START AT 3:07

What does malloc do?

allocates some memory

What is the argument that goes into malloc?

```
malloc ( 8 )
```

↑ size of memory you want in bytes

What does free do?

free up memory

What is the argument that goes into free?

```
free (*ptr)
```

↑ pointer

Segmentation Fault

using memory that you haven't allocated using malloc

CS50 Section 4

Hexadecimal – Base 16

A B C D E F
10 11 12 13 14 15

In writing, we can also indicate a value is in hexadecimal by prefixing it with 0x, as in 0x10, where the value is equal to 16 in decimal, as opposed to 10.

16₁₀ = 16

1's digit zero's digit

33 256 / 16

21 33 in hexadecimal

Recover

use hexadecimal – start with 0x!

Pointers

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // Get two strings
    string s = get_string("s: ");
    string t = get_string("t: ");

    // Compare strings' addresses
    if (s == t)
    {
        printf("Same\n");
    }
    else
    {
        printf("Different\n");
    }
}
```

checks if memory address is the same

s: "Hi!"

t: "Hi!"

Strings in C are pointers!

s going to point to the location in memory of the first char in the string

What does this return and why?

Pointers

We can see the address with the & operator,

```
int main(void)
{
    int n = 50;
    printf("%p\n", &n);
}
```

initialize int

go to the memory address of n

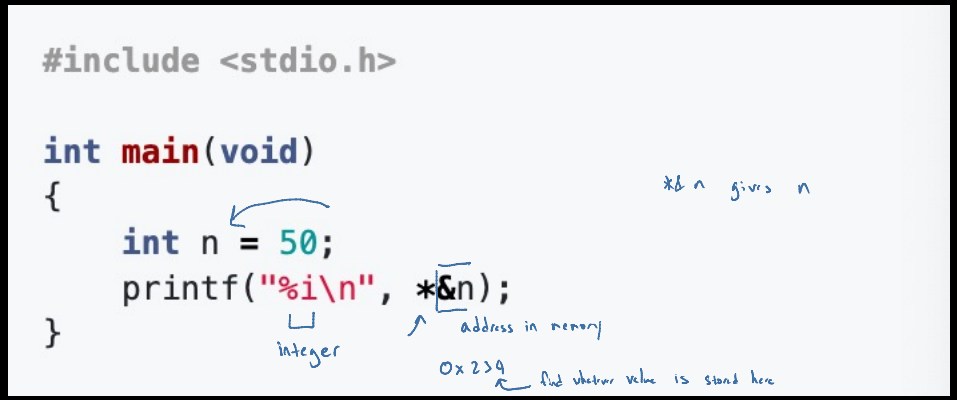
What would this look like?

&n 0x2a3...

Pointers

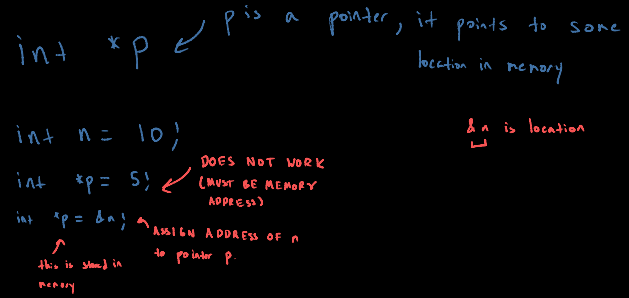
The * operator lets us “go to” the location that a pointer is pointing to.

What does this do?



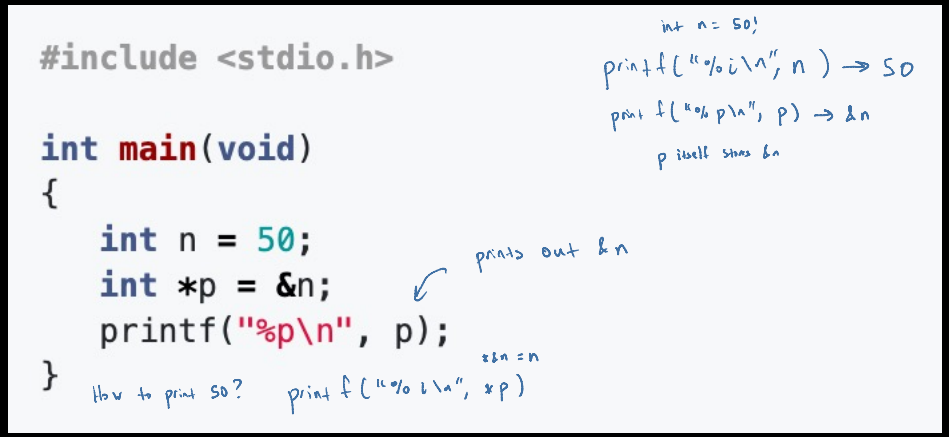
Pointers

How do we declare a variable that we want to be a pointer?



Pointers

How do we declare a variable that we want to be a pointer?



We use int *p to declare a variable, p, that has the type of *, a pointer, to a value of type int, an integer. What would this print?

Strings

How are strings stored in memory?

Strings are just pointers pointing to the location of the first character in memory.

String s = "Hello";

&s[0] → Why can I only store this?

Strings are contiguous in memory

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.

```
int n = 50;
int *p = &n
```

- 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?

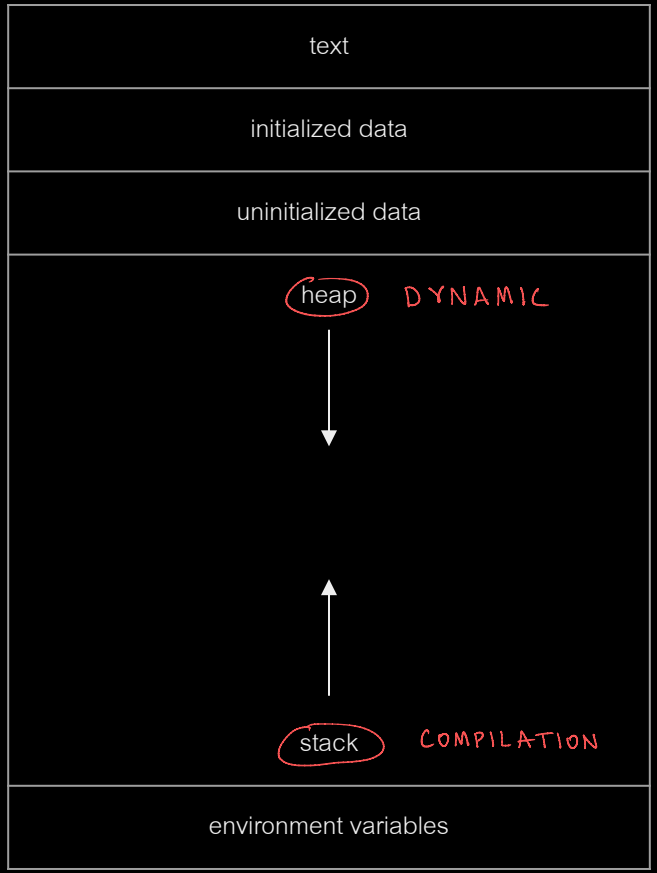
Dynamic memory allocation

- 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.

dynamically allocated

```
int a = 50;
```

compilation


text
initialized data
uninitialized data
heap
DYNAMIC
stack
COMPILATION
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.

malloc(4)

gives me 4 bytes of memory and gives me the pointer where it's stored!

If you call malloc, be sure to check for NULL!

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

] compiler will get memory

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

] DYNAMICALLY, when program runs

name of pointer

- 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.

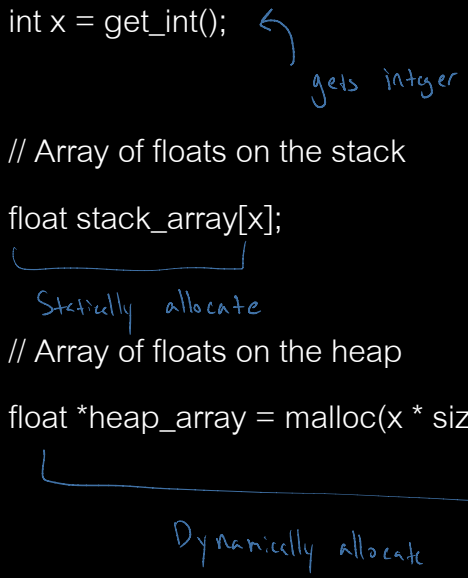
```
// Statically obtain an integer
int x;

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

```
// 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. malloc requires you to free memory

- 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);
```

size of char type in bytes

give memory back to computer

- Every block of memory that you malloc(), you must later free().

Just to check — run valgrind

- **Only** memory that you obtain with malloc() should you later free().

int n = 50;
int *p = &n;] => Do Not FREE(p)

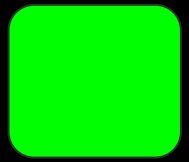
- Do not free() a block of memory more than once.

```
int m;
```

m

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

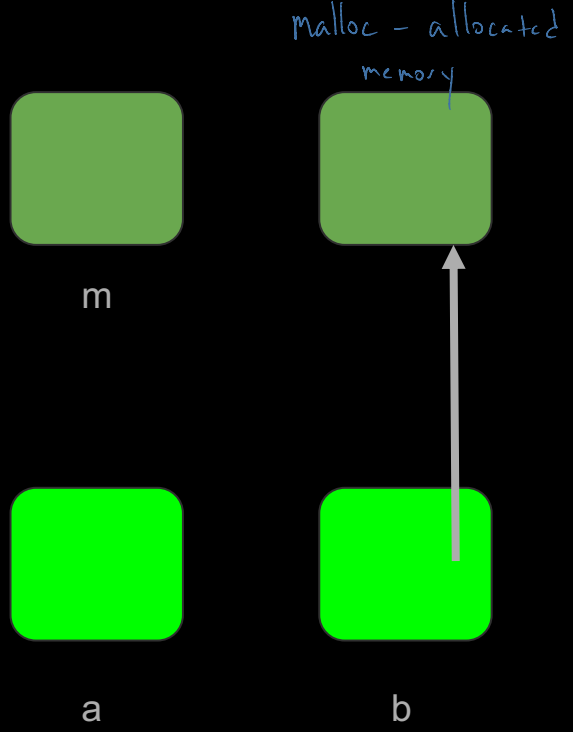
m



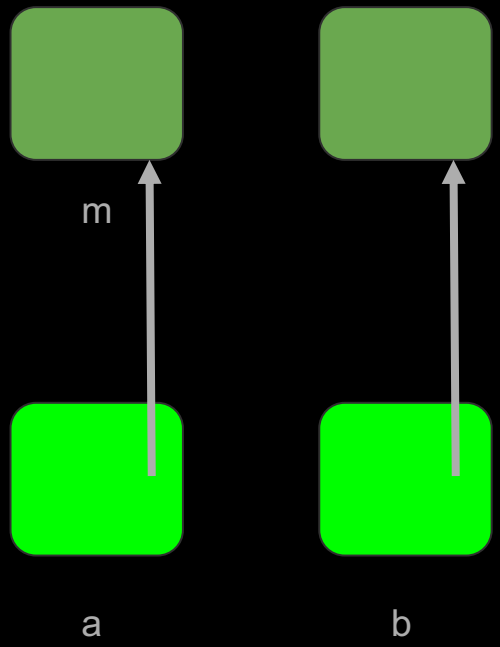
a

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

allocating me memory for an int

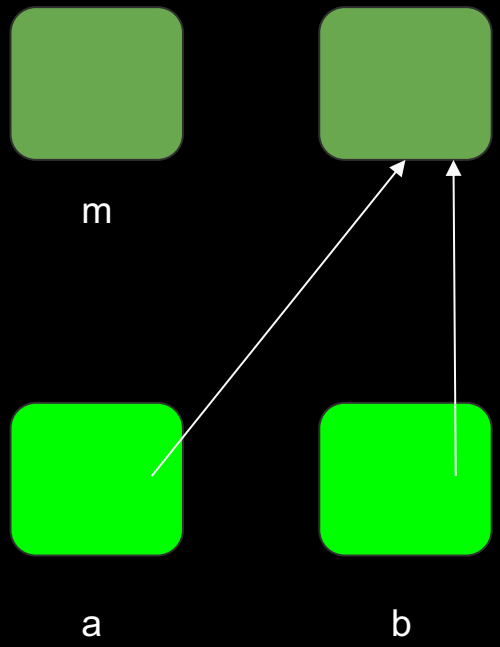


```
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
```

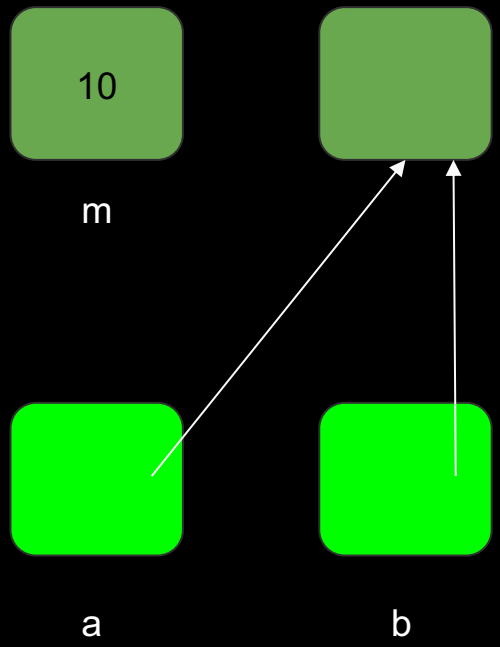


```
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
```

↑ location in memory



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



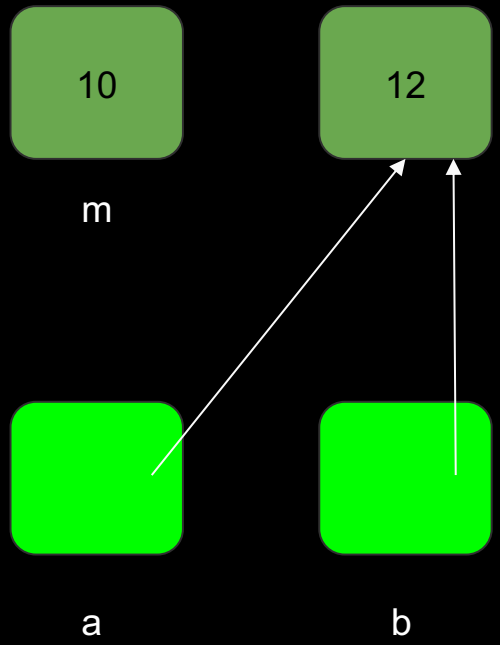
```
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
*b = m + 2;
```

inside memory

12 is assigned to location in memory where b is pointing

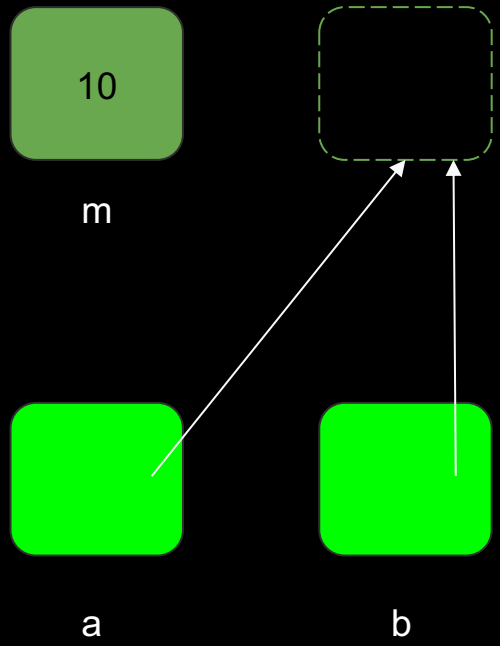
b is address in memory

*b goes inside memory

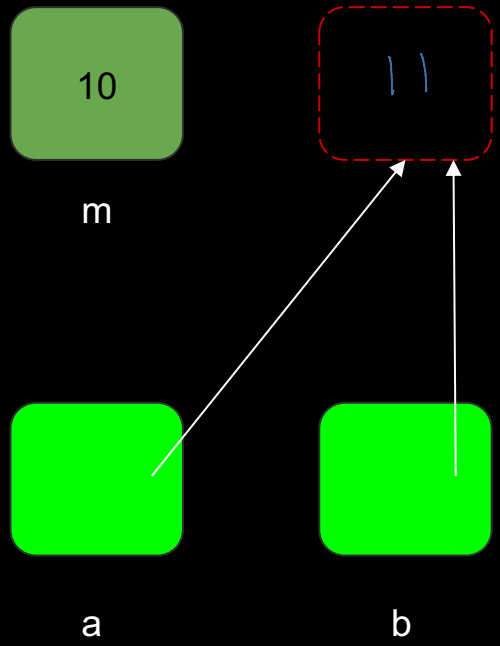


```
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
*b = m + 2;
free(a);
```

location in memory a was pointing to is freed



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



What happens if we do not free memory that we have allocated?

memory leak

Make sure to run valgrind on your code to ensure you don't have any memory leaks.

What happens if we malloc too many times?

run out of heap space

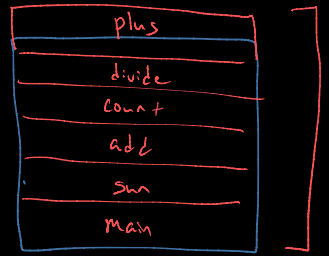
Heap

What happens if we call functions too many times?

run out of stack space

Stack

Stack Overflow



File I/O

fopen

fread

```
FILE *input = fopen("input.txt", _)
```

"r" – read
"w" – write
"a" – append

type — pointer — opens the file and returns an address in memory

fread – reads a file into some other variable, returns # of elements you read in

```
int val;

fread(&val, sizeof(int), 1, input)
```

&val: where we're reading to
sizeof(int): size of things we're reading
1: # things we're reading
input: file we're reading from

What if we've read everything already?

```
int check = fread(&val, sizeof(int), 1, input);
```

check = 0 b/c we read in 0 elements

```
while (fread(____) != 0)

while (fread(____)) {
    do stuff;
}
```

if the condition returns any nonzero value, it will continue

0 => Stop.

```
if (x == 0) {
}

if (!x) {
}
```

x is false => !x is true

x == 0

- 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 FILEs do we actually use pointers to files (aka FILE *).

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

include this!

- 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.

This accepts the actual file (Ex: card.raw)

- 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>);

```
FILE *input = fOpen("input.txt", "r")
if (input == NULL) {
    printf("couldn't open file");
    return 1;
}
```

card.raw

curd.raw ← return NULL pointer cs it doesn't exist

- 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".

input

char c = fgetc(<file pointer>);

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

c = fgetc(input);

fputc(c, output);

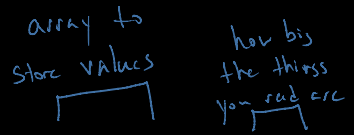
- fgetc() reads and returns the next character from the file, assuming the operation for that file contains "r". fputc() writes or appends the specified character to the file, assuming the operation for that pointer contains "w" or "a".

```
char c = fgetc(ptr1);

fputc('x', ptr2);

fputc('5', ptr3);
```

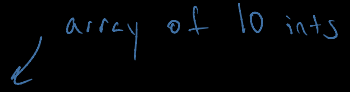
- fread() and fwrite() are analogs to fgetc() and fputc(), but for a generalized quantity (qty) of blocks of an arbitrary (size), holding those blocks in (or writing them from) a temporary buffer, usually an array, for local use within the program.



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.

JPEG Files

```c
#include <stdio.h>
# define  header1  0xff
# define  header2  0xd8
int main(int argc, char *argv[])
{
    // Check usage
    if (argc != 2)
    {
        return 1;
    }

    // Open file
    FILE *file = fopen(argv[1], "r");
    if (!file)
    {
        return 1;
    }

    // Read first three bytes
    unsigned char bytes[3];
    fread(bytes, 3, 1, file);

    // Check first three bytes
    if (bytes[0] == 0xff && bytes[1] == 0xd8 && bytes[2] == 0xff)
    {
        printf("Maybe\n");
    }
    else
    {
        printf("No\n");
    }

    // Close file
    fclose(file);
}
```

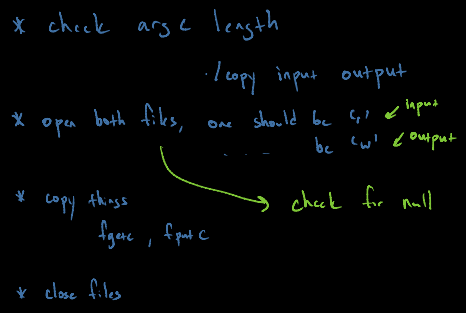
more concise way to write file == NULL

header1 header2 header1

JPEG headers start w/ some common sequence of bytes

Exercise

In copy.c, write a program that copies a text file. Users should be able to run ./copy file1 file2 to copy the contents of text file file1 into file file2.




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

int main(int argc, char* argv[]) {
    if (len(argc) != 3) {
        printf("oops");
        return 1;
    }
    FILE* input = fopen(argv[1], 'r');
    FILE* output = fopen(argv[2], 'w');
    if (!input || !output) {
        printf("oops");
        return 1;
    }
    while (true) {
        char c = fgetc(input);
        if (c == EOF) {
            break;
        }
        fputc(c, output);
    }
    fclose(input)
    fclose(output)
```

Hello

Lab

COPY HEADER

initialize buffer (uint8_t header[HEADER_SIZE])
fread into buffer
fwrite buffer into output

```
uint8_t header[HEADER_SIZE];
fread(header, HEADER_SIZE, 1, input);
fwrite(header, HEADER_SIZE, 1, output);
```

COPY & MULTIPLY

initialize buffer (int16_t buffer)
while stuff left to read:
  fread into buffer
  multiply buffer by scale
  fwrite buffer into output

```
int16_t buffer;
while (fread(&buffer, sizeof(int16_t), 1, input)) {
    buffer *= factor;
    fwrite(&buffer, sizeof(int16_t), 1, output);
}
```

CS50 Section 4