



---

# Operating System, IT308

---

## PROJECT: BETTER MALLOC

### *Author:*

Raj Jakasaniya - 201501408  
Dhruv Patel - 201501452

### *Guided by:*

Prof. Naresh Jotwani

## 1 Abstract

Aim of our project is to create a better version of malloc and free using macros from our header file named mymalloc.h,

Our version (or code) will be used for addressing errors properly by not returning a vague Segmentation Fault and printing the reason as well as the location of the cause of a failed malloc/free.

Free (void \*) will allow the user to tell the system that he/she is done with a dynamically allocated block of memory hence system will remove that block.

## 2 What is malloc

For a programmer, malloc is a Standard C Library function that allocates (i.e. reserves) memory chunks, most people don't know what is really behind, some even think it's a syscall or language keyword. In fact malloc is nothing more than a simple function and can be understood with a little C skills and almost no system knowledge. It obeys the following signature :

```
void * malloc(size_t size)
```

where size is the requested size. The returned pointer should be NULL in case of failure (no space left.). These blocks of memory will come from a 5000-byte (HEAP\_SIZE) char array defined in mymalloc.c

## 3 Common errors with malloc and free

When freeing an address to memory that was not dynamically allocated, the usual result is exiting the program. With our free, the user will instead be told that the address inputted was not on the heap. In this case, the heap is the 5000-byte char array that simulates virtual memory.

```
// inputted address was never returned by malloc
if(ptr == NULL){
    printf("NULL Pointer exception");
    return;
}
```

```
Metadata* meta_ptr = (Metadata*)(ptr - sizeof(Metadata));
```

```
if( meta_ptr < first_metadata ||(char*)ptr > & myblock [ max_size-1 ] ){
    printf("INVALID Pointer exception");
    return;
}
```

Existence of block is verified by the above function, firstly the pointer is compared with boundary condition (out of bounds check) for checking its correctness, after look-

ing through every meta block within virtual memory, the address entered pointed to the first byte of a user data block. If the block was not found, then the address is not a valid address in the heap. Therefore, the user will be told this instead of exiting the program.

When asking for more data than can be requested, the usual result is once again exiting the program with no explanation. With our malloc, it'll tell you the reason why the operation cannot be executed. For example, if you tried `malloc(5001)`, you will be told that there isn't enough space in virtual memory to allocate 5001 bytes. This is reasonable since our emulated memory is only 5000 bytes. Taking into account the size of the meta block (8 bytes), the max amount of memory that can be dynamically allocated is 4992 bytes. For other common errors in malloc and free, check the comments by the print statements in `mymalloc.c`

## 4 Working of malloc

**Meta data contains:**

- Size of the user data block.
- Flag of allocation to indicate whether it is allocated or not.
- Flag of last block to check whether it is last block or not.

The first malloc call will call our bootstrap method which essentially "sets up" virtual memory for us. Setting up virtual memory is basically initializing virtual memory with one meta block that covers all the memory left in the char array. Since a meta block is 8 bytes, the user data it encompasses is 4992 bytes as a result.

When allocating memory, malloc will search through every meta block and find the smallest one that can hold the requested amount of memory. For the first malloc of course, the first and only meta block will be the source of the address returned.

Assuming the user called `malloc(100)`, 4992 bytes is too much memory to supply the user. In fact, if the address, `meta_block + sizeof(meta)`, was returned, then over 4000 bytes would go to waste. So what happens is that the user data block you see above would be split and the result would be two meta blocks that correspond to two different user data blocks,

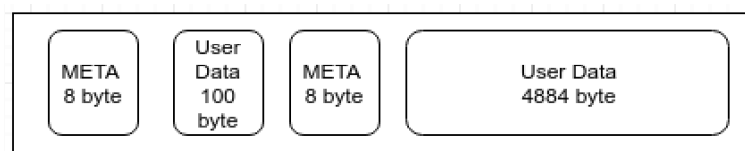


Figure 1: Heap diagram with two block of size (108,4892)

Assuming the first meta block corresponds to the user data to the immediate right of 100 bytes and the second meta block corresponds to the user data to the immediate right of the leftover  $4992 - 100 - 8 = 4884$  bytes, the address returned from a `malloc(100)` call

would be the address of the first meta block + sizeof(meta).

## 5 How free works

Calls to free will look at the inputted address and try to determine whether it is a valid address within main memory. Valid as in if it points to the first byte of a user data block that is in use, meaning it has not been freed already. Each meta block consists of a size variable and an allocated variable.

If the address inputted is valid, then the allocated variable of its corresponding meta block will be set to false.

Suppose the address of the second user data block was entered into free where virtual memory consisted of three meta blocks and three user data blocks with the first two meta blocks whose allocation is true while the last meta blocks allocation is false before the call to free the second block. Here would be the resulting main memory,

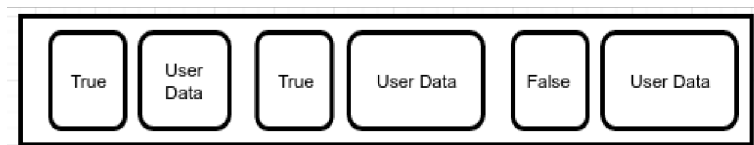


Figure 2: Before freeing second block

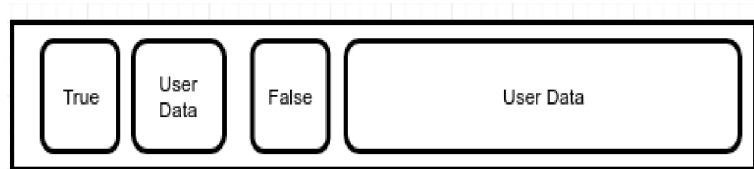


Figure 3: After free of second block

Now, Imagine if the address of the first user data block was entered into free where virtual memory consisted of three meta blocks and three user data blocks with the first two meta blocks whose allocation is true while the last meta blocks allocation is false before the call to free the first block. Here would be the resulting main memory,

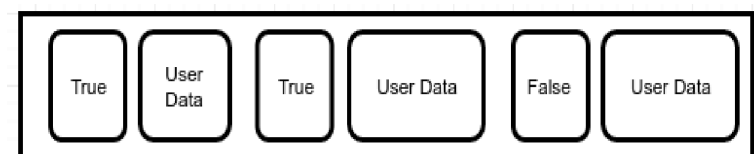


Figure 4: Before freeing second block

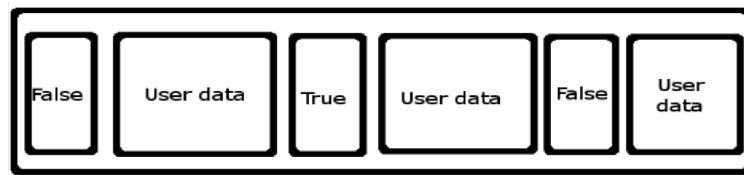


Figure 5: After free of second block

Now imagine if the second data block was requested to be freed, the interesting result would be a coalescing algorithm used in our free, which would be to look for nearby meta blocks to determine whether we could combine blocks that were not in use. So the resulting main memory from freeing the middle two blocks would be one meta block and its user data block that encompassed all of main memory.



Figure 6: Heap diagram with one block