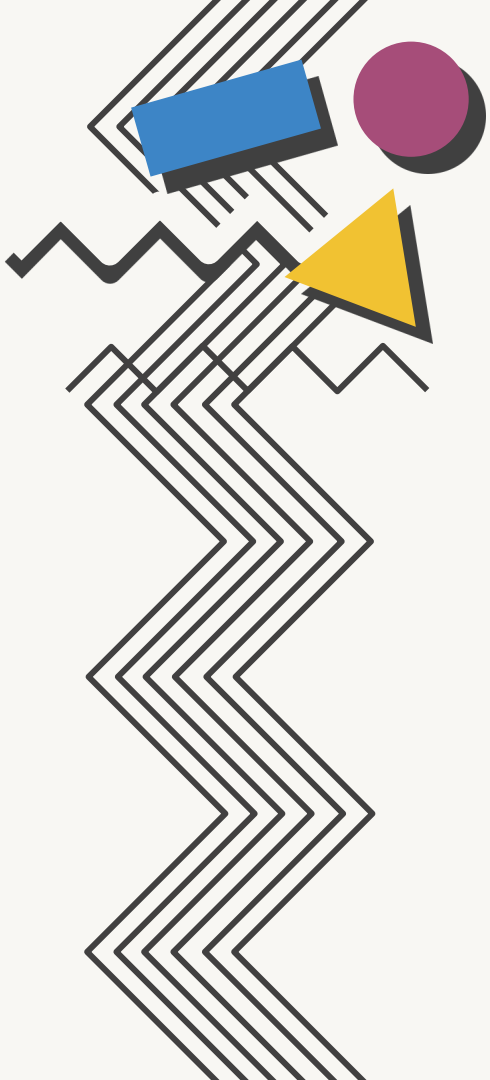
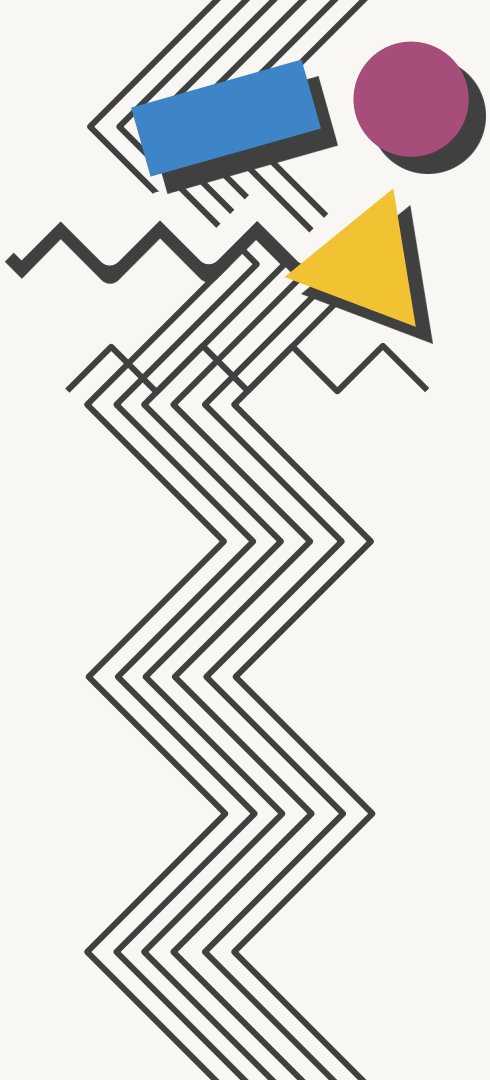
CSE 351 Section 9

Memory Allocation and Lab 5



Administrivia

- **Homework 24**
  - Due Friday, March 4
- **Lab 5**
  - Due Friday, March 11



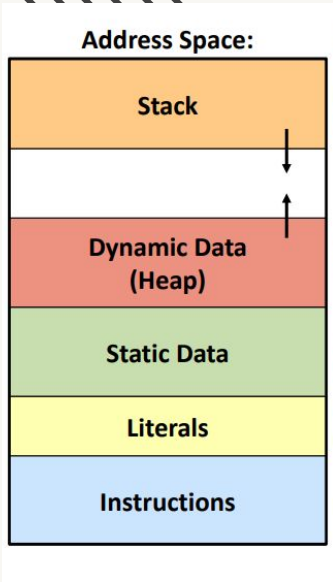
Dynamically Allocated Memory

The Heap

- Dynamic memory is memory that is “requested” at run-time. Dynamic data is stored in the *heap*.
  - Memory allocated dynamically by the programmer (malloc)
  - Must be explicitly freed (free)
    - Free it as soon as you don't need it!
  - Distinct from normal variables, which are always on the stack

Use cases:

- Variable-length data, like arrays or strings (think: Java's ArrayList)
- Long-lived data passed between functions



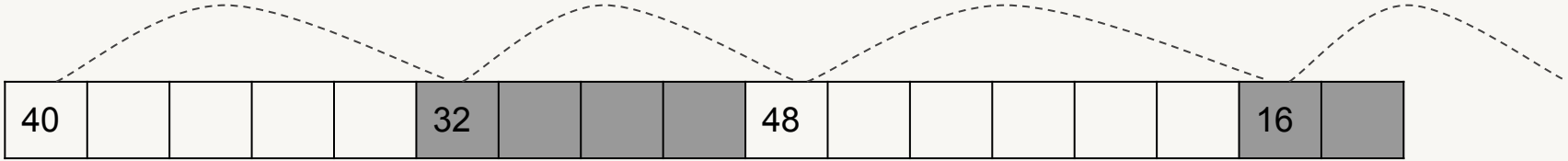
`malloc()` and `free()`

- Provided to you by the C standard library in `<stdlib.h>`
- How to use `malloc()`:
  - Takes a `size_t` representing the number of bytes requested
  - Returns a `void*` pointing to the start of the ***payload*** or `NULL` if there was an error
- How to use `free()`:
  - Pass free() a pointer to a block received from malloc() to deallocate its space on the heap
    - Be careful - don't free the same block twice!

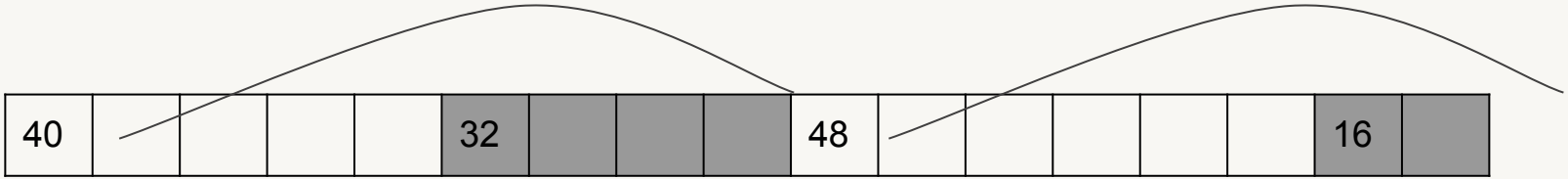
```
int* array = (int*) malloc(10 * sizeof(int))
...
free(array);
```

Implicit vs. Explicit Free List

Implicit: Using sizes to traverse blocks, checking to see if each block is allocated



Explicit: Using pointers to create linked list of free blocks (oft. doubly linked)



Comparison: free-lists

Implicit

- Find the next block via incrementing by the current block's length
- It may or may not be free
  - Potentially lots of extra blocks in the way!
- Requires only knowledge of each block's size

Explicit

- Find the next block by following a pointer
- All blocks in the free-list are guaranteed to be *free*
- Requires space in each free block to store pointers to the blocks before/after it

Reminder: Implicit/explicit free-lists are separate from implicit and explicit allocators.

For the remainder of this section, we'll be looking at explicit free-lists.

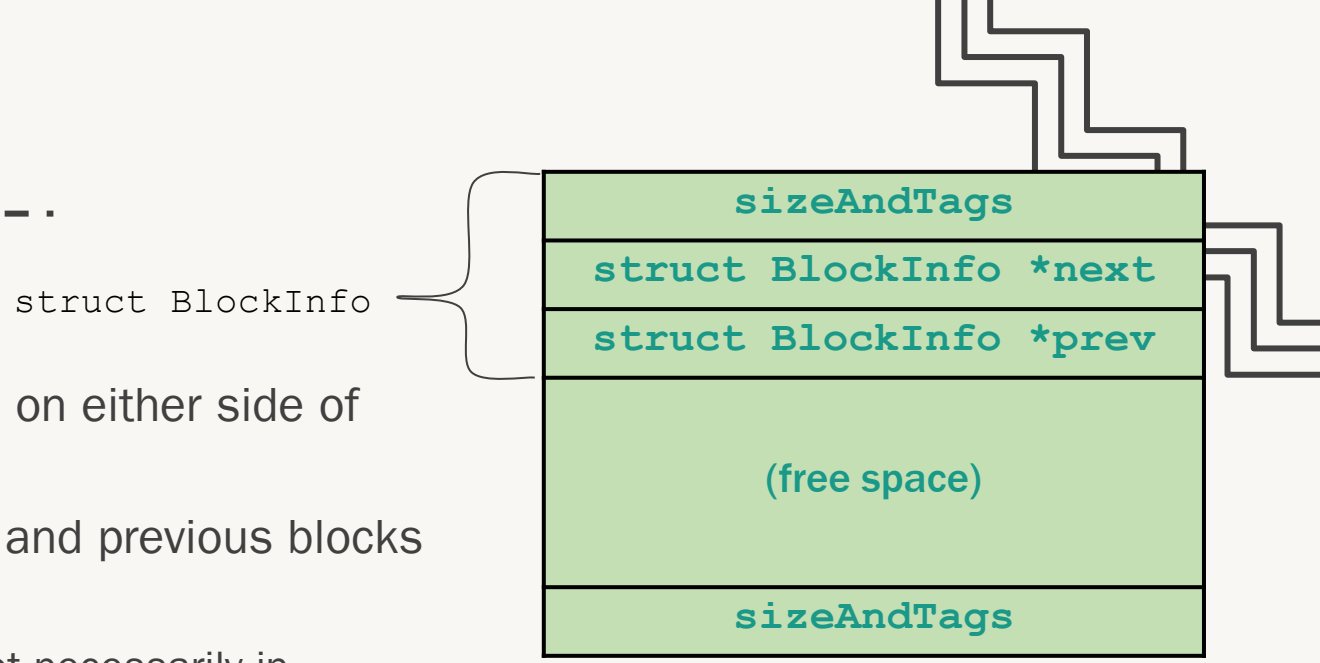
Block Header Format

- Every block has a 8-byte (64-bit) header, and needs to indicate its size, if it is used, and if the prev block is used
- Size must be 8-aligned, so can use lowest 3 bits for tags
  - LSB is set if the block is currently used (not in the free list)
  - Next bit set if the block preceding it (in memory) is used
  - The third bit is not used
  - Be careful with masking!
- The upper 61 bits store the size of the block
- Entire 64-bit value is called “sizeAndTags” in Lab 5

Free Blocks

A free block has:

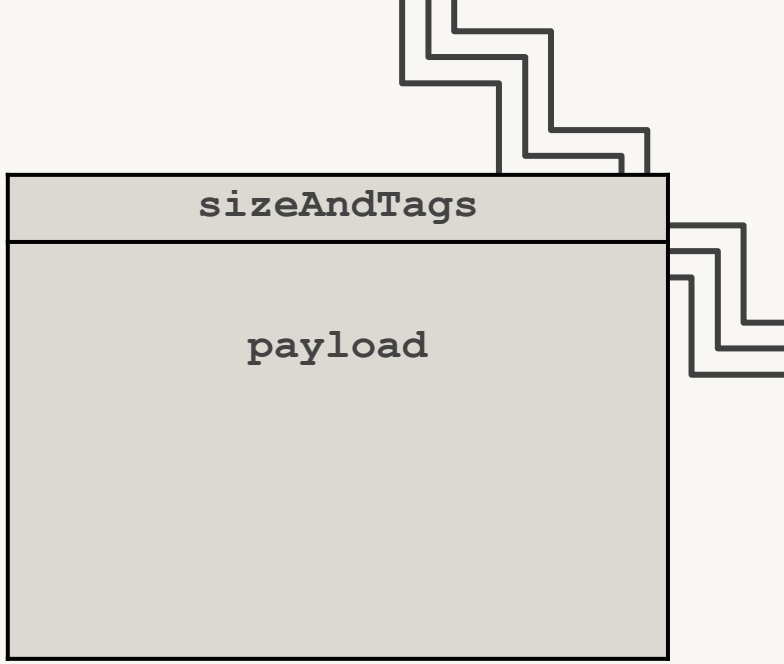
- A sizeAndTags value on either side of the free space.
- Pointers to the next and previous blocks in the list.
  - The blocks are not necessarily in address order, so the pointers can point to blocks anywhere in the heap
- Each free block is a BlockInfo struct followed by free space and the boundary tag (footer)




```
struct BlockInfo {
  size_t sizeAndTags;
  struct BlockInfo *next;
  struct BlockInfo *prev;
};
```

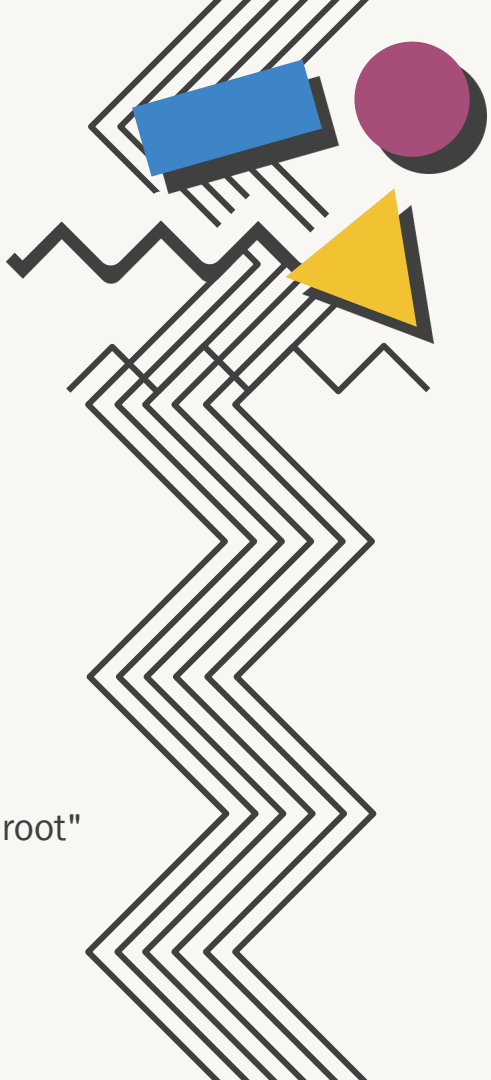
Used Blocks

- Used blocks only have a sizeAndTags, followed by the payload
- In Lab 5, used blocks have no footer!
- The payload is the actual block of memory returned to a user program that invokes malloc()



Key Steps

- Allocation
  - Search for a block of sufficient size
  - If sufficient space for another block, split into two
  - Remove selected block from free-list
  - Mark the allocated block as allocated
  - Return a pointer to the payload
- Deallocation (freeing)
  - Mark as free
  - Coalesce with adjacent blocks if possible
  - Add new larger block to free-list
    - If using LIFO insertion policy, this free block becomes the new "root"



Walkthrough of Example Heap

Initial Heap

Note FREE_LIST_HEAD always points to the first block in the free list

Walkthrough of Example Heap

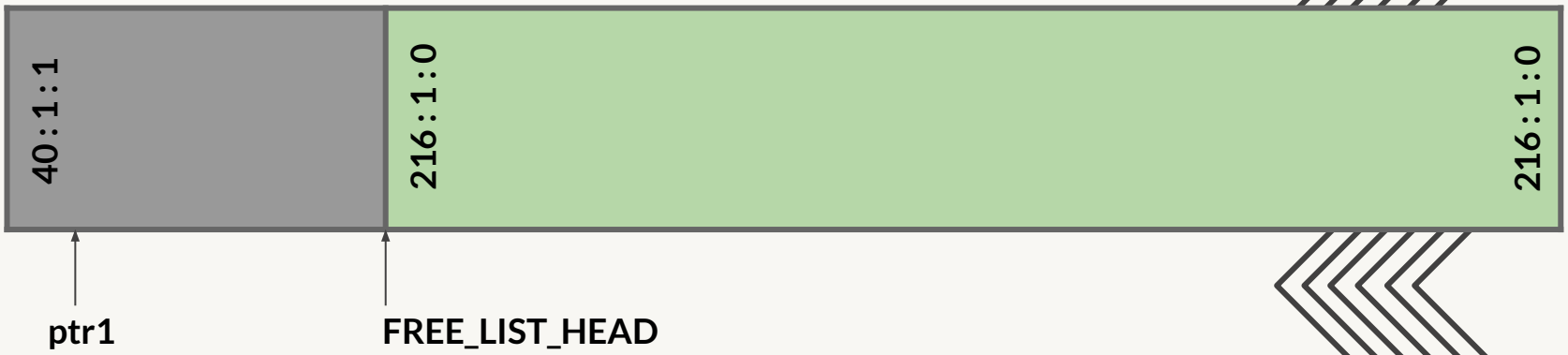
void *ptr1 = malloc(32);

- **Need to search free list to find a block big enough for 40 (32 + header) bytes**

Walkthrough of Example Heap

void *ptr1 = malloc(32);

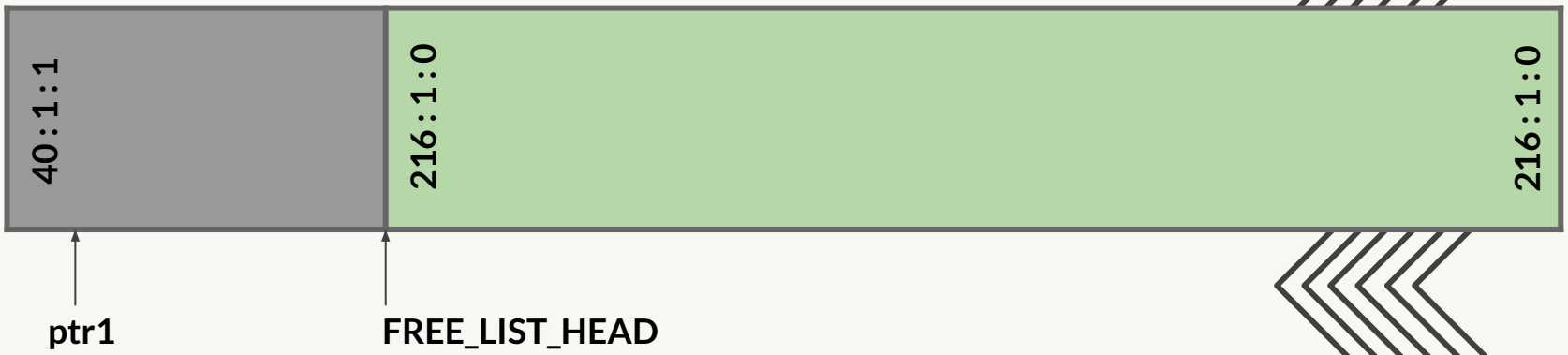
- **Note that ptr1 points to the start of the payload, NOT the start of the block**
- **The initially 256 byte free block is split to maximize memory usage!**



Walkthrough of Example Heap

void *ptr2 = malloc(16);

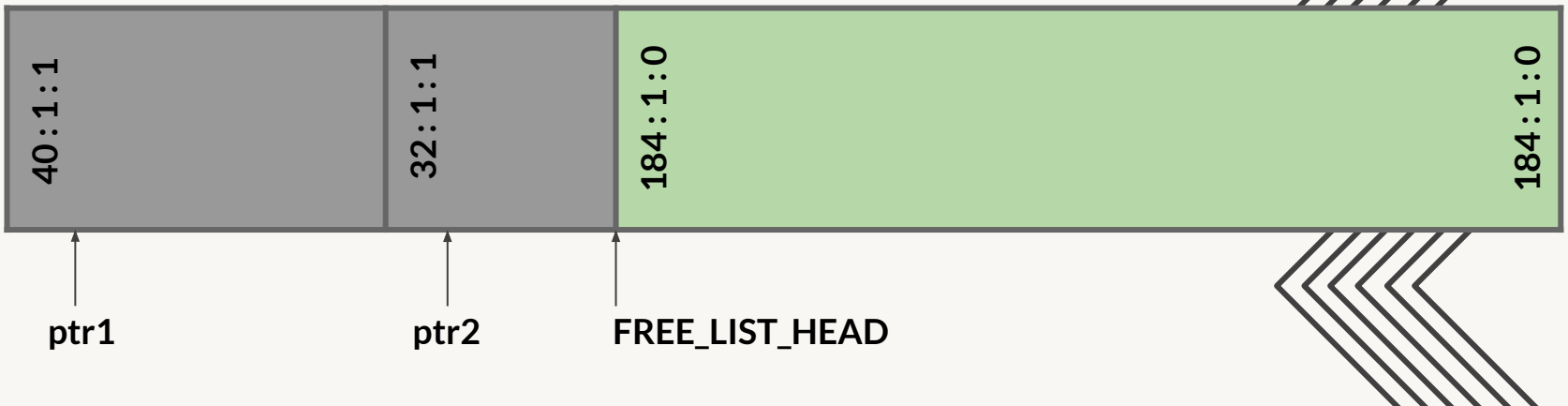
- **Only need a block of 24 (16 + header) bytes, but what if we needed to free it later… think about what the minimum block size needs to be**



Walkthrough of Example Heap

void *ptr2 = malloc(16);

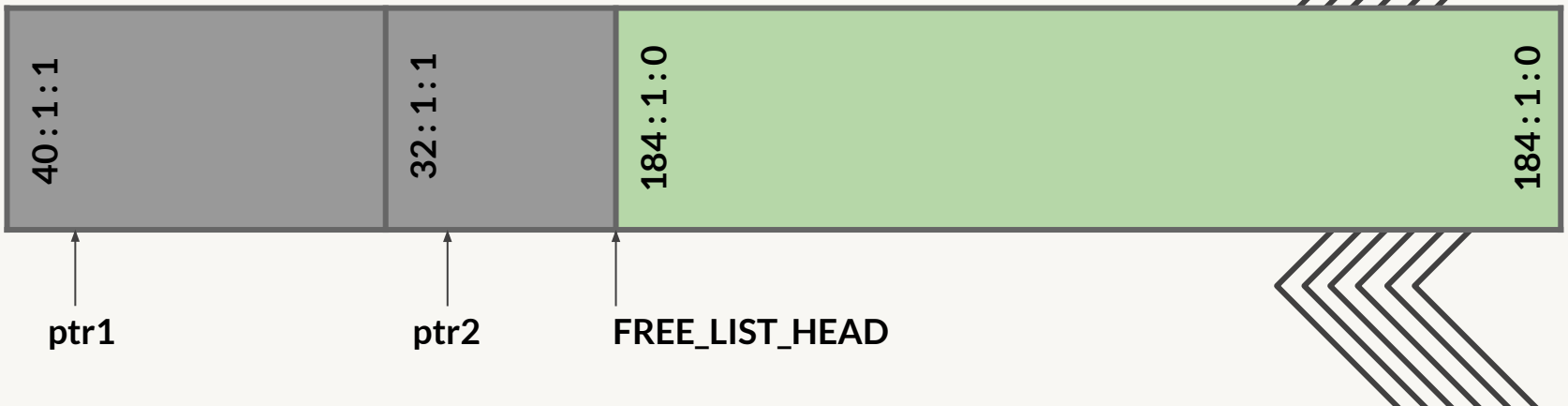
- **Need at least 32 bytes to create a free block, meaning we must allocate at least this much for a used block!**



Walkthrough of Example Heap

void *ptr3 = malloc(24);

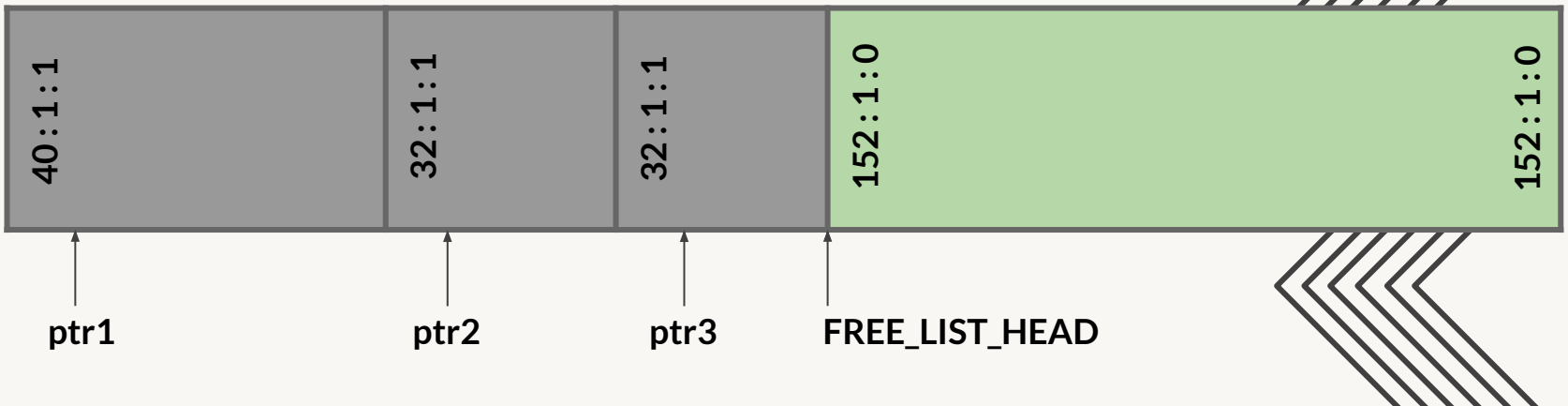
- **Same procedure as before**



Walkthrough of Example Heap

void *ptr3 = malloc(24);

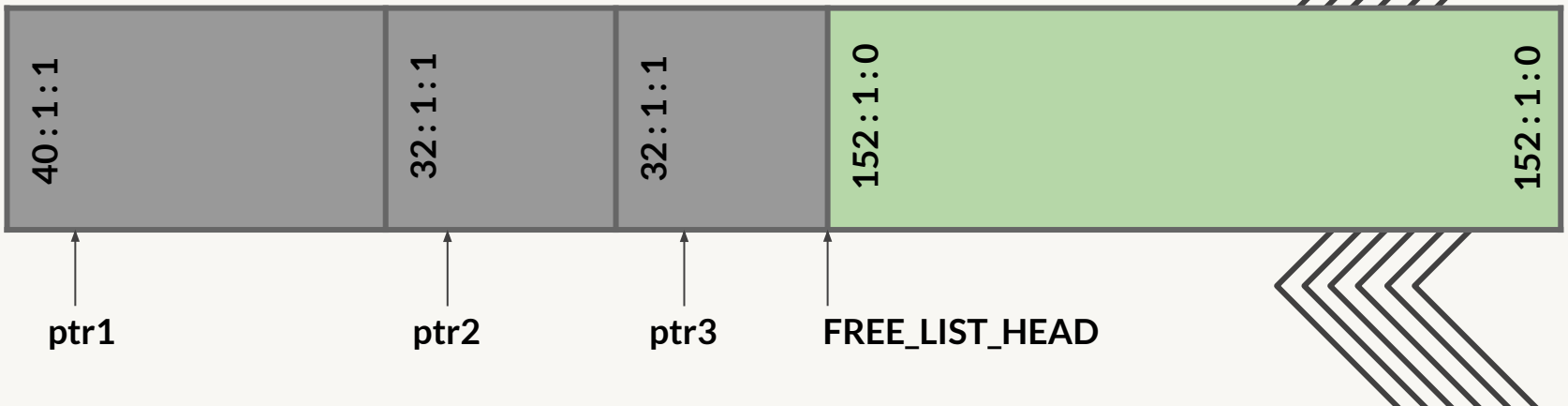
- **Same procedure as before**



Walkthrough of Example Heap

free(ptr2);

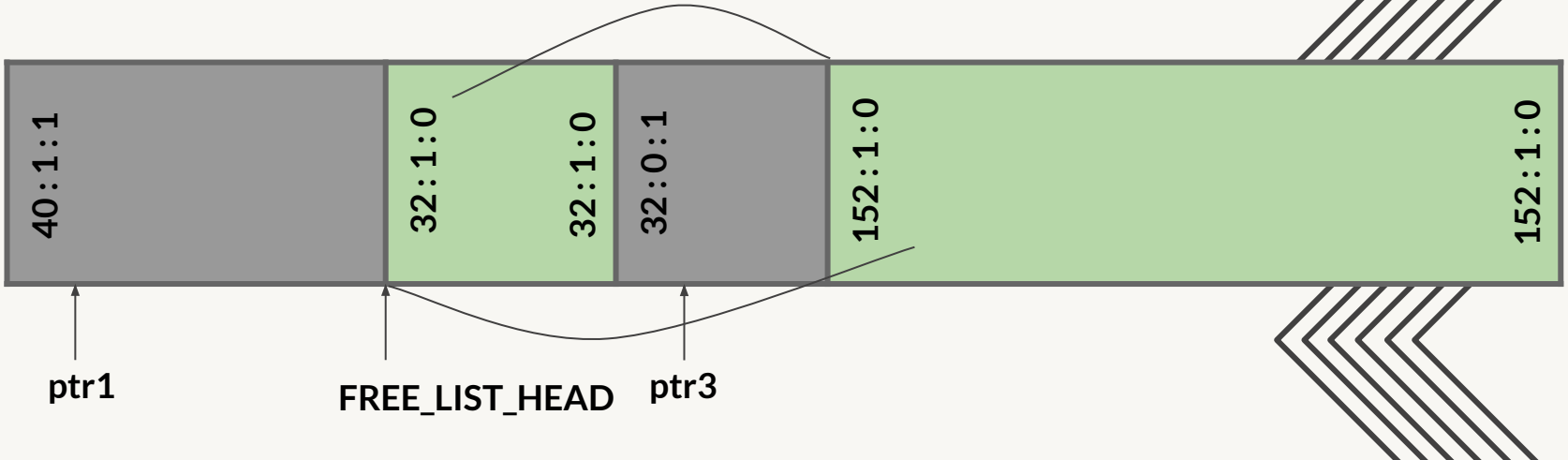
- **Now we need to free a block!**



Walkthrough of Example Heap

free(ptr2);

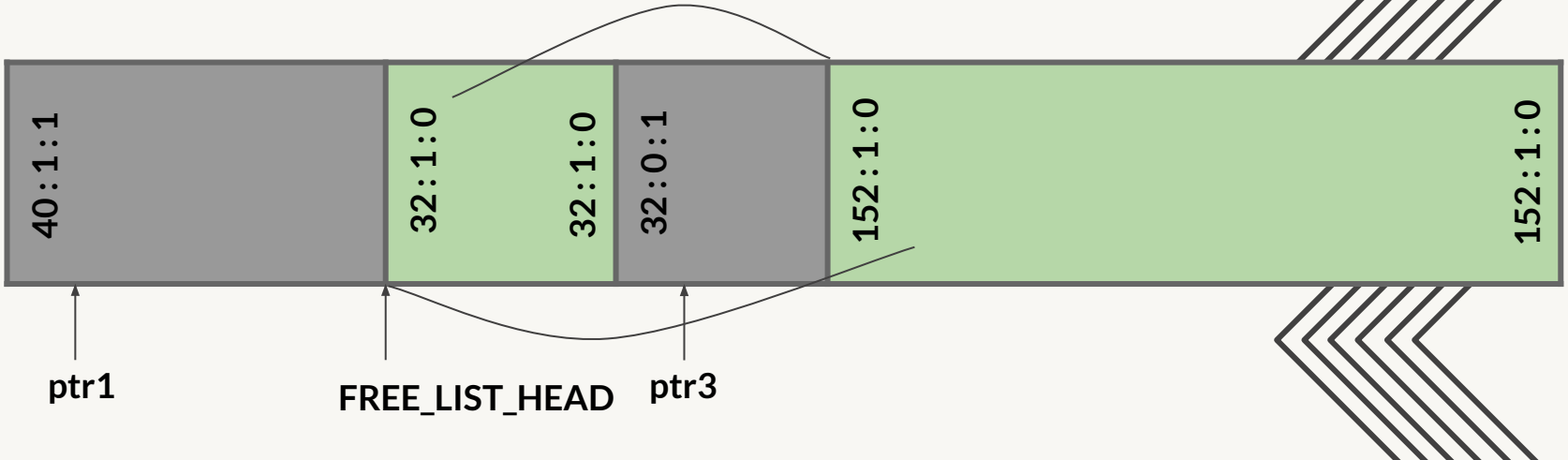
- **Need to insert block allocated for ptr2 into the free list (and update tags!)**
- **Which tags get updated?**



Walkthrough of Example Heap

free(ptr3);

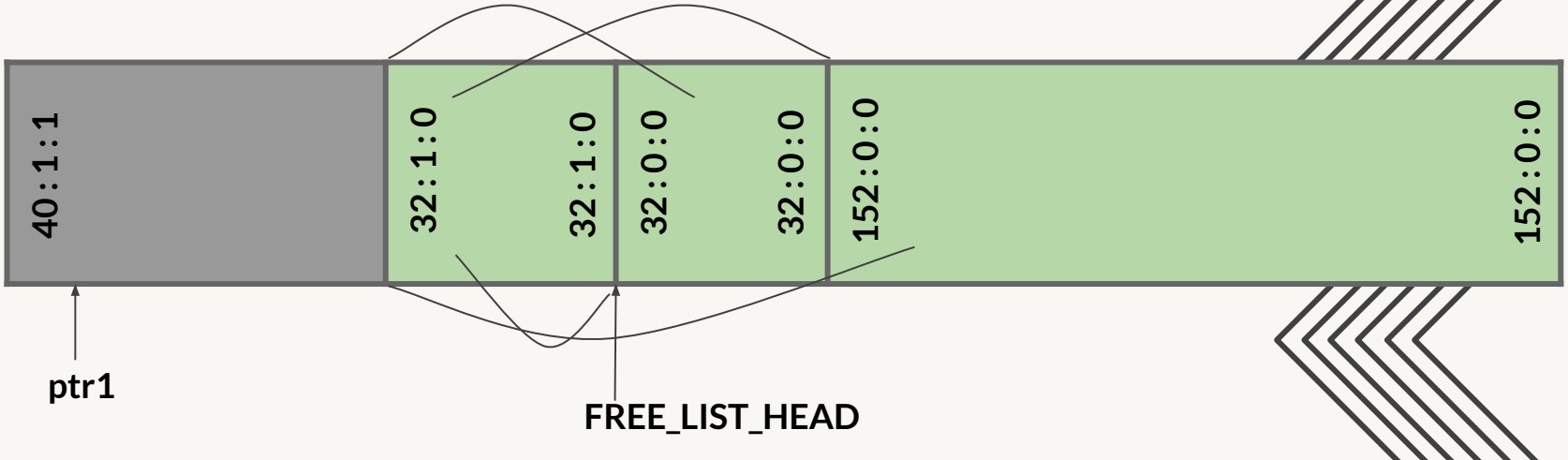
- **Same thing as before, except now the pointers get really messy...**



Walkthrough of Example Heap

free(ptr3);

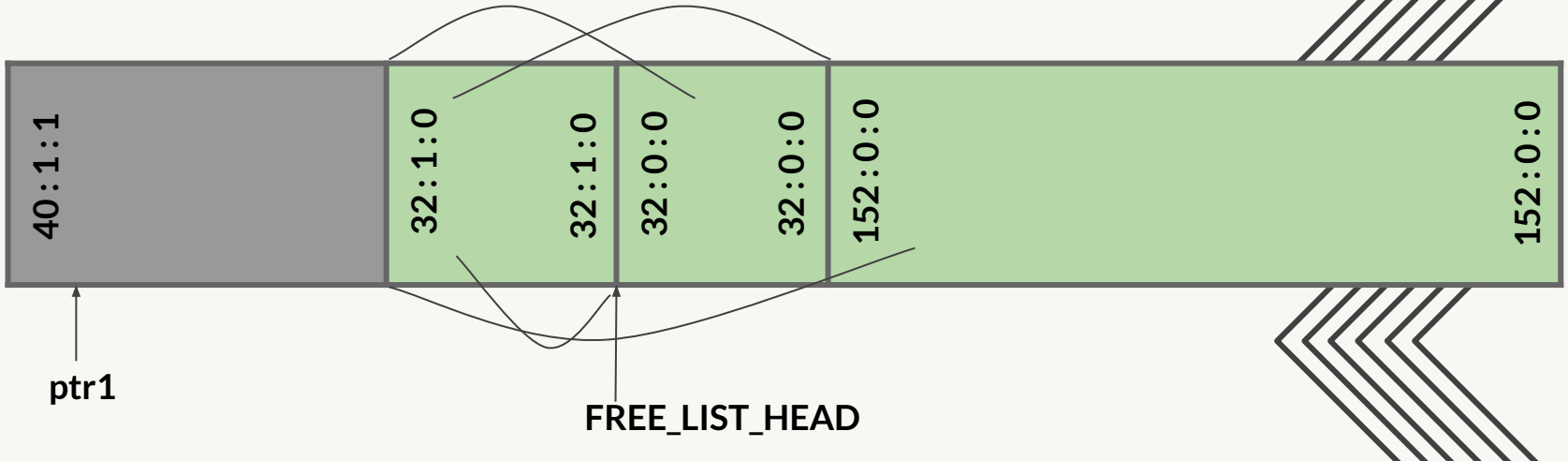
- **Same thing as before, except now the pointers get really messy…**
  - **next pointers are the ones higher up in the diagram, prev lower down…**



Walkthrough of Example Heap

free(ptr3);

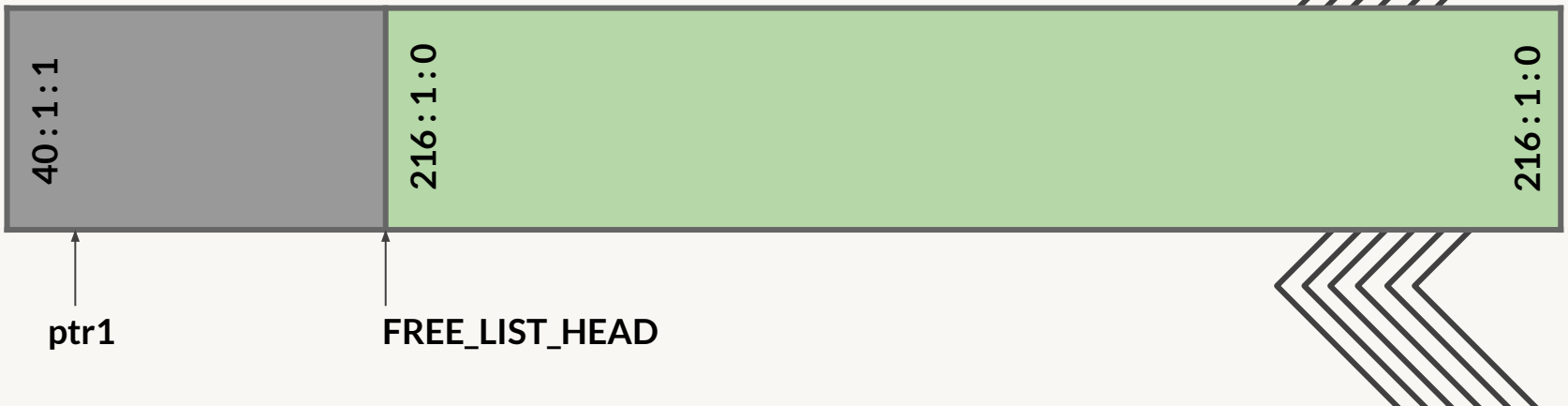
- **Good enough? What happens if user calls malloc(200)?**



Walkthrough of Example Heap

free(ptr3);

- **Coalesce neighboring free blocks into one large free block!**
- **Allows for larger future mallocs, can still split later for smaller chunks**



Heap Simulator

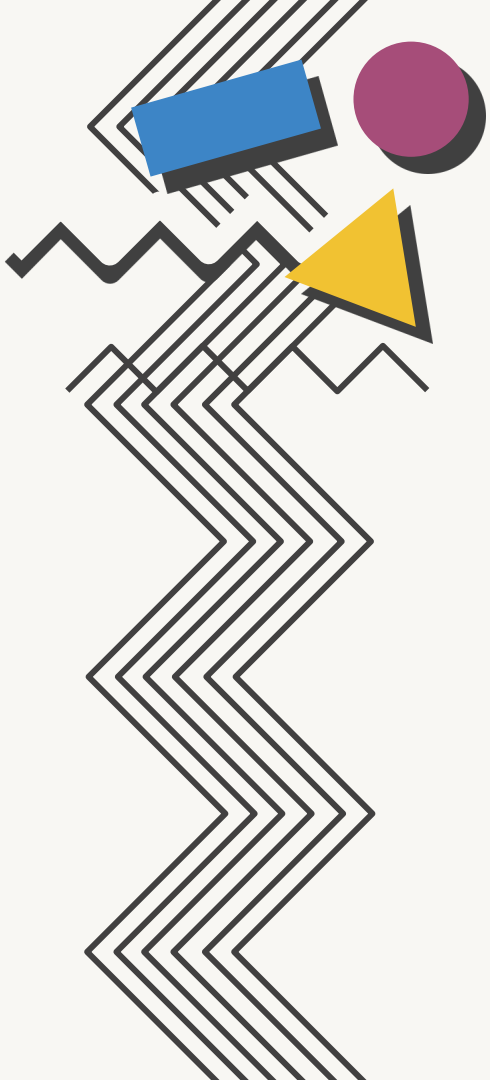
https://courses.cs.washington.edu/courses/cse351/heapsim/

Worksheet

Worksheet Problem 1

Starting with an empty heap (you can empty the heap by refreshing the page), “Execute” the following code:

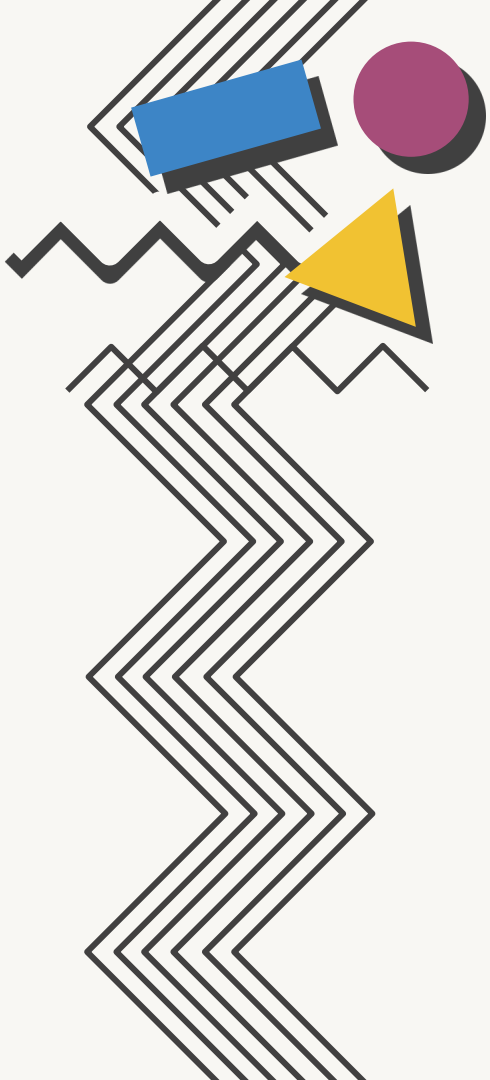
```
void *ptr1 = malloc(30);
void *ptr2 = malloc(40);
void *ptr3 = malloc(70);
```



Getting Started Lab 5

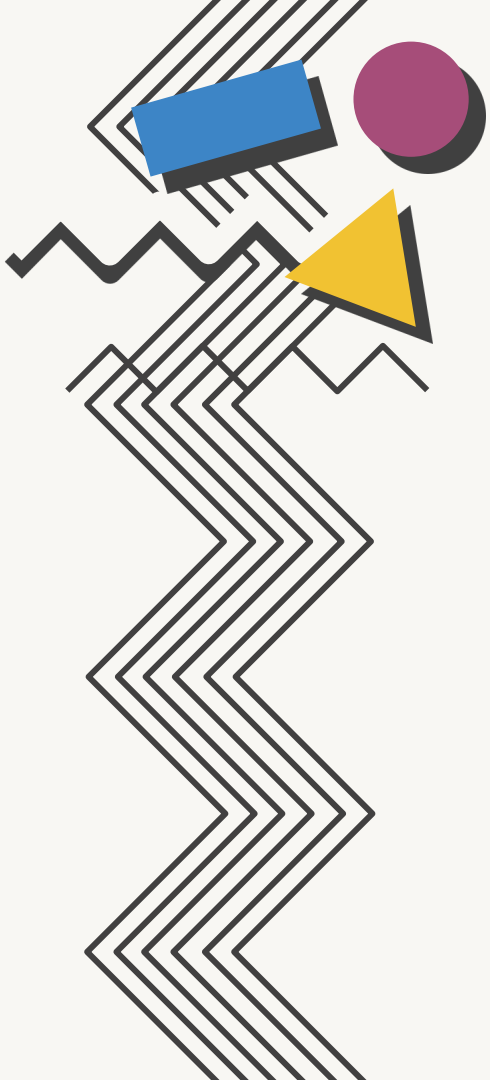
Lab 5

- You get to implement malloc() and free()!
- Less overwhelming than it may sound, we give you many functions already including:
  - `searchFreeList()`
  - `insertFreeBlock()`
  - `removeFreeBlock()`
  - `coalesceFreeBlock()`
  - `requestMoreSpace()`
  - see spec/starter code for full list!



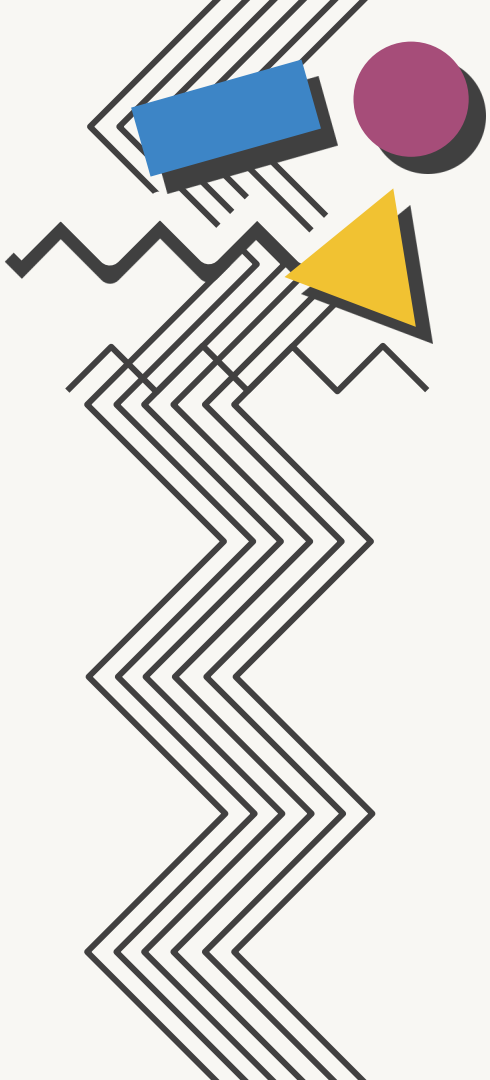
Implementing `malloc()`

- Figure out how big a block you need
- Call `searchFreeList()` to get a free block that is large enough
  - NOTE: If you request 16 bytes, it might give you a block that is 500 bytes
- Remove that block from the list
  - Might have to splice into a smaller/bigger chunk (see NOTE above)
- Update size + tags appropriately (do neighbor blocks need updating?)
- Return a pointer to the payload of that block



Implementing `free()`

- Remember, the pointer you are passed is to the payload!
- Convert the given used block into a free block
- Insert it into the free list
- Update size + tags appropriately (do neighbor blocks need updating?)
- Coalesce if necessary by calling `coalesceFreeBlock()`



C Macros

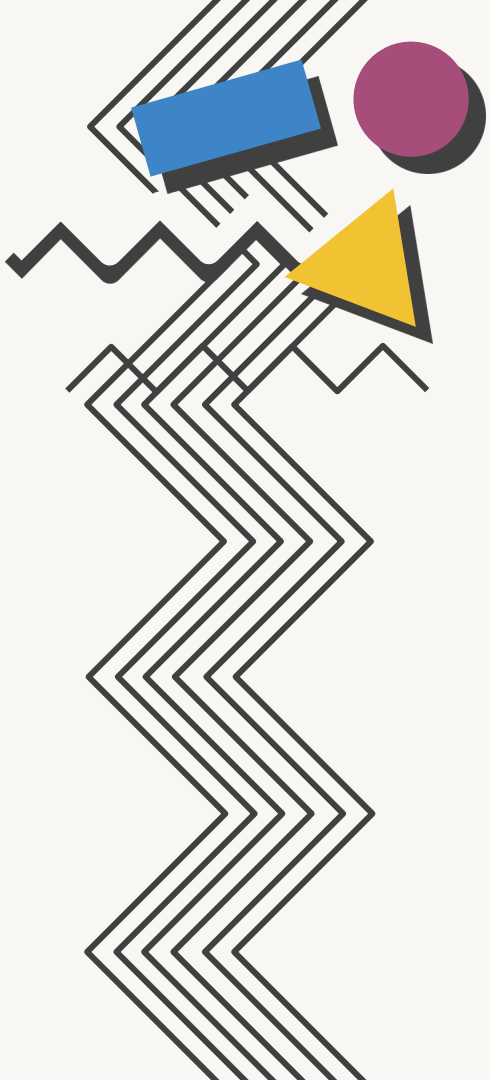
Pre-compile time “find and replace” your code text

Defining constants:

- `#define NUM_ENTRIES 100`
  - **OK**

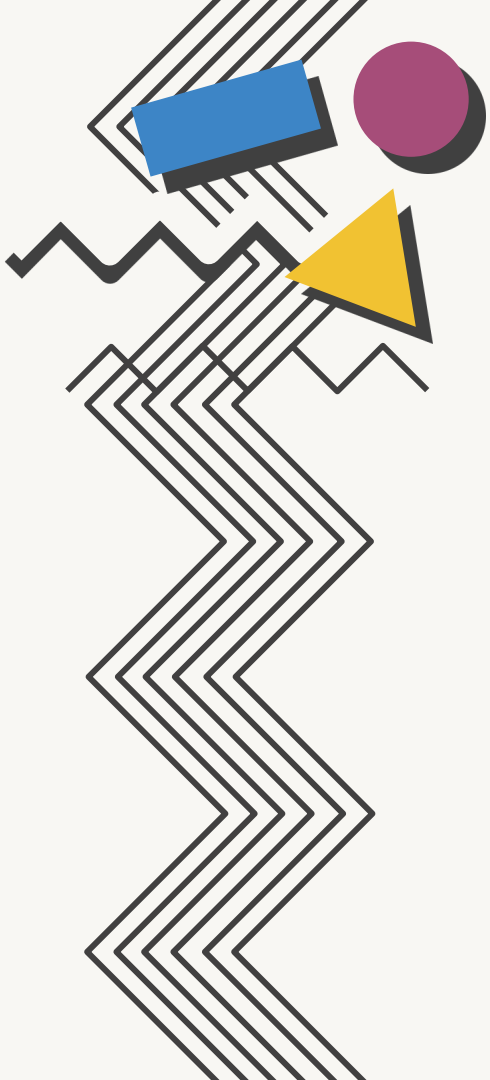
Defining simple operations:

- `#define twice(x) 2*x`
  - ***Not* OK**, `twice(x+1)` becomes `2*x+1` because preprocessor uses naive find and replace
- `#define twice(x) (2*(x))`
  - **OK**, now `twice(x+1)` becomes `2*(x+1)`
  - Always wrap in parentheses!



Why even use Macros?

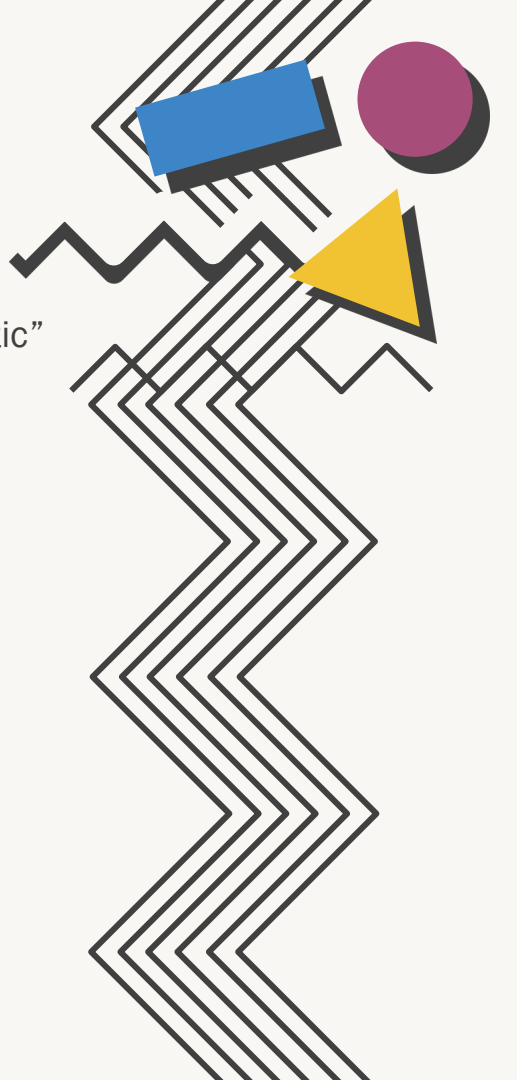
- Why macros?
  - Create more readable/reusable code for constants
  - “Faster” than function calls
  - In malloc: Quick access to header information (payload size, used tag, etc.)
- Drawbacks
  - Less expressive than functions
  - Arguments are not typechecked, local variables
  - They can easily lead to errors that are more difficult to find (see previous slide)



Some Lab 5 Provided Macros

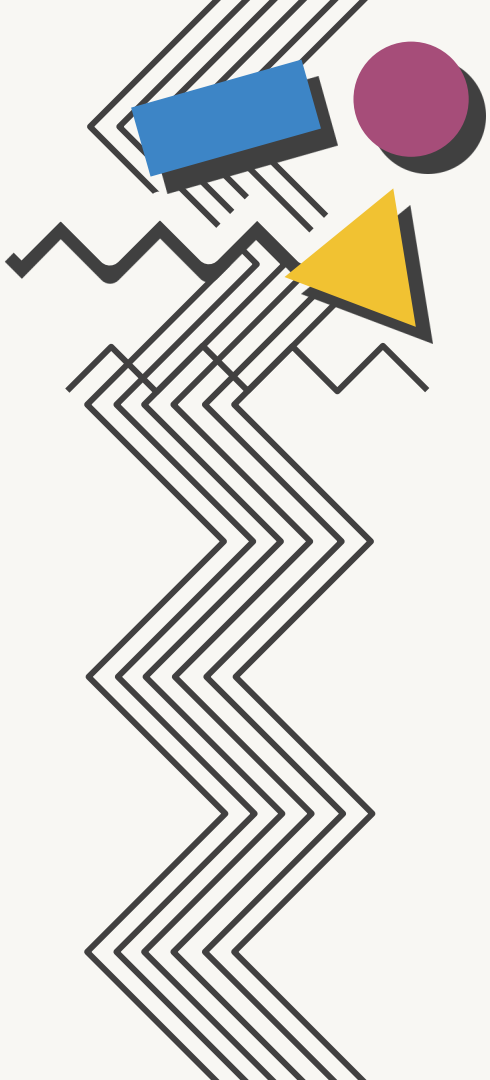
- **`UNSCALED_POINTER_ADD(p,x)`** Add without using “pointer arithmetic”
- **`UNSCALED_POINTER_SUB(p,x)`** Subtract without using “pointer arithmetic”
- **`MIN_BLOCK_SIZE`** The size of the smallest block that is safe to allocate
- **`SIZE(x)`** Gets the size from the ‘sizeAndTags’ field
- **`TAG_USED`** Mask for the used tag
- **`TAG_PRECEDING_USED`** Mask for the preceding used tag

- There are lots more, don’t forget to use them!
  - They will absolutely make your life easier
  - Part of good C style (which will be part of this assignment’s grade)



Getting Started Lab 5:

- If you are struggling to understand where to get started, read through `coalesceFreeBlock()`
  - Understanding the details of this function will provide clarity on the general structure you are manipulating
- Make sure you use the provided macros!
  - This will help to minimize potential bugs and make your code more readable
- HINT: The variables defined for you at the top of the `mm_malloc()` and `mm_free()` functions are good indicators of the code you will write.

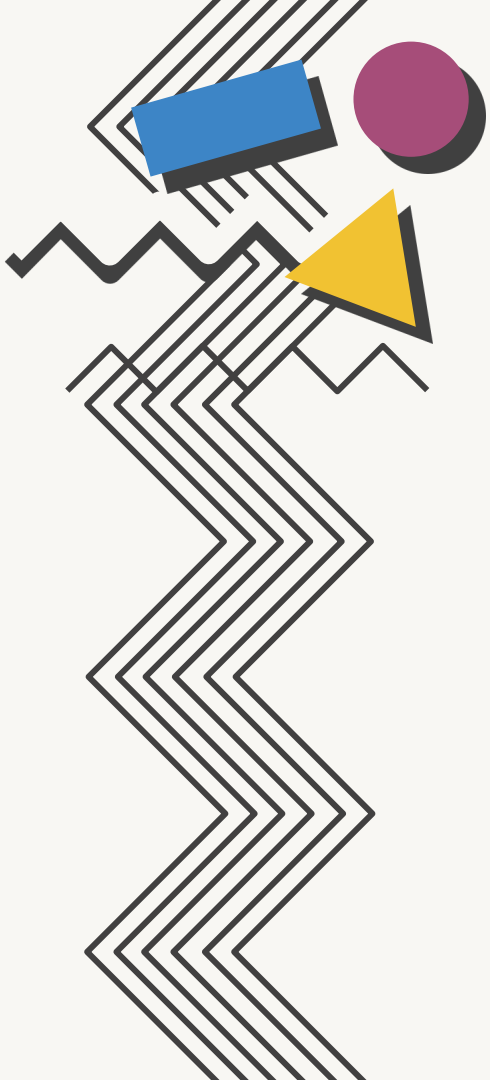


Worksheet

The BlockInfo struct

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

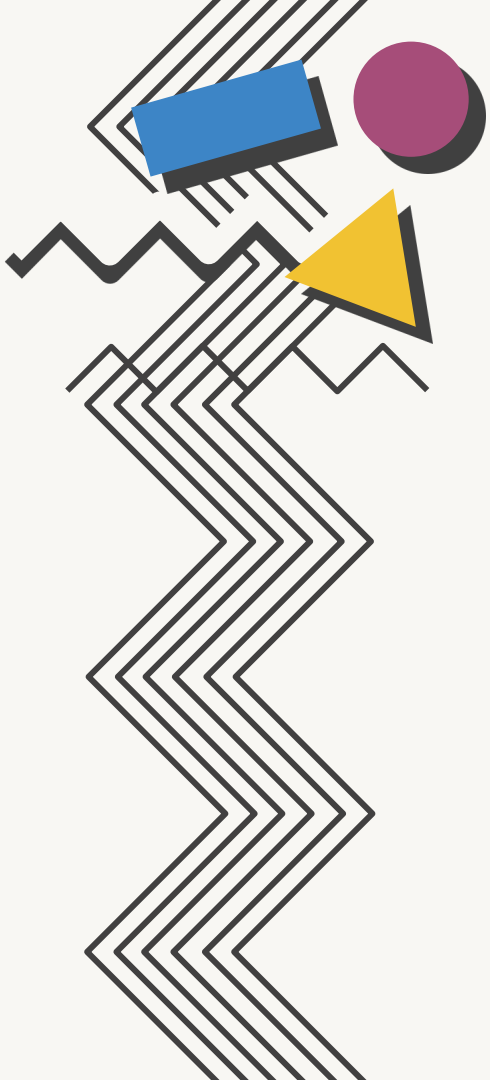


copyTags

```
// Bit masks used to retrieve tags from sizeAndTags.
#define TAG_USED 1
#define TAG_PRECEDING_USED 2
// SIZE(blockInfo->sizeAndTags) extracts the size of a 'sizeAndTags' field.
#define SIZE(X) ((X) & ~(TAG_USED + TAG_PRECEDING_USED))

// Copies the tags (TAG_PRECEDING_USED and TAG_USED) from blockToCopy
// to originalBlock. Leaves the size of originalBlock unchanged.
void copyTags(BlockInfo* originalBlock, BlockInfo* blockToCopy) {
  size_t copyUsed =

  size_t copyPrecedingUsed =

  originalBlock->sizeAndTags =

}
```

removeFreeBlock

```
BlockInfo *FREE_LIST_HEAD;
// Removes a block from the free list.
void removeFreeBlock(BlockInfo* freeBlock) {
  BlockInfo *nextFree, *prevFree;
  nextFree = freeBlock->next;
  prevFree = freeBlock->prev;




}
```



That's All, Folks!

Thanks for attending section! Feel free to stick around for a bit if you have quick questions (otherwise post on Ed or go to office hours).

See you all next week for a surprise!