Ultimate C++ Interview Cheatsheet

A concise and comprehensive reference for preparing for C++ technical interviews, covering syntax, STL containers, object-oriented features, and common utilities.

◇ Headers and Namespaces

Include Essentials

These headers provide access to core C++ Standard Library functionality:

```
#include <iostream>      // Input/output: cin, cout
#include <vector>        // Dynamic arrays
#include <string>        // String manipulation
#include <algorithm>     // Sorting, reversing, max/min, etc.
#include <unordered_map> // Hash table (key-value pairs)
#include <unordered_set> // Hash table (unique keys)
#include <stack>         // LIFO stack
#include <queue>         // FIFO queue and priority queue
#include <deque>         // Double-ended queue
#include <cmath>         // Math operations
#include <sstream>       // String streams (split/join)
#include <iomanip>       // Output formatting
#include <climits>       // Integer limits: INT_MAX, etc.
#include <cstdlib>       // Random numbers: rand, srand
#include <ctime>         // Time for random seed
```

Namespace Declaration

```
using namespace std; // Removes need to prefix std::
```

Convenient for interview coding. Avoid in large-scale codebases.

◇ Memory Model: Stack vs Heap

Stack

- Memory automatically managed (freed when scope ends)
- Fast, limited size
- Used for local variables:

```
TreeNode node; // on stack
```

Heap

- Memory manually managed (via `new` / `delete`)
- Slower, larger capacity
- Needed for dynamic allocation or variable lifetime beyond scope:

```
TreeNode* node = new TreeNode(); // on heap
delete node; // must free manually
```

◇ Values, Pointers, and References

Value

```
TreeNode node;
```

- Stores the actual data.
- Passed by copy unless specified.
- Use `.` to access members: `node.val`

Pointer (*)

```
TreeNode* ptr = &node;
```

- Stores the address of a variable.
- Can be `nullptr`.
- Use `->` or `*` to access or modify data:

```
ptr->val = 5;       // same as (*ptr).val = 5
*ptr = TreeNode();  // replace what ptr points to
```

Reference (&)

```
TreeNode& ref = node;
```

- Alias for an existing variable.
- Cannot be null.
- Acts like the original variable: `ref.val` modifies `node.val`

> 💡 Think:

> - `*` = follows an address
> - `&x` = gets an address
> - `int&` = new name for the same thing

◇ Dot (`.`) vs Arrow (`->`)

Use `.` when you have an **object**

```
TreeNode node;
node.val = 5;
```

Use `->` when you have a **pointer to an object**

```
TreeNode* ptr = new TreeNode();
ptr->val = 5; // same as (*ptr).val = 5
```

◇ Dereferencing (*)

`*` is used to access the value a pointer points to:

```
int x = 10;
int* p = &x;
*p = 20; // changes x to 20
```

For objects:

```
TreeNode* node = new TreeNode();
(*node).val = 5; // same as node->val = 5
```

> 💡 Rule:
>
> - Use `.` for objects
> - Use `->` or `*` for pointers

◇ Input/Output Basics

Standard I/O:

```
int x;
cin >> x;                              // Input
cout << "Value: " << x << endl;        // Output with newline
```

Note: `endl` flushes the output buffer. For performance, prefer `"\n"` unless flushing is necessary.

◇ Data Types & Constants

Common Integer Limits:

```
#include <climits>
INT_MAX, INT_MIN      // Max/min for int
LONG_MAX, LONG_MIN    // Max/min for long
UINT_MAX, ULONG_MAX   // Max values for unsigned types
```

Useful for initializing extreme values in algorithms.

◇ Working with Strings

Operations:

```
string s = "hello";
s[0];                              // Access character
s.size();                          // Get length
s.substr(start, length);           // Substring
s.append("world");                // Append text
s.find("lo") != string::npos;     // Check substring
s.erase(pos, len);                // Remove portion
s.insert(pos, "abc");            // Insert text
s.replace(pos, len, "xyz");      // Replace text
reverse(s.begin(), s.end());      // Reverse in place
```

Conversions:

```
to_string(42);              // int to string
stoi("123");               // string to int
string(1, 'a');             // char to string
```

◇ String Stream Utilities

Build and Convert:

```
stringstream ss;
ss << "year" << 2025;
string result = ss.str();
```

Tokenizing Input:

```
string input = "a,b,c";
istringstream ss(input);
string token;
while (getline(ss, token, ',')) {
    cout << token << endl;
}
```

Handy for simulating `split()` behavior.

◇ Vectors (Dynamic Arrays)

Common Usage:

```
vector<int> v = {1, 2, 3};
v.push_back(4);
v.pop_back();
v.size();
v.empty();
v[0];
sort(v.begin(), v.end());
```

2D Vector:

```
vector<vector<int>> grid(N, vector<int>(M, 0));
```

◇ Maps & Sets

Unordered Map (Hash Table)

```
unordered_map<int, string> mp;
mp[1] = "one";
if (mp.find(1) != mp.end()) {
    cout << mp[1];
}
mp.erase(1);
```

Unordered Set

```
unordered_set<int> st;
st.insert(10);
st.erase(10);
if (st.find(10) != st.end()) {
    cout << "found";
}
```

Great for quick existence checks and uniqueness.

◇ Stack, Queue, and Priority Queue

Stack (LIFO):

```
stack<int> s;
s.push(1);
s.top();
s.pop();
```

Queue (FIFO):

```
queue<int> q;
q.push(1);
q.front();
q.pop();
```

Priority Queue:

```
priority_queue<int> maxHeap; // Default max-heap
priority_queue<int, vector<int>, greater<int>> minHeap;
```

Custom Comparator:

```
struct Node {
    int val;
    Node(int v) : val(v) {}
};
struct cmp {
    bool operator()(Node* a, Node* b) {
```

```
        return a->val > b->val; // Min-heap
    }
};
priority_queue<Node*, vector<Node*>, cmp> pq;
```

◇ Pairs

Create and Access:

```
pair<int, int> p = make_pair(2, 3);
cout << p.first << ", " << p.second;
```

Useful for coordinates, key-value pairs, and sorting.

◇ Math Functions

From <cmath>:

```
sqrt(x);
pow(x, y);
round(x);
abs(x);
M_PI; // 3.1415...
```

Covers most computational geometry or number theory needs.

◇ Random Numbers

Basic Random Usage:

```
#include <cstdlib>
srand(time(0)); // Seed
int randInt = rand() % 100;  // [0, 99]
double randFloat = rand() / double(RAND_MAX); // [0.0, 1.0)
```

Good for randomized testing or probabilistic algorithms.

◇ Common Data Structures (LeetCode Style)

Linked List Node:

```
class ListNode {
public:
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL) {}
};
```

Binary Tree Node:

```
class TreeNode {
public:
    int val;
    TreeNode *left, *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

◇ Miscellaneous STL Tools

Deque:

```
deque<int> dq;
dq.push_front(1);
dq.push_back(2);
dq.pop_front();
dq.pop_back();
```

Used in sliding window and monotonic queue problems.

Rotate Vector:

```
rotate(v.begin(), v.begin() + k, v.end());
```

Moves the first `k` elements to the end.

Swap Containers:

```
cur.swap(next);
```

Efficiently exchanges contents of any STL container.

◇ Iterators

Loop with Iterator:

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

Essential for generic STL usage and custom container traversal.

◇ Operator Precedence (Common Pitfalls)

From Highest to Lowest:

```
() > [] > * > >> > &
```

Parentheses > array access > pointer dereference > bitwise shift > bitwise AND.

☑ Final Notes

- Use `std::vector`, `std::unordered_map`, and `std::string` heavily — they're efficient and versatile.
- Always handle edge cases: empty input, duplicates, zero-length vectors.
- Use iterators and custom comparators to solve problems cleanly.
- Practice writing class-based solutions — many interview problems involve implementing custom types.