ChatGPT

C++ STL Containers Cheat Sheet

The C++ Standard Template Library (STL) provides a collection of generic containers and algorithms [1]. STL **containers** are template classes that implement common data structures (dynamic arrays, linked lists, trees, hash tables, etc.) [2] [3]. They can be divided into categories: **Sequential containers** (vector, list, deque), **Container adapters** (stack, queue, priority_queue), **Associative containers** (set, map), and **Unordered containers** (unordered_set, unordered_map).

Vector

Vectors are dynamic arrays that store elements contiguously. They support fast random access and can grow automatically [4]. Inserting or removing at the end (`push_back`, `pop_back`) is **amortized O(1)** (due to occasional reallocations) [5], but inserting/removing at other positions is **O(n)**.

- **Declaration:**

```
std::vector<T> v;                   // empty vector of T
std::vector<int> a = {1, 2, 3};     // initializer list
std::vector<T> v2(10);              // size 10 (default-initialized)
```

- **Common operations:**

- `v.push_back(x)`, `v.pop_back()`, `v.front()`, `v.back()`, `v.size()`, `v.capacity()`, `v.empty()`.
- Access via `v[i]` or `v.at(i)`.
- `v.insert(pos, x)`, `v.erase(pos)`, `v.clear()`, `v.reserve(n)` (pre-allocate).

- **Iteration:**

- Iterator loop: `for (auto it = v.begin(); it != v.end(); ++it) { /* use *it */ }`.
- Range-based for: `for (auto &x : v) { /* use x */ }`.

- **Example:**

```
std::vector<int> nums = {10, 20, 30};
nums.push_back(40);
for (int n : nums) {
  std::cout << n << " ";
```

```
}
// Output: 10 20 30 40
```

- **Notes:**

- Elements are stored in a **single contiguous array** [5] . Growing the vector may reallocate and move elements.
- Fast random access (pointer arithmetic like a C array), but insertion or erasure in the middle or front is costly (O(n)).

List

`std::list` is a **doubly-linked list**. It allows **constant-time insertion and removal** at any position (given an iterator) and bidirectional iteration [6] . However, it has **no random access** (no `list[i]`) and each element carries extra pointer overhead [7] .

- **Declaration:**

```
std::list<T> l;
std::list<std::string> names;
```

- **Common operations:**

- `l.push_back(x)`, `l.push_front(x)`, `l.pop_back()`, `l.pop_front()`.
- `l.insert(pos, x)`, `l.erase(pos)`, `l.splice(dest_it, other_list, it)` (move elements between lists).
- `l.front()`, `l.back()`, `l.size()`, `l.empty()`.
- `l.sort()`, `l.reverse()`, `l.remove(x)`, `l.remove_if(pred)`, `l.unique()` (remove adjacent duplicates).

- **Iteration:**

- Bidirectional iterators: `for (auto it = l.begin(); it != l.end(); ++it) ...`.
- Range-based: `for (auto &x : l) ...`.

- **Example:**

```
std::list<int> lst = {5, 1, 4};
lst.push_front(10);
for (int x : lst) std::cout << x << " ";
// Output: 10 5 1 4
```

• **Notes:**

• Each element is a separate node with links to neighbors [6].
• No direct indexing (random access is O(n)) [7].
• Best for frequent insert/erase in the middle; poor for random access or cache locality.

Deque

`std::deque` (double-ended queue) is like a vector but allows **fast insertion and deletion at both ends** [8]. It provides random access (like vector), but elements may be stored in separate chunks rather than one contiguous block [8].

• **Declaration:**

```
std::deque<T> dq;
std::deque<int> d = {1, 2, 3};
```

• **Common operations:**

• `dq.push_back(x)`, `dq.push_front(x)`, `dq.pop_back()`, `dq.pop_front()`.
• `dq.front()`, `dq.back()`, `dq[i]`, `dq.at(i)`, `dq.size()`, `dq.empty()`.
• `dq.insert`, `dq.erase` (inserting/removing in middle is slower, O(n)).

• **Iteration:**

• Random-access iterators: `for (auto it = dq.begin(); it != dq.end(); ++it) ...`.
• Range-based: `for (auto &x : dq) ...`.

• **Example:**

```
std::deque<char> dq;
dq.push_back('a');
dq.push_front('z');
for (char c : dq) std::cout << c << " ";
// Output: z a
```

• **Notes:**

• Provides functionality similar to `std::vector`, but with efficient operations at **both** ends [8].
• Elements are not guaranteed contiguous (deque is segmented) [8].
• Insertion/removal at ends is O(1); in the middle is O(n).

Stack

`std::stack` is a container adaptor (default underlying container is `std::deque<T>`) providing **LIFO** (last-in, first-out) operations [9].

- **Declaration:**

```
std::stack<T> st;
std::stack<int> numbers;
```

- **Common operations:**

- `st.push(x)`, `st.pop()`, `st.top()`, `st.empty()`, `st.size()`.
- No iterators or indexing; only access the top element.

- **Example:**

```
std::stack<std::string> st;
st.push("hello");
st.push("world");
std::cout << st.top(); // world
st.pop();
std::cout << st.top(); // hello
```

- **Notes:**

- Internally uses an underlying container (deque, vector, or list) that supports `push_back` / `pop_back` [10].
- All operations are **O(1)**.
- Use for LIFO scenarios (undo stack, DFS, etc.).

Queue

`std::queue` is a container adaptor for **FIFO** (first-in, first-out) operations, using an underlying container (default `std::deque<T>`) [11].

- **Declaration:**

```
std::queue<T> q;
std::queue<int> q;
```

• **Common operations:**

• `q.push(x)`, `q.pop()`, `q.front()`, `q.back()`, `q.empty()`, `q.size()`.

• **Example:**

```
std::queue<int> q;
q.push(1);
q.push(2);
std::cout << q.front(); // 1
q.pop();
std::cout << q.front(); // 2
```

• **Notes:**

• Underlying container must support `push_back` and `pop_front` [12].
• FIFO order: elements are popped from the front, inserted at the back.
• Useful for breadth-first search, buffering tasks, etc.

Priority Queue

`std::priority_queue` is a container adaptor for **max-heaps** (by default), where `top()` is always the largest element [13].

• **Declaration:**

```
std::priority_queue<T> pq;                                          // max-
heap (default)
std::priority_queue<int,std::vector<int>,std::greater<int>> minpq; // min-
heap
```

• **Common operations:**

• `pq.push(x)`, `pq.pop()`, `pq.top()`, `pq.empty()`, `pq.size()`.

• **Example:**

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

```
pq.pop();
std::cout << pq.top(); // 5
```

- **Notes:**

- Underlying container (default `std::vector<T>`) must support random access [14].
- Insertion/pop are **O(log n)**; `top()` is O(1).
- Default compares with `std::less`, so highest element is on top.

Pair

`std::pair<T1, T2>` is a simple struct that **stores two heterogeneous objects** as a single unit [2].

- **Declaration:**

```
std::pair<T1, T2> p;
std::pair<std::string, int> person;
```

- **Initialization:**

- Constructor: `std::pair<int,double> p(1, 2.5);`.
- Using `std::make_pair`: `auto pr = std::make_pair("key", 42);`.

- **Access:**

- `p.first` (type `T1`), `p.second` (type `T2`).

- **Example:**

```
auto pr = std::make_pair(std::string("apple"), 3);
std::cout << pr.first << " -> " << pr.second << "\n"; // apple -> 3
```

- **Notes:**

- Commonly used to return two values from a function or as map elements (map elements are of type `pair<const Key, T>`).
- No extra overhead beyond storing the two values.

Set

`std::set<T>` is an **ordered** associative container of **unique keys**. Elements are sorted by key (default uses `std::less<T>`) [15].

- **Declaration:**

```
std::set<T> s;
std::set<int> s = {3,1,4};
```

- **Common operations:**

- `s.insert(x)`, `s.erase(x)` (or by iterator).
- `s.find(x)`, `s.count(x)` (1 if exists, else 0).
- `s.lower_bound(x)`, `s.upper_bound(x)`, `s.equal_range(x)`.
- `s.begin()`, `s.end()`, `s.clear()`, `s.empty()`, `s.size()`.

- **Iteration:**

- Iterates in ascending order of keys.
- Example: `for (auto it = s.begin(); it != s.end(); ++it) { /* *it is key */ }`.

- **Example:**

```
std::set<int> st = {10, 5, 20};
for (int x : st) std::cout << x << " ";
// Output: 5 10 20
```

- **Notes:**

- Sorted container (often a red-black tree) [15].
- Access, insertion, and deletion are **O(log n)**.
- Allows ordered traversal and range queries; slower per-access than hash containers [16].

Unordered Set

`std::unordered_set<T>` is an **unordered** associative container (hash table) of **unique keys** [17].

- **Declaration:**

```
std::unordered_set<T> us;
std::unordered_set<std::string> words;
```

- **Common operations:**

- `us.insert(x)`, `us.erase(x)`, `us.find(x)`, `us.count(x)`.
- `us.bucket_count()`, `us.rehash(n)`, `us.reserve(n)`.
- `us.begin()`, `us.end()`, etc.

- **Example:**

```
std::unordered_set<int> us = {1, 2, 2, 3};
std::cout << us.count(2); // 1 (duplicates ignored)
```

- **Notes:**

- Elements are not sorted, stored in buckets by hash [18].
- Average **O(1)** for lookup/insert/delete (worst-case O(n)).
- Faster than `std::set` for lookups; iteration order is arbitrary.

Map

`std::map<Key, T>` is an **ordered** associative container of **key-value pairs** [19] with unique keys.

- **Declaration:**

```
std::map<Key, T> m;
std::map<std::string,int> phonebook;
```

- **Common operations:**

- `m.insert({k,v})` or `m[k] = v`.
- `m.find(k)`, `m.count(k)`, `m.erase(k)`.
- `m.lower_bound(k)`, `m.upper_bound(k)`.
- `m.begin()`, `m.end()`, `m.clear()`, etc.

- **Access:**

- `m[key]` (creates default if key not present) [20].
- `m.at(key)` (throws exception if key not present).

- Iterate pairs: `for (auto &p : m) { /* p.first is key, p.second is value */ }`.

- **Example:**

```
std::map<std::string,int> mp;
mp["apple"] = 3;
mp.insert({"banana", 5});
for (auto &p : mp) {
  std::cout << p.first << " -> " << p.second << "\n";
}
```

- **Notes:**

- Keys are sorted; iteration is in ascending key order.
- Operations are **O(log n)**.
- Slower than `unordered_map` for access, but supports ordered traversal and range queries [21].

Unordered Map

`std::unordered_map<Key,T>` is an **unordered** associative container (hash table) of **key-value pairs** with unique keys [22].

- **Declaration:**

```
std::unordered_map<Key, T> um;
std::unordered_map<std::string,int> freq;
```

- **Common operations:**

- `um[key] = value`, `um.insert({k,v})`, `um.find(k)`, `um.erase(k)`.
- `um.rehash(n)`, `um.reserve(n)`.
- `um.begin()`, `um.end()`, etc.

- **Access:**

- `um[key]` (inserts default if absent).
- Iterate: `for (auto &p : um) { /* p.first is key, p.second is value */ }`.

- **Example:**

```cpp
std::unordered_map<std::string,int> um;
um["a"] = 1;
um["b"] = 2;
std::cout << um["a"]; // 1
```

- **Notes:**

- No ordering (keys hashed) [22] .
- Average **O(1)** lookup/insert/delete; iteration is unordered.
- Faster than `std::map` for lookups; use when ordering is not required [23] .

Algorithms and Iteration

The `<algorithm>` and `<numeric>` libraries provide common algorithms that operate on iterator ranges:

- **Sorting:** `std::sort(begin, end);` sorts a range in ascending order [24] (requires random-access iterators).
- **Searching:** `std::find(begin, end, value);` returns an iterator to the first matching element or `end` [25] .
- **Other algorithms:** `std::count`, `std::reverse`, `std::min_element`, `std::accumulate` (sums values) [26] , etc.
- **Loops:**
- Index-based: `for(int i = 0; i < n; ++i)`.
- Iterator-based: `for(auto it = c.begin(); it != c.end(); ++it)`.
- Range-based: `for(auto &x : c)`.
- **Element access:**
- Sequence containers: `.front()`, `.back()`, `operator[]`, `.at()`.
- Associative containers: access key with `it->first`, value with `it->second`.
- **Example:**

```cpp
std::vector<int> a = {3,1,4,1,5};
std::sort(a.begin(), a.end());            // [1,1,3,4,5]
auto it = std::find(a.begin(), a.end(), 4);
if (it != a.end()) std::cout << *it;      // 4
int sum = std::accumulate(a.begin(), a.end(), 0); // 14
```

- **Notes:**
- Algorithms operate on iterator ranges of containers. E.g., `std::sort` requires random-access iterators (works on vector/deque, not on list).
- Consider complexity and iterator requirements (e.g. `std::binary_search` needs a sorted range).

Summary Table

Container	Structure	Ordered	Access	Insertion/ Delete	Notes
vector	Dynamic array (contiguous)	No	O(1) random; O(1) at ends (amortized) [5]	O(1) amortized at end; O(n) elsewhere	Fast random access; use as dynamic array.
list	Doubly-linked list	Yes (insertion order)	No random (sequential)	O(1) (given iterator)	Fast insert/erase anywhere; no random access [27].
deque	Segmented array (blocks)	No	O(1) random	O(1) at ends; O(n) middle	Like vector with efficient front/ back insertion [8].
stack	Container adaptor (deque)	N/A	N/A (only top)	O(1) at top	LIFO (push/pop/ top only).
queue	Container adaptor (deque)	N/A	N/A (front/ back)	O(1) at ends	FIFO (push/back, pop/front).
priority_queue	Heap (vector + comparator)	N/A	O(1) top	O(log n) push/pop	Max-heap by default; largest on top.
set	Balanced BST (tree)	Yes (sorted)	O(log n) by key	O(log n)	Ordered unique keys; bidirectional iteration [15].
unordered_set	Hash table	No	O(1) avg	O(1) avg	Unordered unique keys; fast lookup [18].
map	Balanced BST (tree)	Yes (sorted)	O(log n) by key	O(log n)	Ordered key→value pairs; uses `map[key]`.

Container	Structure	Ordered	Access	Insertion/ Delete	Notes
unordered_map	Hash table	No	O(1) avg	O(1) avg	Unordered key→value; average constant-time lookup [22].

Sources: Official C++ references and documentation for each container and algorithm (see citations) provide the details above [1] [4] [6] [8] [9] [11] [13] [2] [15] [17] [19] [22] [24] [25] [26]. Each container's complexity and behavior can be confirmed in these references.

---

[1] C++ STL Cheat Sheet - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/cpp-stl-cheat-sheet/

[2] std::pair - cppreference.com
http://tool.oschina.net/uploads/apidocs/cpp/en/cpp/utility/pairhtml.html

[3] [15] [16] cplusplus.com
https://cplusplus.com/reference/set/set/

[4] cplusplus.com
https://cplusplus.com/reference/vector/vector/

[5] [8] cplusplus.com
https://cplusplus.com/reference/deque/deque/

[6] [7] [27] cplusplus.com
https://cplusplus.com/reference/list/list/

[9] [10] cplusplus.com
https://cplusplus.com/reference/stack/stack/

[11] [12] cplusplus.com
https://cplusplus.com/reference/queue/queue/

[13] [14] cplusplus.com
https://cplusplus.com/reference/queue/priority_queue/

[17] [18] cplusplus.com
https://cplusplus.com/reference/unordered_set/unordered_set/

[19] [20] [21] cplusplus.com
https://cplusplus.com/reference/map/map/

[22] [23] cplusplus.com
https://cplusplus.com/reference/unordered_map/unordered_map/

[24] cplusplus.com
https://cplusplus.com/reference/algorithm/sort/

25 cplusplus.com
https://cplusplus.com/reference/Algorithm/Find/

26 cplusplus.com
https://cplusplus.com/reference/numeric/accumulate/