This is CS50

Section, Week 4

TA: Andi Peng

Agenda

- Announcements
- Pointers
- Memory Management, `malloc()`
- Redirection
- File I/O
- pset4

Announcements

- Grading
  - Commenting
  - Upload to the correct folder w/correctly named programs
- Yale's Wacky Schedule
- Quiz 0: October 14 or 15
  - Send in topics you would like reviewed next week!
  - andi.peng@yale.edu

Pointers

Pointers

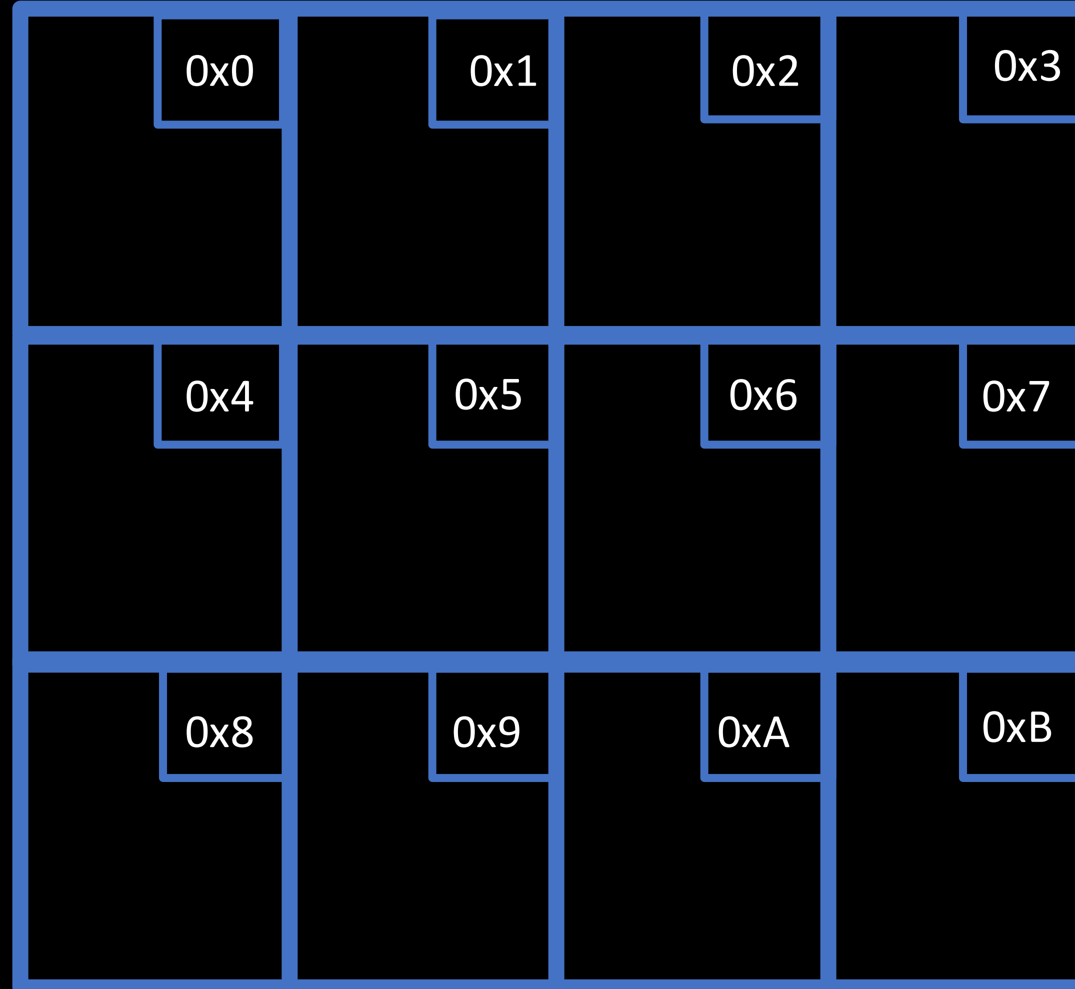
All variables have

- A value
- An address pointing to that value

Your computer has a finite amount of memory

- Pointers allow you to access specific areas from that memory

Memory



MAN, I SUCK AT THIS GAME.
CAN YOU GIVE ME
A FEW POINTERS?
0x3A28213A
0x6339392C,
0x7363682E.
I HATE YOU.

Creating Pointers

Declaring pointers:

<type>* <variable name>

Examples:

```
int* x;
char* y;
float* z;
```

Referencing and Dereferencing

& is the **reference, or address-of, operator**.
It returns the address in memory at which a variable is stored.

* is the **dereference operator.**
When the dereference operator is applied to a pointer, it returns the data stored at that memory address.

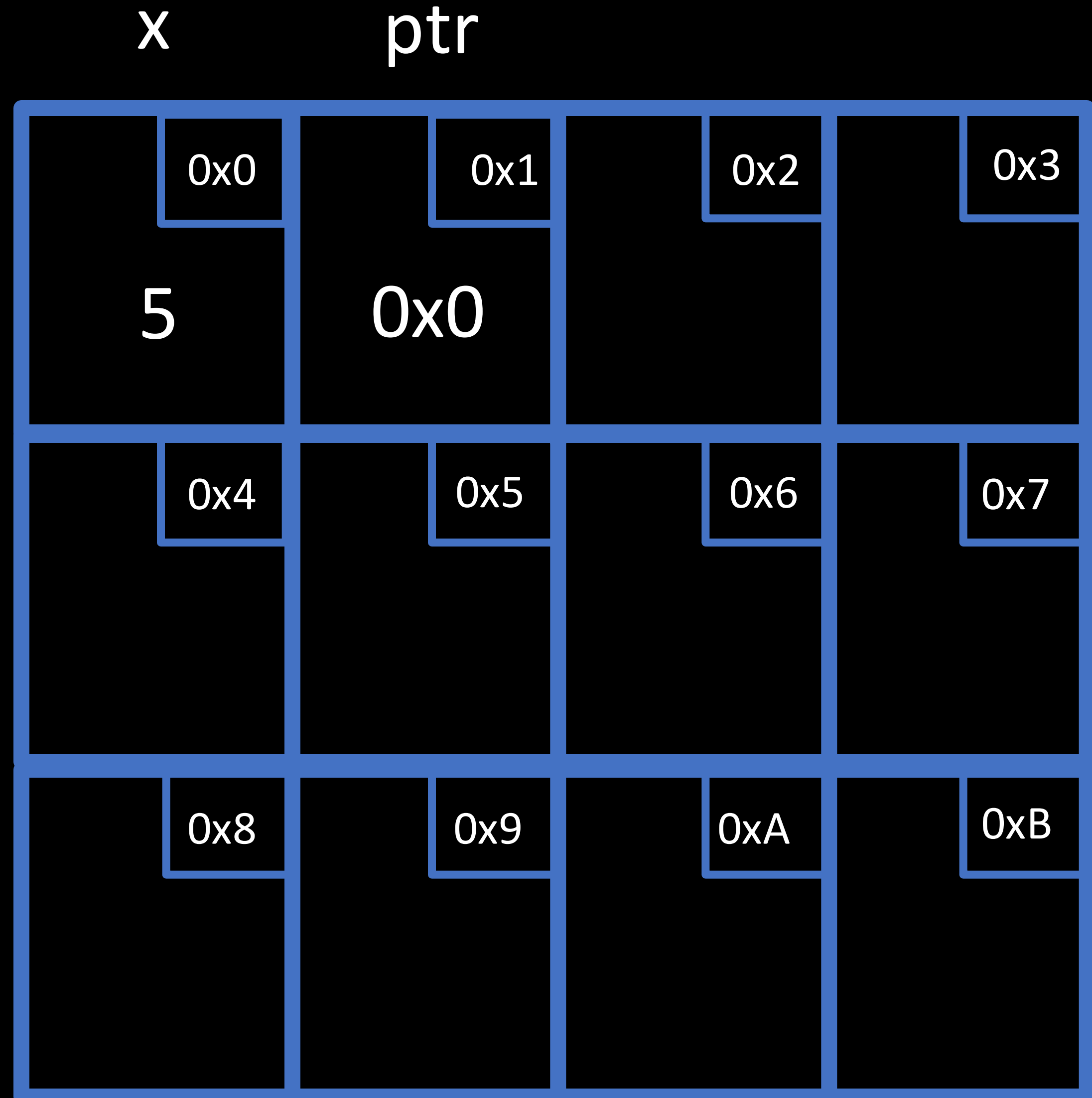
Referencing:
&<variable name>

Dereferencing:
***<pointer name>**

Pointers

A pointer's value IS an address

```
int x = 5;
int* ptr = &x;
```



Under the hood...

```
int x = 5;

int* ptr = &x;

int copy = *ptr;
```

Variable	Address	Value
x	0x00	5
ptr	0x01	0x00
copy	0x0C	5

Track the values

	x	ptr
int x = 5;	5	
int* ptr = &x;	5	&x
*ptr = 35;	35	&x

Test Yourself

```
int a = 3, b = 4, c = 5;
int* pa = &a, *pb = &b, *pc = &c;
```

	a	b	c	pa	pb	pc
a = b * c;						
a *= c;						
b = *pa;						
pc = pa;						
*pb = b * c;						
c = (*pa) * (*pc);						
*pc = a * (*pb);						

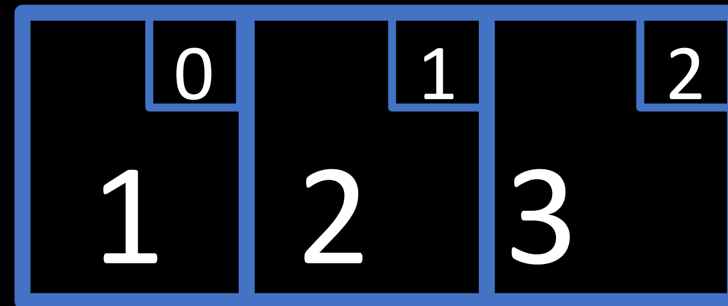
Answers

```
int a = 3, b = 4, c = 5;
int* pa = &a, *pb = &b, *pc = &c;
```

	a	b	c	pa	pb	pc
a = b * c;	20	4	5	&a	&b	&c
a *= c;	100	4	5	&a	&b	&c
b = *pa;	100	100	5	&a	&b	&c
pc = pa;	100	100	5	&a	&b	&a
*pb = b * c;	100	500	5	&a	&b	&a
c = (*pa) * (*pc);	100	500	10000	&a	&b	&a
*pc = a * (*pb);	50000	500	10000	&a	&b	&a

Arrays... are Pointers!

```
int array[3];

*array = 1;
*(array + 1) = 2;
*(array + 2) = 3;
```



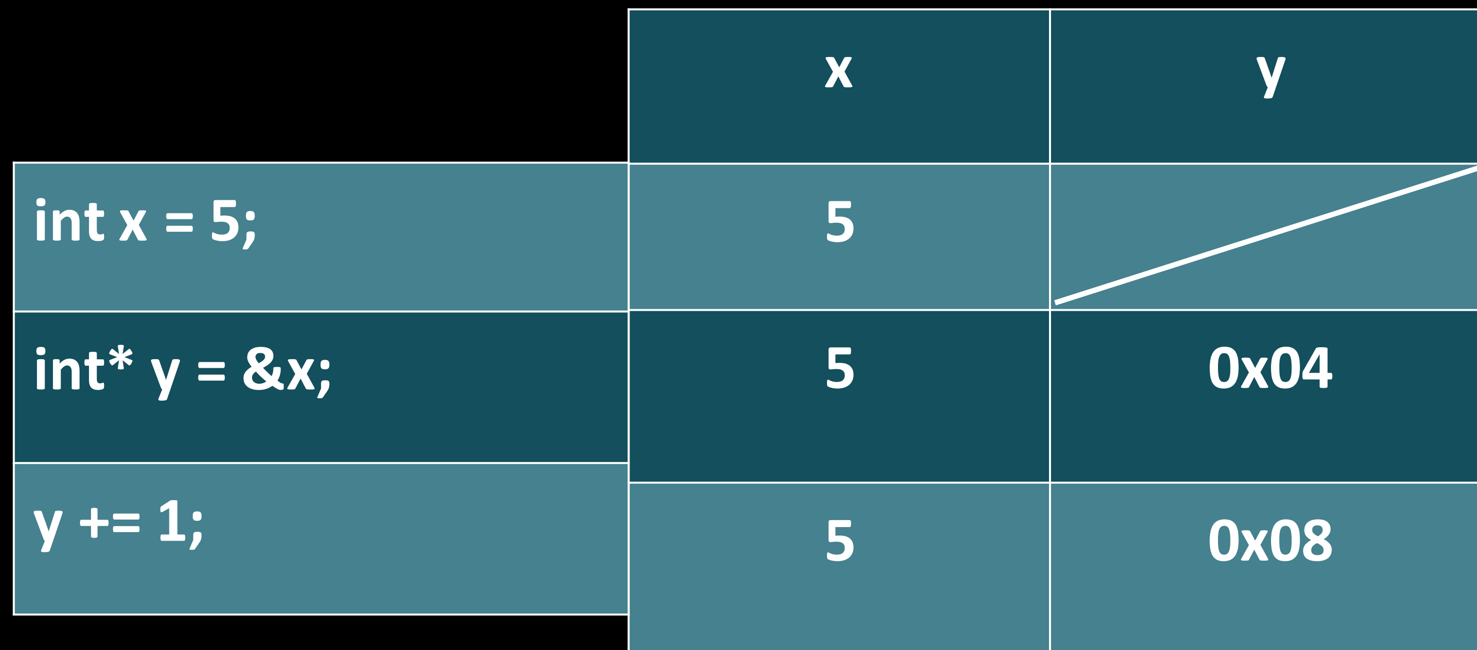
What will print?

```c
int main(void)
{
    char* str = "happy cat";
    int counter = 0;

    for (char* ptr = str; *ptr != '\0'; ptr++)
    {
        counter++;
    }

    printf("%d\n", counter);
}
```

Pointer Arithmetic

Adding/subtracting n adjusts the pointer by

n * sizeof(<type of the pointer>) bytes



	x	y
int x = 5;	5	
int* y = &x;	5	0x04
y += 1;	5	0x08

Two Strings Walk into a Bar…

One says, "I'd like a rum and Coke@[>t6#$%*kl."

The other says, "You'll have to excuse my friend, he's not null-terminated."

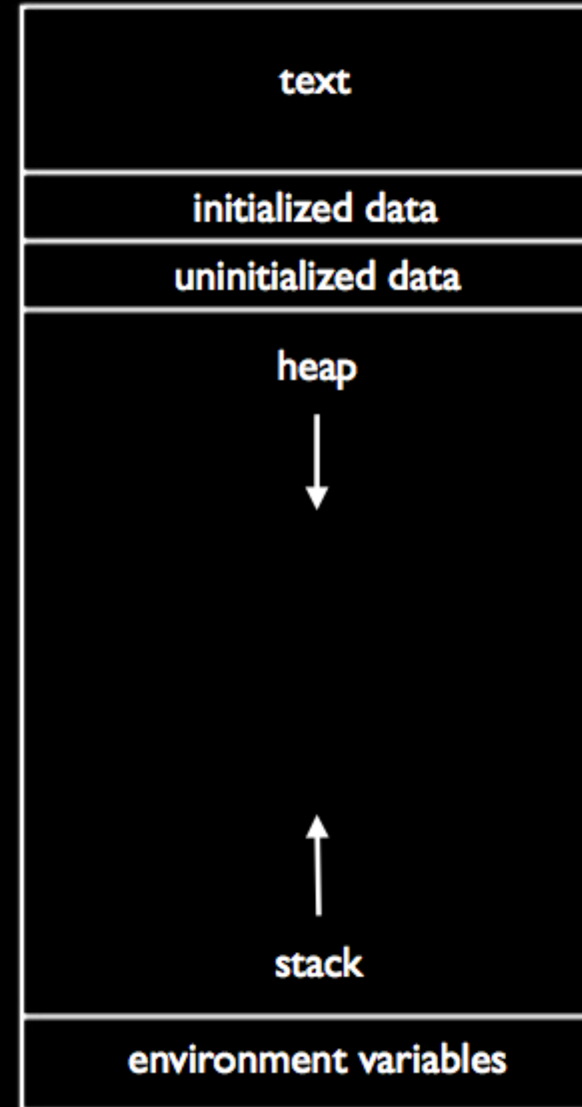
Dynamic Memory Allocation

2 basic regions of memory

- Heap
- Stack

Stack

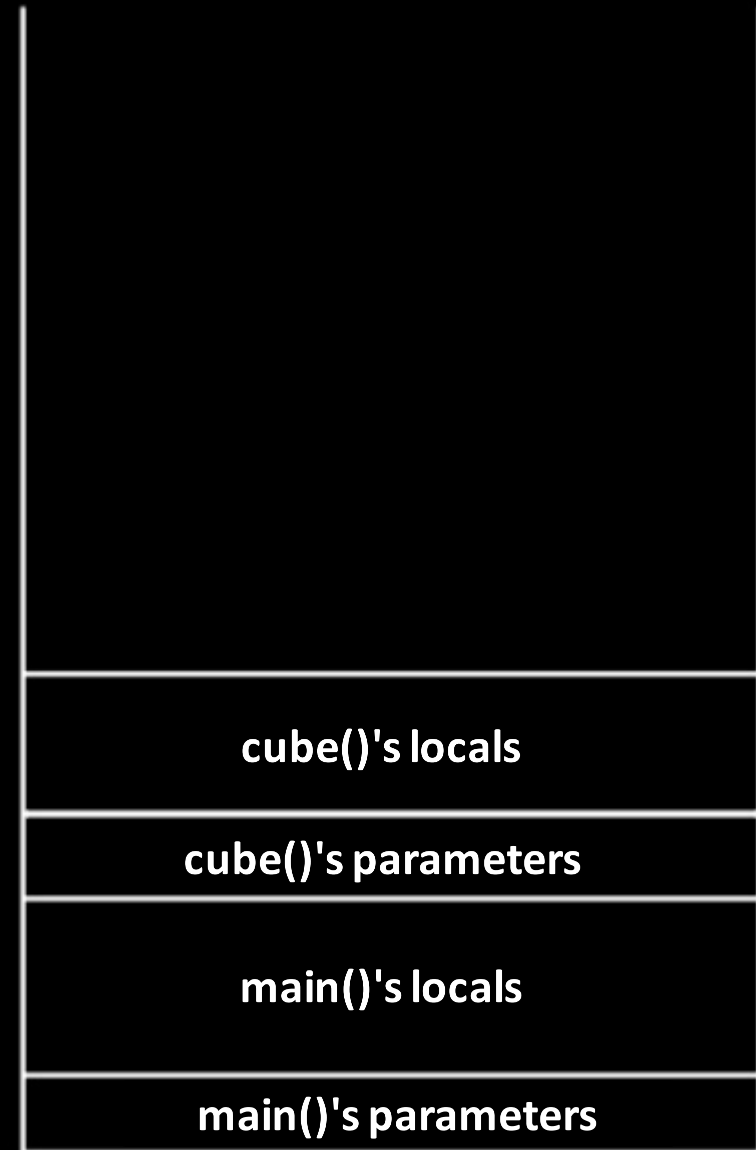
- a contiguous block of memory set aside when a program starts running
- LOCAL VARIABLES



Stack

- Each function that's called gets its own stack frame
- If we ever want to get to the variables held in `main()` again, we'd first have to *pop* the `cube()` stack frame off by returning
- Stack variables disappear when the function returns

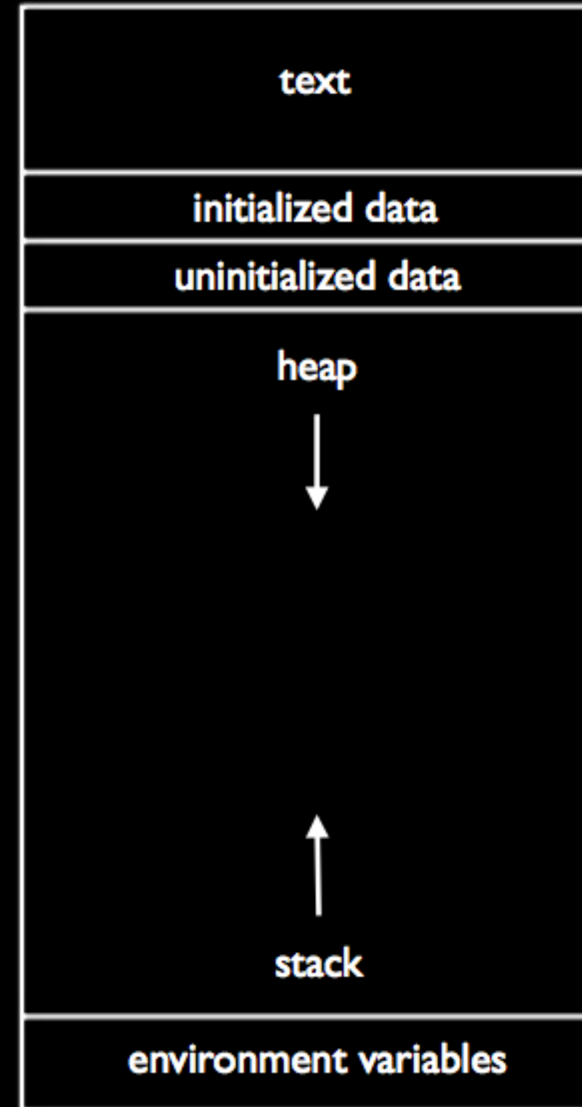
What if, when running a program, the size of a function's stack frame is dependent largely on its local variables, what if the number of variables and their sizes can't be determined before runtime? What if they depend on, say, user input? We can't just create a massive stack frame to cover all possibilities!



Dynamic Memory Allocation

Heap

- A pool of available memory that can be allocated dynamically while your program is running
- A region of unused memory that can be allocated with a call to `malloc()`



malloc()

A way to control your memory!

Creates a way for data to stay in your heap until you don't need it anymore

```
void* malloc(size in bytes);
```

example:

```
int* ptr = malloc(sizeof(int));
```

will create space for 4 bytes on the heap, returning a pointer to this space.

Check for NULL!

```
int* ptr = malloc(sizeof(int) * 10);

if (ptr == NULL)
{
        printf("Error -- out of memory.\n");
        return 1;
}
```

A call to `free()`

After you're done using heap memory, free it! (or else you'll leak memory and bad things will happen)

```
void free(pointer to heap memory);
```

example:

```
free(ptr);
```

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

int main(void)
{
    int* ptr = malloc(sizeof(int));
    if (ptr == NULL)
    {
        printf("Error -- out of
memory.\n");
        return 1;
    }

    *ptr = GetInt();
    printf("You entered %d.\n", *ptr);

    free(ptr);
}
```

Redirection

Main way we've been interacting with our programs is

- Standard input (`GetInt(), GetString(), etc`)
- Standard output (`printf()`)

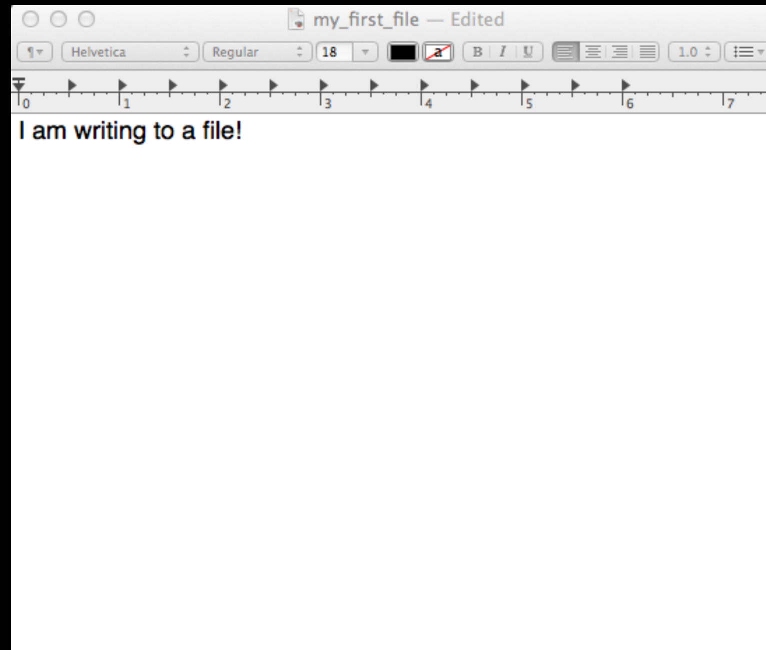
BUT there's more!

- `>`
- `|`

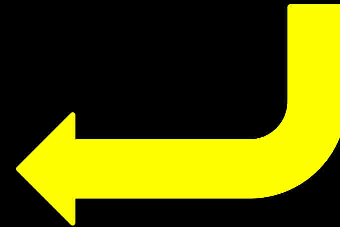
Redirection

We are used to reading from and writing to the terminal:

- read from `stdin`
- write to `stdout`



But we can also read from and write to files!



Redirection

- `>` -- output; print the output of a program to a file
    - `./hello > output.txt`
  - `>>` -- appends to an output file instead of overwriting the data
  - `2>` -- Like above but prints only error messages
- `<` -- input; use the contents of some file as input to a program
    - `./hello < input.txt`
- `|` -- pipe; take the output of one program and use it as input to another
  - `./generate 1000 | ./find 42`

Redirection

What do these commands do?

```
./hello < input.txt > output.txt
./find 42 < numbers.txt
./fifteen 3 < ~cs50/pset3/3x3.txt
```

File I/O

While redirection is very useful, C has a general mechanism for reading and writing files

1. Create a reference to the file.
2. Open the file.
3. Do all the reading or writing.
4. Close the file.

- `stdio.h`

Step 1: Create a reference to the file

```
FILE* file;
```

Step 2: Open the file (make sure it's not NULL!)

```
file = fopen("file.txt",    "r");
```

- 1st argument -- path to the file
- 2nd argument -- mode
  - "r" -- read, "w" -- write, "a" -- append

Step 3a: Read from the file

- `fgetc` -- returns the next character
- `fgets` -- returns a line of text
- `fread` -- reads a certain # of bytes and places them into an array
- `fseek` -- moves to a certain position

Step 3b: Write to the file

- `fputc` -- write a character
- `fputs` -- returns a line of text
- `fprintf` -- print a formatted output to a file
- `fwrite` -- write an array of bytes to a file

Step 4: Close the file

```
fclose(file);
```

Remember!

- Always open a file before reading from or writing to it
- Always close a file if you open it

File I/O – Your Turn!

Write a program that prints "Hello, World!" to a file

File I/O

Important things to keep in mind:

- `fopen()` creates a FILE pointer that points to a FILE struct
  - Passing around this reference to other functions like `fread()` or `fwrite()`
- If you forget `fclose()`, your program will leak memory
  - Bad things will happen
  - `fclose()` kind of *recycles* memory, returning it to the heap
  - Valgrind

Pset4: Forensics

BMP

- `copy.c`
- `whodunit.c`
- `resize.c`

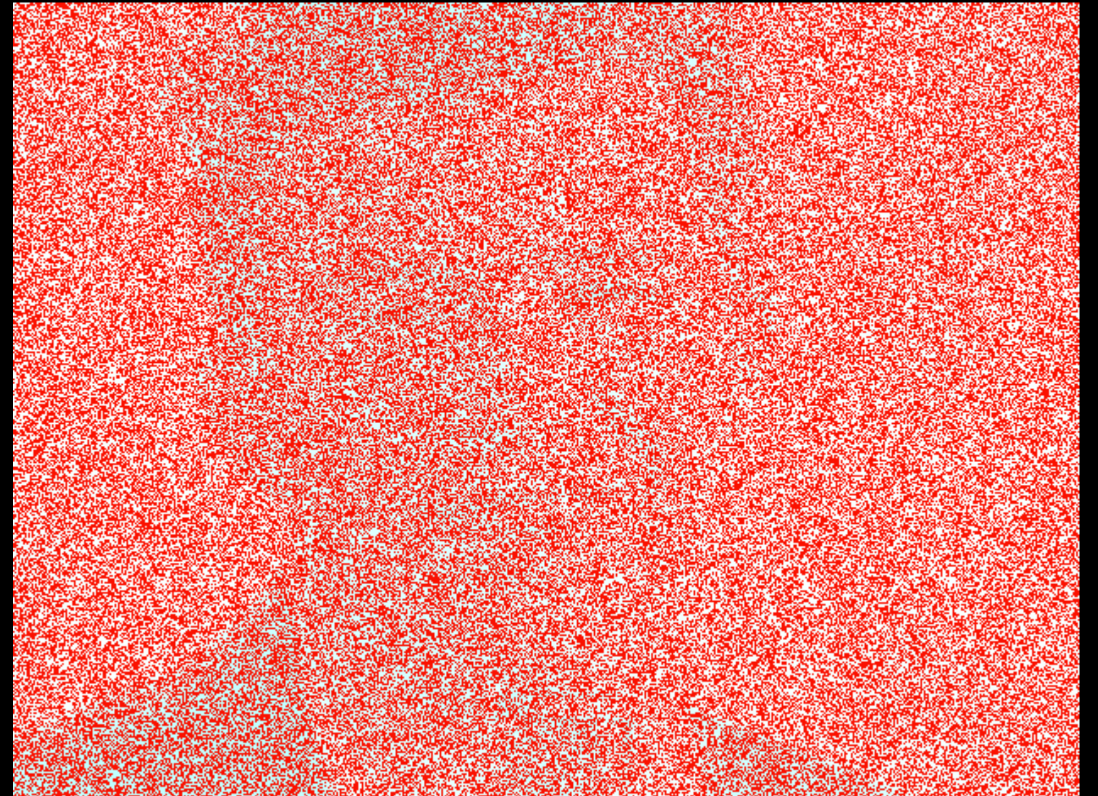
JPG

- `recover.c`

Pset4: Forensics

`whodunit.c`

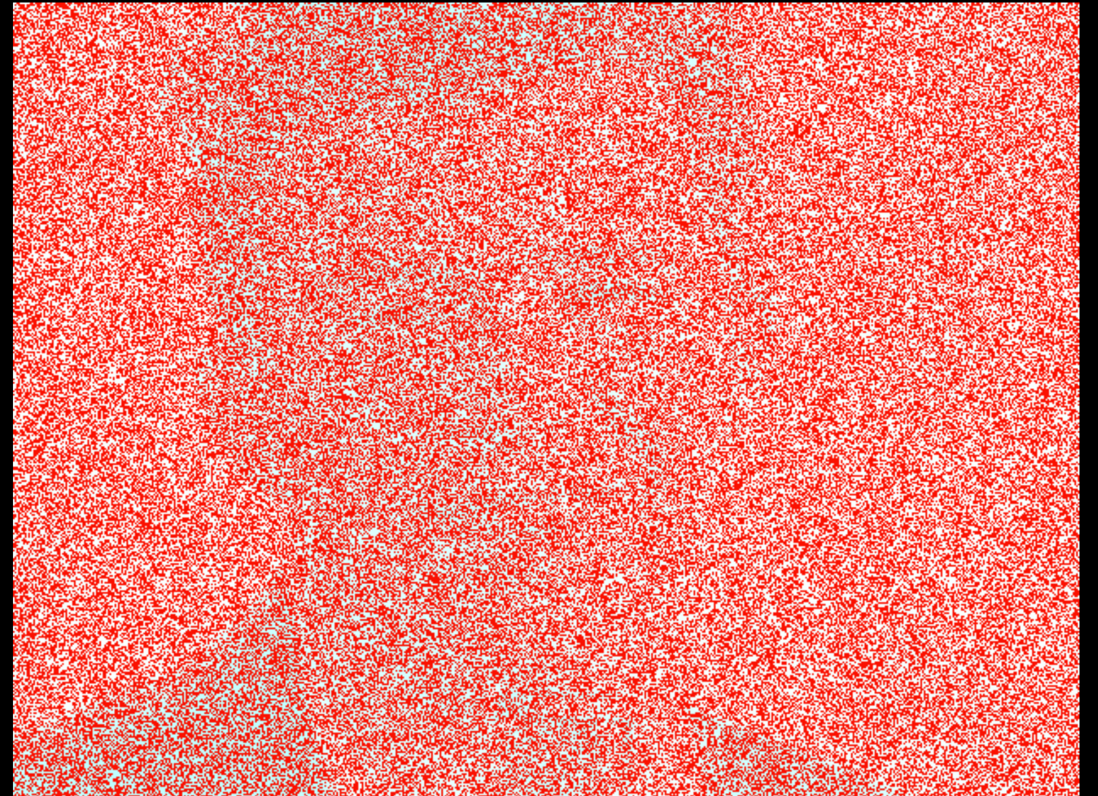
- Familiarize yourself with File I/O!
- How do you change the colors to clear up the picture?
- You would be surprised at how little code you actually need to solve the puzzle:P



Pset4: Forensics

`resize.c`

- What is the size of a BITMAPINFOHEADER, BITMAPFILEHEADER?
- Save dimensions of original
- Write new headers with dimensions after resize
- Write new RGB triples to outfile for scanlines



Pset4: Forensics

`recover.c`

- card.raw
- Track and store each individual JPEG as you find it
- Make sure to use an *unsigned* char buffer!
- How many bytes are in a CF card? That's how many you want to read into your buffer!
- Use `sprintf()` to generate custom filenames for each new JPEG
- Don't forget to close your JPEGs after you're done writing!