📌 C++ Vector — Level 40 NOTES

1. **What vector really is:** vector = dynamic array. Stores elements in contiguous memory. Can grow and shrink automatically.

2. **size() vs capacity()** (MOST IMPORTANT):
   - **size()**: Number of elements currently present.
     Valid indices: 0 to size()-1. Changes when you add/remove elements.
   - **capacity()**: Number of elements vector can hold without reallocating.
     Always: size() ≤ capacity(). Does NOT tell how many elements exist.

3. **Why capacity exists:** If vector increased memory one by one, every push_back() would be slow.
   So **vector:** Allocates extra memory. Grows in chunks (usually doubling).
   Copies old elements during reallocation.
   ☞ **Result:** push_back() is O(1) amortized, not always O(1)

4. **Growth behavior** (DO NOT MEMORIZE NUMBERS):
   Typical pattern: 1 → 2 → 4 → 8 → 16 → ⋯ ⚠ Exact values are implementation-dependent. Never write logic based on exact capacity.

5. **Correct looping rule** (DSA RULE):
   `for(int i = 0; i < v.size(); i++)` ✘ Never loop till capacity().
   Reason: Elements beyond size() are invalid. Access = undefined behavior.

6. **Construction behavior:**
   `vector<int> v;` // size = 0, capacity = 0
   `vector<int> v(5);` // size = 5, capacity ≥ 5.
   `vector<int> v = {1,2,3}` // size = 3, capacity = 3.
   Initializer list → exact capacity, no extra memory.

7. **reserve()** — misunderstood by beginners:
   - Purpose: Pre-allocate memory without changing size.
     `v.reserve(50);` size → unchanged, capacity → ≥ 50.

✘ reserve() does NOT create elements. ✘ You cannot access v[0] after only reserve()

8 **clear()**: v.clear();
Removes all elements. size → 0. capacity → unchanged. 👉 Memory is still held.

9 **shrink_to_fit()**: v.shrink_to_fit();
Requests vector to free unused memory.
capacity → may reduce to size(). Not guaranteed (depends on implementation).
Use only when memory really matters.

10 **Time Complexity Truth Table**:

Operation	Time
push_back()	O(1) amortized
push_back() (reallocation)	O(n)
pop_back()	O(1)
Access v[i]	O(1)
clear()	O(n)
reserve(n)	O(n).

4 🧠 **Common Vector Functions** (You MUST know):
- **Size & capacity**: v.size(), v.capacity(), v.empty(), v.reserve(n), v.shrink_to_fit().
- **Modifiers**: v.push_back(x), v.pop_back(), v.clear(), v.insert(pos, value), v.erase(pos).
- **Access**: v[i], v.at(i) (safer), v.front(), v.back().
- **Iterators**: v.begin(), v.end().

🔁 **Reallocation — Proper Definition** (C++ vector):
- **What is reallocation?** Process where a vector: Runs out of capacity, Allocates new, larger block, Copies/moves elements, Frees old memory, Updates pointers. Happens automatically during growth (e.g., push_back()).
- **When does it happen?** size() == capacity() AND you try to insert.
- **Why expensive?** Every element copied/moved. Time cost = O(n).
- **What happens to memory?** Before: [1][2][3] ← capacity full.
  After: [1][2][3][ ][ ][ ] ← new larger block.
  Old memory → destroyed. New memory → active.

🔥 One more mental grenade (don't dodge)

Answer YES or NO only:

```
vector<int> v;
v.reserve(10);
for(int i=0;i<11;i++)
  v.push_back(i);
```

⇒ Does reallocation happen?

What you got RIGHT

✓ Loop runs 11 times → correct
For elements 0 → 9 size ≤ capacity

✗ no reallocation → correct
On inserting the 11th element (index 10)
size == capacity ⇒ reallocation happens
→ correct

Final size = 11 → correct

So the trigger logic in your head is now correct. Good.

What you got WRONG (important)

✗ "capacity becomes 20"

🚫 NOT guaranteed. This is where people mess up interviews.

Truth: Capacity becomes some value ≥ 11

Often doubles (10 → 20), but:

- Could be 15
- Could be 16
- Could be something else

⇒ Never rely on exact growth numbers