C++ STL

Complete Beginner's Reference Guide

Pairs • Vectors • Iterators • Stack • Queue • Priority Queue • Set • Map

0. What Is the STL? (Start Here!)

STL stands for **Standard Template Library**. Think of it as a **toolbox** that C++ gives you for free. Instead of building your own data structures from scratch, you just **use these ready-made containers**. They are well-tested, fast, and save you enormous time — especially in DSA.

Every program in this guide starts with:

```
#include <bits/stdc++.h>   // includes EVERYTHING from STL
using namespace std;      // so you don't write std:: before everything
```

TIP: Think of each STL container as a real-world object. A Stack = stack of plates. A Queue = line at a ticket counter. Visualizing them this way makes them stick.

1. Pair

What is it?

A **pair** simply holds **two values together** as one unit. Like coordinates (x, y), or a name with a score.

Declare and Access

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

Nested Pair (pair inside a pair)

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

cout << p2.first;          // 1
cout << p2.second.first;   // 3
cout << p2.second.second;  // 5
```

Array of Pairs

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

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

TIP: Pairs are used constantly in DSA — especially in sorting and maps. Master .first and .second!

2. Vector

What is it?

A vector is a **dynamic array** — it **grows and shrinks** in size automatically. Unlike regular C++ arrays (fixed size), vectors adjust when you add or remove elements.

Declaring a Vector

```
vector<int> v;            // empty vector
vector<int> v1(5, 100);  // {100, 100, 100, 100, 100}  <- 5 elements, each = 100
vector<int> v2(5);       // {0, 0, 0, 0, 0}            <- 5 elements, default = 0
vector<int> v3(v1);      // exact copy of v1
```

Adding Elements

```
vector<int> v;
v.push_back(1);    // adds 1 at END  -> {1}
v.push_back(2);    // adds 2 at END  -> {1, 2}
v.emplace_back(3); // same as push_back, slightly faster -> {1, 2, 3}
```

□ **NOTE:** push_back and emplace_back both add to the end. For simple types like int, they behave identically. Prefer emplace_back for performance.

Accessing Elements

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

cout << v[0];      // 10  (index 0 = first element)
cout << v[2];      // 30
cout << v.at(1);   // 20  (safe version — throws error if index out of range)
cout << v.front(); // 10  (first element)
cout << v.back();  // 50  (last element)
```

Useful Vector Functions

Function	What it does	Example
`v.size()`	Number of elements	`v.size() -> 5`
`v.empty()`	1 if empty, 0 if not	`v.empty() -> 0`
`v.pop_back()`	Remove last element	`{1,2,3} -> {1,2}`
`v.clear()`	Remove ALL elements	`{1,2,3} -> {}`
`v.swap(v2)`	Swap contents with v2	`v and v2 exchange`

Vector of Pairs — Very common in DSA!

```
vector<pair<int,int>> vec;

vec.push_back({1, 2});   // adds pair (1,2)
vec.emplace_back(3, 4);  // adds pair (3,4)  <- no braces needed

cout << vec[0].first;    // 1
cout << vec[1].second;   // 4
```

3. Iterators

What is an Iterator?

An iterator is like a **pointer or cursor** that points to an element inside a container. It lets you traverse (walk through) elements one by one. Think of it as a bookmark that can move through a vector.

begin() and end()

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

//  v.begin()  points TO the 1st element (10)
//  v.end()    points PAST the last element (after 50)

auto it = v.begin();
cout << *it;  // 10   (* means: get the VALUE at this position)

it++;         // move to next
cout << *it;  // 20

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

□ **REMEMBER:** v.end() does NOT point to the last element — it points ONE PAST the last. Never dereference end() directly or you'll get garbage/crash.

Three Ways to Loop Through a Vector

Method 1: Classic iterator (verbose, educational to understand)

```
for(vector<int>::iterator it = v.begin(); it != v.end(); it++) {
    cout << *it << " ";
}
```

Method 2: Using 'auto' keyword (shorter)

```
for(auto it = v.begin(); it != v.end(); it++) {
    cout << *it << " ";
}
```

Method 3: Range-based for loop (cleanest — use this most often!)

```
for(auto x : v) {
    cout << x << " ";    // x is a copy of each element
}
```

□ **TIP:** For 90% of DSA problems, use the range-based for loop (Method 3). It's the shortest and clearest.

Deleting with erase()

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

v.erase(v.begin() + 1);           // removes index 1 (20) -> {10, 30, 40, 50, 60}
v.erase(v.begin() + 1, v.end() - 2); // removes [index1, index3) -> removes 30, 40
// v is now {10, 50, 60}
```

□ **NOTE:** erase(start, end) removes from 'start' UP TO BUT NOT INCLUDING 'end'. This is called a half-open range: [start, end).

Inserting with insert()

```
vector<int> v = {100, 100};                // {100, 100}
v.insert(v.begin(), 300);                  // {300, 100, 100}
v.insert(v.begin() + 1, 2, 10);           // {300, 10, 10, 100, 100}

vector<int> other = {50, 50};
v.insert(v.begin(), other.begin(), other.end()); // {50, 50, 300, 10, 10, 100, 100}
```

4. Stack

What is it?

A stack is **Last In, First Out (LIFO)**. Like a stack of plates — the last plate you put on is the first one you take off. You can ONLY access the **top** element.

Stack Operations

```
stack<int> st;

st.push(1);  // {1}
st.push(2);  // {1, 2}
st.push(3);  // {1, 2, 3}
st.push(4);  // {1, 2, 3, 4}

cout << st.top();   // 4  (most recently added)
st.pop();           // removes 4 -> {1, 2, 3}
cout << st.top();   // 3
cout << st.size();  // 3
cout << st.empty(); // 0 (false — not empty)
```

REMEMBER: You CANNOT use st[i] on a stack. The ONLY way to access an element is st.top(). Pop to get the next one.

Operation	What it does
`st.push(x)`	Add x to the top
`st.emplace(x)`	Same as push, slightly faster
`st.top()`	READ the top element (does NOT remove)
`st.pop()`	REMOVE the top element (does NOT return it)
`st.size()`	Number of elements
`st.empty()`	1 if empty, 0 otherwise
`st1.swap(st2)`	Swap contents of two stacks

5. Queue

What is it?

A queue is **First In, First Out (FIFO)**. Like a line at a movie theatre — first person in line is served first. Elements enter at the **back** and leave from the **front**.

Queue Operations

```
queue<int> q;

q.push(1);  // {1}
q.push(2);  // {1, 2}
q.push(3);  // {1, 2, 3}

cout << q.front();  // 1  (oldest element, first to enter)
cout << q.back();   // 3  (newest element, last to enter)

q.back() += 5;      // modify back: {1, 2, 8}
cout << q.back();   // 8

q.pop();            // removes front -> {2, 8}
cout << q.front();  // 2
```

Operation	What it does
`q.push(x)`	Add x to the BACK
`q.emplace(x)`	Same as push, faster
`q.front()`	READ the front element (oldest)
`q.back()`	READ the back element (newest)

q.pop()	REMOVE the front element
q.size()	Number of elements
q.empty()	1 if empty

6. Priority Queue

What is it?

A priority queue always gives you the **most important element first**. By default, the **largest element is always at the top (Max-Heap)**. You can also make it give the smallest element first (Min-Heap).

Max-Heap (Default) — Largest element at top

```
priority_queue<int> pq;

pq.push(5);  // top = 5
pq.push(2);  // top = 5  (5 > 2, so 5 stays on top)
pq.push(8);  // top = 8  (8 is now the largest)
pq.push(1);  // top = 8

cout << pq.top();  // 8  (always the maximum!)
pq.pop();
cout << pq.top();  // 5  (next largest)
```

Min-Heap — Smallest element at top

```
priority_queue<int, vector<int>, greater<int>> minpq;
//                              ^^^^^^^^^^^^^
//                    This changes it to a Min-Heap

minpq.push(5);  // top = 5
minpq.push(2);  // top = 2  (2 < 5, comes to top)
minpq.push(8);  // top = 2
minpq.push(1);  // top = 1  (1 is the new minimum)

cout << minpq.top();  // 1  (always the minimum!)
minpq.pop();
cout << minpq.top();  // 2
```

□ **TIP:** Max-Heap: repeatedly need the LARGEST element. Min-Heap: repeatedly need the SMALLEST. Both run in O(log n) per push/pop — much faster than sorting every time.

7. Set

What is it?

A set stores **unique elements in automatically sorted order**. Duplicates are silently ignored. You cannot access elements by index — use find() or iterate.

Basic Operations

```
set<int> s;

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

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

cout << s.size();   // 3
cout << s.count(2); // 1  (means 2 IS in the set)
cout << s.count(9); // 0  (means 9 is NOT in the set)
```

Finding Elements

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

auto it = s.find(2);          // iterator pointing to 2
cout << *it;                  // 2

auto it2 = s.find(9);         // 9 not found
if(it2 == s.end()) {
    cout << "9 is not in set";
}
```

lower_bound and upper_bound

lower_bound(x): iterator to **first element >= x**

upper_bound(x): iterator to **first element > x**

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

auto lb = s.lower_bound(6);  // points to 6 (first element >= 6)
auto ub = s.upper_bound(6);  // points to 8 (first element > 6)

cout << *lb;  // 6
cout << *ub;  // 8

// lower_bound(5) -> points to 6 (first element >= 5)
// upper_bound(5) -> points to 6 (first element > 5)
```

Erasing a Range

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

auto it1 = s.find(2);
auto it2 = s.find(5);
s.erase(it1, it2);  // deletes [it1, it2) -> removes 2 and 4, NOT 5

// s is now {1, 5}
for(auto x : s) cout << x << " ";  // prints: 1 5
```

NOTE: Set always stays sorted. You never sort it manually. find() and insert() are O(log n) — very fast.

8. Map

What is it?

A map stores **key-value pairs** — like a dictionary. Each **key is unique** and maps to exactly one value. Keys are **sorted automatically**. Example real-world uses: student name → marks, word → frequency, city → population.

Basic Map Operations

```
map<int, int> m;

m[1] = 100;           // key=1, value=100
m[2] = 200;           // key=2, value=200
m.emplace(3, 300);    // key=3, value=300
m.insert({4, 400});   // key=4, value=400

cout << m[2];  // 200
cout << m[3];  // 300

for(auto it : m) {
    cout << it.first << " -> " << it.second << endl;
}
// Output (always sorted by key):
// 1 -> 100
// 2 -> 200
// 3 -> 300
// 4 -> 400
```

REMEMBER: If you access m[key] for a key that does NOT exist, C++ creates that key with value 0. Use m.find(key) == m.end() to safely check existence.

Finding Elements in a Map

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

auto it = m.find(3);

cout << it->first;   // 3   (the key)
cout << it->second;  // 300 (the value)
```

```
// Alternative syntax:
cout << (*it).first;   // 3
cout << (*it).second;  // 300

if(m.find(99) == m.end()) {
    cout << "Key 99 not found!";
}
```

Map with Pair as Key

```
map<pair<int,int>, int> m2;

m2[{1, 2}] = 100;  // key=(1,2), value=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

for(auto i : m3) {
    cout << i.first << " " << i.second.first << " " << i.second.second << endl;
}
// Output:
// 1 100 200
// 2 300 400
// 3 500 600
```

9. Sort & Useful Extras

sort() function

```
int a[] = {8, 9, 7, 3, 5, 2, 6, 1};

sort(a, a + 8);                    // sorts all: {1,2,3,5,6,7,8,9}
sort(a + 2, a + 6);                // sorts only index 2 to 5
sort(a, a + 8, greater<int>());    // descending: {9,8,7,6,5,3,2,1}

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

Custom Comparator

```
// Sort pairs: by second element (ascending), tie-break by first (descending)
bool myComp(pair<int,int> p1, pair<int,int> p2) {
    if(p1.second != p2.second)
        return p1.second < p2.second;
    return p1.first > p2.first;
}

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

max_element and min_element

```
int a[] = {8, 9, 7, 3, 5};
int maxVal = *max_element(a, a + 5);  // 9
int minVal = *min_element(a, a + 5);  // 3

vector<int> v = {4, 1, 7, 2};
int mx = *max_element(v.begin(), v.end());  // 7
```

Counting Set Bits (__builtin_popcount)

A 'set bit' is a bit with value 1 in the binary representation of a number:

```
int num = 7;   // binary: 0111  -> 3 set bits
cout << __builtin_popcount(num);    // 3

num = 6;       // binary: 0110  -> 2 set bits
cout << __builtin_popcount(num);    // 2

long long big = 111123456789LL;
cout << __builtin_popcountll(big);  // use popcountll for long long
```

next_permutation — All Permutations

```
string str = "123";  // must be sorted first for ALL permutations!
sort(str.begin(), str.end());

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

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

10. Quick Reference Cheat Sheet

Which container to use?

Situation	Use This
Need a resizable array	vector<T>
Need LIFO (last in, first out)	stack<T>
Need FIFO (first in, first out)	queue<T>
Need largest/smallest element fast	priority_queue<T>
Need unique sorted elements	set<T>
Need key-value pairs, unique+sorted keys	map<K,V>
Need to hold two values together	pair<T1,T2>

Time Complexity Summary

Container	Insert	Delete	Search	Access
vector	O(1) back	O(1) back	O(n)	O(1) by index
stack	O(1) top	O(1) top	N/A	O(1) top only
queue	O(1) back	O(1) front	N/A	O(1) front/back
priority_queue	O(log n)	O(log n)	N/A	O(1) top only
set	O(log n)	O(log n)	O(log n)	N/A (no index)
map	O(log n)	O(log n)	O(log n)	O(log n) by key

NOTE: O(1) = instant (fastest). O(log n) = very fast. O(n) = slows with more elements. n = number of elements stored.

11. Golden Rules to Never Forget

TIP: 1. Pairs use .first and .second — always, no exceptions.

TIP: 2. Vector index starts at 0. v[0] is the first element.

TIP: 3. v.end() points PAST the last element. Never dereference it directly.

TIP: 4. Stack: only access TOP. Queue: access FRONT and BACK.

TIP: 5. Set auto-sorts and rejects duplicates. Map auto-sorts by key.

TIP: 6. Use m.find(key) == m.end() to safely check if a key exists in a map.

TIP: 7. priority_queue is Max-Heap by default. Add 'greater<int>' for Min-Heap.

TIP: 8. erase(start, end) removes [start, end) — the 'end' is NOT deleted.

TIP: 9. Always start every C++ file with: #include<bits/stdc++.h> and using namespace std;

You're ready for DSA. Keep this guide by your side!