Memory Management Code Documentation

Overview

This document explains the implementation of a memory management system in C. The code manages a fixed-size memory array using a custom structure, ‘struct data‘, to keep track of block metadata.

Code Listing

Below is the full implementation of the memory management code:

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>
#define MEMORY_SIZE 25000

struct data {
    int size;
    bool isAllocated;
};//this struct describe block size and isallocate or not

char Memory[MEMORY_SIZE] = {0};

int MyCheck(int size) {
    int i = 0;

    while (i < MEMORY_SIZE) {//check full array
        struct data *meta = (struct data *)&Memory[i];

        if (meta->isAllocated) {// if block is allocated then move to next space
            i = i + meta->size + sizeof(struct data);
        }
        else {
            bool loopStatus = false;
            int index;
            for(index = i; index <(i + meta->size + sizeof(struct data)) && index <
    MEMORY_SIZE; index++)
            {// if block is allocated then move to next space
                if (meta->isAllocated) {
                    i = index;
                    loopStatus = false;
                    break;
                }
                loopStatus = true;//free space allocation flag
            }
            if (loopStatus) {// if has free space print details
                printf("Starting [status box]: %d\n", i);
                printf("Ending [box]: %d\n", i + size + sizeof(struct data) - 1);
                return i;
            }
        }
    }
    return -1;
}

void SetBlock(int index, int size) {
    struct data *meta = (struct data *)&Memory[index];
    meta->size = size;
    meta->isAllocated = true;  }// update data

void *MyMalloc(int size) {
```

```
    int ptr = MyCheck(size);
    if (ptr == -1) {
        printf("Memory Allocation failed\n");
        return NULL;
    } else {
        SetBlock(ptr, size);
        return &Memory[ptr + sizeof(struct data)];
    }// malloc function handle
}

void MyFree(void *ptr) {
    if (ptr == NULL) {
        printf("Invalid pointer. Nothing to free.\n");
        return;
    }

    int blockStart = (char *)ptr - Memory - sizeof(struct data);

    if (blockStart < 0 || blockStart >= MEMORY_SIZE) {
        printf("Pointer out of bounds. Free failed.\n");
        return;
    }

    struct data *meta = (struct data *)&Memory[blockStart];
    if (!meta->isAllocated) {
        printf("Memory block not allocated or already freed.\n");
        return;
    }

    meta->isAllocated = false;
    printf("Memory block starting at index %d has been freed.\n", blockStart);
}// free up the heap

int GetBlockSize(int index) {
    struct data *meta = (struct data *)&Memory[index];
    return meta->size;
}

int main() {
    void *allocatedMemory = MyMalloc(1000);

    if (allocatedMemory) {
        printf("Memory allocated at address: %p\n", allocatedMemory);

        struct data *meta = (struct data *)&Memory[0];
        printf("Block Marker: [%d], Block Size: [%d]\n", meta->isAllocated, meta->size);
    }

    MyFree(allocatedMemory);

    MyFree(allocatedMemory);

    return 0;
}
```

Listing 1: Memory Management Code

Explanation of Components

Structure Definition

- `struct data`:
  - `size`: Specifies the size of the memory block.
  - `isAllocated`: A flag indicating whether the block is allocated or free.

Functions

- `MyCheck(int size)`: Checks the memory array for a free block of sufficient size. Returns the starting index of the block if found, or -1 if no suitable block is available.

- `SetBlock(int index, int size)`: Updates the metadata at the given index to mark the block as allocated.
- `void *MyMalloc(int size)`: Allocates a block of memory of the specified size. If successful, returns a pointer to the usable memory; otherwise, returns `NULL`.
- `void MyFree(void *ptr)`: Frees the allocated memory block at the specified pointer.
- `int GetBlockSize(int index)`: Returns the size of the block at the specified index.

Usage Example

The `main()` function demonstrates the usage of `MyMalloc()` and `MyFree()`:

- Memory of size 1000 bytes is allocated using `MyMalloc()`.
- The metadata (block size and allocation status) is printed.
- The allocated memory is freed using `MyFree()`.
- A second attempt to free the same memory block is made to demonstrate error handling.

Output

Example output for the program:

```
Starting [status box]: 0
Ending [box]: 1035
Memory allocated at address: 0x55ccf1d04060
Block Marker: [1], Block Size: [1000]
Memory block starting at index 0 has been freed.
Memory block not allocated or already freed.
```