**CSL 316 - Language Processors - Assignment 1 - Semester 6th, 3rd Year**

Programmer: **Smit Gharat**

Roll no: BT21CSE108

Batch: R5

Professor: Dr. Manish Kurhekar

Date Due: 24 Jan 2024

Purpose: The purpose of this assignment is to implement a basic memory management system in C++, utilizing linked lists to manage allocated and free memory blocks. The program interprets transactions from an input file, including memory allocation, deallocation, and variable assignments, while also incorporating memory compaction to reduce fragmentation.

**Assumptions**

* **Total Memory Space:**
  + The system provides a fixed total memory space of 64 MB for memory allocations.
* **Compaction Functionality:**
  + The **compact\_memory** function is designed to reduce memory fragmentation. It attempts to provide larger contiguous memory chunks for allocation towards the end of the memory space.
* **Input Format:**
  + The input consists of statements in the following formats:
    1. **var = allocate size**: Allocates memory of the specified size and associates it with the variable name **var**.
    2. **free var**: Deallocates memory associated with the variable **var**.
    3. **var1 = var2**: Assigns the memory reference of **var2** to **var1**.
* **Output Format:**
  + The output is printed to the **output.txt** file. It includes the status of used and free memory blocks after processing all input transactions.

**INPUT FILE CONTENT**

Here for the example sake we are assuming the maximum memory to be 1024 bytes.

a = allocate 500

b = a

c = allocate 100

d = allocate 300

free c

e = allocate 220

free a

**OUTPUT FILE CONTENT**

Used Blocks:

Address: 800, Size: 220, Reference Count: 1

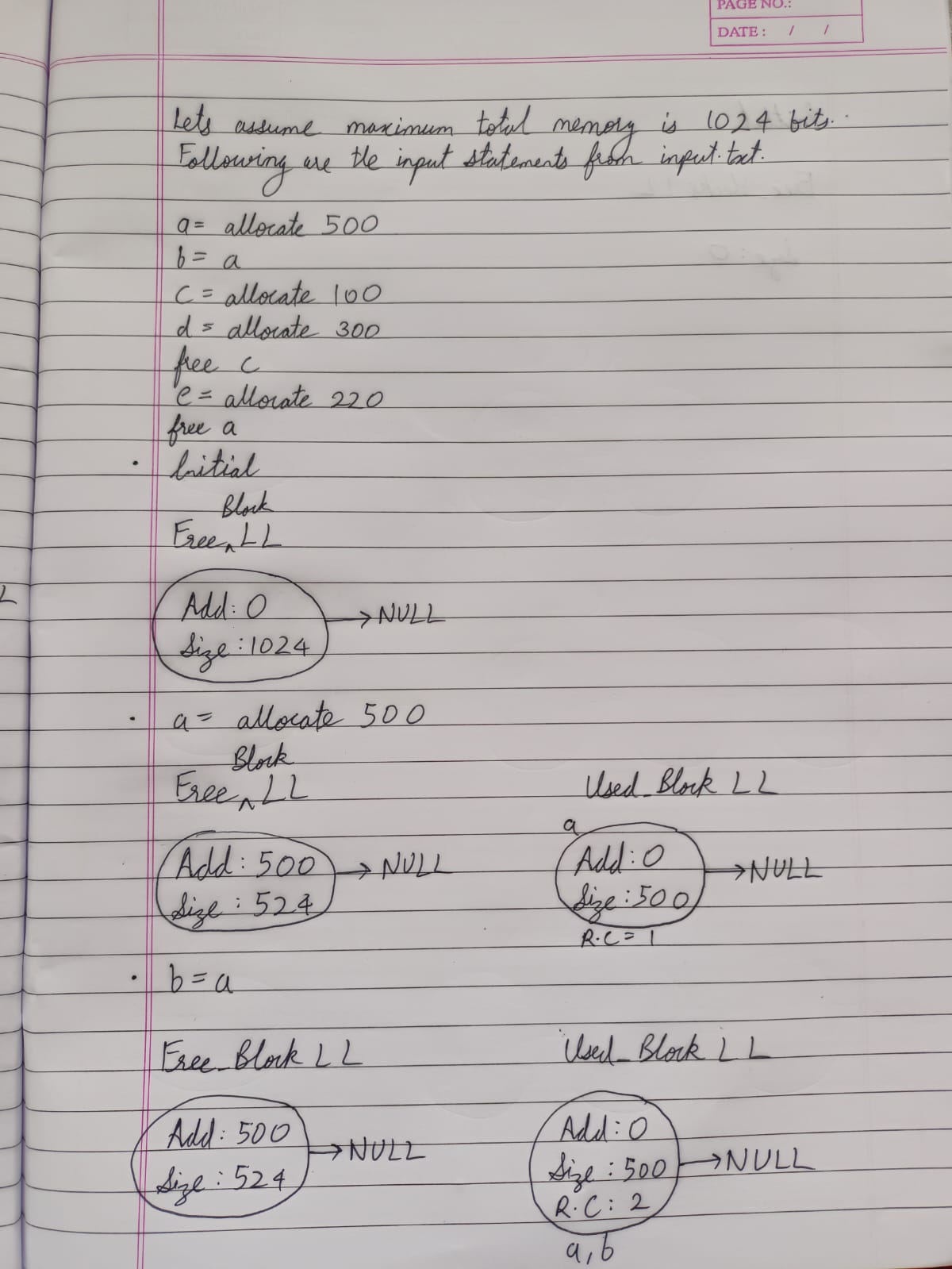
Address: 500, Size: 300, Reference Count: 1

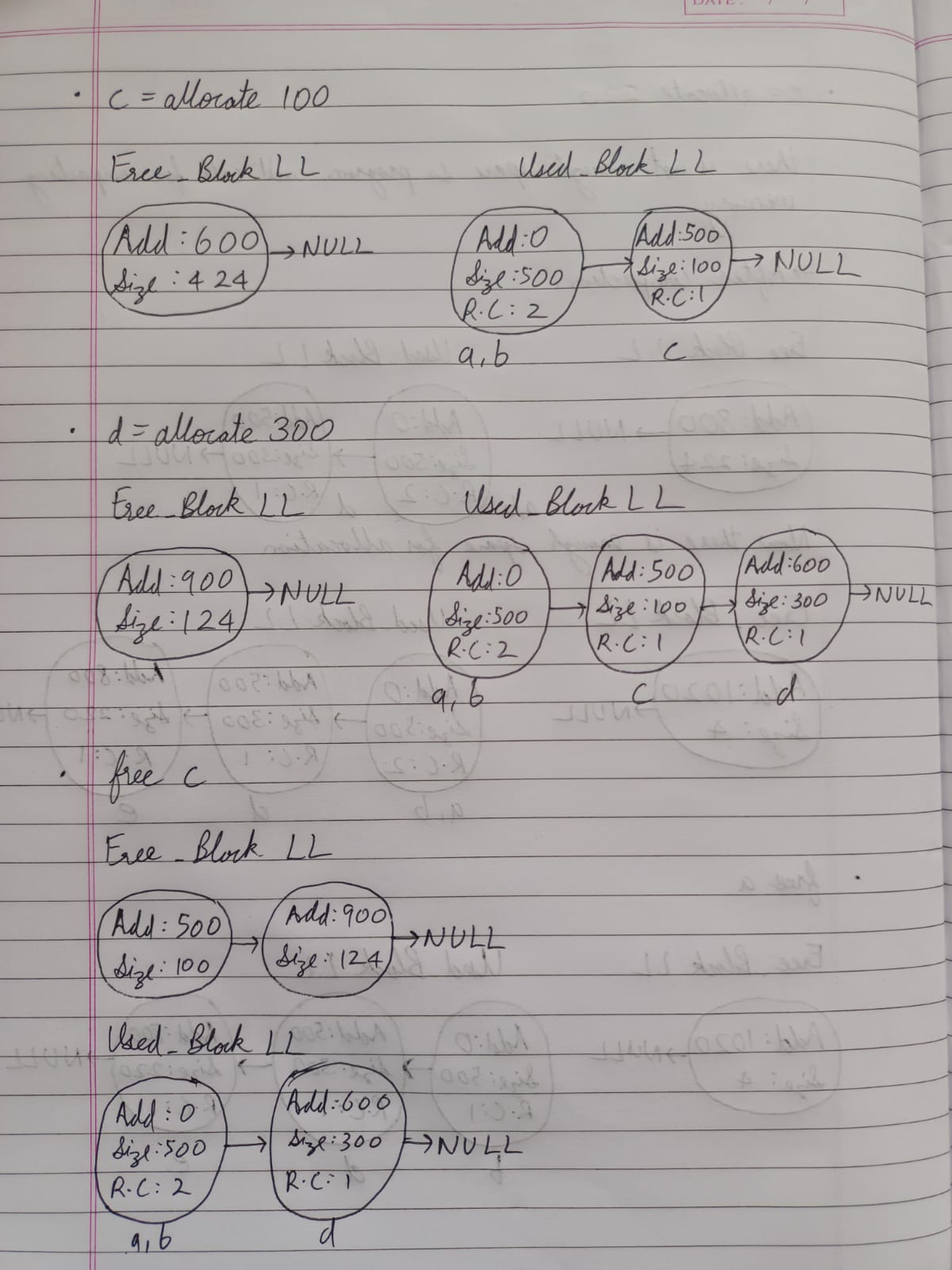
Address: 0, Size: 500, Reference Count: 1

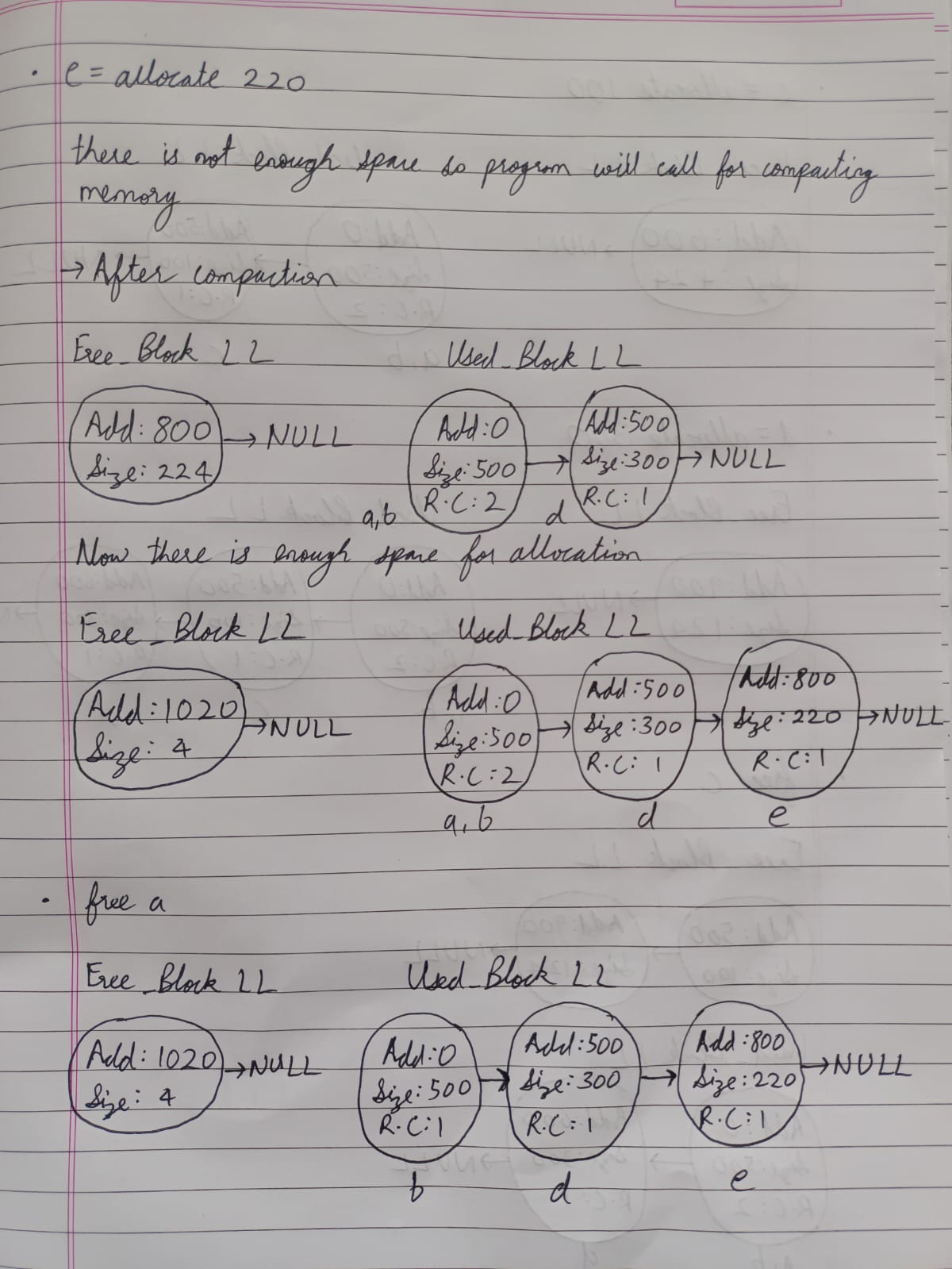
Free Blocks:

Address: 1020, Size: 4

**Example diagrams**

****





**Function documentation(also mentioned in code)**

Structure: MemoryBlock

Use: Defines a structure representing a memory block with details such as size, start address, reference count, and a next pointer.

Members:

    - size (int): Size of the memory block in bytes.

    - start\_address (int): Starting address of the memory block.

    - reference\_count (int): Number of references to this memory block.

    - next (MemoryBlock\*): Pointer to the next node in the linked list of memory blocks.

Class: MemoryManager

Use: Manages memory allocation and deallocation using a simple memory block structure.

Members:

    - memory\_chunk (int): Total size of the memory managed by the MemoryManager.

    - used\_blocks (MemoryBlock\*): Pointer to the head of the linked list of used memory blocks.

    - free\_blocks (MemoryBlock\*): Pointer to the head of the linked list of free memory blocks.

Public Member Functions:

    1. MemoryManager(int memory\_chunk)

        - Constructor for initializing the MemoryManager with a specified memory chunk size.

    2. int allocateBlock(int size)

        - Allocates a block of memory with the given size from the free memory blocks.

    3. int allocate(int size)

        - Allocates a block of memory with the given size, trying to compact memory if no sufficiently large block is found.

    4. void deallocate(int start\_address)

        - Deallocates the memory block at the specified start address.

    5. void compact\_memory()

        - Compacts the memory by moving used blocks closer together and merging adjacent free blocks.

    6. void print\_memory\_status()

        - Prints the current status of used and free memory blocks.

    7. ~MemoryManager()

        - Destructor to clean up allocated memory blocks when the MemoryManager object is destroyed.

Private Member Functions:

    - Nothing

Notes:

    - Memory allocation is performed by the `allocateBlock` function, and if no sufficiently large block is found, it tries to compact memory before retrying.

    - Memory deallocation is performed by the `deallocate` function, which adjusts reference counts and moves blocks between used and free lists.

    - Memory compaction is triggered by the `compact\_memory` function, which rearranges used and free blocks to reduce fragmentation.

    - The `print\_memory\_status` function outputs details of used and free memory blocks to the console.

Constructor: MemoryManager

Use: Initializes a MemoryManager object with the specified memory chunk size.

Arguments:

    - memory\_chunk (int): The total size of the memory managed by the MemoryManager.

Members:

    - memory\_chunk (int): Total size of the memory managed by the MemoryManager.

    - used\_blocks (MemoryBlock\*): Pointer to the head of the linked list of used memory blocks.

    - free\_blocks (MemoryBlock\*): Pointer to the head of the linked list of free memory blocks.

Initialization:

    - Allocates the initial free memory block covering the entire memory chunk.

    - Sets used\_blocks to nullptr as there are no allocated blocks initially.

Notes:

    - This constructor is called when creating a MemoryManager object to manage a specific amount of memory.

    - The allocated memory is initially considered as a single free block, covering the entire memory chunk.

Function: allocateBlock

Use: Allocates a block of memory with the given size from the free memory blocks.

Arguments:

    - size (int): Size of the memory block to be allocated.

Returns:

    - Start address of the allocated memory block, or -1 if allocation fails.

Functionality:

    - Searches the linked list of free memory blocks to find a block of sufficient size.

    - If a suitable block is found:

        - Allocates a new memory block in the used memory blocks list.

        - Adjusts the free block's start address and size accordingly.

        - Deletes the free block if its size becomes zero.

    - Returns the start address of the allocated block or -1 if allocation fails.

Notes:

    - This function is called by the `allocate` function when a new memory block needs to be allocated.

Function: allocate

Use: Allocates a block of memory with the given size, trying to compact memory if no sufficiently large block is found.

Arguments:

    - size (int): Size of the memory block to be allocated.

Returns:

    - Start address of the allocated memory block, or -1 if allocation fails.

Functionality:

    - Calls the `allocateBlock` function to attempt memory allocation.

    - If allocation fails, it tries to compact memory using the `compact\_memory` function and retries the allocation.

    - If allocation still fails, outputs an error message to the standard error stream.

    - Returns the start address of the allocated block or -1 if allocation fails.

Notes:

    - This function is the primary interface for allocating memory in the MemoryManager.

Function: deallocate

Use: Deallocates the memory block at the specified start address.

Arguments:

    - start\_address (int): Start address of the memory block to be deallocated.

Returns:

    - nothing

Functionality:

    - Searches the linked list of used memory blocks to find the block with the given start address.

    - Decreases the reference count of the found block.

    - If the reference count becomes zero:

        - Removes the block from the used memory blocks list.

        - Adds the block to the free memory blocks list.

    - Outputs an error message to the standard error stream if the block is not found.

Notes:

    - This function is responsible for deallocating memory, adjusting reference counts, and managing the used and free memory block lists.

Function: compact\_memory

Use: Compacts the memory by moving used blocks closer together and merging adjacent free blocks.

Functionality:

    - Iterates through the linked list of used memory blocks and adjusts their start addresses based on the available free space.

    - Updates the start address of corresponding free blocks to reflect the changes made to used blocks.

    - Checks for adjacent free blocks and merges them if possible.

Arguments:

    - Nothing

Returns:

    - Nothing

Notes:

    - This function is called when memory needs to be compacted, typically during the allocation process when no sufficiently large block is found.

    - It aims to reduce fragmentation by moving used blocks closer together and merging adjacent free blocks.

Function: print\_memory\_status

Use: Prints the current status of used and free memory blocks to the standard output.

Functionality:

    - Outputs information about used memory blocks, including their start address, size, and reference count.

    - Outputs information about free memory blocks, including their start address and size.

    - Utilizes cout for printing to the standard output.

Arguments:

    - Nothing

Returns:

    - Nothing

Notes:

    - This function is a diagnostic tool to observe the current state of used and free memory blocks.

    - It helps in monitoring memory usage and checking the effectiveness of memory allocation and deallocation processes.

Destructor: ~MemoryManager

Use: Cleans up allocated memory blocks when the MemoryManager object is destroyed.

Functionality:

    - Iterates through the linked list of used memory blocks, deletes each block, and moves to the next block.

    - Iterates through the linked list of free memory blocks, deletes each block, and moves to the next block.

    - Frees the memory used by both used and free memory block lists.

Arguments:

    - Nothing

Returns:

    - Nothing

Notes:

    - This destructor is automatically called when the MemoryManager object goes out of scope or is explicitly deleted.

    - It is responsible for releasing the memory used by all allocated memory blocks, preventing memory leaks.

Function: process\_transaction

Use: Processes a transaction string, performing memory management operations based on the specified actions.

Arguments:

    - transaction (const string&): The input string representing a memory management transaction.

    - memory\_manager (MemoryManager&): A reference to the MemoryManager object responsible for managing memory.

    - variables (unordered\_map<string, int>&): A mapping of variable names to their corresponding memory block start addresses.

Functionality:

    - Parses the input transaction string to extract variable name, action, and size (if applicable).

    - Performs memory management operations based on the specified action:

        - "allocate": Allocates a memory block of the specified size and associates it with the given variable.

        - "free": Deallocates the memory block associated with the specified variable.

        - Variable assignment: Copies the memory block address from one variable to another, increasing the reference count.

    - Outputs error messages to the standard error stream for unsupported operations or incorrect syntax.

Arguments:

    - transaction (const string&): The input transaction string to be processed.

    - memory\_manager (MemoryManager&): A reference to the MemoryManager object for memory management operations.

    - variables (unordered\_map<string, int>&): A mapping of variable names to their corresponding memory block start addresses.

Returns:

    - Nothing

Notes:

    - This function is a central part of the memory management system, interpreting and executing transactions.

    - It handles memory allocation, deallocation, and variable assignments based on the specified actions.

    - Errors and unsupported operations are reported to the standard error stream.

Function: main

Use: Entry point of the program, responsible for initializing memory management, processing transactions, and printing

     the final memory status to an output file.

Functionality:

    - Creates a MemoryManager object with a specified total memory size.

    - Initializes an unordered\_map to store variable names and their corresponding memory block start addresses.

    - Attempts to open input and output files, displaying error messages if unsuccessful.

    - Reads each line from the input file, processes transactions using the MemoryManager and variables map.

    - Redirects cout to the output file to print the final memory status.

    - Resets cout to its original buffer after printing.

    - Closes input and output files.

Parameters:

    - None

Returns:

    - 0 if the program executes successfully, 1 otherwise.

Notes:

    - The program processes memory management transactions from an input file and outputs the final memory status to

      an output file.

    - Errors during file operations are reported to the standard error stream.