# CS50 Week 4

Connor Leggett
cjleggett@college.harvard.edu
attendance: tinyurl.com/y3ghtnve

# Questions?

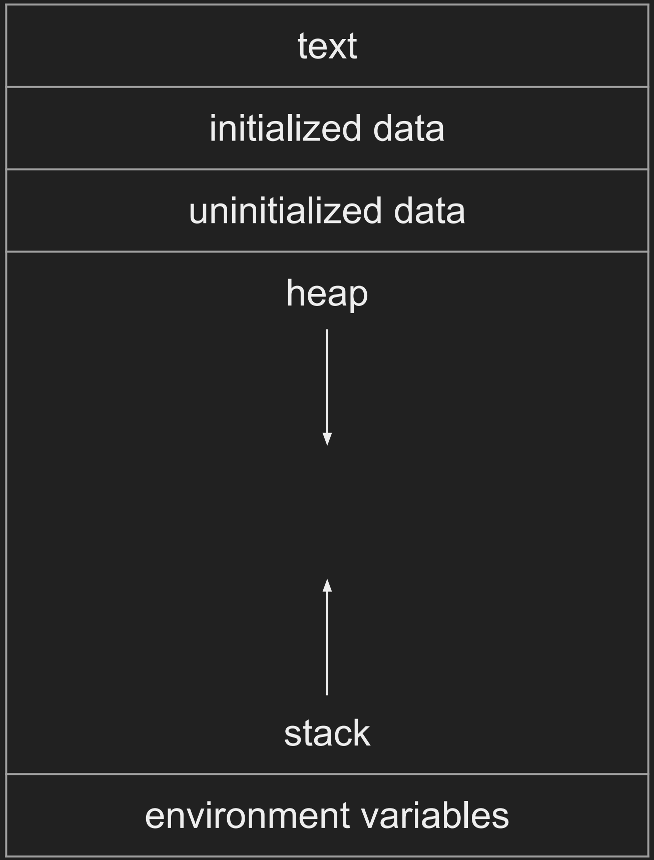
# Content Recap

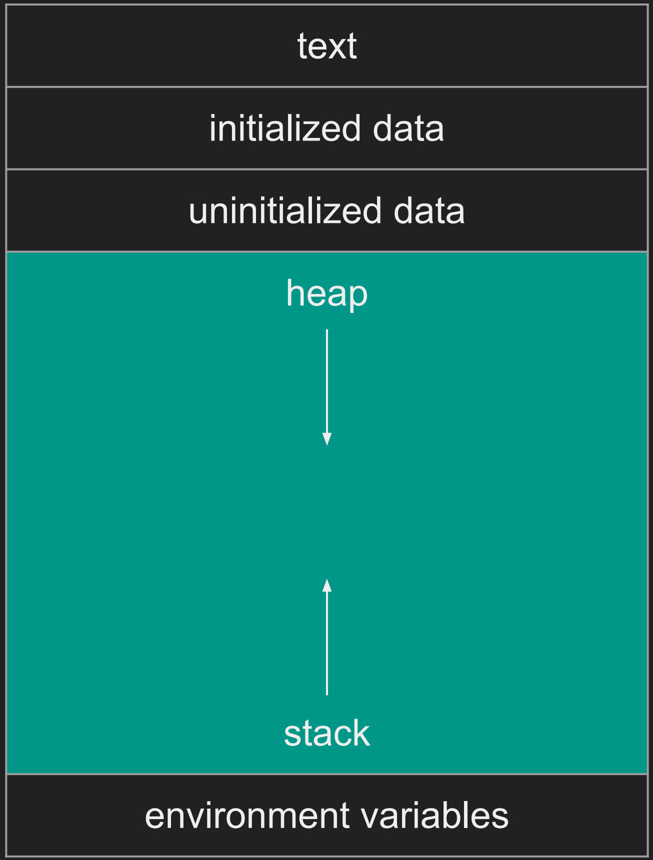
# Dynamic Memory Allocation

- We know one way to use pointers -- connecting a pointer variable by pointing it at another variable that already exists in our program.

- But what if we don’t know in advance how much memory we’ll need at compile time? How do we access more memory at runtime?

- Pointers can also be used to do this. Memory allocated *dynamically* (at runtime) comes from a pool of memory called the heap. Memory allocated at compile time typically comes from a pool of memory called the stack.


text
initialized data
uninitialized data
heap
stack
environment variables


text
initialized data
uninitialized data
heap
stack
environment variables

- We get this dynamically-allocated memory via a call to the function `malloc()`, passing as its parameter the number of *bytes* we want. `malloc()` will return to you a **pointer** to that newly-allocated memory.
    - If malloc() can't give you memory (because, say, the system ran out), you get a NULL pointer.

```
// Statically obtain an integer
int x;

// Dynamically obtain an integer
int *px = malloc(4);
```

- We get this dynamically-allocated memory via a call to the function `malloc()`, passing as its parameter the number of *bytes* we want. `malloc()` will return to you a **pointer** to that newly-allocated memory.
    - If malloc() can't give you memory (because, say, the system ran out), you get a NULL pointer.

```
// Statically obtain an integer
int x;

// Dynamically obtain an integer
int *px = malloc(sizeof(int));
```

```c
// Get an integer from the user
int x = get_int();

// Array of floats on the stack
float stack_array[x];

// Array of floats on the heap
float *heap_array = malloc(x * sizeof(float));
```

- There’s a catch: Dynamically allocated memory is not automatically returned to the system for later use when no longer needed.

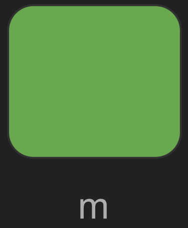
- Failing to return memory back to the system when you no longer need it results in a **memory leak**, which compromises your system’s performance.

- All memory that is dynamically allocated must be released back by `free()`-ing its pointer.

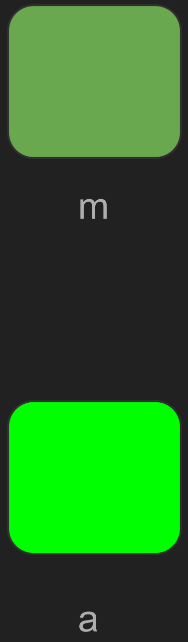
```
char *word = malloc(50 * sizeof(char));

// do stuff with word

// now we’re done
free(word);
```

- Every block of memory that you `malloc()`, you must later `free()`.
- **Only** memory that you obtain with `malloc()` should you later `free()`.
- Do not `free()` a block of memory more than once.

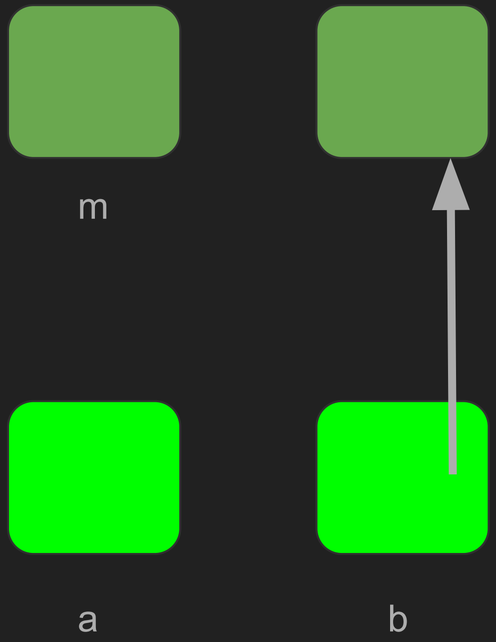
```
int m;
```



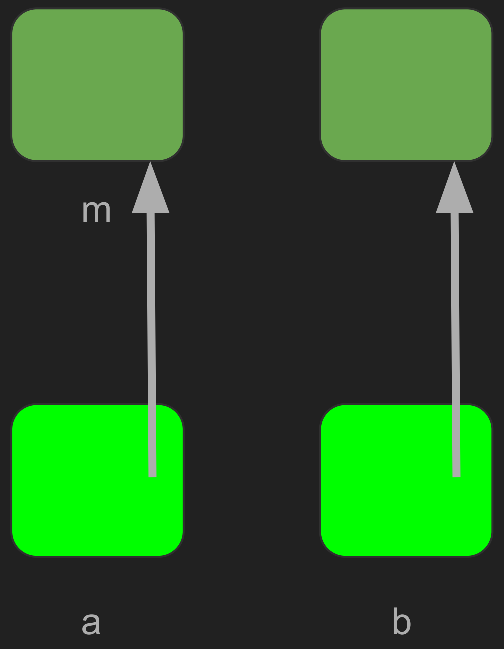
```c
int m;
int *a;
```



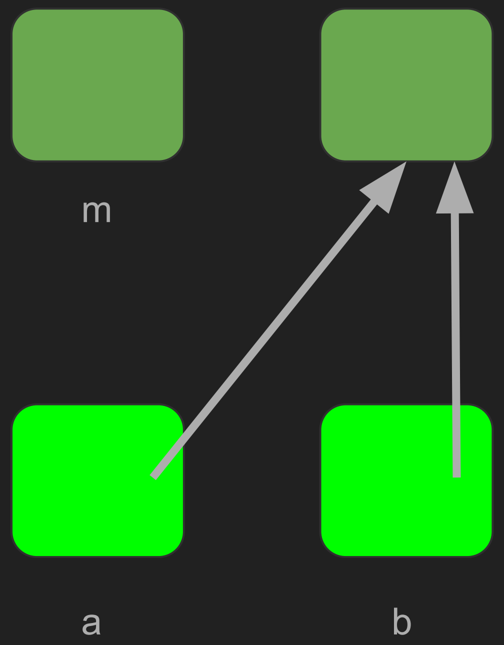
```
int m;
int *a;
int *b = malloc(sizeof(int));
```



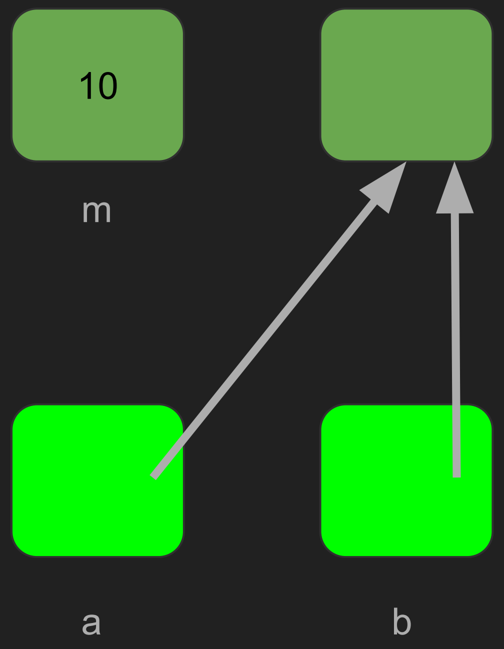
```c
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
```



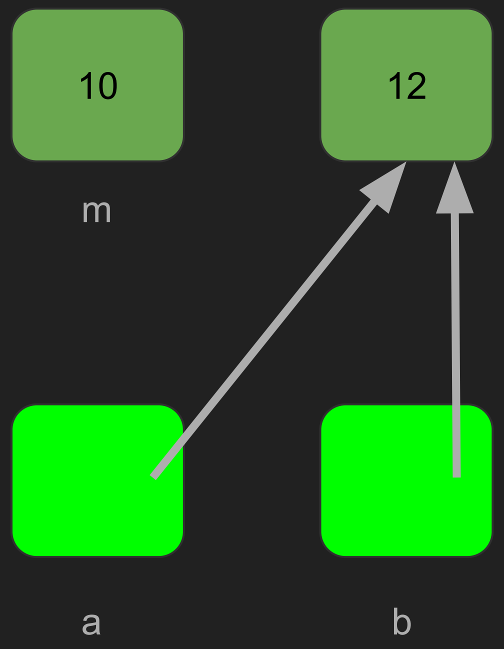
```c
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
```



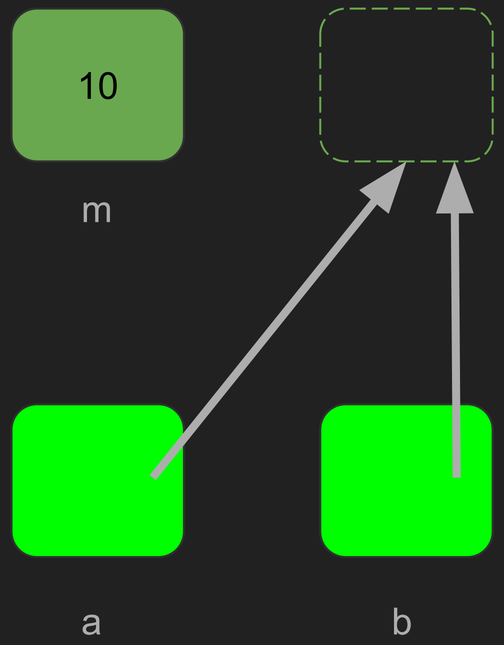
```c
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
```



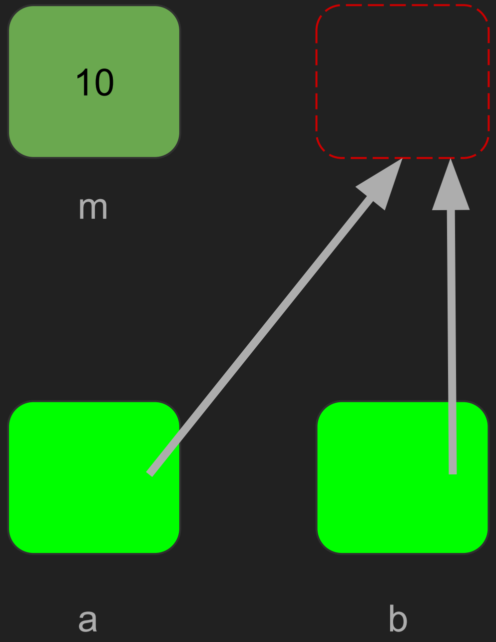
```c
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
*b = m + 2;
```



```c
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
*b = m + 2;
free(a);
```



```c
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
*b = m + 2;
free(a);
*b = 11;
```



# File I/O

- The ability to read data from and write data to files is the primary means of storing **persistent data**, which exists outside of your program.

- In C, files are abstracted using a data structure called a `FILE`. Almost universally, though, when working with `FILE`s do we actually use pointers to files (aka `FILE *`).

- The functions we use to manipulate files all are found in **stdio.h**.

- Every one of them accepts a `FILE *` as one of its parameters, except `fopen()` which is used to get a file pointer in the first place.

- Some of the most common file input/output (I/O) functions we'll use are the following:

| | | | | | |
|---|---|---|---|---|---|
| `fopen()` | `fclose()` | `fgetc()` | `fputc()` | `fread()` | `fwrite()` |

- `fopen()` opens a file and returns a pointer to it. Always check its return value to make sure you don’t get back `NULL`.

```
FILE *ptr = fopen(<filename>, <operation>);
```

- `fopen()` opens a file and returns a pointer to it. Always check its return value to make sure you don't get back `NULL`.

```
FILE *ptr = fopen("test.txt", "r");
FILE *ptr2 = fopen("test2.txt", "w");
FILE *ptr3 = fopen("test3.txt", "a");
```

- `fgetc()` reads and returns the next character from the file, assuming the operation for that file contains "r". `fputc()` writes or appends the specified character to the file, assuming the operation for that pointer contains "w" or "a".

```
fgetc(<file pointer>);

fputc(<character>, <file pointer>);
```

- `fgetc()` reads and returns the next character from the file, assuming the operation for that file contains "r". `fputc()` writes or appends the specified character to the file, assuming the operation for that pointer contains "w" or "a".

```
char c = fgetc(ptr1);
fputc('x', ptr2);
fputc('5', ptr3);
```

- `fread()` and `fwrite()` are analogs to `fgetc()` and `fputc()`, but for a generalized quantity (`qty`) of blocks of an arbitrary (`size`), holding those blocks in (or writing them from) a temporary buffer, usually an array, for local use within the program.

```
fread(<buffer>, <size>, <qty>, <file pointer>);

fwrite(<buffer>, <size>, <qty>, <file pointer>);
```

- `fread()` and `fwrite()` are analogs to `fgetc()` and `fputc()`, but for a generalized quantity (`qty`) of blocks of an arbitrary (`size`), holding those blocks in (or writing them from) a temporary buffer, usually an array, for local use within the program.

```
int arr[10];
fread(arr, sizeof(int), 10, ptr);
fwrite(arr, sizeof(int), 10, ptr2);
fwrite(arr, sizeof(int), 10, ptr3);
```

- `fclose()` closes a previously opened file pointer.

```
fclose(<file pointer>);
```

- `fclose()` closes a previously opened file pointer.

```
fclose(ptr);
fclose(ptr2);
fclose(ptr3);
```

- Lots of other useful functions abound in stdio.h for you to work with. Here are some you might find useful.

| | |
|---|---|
| `fgets()` | Reads a full string from a file. |
| `fputs()` | Writes a full string to a file. |
| `fprintf()` | Writes a formatted string to a file. |
| `fseek()` | Allows you to rewind or fast-forward within a file. |
| `ftell()` | Tells you at what (byte) position you are at within a file. |
| `feof()` | Tells you whether you’ve read to the end of a file. |
| `ferror()` | Indicates whether an error has occurred in working with a file. |

# Hexadecimal

- Most Western cultures use the decimal system (base 10) to represent numeric data.

- As we know, computers use the binary system (base 2) to represent numeric (and indeed all) data.

- As computer scientists, it’s useful to be able to express data the same way the computer does, but trying to parse a big chain of 0s and 1s can be annoying.

- The **hexadecimal system** (base 16) is a much more concise way to express data on a computer system.

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
```

- Hexadecimal makes it so that a group of four binary digits (bits) can be expressed with a single character, since there are 16 possible combinations of 4 bits (0 or 1).

| Decimal | Binary | Hex |
|---|---|---|
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |

| Decimal | Binary | Hex |
|---|---|---|
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |
| 10 | 1010 | a |
| 11 | 1011 | b |
| 12 | 1100 | c |
| 13 | 1101 | d |
| 14 | 1110 | e |
| 15 | 1111 | f |

| Decimal | Binary | Hex |
|---|---|---|
| 0 | 0000 | 0x0 |
| 1 | 0001 | 0x1 |
| 2 | 0010 | 0x2 |
| 3 | 0011 | 0x3 |
| 4 | 0100 | 0x4 |
| 5 | 0101 | 0x5 |
| 6 | 0110 | 0x6 |
| 7 | 0111 | 0x7 |

| Decimal | Binary | Hex |
|---|---|---|
| 8 | 1000 | 0x8 |
| 9 | 1001 | 0x9 |
| 10 | 1010 | 0xa |
| 11 | 1011 | 0xb |
| 12 | 1100 | 0xc |
| 13 | 1101 | 0xd |
| 14 | 1110 | 0xe |
| 15 | 1111 | 0xf |

- To convert a binary number to hexadecimal, group four bits together from **right to left**, padding the leftmost group with extra 0s at the front if necessary.

- To convert a binary number to hexadecimal, group four bits together from **right to left**, padding the leftmost group with extra 0s at the front if necessary.

`01000110101000101011100100111101`

- To convert a binary number to hexadecimal, group four bits together from **right to left**, padding the leftmost group with extra 0s at the front if necessary.

01000110101000101011100100111101

0100 0110 1010 0010 1011 1001 0011 1101

- To convert a binary number to hexadecimal, group four bits together from **right to left**, padding the leftmost group with extra 0s at the front if necessary.

```
 01000110101000101011100100111101

0100 0110 1010 0010 1011 1001 0011 1101

  4    6    A    2    B    9    3    D
```

- To convert a binary number to hexadecimal, group four bits together from **right to left**, padding the leftmost group with extra 0s at the front if necessary.

```
  01000110101000101011100100111101

0100 0110 1010 0010 1011 1001 0011 1101

 4    6    A    2    B    9    3    D

            0x46A2B93D
```

# Lecture Recap

- Watch the shorts!

# Practice Problems

github.com/cjleggett/2021-section

# file_practice.c

# copy/capitalize

- Task: Write a program that copies the contents of one file to another, and then copies a capitalized version to a third file.
- Expected Behavior:
  ./filepractice file1.txt file2.txt file3.txt
  (nothing printed, but check file contents!)

# gif_detector.c

- Task: write a program that takes in a file, and determines whether or not it is a GIF.
- Background info: We will assume all GIFs begin with the file header: **GIF89a**
- Expected behavior:

  ./gif_detector file_that_is_a_gif
  GIF
  ./gif_detector file_that_is_not_a_gif
  NOT GIF

# Lab!

pointer_practice.c

# swap(int*, int*)

- Task: write a function that takes in pointers to two integers, and switches the values of the two
- Expected behavior:

```
./pointer_practice
Integer 1: 4
Integer 2: 5
Integer 1: 5
Integer 2: 4
```

# concat(char*, char*)

- Task: write a function that takes in pointers two char *s, and returns their concatenation
- Expected behavior:

  ./pointer_practice
  String 1: snow
  String 2: man
  snow + man = snowman

# challenge: append(int*, int, int*)

- Task: write a function that takes in one int* array, one integer representing the next int to be added, and one int* representing the current length of the array. Then, add the new int to the end, update the length int*, and return the new int* array.
- Expected behavior: Not easily visualized. See github code

# Problem Set Preview: Filter

- Making changes to photographs
- Lots of looping over 2-dimensional arrays
- Run through your code manually (not actually running it) with small examples to test.

# Problem Set Preview: Recover

- Recovering *secret* photographs from a raw file
- Watch the walkthrough!!!
- Go to tutorials or use Ed when stuck