C Data Structures & Dynamic Mem

Introduction to Computer Systems, Fall 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Adam Gorka Daniel Gearhardt
Ash Fujiyama Emily Shen

TAs:

Ahmed Abdellah
Angie Cao
August Fu
Caroline Begg
Cathy Cao
Claire Lu
Eric Sungwon Lee
Ethan Weisberg
Garrett O'Malley Kirsch
Hassan Rizwan
Iain Li
Jerry Wang
Juan Lopez
Keith Mathe
Maya Huizar
Meghana Vasireddy
Perrie Quek
Sidharth Roy
Sydnie-Shea Cohen
Vivi Li
Yousef AlRabiah

Poll Everywhere

pollev.com/cis2400

- How are you? Any Questions from last lecture?

Upcoming Due Dates

- HW01 Due tomorrow
  - Can always ask for extensions if you want
  - Reminder: only need to do 3 of the 5 "rating 4" puzzles
- Check-in out tonight or tomorrow, due before lecture on Tuesday
- 1-on-1 from is live
- Recitation was recorded ☺

Lecture Outline

- **Structs Warm-up**
- The Heap
  - malloc() & free()
- Modules & Header Files
- Dynamic Memory Pitfalls
- GDB & Valgrind

Poll Everywhere

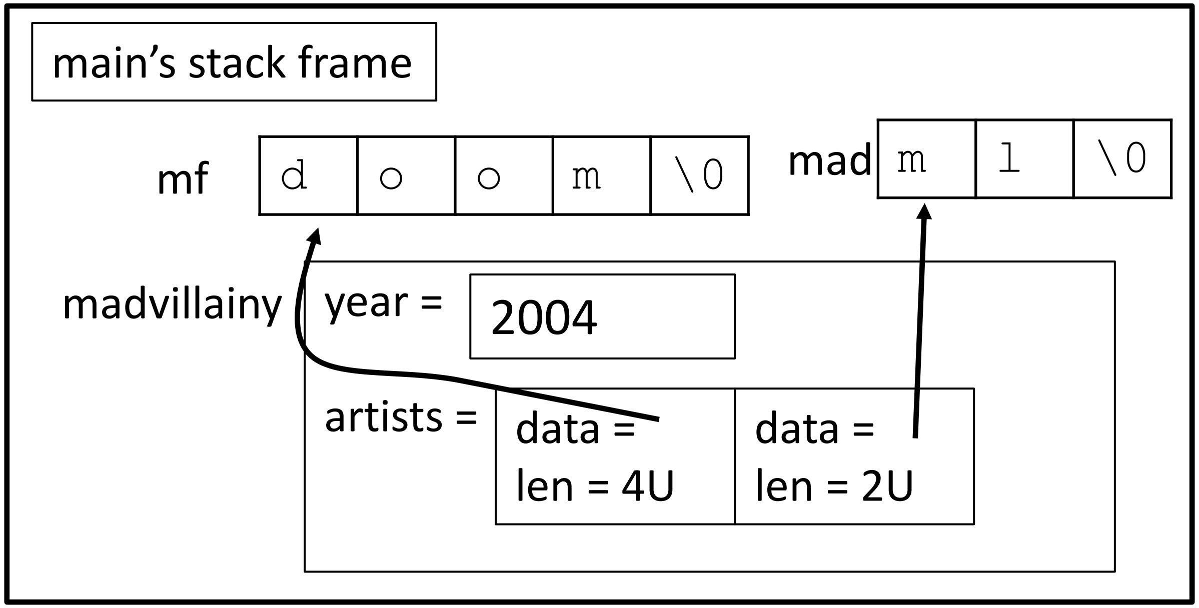
Discuss

- What's the state after calling remaster()?

```
typedef struct {
  char* data;
  unsigned int len;
} string;

typedef struct {
  int release_year;
  string artists[2];
} album;
```

```
void ALL_CAPS(string name) {
  for (int i = 0; i < name.len; i++) {
    name.data[i] = toupper(name.data[i]);
  }
}
void remaster(album *a) {
  album copy = *a;
  copy.release_year = 2025;
  ALL_CAPS(copy.artists[0]);
  copy.artists[1] = { copy.artists[0].data, 4U};
  a = &copy;
}
int main() {
  char mf[] = "doom";
  char mad[] = "ml";
  album madvillainy = (album) {
    .release_year = 2004,
    .artists = { {mf, 4U}, {mad, 2U} },
  };

  remaster(&madvillainy);
  // what is the state here?
}
```

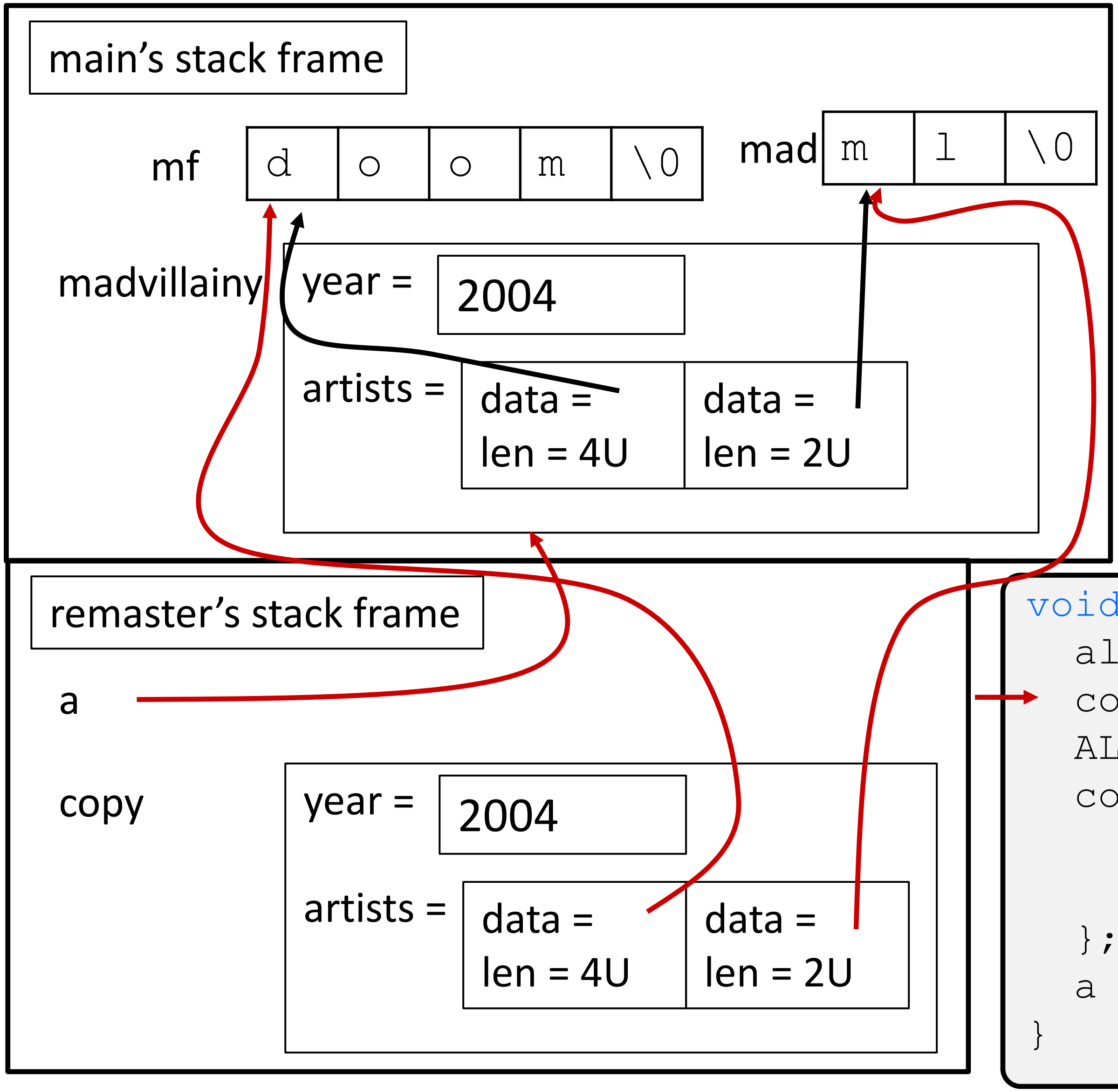
Visualization: Albums start



```
typedef struct {
  char* data;
  unsigned int len;
} string;

typedef struct {
  int release_year;
  string artists[2];
} album;
```

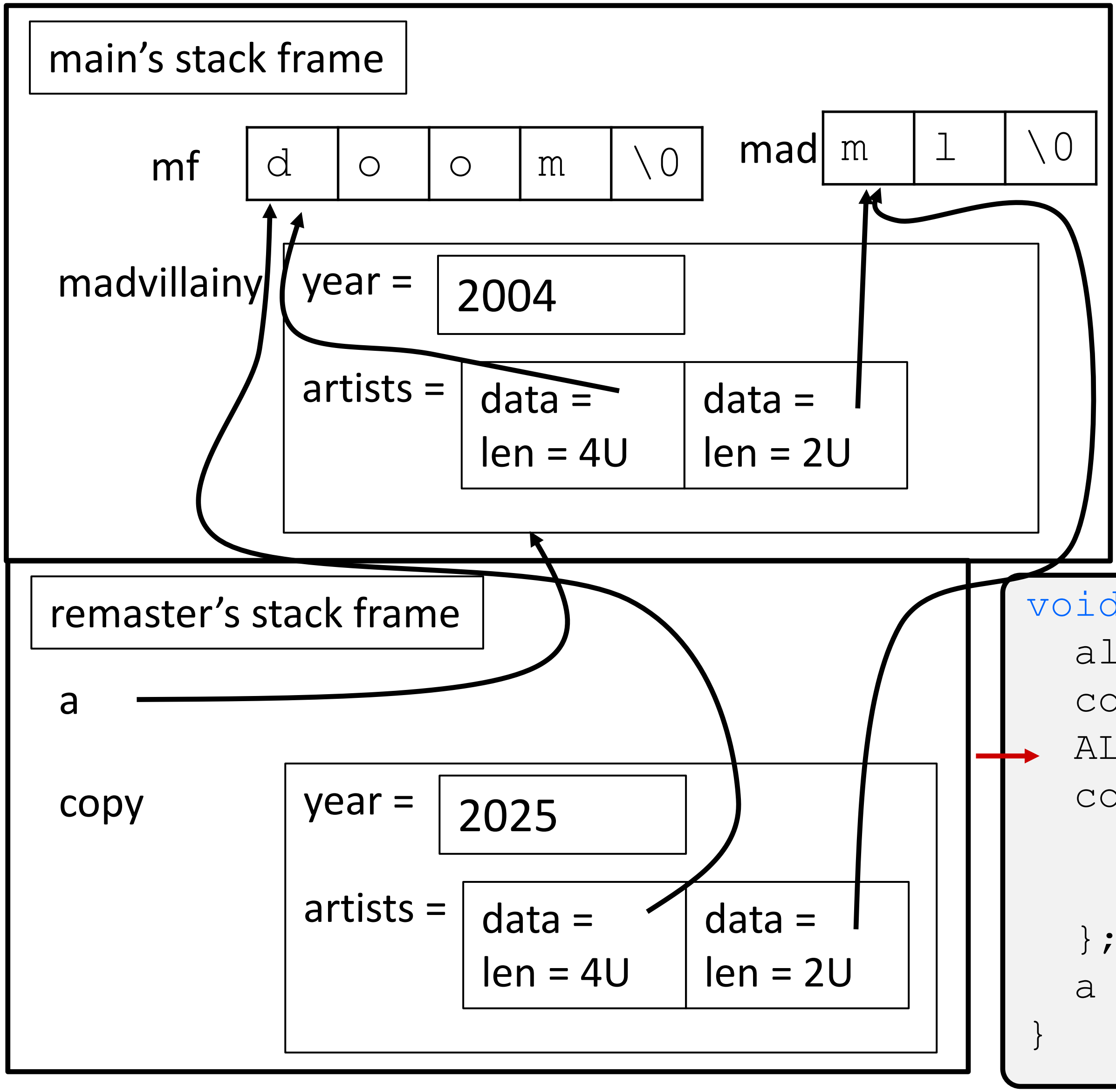
```
int main() {
  char mf[] = "doom";
  char mad[] = "ml";
  album madvillainy = (album) {
    .release_year = 2004,
    .artists = { {mf, 4U}, {mad, 2U} },
  };

  remaster(&madvillainy);
  // what is the state here?
}
```

Visualization: Albums start



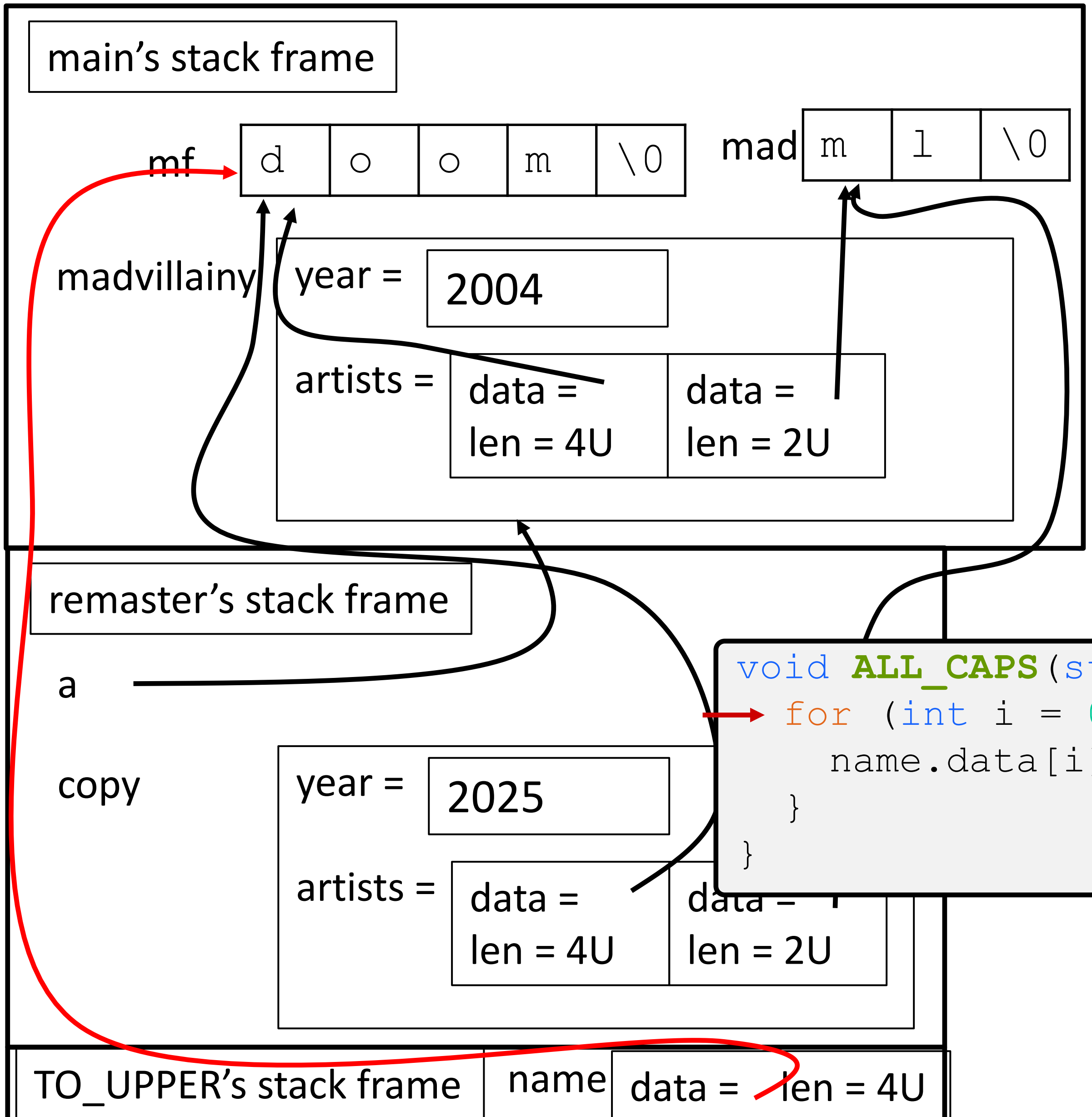
```
void remaster(album *a) {
  album copy = *a;
  copy.release_year = 2025;
  ALL_CAPS(copy.artists[0]);
  copy.artists[1] = {
    copy.artists[0].data,
    4U
  };
  a = &copy;
}
```

Visualization: Albums start



```
void remaster(album *a) {
  album copy = *a;
  copy.release_year = 2025;
  ALL_CAPS(copy.artists[0]);
  copy.artists[1] = {
    copy.artists[0].data,
    4U
  };
  a = &copy;
}
```

Visualization: Albums start



main's stack frame

mf `d` `o` `o` `m` `\0`

mad `m` `l` `\0`

madvillainy

year = 2004

artists = data = len = 4U | data = len = 2U

remaster's stack frame

a

copy

year = 2025

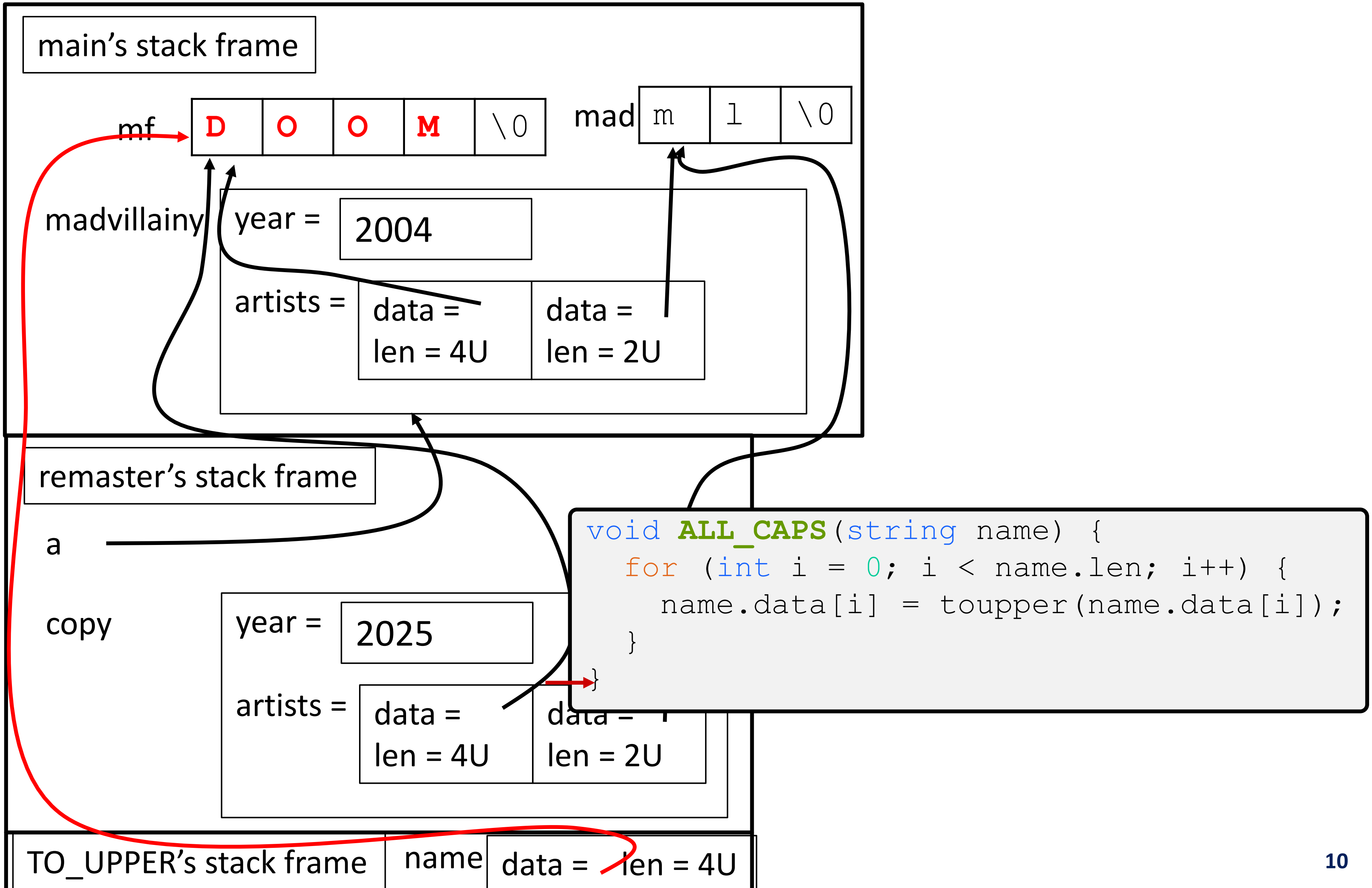
artists = data = len = 4U | data = len = 2U

TO_UPPER's stack frame

name data = len = 4U

```
void ALL_CAPS(string name) {
  for (int i = 0; i < name.len; i++) {
    name.data[i] = toupper(name.data[i]);
  }
}
```

Visualization: Albums start



main's stack frame

mf

D O O M \0

mad

m l \0

madvillainy

year = 2004

artists = data = len = 4U | data = len = 2U

remaster's stack frame

a

copy

year = 2025

artists = data = len = 4U | data = len = 2U

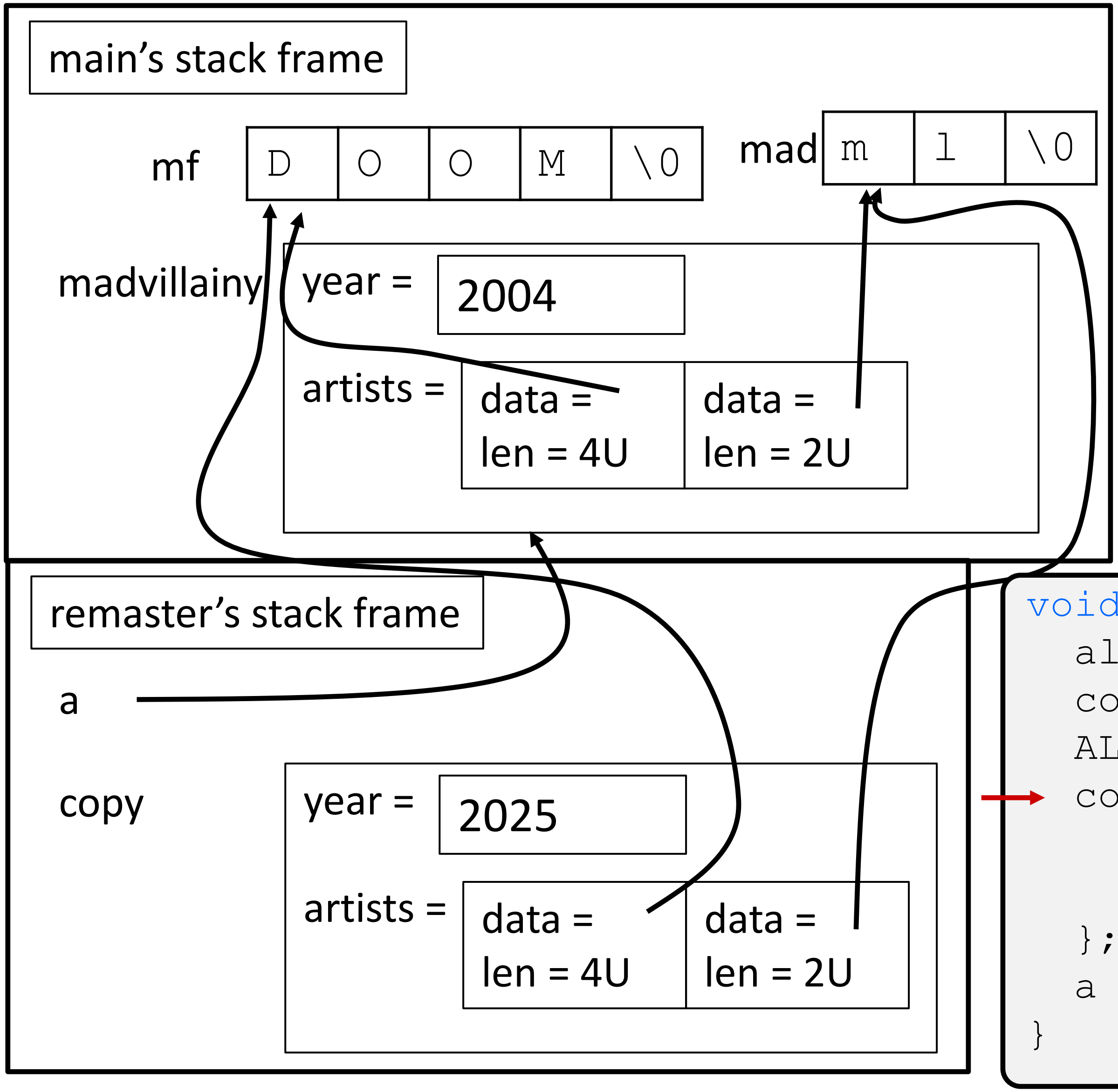
TO_UPPER's stack frame

name

data = len = 4U

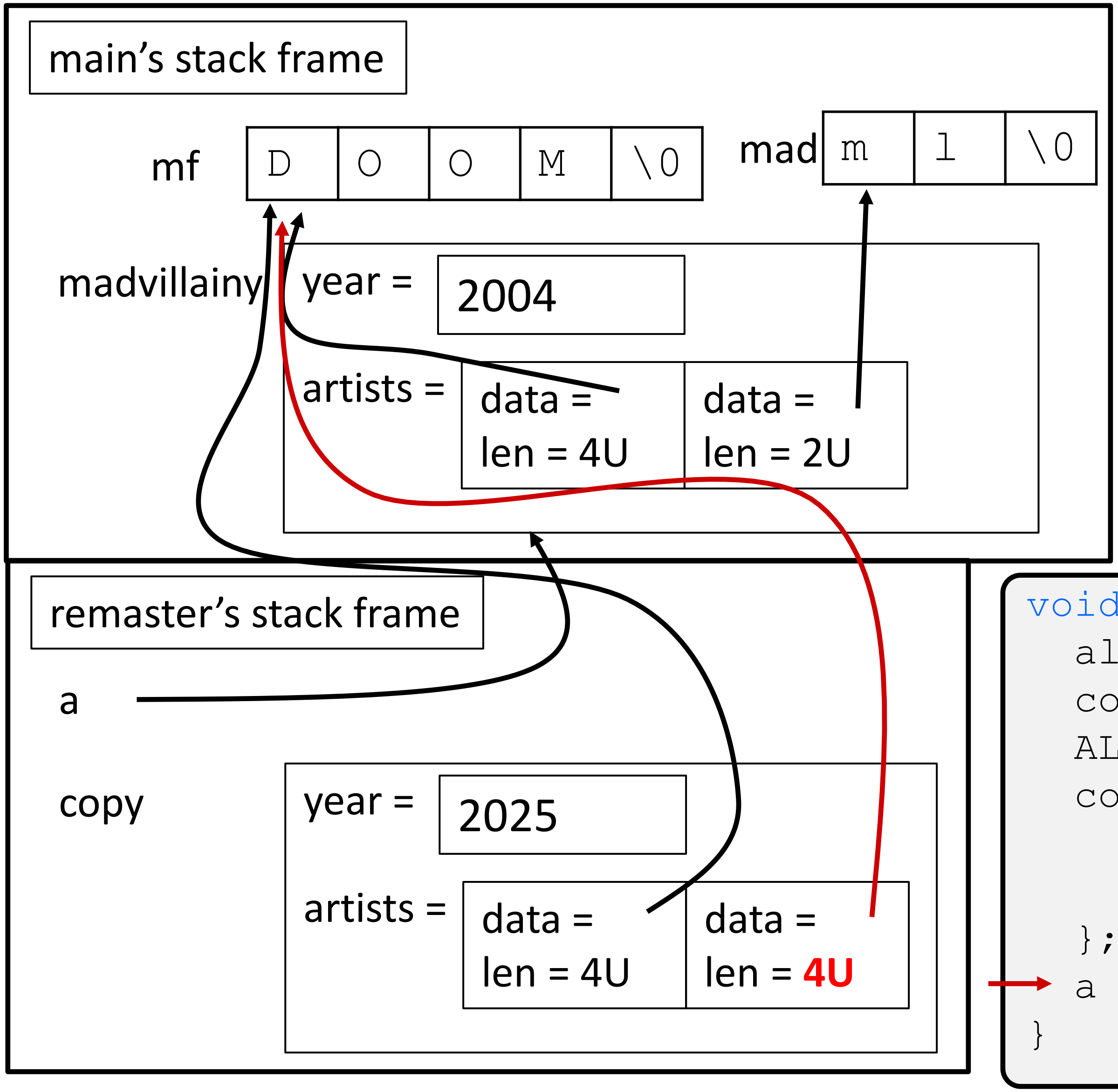
```
void ALL_CAPS(string name) {
  for (int i = 0; i < name.len; i++) {
    name.data[i] = toupper(name.data[i]);
  }
}
```

Visualization: Albums start



```
void remaster(album *a) {
  album copy = *a;
  copy.release_year = 2025;
  ALL_CAPS(copy.artists[0]);
→ copy.artists[1] = {
    copy.artists[0].data,
    4U
  };
  a = &copy;
}
```

Visualization: Albums start



main's stack frame

mf | D | O | O | M | \0

mad | m | l | \0

madvillainy | year = 2004

artists = | data = len = 4U | data = len = 2U

remaster's stack frame

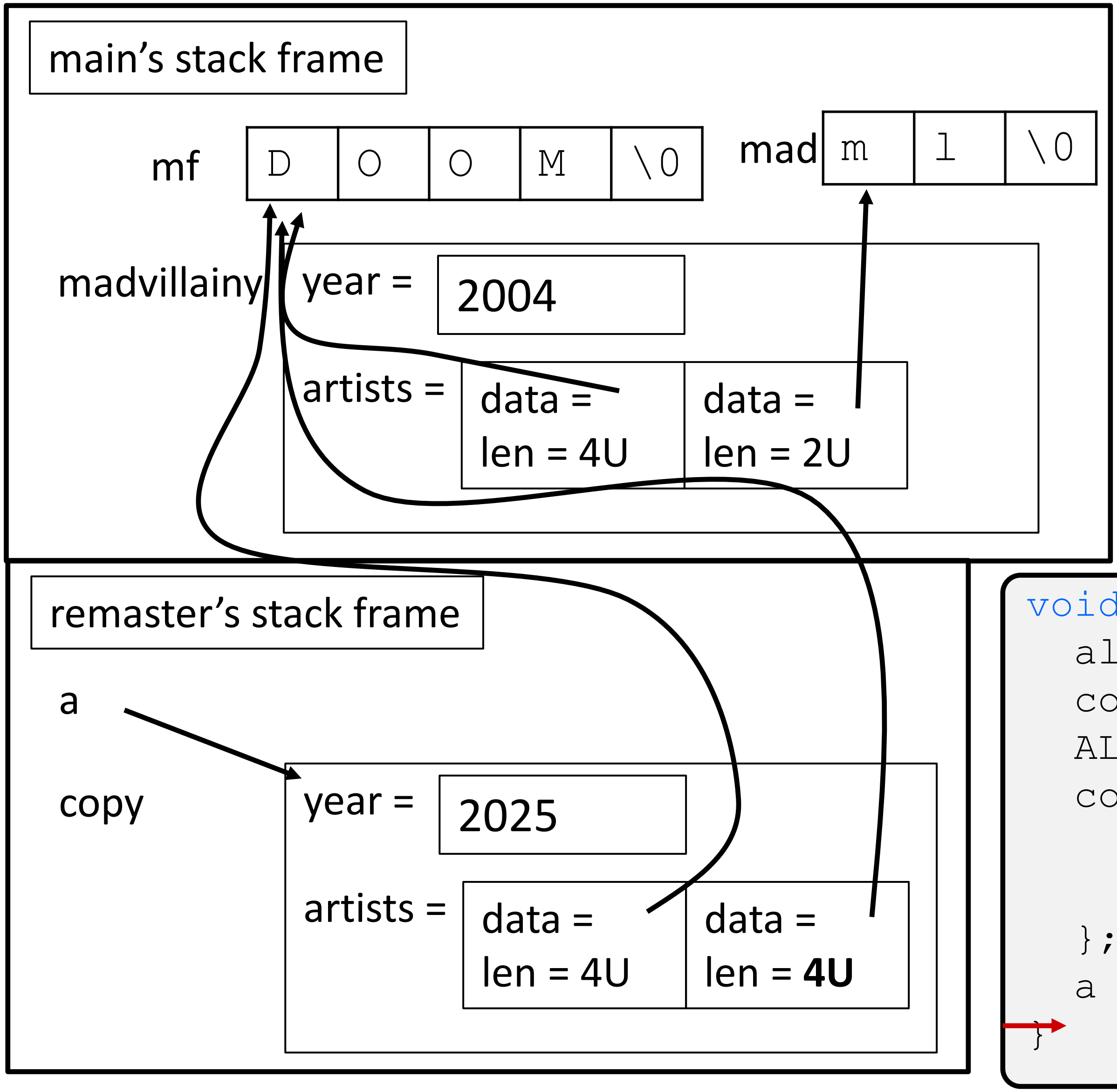
a

copy | year = 2025

artists = | data = len = 4U | data = len = **4U**

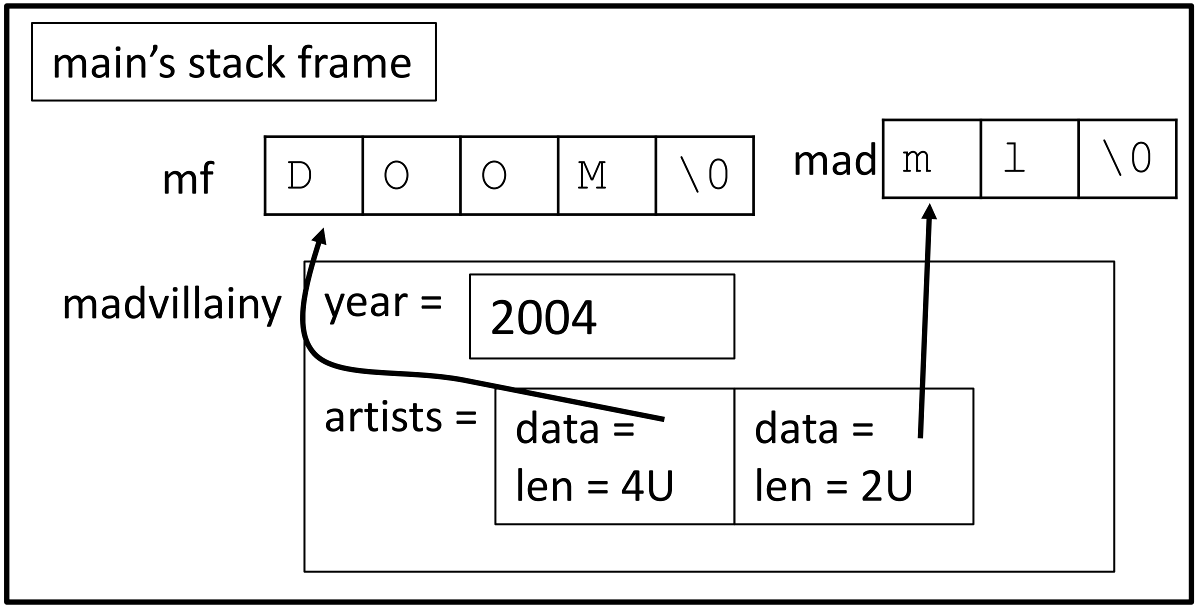
```
void remaster(album *a) {
  album copy = *a;
  copy.release_year = 2025;
  ALL_CAPS(copy.artists[0]);
  copy.artists[1] = {
    copy.artists[0].data,
    4U
  };
→ a = &copy;
}
```

Visualization: Albums start



```
void remaster(album *a) {
  album copy = *a;
  copy.release_year = 2025;
  ALL_CAPS(copy.artists[0]);
  copy.artists[1] = {
    copy.artists[0].data,
    4U
  };
  a = &copy;
}
```

Visualization: Albums start



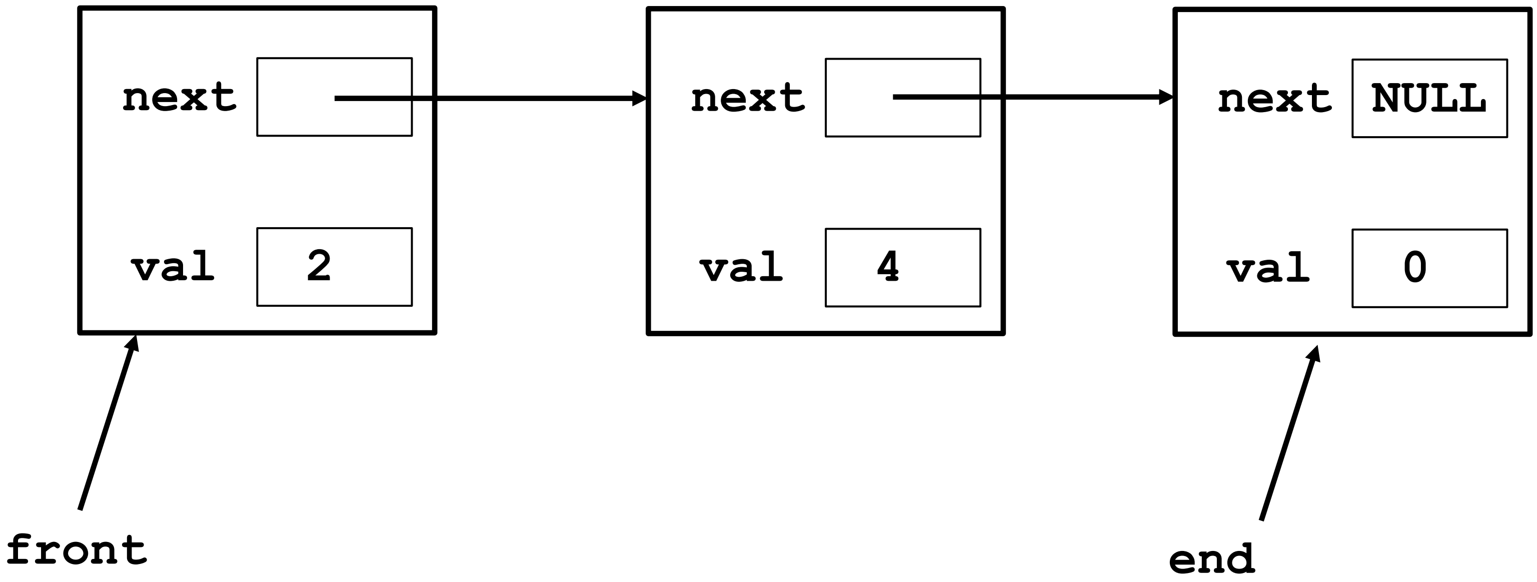
```
int main() {
  char mf[] = "doom";
  char mad[] = "ml";
  album madvillainy = (album) {
    .release_year = 2004,
    .artists = { {mf, 4U}, {mad, 2U} },
  };

  remaster(&madvillainy);
  // what is the state here?
}
```

Lecture Outline

- Structs Warm-up
- **The Heap**
  - **malloc() & free()**
- Modules & Header Files
- Dynamic Memory Pitfalls
- GDB & Valgrind

Queue Example

- Simple Data structure modeling a queue
  - Implemented with a singly linked list
- Items added to the end and removed from the front.
- We maintain a list of queue elements chained together with pointers.

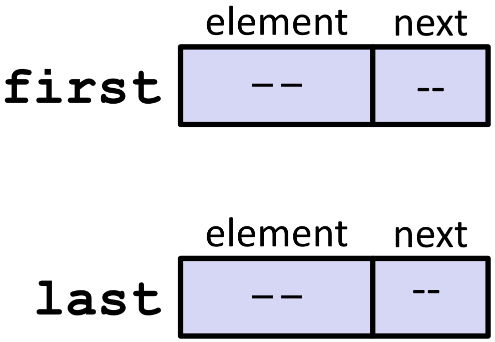


Queue Implementation Demo

- Let’s create a naïve implementation for our queue

```
#include <stdio.h>

typedef struct node_st {
  struct node_st* next;
  int val;
} Node;

int main(int argc, char** argv) {
  Node first, last;

  first.val = 2;
  first.next = &last;
  last.val = 0;
  last.next = NULL;
  return 0;
}
```

naive_queue.c

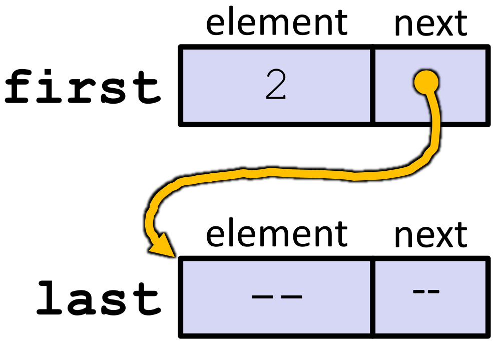


Queue Implementation Demo

- Let’s create a naïve implementation for our queue

```
#include <stdio.h>

typedef struct node_st {
  struct node_st* next;
  int val;
} Node;

int main(int argc, char** argv) {
  Node first, last;

  first.val = 2;
  first.next = &last;
  last.val = 0;
  last.next = NULL;
  return 0;
}
```

naive_queue.c

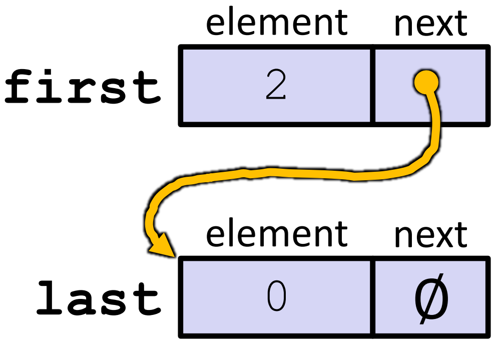


Queue Implementation Demo

- Let's create a naïve implementation for our queue

```
#include <stdio.h>

typedef struct node_st {
  struct node_st* next;
  int val;
} Node;

int main(int argc, char** argv) {
  Node first, last;

  first.val = 2;
  first.next = &last;
  last.val = 0;
  last.next = NULL;
  return 0;
}
```

naive_queue.c

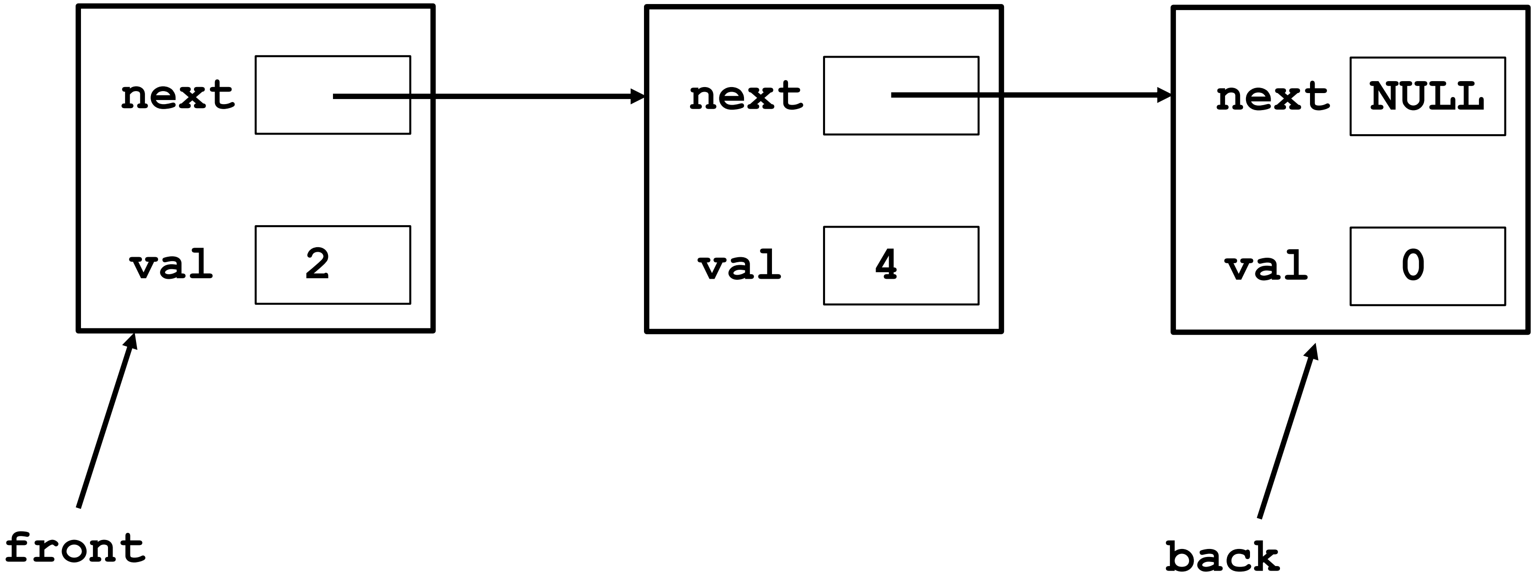


What happens if we want more than two elements?

What happens if we don't know the size we need until run-time?

Revisiting the Queue Example

- Simple Data structure modeling a queue
  - Implemented with a singly linked list
- Items added to the end and removed from the front.
- We maintain a list of queue elements chained together with pointers.
- We can use Dynamic Allocation to create new elements

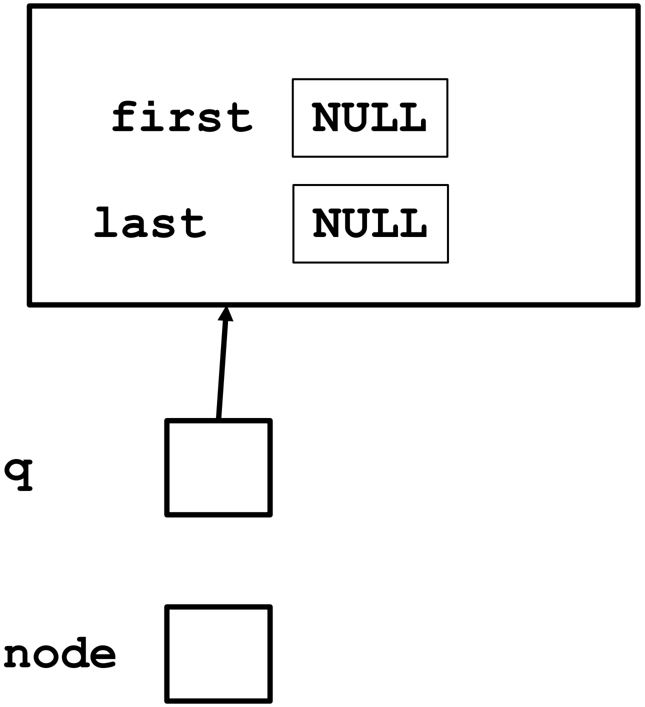


Dynamically Allocated Queue Demo

- See code on course website:
  - main_queue.c
  - queue.h
  - queue.c
  - Makefile

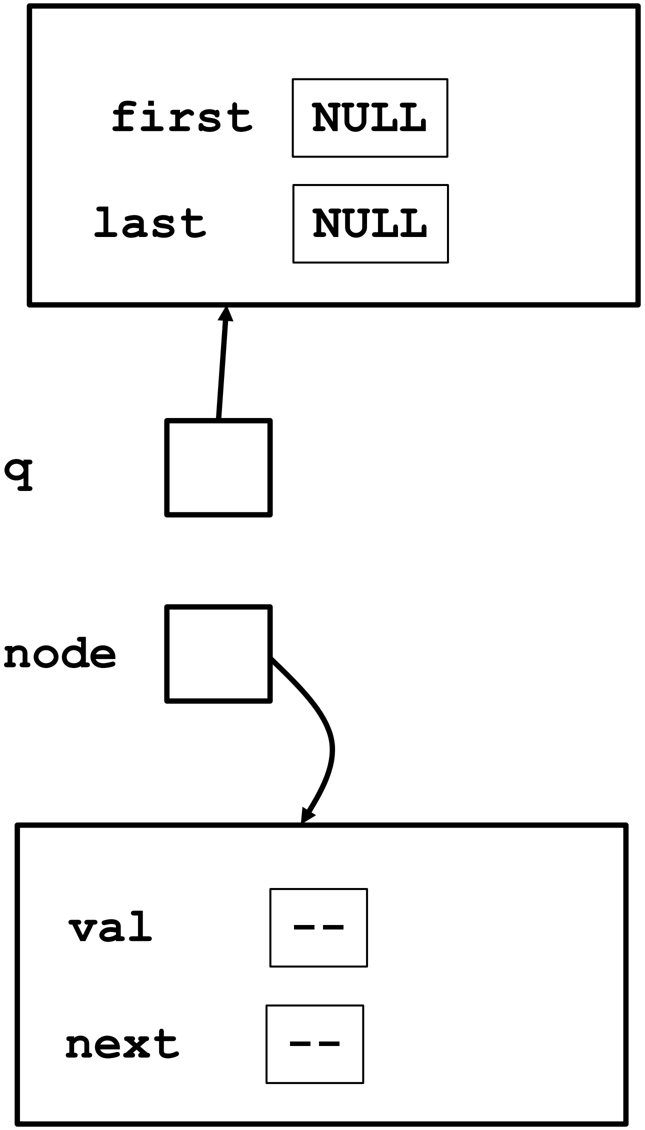
Queue_Add

```
void Queue_Add(Queue *q, int val) {
  Queue_Node* node;
  node = malloc(sizeof(Queue_Node));
  if (node == NULL) {
    printf("ERROR");
    exit(EXIT_FAILURE);
  }

  node->next = NULL;
  node-> val = val;
  if (q->last != NULL) {
    q->last->next = node;
    q->last = node;
  } else {
    q->first = node;
    q->last = node;
  }
}
```



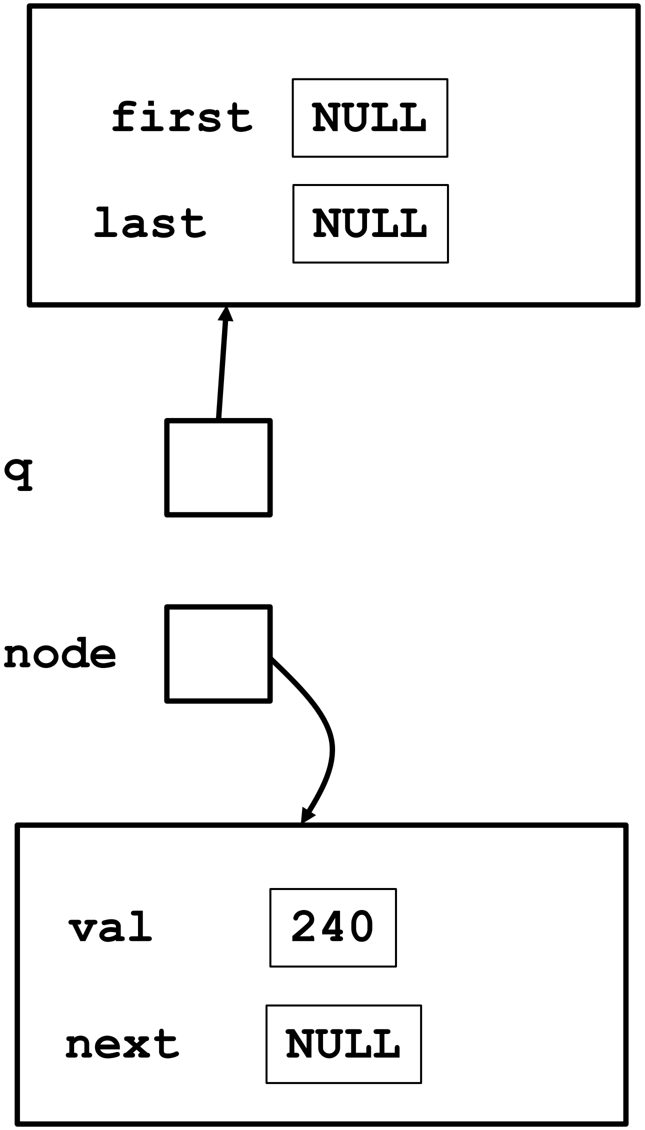
Queue_Add

```
void Queue_Add(Queue *q, int val) {
  Queue_Node* node;
  node = malloc(sizeof(Queue_Node));
  if (node == NULL) {
    printf("ERROR");
    exit(EXIT_FAILURE);
  }

  node->next = NULL;
  node-> val = val;
  if (q->last != NULL) {
    q->last->next = node;
    q->last = node;
  } else {
    q->first = node;
    q->last = node;
  }
}
```



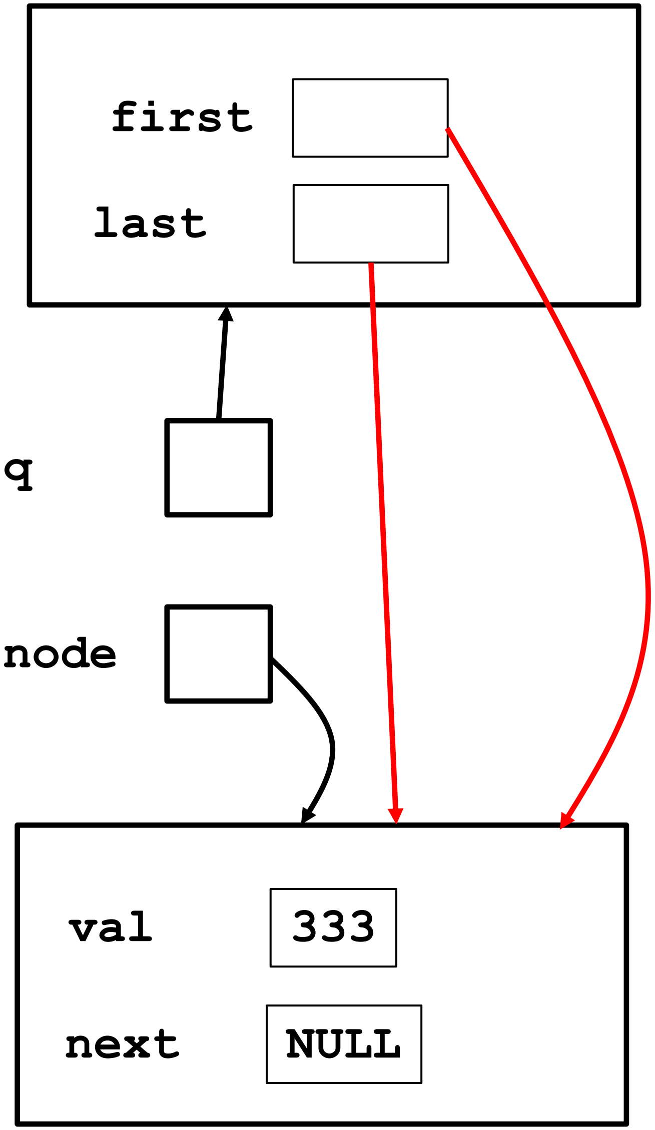
Queue_Add

```
void Queue_Add(Queue *q, int val) {
  Queue_Node* node;
  node = malloc(sizeof(Queue_Node));
  if (node == NULL) {
    printf("ERROR");
    exit(EXIT_FAILURE);
  }

  node->next = NULL;
  node-> val = val;
  if (q->last != NULL) {
    q->last->next = node;
    q->last = node;
  } else {
    q->first = node;
    q->last = node;
  }
}
```



Queue_Add

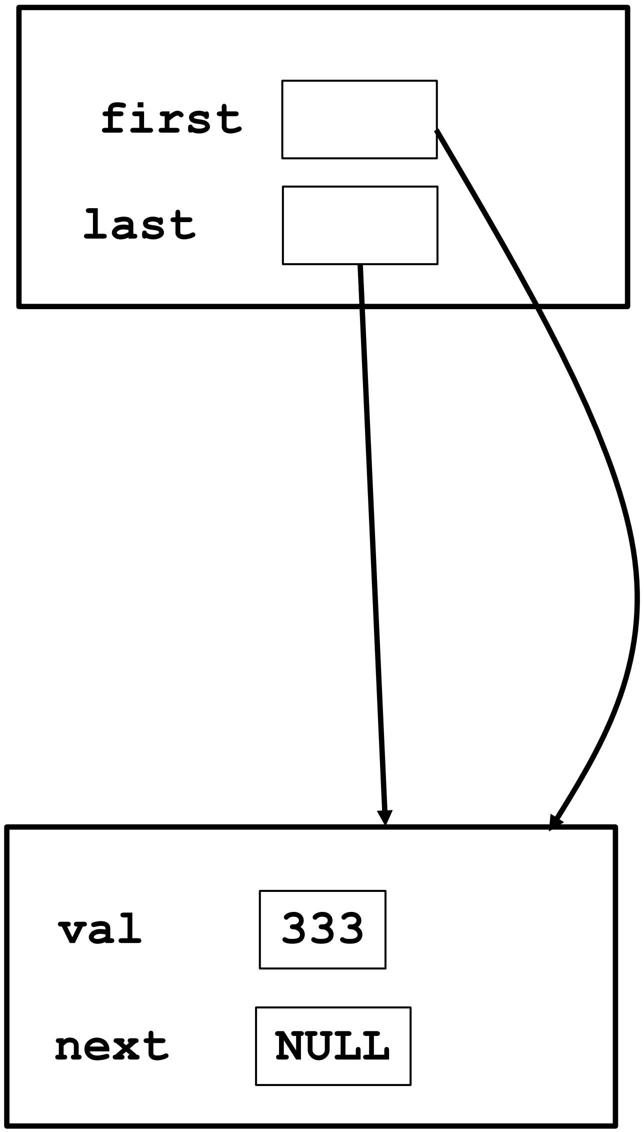
```
void Queue_Add(Queue *q, int val) {
  Queue_Node* node;
  node = malloc(sizeof(Queue_Node));
  if (node == NULL) {
    printf("ERROR");
    exit(EXIT_FAILURE);
  }

  node->next = NULL;
  node-> val = val;
  if (q->last != NULL) {
    q->last->next = node;
    q->last = node;
  } else {
    q->first = node;
    q->last = node;
  }
}
```



Queue_Add

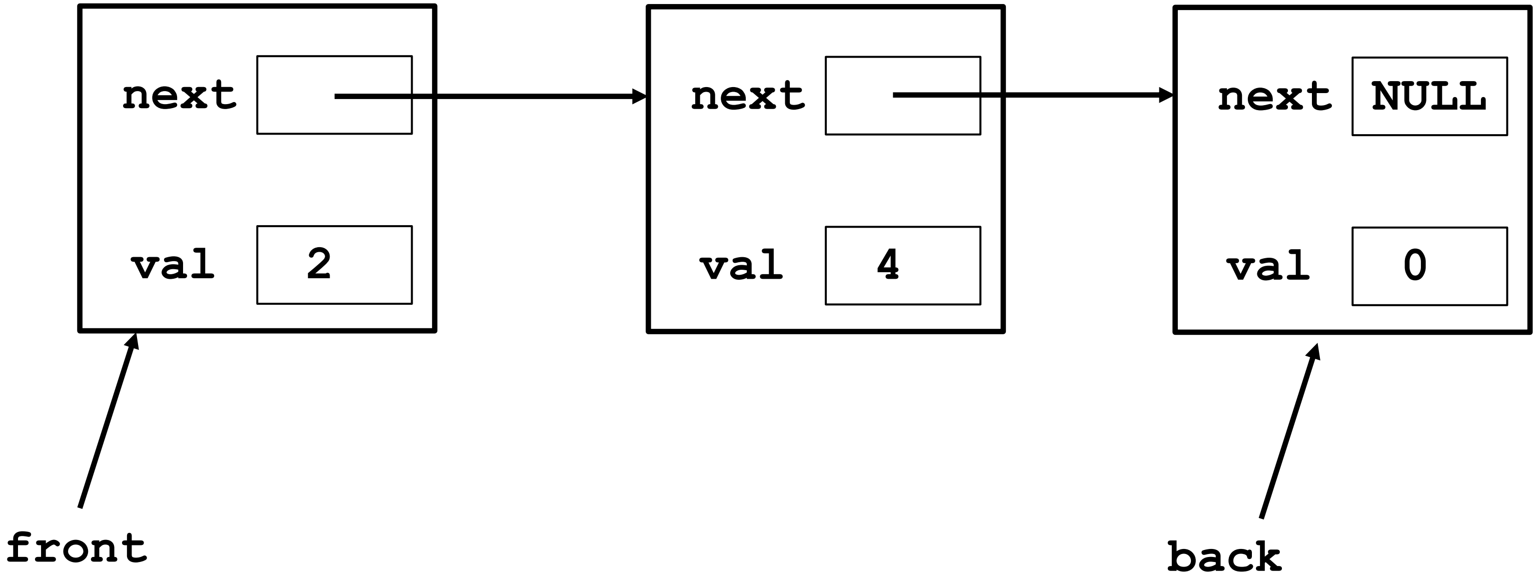
```
void Queue_Add(Queue *q, int val) {
  Queue_Node* node;
  node = malloc(sizeof(Queue_Node));
  if (node == NULL) {
    printf("ERROR");
    exit(EXIT_FAILURE);
  }

  node->next = NULL;
  node-> val = val;
  if (q->last != NULL) {
    q->last->next = node;
    q->last = node;
  } else {
    q->first = node;
    q->last = node;
  }
}
```

Since node is dynamically allocated, it persists after the function returns



Revisiting the Queue Example

- Simple Data structure modeling a queue
  - Implemented with a singly linked list
- Items added to the end and removed from the front.
- We maintain a list of queue elements chained together with pointers.
- We can use Dynamic Allocation to create new elements



Lecture Outline

- Structs Warm-up
- The Heap
  - malloc() & free()
- **Modules & Header Files**
- Dynamic Memory Pitfalls
- GDB & Valgrind

Multi-File C Programs (Modules)

- In our previous example, we created a queue *module*

- A module is a self-contained piece of an overall program
  - Has externally visible functions that customers can invoke
  - Has externally visible typedefs, and perhaps global variables, that customers can use
  - May have internal functions, typedefs, or global variables that customers should not look at

- The module's *interface* is its set of public functions, typedefs, and global variables

C Header Files

- Header: a file whose only purpose is to be #include'd
  - Generally, has a filename `.h` extension
  - Holds the variables, types, and function prototype declarations that make up the interface to a module
  - There are <system-defined> and "programmer-defined" headers

```
#include <stdio.h>    #include "./cstring.h"
```

- Main Idea:
  - Every `name.c` is intended to be a module that has a `name.h`
  - `name.h` declares the interface to that module
  - Other modules can use name by `#include`-ing `name.h`
    - They should assume as little as possible about the implementation in name.c

C Module Conventions

- File contents:
  - `.h` files only contain declarations, **never** definitions
  - `.c` files never contain prototype declarations for functions that are intended to be exported through the module interface

- Including:
  - ***NEVER*** #include a .c file
  - Only #include .h files
  - #include all of headers you reference, even if another header (transitively) includes some of them

C Header Guards

- Header Files in C (and C++) need to have two lines at the top and a line at the bottom.
  - These are to prevent a file from being include'd twice (which would get an error from having multiple definitions of the same thing).

pair.h

```
#ifndef PAIR_H_
#define PAIR_H_

typedef struct {
  int a;
  int b;
} pair;

#endif // PAIR_H_
```

util.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
pair* make_pair(int a, int b);

#endif // UTIL_H_
```

Note the:
FILE_NAME_H_
naming convention

Last line ends the
header file

main.c

```
#include "pair.h"
#include "util.h"
int main(int atgc, char* argv[]) {
```

Lecture Outline

- Structs Warm-up
- The Heap
  - malloc() & free()
- Modules & Header Files
- **Dynamic Memory Pitfalls**
- GDB & Valgrind

Dynamic Memory Pitfalls

- Buffer Overflows
  - E.g. ask for 10 bytes, but write 11 bytes
  - Could overwrite information needed to manage the heap
  - Common when forgetting the null-terminator on malloc'd strings
- Not checking for `NULL`
  - Malloc returns NULL if out of memory
  - Should check this after every call to malloc
- Giving `free()` a pointer to the middle of an allocated region
  - Free won't recognize the block of memory and may crash
- Giving free() a pointer that has already been freed
  - Will interfere with the management of the heap and likely crash
- `malloc` does NOT initialize memory
  - There are other functions like `calloc` that will zero out memory

Memory Leaks

- The most common Memory Pitfall
- What happens if we malloc something, but don’t free it?
  - That block of memory cannot be reallocated, even if we don’t use it anymore, until it is **`free`**d
  - If this happens enough, we run out of heap space and program may slow down and eventually crash

- Garbage Collection
  - Automatically “frees” anything once the program has lost all references to it
  - Affects performance, but avoid memory leaks
  - Java has this, C doesn’t

Poll Everywhere

pollev.com/tqm

- Which line below is first to cause a crash?
  - Yes, there are a lot of bugs, but not all cause a crash ☺
  - See if you can find all the bugs!

A. Line 1

B. Line 4

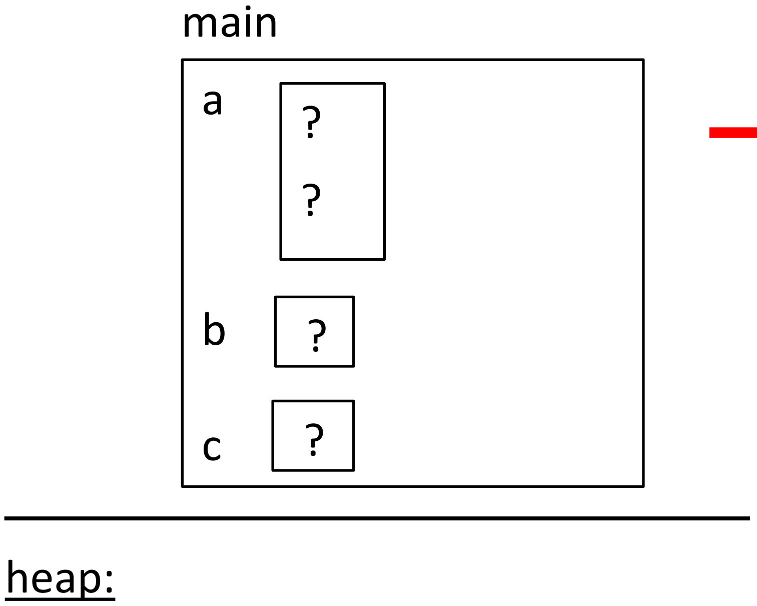
C. Line 6

D. Line 7

E. We're lost…

```
  #include <stdio.h>
  #include <stdlib.h>

  int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

1   a[2] = 5;
2   b[0] += 2;
3   c = b+3;
4   free(&(a[0]));
5   free(b);
6   free(b);
7   b[0] = 5;

    return 0;
  }
```

Memory Corruption - What Happens?

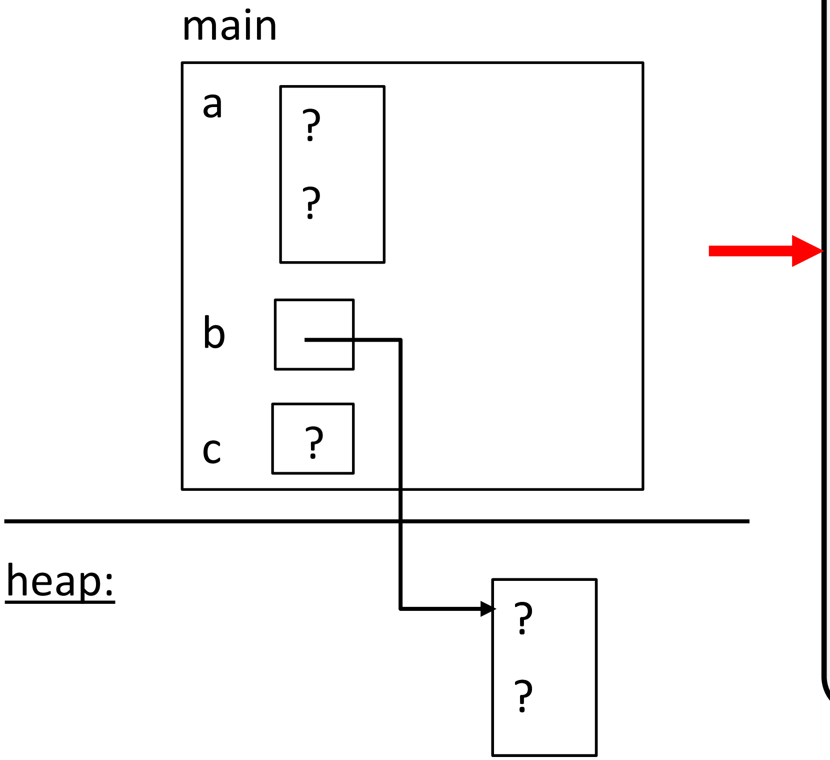


```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

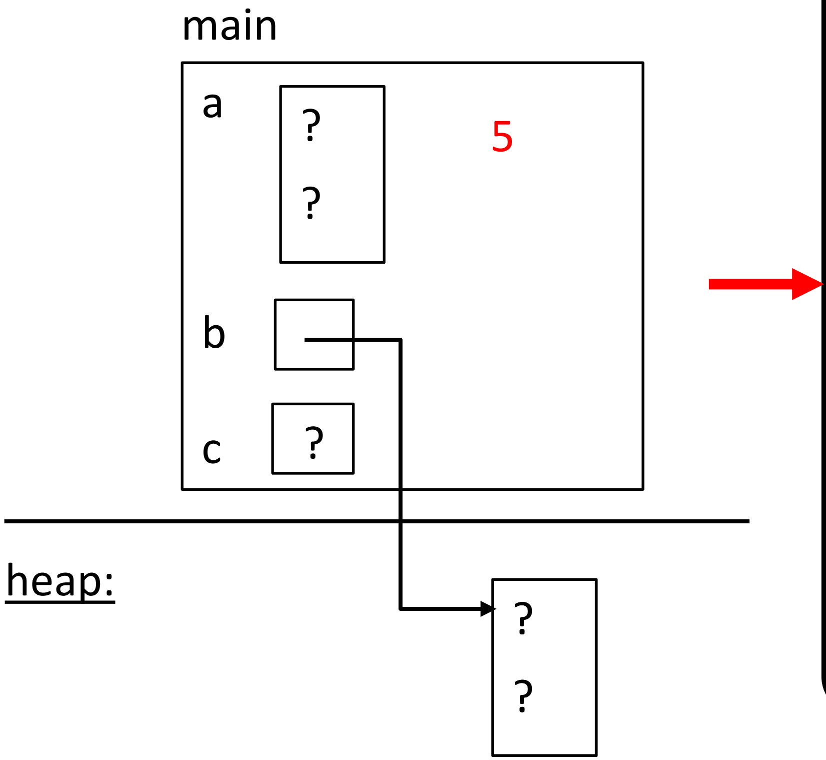


```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assigns past the end of an array
  b[0] += 2;  // assumes malloc zeros out memory
  c = b+3;    // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);    // double-free the same block
  b[0] = 5;   // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

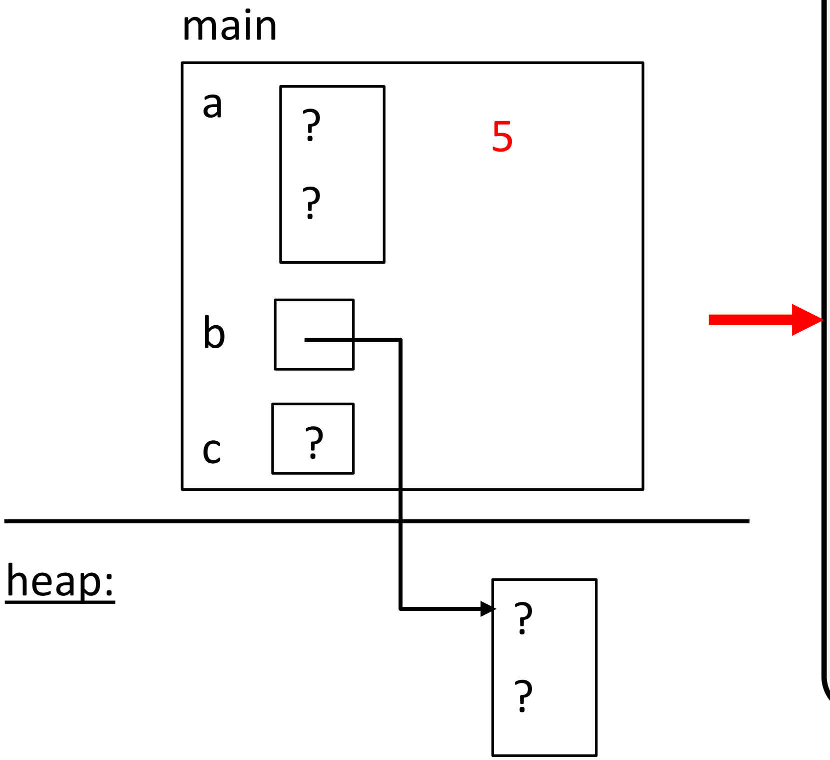


```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assigns past the end of an array
  b[0] += 2;  // assumes malloc zeros out memory
  c = b+3;    // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);    // double-free the same block
  b[0] = 5;   // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

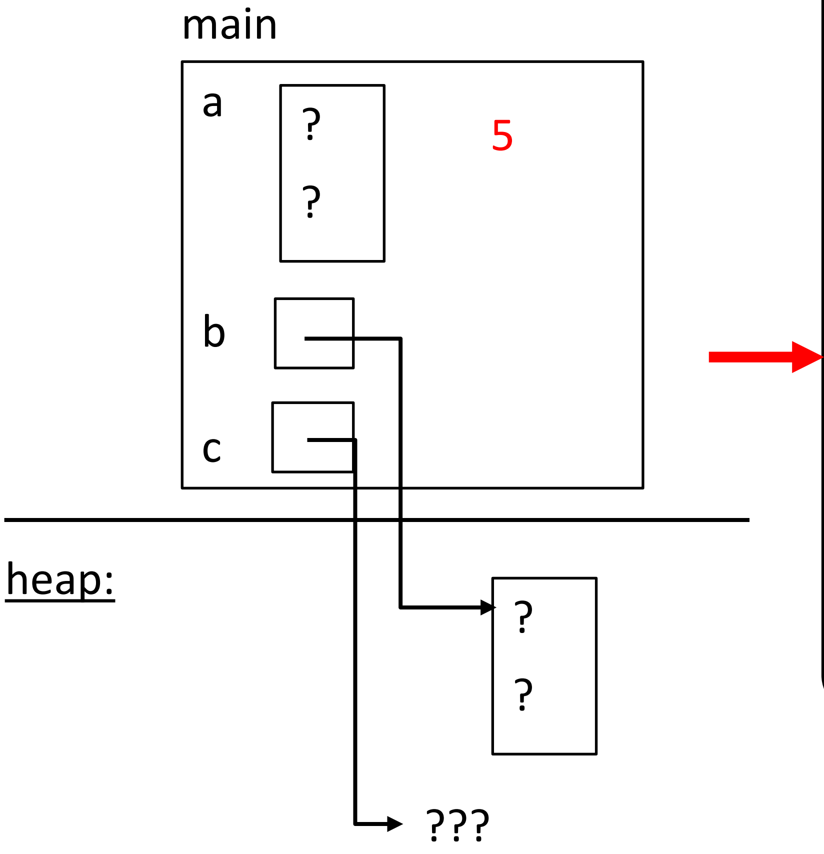


```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

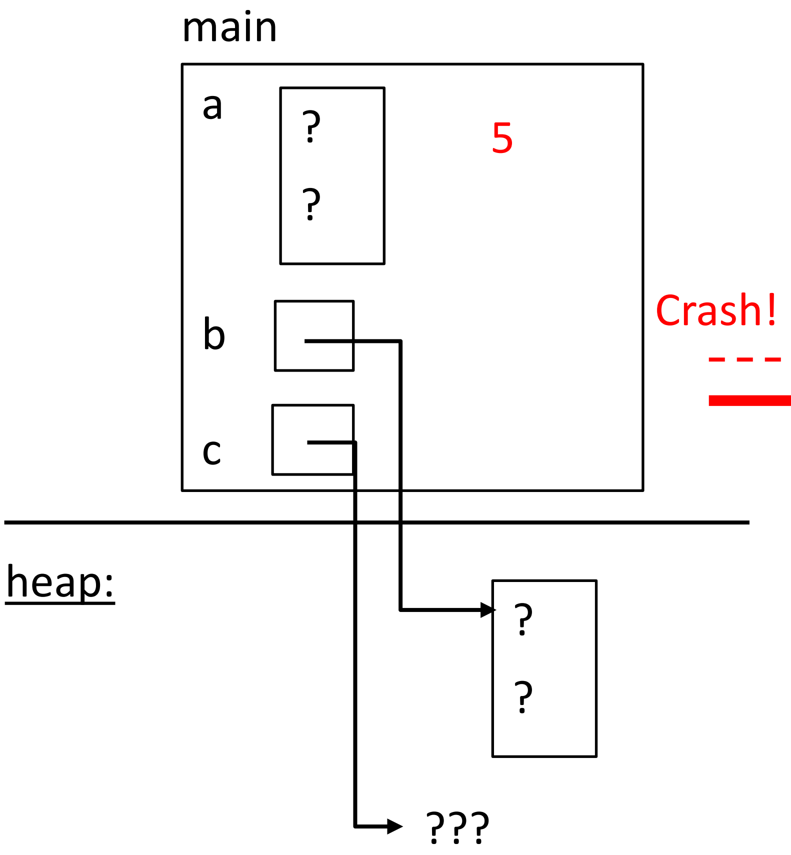


```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));   // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?



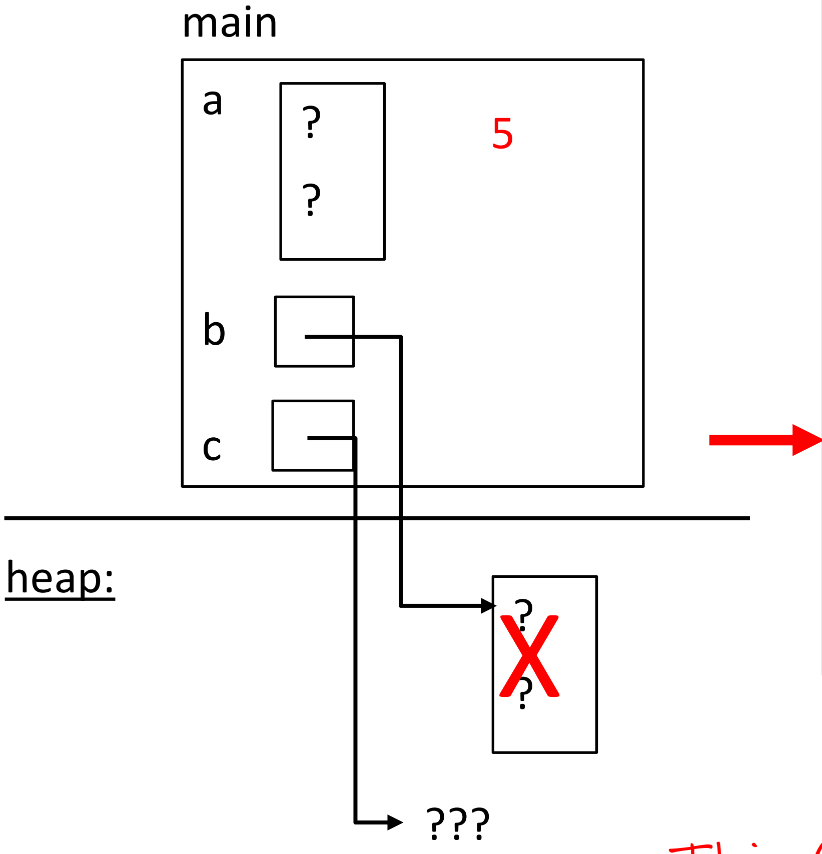
Crash!

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?



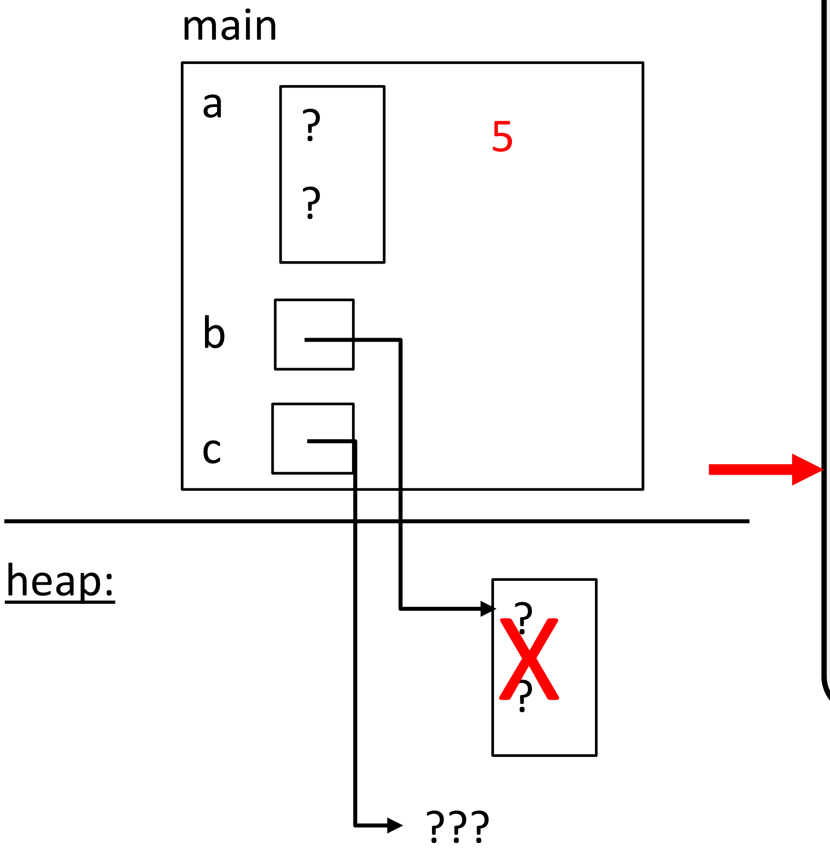
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));   // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

This "double free" would also cause the program to crash

memcorrupt.c

Memory Corruption - What Happens?

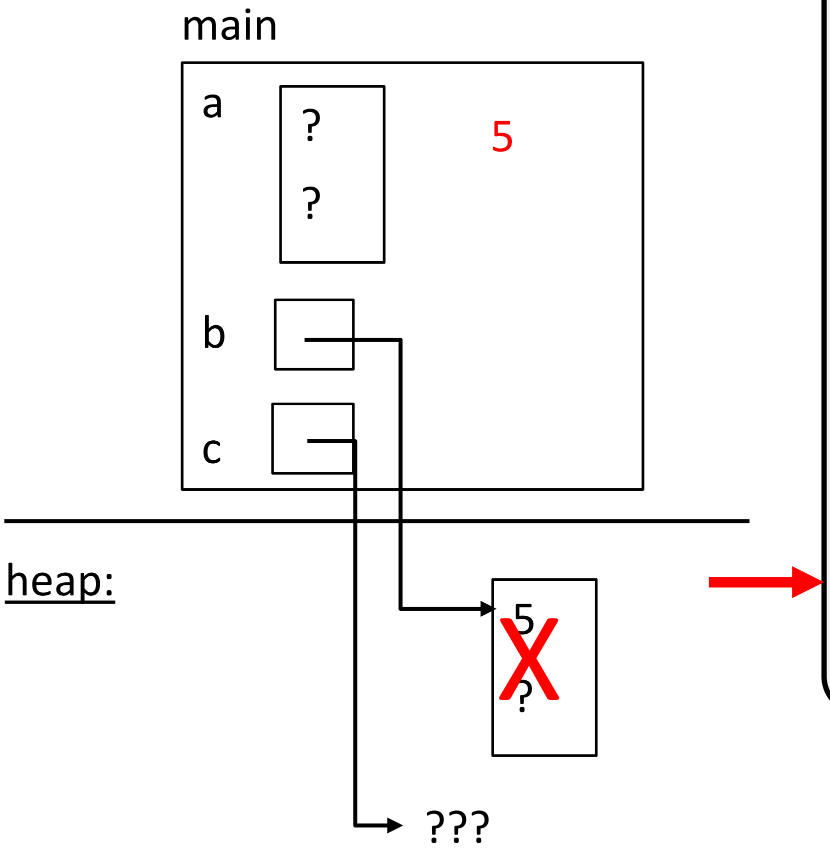


```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));   // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?



```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assigns past the end of an array
  b[0] += 2;  // assumes malloc zeros out memory
  c = b+3;    // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);    // double-free the same block
  b[0] = 5;   // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Lecture Outline

- Structs Warm-up
- The Heap
  - malloc() & free()
- Modules & Header Files
- Dynamic Memory Pitfalls
- **GDB & Valgrind**

Motivation

- The assignments will start getting bigger and are more open ended. Lots of potential for bugs

- **Debugging is a skill that you will need throughout your programming career**

- gdb (GNU Debugger) is a debugging tool
  - Very useful in tracking undefined behavior

- Valgrind
  - Checks for various memory errors
  - If you have odd behavior, valgrind may point out the cause.

Valgrind

- Tool used for identifying memory errors
- Will be used on your HW submissions going forward
- Detects:
  - Use of uninitialized memory
  - Reading/writing memory after it has been freed
  - Reading/writing to the end of malloc'd blocks
  - Reading/writing to inappropriate areas on the stack
  - Memory leaks where pointers to malloc'd blocks are lost

- Run with

  -
    ```
    valgrind --leak-check=full ./executable
    ```

Brief GDB & Valgrind Demo: Seg Faults

- IF NOTHING ELSE FROM GDB: GDB is very useful for finding a segmentation fault
  - Run the code on gdb till segmentation fault
  - Type in the command `backtrace`

- Commands:
  - `gdb ./executable`
  - `run`
  - `backtrace`

- `segfault.c`

Poll Everywhere

pollev.com/tqm

- What is the error here?

```c
28 int* range_array(int n, int m) {
29   int length = m - n + 1;
30 
31   // Heap allocate the array needed to return
32   int* array = malloc(sizeof(int) * length);
33 
34   // Initialized the elements
35   for (int i = 0; i <= length; i++) {
36     array[i] = i + n;
37   }
38 
39   return array;
40 }
```

```
==28602== Invalid write of size 4
==28602==    at 0x10926C: range_array (leaky.c:36)
==28602==    by 0x1091B6: main (leaky.c:15)
==28602==  Address 0x4a9404c is 0 bytes after a block of size 12 alloc'd
==28602==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.s
==28602==    by 0x109246: range_array (leaky.c:32)
==28602==    by 0x1091B6: main (leaky.c:15)
```

Demo: The rest of Leaky.c

- Valgrind will tell you which line had bad memory accesses
- Valgrind will let you know when you have a memory leak, and where that leak was allocated
  - Note: where it is allocated is almost always different from where we need to free it.
- See course website:
  - **`leaky.c`**

Next Time

- Makefile
- #define constants
- Reading & parsing from stdin
  - getline
  - sscanf
- void* generics
- Maybe more GDB?
- Maybe more strings?