Karl Lewis
Real Time Debugging

Project 1: Memory Management Report

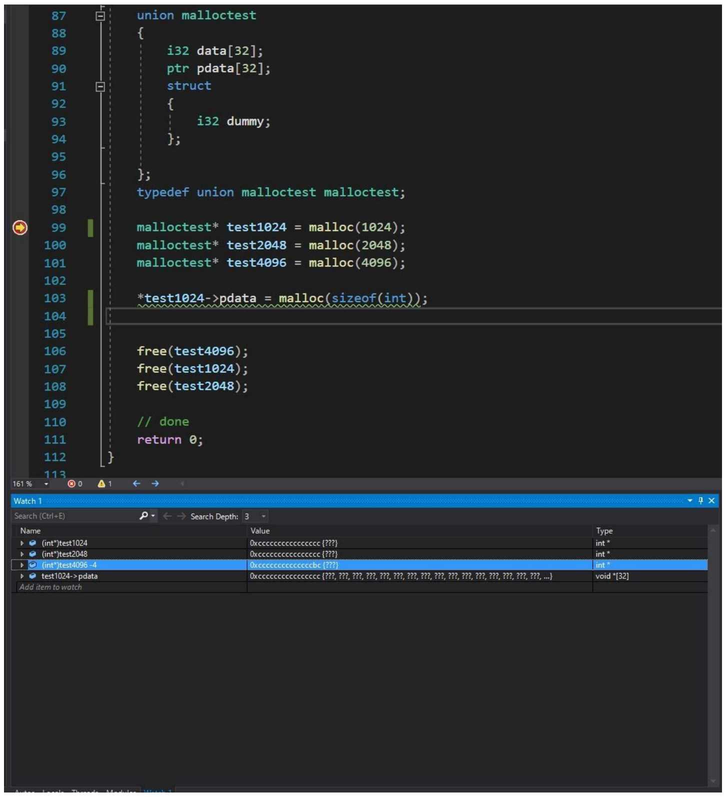
Description of Issues/Scenarios

The core issues at hand are related to determining the functionality of the **malloc and free** function pair in the C programming language, specifically how these two functions operate under the hood, including **what data structure is used** when allocating memory with malloc, **how memory blocks are organized** once allocated into this data structure, and how these organized blocks of allocated memory are **processed and reset** once free is called when the memory is no longer needed.

Details of Issues/Scenarios

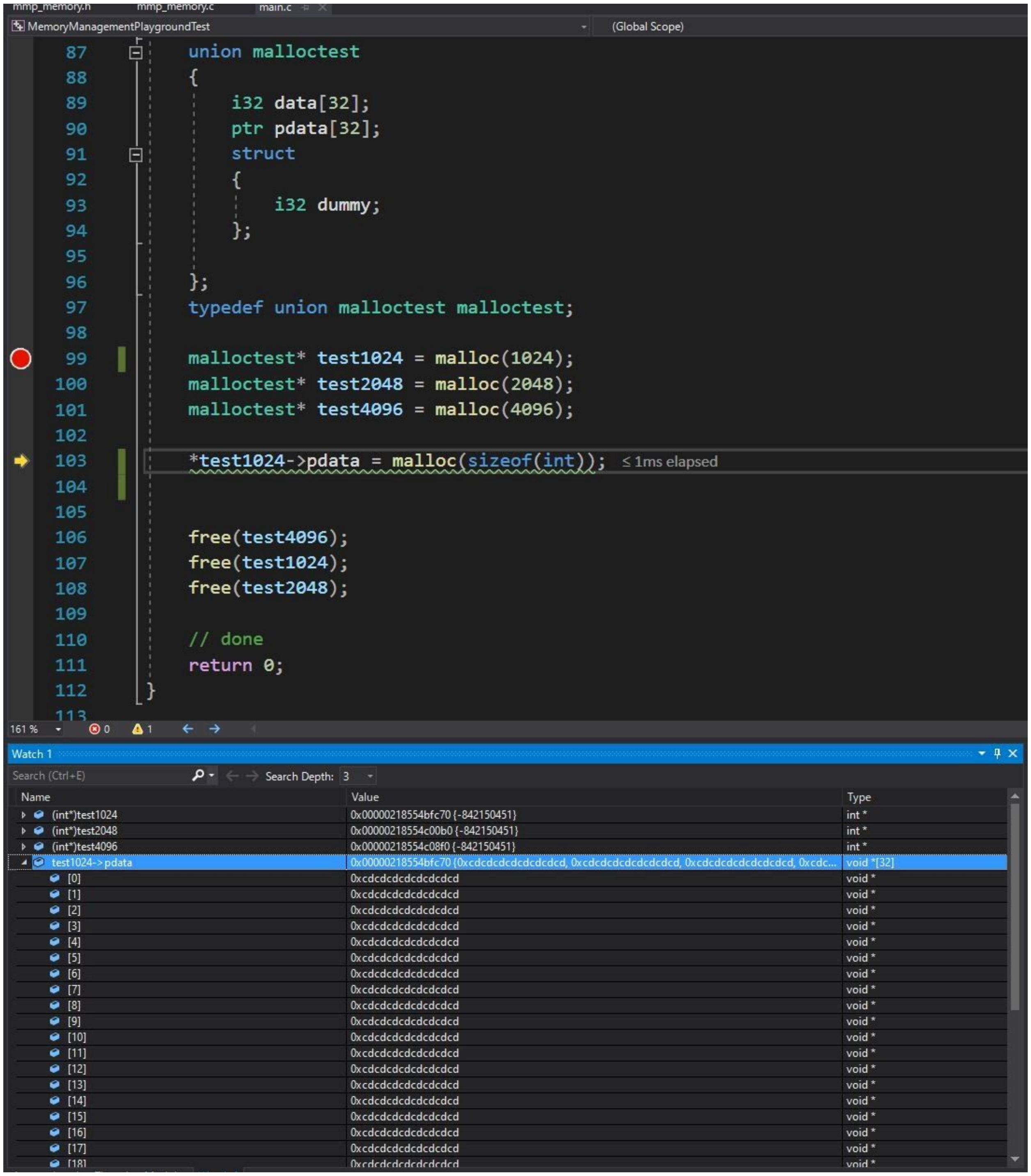
Malloc's Data Structure:
This issue pertains to the lowest level structure of malloc possible: the data structure and its contained variables that are used to determine various conditions and parameters when actually performing the memory allocation, e.g if a chunk of memory on the heap is free or not. Theoretically, this data structure should have some way of determining if a certain chunk of heap memory is free or not, some way to store the size of memory to be/currently allocated, and some way to return this data for use once allocation is finished. What is actually visible through code is hard to determine, but using breakpoints with watches using the mallocTest structure from class provides a bit of insight into what's actually happening:

Before stepping into the objects and initializing them, the address of each value is 0'd out to null since they haven't been allocated yet. Casting to (int*) allows us to see values behind the scenes as the memory is allocated.



```
union malloctest
{
    i32 data[32];
    ptr pdata[32];
    struct
    {
        i32 dummy;
    };

};
typedef union malloctest malloctest;

malloctest* test1024 = malloc(1024);
malloctest* test2048 = malloc(2048);
malloctest* test4096 = malloc(4096);

*test1024->pdata = malloc(sizeof(int));


free(test4096);
free(test1024);
free(test2048);

// done
return 0;
}
```

Name	Value	Type
(int*)test1024	0xcccccccccccccccc {???}	int *
(int*)test2048	0xcccccccccccccccc {???}	int *
(int*)test4096 -4	0xccccccccccccccbc {???}	int *
test1024->pdata	0xcccccccccccccccc {???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ...}	void *[32]
Add item to watch		



mmp_memory.h   mmp_memory.c   main.c

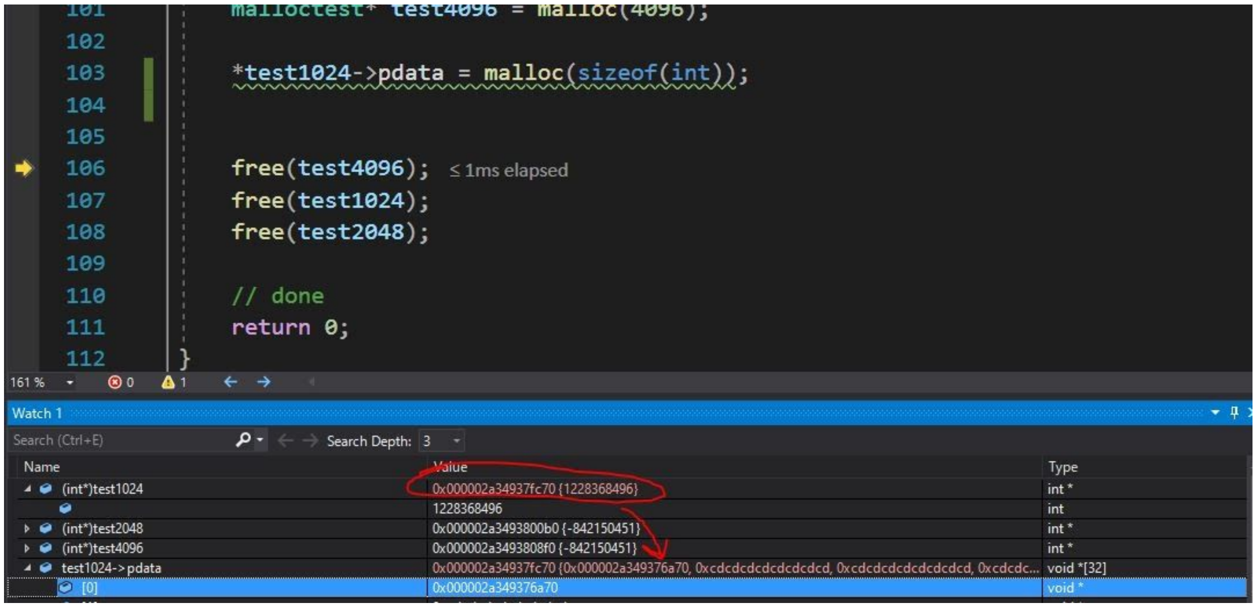
MemoryManagementPlaygroundTest   (Global Scope)

```c
union malloctest
{
    i32 data[32];
    ptr pdata[32];
    struct
    {
        i32 dummy;
    };

};
typedef union malloctest malloctest;

malloctest* test1024 = malloc(1024);
malloctest* test2048 = malloc(2048);
malloctest* test4096 = malloc(4096);

*test1024->pdata = malloc(sizeof(int));  ≤ 1ms elapsed


free(test4096);
free(test1024);
free(test2048);

// done
return 0;
}
```

Watch 1

Name	Value	Type
(int*)test1024	0x00000218554bfc70 {-842150451}	int *
(int*)test2048	0x00000218554c00b0 {-842150451}	int *
(int*)test4096	0x00000218554c08f0 {-842150451}	int *
test1024->pdata	0x00000218554bfc70 {0xcdcdcdcdcdcdcdcd, 0xcdcdcdcdcdcdcdcd, 0xcdcdcdcdcdcdcdcd, 0xcdc...	void *[32]
[0]	0xcdcdcdcdcdcdcdcd	void *
[1]	0xcdcdcdcdcdcdcdcd	void *
[2]	0xcdcdcdcdcdcdcdcd	void *
[3]	0xcdcdcdcdcdcdcdcd	void *
[4]	0xcdcdcdcdcdcdcdcd	void *
[5]	0xcdcdcdcdcdcdcdcd	void *
[6]	0xcdcdcdcdcdcdcdcd	void *
[7]	0xcdcdcdcdcdcdcdcd	void *
[8]	0xcdcdcdcdcdcdcdcd	void *
[9]	0xcdcdcdcdcdcdcdcd	void *
[10]	0xcdcdcdcdcdcdcdcd	void *
[11]	0xcdcdcdcdcdcdcdcd	void *
[12]	0xcdcdcdcdcdcdcdcd	void *
[13]	0xcdcdcdcdcdcdcdcd	void *
[14]	0xcdcdcdcdcdcdcdcd	void *
[15]	0xcdcdcdcdcdcdcdcd	void *
[16]	0xcdcdcdcdcdcdcdcd	void *
[17]	0xcdcdcdcdcdcdcdcd	void *
[18]	0xcdcdcdcdcdcdcdcd	void *

Once the objects are initialized, we can see that they all have the same value of -842150451. What this value actually means on its own is difficult to decipher and is the core issue at hand for determining this data structure. The use of the pdata member also reflects this, as every index is initialized to the same value of 0 despite the malloc call.



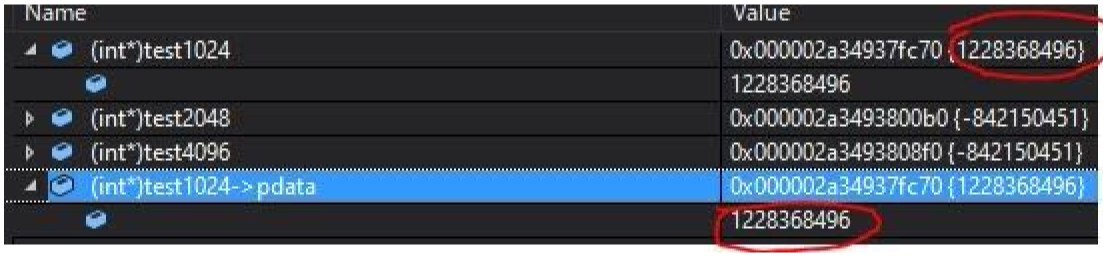
```
    mallocTest* test4096 = malloc(4096);

    *test1024->pdata = malloc(sizeof(int));


    free(test4096);  ≤ 1ms elapsed
    free(test1024);
    free(test2048);

    // done
    return 0;
}
```

Watch 1

Search (Ctrl+E)   Search Depth: 3

Name	Value	Type
(int*)test1024	0x000002a34937fc70 {1228368496}	int *
	1228368496	int
(int*)test2048	0x000002a3493800b0 {-842150451}	int *
(int*)test4096	0x000002a3493808f0 {-842150451}	int *
test1024->pdata	0x000002a34937fc70 {0x000002a349376a70, 0xcdcdcdcdcdcdcdcd, 0xcdcdcdcdcdcdcdcd, 0xcdcdc...	void *[32]
[0]	0x000002a349376a70	void *

When malloc is called on the pdata member within test1024, the value of test1024 changes to a different seemingly randomized garbage number. The first index's address in pdata also changes, but the subsequent indexes remain uninitialized.

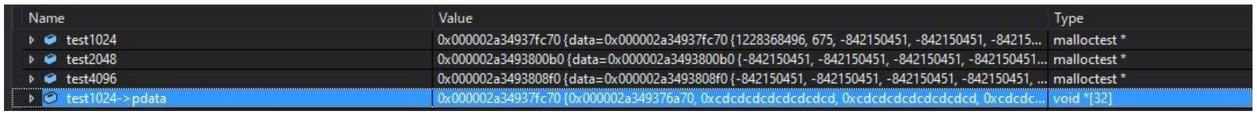


Name	Value
(int*)test1024	0x000002a34937fc70 {1228368496}
	1228368496
(int*)test2048	0x000002a3493800b0 {-842150451}
(int*)test4096	0x000002a3493808f0 {-842150451}
(int*)test1024->pdata	0x000002a34937fc70 {1228368496}
	1228368496

If we look at the value of pdata after its malloc'd, we can see that the value matches that of the object it resides within. What's happening here is that the member variables within our test objects have the same address in memory and value as the objects themselves when allocated despite the fact that they were allocated with different sizes.

Organization of Allocated Memory:

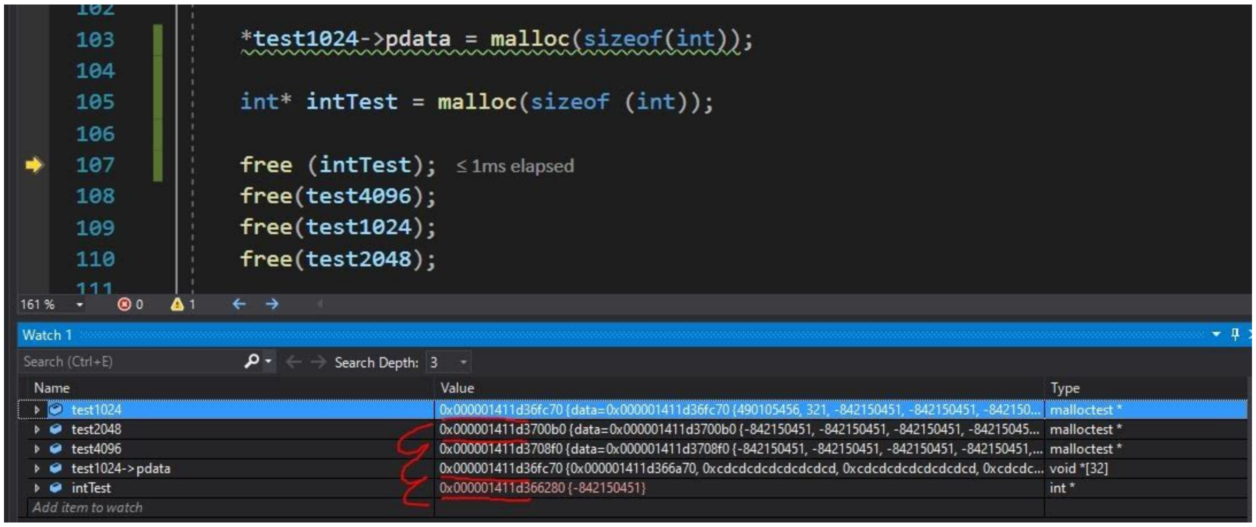
The second issue at hand pertains to the actual way allocated blocks of memory are organized by malloc. This ties in with the issue of the data structure, since determining the data structure used by malloc will provide more insight into how different allocated blocks of memory are allocated before and after one another. In essence, this is about figuring out how malloc determines which memory to use and which memory to not use and how it keeps these distinctions organized.



Name	Value	Type
test1024	0x000002a34937fc70 {data=0x000002a34937fc70 {1228368496, 675, -842150451, -842150451, -84215...	malloctest *
test2048	0x000002a3493800b0 {data=0x000002a3493800b0 {-842150451, -842150451, -842150451, -842150451...	malloctest *
test4096	0x000002a3493808f0 {data=0x000002a3493808f0 {-842150451, -842150451, -842150451, -842150451, ...	malloctest *
test1024->pdata	0x000002a34937fc70 {0x000002a349376a70, 0xcdcdcdcdcdcdcdcd, 0xcdcdcdcdcdcdcdcd, 0xcdcdc...	void *[32]

Looking at the allocated objects again (this time not casted so we can see the addresses), we can see that each one of the allocated objects has a different memory address from one another, which they obviously should since they are different objects. What's interesting is that they have the same address up until the last 5 characters (0x000002a3493xxxxx), after which they divert and change. What seems to be happening here is that the different objects are being put into the same blocks of memory when allocated with malloc, but are **organized into different locations within those blocks** (which are the last 5 characters in the addresses) based on some unknown parameter that we're searching for.

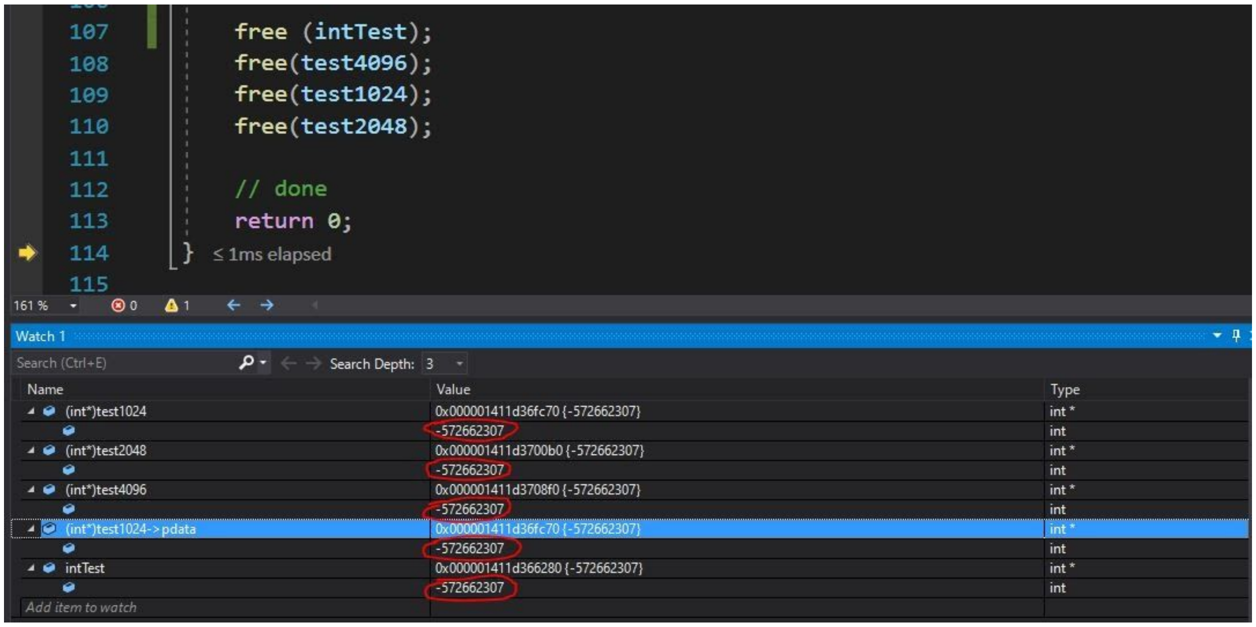
Resetting Memory with Free:
The issue revolves around the usage of free to return malloc'd memory to the heap once it we no longer need it to be allocated, specifically how this process is actually accomplished behind the scenes. Memory that is freed with free should in theory be processed in some way to indicate that the memory has been allocated (i.e, if there's nothing to free, do nothing; if there is something to free, we need some way of knowing that it's been allocated) and then return any memory determined to be allocated to the heap afterwards. What's happening in the debugging code is rather vague:



```
*test1024->pdata = malloc(sizeof(int));

int* intTest = malloc(sizeof (int));

free (intTest);  ≤ 1ms elapsed
free(test4096);
free(test1024);
free(test2048);
```

```
Watch 1
Search (Ctrl+E)    Search Depth: 3
Name              Value                                                                                              Type
test1024          0x000001411d36fc70 {data=0x000001411d36fc70 {490105456, 321, -842150451, -842150451, -842150...   malloctest *
test2048          0x000001411d3700b0 {data=0x000001411d3700b0 {-842150451, -842150451, -842150451, -84215045...     malloctest *
test4096          0x000001411d3708f0 {data=0x000001411d3708f0 {-842150451, -842150451, -842150451, -842150451,...   malloctest *
test1024->pdata   0x000001411d36fc70 {0x000001411d366a70, 0xcdcdcdcdcdcdcdcd, 0xcdcdcdcdcdcdcdcd, 0xcdcdc...       void *[32]
intTest           0x000001411d366280 {-842150451}                                                                    int *
Add item to watch
```

This snippet is stepped into after all of our objects and variables have been allocated with malloc. I've added a new int* variable to test a smaller amount of bytes declared with a sizeof (in this case an int, so it should be 4 bytes) to see if there's any differentiation between sizes. From this information, we can see that once again, all of our malloc'd data is allocated to the same memory address up until the final 5 characters (in this instance, 0x000001411d3xxxxx), regardless of their size.



```
    free (intTest);
    free(test4096);
    free(test1024);
    free(test2048);

    // done
    return 0;
} ≤ 1ms elapsed
```

Watch 1

Search Depth: 3

```
Name                      Value                                   Type
◢ (int*)test1024          0x000001411d36fc70 {-572662307}         int *
                          -572662307                              int
◢ (int*)test2048          0x000001411d3700b0 {-572662307}         int *
                          -572662307                              int
◢ (int*)test4096          0x000001411d3708f0 {-572662307}         int *
                          -572662307                              int
◢ (int*)test1024->pdata   0x000001411d36fc70 {-572662307}         int *
                          -572662307                              int
◢ intTest                 0x000001411d366280 {-572662307}         int *
                          -572662307                              int
Add item to watch
```

After calling Free on all of the data, what’s interesting is that the **addresses in memory remain completely unchanged from when they were allocated**. Rather, the value at each of these addresses is set to a seemingly random garbage number. However, on subsequent runs of the program, the value is **always** -572662307 when the data is freed, meaning this number isn’t random after all but rather indicates that all of these data are being set to the same value. What that value actually is is the issue at hand; we know that Free should be putting these values back into the heap, so that means they should be within the *same chunk of memory* somewhere once free’d back onto the heap.

Discovery/Replication of Issues

Each of these issues were discovered using the same process. The code used to discover the issues is based on code provided in class with a few additional allocations to test different sizes of memory and different data types. The core of the process revolves around using breakpoints and watches on the data being used, with different types of type casts being used to see how the values and addresses of the data are changing as the program calls malloc and free.

Within a C file, a union struct is declared (called “mallocTest”) which contains three members: an i32 array of size 32 called “data”; a ptr of size 32 called pdata; a struct containing a single i32 called “dummy”. These three members are used to see what happens to different types of data when malloc is called. This union is then declared with a typedef so data of its type can be created and used with malloc.

Once the union is created, we can start replicating the discovery of issues. In the test implementation, three mallocTest pointers are declared, each of which tests three different sizes of memory: test1024 is set to use malloc with a size of 1024; test2048 is set to use 2048; test4096 to a size of 4096. Following this, the pdata member of test1024 is set to malloc with a size of int in order to see how allocations change when members within a data structure are allocated. A separate test variable of type int* pointer called intTest is also declared and malloc'd to a sizeof int in order to see the difference between data structures and standard data types. After each of these malloc calls are finished, each malloc'd data is then placed in a call to Free in the reverse order in which they were malloc'd. Then the program ends.

For replication, run this program with a breakpoint at the first malloc statement (malloctest* test1024 = malloc (1024)). The program will then stop allowing usage of the Watch debug feature to see how each of the values and their addresses are changing as malloc and free are used.

To replicate issues related to the **Malloc's Data Structure** issue, create Watches for each "test" mallocTest with an (int*) cast in front of them to view the values of each data during the malloc process. To see the value in the pdata array from test1024, do the same in front of it. Then observe how the values at each address change as the program runs by stepping through the function with the Step In commands.

To replicate issues related to the **Organization of Allocated Memory** issue, create the same watches as the previous issue, except *do not cast them as an int**. Instead, leave them as is; this allows the addresses of each data to be viewed and see how they correlate and differ between one another.

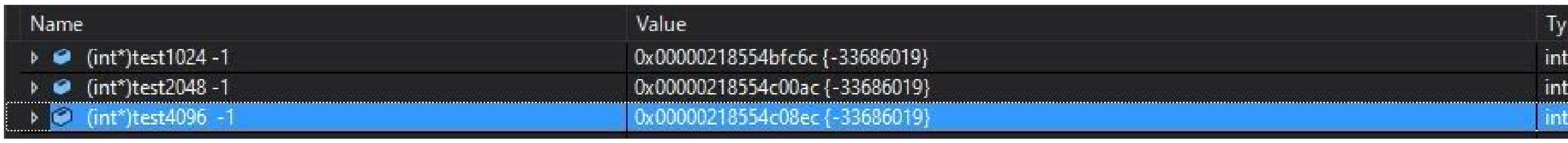
To replicate issues related to the **Resetting Memory with Free** issue, first use the same exact Watches from the previous issue and make sure to step through the entire program from the malloc statements all the way to the end of the free statements. Observe how the memory addresses are similar and different from each other. Then, repeat this same process, except casting each watch to an int* in order to see how the values at the addresses are changing. Step through the program again and observe how each value is set to the same value as each other once the Free functions are called.

Debugging Process

The following process was used to get to the core of each issue. Since all three of these issues are incredibly closely intertwined, only one process was needed to decipher each issue and ultimately determine how malloc and free work at a lower level. The process involves casting each Watch described in the discover process to an int* and **gradually subtracting from it to see what is happening at different segments in the malloc and free processes**. This process provides effectively describes how these functions operate, as it reveals patterns between different variables and types that lead to specific conclusions.

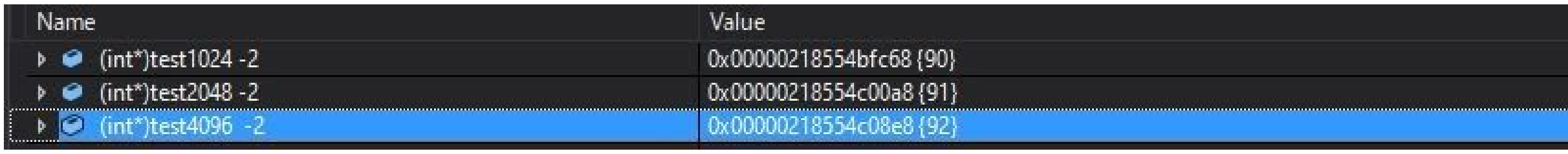
The Process:

First we take our data cast as int pointers in the Watches and subtract 1 from each of them. We can see that at the value -1, **each variable has the same value**, in this case -33686019. The number itself isn't necessarily important, but rather the fact that *each different malloc'd variable, regardless of size, has the same value at itself minus one*.



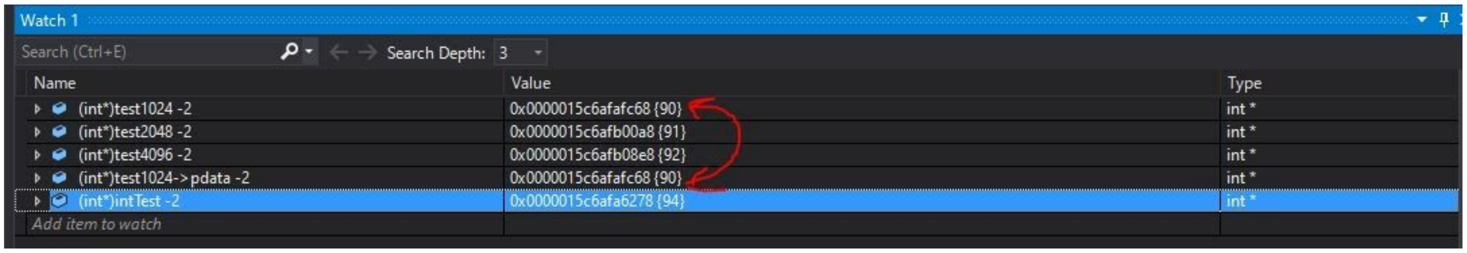
Name	Value	Ty
(int*)test1024 -1	0x00000218554bfc6c {-33686019}	int
(int*)test2048 -1	0x00000218554c00ac {-33686019}	int
(int*)test4096 -1	0x00000218554c08ec {-33686019}	int

Subtracting another interval of one from the Watches yields are first tangible result where the values are different between each declaration. Here we have the first malloc'd data (test1024) with a value of 90; the two subsequently malloc'd data have values of 91 (test2048) and 92 (test4096) respectively. What this indicates is that *malloc is counting something incrementally in order* as new data are allocated, meaning that somewhere, these steps are being stored and accounted for.



Name	Value
(int*)test1024 -2	0x00000218554bfc68 {90}
(int*)test2048 -2	0x00000218554c00a8 {91}
(int*)test4096 -2	0x00000218554c08e8 {92}

This counter is reinforced by the fact that our pdata in test1024 is also equal to 90, meaning malloc keeps track of these allocations **by memory block**; since pdata is part of the test1024 data, it resides in the same block of memory and is not counted as a separate allocated block.

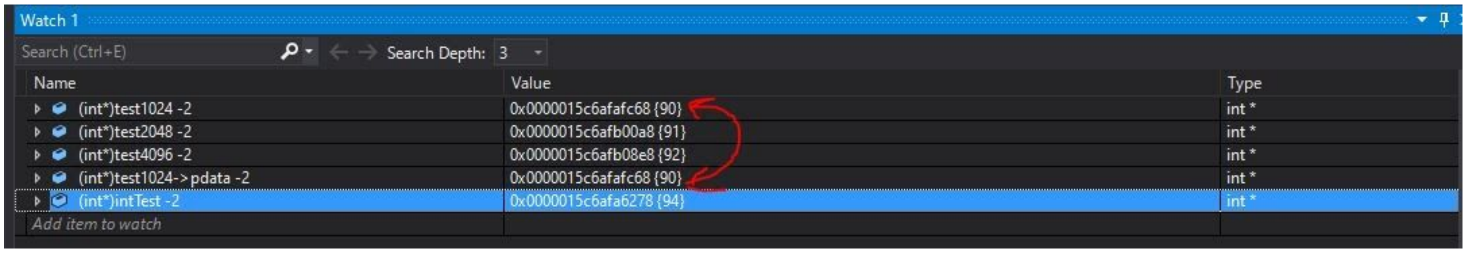


Watch 1

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
(int*)test1024 -2	0x0000015c6afafc68 {90}	int *
(int*)test2048 -2	0x0000015c6afb00a8 {91}	int *
(int*)test4096 -2	0x0000015c6afb08e8 {92}	int *
(int*)test1024->pdata -2	0x0000015c6afafc68 {90}	int *
(int*)intTest -2	0x0000015c6afa6278 {94}	int *
Add item to watch		

Going down to -3 yields another equal value between each of our malloc'd data, this time yielding a simple result of 0. On its own, this doesn't mean too much, but once we get down to -5, we'll start to see an interesting correlation between what happens at -5, -4, and -3 that reveals a key part of the issue at hand.

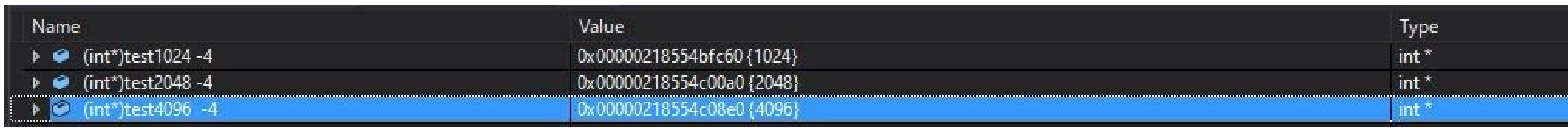


Watch 1

Search (Ctrl+E) Search Depth: 3

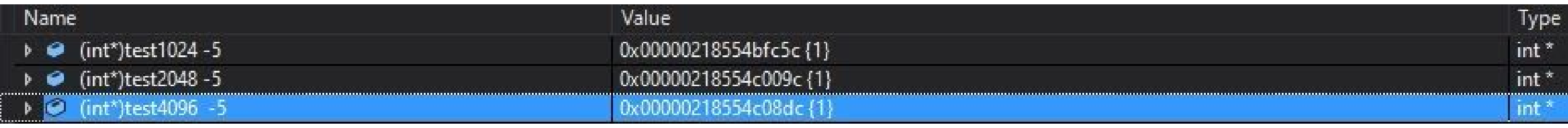
Name	Value	Type
(int*)test1024 -2	0x0000015c6afafc68 {90}	int *
(int*)test2048 -2	0x0000015c6afb00a8 {91}	int *
(int*)test4096 -2	0x0000015c6afb08e8 {92}	int *
(int*)test1024->pdata -2	0x0000015c6afafc68 {90}	int *
(int*)intTest -2	0x0000015c6afa6278 {94}	int *
Add item to watch		

At -4, we get a bit of an "aha!" moment: the values at -4 **directly correlate to the amount of memory, or "size" of the malloc operation** that we specified when calling the malloc statements! Now we're starting to get somewhere; it seems malloc is storing this sizing information somewhere along this process, meaning once its called, *the passed in parameter is being saved somewhere*.



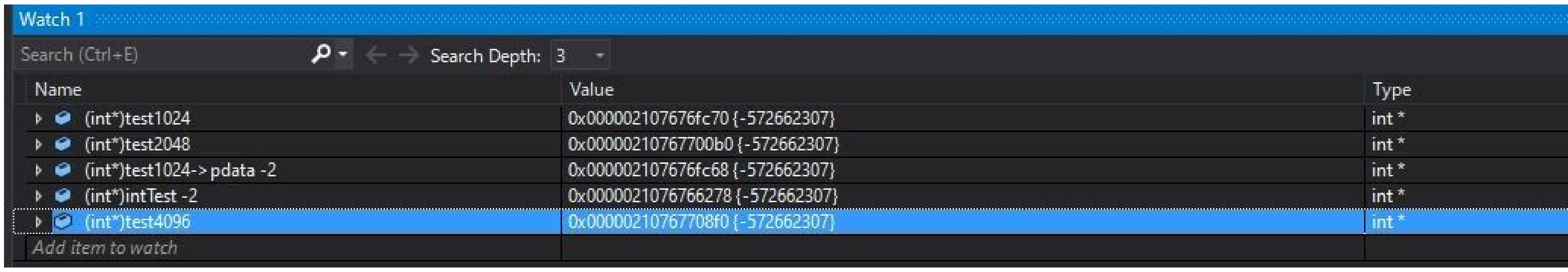
Name	Value	Type
(int*)test1024 -4	0x00000218554bfc60 {1024}	int *
(int*)test2048 -4	0x00000218554c00a0 {2048}	int *
(int*)test4096 -4	0x00000218554c08e0 {4096}	int *

I mentioned in -3 that we'd really understand what's happening with malloc's structure and organization once we hit -5, and here we are. At -5, all three of our data are set to the value of simply 1. This seems like it could mean nothing, but in the context of going from a value of 1 at -5 to the allocated size at -4 to 0 at -3, we can see that something tangible is happening here. What appears to be happening is that the **value of 1 at -5 is some sort of indicator or boolean to malloc that indicates if the memory being requested is free to allocate or not**, which is why our value is at 1. Once that check runs and we get to -4, we can assume malloc is then saying “the memory is free, how big should it be?” so it can allocate it; that’s why we get our **allocated size** at -4. In between -4 and -3 is where the allocation itself happens, meaning that **our memory is no longer free to use for other malloc operations**. This can be assumed to mean that our value of 1 thats keeping track of our free state is **no longer true and is set to 0 to indicate as such** so that no other malloc statements can grab the same chunk of memory. Then at -2, malloc adds this allocated block to a **list of allocated blocks and organizes them in order**



Name	Value	Type
(int*)test1024 -5	0x00000218554bfc5c {1}	int *
(int*)test2048 -5	0x00000218554c009c {1}	int *
(int*)test4096 -5	0x00000218554c08dc {1}	int *

of allocation, which explains the increment from 90 to 91 to 92 between our allocations. That covers the malloc side of things, but what about free? No matter what value is added or subtracted to the cast int* values in the watch list, each data remains the same value of -572662307 once they’re freed with free. What can be assumed from this is that once free is called, it returns **all freed data to the same place in the heap**, which in this case is indicated by that static value as well as the fact that each memory address stays the same between malloc and free (apart from the last 5 characters). If all of the data is being returned to the same place, then it can also be assumed that some sort of list is being kept in the heap of the freed memory blocks so that malloc can then use them again.

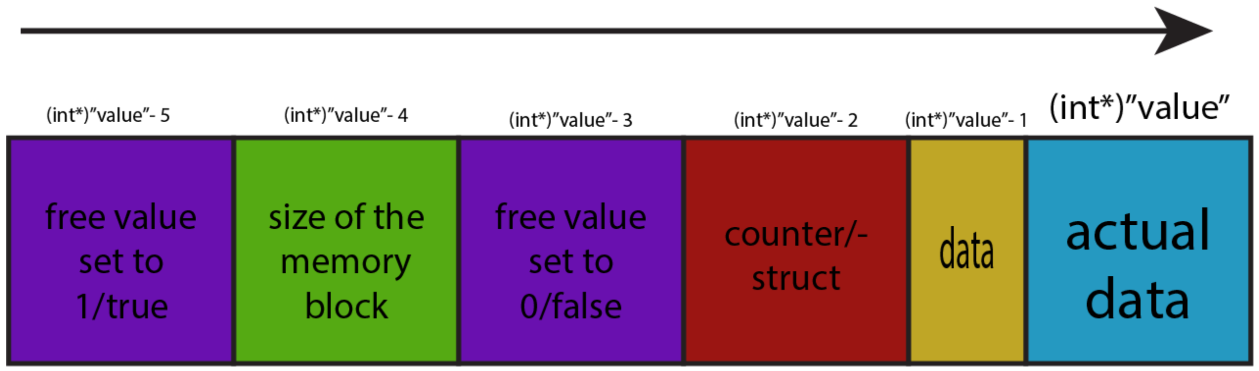


Watch 1

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
(int*)test1024	0x000002107676fc70 {-572662307}	int *
(int*)test2048	0x00000210767700b0 {-572662307}	int *
(int*)test1024->pdata -2	0x000002107676fc68 {-572662307}	int *
(int*)intTest -2	0x0000021076766278 {-572662307}	int *
(int*)test4096	0x00000210767708f0 {-572662307}	int *
Add item to watch		

The following diagram shows this on a visual reference. We can see the malloc process from left to right in the order described above:



Solution

Based on the findings from the discovery and debugging processes, I propose the following solutions to each issue:

Malloc's Data Structure/Organization of Allocated Memory:
The data structure that malloc uses is some form of a **linked list** that contains two member variables, one that stores the status of a memory block (free (1) or not free (0); what is seen at value -5 and value -3 when debugging) and one that simply stores the size to be allocated (the value seen at value -4 in the debugging process). In addition, there is another pointer variable of the same structure that points to the next memory block in the list and is used to organize and count up the number of allocated blocks (as seen in value -2 in debugging process). Using a linked list structure is important because it helps malloc determine which memory block to allocate next after its finished with its current allocation so it doesn't try and overwrite allocated memory from earlier in the list. It also determines the **organized location of each chunk of memory**; each struct in the list is a different location for the data of the memory to be stored at.

Essentially, what is happening with malloc is the following:

1. Malloc looks through a list of these structures containing the data described above. These structures are **the memory blocks.**
2. Using the specified size in the malloc call, malloc then finds this block of memory in the heap and returns the pointer to it to be used.
3. Then malloc modifies the returned block's struct to **record it as no longer being free** (0) and **store its size** (1024, 4096, 4, etc) so that free can later see this information.

To implement this, using a structure like the following would be effective. This is effective because it wraps all of the data malloc needs into one easily accessible struct:

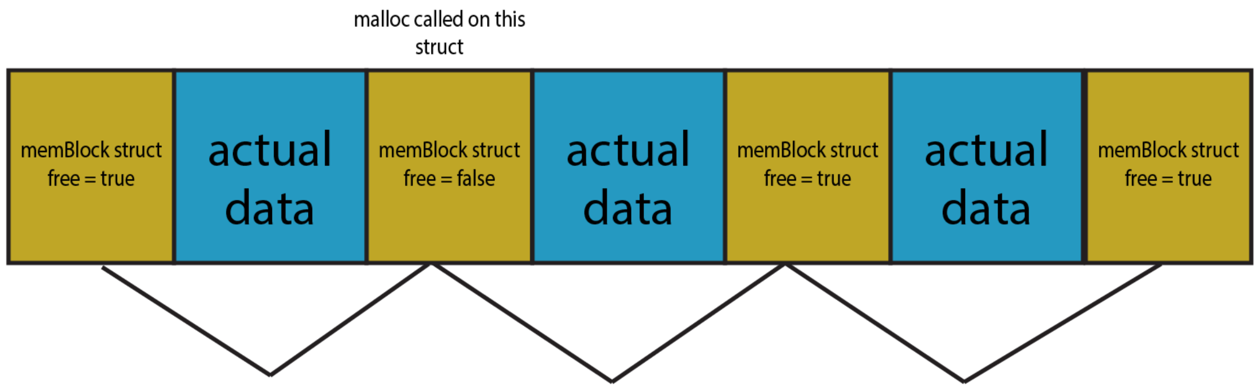
```
// A block of memory in the form of a linked list
struct memBlock
{
        // Used to track if this memory block is free for malloc to use; true if free, otherwise
        false
        bool free;

        // The size of this block. Equal to the passed in size from the malloc call
        size_t blockSize;
        // Pointer to the next block of memory for malloc to use once this block is allocated
        memBlock *nextBlock
};
```

Resetting Memory with Free:
Now that we know the structure and organization of malloc, proposing a solution to how free works is simple. When malloc reads through the list of memBlock structs, it sets the free bool of each allocated block to **false**, meaning that unless reset to true, no more memory can be allocated using these structs in the list. When we call free on a malloc'd data, its corresponding memBlock's free bool is once again set to **true**, meaning data can be allocated to it again if need be. Once the free bool is set to true again, the size member of the memBlock is used to see how large the block is and then returns it to the heap, which is essentially in and of itself just another linked list of memBlocks with their free bool set to true.

This sort of easily implementation can be done with a function such as the psuedo one below:

```
void CustomFree (memBlock* blockToFree)
{
        // check how large the size of the block is; if its less than the size of our memory we can
        return it to the heap
        if (blockToFree->size <= sizeOfMemory)
        {
                // set the block back to free so it can be used again
                blockToFree->free = true;
        }

}
```

The following diagram shows the linked list structure between memory blocks. Essentially, the linked list contains data about the actual data to be used (its sized, if its free or not, etc) and malloc uses this to reserve chunks of the actual data to be used with this descriptive data


malloc called on this struct
memBlock struct free = true
actual data
memBlock struct free = false
actual data
memBlock struct free = true
actual data
memBlock struct free = true