memory allocator implementation

Advantages:

- By using mmap to obtain memory from the OS in larger chunks, the allocator reduces overhead compared to frequent malloc calls, which can be expensive.

- The allocator manages memory internally using a freelist, which helps in efficiently tracking and reusing memory blocks of fixed sizes.

- Parameters like pool_element_size, num_elements, and max_num_pools are configurable at runtime, providing flexibility to adapt to different application requirements.

- No Additional Allocation Overhead: The freelist pointers (Next and Prev) are embedded within each pool element, eliminating the need for additional memory allocation to manage the freelist.

Disadvantages:

- The allocator may suffer from fragmentation if memory blocks of different sizes are requested frequently.

- Due to the fixed-size nature of the pools and elements, it may not be optimal for scenarios requiring variable-sized allocations or highly dynamic memory management.

- Error handling for mmap failures can be complex to manage when dealing with multiple pools and potential resource limits.

Alternate Implementation Suggestions:

- Should implement logic to dynamically adjust the size of pools based on demand or usage patterns (can help in optimizing memory usage and reducing fragmentation).

- Extend the allocator to support variable-sized memory blocks within a pool, accommodating applications with diverse memory allocation requirements.

- Instead of a simple linked list, can try using more advanced data structures like segregated lists,, or hierarchical freelists to improve allocation and deallocation efficiency.

- Enhance error handling to manage edge cases, such as out-of-memory conditions, and provide informative error messages for debugging and troubleshooting.