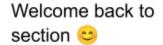
CSE 351 Welcome back to section **Section 9 Worksheet: Memory Allocation**



To understand the computer, you must become the computer.

For this problem, you will use a heap simulator to answer questions about heap memory allocation in C using malloc and free. First, access the heap simulator, either by going to the Lab 5 page on the course website, or by following the link:

https://courses.cs.washington.edu/courses/cse351/heapsim/

Read through the information on the page so you can understand how the simulator works. If you check "simulation mode" it will walk you through each step in the process of allocating a block of memory on the heap. Notice that each header (and footer) contains three fields. They correspond to:

```
block size: preceding block allocated?: this block allocated?
```

where 1 means "allocated" and 0 means "unallocated." Remember that block size **includes** the size of the headers and footers. In Lab 5, all of this will be stored within 8 bytes using the last few bits of the size (see the spec for more information).

Now, answer the following questions:

1. Starting with an empty heap (you can empty the heap by refreshing the page), "Execute" the following code:

```
void* ptr1 = malloc(30);
void* ptr2 = malloc(40);
void* ptr3 = malloc(70);
```

- a. What pointer is returned if we execute another malloc now?
- b. Which block(s) could you free that would cause fragmentation in the heap?
- c. Which block(s) could you free that would cause coalescing to occur?
- d. Suppose free (ptr2) is run immediately after malloc (70). Draw a diagram of what the free list looks like afterwards.
- e. What is the maximum size **payload** that we could allocate (i.e. the argument to malloc) such that we are returned a pointer to the address 48 (0x30)

Lab 5

In Lab 5, we will implement a memory management system that uses an explicit free list. Each block has pointers to the next and previous blocks. This is the block struct we will use:

```
struct block_info {
    // Size of the block (in the high bits) and tags for whether the
    // block and its predecessor in memory are in use.
    size_t size_and_tags;
    struct block_info* next;
    struct block_info* prev;
};
typedef struct block info block info;
```

Macros & Static Inline Functions:

We provide you with a lot of resources (helper functions, macros and static inline functions, read fully through mm.c for all of them). Here are some of them:

UNSCALED_POINTER_ADD(p,x) Add without using "pointer arithmetic", returns void*
UNSCALED_POINTER_SUB(p,x) Subtract without using "pointer arithmetic", returns void*
SIZE(x) Extracts the size from the size_and_tags field
MIN_BLOCK_SIZE The size of the smallest block that is safe to allocate
TAG_USED Mask for the used tag (1 = 0b1)
TAG_PRECEDING_USED Mask for the preceding used tag (2 = 0b10)
WORD_SIZE Size of a word on this architecture

- 2. Given that **void* ptr** is a pointer to the *beginning* of a free block.
 - a. Give a C expression that sets the *previous* block's *next pointer* to *ptr's* next block, as would be done if we were removing **ptr** from the free list.
- 3. Assume **void* ptr** is now a pointer to the *payload* of an allocated block. Using above macros and functions to provide C expressions that get the following in terms of **ptr**:
 - a. Size of allocated block

```
size_t size_curr_blk = ____;
```

b. Set **TAG_PRECEDING_USED** of following block to **True** (can use answer from part a)

```
block_info* flw_blk = ____;
flw blk->size and tags = ____;
```

4. Implement the following functions. Try using bitwise operators to access the tags in size_and_tags.

```
// Bit masks used to retrieve tags from size and tags.
#define TAG USED 1
#define TAG PRECEDING USED 2
// SIZE(ptr->size and tags) extracts size of 'size and tags' field.
static inline size t SIZE(size t x) {return ((x) & ~(ALIGNMENT -
1));}
// Copies the tags (TAG PRECEDING USED and TAG USED) from
// block to copy to original block. Leaves the size of
// original block unchanged.
void copy tags(block info* original block, block info*
block to copy) {
     size_t copy_used = ____;
     size_t copy_preceding_used = _____;
     original block->size and tags = ;
}
block info* FREE LIST HEAD;
// Removes a block from the free list. Note, this is not the full
implementation of remove free block, just the part where we are
reworking the links of the free list.
void remove free block(block info* free block) {
    block info* next free, *prev_free;
     next free = free block->next;
     prev free = free block->prev;
     // Your implementation goes here...
```

}	