- In main:
    - I created an array of thread ids and call init() function, then create each of the threads using the pthread_create() function. I passed the id of that thread as an argument here. Then they start their execution in the thread_function in which:
        - I generate the random memory size that will be requested
        - Cast the void* type to int* type. When we use the value of the id, we dereference it by *id.
        - Locked the mutex because we enter the critical region
        - Called my_malloc in which:
            - I create a node with the requested memory size and id of the thread and push that node into the queue
        - Blocked that thread with semaphore until the server_thread evaluates its request by popping nodes from the queue. In server_thread function:
            - There is a loop that iterates until all other threads are joined in the main. While there is a node in the queue, it pops that node to determine the source of the request by the id and also the size.
            - If there is enough memory, it updates the message array's id'th location with the start index of the allocation and updates the next start point.
            - Else, it updates the id'th location of the message array with -1 to indicate that there is not enough space.
            - Since after that process, the request is evaluated; the semaphore of that requester thread is unblocked by an up operation.
        - After the server_thread unblocked the thread that requests a memory allocation, the thread_function updates the respective memory slots with the id of that thread in order to indicate that these slots are allocated for the thread which has that particular id if there is enough memory, else it prints an error message to indicate there is not enough memory.
        - Then, it unlocks the mutex because it exits the critical region.
        - In order to guarantee the termination of the thread, besides joining them, I also use a return statement at the end of the thread_function.
- After all threads make their requests and these requests are evaluated by the server_thread, in the main:
    - The threads are joined using a for loop.
    - We print the content of the memory using the dump_memory() function.
    - We print the memory indices where the thread specific allocations start and ends.
    - Then we make the notjoined variable false, because we already joined the threads. This is the variable that we control the server_thread. It terminates its while loop and terminates the execution of the thread.
    - We print a terminating message and terminate the execution of the program.

Note: Because of the fact that the type of the memory array is a char array, for 10 threads I give id's from 0 to 9 to fit these values into a single char. This means that the memories allocated by thread 0 are indicated with 0 as well as the nonallocated spaces. We can differentiate the nonallocated spaces and the allocated spaces by thread 0 using the memory indices which we also print at the end of the program.