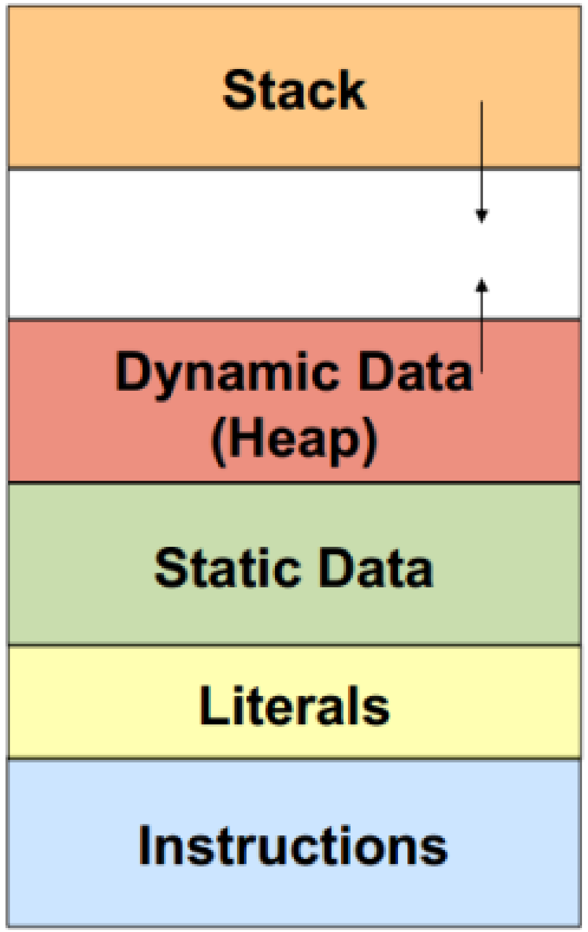
Malloc Lab, Part 1

OMET Survey

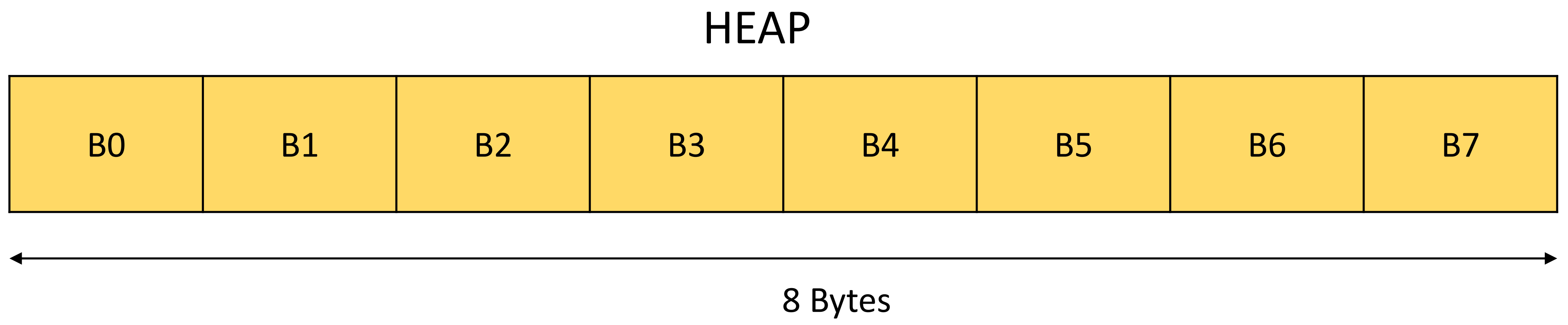
- Please take some time to complete the OMET survey of my recitations.

Heap



https://i.stack.imgur.com/HOY4C.png

Heap

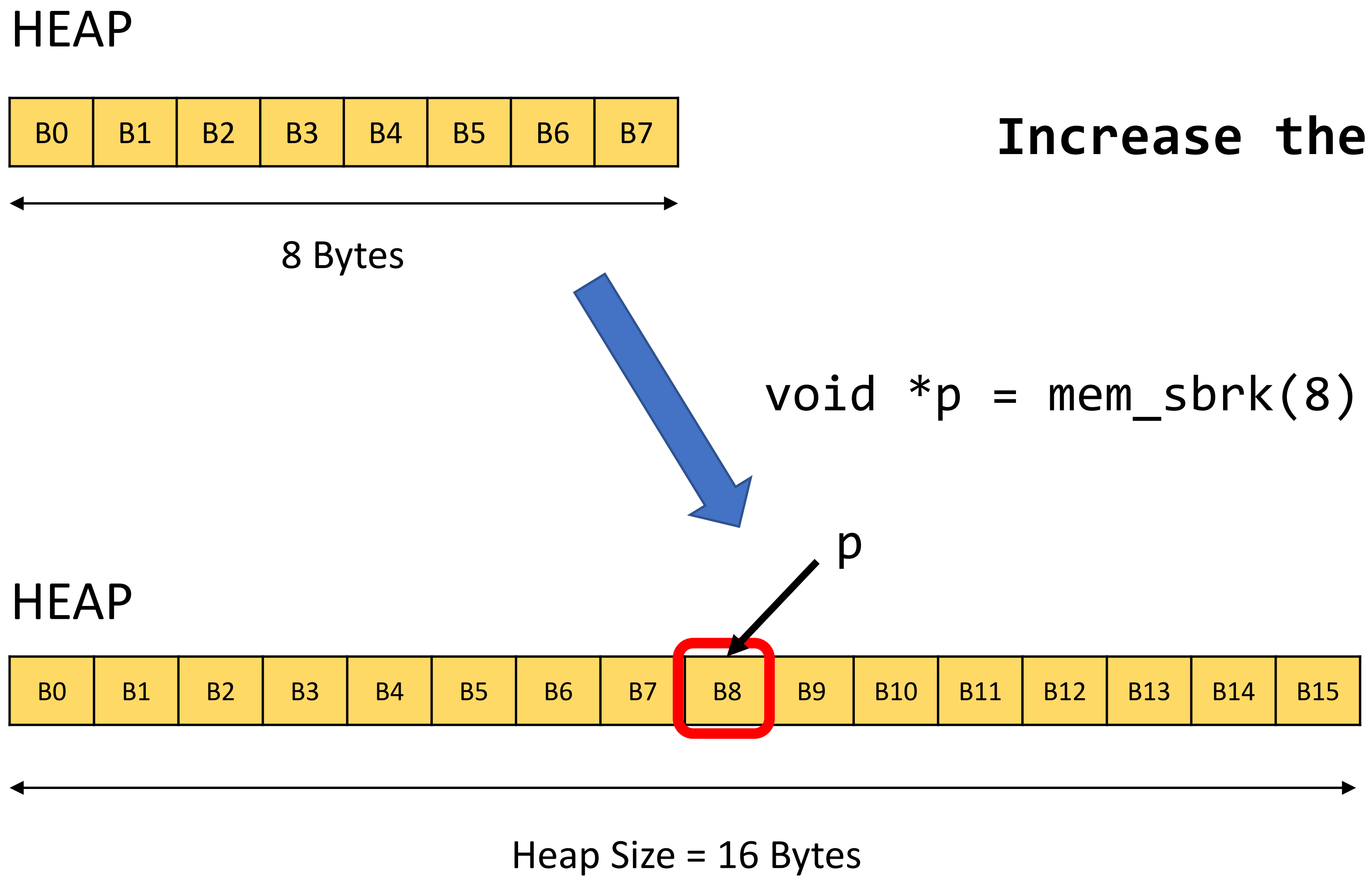


A sequence of Bytes!

Increasing the heap size

`Increase the Heap by 8 Bytes`

```
void *p = mem_sbrk(8)
```

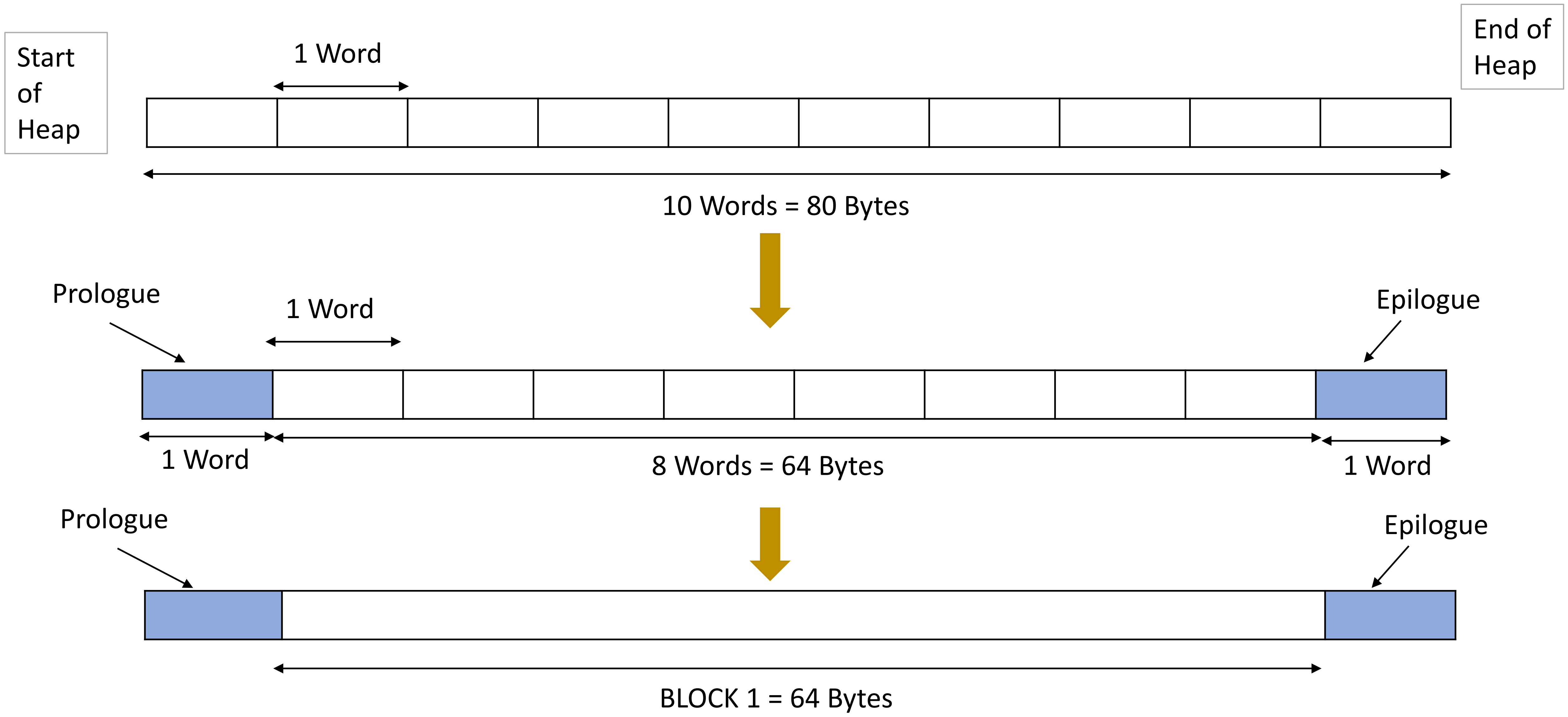


mem_sbrk() (part of *"memlib.c"*)

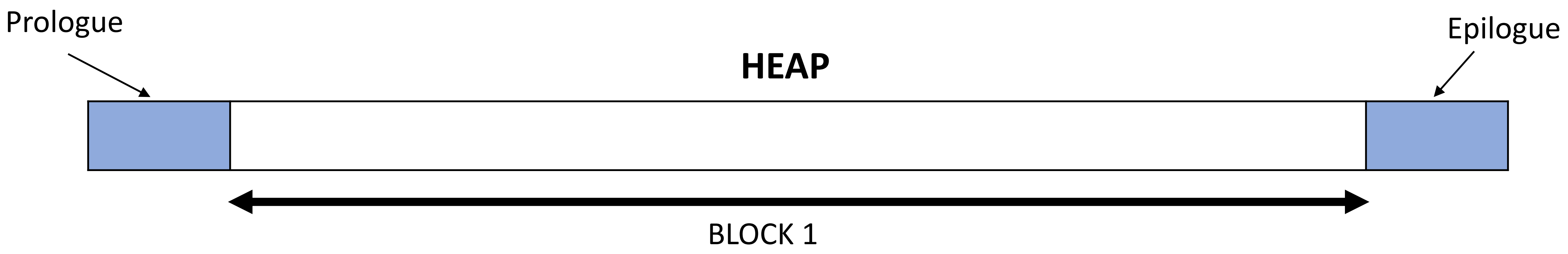
```
void *mem_sbrk(int incr)
{
    char *old_brk = mem_brk;

    if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
      errno = ENOMEM;
      fprintf(stderr, "ERROR: mem_sbrk failed. Ran out of memory...\n");
      return (void *)-1;
    }
    mem_brk += incr;
    return (void *)old_brk;
}
```

Visualizing the Heap

1 Word = 8 Bytes



Let’s Allocate (malloc()) some space from Heap



```
int *p = (int*) malloc (12);
```

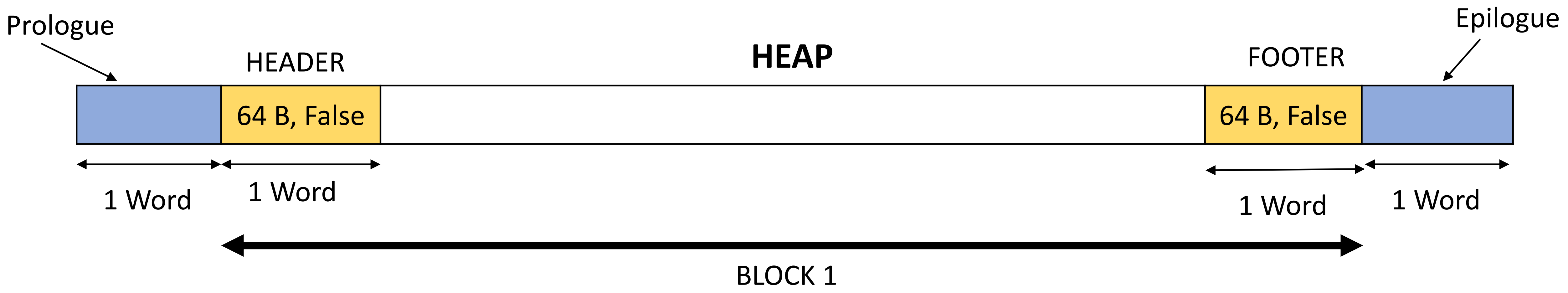
Is Block 1 free ??

- Can allocate there only if it is FREE

Or is there some data in there already??

How Will I know??!

Header and Footer



BLOCK 1:

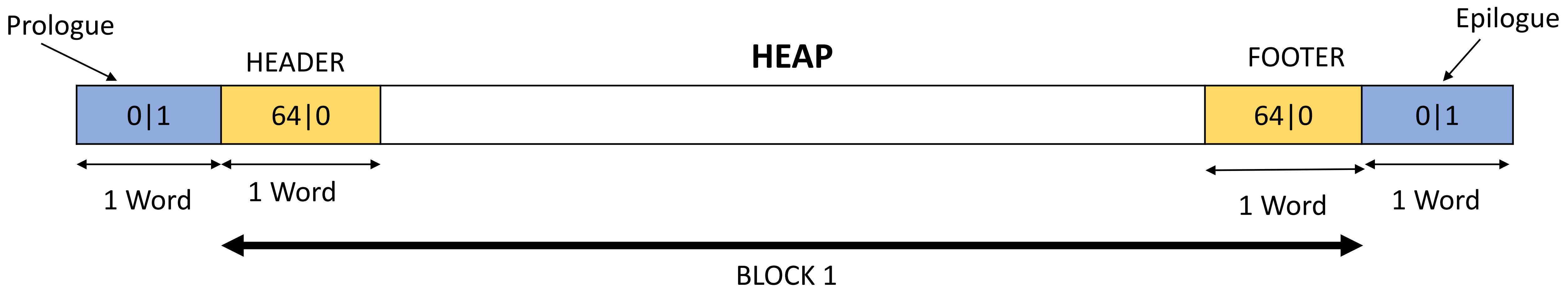
<u>Size:</u> 64 Bytes

<u>Allocation Status:</u> False

Now I know that Block 1 is free because:

- I can check the header and/or footer to check the allocation status.

Header and Footer



BLOCK 1:

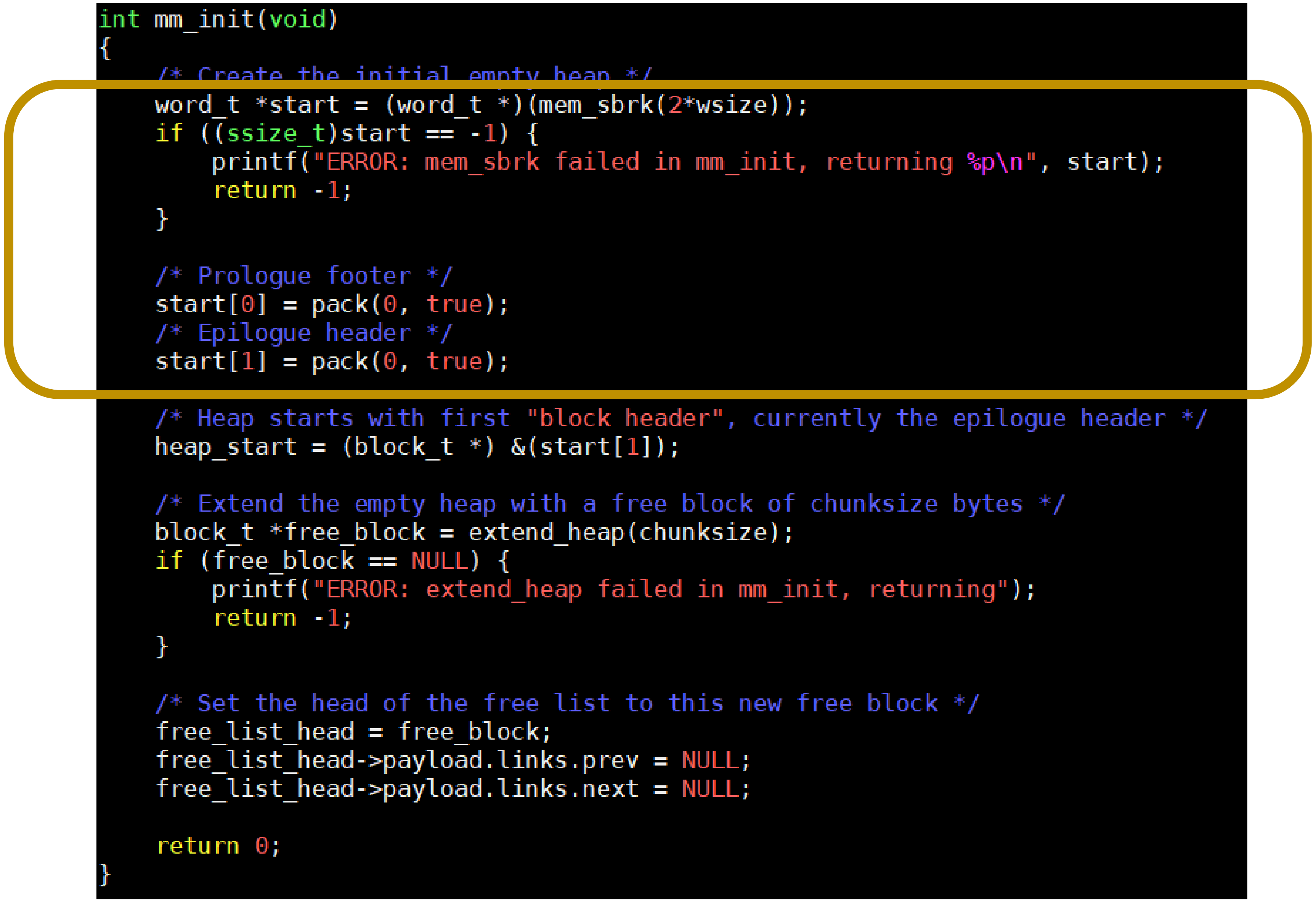
Size: 64 Bytes

Allocation Status: False

Now I know that Block 1 is free because:

- I can check the header and/or footer to check the allocation status.

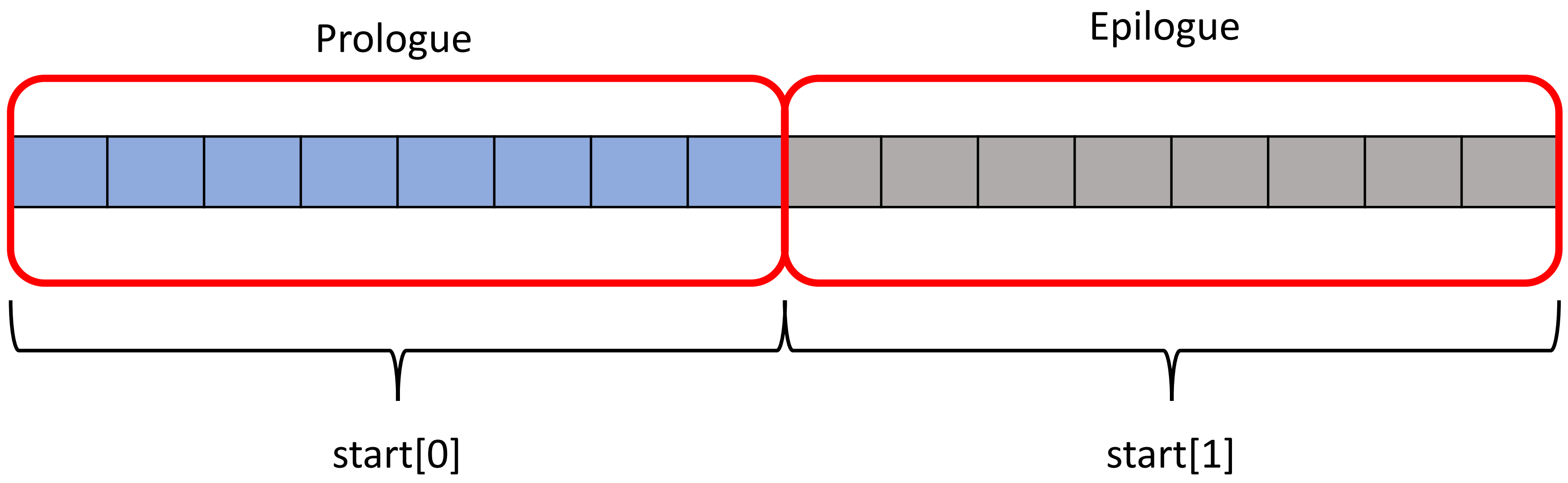
Let’s See Some Code



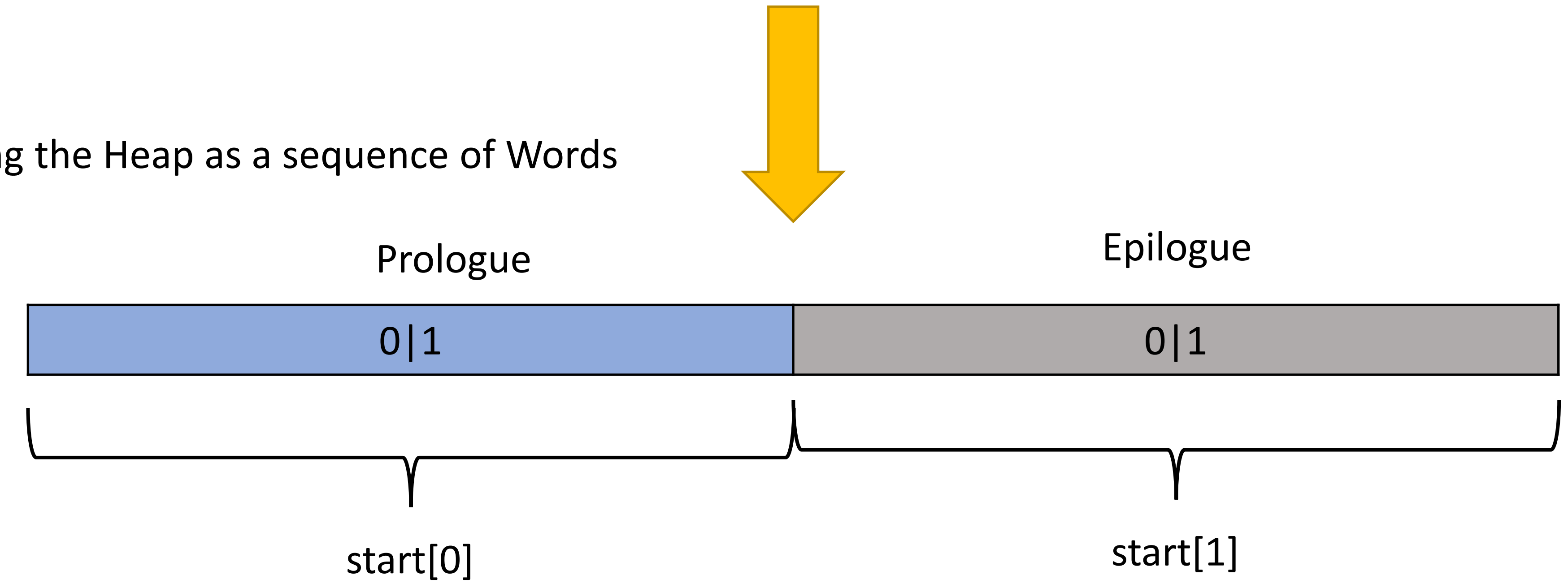
```c
int mm_init(void)
{
    /* Create the initial empty heap */
    word_t *start = (word_t *)(mem_sbrk(2*wsize));
    if ((ssize_t)start == -1) {
        printf("ERROR: mem_sbrk failed in mm_init, returning %p\n", start);
        return -1;
    }

    /* Prologue footer */
    start[0] = pack(0, true);
    /* Epilogue header */
    start[1] = pack(0, true);

    /* Heap starts with first "block header", currently the epilogue header */
    heap_start = (block_t *) &(start[1]);

    /* Extend the empty heap with a free block of chunksize bytes */
    block_t *free_block = extend_heap(chunksize);
    if (free_block == NULL) {
        printf("ERROR: extend_heap failed in mm_init, returning");
        return -1;
    }

    /* Set the head of the free list to this new free block */
    free_list_head = free_block;
    free_list_head->payload.links.prev = NULL;
    free_list_head->payload.links.next = NULL;

    return 0;
}
```

1 word = 8 Bytes

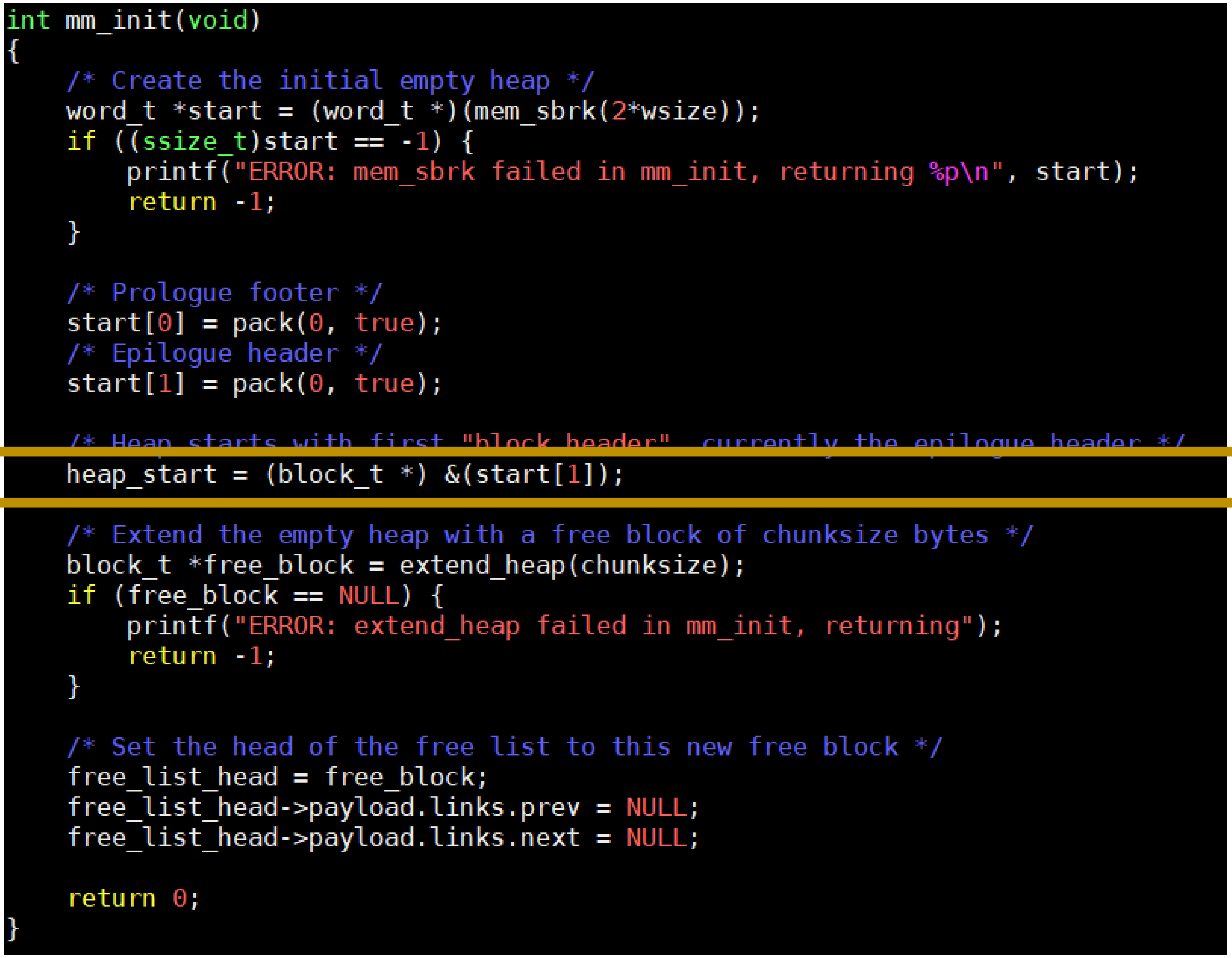
Representing the Heap as a sequence of Bytes



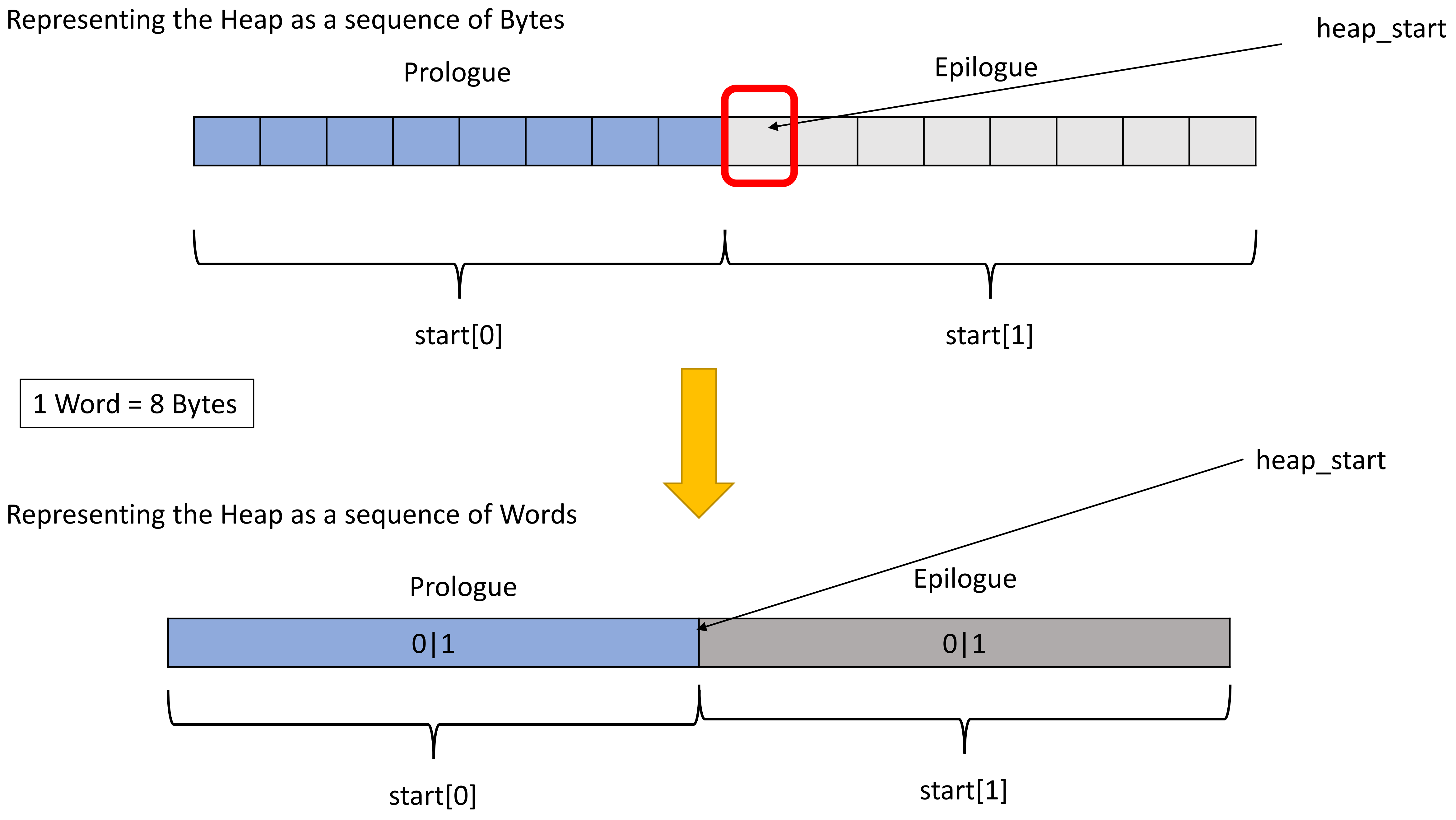
Representing the Heap as a sequence of Words

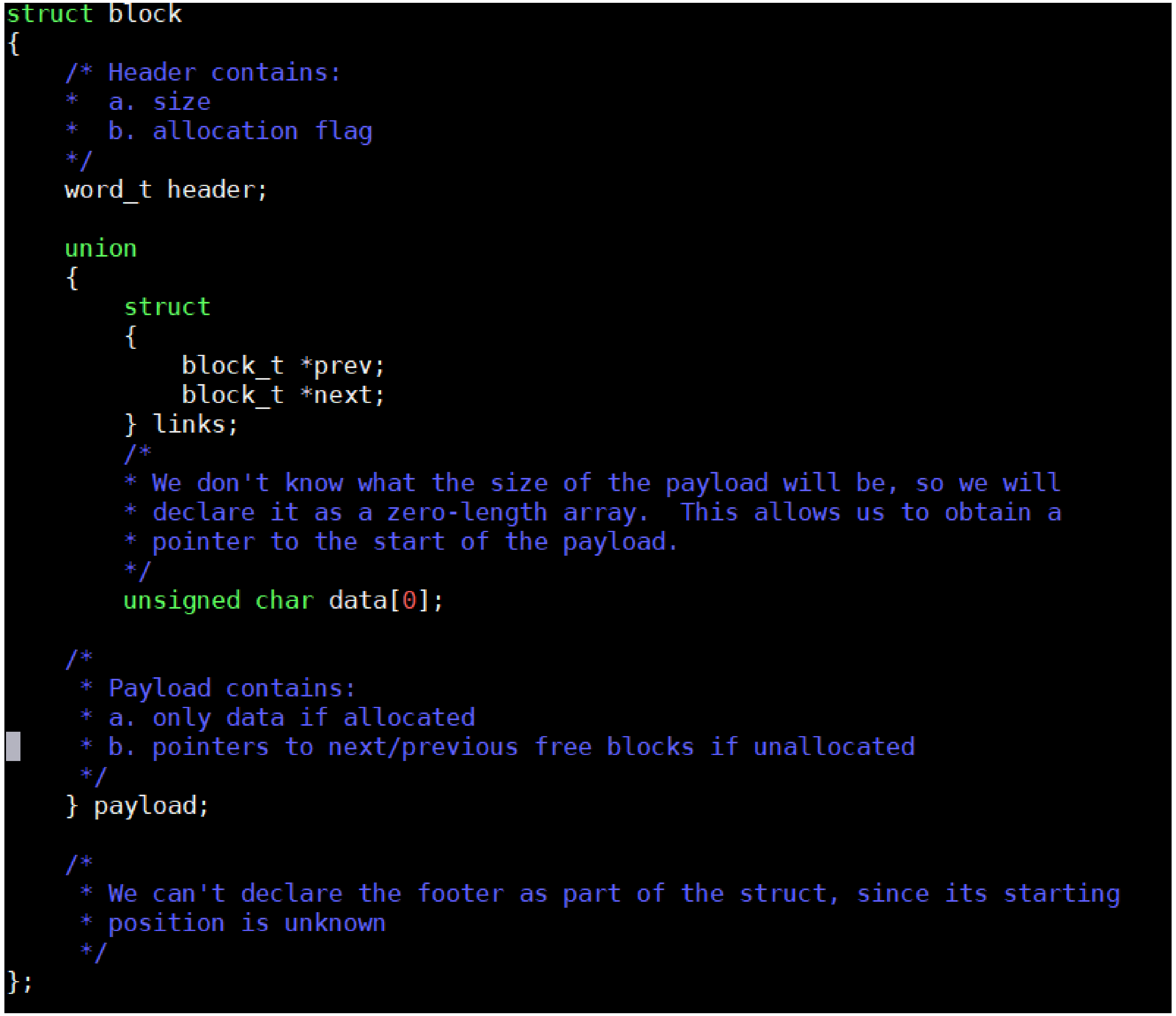


Let’s See Some Code



```
int mm_init(void)
{
    /* Create the initial empty heap */
    word_t *start = (word_t *)(mem_sbrk(2*wsize));
    if ((ssize_t)start == -1) {
        printf("ERROR: mem_sbrk failed in mm_init, returning %p\n", start);
        return -1;
    }

    /* Prologue footer */
    start[0] = pack(0, true);
    /* Epilogue header */
    start[1] = pack(0, true);

    /* Heap starts with first "block header", currently the epilogue header */
    heap_start = (block_t *) &(start[1]);

    /* Extend the empty heap with a free block of chunksize bytes */
    block_t *free_block = extend_heap(chunksize);
    if (free_block == NULL) {
        printf("ERROR: extend_heap failed in mm_init, returning");
        return -1;
    }

    /* Set the head of the free list to this new free block */
    free_list_head = free_block;
    free_list_head->payload.links.prev = NULL;
    free_list_head->payload.links.next = NULL;

    return 0;
}
```


Representing the Heap as a sequence of Bytes
heap_start
Prologue
Epilogue
start[0]
start[1]
1 Word = 8 Bytes
heap_start
Representing the Heap as a sequence of Words
Prologue
Epilogue
0|1
0|1
start[0]
start[1]



```c
struct block
{
    /* Header contains:
    *  a. size
    *  b. allocation flag
    */
    word_t header;

    union
    {
        struct
        {
            block_t *prev;
            block_t *next;
        } links;
        /*
        * We don't know what the size of the payload will be, so we will
        * declare it as a zero-length array.  This allows us to obtain a
        * pointer to the start of the payload.
        */
        unsigned char data[0];

    /*
     * Payload contains:
     * a. only data if allocated
     * b. pointers to next/previous free blocks if unallocated
     */
    } payload;

    /*
     * We can't declare the footer as part of the struct, since its starting
     * position is unknown
     */
};
```

Union

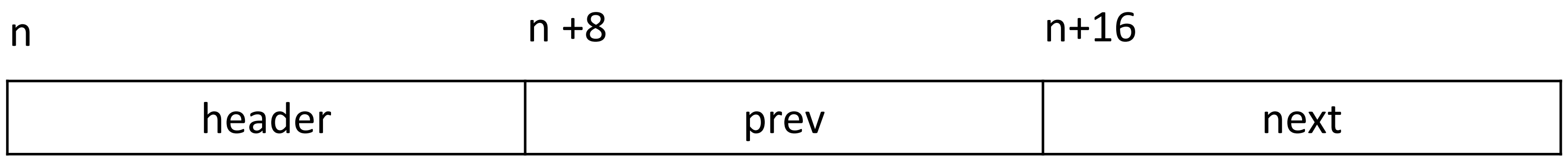
- Union allocates one common storage space for all its members

```
union student {
        int a;
        int b;
        struct s{
                int x;
                int y;
        }struct_var;
};
```

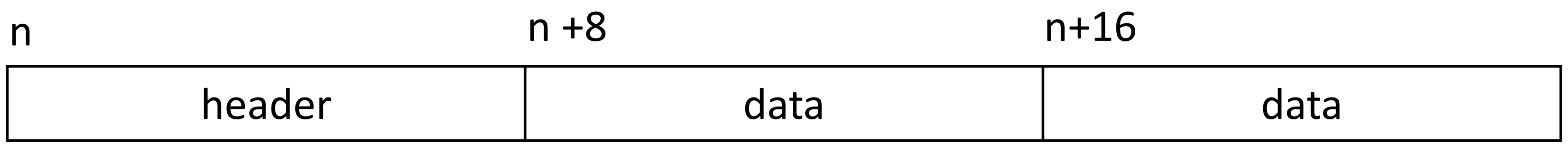
```
int main(){
        union student st;
        printf("Size of Union:%ld", sizeof(st)); //8
        printf("\nAddress of a:%p",&st.a );//0x04
        printf("\nAddress of b:%p",&st.b );//0x04
        printf("\nAddress of x:%p",&st.struct_var.x);//0x04
        printf("\nAddress of x:%p",&st.struct_var.y);//0x08
        return 0;
}
```

struct block

If NOT allocated

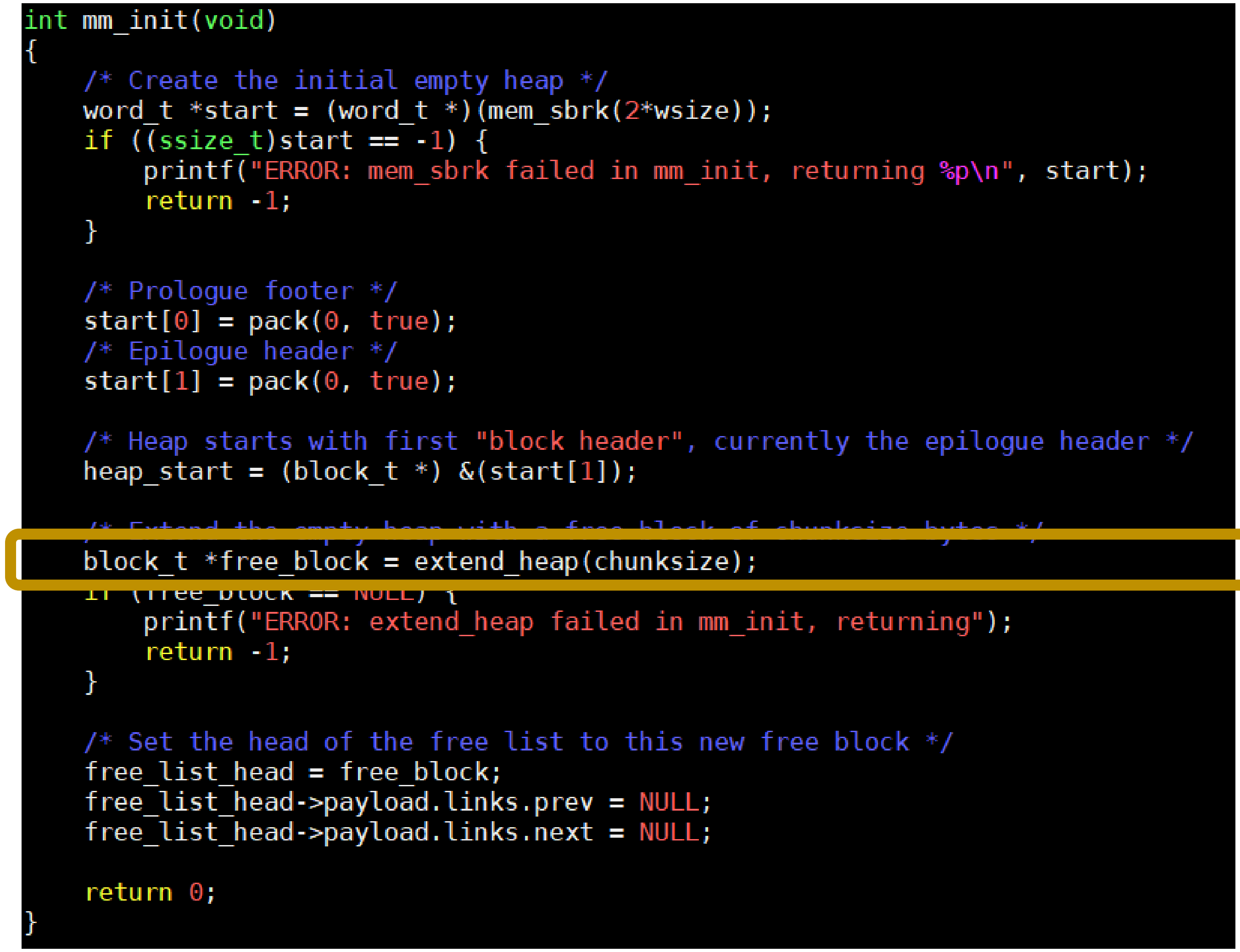


If allocated

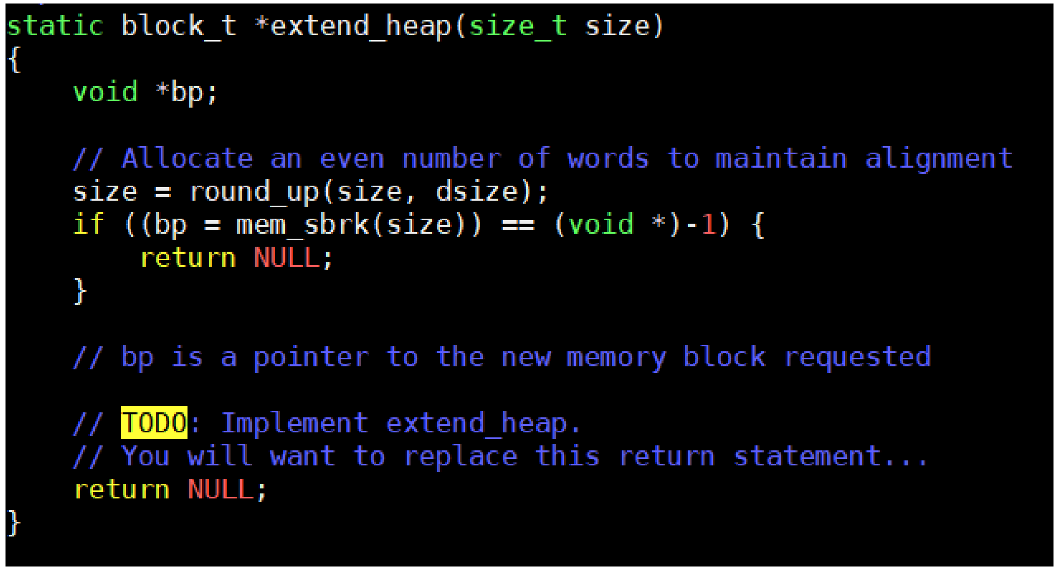


- You DO NOT need to put any data in the allocated space
  - **You just need to know the STARTING ADDRESS OF data (which is the address of header + size of a word)**
- Utilize the prev and next pointer when using explicit list traversal

Let’s See Some Code

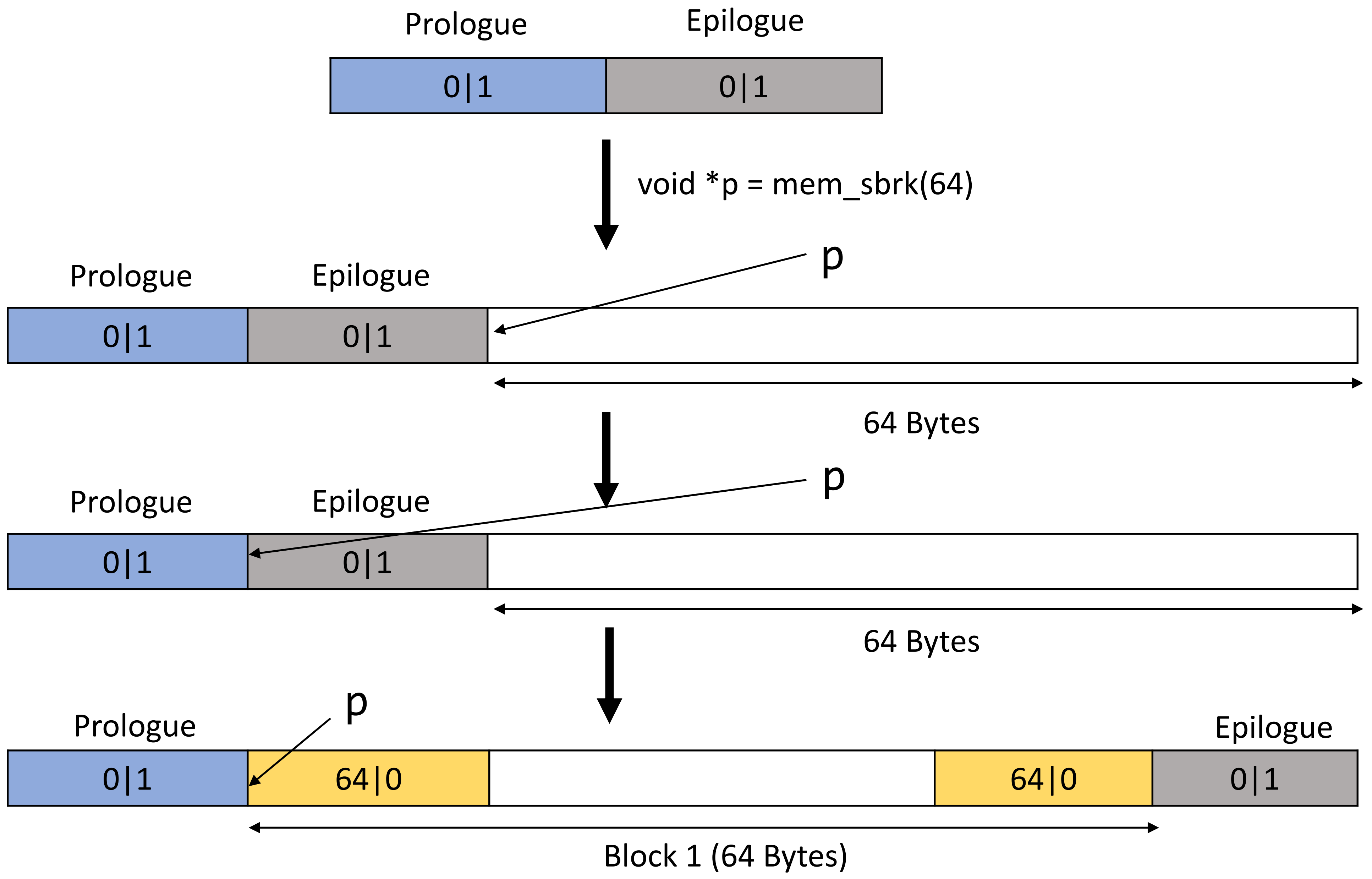


```c
int mm_init(void)
{
    /* Create the initial empty heap */
    word_t *start = (word_t *)(mem_sbrk(2*wsize));
    if ((ssize_t)start == -1) {
        printf("ERROR: mem_sbrk failed in mm_init, returning %p\n", start);
        return -1;
    }

    /* Prologue footer */
    start[0] = pack(0, true);
    /* Epilogue header */
    start[1] = pack(0, true);

    /* Heap starts with first "block header", currently the epilogue header */
    heap_start = (block_t *) &(start[1]);

    /* Extend the empty heap with a free block of chunksize bytes */
    block_t *free_block = extend_heap(chunksize);
    if (free_block == NULL) {
        printf("ERROR: extend_heap failed in mm_init, returning");
        return -1;
    }

    /* Set the head of the free list to this new free block */
    free_list_head = free_block;
    free_list_head->payload.links.prev = NULL;
    free_list_head->payload.links.next = NULL;

    return 0;
}
```

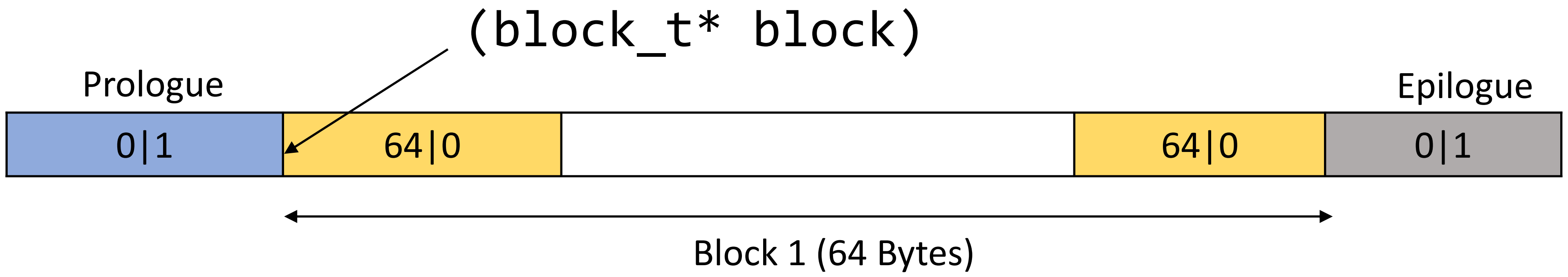


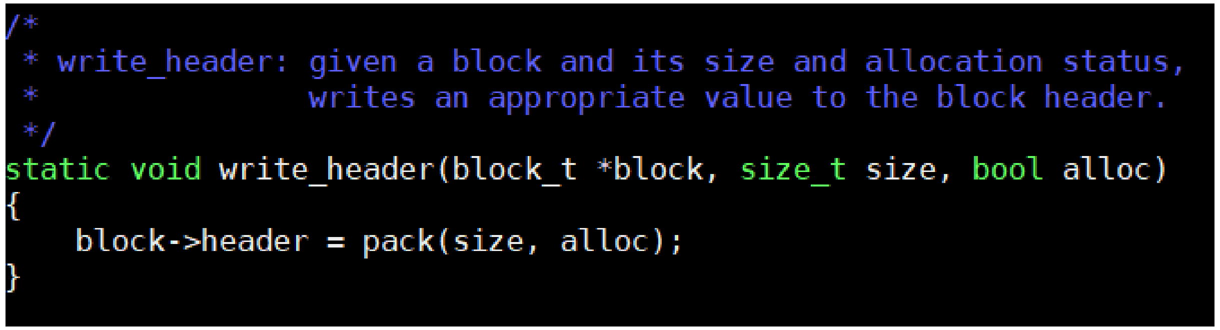
```c
static block_t *extend_heap(size_t size)
{
    void *bp;

    // Allocate an even number of words to maintain alignment
    size = round_up(size, dsize);
    if ((bp = mem_sbrk(size)) == (void *)-1) {
        return NULL;
    }

    // bp is a pointer to the new memory block requested

    // TODO: Implement extend_heap.
    // You will want to replace this return statement...
    return NULL;
}
```

Example: extend_heap by 64 Bytes

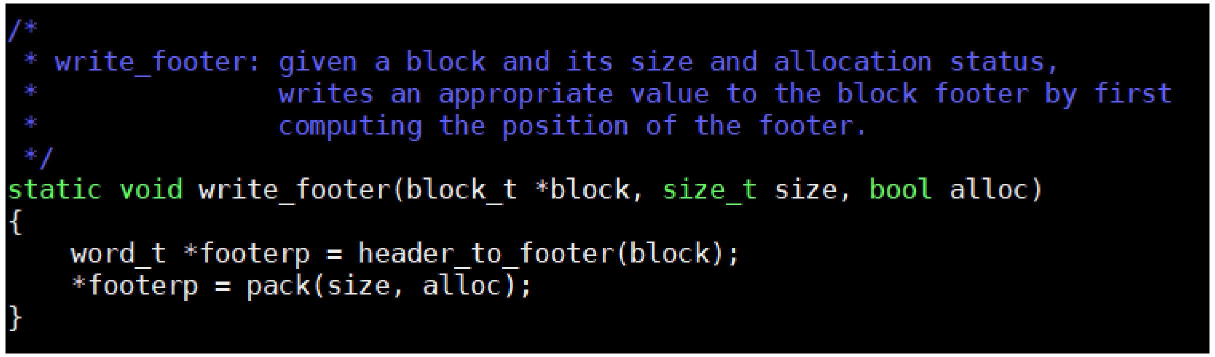


Writing header and footer of a Block

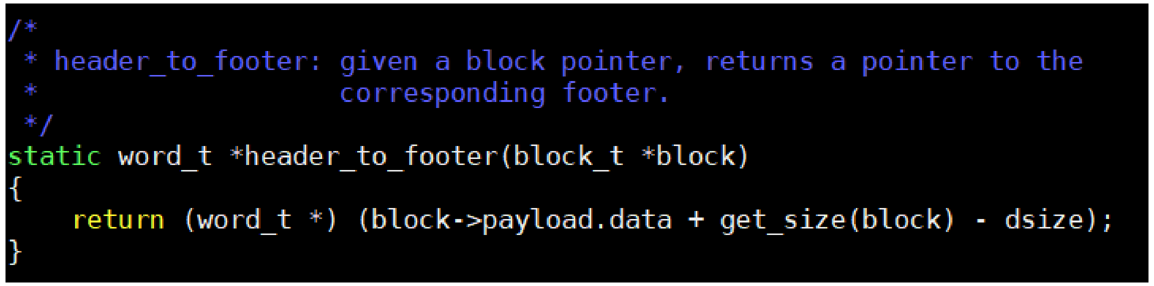




```
/*
 * write_header: given a block and its size and allocation status,
 *               writes an appropriate value to the block header.
 */
static void write_header(block_t *block, size_t size, bool alloc)
{
    block->header = pack(size, alloc);
}
```

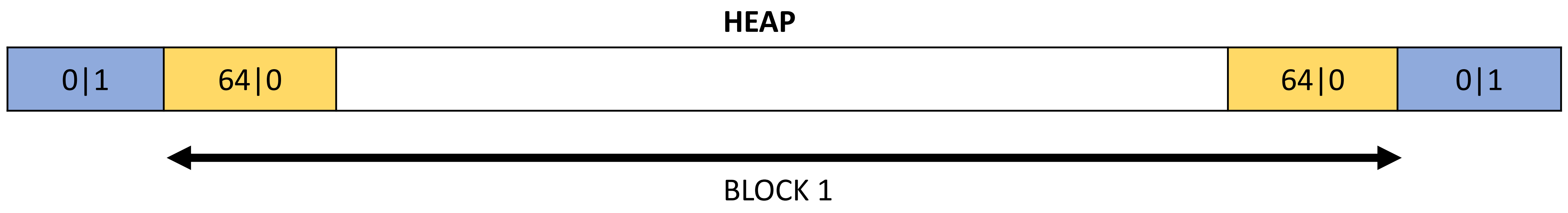


```
/*
 * write_footer: given a block and its size and allocation status,
 *               writes an appropriate value to the block footer by first
 *               computing the position of the footer.
 */
static void write_footer(block_t *block, size_t size, bool alloc)
{
    word_t *footerp = header_to_footer(block);
    *footerp = pack(size, alloc);
}
```



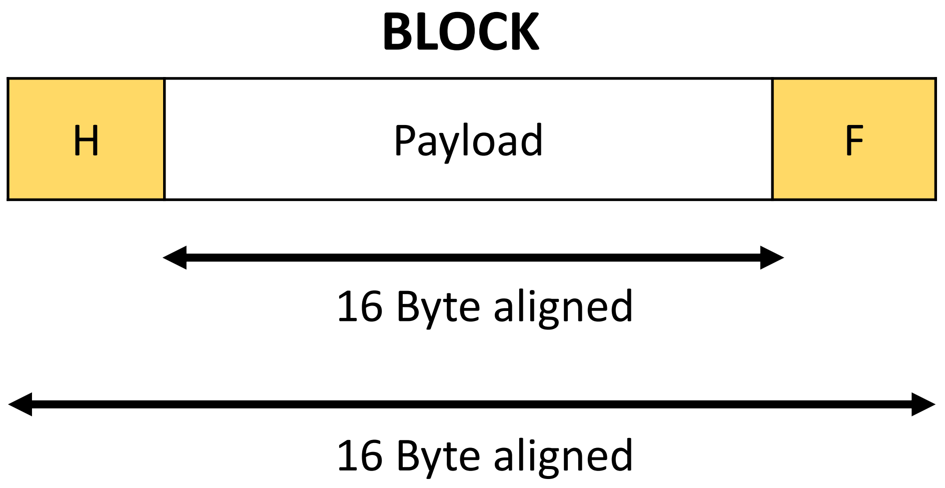
```
/*
 * header_to_footer: given a block pointer, returns a pointer to the
 *                   corresponding footer.
 */
static word_t *header_to_footer(block_t *block)
{
    return (word_t *) (block->payload.data + get_size(block) - dsize);
}
```

Allocation (mm_malloc())

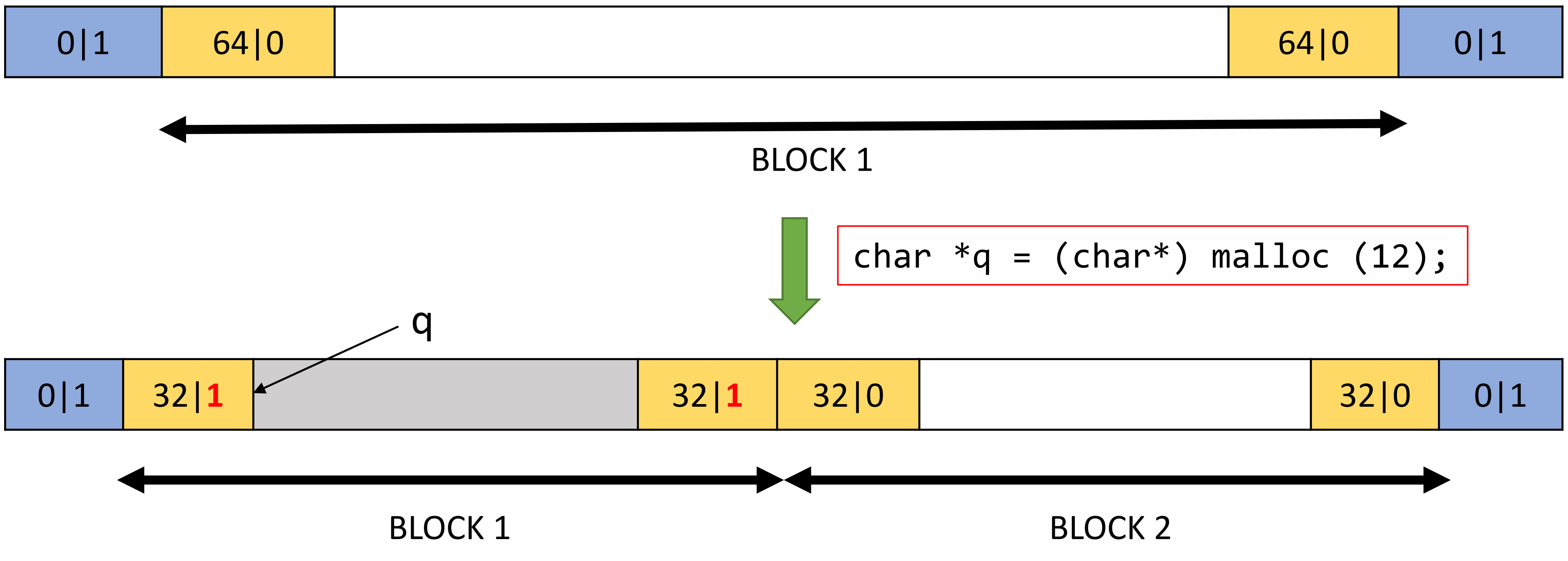


```
char *q = (char*) malloc (12);
```

- What will be the size of the payload?
  - 16 Bytes
- What will be the size of the new block?
  - Size of Payload + Size of Header + Size of Footer = 32 Bytes
- So, what is the minimum Block Size?
  - 32 Bytes



HEAP

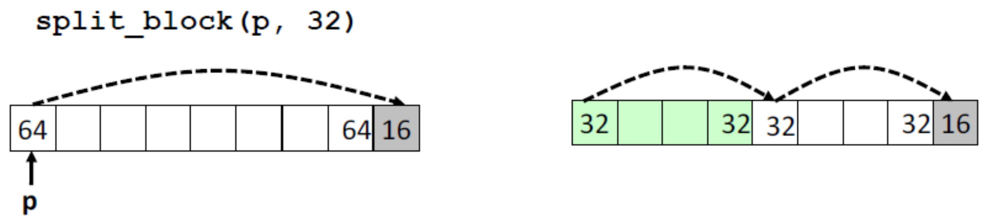


```
char *q = (char*) malloc (12);
```

- How much space did we waste in Block 1?
  - 32 Bytes - 12 Bytes = 20 Bytes

This is called **Internal Fragmentation**

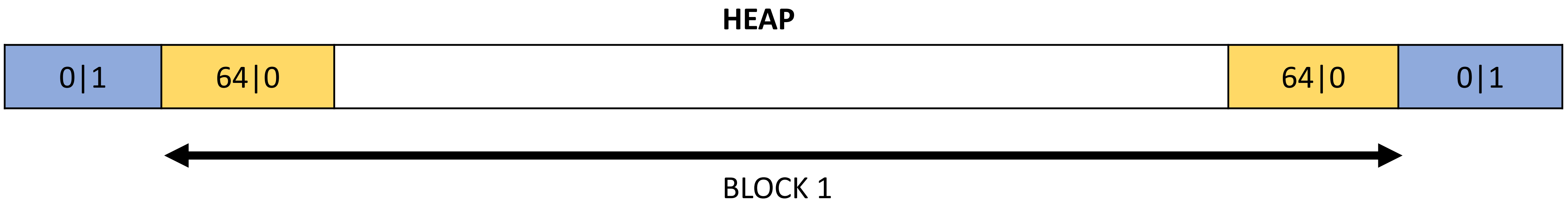
Splitting A Free Block to Two Blocks



```
static void split_block(block_t *block, size_t asize) {
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        write_footer(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next, block_size - asize, false);
        write_footer(block_next, block_size - asize, false);
    }
}
```

```
/*
 * find_next: returns the next consecutive block on the heap by adding the
 *            size of the block.
 */
static block_t *find_next(block_t *block)
{
    return (block_t *) ((unsigned char *) block + get_size(block));
}
```

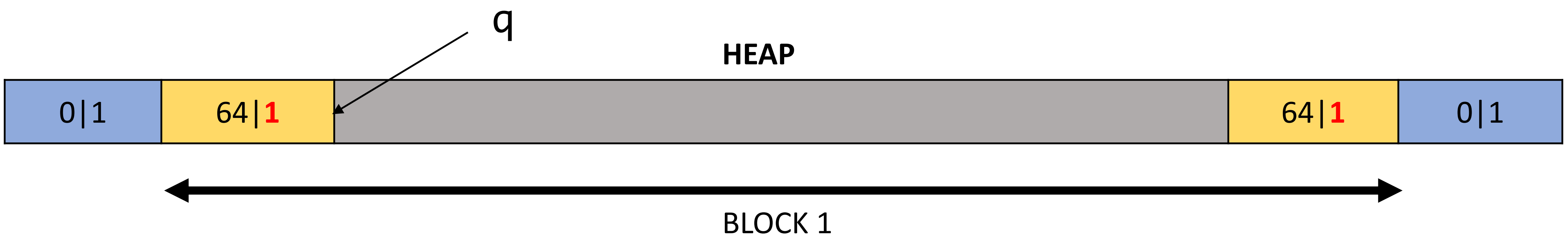
Should I split Here?



```
char *q = (char*) malloc (20);
```

- What will be the size of the payload?
  - 32 Bytes
- What will be the size of the new block?
  - 48 Bytes
- What is the remaining space?
  - 16 Bytes
- Is the remaining space enough for a block?
  - NO. Minimum Block Size is 32 B

Should I split Here? No



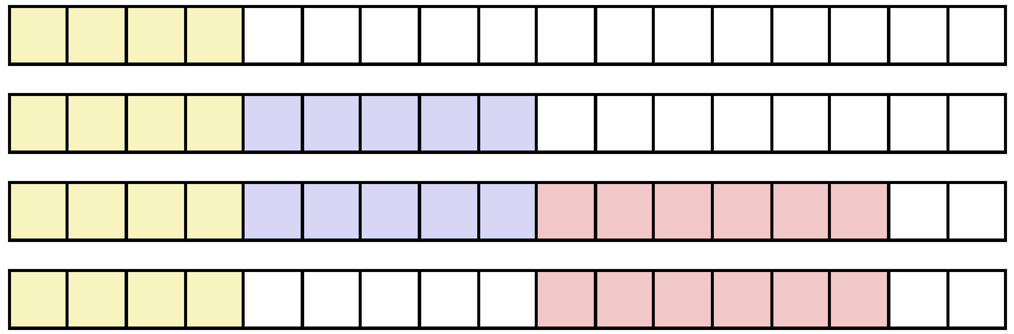
```
char *q = (char*) malloc (20);
```

- What will be the size of the payload?
  - 32 Bytes
- What will be the size of the new block?
  - 48 Bytes
- What is the remaining space?
  - 16 Bytes
- Is the remaining space enough for a block?
  - NO. Minimum Block Size is 32 B

External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough
  - The aggregate payload is non-continuous

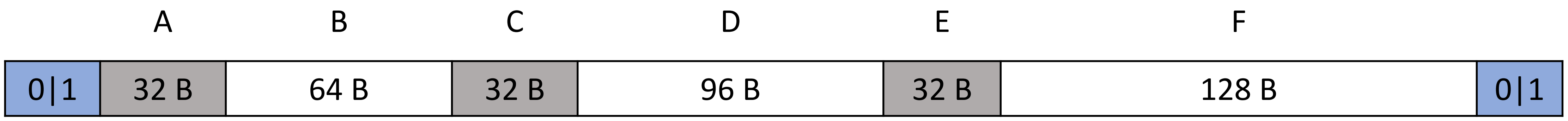
```
p1 = malloc(32)
p2 = malloc(40)
p3 = malloc(48)
free(p2)
p4 = malloc(48)
```



Oh no! (What would happen now?)

- Depends on the pattern of *future* requests
  - Thus, difficult to measure

Consider this scenario



`malloc (16)`

- Block Size = 32 Bytes

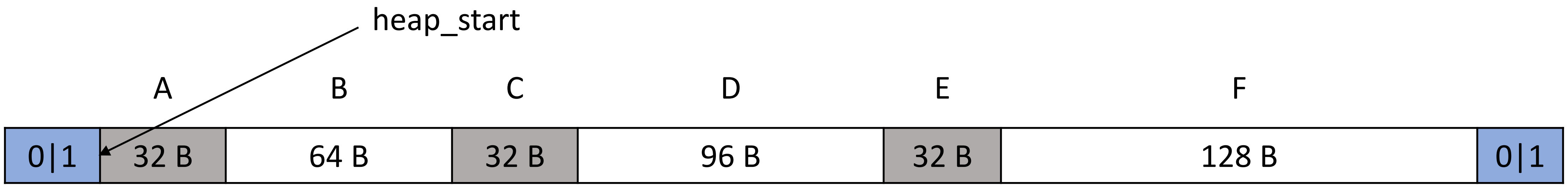
Need to allocate 32 Bytes in a FREE block

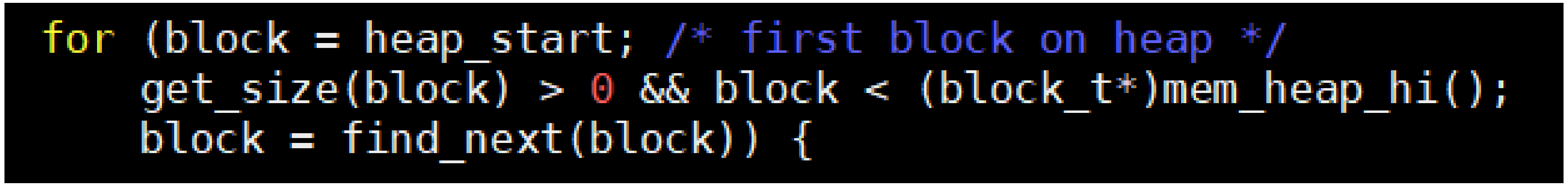
How do I find the Blocks which are Free?

- Implict List
    - Traverse all block and check if it is free

- Explicit List
    - Traverse only the FREE blocks

Implicit List





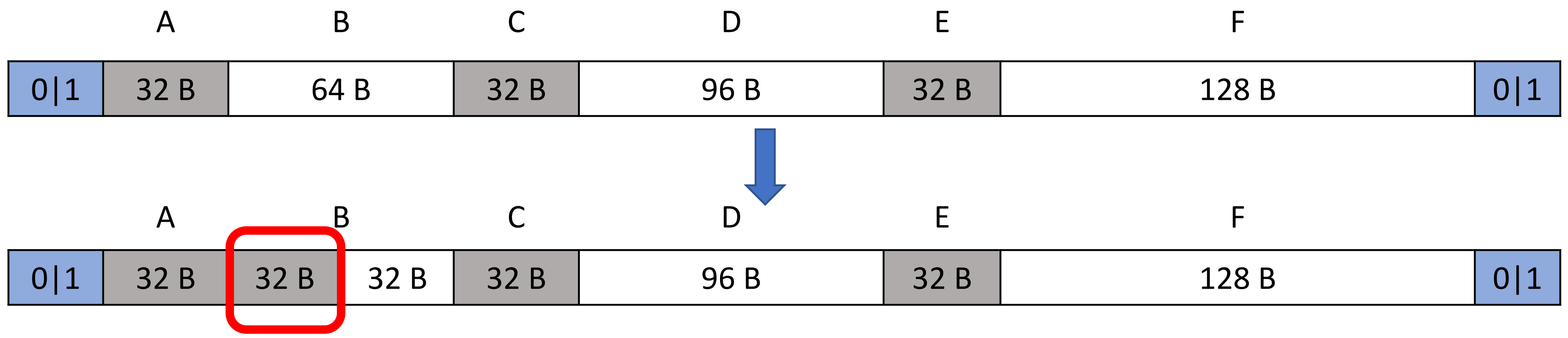
```
for (block = heap_start; /* first block on heap */
    get_size(block) > 0 && block < (block_t*)mem_heap_hi();
    block = find_next(block)) {
```

Which Free Block To allocate in?

`malloc (16)`

- Block Size = 32 Bytes

First Fit

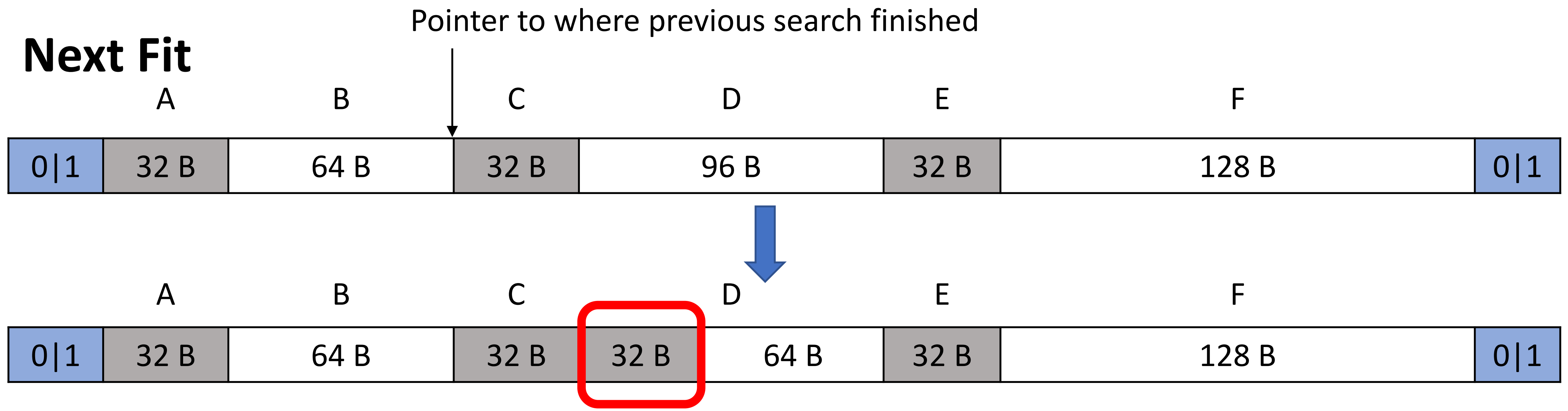


Which Free Block To allocate in?

`malloc (16)`

- Block Size = 32 Bytes

Next Fit

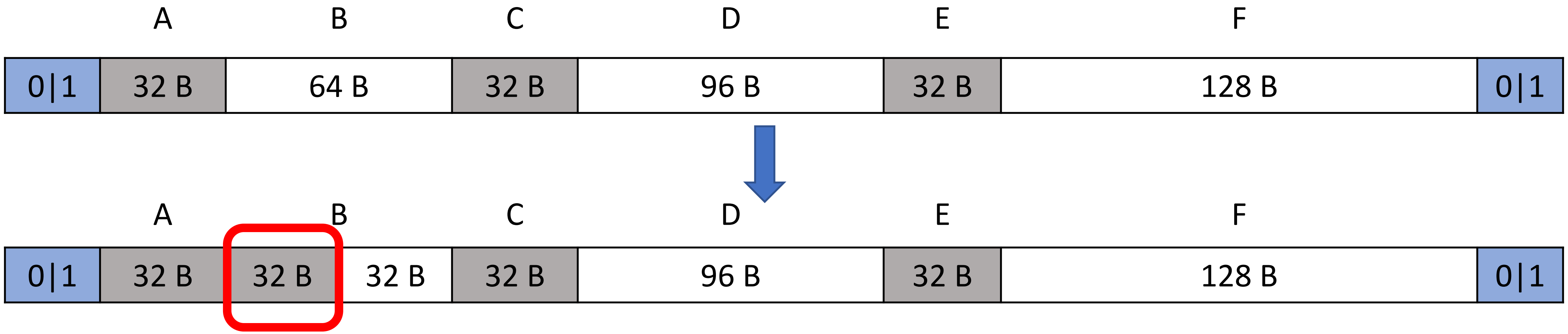


Which Free Block To allocate in?

`malloc (16)`

- Block Size = 32 Bytes

Best Fit

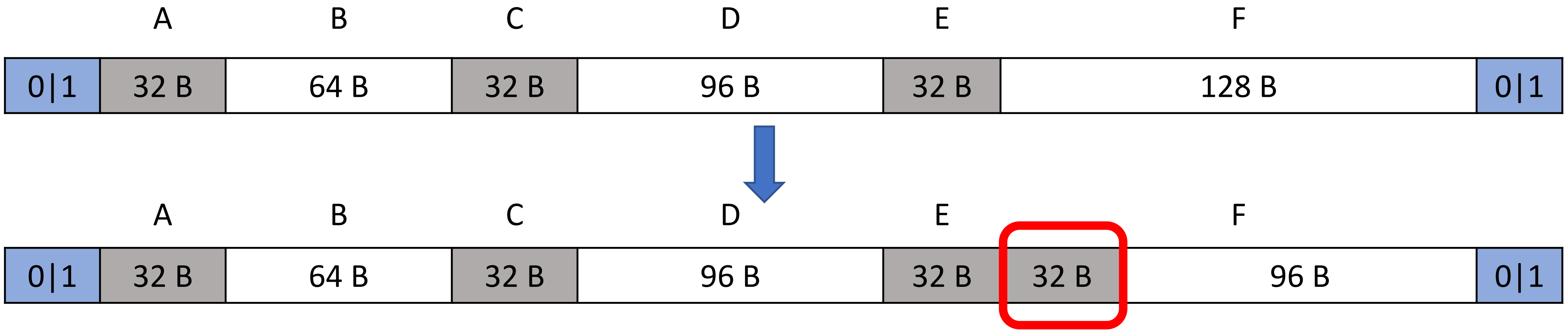


Which Free Block to allocate in?

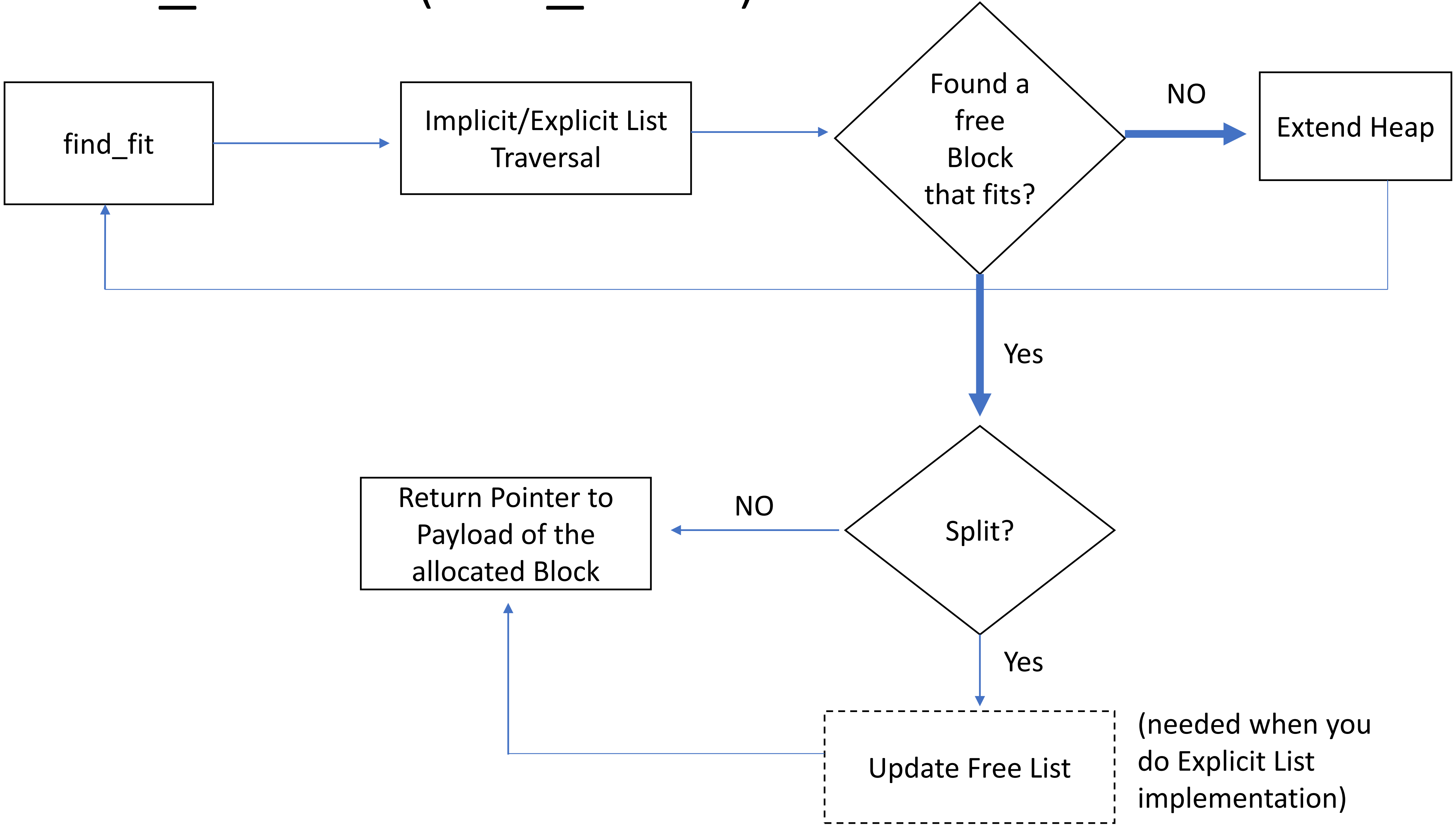
`malloc (16)`

- Block Size = 32 Bytes

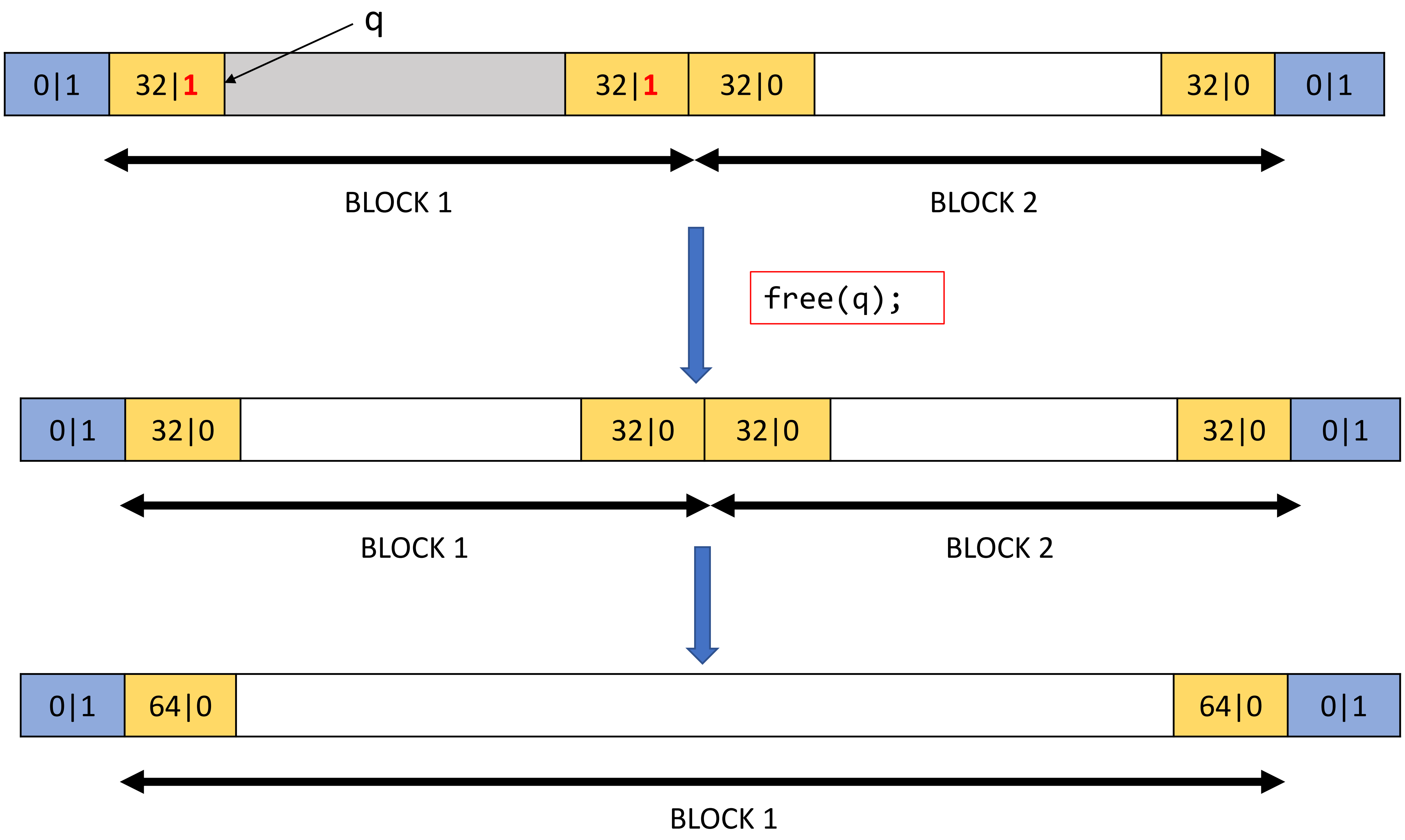
Worst Fit



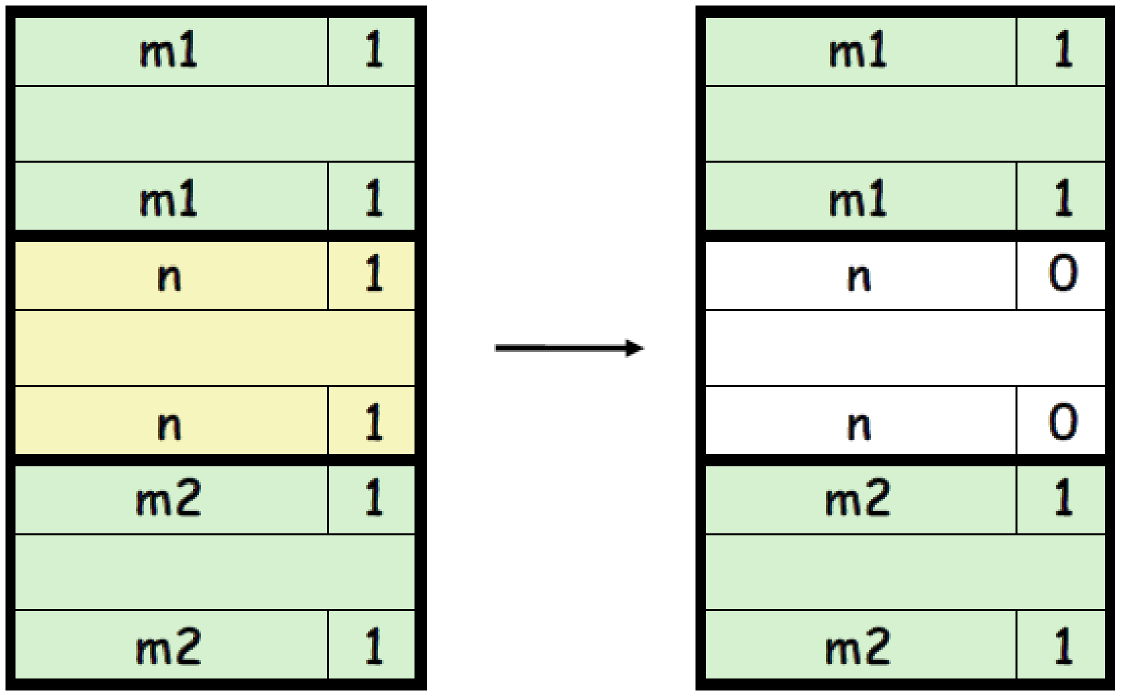
mm_malloc (size_t size)


find_fit
Implicit/Explicit List Traversal
Found a free Block that fits?
NO
Extend Heap
Yes
Split?
NO
Return Pointer to Payload of the allocated Block
Yes
Update Free List
(needed when you do Explicit List implementation)

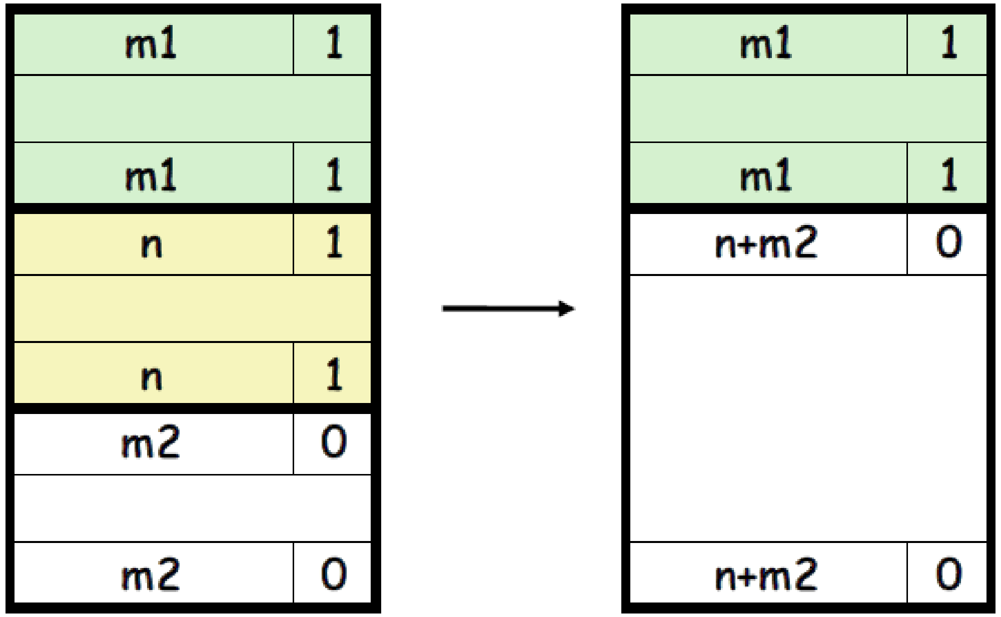
De-Allocation (mm_free())


q
0|1
32|1
32|1
32|0
32|0
0|1
BLOCK 1
BLOCK 2
free(q);
0|1
32|0
32|0
32|0
32|0
0|1
BLOCK 1
BLOCK 2
0|1
64|0
64|0
0|1
BLOCK 1

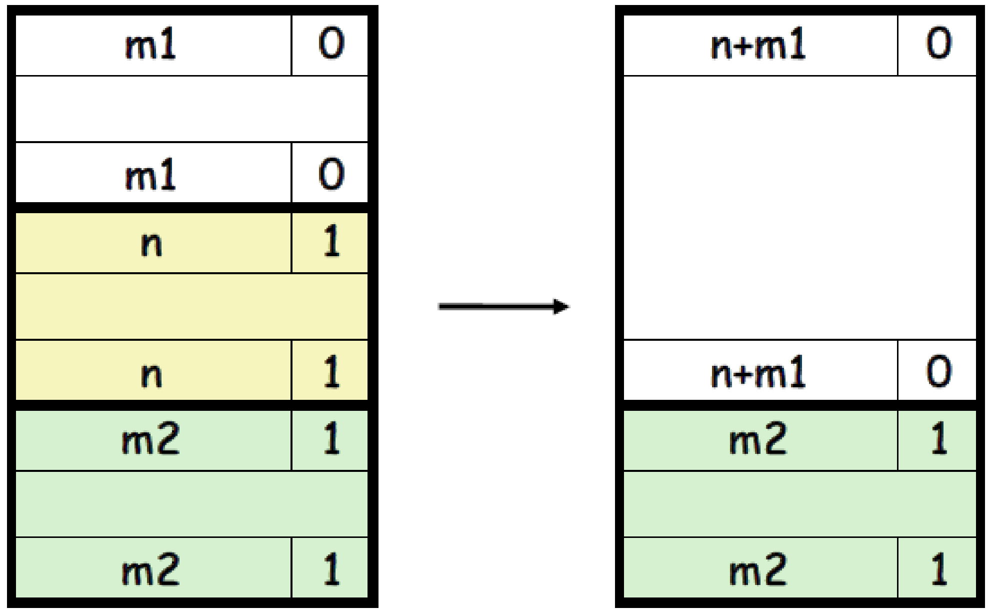
Coalescing



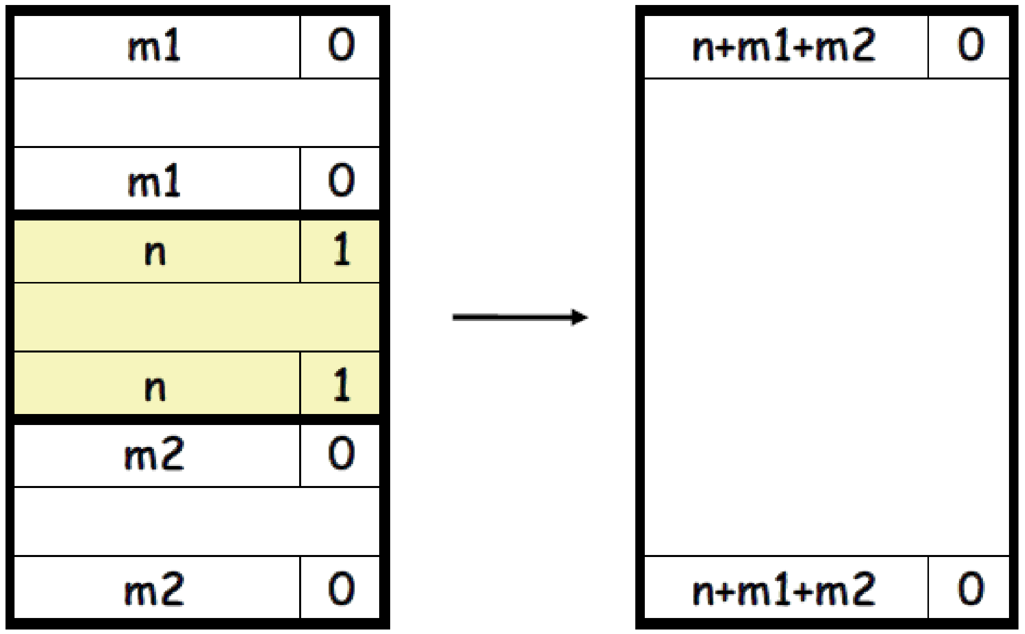
Coalescing



Coalescing



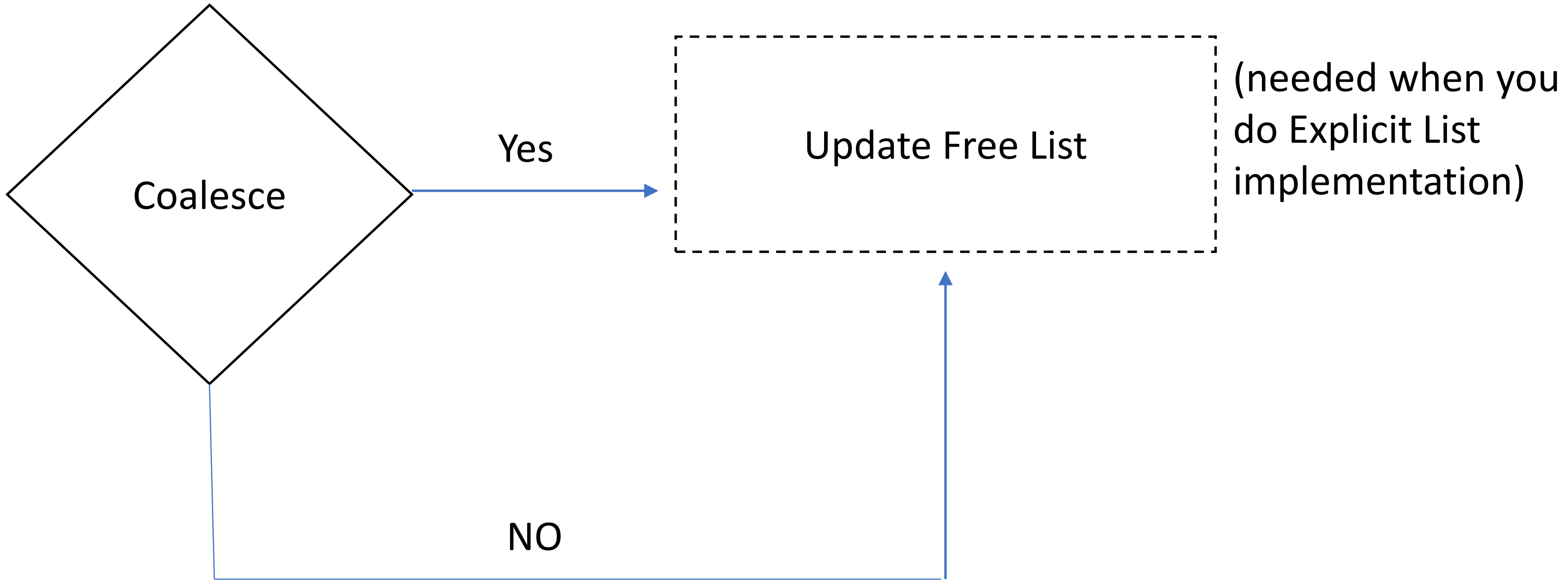
Coalescing



Where else do you need to Coalesce?

- **IMPORTANT REMINDER**
  - No Two Adjacent Blocks should be free.
    - If adjacent blocks are free, coalesce them

mm_free(void *p)



Debugging

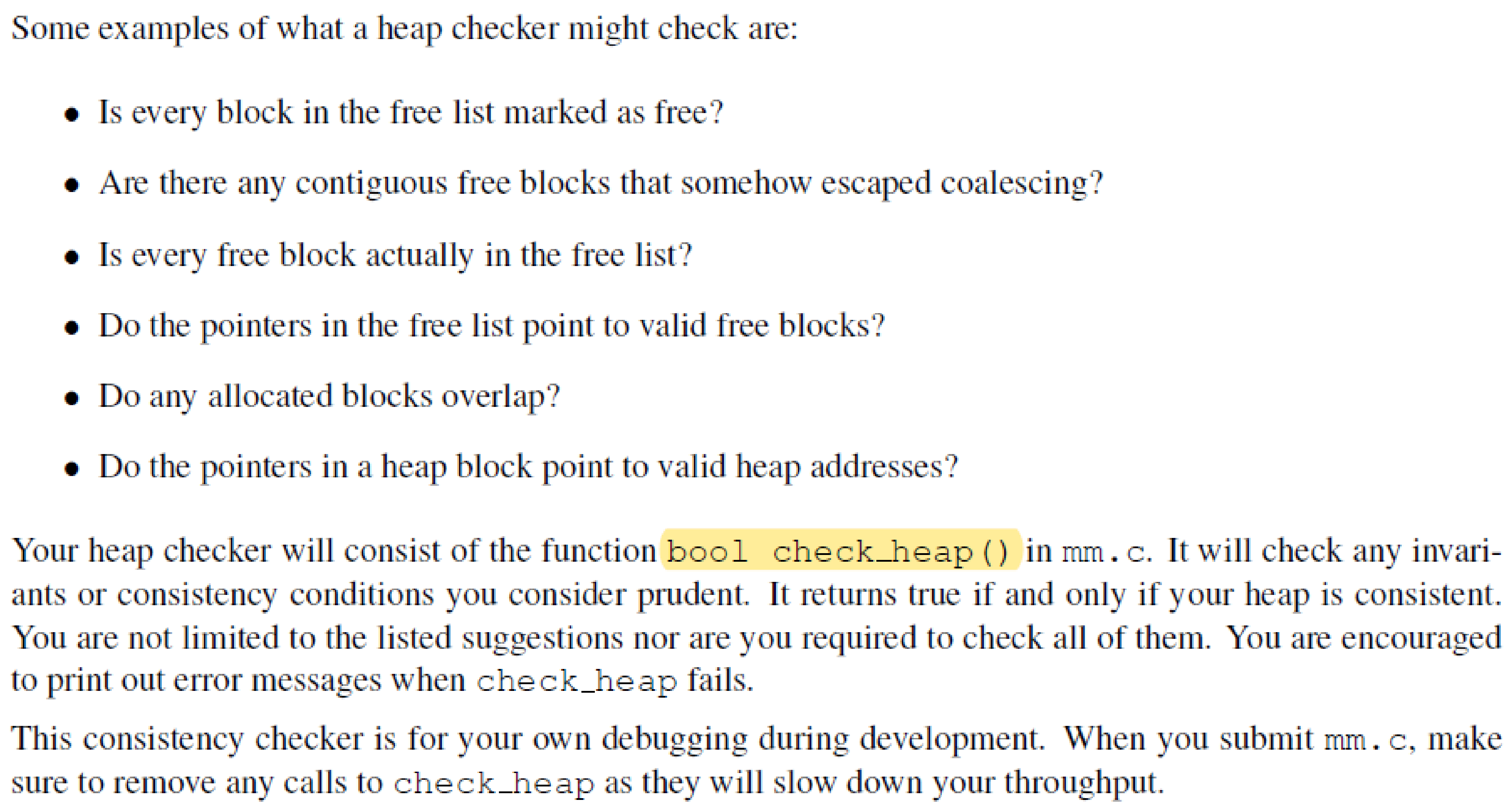
examine_heap()

- Already implemented
  - Prints content of the heap

check_heap()

- Partially Implemented
  - Left for you to implement as per need
  - Optional. But Highly Recommended
- **Write the check_heap() as you go, do not write it at the end.**
- Consider using a **macro** to turn the calls to check_heap() and/or examine_heap() on or off
- Can make use of a built-in macro called **`__LINE__`** that gets replaced with the current line number in the source file
  - You can use this with your check_heap() to indicate where the check_heap() call failed or gave an error.

Some ideas of check_heap()



Some examples of what a heap checker might check are:

- Is every block in the free list marked as free?
- Are there any contiguous free blocks that somehow escaped coalescing?
- Is every free block actually in the free list?
- Do the pointers in the free list point to valid free blocks?
- Do any allocated blocks overlap?
- Do the pointers in a heap block point to valid heap addresses?

Your heap checker will consist of the function `bool check_heap()` in `mm.c`. It will check any invariants or consistency conditions you consider prudent. It returns true if and only if your heap is consistent. You are not limited to the listed suggestions nor are you required to check all of them. You are encouraged to print out error messages when `check_heap` fails.

This consistency checker is for your own debugging during development. When you submit `mm.c`, make sure to remove any calls to `check_heap` as they will slow down your throughput.

More Ideas for check_heap()

- Block level:
  - Header and footer match.
  - Payload area is aligned.
- List level:
  - Next/prev pointers in consecutive free blocks are consistent.
  - Free list contains no allocated blocks.
  - All free blocks are in the free list.
  - No contiguous free blocks in memory (unless you defer coalescing).
  - There are no cycles in the list (unless you use circular lists).
  - Segregated list contains only blocks that belong to the size class
- Heap level:
  - Prologue/Epilogue blocks are at the boundaries and have special size/alloc fields.
  - All blocks stay in between the heap boundaries.
- And your own invariants (e.g. address order)

http://www.cs.cmu.edu/afs/cs/academic/class/15213-f13/www/recitations/rec12_marjorie.pdf

How to use a macro to turn on/off *check_heap*??

- Turning it ON

```
#define check_heap_actual() check_heap();
//#define check_heap_actual();
..
..
mm_free()
{     ..
      ..
      check_heap_actual();
}
```

How to use a macro to turn on/off *check_heap*??

- Turning it OFF

```
//#define check_heap_actual() check_heap();
#define check_heap_actual();
..
..
mm_free()
{     ..
      ..
      check_heap_actual();
}
```

Important Points

- Make sure to turn OFF calls to check_heap() and examine_heap() before your final submission and when you are testing the performance of your code
  - Otherwise, the performance of your code will decrease because of the time taken to execute these functions.

- Turn ON calls to check_heap() and examine_heap() ONLY when you are debugging your code.