MY_MALLOC DOCUMENTATION

mymal.h:

- This file contains the declarations for the my_malloc and my_free functions.
It contains a rather lengthy comment for each one of them describing how the code will work as well as describing the arguments for each function.
-It also contains the struct called mem_block which is the mem_entry equivalent for my own malloc implementation. The fields of this struct are as follows.

1. int size: the size of the block of memory described by this struct.
2. next / prev: it is the implementation of my double linked list of structs that model the heap and alocated memory
3. free : an integer indicator for whether a given block is free or not. 0 means taken , 1 means free.
4. int line / char *file : The variables for storing the __LINE__ and __FILE__ arguments from malloc , used for printing diagnostics for my_malloc.
5. int signature : a method of comparason to check if the pointer passed actually does belong to what we malloced and doesnt just happen to line up with some section of memory in the heap for some reason. Used for diagnostics and error printing.

There is also a typedef statement: typedef struct mem_block *pt_block;
this is short for “pointer to block” just meant to save some typing.

This is it as far as the functions in mymal.h since the user does not need to know of the helper functions used to implement my malloc and my_free.

mymal.c

- This file contains the definitions for both my_malloc and my_free along with all the helper functions given to make malloc and free possible.

The list of macros in the mymal.c file – all of them are more for readability than functional use

```
1. #define BLOCK_SIZE        sizeof(struct mem_block)
2. #define SIGN                      1010101010
3. #define RED                       "\x1b[2;31;31m"
4. #define RESET               "\x1b[0m"
```

1. BLOCK_SIZE :	just a replacement for typing the sizeof(struct mem_block) , readability increase
2. SIGN :	is the chosen signature saved in the struct mem_block
3. RED	Color code for print statements, makes the code neater, avoid retyping ugly string
4. RESET	Color code for reset , again to make code neater and avoid retyping an ugly string

The list of functions is as follows:

```
1. int  is_valid_memblock( void * , int , char*);
2. void set_meta( pt_block , int , char * );
3. void trim_to_size( pt_block , int);
4. void mem_leak_check( void );
5. void *get_block_struct(void *p);
6.  pt_block combine_blocks(pt_block);
```

```
pt_block heap_extend( pt_block , int);
pt_block get_free_block(pt_block *last, int size);
```

First thing to note is the “pt_block”

1. is_valid_memblock: The comment in code describes in detail how this function works. The broad reason for this is to asser whether or not the passed in pointer is actually allocated by my_malloc and in valid range as well as not corrupted. This function calls the get_block_struct function (described below).

2. set_meta : a simple function to follow the DRY principle. It sets the meta data of a block (__LINE__ , __FILE__) so as to not clutter the my_malloc code. The arguments are the struct , the line and file name

3. trim_to_size : this function looks at the block of memory obtained from heap_extend. Since heap extend returns the first free block that will fit the specified size , this function attempts to trim this block to an exact size requested to avoid internal fragmentation.

4. mem_leak_check : used in my_malloc function upon the first time the function is called. Mem_leak_check is passed to the atexit() function to be executed at program termination. It checks if there are any blocks of memory that have been my_malloc'd and not my_free'd. Then it prints their meta data(line and file) as an error (all in red)

5. get_block_struct : this function though somewhat badly named (sorry I couldnt think of a more descriptive way) is used in my_free. It takes a void *p passed into my free by the user, and casts the pointer to char., preforms pointer arithmetic to get from the memory pointer to the actual struct describing said memory and returns a pointer to the beginning of the struct. It is mostly for neater, modular code and to not repeat myself.

6. combine_blocks: This is one of the more important parts of my_malloc. This function looks at the block ahead of the specified one and checks if its is free and if it can combine the 2 blocks into one bigger chunk of memmory. This is the biggest way my_malloc deals with memory fragmentation, as it results in the maximum possible single free block. This is called every time a free is called and checks for both previous block and next block to be combined. Keeps the memory organized and much less fragmented.

7. heap_extend: This is an abstraction called when the malloc could not obtain a free memory block from get free block in the existing memory heap , then this function extends the heap using sbrk(size +BLOCK_SIZE) and puts a mem_block struct in front of – links said struct to the linked list, sets free = 0 since malloc is already requesting this to be used , and returns the pointer to malloc.

In general the way my_malloc works is :
1. check for valid size = non zero , non negative
2.check if root is null, if it is , then this is the first time malloc is called, set atexit(mem_leak_check) then extend heap, make root pointer returned from extend heap, set meta data and return the popinter right past the mem_block struct.
3. if root is not null
    - attempt to get free block using get_free_block.
    - if block was obtained, attempt to trim it to the size requestsed using trim_to_size.

-set free to 0
-set metadata and return it to user

4. If root was not null but get_free block failed , malloc :
   - extends heap
   - sets meta data and free =0
   - return the pointer to user.
5. Increment num alloc

The important part of this implementation is the trim to size as it prevents internal fragmentation of the heap. It is also worth noticing that trim to size only works if the memory left after the requested size is obtained is BLOCK_SIZE + 8. The 8 is there in case you need to malloc at least an int.

Description of my free:

1. check if we are trying to free a null pointer, error if its so
2. use is_valid_memblock to preform error checks on the pointer , if they pass we are good to go in freeing the memory.
3. check if pointer has a previous block that is free, if so combine blocks.
4. check if we can combine the next block with the current block – if yes do so.

   It is worth noting that this is where external fragmentation gets preveneted (to a degree) as the we no longer have 3 smaller blocks but one large chunk of memory available so if we lets say had 3x 400 byte blocks and theey all got freed we can now allocate a 1200 byte block in this space as opposed to being limited to exactly 400 bytes and having to extend the heap furhter.

5. If we cant combine because there is no next block, we are at the end of the heap so we just cut off the memory , make next pointer of our previous block null,set the heap break using brk() to the prev pointer.
6. decrement num_alloc