Malloc Final Bootcamp

Caleb, Miles, Parth

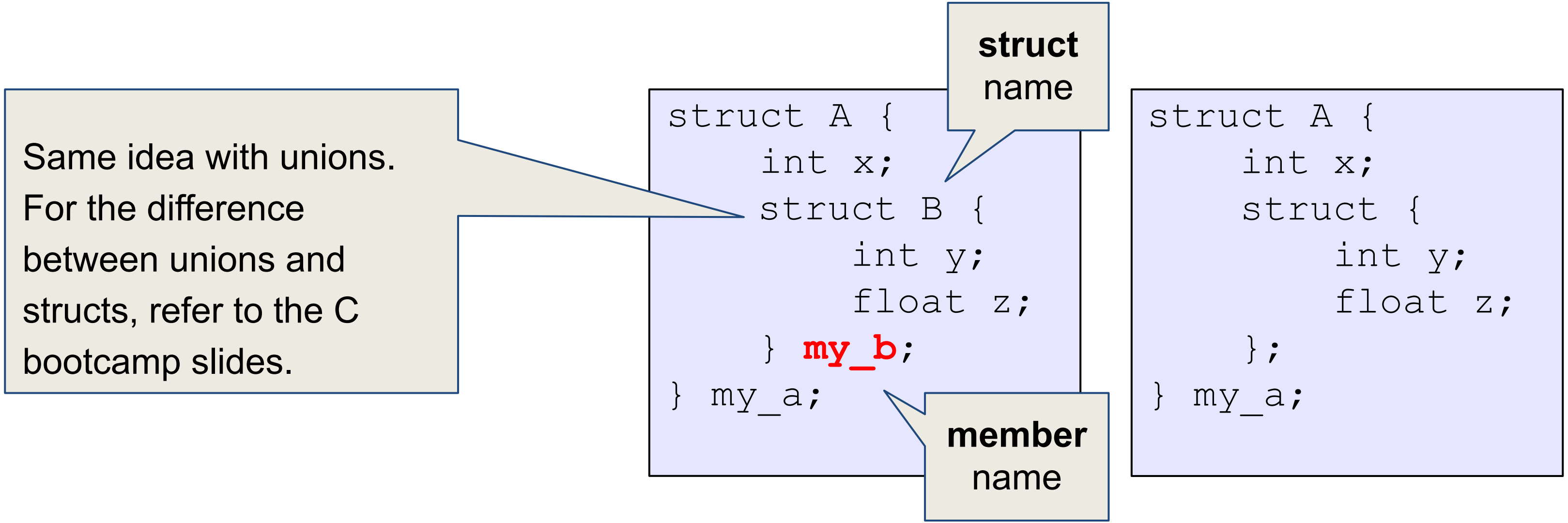
March 17th, 2024

Agenda

- Reminders about structs/unions
- Modularity and Design
- Increasing Utilization
  - Eliminating footers
  - Decreasing minimum block size
  - Other improvements
- Asking for Help
- Appendix

Conceptual Outline

Anonymous Structs/Unions



Same idea with unions. For the difference between unions and structs, refer to the C bootcamp slides.

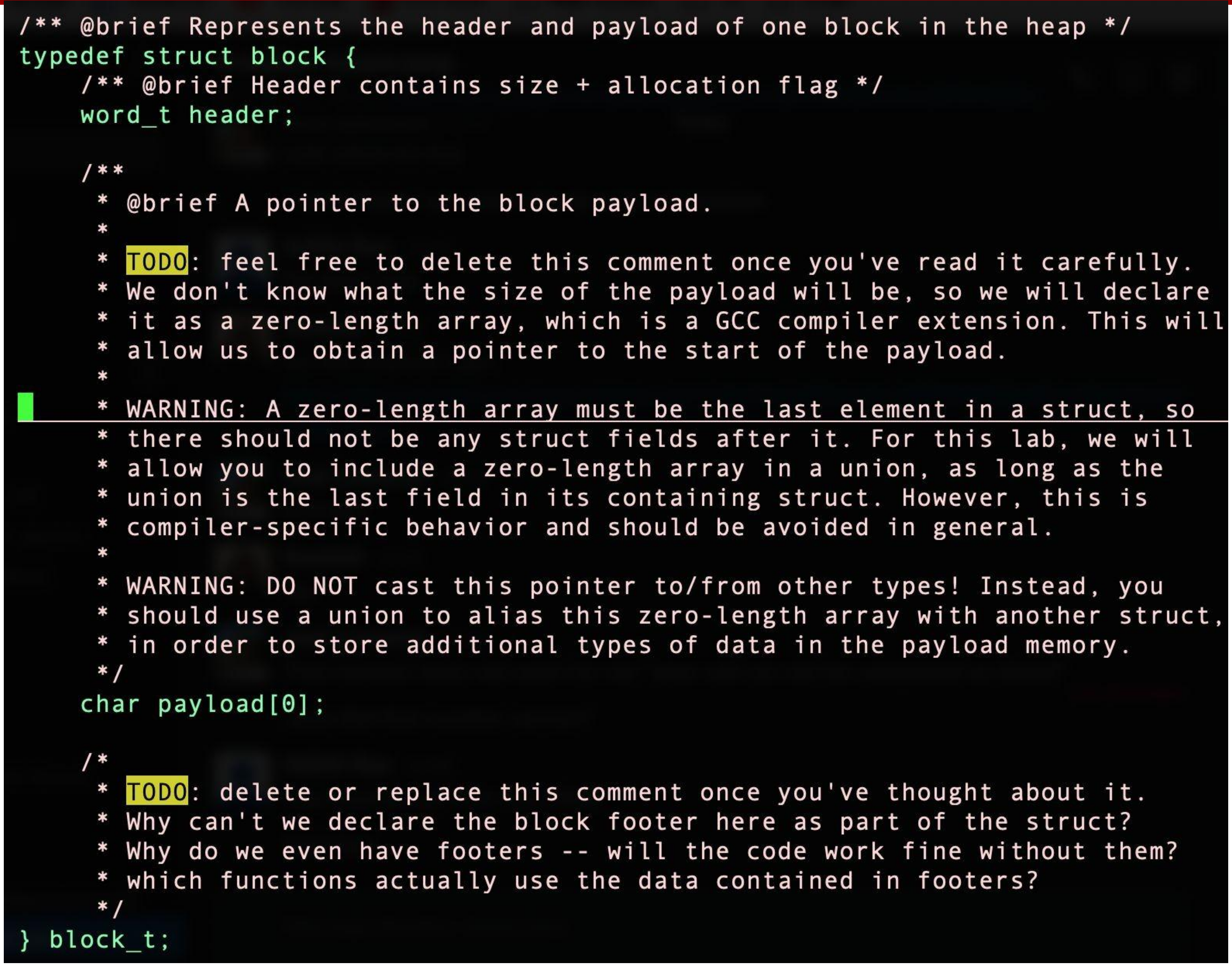
```
struct A {
     int x;
     struct B {
          int y;
          float z;
     } my_b;
} my_a;
```

struct name

member name

```
struct A {
     int x;
     struct {
          int y;
          float z;
     };
} my_a;
```

- What is the type of `x`? **`int`**
- How do we access `x` of `my_a`? **`my_a.x`**
- What is the type of `my_b`? **`struct B`**
- How do we access `y` of `my_a`? **`my_a.my_b.y`** **`my_a.y`**



```
/** @brief Represents the header and payload of one block in the heap */
typedef struct block {
    /** @brief Header contains size + allocation flag */
    word_t header;

    /**
     * @brief A pointer to the block payload.
     *
     * TODO: feel free to delete this comment once you've read it carefully.
     * We don't know what the size of the payload will be, so we will declare
     * it as a zero-length array, which is a GCC compiler extension. This will
     * allow us to obtain a pointer to the start of the payload.
     *
     * WARNING: A zero-length array must be the last element in a struct, so
     * there should not be any struct fields after it. For this lab, we will
     * allow you to include a zero-length array in a union, as long as the
     * union is the last field in its containing struct. However, this is
     * compiler-specific behavior and should be avoided in general.
     *
     * WARNING: DO NOT cast this pointer to/from other types! Instead, you
     * should use a union to alias this zero-length array with another struct,
     * in order to store additional types of data in the payload memory.
     */
    char payload[0];

    /*
     * TODO: delete or replace this comment once you've thought about it.
     * Why can't we declare the block footer here as part of the struct?
     * Why do we even have footers -- will the code work fine without them?
     * which functions actually use the data contained in footers?
     */
} block_t;
```

Zero-Length Arrays

```
struct line {
    int length;
    char contents[0];
};

int main() {
    struct line my_line;
    printf(“sizeof(contents) = %zu\n”, sizeof(L.contents)); // 0
    printf(“sizeof(struct line) = %zu\n”, sizeof(struct line)); // 4
}
```

- It’s a GCC extension - not part of the C specification!
- Must be at the end of a struct
  - Can be a member of a union that’s at the end of a struct
- **`sizeof`** on a zero-length array returns zero
- But, at runtime, the zero-length array expands to fill any space after the struct
  - `struct line *l = malloc(sizeof(struct line) + 23);`
  - Can use `l->contents[0]` through `l->contents[22]`

Time Management

- Labs in this course are NOT meant to be done in one sitting
  - If one of the TAs or faculty sat down to redo this lab from scratch, it would still take them at least a week
- Plan ahead, leave plenty of time for **design**
  - Measure twice, cut once
- Work in small chunks of time
  - One or two hours, then take a break
  - Your brain can keep working subconsciously
  - Leave time for “aha!” moments

Modularity and Design

- Good style shouldn’t be an afterthought
  - If you can read your own code it’s easier to debug
  - It will make it easier to explain to students when you become a TA later :)
- Suggestions:
  - Avoid long if-else chains (could you be using a loop?)
  - Think carefully about how much work each function should do
  - Use structs and unions to minimize pointer arithmetic
  - Dedicate a few helper functions to capture all of the pointer arithmetic
- Descriptive file header comment explaining your block structure
- Descriptive function header comments
- Comment as you go!
  - Not just for style points, you’ll get confused too

Quick Example of Good and Bad Style

```
static const size_t
bucket_sizes[N_BUCKETS] = {
   // (some numbers)
};

static size_t
get_bucket_size(int bucket) {
    for (int i = 0; i < N_BUCKETS; i++) {
        if (i == bucket) {
            return bucket_sizes[i];
        }
    }
    return 0;
}
```

Quick Example of Good and Bad Style

```
static const size_t
bucket_sizes[N_BUCKETS] = {
   // (some numbers)
};

static size_t
get_bucket_size(int bucket) {
    for (int i = 0; i < N_BUCKETS; i++) {
        if (i == bucket) {
            return bucket_sizes[i];
        }
    }
    return 0;
}
```

```
static const size_t
bucket_sizes[N_BUCKETS] = {
   // (some numbers)
};

static size_t
get_bucket_size(int bucket) {
    assert(bucket >= 0 && bucket < N_BUCKETS);
    return bucket_sizes[bucket];
}
```

Quick Example of Good and Bad Style

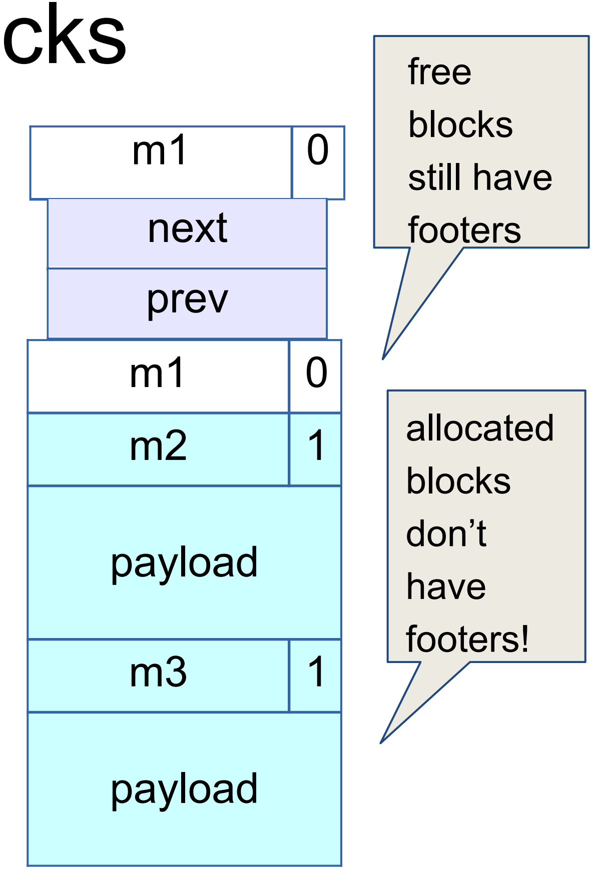
```
/**
 * Array of bucket sizes.
 */
static const size_t
bucket_sizes[N_BUCKETS] = {
   // (some numbers)
};
```

```
/**
 * “Bucket” sizes for free lists.
 * Free list `i` holds free blocks whose
 * allocated size is <= `bucket_size[i]`
 * but >= `bucket_size[i-1]`.
 * (Notionally, `bucket_size[-1]` is zero.)
 */
static const size_t
bucket_sizes[N_BUCKETS] = {
   // (some numbers)
};
```

Eliminate footers in allocated blocks

Reduces internal fragmentation (increases utilization)

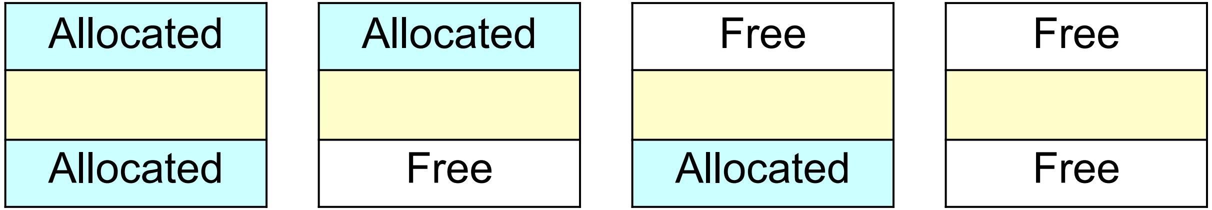
- Why do we need footers?
  - Coalescing blocks
  - What kind of blocks do we coalesce?
- Do we need to know the size of a block if we're not going to coalesce it?
- Based on that idea, can you design a method that helps you determine when to coalesce?
  - Hint: where could you store a little **bit** of extra information for each block?


m1
0
next
prev
m1
0
m2
1
payload
m3
1
payload
free blocks still have footers
allocated blocks don't have footers!

Coalescing Memory

- Combine adjacent blocks if both are free

- Four cases:



Allocated

Allocated

Allocated

Free

Free

Allocated

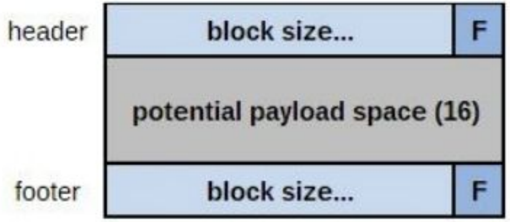
Free

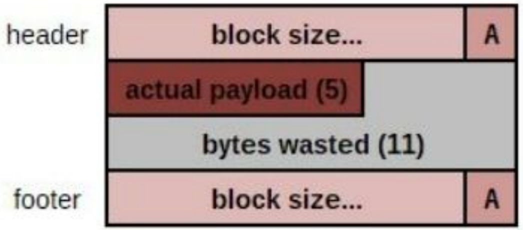
Free

- **footerless**: if free, use footer like before

Decrease the minimum block size

- Reduces internal fragmentation (increase utilization)
- Currently, min block size is 32.
  - 8 byte header
  - 16 byte payload (or 2 8 byte pointers for free)
  - 8 byte footer
- If you just need to malloc(5), and the payload size is 16 bytes, you waste 11 bytes.
- Must manage free blocks that are too small to hold the pointers for a **doubly** linked free list





Decrease the minimum block size

HINT: Your minimum block size should be 16 in order to get a 100 on final, meaning you only keep 2 of the fields that we had before. We already removed footers, so remove one more!

- Remove Prev pointer
- Remove Next pointer
- Remove Header

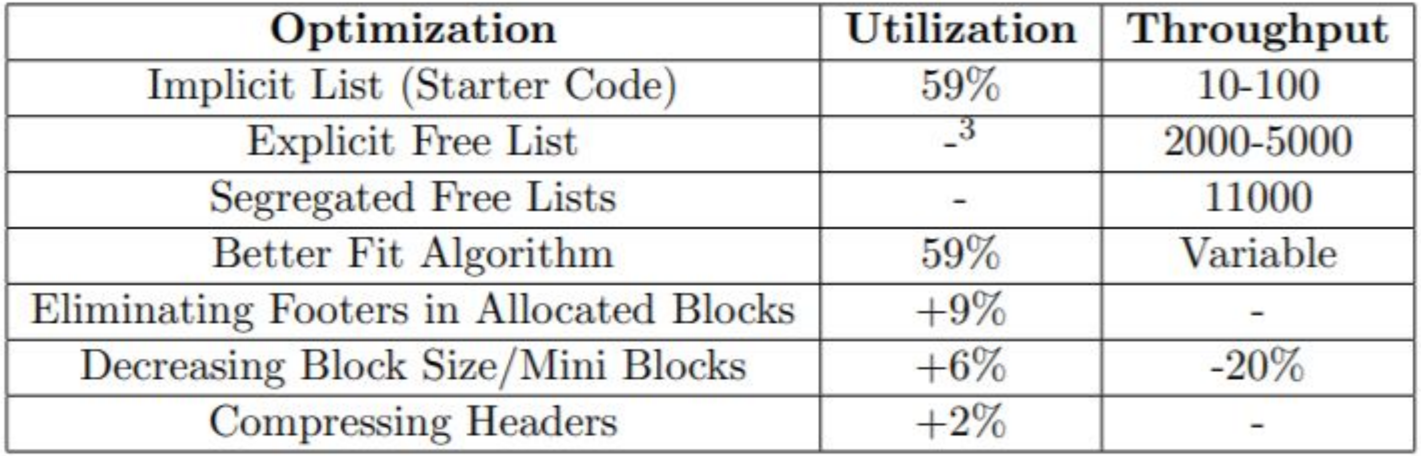
Small utilization improvements

- Insertion Policy
    - LIFO (last-in-first-out) vs FIFO (first-in-first-out)
- Segregated List Buckets
    - Potentially reconsider size classes (only 128 bytes for global variables)
        - Diminishing returns
        - Adjust buckets based trace files (please don’t hard code)
- Chunksize
    - Potentially reconsider smaller size
- Fit Algorithm
    - First-fit
    - Best-fit (which segregated list approximates)
    - Better Fit (ex. search for the next 20 blocks after finding a fit)

How to Ask for Help

- Be specific about what the problem is, and how to cause it
  - **BAD:** “My program segfaults.”
  - **GOOD:** “I ran mdriver in gdb and it says that a segfault occurred due to an invalid next pointer, so I set a watchpoint on the segfaulting next pointer. How can I figure out what happened?”
  - **GOOD:** “My heap checker indicates that my segregated list has a block of the wrong size in it after performing a coalesce(). Why might that be the case?”
  - What sequence of events do you expect around the time of the error? What part of the sequence has already happened?
- Have you written your mm_checkheap function, and is it working?
  - We **WILL** ask to see it!
- Use a rubber duck!

Ways to Improve



Optimization	Utilization	Throughput
Implicit List (Starter Code)	59%	10-100
Explicit Free List	-[3]	2000-5000
Segregated Free Lists	-	11000
Better Fit Algorithm	59%	Variable
Eliminating Footers in Allocated Blocks	+9%	-
Decreasing Block Size/Mini Blocks	+6%	-20%
Compressing Headers	+2%	-

source: writeup

If You Get Stuck

- ***Please read the writeup!***
  - CS:APP Chapter 9
  - View lecture notes and course FAQ at http://www.cs.cmu.edu/~213
  - Post a **private** question on Piazza

Debugging: GDB & The Almighty Heap Checker

When your scattered print statements
don't reveal where the error is

What’s better than printf? Using GDB

- Use GDB to determine where segfaults happen!
- **gdb mdriver** will open the malloc driver in gdb
  - Type run and your program will run until it hits the segfault!
- **step/next** - (abbrev. **s/n**) step to the next line of code
  - **next** steps over function calls
- **finish** - continue execution until end of current function, then break
- **print <expr>** - (abbrev. **p**) Prints **any C-like expression** (including results of function calls!)
  - Consider writing a heap printing function to use in GDB!
- **x <expr>** - Evaluate <expr> to obtain address, then examine memory at that address
  - **x /a <expr>** - formats as address
  - See **help p** and **help x** for information about more formats

Using GDB - Fun with frames

- **backtrace** - (abbrev. **bt**) print call stack up until current function
  - **backtrace full** - (abbrev. **bt full**) print local variables in each frame

```
(gdb) backtrace
#0  find_fit (...)
#1  mm_malloc (...)
#2  0x0000000000403352 in eval_mm_valid
(...)       #3 run_tests (...)
#4  0x0000000000403c39 in main (...)
```

- **frame 1** - (abbrev. **f 1**) switch to mm_malloc’s stack frame
  - Good for inspecting local variables of calling functions

Using GDB - Setting breakpoints/watchpoints

- **break mm_checkheap** - (abbrev. **b**) break on “mm_checkheap()”
  - **b mm.c:25** - break on line 25 of file “mm.c” - **very useful!**
- **b find_fit if size == 24** - break on function “find_fit()” if the local variable “size” is equal to 24 - “**conditional breakpoint**”

- **watch heap_listp** - (abbrev. **w**) break if value of “heap_listp” changes - “**watchpoint**”
- **w block == 0x80000010** - break if “block” is equal to this value
- **w *0x15213** - watch for changes at memory location 0x15213
  - Can be *very* slow

- **rwatch <thing>** - stop on reading a memory location
- **awatch <thing>** - stop on *any* memory access

Heap Checker

- int mm_checkheap(int verbose);
- critical for debugging
  - **write this function early!**
  - update it when you change your implementation
  - check all heap invariants, make sure you haven't lost track of any part of your heap
    - check should pass if and only if the heap is truly well-formed
  - should only generate output if a problem is found, to avoid cluttering up your program's output
- meant to be correct, **not** efficient
- call before/after major operations **when the heap should be well-formed**

Heap Invariants (**Non-Exhaustive**)

- **Block Level**
  - header and footer match
  - payload area is aligned, size is valid
  - no contiguous free blocks unless you defer coalescing
- **Explicit/Segregated List Level**
  - next/prev pointers in consecutive free blocks are consistent
  - no allocated blocks in free list, all free blocks are in the free list
  - no cycles in free list unless you use a circular list
  - each segregated list contains only blocks in the appropriate size class
- **Heap Level**
  - all blocks between heap boundaries, correct sentinel blocks (if used)

Internal Fragmentation

- Occurs when the *payload* is smaller than the block size
  - due to alignment requirements
  - due to management overhead
  - as the result of a decision to use a larger-than-necessary block
- Depends on the current allocations, i.e. the pattern of *previous* requests

External Fragmentation

- Occurs when the total free space is sufficient, but no single free block is large enough to satisfy the request
- Depends on the pattern of *future* requests
  - thus difficult to predict, and any measurement is at best an estimate
- Less critical to malloc traces than internal fragmentation

C: Pointer Arithmetic

- Adding an integer to a pointer is different from adding two integers
- The value of the integer is always multiplied by the size of the type that the pointer points at
- Example:
  - type_a *ptr = ...;
  - type_a *ptr2 = ptr + a;
- is really computing
  - ptr2 = ptr + (a * sizeof(type_a));
  - lea(ptr, a, sizeof(type_a)), ptr2
- Pointer arithmetic on void* is undefined (what's the size of a void?)

C: Pointer Arithmetic

-
  ```
  int *ptr = (int*)0x152130;
  int *ptr2 = ptr + 1;
  ```

-
  ```
  char *ptr = (char*)0x152130;
  char *ptr2 = ptr + 1;
  ```

-
  ```
  char *ptr = (char*)0x152130;
  void *ptr2 = ptr + 1;
  ```

-
  ```
  char *ptr = (char*)0x152130;
  char *p2 = ((char*)(((int*)ptr)+1));
  ```

C: Pointer Arithmetic

-
  ```
  int *ptr = (int*)0x152130;
  int *ptr2 = ptr + 1;
  ```
  ptr2 is 0x152134

-
  ```
  char *ptr = (char*)0x152130;
  char *ptr2 = ptr + 1;
  ```
  ptr2 is 0x152131

-
  ```
  char *ptr = (char*)0x152130;
  void *ptr2 = ptr + 1;
  ```
  ptr2 is **still** 0x152131

-
  ```
  char *ptr = (char*)0x152130;
  char *p2 = ((char*)(((int*)ptr)+1));
  ```
  p2 is 0x152134

C: Pointer Casting

- Notation: `(b*)a` "casts" `a` to be of type `b*`
- Casting a pointer doesn't change the bits!
  - `type_a *ptr_a=...; type_b *ptr_b=(type_b*)ptr_a;` makes ptr_a and ptr_b contain identical bits
- But it does change the behavior when dereferencing
  - because we *interpret* the bits differently
- Can cast `type_a*` to long/unsigned long and back
  - pointers are really just 64-bit numbers
  - such casts are important for malloclab
  - but be careful – this can easily lead to hard-to-find errors

Debugging Tip: Using the Preprocessor

- Use conditional compilation with #if or #ifdef to easily turn debugging code on or off

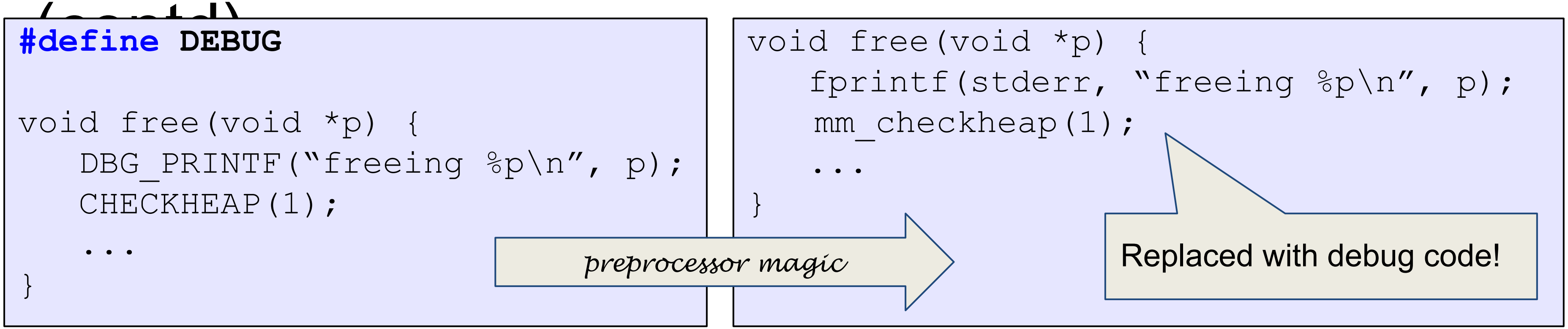
```
#ifdef DEBUG
#define DBG_PRINTF(...) fprintf(stderr,     VA_ARGS     )
#define CHECKHEAP(verbose) mm_checkheap(verbose)

#else
#define DBG_PRINTF(...)
#define CHECKHEAP(verbose)
#endif /* DEBUG */
```

```
// comment line below to disable debug code!
#define DEBUG

void free(void *p) {
   DBG_PRINTF("freeing %p\n", p);
   CHECKHEAP(1);
   ...
}
```

Debugging Tip: Using the Preprocessor (contd)



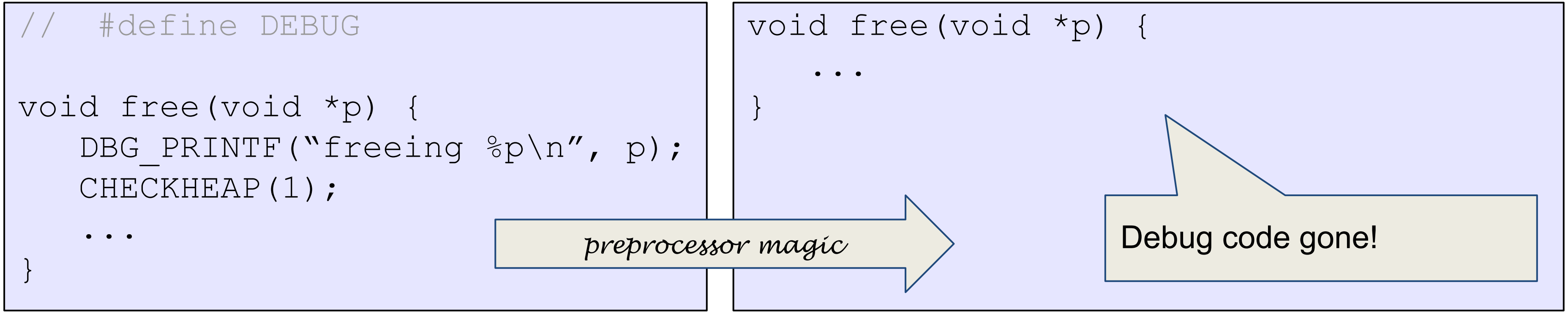
```
#define DEBUG

void free(void *p) {
   DBG_PRINTF("freeing %p\n", p);
   CHECKHEAP(1);
   ...
}
```

preprocessor magic

```
void free(void *p) {
   fprintf(stderr, "freeing %p\n", p);
   mm_checkheap(1);
   ...
}
```

Replaced with debug code!



```
//  #define DEBUG

void free(void *p) {
   DBG_PRINTF("freeing %p\n", p);
   CHECKHEAP(1);
   ...
}
```

preprocessor magic

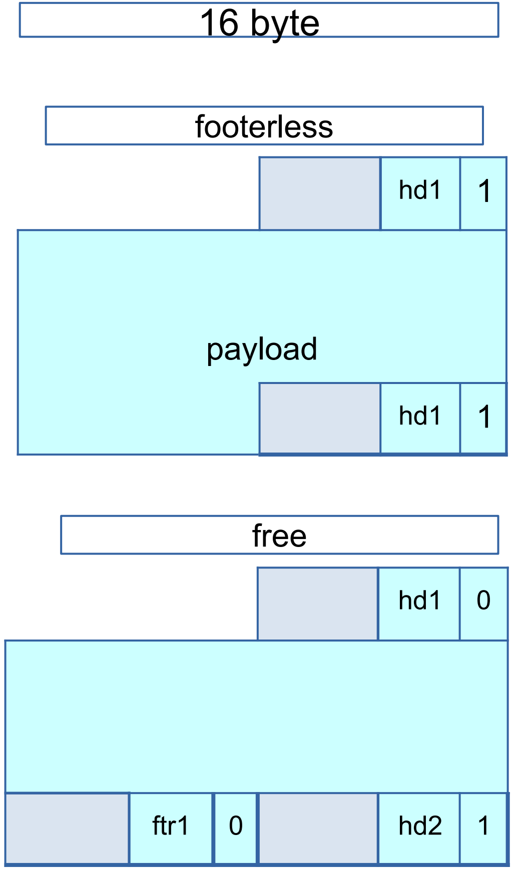
```
void free(void *p) {
   ...
}
```

Debug code gone!

Header Reduction

- **Note:** this is completely optional and generally **discouraged** due to its relative difficulty
  - Do **NOT** attempt unless you are satisfied with your implementation as-is

- When to use 8 or 4 byte header? (must support all possible block sizes)
- If 4 byte, how to ensure that payload is aligned?
- Arrange accordingly
- How to coalesce if 4 byte header block is followed by 8 byte header block?
- Store extra information in headers
- Can you do this with 2 byte headers instead?


16 byte
footerless
hd1
1
payload
hd1
1
free
hd1
0
ftr1
0
hd2
1