CS50 Section 4
Somewhere in Between

Annaleah Ernst, TF

The Agenda…

- Reeeeally quick recap
- Hexadecimal
- Structs
- Pointers
- Memory Management
  - Stack
  - Heap
  - malloc()
- Command Line Redirection
- File I/O

Recap

- Asymptotic notation
  - Big O notation
    - What's the big O of the four sorts we talked about last week?
  - Ω notation
    - What's the Ω of the four sorts from last week
  - Θ notation
    - What even is this?
      - It's when the O and Ω functions are the same
    - Which of our sorts have a theta function?

Recap

	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort
O	n^2	n^2	n^2	$n\log n$
Ω	n	n^2	n	$n\log n$
Θ		n^2		$n\log n$

Sizes of data types

Data Type	Size (bytes)
int	*4*
float	*4*
double	*8*
long long	*8*
char	*1*
string (char)*	*???*

Recap

- Recursion
  - Function calls itself from within itself
  - HIGHLY RECOMMEND you look at recursion on CS50 Study
  - https://study.cs50.net/
  - If you have questions, let me know, we can make time

Data Structures

- What have we learned about so far
  - Arrays
- Why do we use arrays?
  - Store a bunch of things of the same data type
- What are some limitations of arrays?
  - Can only store data of one data type

Structs

- Allow us to create out own data type to hold data of different type
  - Recall the student struct from lecture
  - What’s the difference between these two structs?

```
typedef struct
{
    int id;
    string name;
} student;
```

- This creates a new type called student
- To declare:
  - `student stu_1;`

```
struct student
{
    int id;
    string name;
};
```

- This creates a structure called student
- To declare:
  - `struct student stu_1;`

Structs: creating and accessing

- Declare using the struct name as the variable type
- Access using the . operator

```
typedef struct
{
    int id;
    string name;
} student;
```

```
student stu_1;
stu_1.id = 8;
stu_1.name = "John Smith";
```

Hexadecimal – base 16

- As computer scientists, sometimes we want to see what the computer sees
- Looking at long binary strings is tedious, so we often use hexadecimal
- Convenient for converting from binary
  - Each group of four bits is able to make 16 different combinations
  - Each group of four bits maps onto a single hexadecimal digit

Decimal	Binary	Hexadecimal
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	Hexadecimal
8	1000	0x8
9	1001	0x9
10	1010	0xA
11	1011	0xB
12	1100	0xC
13	1101	0xD
14	1110	0xE
15	1111	0xF

Hexidecimal

- Just like binary or decimal notation, hexadecimal has “places”
  - Remember elementary school: the 1’s place, 10’s place, 100’s place, etc
  - We can rephrase that as having a 10^0’s place, 10^1’s place, 10^2’s place, and so on
- Instead of being powers of ten or powers of two, hexadecimal has powers of 16
  - So we have a 16^0’s place, 16^1’s place, 16^2’s place, etc…
- How do we tell if a number is hexadecimal?
  - Preceded by 0x

	16^3	16^2	16^1	16^0
0x	*2*	*a*	*5*	*f*

Binary to Hex

01000110101000101011100100111101

0100 0110 1010 0010 1011 1001 0011 1101

		10		11			13
4	6	A	2	B	9	3	D

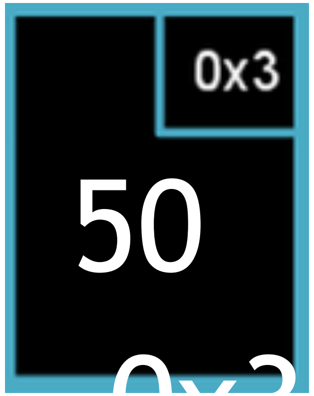
0x46A2B93D

Your turn! Base Questions

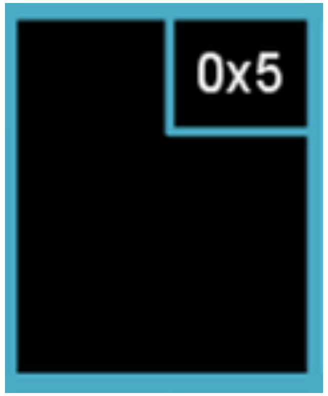
- What is 111111_2 in hexadecimal?
  - Divide into 4 bit sections: 0011 1111
  - Convert each section into base 16 value: 3 f
  - 0x3f
- What is 0xA5 in binary?
  - Reverse the above process: $A_{16} = 10_{10} = 1010_2$, $5_{16} = 5_{10} = 0101_2$
  - 10100101

Pointers

- Recall that computers have to store data in hardware, and we need to access it
- Every variable in memory has an address
  - Think about arrays and how we use the index as an address
- A pointer's value *is* an address
- Int x = 50;



- Int* ptr = &x;



Pointers

- Referencing
  - Get and work with the address of a variable (versus its value)
  - Passing by reference means you’re passing a variable by its address
  - Get the address of a variable: `&<variable name>`
- Dereferencing
  - Use an address to get an actual value
  - We use this to get the value the address is pointing to
  - Go to the value held at address: `*<pointer name>`

Pointers

```c
// declare an int pointer
int* ptr;

// get an address of a local variable and store it in ptr
int x = 50;
ptr = &x;

// go to address and get its value
printf("%i\n", *ptr);
```

Your turn!

- Sketch out what this code does on a piece of paper
- Ie, tell me what each of these lines do in concrete terms.
- What does each term equal after each step?

```
int x = 2, y = 8, z = 12;

int* ptr_x = &x;
int* ptr_y = &y;
int* ptr_z = &z;

z = x * y;
x *= y;
y = *ptr_x;

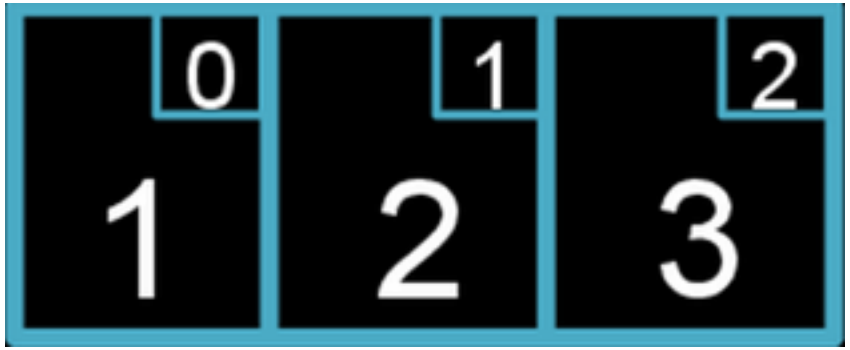
*ptr_x = x * y;
ptr_x = ptr_y;

x = (*ptr_y) * (*ptr_z);
```

Pointer and arrays

- Under the hood, an array is treated like a pointer to the first element
- Add the index to initial address to go to the next element

- int array[3];
- *array = 1;
- *(array + 1) = 2;
- *(array + 2) = 3;



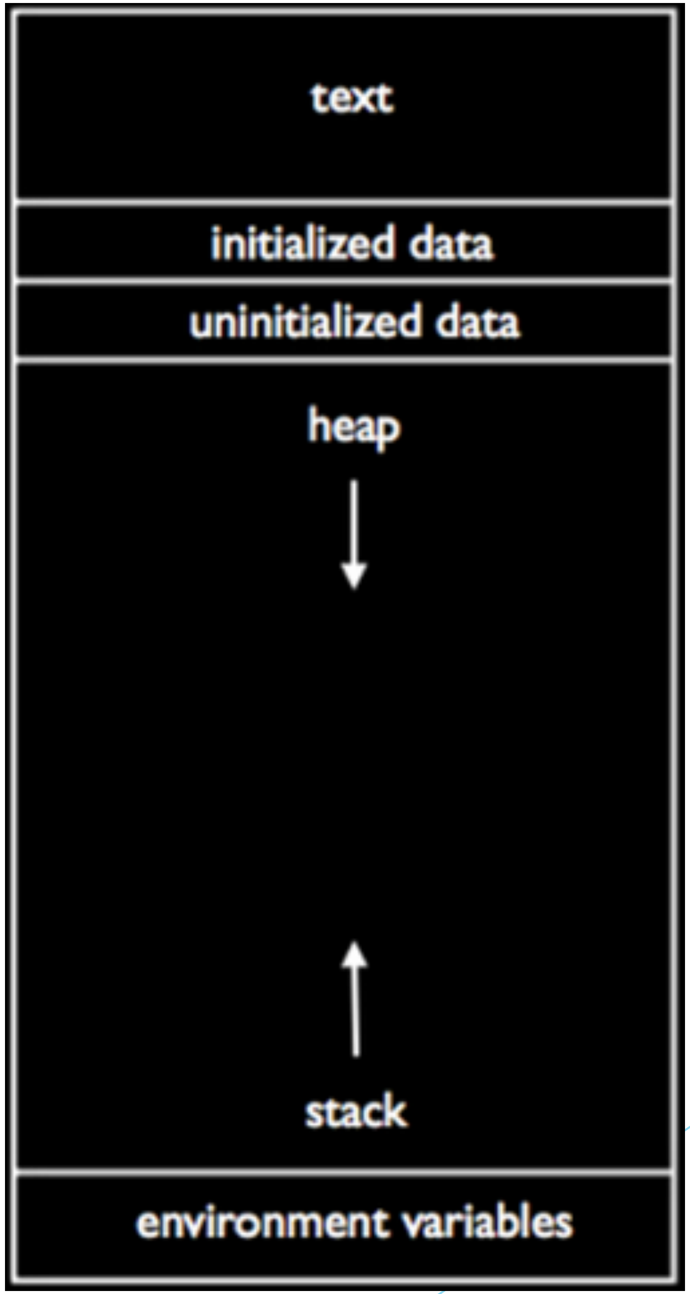
- Based on what we now know why do you think arrays are zero indexed?

Sizes of data types

Data Type	Size (bytes)
int	*4*
float	*4*
double	*8*
long long	*8*
char	*1*
string (char)*	*8 (on 64 bit architecture)*

Memory Management

- Two basic regions of memory
  - Heap
  - Stack



Memory – The Stack

- Contiguous block of memory set aside when program starts running
- LIFO data structure
- Each function gets its own stack frame
  - Metadata
  - Variables held in read only memory
  - Local variable
- When we call a function we *push* it on top of the stack
- To get at the contents of earlier frames, we need to *pop* it off
  - Ie, we need to return
- Size of stack frame largely dependent on local variables

- What if we don't know the number of variables/sizes?

Memory – The Heap

- We use the heap for memory allocated at runtime
  - dynamically allocated memory
- Region of unused memory that can be allocated with a call to malloc()
- Use malloc() to allocate memory on the heap
- malloc() allows us to give our pointers something to point to, eg

Memory – malloc()

- Give out pointers some persistent memory to point to, eg
  - Creates a space for 4 bytes on the heap
  - Returns a pointer to this space
  - This space can be passed between functions!!!!
    - Unlike stack variables, it won't be lost when a function returns.
  - Important to check for NULL pointer on call to `malloc()`

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

if (ptr == NULL)
    return 1;
```

- All calls to `malloc()` MUST be accompanied by a call to `free()` later in the program, eg
  - ```
    free(ptr);
    ```
  - All malloc()'d memory should be freed (but only once) and only malloc()'d memory should be free()'d.
  - Otherwise, we get memory leaks
    ```

Redirection

- At the command line, we can use redirection commands to control output
- `>` - output of program to file instead of stdout
  - Eg, `./hello > output.txt`
  - `>>` - append to output file instead of writing over data
  - `2>` - only print error messages to file
- `<` - input; use contents of file as some input to a program
  - Eg, `./hello < input.txt`
- `|` - pipe; take output of one program and use it as input in another
  - Eg. `A | B`
  - Output of a A becomes the input of B

File I/O

- So far, we've been printing output to stdout (think `printf()`), and getting input by prompting the user (think `get_string()`)
- It's just as easy to read and write files!
- `fopen()` - creates a FILE pointer to a FILE struct
  - Reference gets passed to `fread()` and `fwrite()`
- `fclose()` - gets rid of this pointer and prevents memory leaks
  - We'll talk more about leaks in memory management

File I/O - Commands

- FILE* fopen(<name of file>, <mode>)
- fread(<storage ptr>,<elt size>,<number of elts>,<file* stream>)
- fwrite(<cont info ptr>,<elt size>,<number of elts>,<file* stream>)
- fgets(<storage ptr>, <int size of string>, <file* stream>)
- fputs(<const char array>,<file* stream>)
- char fgetc(<file pointer>)
- fputc(<char c>, <file* stream>)
- fclose(<file pointer>)

File I/O - Structure

- Open file in appropriate mode (read, write, append)
- Check to make sure it opened
- <code>
- Before the program ends, close the file

File I/O

- Open file in appropriate mode (read, write, append)
- Check to make sure it opened
- <code>
- Before the program ends, close the file

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

int main(void)
{
    // open file "input.txt" in read only mode
    FILE* in = fopen("input.txt", "r");

    // always make sure fopen() doesn't return NULL!
    if(in == NULL)
    {
        return 1;
    }

    // open file "output.txt" in write only mode
    FILE* out = fopen("output.txt", "w");

    // make sure you could open file
    if(out == NULL)
    {
        return 2;
    }

    // get character
    int c = fgetc(in);

    while(c != EOF)
    {
        // write character to output file
        fputc(c, out);
        c = fgetc(in);

    }

    // close files to avoid memory leaks!
    fclose(in);
    fclose(out);
}
```

Your turn - iohello.c

- Use what we just learned about file I/O to write a textfile called hello.txt that contains the words: “hello, world!”
  - How could we have done this with redirection?