CECS 342 - Lab Assignment 3 - Dynamic Memory Management

Due Date: Saturday, March 16th

Team Members: Bryan Tineo & Maxwell Guillermo

Completion of Lab Assignment:

Both team members contributed equally and collaborated throughout the completion of the lab assignment.

Code Main.c:

```
// Standard library includes for input/output and dynamic memory allocation
#include <stdio.h>
#include <stdlib.h>

// Definition of a Block structure for managing dynamic memory allocation
struct Block
{
   int block_size;            // Size of the data portion of the block
   struct Block *next_block; // Pointer to the next block in a linked list
};

// Constants representing the size of a Block structure and the size of a pointer
const int OVERHEAD_SIZE = sizeof(struct Block); // Size of the metadata (Block
structure)
const int POINTER_SIZE = sizeof(void *);         // Size of a pointer, used to align
allocations
struct Block *free_head;                         // Global variable pointing to the
head of the free list

// Function to initialize the heap (dynamic memory area managed by this allocator)
void my_initialize_heap(int size)
{
   // Allocate memory for the heap, including space for the Block structure itself
   //(struct Block *): This is a type cast. The malloc function returns a pointer of
type void*, which is a generic pointer type in C that can point to any type of data.
```

```
    // However, in C++, and also in C when you need to use the pointer with a specific
type, you often cast this void* pointer to the desired data type. In this case, it's
being cast to a pointer of struct Block
    free_head = (struct Block *)malloc(size + sizeof(struct Block));
    if (free_head != NULL) // Check if allocation was successful
    {
        // Initialize the first block in the heap
        free_head->block_size = size; // Set block size
        free_head->next_block = NULL; // Currently, there is no next block
    }
}

// Function to allocate memory from the heap
void *my_alloc(int size)
{
    if (size <= 0) // Ensure requested size is positive
    {
        printf("Size must be greater than 0.\n");
        return NULL; // Return NULL for invalid size requests
    }

    // Adjust the requested size for alignment and add overhead for the block metadata

    // Assume size is the requested size (14 bytes) and POINTER_SIZE is 8 bytes (on a
64-bit system).
    // First step: Add POINTER_SIZE - 1 to the requested size. This ensures that if the
requested size
    // is not a multiple of POINTER_SIZE, it gets rounded up to the next multiple.
    // Calculation: size + POINTER_SIZE - 1 = 14 + 8 - 1 = 21.
    // Second step: Apply bitwise AND with ~(POINTER_SIZE - 1) to the result from the
first step.
    // This operation zeroes out the least significant bits to align the size up to the
nearest multiple of POINTER_SIZE.
    // Calculation:
    // 1. ~(POINTER_SIZE - 1) creates a mask. For POINTER_SIZE = 8, POINTER_SIZE - 1 =
7, which is 00000111 in binary.
    //    Applying bitwise NOT (~) to 00000111 gives us 11111000, which zeroes out the
three least significant bits.
    // 2. Apply this mask to the result from the first step (21 in decimal or 00010101
in binary) using bitwise AND.
    //    00010101 & 11111000 results in 00010000, which is 16 in decimal.
```

```c
    // The alignedSize is therefore 16, which is the nearest multiple of 8 (the
POINTER_SIZE)
    // that is at least as large as the original size request of 14.
    int alignedSize = (size + POINTER_SIZE - 1) & ~(POINTER_SIZE - 1); // Align size up
to nearest pointer size

    // After aligning size, this line adds the size of the overhead (OVERHEAD_SIZE),
which is the size of the Block structure that precedes the user's memory block in this
custom allocator's implementation.
    // This overhead is necessary to keep track of the block's properties, such as its
size and a pointer to the next block in a memory management list.
    int requiredSize = alignedSize + OVERHEAD_SIZE; // Total size required including
overhead

    struct Block *curr = free_head; // Start at the head of the free list
    struct Block *prev = NULL;      // Previous block pointer for traversal

    // Traverse the free list to find a suitable block
    while (curr != NULL)
    {
        if (curr->block_size >= requiredSize) // Check if the current block is large
enough
        {
            // Determine if there's enough space in the current block to split it
            if (curr->block_size >= requiredSize + OVERHEAD_SIZE + POINTER_SIZE)
            {
                // Split the block
                // Calculate the starting address of the new block by adding the
required size to the current block's address.
                // This operation is done in two steps:
                // 1. (char *)curr - Casts the current block's pointer to a char* to
enable byte-level arithmetic.
                //    This is necessary because adding to a char* pointer increments
the address by bytes, which allows
                //    for precise memory address calculation needed in memory
allocation routines.
                // 2. + requiredSize - Adds the size of the memory that is being
allocated to the current block's address.
                //    The 'requiredSize' includes the size of the memory requested by
the user as well as the overhead for
                //    the block's metadata. This calculation effectively finds the
start of the new (split) block in memory,
```

```
            //    positioned immediately after the space being allocated to fulfill
the current request.
            struct Block *newBlock = (struct Block *)((char *)curr + requiredSize);

            newBlock->block_size = curr->block_size - requiredSize; // Set new
block's size
            newBlock->next_block = curr->next_block;                // Link new
block to the next block

            curr->block_size = alignedSize; // Update current block's size

            // Update the free list
            // checks if the block being split is the first block in the free list.
            // This is inferred from the prev pointer being NULL, indicating that
we haven't traversed any part of the list before reaching the block that we're
currently working with (curr).
            if (prev == NULL) // If splitting the first block in the list
            {
                // sets the global free_head pointer to point to newBlock.
                // Since the block being split is the first in the list, updating
free_head is necessary to ensure the linked list's integrity.
                // newBlock is the remaining part of the split and now becomes the
first block in the free list.
                free_head = newBlock; // Set free_head to point to the new block
            }
            else // If not the first block
            {
                // "else" case handles the situation where the block being split is
not the first block in the list. This means we have a prev (previous) block.

                // prev->next_block = newBlock; updates the next_block pointer of
the prev (previous) block to point to newBlock.
                // This action inserts newBlock into its correct position in the
linked list, maintaining the continuity of the free list.
                // Since newBlock represents the leftover memory after the split,
this ensures that it's properly linked from the previous block, effectively updating
the list to reflect the new state of the memory blocks.
                prev->next_block = newBlock; // Update previous block to point to
the new block
            }
        }
        else // If not enough space to split, allocate the entire block
```

```
        {
            // When the allocator determines there's not enough space left in a block to split it (meaning, there isn't enough space after fulfilling the current request to create a new, smaller free block that meets the minimum size requirements),
            // it opts to allocate the entire block. After deciding this, the allocator must update the free list to remove the allocated block.

            // This condition checks if the current block (curr) is the first block in the free list.
            // This is determined by checking if prev is NULL, which indicates that curr is at the start of the list since we haven't moved past any other blocks in our traversal.
            if (prev == NULL) // If allocating the first block
            {

                // To remove the first block from the free list (since it's being allocated in its entirety), the allocator updates free_head to point to the next block (curr->next_block).
                //  This effectively removes curr from the free list, as free_head now references what was the second block in the list.
                free_head = curr->next_block; // Update free_head to skip the allocated block
            }
            else // If not the first block
            {
                // Since curr is being allocated, it needs to be removed from the free list.
                // To do this, the allocator sets the next_block pointer of the previous block (prev) to curr's next block (curr->next_block).
                // This action effectively skips over curr in the list, removing it and linking prev directly to curr's subsequent block
                prev->next_block = curr->next_block; // Remove the current block from the list by updating previous block's next pointer
            }
        }

        // Return a pointer to the allocated memory (data portion of the block):
        // When allocating memory from a custom heap, each block of memory managed by the allocator consists of two parts:
        // 1. Metadata (Overhead): Contains management information such as the block's size and a pointer to the next free block.
        //    The size of this metadata is defined by OVERHEAD_SIZE.
```

```c
            // 2. User Data Area: The actual space available for user data, located
immediately after the metadata.
            //
            // The return statement breakdown:
            // - (char *)curr: Casts the current block's pointer to a char*. Since a
char is 1 byte in C, this allows for precise byte-level arithmetic.
            // - + OVERHEAD_SIZE: Adds the size of the overhead to the block's starting
address, moving the pointer to the beginning of the user data area.
            // - (void *): Casts the result to void* to return a generic memory block
pointer, enabling the caller to cast it to any type as needed.
            //
            // Adjusting the pointer by OVERHEAD_SIZE is crucial to prevent the user
from accidentally overwriting the metadata. This adjustment provides a pointer that
safely points to the start of the user data area, ensuring the integrity of the
block's management information.
            return (void *)((char *)curr + OVERHEAD_SIZE);

            // Consider a block with a total size of 32 bytes, where the OVERHEAD_SIZE
is 8 bytes.
            // If curr points to the start of this block (byte 0), then (char *)curr +
OVERHEAD_SIZE points to byte 8, which is the start of the user data area.
            // The function returns this address, ensuring the first 8 bytes (the
metadata) are preserved and not overwritten by user data.
        }

        // Move to the next block in the list
        prev = curr;
        curr = curr->next_block;
    }

    // If no suitable block was found, return NULL
    return NULL;
}

// Function to free allocated memory and add it back to the free list
// The my_free function is responsible for freeing memory that was previously
allocated with a custom memory allocation function (like my_alloc)
void my_free(void *ptr)
{
    if (ptr == NULL) // Do nothing if NULL pointer is passed
        return;
```

```
    // This line calculates the address of the Block structure that precedes the user data in memory.
    // When memory is allocated, the user receives a pointer to the space immediately after the Block structure (the metadata).
    // To free the memory, the function needs to access this Block structure, so it subtracts the size of the overhead (OVERHEAD_SIZE) from the given data pointer (ptr).
    // This calculation effectively "rewinds" the pointer to the start of the Block structure.
    struct Block *blockToFree = (struct Block *)((char *)ptr - OVERHEAD_SIZE);

    // The block is then added back to the free list.
    // It does this by setting its next_block pointer to the current free_head (the start of the free list) and then updating free_head to point to this block.
    // This effectively inserts the block at the beginning of the free list.
    blockToFree->next_block = free_head;
    free_head = blockToFree;
}

// First test case: Allocate and then free an integer, followed by allocating another integer
void menuOptionOne()
{
    // Allocate memory for an integer and print its address
    int *numOne = my_alloc(sizeof(int));
    printf("Address of int A: %p\n", numOne);
    // Free the previously allocated memory
    my_free(numOne);

    // Allocate memory for another integer and print its address
    // This demonstrates that the allocator reuses freed memory
    int *numTwo = my_alloc(sizeof(int));
    printf("Address of int B: %p\n", numTwo);
};

// Second test case: Allocate two integers and check their addresses
void menuOptionTwo()
{
    // Allocate memory for the first integer and check for allocation failure(Checking for enough memory to allocate)
    int *numOne = my_alloc(sizeof(int));
    if (numOne == NULL)
    {
```

```
        printf("Allocation for int A failed.\n");
        return; // Return early if allocation failed
    }
    printf("Address of int A: %p\n", (void *)numOne);

    // Allocate memory for the second integer and check for allocation failure(Checking
for enough memory to allocate)
    int *numTwo = my_alloc(sizeof(int));
    if (numTwo == NULL)
    {
        printf("Allocation for int B failed.\n");
        return; // Return early if allocation failed
    }
    printf("Address of int B: %p\n", (void *)numTwo);

    // Verify and print the distance between the two allocated integers
    printf("Verifying Results...\n");

    // OVERHEAD_SIZE refers to the size of the metadata for each block allocated by the
custom memory allocator.
    // This metadata typically includes information like the block size and pointers
for managing free blocks in a linked list.
    //(sizeof(int) > POINTER_SIZE ? sizeof(int) : POINTER_SIZE) compares the size of an
integer (sizeof(int)) with the size of a pointer (POINTER_SIZE) on the system.
    // It selects the larger of the two. This comparison is crucial because the
allocator might need to align allocated memory blocks to the larger of these sizes to
adhere to system or architecture alignment requirements, ensuring efficient memory
access.

    // overheadPlusLarger then represents the total minimum overhead for each allocated
block, combining the static overhead for the block metadata and the dynamic part which
ensures alignment.
    int overheadPlusLarger = OVERHEAD_SIZE + (sizeof(int) > POINTER_SIZE ? sizeof(int)
: POINTER_SIZE);
    printf("Size of overhead + larger of (the size of an integer; the minimum block
size): %d bytes\n", overheadPlusLarger);

    // This calculates the byte-wise distance between the memory addresses of the first
and second allocated integers.
    // Since the pointers are cast to (char *), subtracting them gives the distance in
bytes, as each char represents one byte
    int distance = (char *)numTwo - (char *)numOne;
```

```
    printf("Address B - Address A: %d bytes\n", distance);
}

// Third test case: Allocate three integers, free the second, and allocate other data types
void menuOptionThree()
{

    // Allocating Three Integers: The function starts by allocating memory for three integers (numOne, numTwo, numThree) and prints their addresses.
    // This sets up a scenario where the memory is being used, and there are allocated blocks in the allocator's managed space.
    int *numOne = my_alloc(sizeof(int));
    printf("Address of int A: %p\n", numOne);
    int *numTwo = my_alloc(sizeof(int));
    printf("Address of int B: %p\n", numTwo);
    int *numThree = my_alloc(sizeof(int));
    printf("Address of int C: %p\n", numThree);

    // Freeing One Integer: It then frees the memory allocated for the second integer (numTwo).
    // This action introduces a "hole" in the memory space managed by the allocator, as there's now a block of memory that has been returned to the pool of free memory, making it available for future allocations.
    my_free(numTwo);

    // Allocate memory for an array of 2 double values and print its address
    // This tests the allocator's ability to handle requests of different sizes and types
    printf("After freeing int B...\n");

    // double *arr = my_alloc(2 * sizeof(double)); This line requests memory sufficient to store two double values.
    // The size of a double is typically larger than the size of an int (often double is 8 bytes and int is 4 bytes on many systems), so this allocation tests the allocator's ability to handle a request for a larger block of memory than was previously freed (numTwo was an int, smaller than two doubles).
    double *arr = my_alloc(2 * sizeof(double));
    printf("Address of array of 2 double values: %p\n", arr);

    // Allocate memory for another integer and print its address
```

```
    // Allocating another integer (numFour) after freeing numTwo and allocating space
for two doubles tests the allocator's space reuse efficiency.
    // If numFour is allocated at numTwo's previous address, it indicates effective
reuse of freed space, suggesting a "first fit" or "best fit" allocation strategy.
    // This behavior highlights the allocator's approach to managing freed space and
its strategy for minimizing fragmentation. A different address for numFour could imply
alternative strategies or impacts from the double array allocation on the free list's
structure.
    int *numFour = my_alloc(sizeof(int));

    printf("Address of int D (should be the int B): %p\n", numFour);
};

// Fourth test case: Allocate a char and an int, and compare their addresses
void menuOptionFour()
{
    // Allocate memory for a char and print its address
    char *charOne = my_alloc(sizeof(char));
    printf("Address of char A: %p\n", charOne);

    // Allocate memory for an integer and print its address
    // This tests how the allocator manages different data sizes and alignment
requirements
    int *numTwo = my_alloc(sizeof(int));
    printf("Address of int B: %p\n", numTwo);
};

// Fifth test case: Allocate a large array and verify the allocator's behavior
void menuOptionFive()
{
    // This line allocates memory for an array of 80 integers and prints the array's
starting address.
    // This tests the allocator's capability to handle large memory requests.
    int *arr = my_alloc(80 * sizeof(int));
    printf("Address of array: %p\n", arr);

    // Following the large array allocation, it allocates memory for a single integer.
    // This step tests how the allocator handles smaller allocations immediately
following a large one.
    int *numOne = my_alloc(sizeof(int));
    printf("Address of int A: %p\n", numOne);
    printf("Value of int A: %d\n", *numOne);
```

```c
    // The code calculates and prints the byte-wise difference between the start of the
large array and the single integer allocation.
    // This can reveal how much space is used or left between two allocations and might
help in understanding the internal memory management strategies, such as padding or
alignment requirements.

    //(char *)numOne - (char *)arr: This calculates the byte-wise distance between the
start address of the single integer allocation (numOne) and the start address of the
array (arr).
    // Since both pointers are cast to (char *), subtracting them gives the distance in
bytes.
    //- 80 * sizeof(int): This part of the expression subtracts the total size of the
allocated array (80 integers) from the previously calculated distance.
    printf("Difference between array start and int A: %ld bytes\n", (char *)numOne -
(char *)arr - 80 * sizeof(int));

    // Free the array to test the allocator's ability to reclaim and reuse freed memory
    my_free(arr);

    // After freeing the array, verify and print the address and value of the
additional integer
    // This checks if the address and value remain consistent after freeing adjacent
memory
    printf("After freeing array...\n");
    printf("Address of int value: %p\n", numOne);
    printf("Value of int A: %d\n", *numOne);
}

// Main function to run the allocator tests
int main()
{
    int menuChoice = 0; // Variable to store the user's menu choice
    int runAgain = 1;   // Flag to control the menu loop

    // Initialize the custom heap with a specific size before running tests
    my_initialize_heap(1000);

    // Loop to repeatedly show the menu and process user input until the user chooses
to quit
    while (runAgain == 1)
    {
```

```
        // Display menu options to the user
        printf("\n1. Allocate an int \n2. Allocate two ints \n3. Allocate three ints
\n4. Allocate one char \n5. Allocate space for an 80-element int array \n6. Quit
\nChoose a menu option: ");
        // Read the user's menu choice
        scanf("%d", &menuChoice);
        // Announce the selected test case
        printf("\n---Test Case %d---\n", menuChoice);

        // Switch statement to execute the appropriate test case based on the user's
choice
        if (menuChoice == 1)
        {
            menuOptionOne(); // Run the first test case
        }
        else if (menuChoice == 2)
        {
            menuOptionTwo(); // Run the second test case
        }
        else if (menuChoice == 3)
        {
            menuOptionThree(); // Run the third test case
        }
        else if (menuChoice == 4)
        {
            menuOptionFour(); // Run the fourth test case
        }
        else if (menuChoice == 5)
        {
            menuOptionFive(); // Run the fifth test case
        }
        else if (menuChoice == 6)
        {
            // If the user chooses to quit, print a message and exit the loop
            printf("Done!");
            runAgain = 0; // Set flag to exit the loop
        }
    }
    return 0; // End of program
}
```

Output:

Test Case #1:

```
maxi@dhcp-39-9-135 Lab-assignment-3---Dynamic-Memory-Management % ./test

1. Allocate an int
2. Allocate two ints
3. Allocate three ints
4. Allocate one char
5. Allocate space for an 80-element int array
6. Quit
Choose a menu option: 1

---Test Case 1---
Address of int A: 0x12c008810
Address of int B: 0x12c008828
```

Explanation:

In Test Case 1, the program gives us a small piece of memory to store a number, an integer. The program keeps track of all its free chunks of memory in a list. When we ask for memory, it checks this list for a chunk that's just the right size and hands it over.Then we tell the program we're done with that chunk of memory. It takes it back and puts it at the top of the free memory list, ready to be used again. The next time we ask for a piece of memory of the same size, the program gives us the chunk we just returned since it's now at the top of the list. This is a simple and quick way to manage memory using a list, making sure no memory gets wasted and everything we return can be reused efficiently. The slight difference in the last digit of the memory addresses for int A and int B can be explained by the memory allocator's handling of overhead and alignment. Each allocation includes a small overhead for management purposes and aligns memory to certain boundaries for efficiency. Therefore, even though int B reuses the memory released by int A, the actual address it occupies may be slightly adjusted due to these requirements, resulting in the observed variation in the last digit of the addresses.

Test Case #2:

```
---Test Case 2---
Address of int A: 0x12c008840
Address of int B: 0x12c008858
Verifying Results...
Size of overhead + larger of (the size of an integer; the minimum block size)
: 24 bytes
Address B - Address A: 24 bytes
```

Explanation:

Just like in the previous example, the program is using a list to keep track of free memory. In Test Case 2, we asked for space to store two numbers one after the other. The program gave us a spot for the first number (int A) and then another spot for the second number (int B). The output tells us that the second spot is 24 bytes away from the first. That gap is precisely the room needed for the program's bookkeeping (which is like a sticky note telling the program what this piece of memory is for) plus a little extra to make sure everything lines up nicely in memory, which computers like. So, this test is making sure the program isn't just keeping track of the memory it's using but also organizing it neatly, so it's easy for the computer to work with. The addresses being 24 bytes apart confirms the program is doing this right.

Test Case #3:

```
---Test Case 3---
Address of int A: 0x12c008870
Address of int B: 0x12c008888
Address of int C: 0x12c0088a0
After freeing int B...
Address of array of 2 double values: 0x12c0088b8
Address of int D (should be the int B): 0x12c0088d8
```

Explanation:

In Test Case 3, after int B is freed, its space goes back to the free list, and then a request for an array of two double values comes in. This array is larger than a single integer, so the allocator looks for a suitable space. If the space from int B isn't enough, due to the size and alignment requirements for doubles, it must find a different block that's large enough. When we ask for space for int D, the allocator assigns it the next available spot. The address for int D is different from int B because, although int B's space was reused for the array of doubles, the allocator must align the memory for int D correctly. This alignment ensures efficient memory access and can result in different starting addresses even for blocks of the same type, as seen with the different address for int D.

Test Case #4:

```
---Test Case 4---
Address of char A: 0x12c0088f0
Address of int B: 0x12c008908
```

Explanation:

In Test Case 4, the program is tasked with allocating memory for different data sizes in a structured way. Initially, it allocates a small segment for a character (char A), and it marks this place in its memory list. Then, when a larger segment is needed for an integer (int B), it consults the list again. The program allocates the larger segment at a new location, but it also ensures there's a buffer zone between the two. This buffer is not just empty space; it's essential for alignment, which keeps data access efficient and prevents errors. The allocator's strategy ensures that smaller data doesn't crowd larger data, much like careful packing ensures delicate items don't get crushed. This approach demonstrates the allocator's ability to maintain order and accessibility within memory, which is critical for system reliability and performance.

Test Case #5:

```
---Test Case 5---
Address of array: 0x12c008920
Address of int A: 0x12c008a70
Value of int A: 0
Difference between array start and int A: 16 bytes
After freeing array...
Address of int value: 0x12c008a70
Value of int A: 0
```

Explanation:

In Test Case 5, the memory allocator is instructed to reserve a substantial block for a number array and then a smaller block for a single integer (int A). The allocator efficiently assigns two appropriately sized blocks, ensuring the single integer closely follows the larger array block. After releasing the array block back to the system, a verification shows that the integer remains unaffected in its designated location. This demonstrates the allocator’s precision in handling both allocation and deallocation, confirming that it effectively segregates different-sized blocks while maintaining the integrity of each within the memory's structure.