C++ MASTERY GUIDE

STL, Data Structures & Algorithms

From Complete Beginner to Confident C++ Developer

WHAT YOU WILL MASTER

Pairs • Vectors • Iterators • Stack • Queue • Priority Queue
Set • MultiSet • Map • MultiMap • Unordered Containers
Sorting • Custom Comparators • Built-in Algorithms

Introduction to C++ & STL

Welcome! This guide will take you from zero knowledge of C++ to a confident understanding of the Standard Template Library (STL) — the collection of powerful, pre-built tools that every professional C++ developer uses daily.

REMEMBER

The STL (Standard Template Library) is a set of ready-made tools built into C++. Instead of writing your own data structures from scratch, you use these — they are fast, well-tested, and used everywhere in the real world.

What is #include<bits/stdc++.h>?

At the top of every file in our examples you see:

```
#include<bits/stdc++.h>
using namespace std;
```

The line `#include<bits/stdc++.h>` is a shortcut that includes EVERY C++ standard library header at once. This means you get access to vectors, maps, sets, algorithms — everything — in one line. using namespace std; means you can write `cout` instead of `std::cout` — it just removes the need to type `std::` before everything.

IMPORTANT

In professional/production code, you should include only what you need. But for learning and competitive programming, #include<bits/stdc++.h> is perfectly fine and saves a lot of time.

How C++ Programs Work

Every C++ program starts executing from the main() function. Think of it as the "entry point" — the door through which execution walks in.

Basic Program Structure

```
int main() {
    // Your code goes here
    cout << "Hello World" << endl;
    return 0;   // 0 means the program ran successfully
}
```

Key C++ output tool: `cout` (pronounced 'see-out') sends text to the screen. The `<<` operator feeds data into cout. `endl` moves to a new line (like pressing Enter).

Chapter 1: Pairs

What is a Pair?

A pair is the simplest container in C++ — it holds exactly TWO values together, like a small box with two compartments. The values can be of any type, and they don't have to be the same type.

Declaring and Using Pairs

Basic Pair

```
// Syntax: pair<Type1, Type2> name = {value1, value2};

pair<int, int> p = {1, 3};
cout << p.first;    // Outputs: 1
cout << p.second;   // Outputs: 3
```

Access the values with `.first` and `.second`. These are the only two "slots" a pair has — no more, no less.

Nested Pairs

You can put a pair inside a pair. This is called nesting. The inner pair becomes the value of one of the slots:

Nested Pairs

```
pair<int, pair<int, int>> p2 = {1, {3, 5}};

cout << p2.first;          // Outputs: 1
cout << p2.second.first;   // Outputs: 3  (first of the inner pair)
cout << p2.second.second;  // Outputs: 5  (second of the inner pair)
```

TIP

Think of it like a Russian nesting doll. The outer pair contains an int and another pair. To reach inside the inner pair, you chain .second.first or .second.second.

Arrays of Pairs

Pairs can be used just like any other data type — you can make arrays of them:

Array of Pairs

```
pair<int, int> arr[] = {{1,2}, {3,4}, {5,6}};

cout << arr[0].first;   // Outputs: 1
cout << arr[1].second;  // Outputs: 4
cout << arr[2].first;   // Outputs: 5
```

When to Use Pairs

Use Case	Example
Storing coordinates	`pair<int,int> point = {x, y};`
Key-value combos	`pair<string,int> score = {"Alice", 95};`
Returning two values	`return {minVal, maxVal};`
Sorting by two criteria	`pair<int,int>` in a vector, then sort

Chapter 2: Vectors

What is a Vector?

A vector is like a smart, resizable array. Unlike a normal C++ array whose size is fixed at creation, a vector can grow and shrink automatically. It's one of the most-used containers in all of C++.

Creating Vectors

Ways to Create Vectors

```
vector<int> v;                 // Empty vector of integers
vector<string> names;          // Empty vector of strings
vector<pair<int,int>> pairs;   // Empty vector of pairs

vector<int> v1(5, 100);  // 5 elements, each = 100 → {100, 100, 100, 100,
100}
vector<int> v2(5);       // 5 elements, each = 0   → {0, 0, 0, 0, 0}

vector<int> original = {1, 2, 3, 4, 5};
vector<int> copy(original);  // Copy of original
```

Adding Elements

Adding to a Vector

```
vector<int> v;

v.push_back(10);    // Adds 10 at the END → {10}
v.push_back(20);    // Adds 20 at the END → {10, 20}
v.emplace_back(30);// Same as push_back, but slightly faster → {10, 20, 30}
```

Both `push_back` and `emplace_back` add to the end. Use `emplace_back` when you can — it's slightly more efficient because it constructs the element directly in place instead of making a copy.

Accessing Elements

Accessing Vector Elements

```
vector<int> v = {10, 20, 30, 40, 50};

cout << v[0];       // 10  (index-based, NO bounds check)
cout << v.at(1);    // 20  (safer, throws exception if out of bounds)
cout << v.front();  // 10  (first element)
```

```
cout << v.back();    // 50  (last element)
```

⚠ IMPORTANT

v[i] is faster but dangerous — if you access an invalid index, it causes undefined behavior (your program may crash or silently produce wrong results). v.at(i) is safer because it throws an error you can catch. During learning, prefer .at() to catch mistakes early.

Useful Vector Functions

Function	What It Does	Example
`v.size()`	Returns number of elements	`v.size() → 5`
`v.empty()`	Returns true if empty	`v.empty() → false`
`v.pop_back()`	Removes last element	`{1,2,3} → {1,2}`
`v.clear()`	Removes ALL elements	`{1,2,3} → {}`
`v.swap(v2)`	Swaps contents with v2	`v↔v2 exchange`

Looping Through a Vector

Three Ways to Loop Through a Vector

```
vector<int> v = {10, 20, 30, 40};

// Method 1: Index loop (classic, like arrays)
for(int i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
}

// Method 2: Range-based for loop (cleanest — USE THIS MOST)
for(auto x : v) {
    cout << x << " ";
}

// Method 3: Iterator loop (you'll learn this next chapter)
for(auto it = v.begin(); it != v.end(); it++) {
    cout << *it << " ";
}
```

✓ TIP

The range-based for loop (for auto x : v) is your best friend. It's clean, readable, and less error-prone. The 'auto' keyword automatically detects the type — you don't have to write 'int' manually.

Erasing Elements

Erasing from a Vector

```
vector<int> v = {10, 20, 30, 40, 50, 60};

// Erase single element at index 1 (the value 20)
v.erase(v.begin() + 1);
// v is now: {10, 30, 40, 50, 60}

// Erase a RANGE [start, end)  — end is NOT deleted
v.erase(v.begin() + 1, v.end() - 1);
// Deletes index 1 to second-to-last → v is now: {10, 60}
```

REMEMBER

The range in erase(start, end) is [start, end) — this means start IS deleted, but end is NOT deleted. This 'half-open' range is a pattern used throughout the STL.

Insert Function

Inserting into a Vector

```
vector<int> v = {100, 100};  // {100, 100}

v.insert(v.begin(), 300);         // {300, 100, 100}
v.insert(v.begin() + 1, 2, 10); // insert '10' twice at index 1 → {300, 10,
10, 100, 100}

// Insert another vector's range
vector<int> extra = {50, 50};
v.insert(v.begin(), extra.begin(), extra.end());
// → {50, 50, 300, 10, 10, 100, 100}
```

Chapter 3: Iterators

What is an Iterator?

An iterator is like a cursor or pointer that points to a position inside a container. It lets you navigate through elements one by one. Every STL container supports iterators, which is why you can use the same algorithm functions on different containers.

begin() and end()

Using Iterators

```
vector<int> v = {10, 20, 30, 40, 50};

// v.begin() → points to FIRST element (value 10)
// v.end()   → points to ONE PAST the last element (NOT valid element!)

auto it = v.begin();
cout << *it;    // 10  (the * 'dereferences' — gets the actual value)

it++;           // Move iterator forward by 1
cout << *it;    // 20

it += 2;        // Move forward by 2
cout << *it;    // 40
```

IMPORTANT

NEVER dereference v.end(). It does not point to a real element — it's just a sentinel marker saying 'you've passed the last element'. Dereferencing it causes undefined behavior.

Reverse Iterators

Reverse Iteration

```
vector<int> v = {10, 20, 30, 40, 50};

auto it = v.end();  // Points PAST the last element
it--;               // Now points TO the last element
cout << *it;        // 50

// Or use rbegin/rend for reverse iteration
for(auto it = v.rbegin(); it != v.rend(); it++) {
    cout << *it << " ";  // 50 40 30 20 10
}
```

TIP

The 'auto' keyword is your best friend with iterators. Instead of writing 'vector<int>::iterator it = v.begin()', just write 'auto it = v.begin()'. C++ will figure out the type for you.

The Dereference Operator *

The * operator on an iterator gives you the actual value at that position. Think of an iterator as an address, and * as 'go to that address and give me what's there'.

Dereference Example

```
auto it = v.begin();
// 'it' is the ADDRESS of the first element
// '*it' is the VALUE at that address

cout << it;    // Prints a memory address (not useful to read)
cout << *it;   // Prints 10 (the actual value — useful!)
```

Chapter 4: Stack

What is a Stack?

A stack is a LIFO (Last In, First Out) container. Imagine a pile of plates — you always add to the top and take from the top. Whatever you added LAST is what you remove FIRST. You cannot access the middle.

Real World Analogy

A stack of plates. The plate you put on TOP is the first one you pick up. You cannot grab from the middle or bottom.

Common Uses in Programming

Undo/Redo operations, browser back button, expression evaluation, function call tracking (call stack).

Stack Operations

Stack in Action

```
stack<int> st;

st.push(1);      // Stack: [1]
st.push(2);      // Stack: [1, 2]
st.push(3);      // Stack: [1, 2, 3]
st.emplace(4);   // Stack: [1, 2, 3, 4]  (same as push, slightly faster)

cout << st.top();   // 4  (peek at top WITHOUT removing)
st.pop();           // Removes 4. Stack: [1, 2, 3]
cout << st.top();   // 3  (now top is 3)

cout << st.size();  // 3
cout << st.empty(); // 0 = false (not empty)
```

IMPORTANT

You CANNOT access elements by index with a stack! st[0] or st.at(0) do NOT work. The only element you can ever access is st.top(). This restriction is intentional — it enforces LIFO behavior.

Operation	Syntax	What it does
Add element	st.push(x) or st.emplace(x)	Adds x to the TOP
Remove top	st.pop()	Removes (and discards) top element
View top	st.top()	Returns top WITHOUT removing
Check size	st.size()	Number of elements

Check empty	st.empty()	true if no elements
Swap stacks	st1.swap(st2)	Exchange contents

Chapter 5: Queue

What is a Queue?

A queue is a FIFO (First In, First Out) container. Think of a real line at a coffee shop — the first person to arrive is the first to be served. Elements enter at the BACK and leave from the FRONT.

Queue Operations

Queue in Action

```
queue<int> q;

q.push(1);      // Queue: [1]         (front→back)
q.push(2);      // Queue: [1, 2]
q.emplace(3);   // Queue: [1, 2, 3]

// Modify the back element directly
q.back() += 5;
// Queue: [1, 2, 8]

cout << q.front();  // 1  (peek at front)
cout << q.back();   // 8  (peek at back)

q.pop();            // Removes 1 (the front). Queue: [2, 8]
cout << q.front();  // 2
```

Stack vs Queue	Details
Order	Stack = LIFO (last in, first out) | Queue = FIFO (first in, first out)
Add element	Stack: push() → to TOP | Queue: push() → to BACK
Remove element	Stack: pop() from TOP | Queue: pop() from FRONT
View only	Stack: top() | Queue: front() and back()
Real-world	Stack = undo history | Queue = print job lineup

Chapter 6: Priority Queue

What is a Priority Queue?

A priority queue always gives you the MOST IMPORTANT element first, regardless of insertion order. By default in C++, the largest element is always at the top (max-heap). You can also configure it to give the smallest element first (min-heap).

Max-Heap (Default)

Max-Heap Priority Queue

```
priority_queue<int> pq;  // Default: MAX-HEAP (largest at top)

pq.push(5);    // Heap internally arranges: {5}
pq.push(2);    // {5, 2}
pq.push(8);    // {8, 5, 2}  — 8 moved to top!
pq.emplace(1); // {8, 5, 2, 1}

cout << pq.top();  // 8  (always the LARGEST element)
pq.pop();          // Removes 8. Top becomes 5.
cout << pq.top();  // 5
```

Min-Heap (Smallest First)

Min-Heap Priority Queue

```
// To make a MIN-HEAP, add two extra template arguments:
priority_queue<int, vector<int>, greater<int>> minpq;
//                               ^^^^^^^^^^^^
//                               This flips the comparison

minpq.push(5);  // {5}
minpq.push(2);  // {2, 5}  — 2 moved to top (it's smaller!)
minpq.push(8);  // {2, 5, 8}
minpq.emplace(1);// {1, 2, 5, 8}

cout << minpq.top();  // 1  (always the SMALLEST element)
minpq.pop();          // Removes 1. Top becomes 2.
cout << minpq.top();  // 2
```

TIP

The 'greater<int>' template argument is the key difference between max-heap and min-heap. Default (no third argument) = max-heap. Adding 'greater<int>' = min-heap. Memorize this pattern — it comes up in every DSA problem.

REMEMBER

Priority queues do NOT support random access. Like stack, the only element you can access is top(). The internal heap structure is optimized for finding the min/max quickly, not for general access.

Chapter 7: Set

What is a Set?

A set stores UNIQUE elements in SORTED order automatically. If you insert 5 into a set that already contains 5, nothing happens — duplicates are silently rejected. This makes sets perfect for tracking which items you've seen.

Basic Set Operations

Set Operations

```
set<int> s;

s.insert(1);   // {1}
s.emplace(2);  // {1, 2}
s.insert(2);   // {1, 2}  ← duplicate ignored!
s.insert(4);   // {1, 2, 4}
s.insert(3);   // {1, 2, 3, 4}  ← auto-sorted!

s.erase(3);    // {1, 2, 4}

// Check if element exists
int count = s.count(2);  // Returns 1 (exists) or 0 (not found)

// Find and get iterator
auto it = s.find(2);     // Returns iterator to 2
if(it != s.end()) {
    cout << *it;         // 2
}

auto it2 = s.find(99);   // Returns s.end() (99 not found)
```

lower_bound and upper_bound

lower_bound and upper_bound

```
set<int> s = {2, 4, 6, 8, 10};

// lower_bound(x): iterator to FIRST element >= x
auto lb = s.lower_bound(5);
cout << *lb;  // 6  (first element that is >= 5)

// upper_bound(x): iterator to FIRST element > x
auto ub = s.upper_bound(6);
cout << *ub;  // 8  (first element that is strictly > 6)

// What about lower_bound for an existing element?
auto lb2 = s.lower_bound(6);
cout << *lb2; // 6  (6 >= 6, so it points to 6 itself)
```

REMEMBER

lower_bound = first element GREATER THAN OR EQUAL TO x (>=). upper_bound = first element STRICTLY GREATER THAN x (>). This distinction is critical in DSA problems.

Erasing a Range from Set

Range Erase in Set

```
set<int> s = {1, 2, 3, 4, 5};

auto it1 = s.find(2);  // Points to 2
auto it2 = s.find(5);  // Points to 5

s.erase(it1, it2);     // Erases [it1, it2) → erases 2, 3, 4 (NOT 5)

// s is now: {1, 5}
```

Chapter 8: MultiSet

What is a MultiSet?

A multiset is exactly like a set — elements are stored in sorted order — but it ALLOWS duplicates. If you insert 5 three times, all three 5s are stored. This is useful when you need to count occurrences while maintaining sorted order.

MultiSet Operations

MultiSet Behavior

```
multiset<int> ms;

ms.insert(1);  // {1}
ms.insert(1);  // {1, 1}   ← duplicate allowed!
ms.insert(1);  // {1, 1, 1}
ms.insert(2);  // {1, 1, 1, 2}
ms.insert(2);  // {1, 1, 1, 2, 2}

int cnt = ms.count(1);  // Returns 3 (three 1s exist)

// DANGER: erase(value) removes ALL occurrences!
ms.erase(1);  // Removes ALL 1s → {2, 2}

// To remove just ONE occurrence, use find() first:
ms.erase(ms.find(2));  // Removes only one 2 → {2}
```

IMPORTANT

This is a VERY common mistake: ms.erase(value) removes ALL copies of that value. To remove just one copy, always use ms.erase(ms.find(value)). This finds the first occurrence and removes only that one.

Set vs MultiSet	Key Difference
`set<int>`	Only stores UNIQUE elements. Duplicates silently ignored.
`multiset<int>`	Stores ALL elements including duplicates.
`set.count(x)`	Returns 0 or 1 (can't be more than 1)
`multiset.count(x)`	Returns actual count (0, 1, 2, 3, ...)
`erase(val)`	Both: set removes it. Multiset removes ALL copies.

Chapter 9: Unordered Set

What is an Unordered Set?

An unordered_set is like a set (unique elements only) but WITHOUT sorting. It uses a hash table internally, which makes average-case lookups O(1) — extremely fast. The tradeoff: elements have no guaranteed order when you iterate.

Unordered Set

```
unordered_set<int> us;

us.insert(5);  // {5}
us.insert(3);  // {5, 3}  or {3, 5}  — ORDER NOT GUARANTEED
us.insert(1);  // could be any order
us.insert(5);  // ignored — duplicates not allowed

// All the same functions: find, count, erase, size, empty
auto it = us.find(3);
if(it != us.end()) cout << *it;  // 3

// NO lower_bound or upper_bound — unordered!
```

Container	Sorted?	Duplicates?	Lookup Speed	Use When...
set<T>	Yes	No	O(log n)	Need sorted unique elements or bounds
multiset<T>	Yes	Yes	O(log n)	Need sorted elements with duplicates
unordered_set<T>	No	No	O(1) avg	Just need fast existence check, no order needed

Chapter 10: Map

What is a Map?

A map stores KEY-VALUE pairs, like a dictionary. Every key is unique and sorted. Given a key, you can instantly look up its value. Think of a phone book: the name (key) maps to a phone number (value).

Creating and Adding to Maps

Basic Map Usage

```
map<int, int> m;

// Three ways to add key-value pairs:
m[1] = 100;           // Using [] operator
m[2] = 200;
m.emplace(3, 300);    // Using emplace()
m.insert({4, 400});   // Using insert() with pair

// m is: {1:100, 2:200, 3:300, 4:400}

// Access values by key
cout << m[1];     // 100
cout << m[2];     // 200

// CAREFUL: m[key] CREATES the key if it doesn't exist!
cout << m[99];    // Creates key 99 with value 0, then prints 0
```

IMPORTANT

The [] operator on a map is powerful but tricky. If the key doesn't exist, it CREATES it with a default value (0 for int). This can bloat your map with unintended entries. Use m.count(key) or m.find(key) to safely check existence without creating entries.

Looping Through a Map

Iterating a Map

```
map<int, int> m = {{1, 100}, {2, 200}, {3, 300}};

// Each element 'it' is a pair: it.first=key, it.second=value
for(auto it : m) {
    cout << it.first << " -> " << it.second << endl;
}
// Output:
// 1 -> 100
// 2 -> 200
```

```
// 3 -> 300

// Or use structured bindings (C++17):
for(auto [key, value] : m) {
    cout << key << " -> " << value << endl;
}
```

Map with Complex Keys and Values

Complex Key/Value Types

```
// Map with PAIR as key
map<pair<int,int>, int> m2;
m2[{1, 2}] = 100;
m2[{3, 4}] = 200;
cout << m2[{3, 4}];  // 200

// Map with PAIR as value
map<int, pair<int,int>> m3;
m3[1] = {100, 200};
m3.insert({2, {300, 400}});
m3.emplace(3, make_pair(500, 600));

cout << m3[2].first;   // 300
cout << m3[2].second;  // 400
```

Finding Elements in a Map

Safe Element Access

```
map<int, int> m = {{1, 100}, {2, 200}, {3, 300}};

// SAFE way to check if key exists:
if(m.count(2)) {         // count returns 0 or 1 for map
    cout << m[2];        // 200
}

// Using find():
auto it = m.find(3);
if(it != m.end()) {
    cout << it->first;   // 3  (using -> instead of (*it).)
    cout << it->second;  // 300
}

// it->second is same as (*it).second — just cleaner syntax
```

TIP

When using iterators with pairs (like in map), you can use 'it->first' and 'it->second' instead of '(*it).first' and '(*it).second'. The arrow (->) is shorthand for 'dereference then access' and is much cleaner to write.

Map Variants	Key Property
`map<K,V>`	Unique keys, sorted by key, O(log n) ops
`multimap<K,V>`	Duplicate keys allowed, sorted, O(log n) ops
`unordered_map<K,V>`	Unique keys, NOT sorted, O(1) avg ops
`unordered_multimap<K,V>`	Duplicate keys, NOT sorted, O(1) avg ops

Chapter 11: Sorting & Algorithms

The sort() Function

C++'s built-in sort() is extremely fast — it uses a hybrid algorithm (introsort) that runs in O(n log n). You can sort arrays, vectors, and any random-access container with it.

Basic Sorting

Sorting Arrays and Vectors

```
int a[] = {8, 3, 5, 1, 9, 2, 7, 4};

sort(a, a + 8);               // Sort entire array ascending
// a is now: {1, 2, 3, 4, 5, 7, 8, 9}

sort(a + 2, a + 6);           // Sort only index [2, 6) ascending

sort(a, a + 8, greater<int>()); // Sort descending
// a is now: {9, 8, 7, 5, 4, 3, 2, 1}

// For vectors:
vector<int> v = {5, 3, 1, 4, 2};
sort(v.begin(), v.end());     // Ascending
sort(v.begin(), v.end(), greater<int>()); // Descending
```

Sorting Pairs

Default Pair Sorting

```
pair<int, int> arr[] = {{3, 1}, {1, 3}, {2, 2}};

// Default sort: by first element, then by second if first is equal
sort(arr, arr + 3);
// Result: {{1, 3}, {2, 2}, {3, 1}}
```

Custom Comparator

A custom comparator is a function that tells sort() how to order two elements. It returns true if the first element should come BEFORE the second.

Custom Comparator Function

```
// Sort pairs by SECOND element ascending.
// If second is equal, sort by FIRST descending.
bool myComp(pair<int,int> p1, pair<int,int> p2) {
    if(p1.second != p2.second) {
        return p1.second < p2.second;  // Sort by second ascending
```

```
    }
    return p1.first > p2.first;  // If equal second, first descending
}

pair<int,int> arr[] = {{1, 2}, {3, 1}, {2, 2}};
sort(arr, arr + 3, myComp);
// Result: {{3, 1}, {2, 2}, {1, 2}}
//          ↑ second=1    ↑ second=2 (first=2 > 1 when second equal)
```

⚠ IMPORTANT

In your code, the comp function has a bug: 'if(p1.second < p2.second) true;' — the return keyword is missing! It should be 'return true;'. Always write 'return' before your boolean expression.

Sorting Strings

Sorting Strings

```
string str = "dcba";
sort(str.begin(), str.end());
// str is now: "abcd"  (sorts characters lexicographically)

// You can also sort a vector of strings:
vector<string> words = {"banana", "apple", "cherry"};
sort(words.begin(), words.end());
// words: {"apple", "banana", "cherry"}
```

next_permutation

Generating All Permutations

```
string str = "123";  // Must be sorted to get ALL permutations

do {
    cout << str << endl;
} while (next_permutation(str.begin(), str.end()));

// Output:
// 123
// 132
// 213
// 231
// 312
// 321
```

TIP

Always sort the string/array BEFORE using next_permutation if you want ALL permutations. If you start in the middle of the sequence, you'll only get the permutations from that point forward.

max_element and min_element

Finding Min and Max

```
int a[] = {3, 1, 4, 1, 5, 9, 2, 6};

int maxVal = *max_element(a, a + 8);  // 9
int minVal = *min_element(a, a + 8);  // 1

// For vectors:
vector<int> v = {3, 1, 4, 1, 5, 9};
int maxV = *max_element(v.begin(), v.end());  // 9

// Note the * — max_element returns an ITERATOR, not the value.
// You must dereference it with * to get the actual number.
```

__builtin_popcount — Counting Set Bits

Counting Set Bits

```
// __builtin_popcount(n) counts how many 1-bits are in n

int num = 7;   // 7 in binary = 0111
cout << __builtin_popcount(7);   // 3  (three 1-bits)

cout << __builtin_popcount(6);   // 2  (6 = 0110, two 1-bits)
cout << __builtin_popcount(15);  // 4  (15 = 1111)

// For 64-bit (long long) numbers, use popcountll:
long long big = 111123456789LL;
cout << __builtin_popcountll(big);  // counts 1-bits in big number
```

Quick Reference: All Containers at a Glance

Container	Ordered?	Unique?	Access	Add/Remove	Best For
vector<T>	Insert order	No	O(1) index	O(1) back, O(n) mid	General purpose list
stack<T>	LIFO	No	top() only	O(1)	Undo, backtracking
queue<T>	FIFO	No	front/back	O(1)	BFS, job scheduling
priority_queue<T>	By value	No	top() only	O(log n)	Always get min/max fast
set<T>	Sorted	Yes	Iterator	O(log n)	Unique sorted elements
multiset<T>	Sorted	No	Iterator	O(log n)	Sorted with duplicates
unordered_set<T>	None	Yes	Iterator	O(1) avg	Fast existence check
map<K,V>	By key	Keys only	By key	O(log n)	Key-value dictionary
unordered_map<K,V>	None	Keys only	By key	O(1) avg	Fast key-value lookup

Common Patterns & Mistakes to Avoid

✗Common Mistakes	✓Correct Approach
Dereferencing v.end()	Always check it != container.end() before dereferencing
ms.erase(val) to remove one copy	Use ms.erase(ms.find(val)) to remove just one copy
m[key] to check if key exists (creates it!)	Use m.count(key) or m.find(key) != m.end()
Forgetting * when using max_element	int max = *max_element(v.begin(), v.end())
Missing return in comparator function	Always use 'return true;' not just 'true;'
Using st[i] to access stack element	Stacks only support top() — no index access
Starting next_permutation on unsorted string	Sort the string FIRST before the do-while loop

Your C++ Learning Roadmap

Now that you have this foundation, here is the recommended path to becoming a confident C++ developer and DSA problem solver:

Phase 1	Master this guide. Be able to write each container from memory and explain what it does.
Phase 2	Practice array and string problems. Use vectors confidently as your primary container.
Phase 3	Learn recursion and basic searching/sorting (binary search, merge sort, quick sort).
Phase 4	Tackle Two Pointers, Sliding Window problems — these use vectors and iterators heavily.
Phase 5	Hash table problems — this is where unordered_map and unordered_set shine.
Phase 6	Tree and Graph problems — learn BFS (queue) and DFS (stack/recursion).
Phase 7	Heap problems (priority_queue), advanced sorting, and greedy algorithms.
Phase 8	Dynamic Programming — the final boss. All foundations click here.

Final Advice

Consistency beats intensity. 30 minutes every day beats 5 hours once a week.
Read code. Read other people's solutions after you solve a problem.
The STL is your toolkit — know it so well that reaching for the right tool is instinctive.
C++ rewards those who understand what's happening under the hood. This guide gave you that foundation.