
Category 1 — Smart Pointers (10 problems)

1. Implement a **custom reference-counted shared pointer** with weak pointer support.
2. Build a **cyclic data structure** using std::shared_ptr and std::weak_ptr to avoid memory leaks.
3. Implement a **thread-safe shared pointer** that supports atomic reference counting.
4. Write a **custom deleter** for std::unique_ptr that logs destruction.
5. Implement a **polymorphic object manager** using std::unique_ptr and base class pointers.
6. Build a **pool allocator** integrated with std::unique_ptr for fixed-size objects.
7. Implement **lazy initialization** of an object using std::shared_ptr and std::call_once.
8. Build a **graph structure** where nodes are managed with std::shared_ptr and edges with std::weak_ptr.
9. Implement a **resource manager** that safely transfers ownership using std::unique_ptr.
10. Use std::shared_ptr to implement a **copy-on-write string class**.

Category 2 — Manual Memory Management & RAII (10 problems)

11. Implement a **RAII wrapper** for a dynamically allocated array.
12. Create a **custom memory pool** with allocation and deallocation functions.
13. Implement a **smart file handle** using RAII (fopen/fclose).
14. Build a **custom allocator** for std::vector to reduce allocations for large numbers of small objects.
15. Implement **placement new** usage for a buffer of objects and safely destroy them.
16. Implement a **stack-allocated memory arena** with manual lifetime management.
17. Build a **resource pool** where objects are reused instead of deleted.
18. Implement a **RAII-based mutex wrapper** for automatic locking/unlocking.
19. Build a **temporary buffer manager** using RAII to manage memory slices.
20. Implement a **dynamic array** from scratch with manual new[]/delete[] management.

Category 3 — Memory Leaks, Dangling Pointers & Debugging (10 problems)

21. Detect **dangling pointers** in a class with multiple owners.
22. Implement a **memory leak detector** using custom new/delete operators.
23. Build a **smart pointer simulator** and verify no memory leaks on circular references.
24. Debug and fix a **double deletion bug** in a class hierarchy.

25. Write a **container wrapper** that detects invalidated iterators and reports errors.
26. Implement a **tracked allocator** that counts allocations and deallocations.
27. Build a **debug mode unique pointer** that asserts if copied.
28. Implement a **manual reference counting object** and simulate over-release.
29. Write a **memory sanitizer tool** that records allocation call sites.
30. Detect **use-after-free** in a linked list implementation.

Category 4 — Memory Pooling & Low-Level Allocators (10 problems)

31. Implement a **fixed-size object memory pool** with free list management.
32. Build a **slab allocator** for different object sizes.
33. Implement a **chunked allocator** for vector elements to reduce fragmentation.
34. Build a **stack allocator** that allocates memory in a linear fashion and rolls back.
35. Implement a **buddy allocator** for dynamic memory management.
36. Create a **cache-aligned memory allocator** for high-performance data structures.
37. Implement a **thread-local memory pool** to reduce locking overhead.
38. Build a **block allocator** for small objects that reuses freed blocks efficiently.
39. Implement a **memory arena** shared across multiple containers.
40. Build a **hybrid allocator** combining pool and heap for variable object sizes.

Category 5 — Advanced & Real-World Scenarios (10 problems)

41. Implement **copy-on-write** for a large buffer using smart pointers.
42. Build a **resizable buffer** with exponential growth and manual memory management.
43. Implement **deferred deletion** for objects accessed across threads.
44. Build a **graph structure** with shared ownership and cyclic references handled safely.
45. Implement **object versioning** with memory snapshots and rollback support.
46. Build a **memory-efficient sparse matrix** with RAII and dynamic allocation.
47. Implement **memory leak detection** in a multithreaded allocator scenario.
48. Build a **pool of pre-allocated network buffers** for low-latency applications.
49. Implement a **hierarchical memory manager** with parent/child ownership.
50. Design a **high-performance cache** using custom allocator and smart pointers for objects.
