unordered\_map<int,int> cnt;  
for (int x : v) cnt[x]++;

### 12.2 Sorting with Custom Key (use lambda)

struct Item{int id; std::string name; int score;};  
std::vector<Item> a;  
std::sort(a.begin(), a.end(), [](const Item& A, const Item& B){  
 if (A.score != B.score) return A.score > B.score; // desc by score  
 return A.name < B.name; // tie‑break  
});

### 12.3 Stable Partition to Move Odds to Front (preserve order)

std::stable\_partition(v.begin(), v.end(), [](int x){ return x%2; });

### 12.4 Two‑sum using unordered\_map

std::pair<int,int> two\_sum(const std::vector<int>& a, int target){  
 std::unordered\_map<int,int> idx; // value -> index  
 for (int i = 0; i < (int)a.size(); ++i){  
 if (auto it = idx.find(target - a[i]); it != idx.end())  
 return {it->second, i};  
 idx[a[i]] = i;  
 }  
 return {-1,-1};  
}

### 12.5 Deduplicate & Sort

std::sort(v.begin(), v.end());  
v.erase(std::unique(v.begin(), v.end()), v.end());

## 13) Iterator Invalidation Rules (must‑know)

* **vector**: push\_back may invalidate *all*; erase invalidates from point to end.
* **deque**: inserting/erasing anywhere except ends may invalidate all; push/pop at ends may invalidate iterators to ends.
* **list/forward\_list**: iterators/reference remain valid except for erased elements.
* **Ordered/unordered maps/sets**: insert doesn’t invalidate iterators; erase invalidates only erased elements; **rehash** in unordered containers invalidates all iterators.

## 14) emplace vs insert vs push

* push\_back(x) copies/moves x into container.
* emplace\_back(args...) constructs in place → avoids temporary.
* insert inserts existing element(s) or range at pos.

Example (avoid constructing std::pair twice):

std::map<int,std::string> m;  
m.emplace(1, "one"); // constructs the pair inside the map

## 15) Comparators & Custom Ordering

* Comparator must define a **strict weak ordering**.
* For priority\_queue with custom compare (min‑heap of pairs by second):

using P = std::pair<int,int>;  
std::priority\_queue<P, std::vector<P>,  
 std::function<bool(const P&, const P&)>> pq(  
 [](const P& a, const P& b){ return a.second > b.second; });

Prefer transparent comparators for heterogeneous lookup (C++14+):

struct StrLess {  
 using is\_transparent = void; // enables lookup by string\_view  
 bool operator()(std::string\_view a, std::string\_view b) const { return a < b; }  
};  
std::set<std::string, StrLess> S;  
S.find("abc"sv); // no temporary std::string

## 16) Hash Customization (Unordered Containers)

* Provide both Hash and KeyEqual when your key is a struct.
* Combine fields using standard hashes; consider boost::hash\_combine or a simple mixing constant.

struct Key{int a; int b;};  
struct KeyHash {  
 size\_t operator()(const Key& k) const noexcept {  
 size\_t h1 = std::hash<int>{}(k.a);  
 size\_t h2 = std::hash<int>{}(k.b);  
 return h1 ^ (h2 + 0x9e3779b97f4a7c15ULL + (h1<<6) + (h1>>2));  
 }  
};  
struct KeyEq {  
 bool operator()(const Key& x, const Key& y) const noexcept {  
 return x.a==y.a && x.b==y.b;  
 }  
};  
std::unordered\_map<Key,int,KeyHash,KeyEq> M;

## 17) Numeric & Utility Goodies

#include <numeric>  
std::vector<int> v{1,2,3,4};  
int sum = std::accumulate(v.begin(), v.end(), 0);  
  
#include <bitset>  
std::bitset<8> b(0b10110100);  
b.flip(0);  
  
#include <tuple>  
auto tup = std::make\_tuple(1, 2.5, "hi");  
auto [i, d, s] = tup; // structured binding (C++17)  
  
#include <optional>  
std::optional<int> maybe;  
maybe = 42;  
if (maybe) {/\*...\*/}  
  
#include <variant>  
std::variant<int, std::string> var = 5;  
var = std::string("ok");

## 18) Parallel Algorithms (C++17) — <execution>

#include <execution>  
std::sort(std::execution::par, v.begin(), v.end());

Policies: seq, par, par\_unseq. Use only with safe operations (no data races, iterators valid, etc.).

## 19) Ranges (C++20) — quick tour

* Headers: <ranges>, namespace std::ranges / std::views.
* Algorithms become range‑aware: std::ranges::sort(vec);
* Views are lazy, composable. Common views: filter, transform, take, drop, iota.

#include <ranges>  
#include <iostream>  
for (int x : std::views::iota(1, 10) | std::views::filter([](int x){return x%3==0;}))  
 std::cout << x << ' '; // 3 6 9

## 20) Practical Gotchas & Tips

* Prefer vector unless a measurable need says otherwise.
* Pre‑reserve for known sizes: v.reserve(n);.
* Don’t keep iterators across operations that may invalidate them.
* Use auto + structured bindings for clarity:
* for (auto &[k, v] : mymap) { /\*...\*/ }
* For map/set, avoid operator[] on map when you don’t want insertion; use find.
* Erase while iterating safely:
* for (auto it = v.begin(); it != v.end(); ) {  
   if (\*it % 2 == 0) it = v.erase(it); else ++it;  
  }
* Prefer algorithm + iterator style over manual loops when possible (clearer + fewer bugs).
* Use span and string\_view for non‑owning views to avoid copies.

## 21) Mini‑Recipes (copy‑paste ready)

**Top N largest elements**

std::nth\_element(v.begin(), v.end()-N, v.end());  
std::vector<int> topN(v.end()-N, v.end());  
std::sort(topN.begin(), topN.end(), std::greater<>());

**Median of vector**

auto mid = v.begin() + v.size()/2;  
std::nth\_element(v.begin(), mid, v.end());  
int median = \*mid;

**K‑way merge (merge multiple sorted vectors)**

struct Node{int val, i, j;};  
auto cmp = [](const Node& a, const Node& b){ return a.val > b.val; };  
std::priority\_queue<Node, std::vector<Node>, decltype(cmp)> pq(cmp);

**Group by key (stable)**

std::stable\_sort(a.begin(), a.end(), [](auto &x, auto &y){return x.key < y.key;});  
auto it = a.begin();  
while (it != a.end()) {  
 auto jt = std::upper\_bound(it, a.end(), it->key, [](auto k, auto &obj){return k < obj.key;});  
 // [it, jt) is the group for key it->key  
 it = jt;  
}

## 22) Complexity Guarantees Snapshot

* std::sort average/worst O(n log n); stable\_sort worst O(n log^2 n) (often O(n log n)).
* unordered\_\* average O(1) for find/insert/erase; worst O(n) (bad hash or many collisions).
* map/set operations are O(log n) due to balanced trees.
* vector::push\_back amortized O(1); reallocation doubles capacity (implementation‑dependent growth factor, commonly 1.5–2x).

## 23) Practice Exercises (with hints)

1. **Frequency of words**: Read N words, print top‑k by frequency (desc), ties by lexicographic order. *Hints:* unordered\_map, then move into vector<pair<string,int>>, sort with custom comparator, or use partial\_sort for top‑k.
2. **Interval scheduling**: Given intervals [l,r), select max non‑overlapping. *Hints:* sort by end; greedy.
3. **Distinct in sliding window**: Count distinct numbers in each window of size W. *Hints:* unordered\_map counts, slide with -- and ++.
4. **LRU cache**: Implement with list + unordered\_map<Key, list::iterator>.
5. **Autocomplete**: Given dictionary, return words with prefix p. *Hints:* store in vector, sort, then use lower\_bound on prefix ranges.

*If you want, we can turn these into template files you can compile and run.*

## 24) Quick Reference (Headers)

* <vector>, <array>, <deque>, <list>, <forward\_list>, <stack>, <queue>
* <set>, <map>, <unordered\_set>, <unordered\_map>
* <algorithm>, <numeric>, <iterator>, <functional>
* <string>, <string\_view>, <span> (C++20), <bitset>
* <tuple>, <optional>, <variant>
* <ranges> (C++20), <execution> (C++17)

## 25) STL Interview‑Style Traps You Should Nail

* **Explain erase‑remove:** why two calls? (remove shifts, returns new logical end; erase shrinks container.)
* **map vs unordered\_map:** order/log‑n vs avg O(1), iterates unordered.
* **stable\_sort vs sort:** stability matters when key ties must preserve input order.
* **lower\_bound/upper\_bound/equal\_range:** correct usage and invariants.
* **Why vector usually beats list even for many inserts?** Cache locality dominates.
* **What invalidates what?** Be precise per container.

## 26) Next Steps

* Pick 2–3 exercises above and implement.
* Convert loops to algorithms (std::transform, std::accumulate, etc.).
* Try C++20 ranges on a small project (log filter, CSV processing).

### Need a printable cheat‑sheet PDF or ready‑to‑run example files? Ask and I’ll generate them here.