Assignment 6 — Introduction to C

Due: Monday, February 26th

After weeks of assembly, we have reached the figurative promised land: the C programming language. C is a powerful language, and pointers are one of its most powerful features. For this assignment, you'll gain experience with the basic functionality of C and its pointers by implementing an array-backed stack.

Deliverables:

Demo — *none*

Write-up — *none*

Handin — `stack.h`, `stackFuncs.c`, `stackDriver.c`
GitHub Classroom Link: `https://classroom.github.com/a/B3-S2HLc`

This is an **individual assignment**. You are not allowed a lab partner, nor are you allowed to work together, even if you write your own code. All your work must be your own.

Part 1: Ground Rules

- You may not use any global variables.
- All of your code must compile on Cal Poly's Unix servers using:

```
gcc -Wall -Werror -ansi -pedantic
```

Part 2: Push and Pop Functions

Begin by accepting the GitHub Classroom assignment: `https://classroom.github.com/a/B3-S2HLc`.

Complete the `push` and `pop` functions in `stackFuncs.c`, whose prototypes have already been written for you and copied into `stack.h`:

```
int push(int stack[], int *size, int val);
```

- Pushes a value onto a stack of integers.
- Takes three arguments:
  - `stack` — The array containing the stack
  - `size` — A pointer to the number of elements in the stack
  - `val` — The value to push
- Returns 0 on success, 1 on overflow.

```
int pop(int stack[], int *size, int *val);
```

- Pops a value off of a stack of integers.
- Takes three arguments:
  - `stack` — The array containing the stack
  - `size` — A pointer to the number of elements in the stack
  - `val` — A pointer to the variable in which to place the popped value
- Returns 0 on success, 1 on underflow.

Requirements:

- Do not alter the existing prototypes, do not alter `stack.h`, and do not use functions defined in other files. Your functions will be tested individually, using different drivers and header files.
- You may add helper functions if desired, *however*, their prototypes must be declared at the top of `stackFuncs.c`, *not* in `stack.h`.
- Your functions should grow the stack from low indices to high indices; that is, newer elements should be added at higher array indices.
- If an overflow or an underflow occurs, your functions must return an error code; they should not attempt to modify the stack.
    - Note that the maximum size of the stack, `MAX_SIZE`, is defined in `stack.h`.
- `push` and `pop` should only manipulate the given stack; they should not print anything.

Part 3: Printing the Stack

Complete the remaining function in `stackFuncs.c`:

```
void printStack(int stack[], int size, int mode);
```

- Prints a stack of integers.
- Takes three arguments:
    - `stack` — The array containing the stack
    - `size` — The number of elements in the stack
    - `val` — How to print elements, one of: `DEC_MODE`, `HEX_MODE`, or `CHAR_MODE`

Every value in the stack is an integer. However, as we now know from our experience with the LC-3, integers can be interpreted in multiple ways, and `printf` has format specifiers to reflect that.

If the given stack contains 1, 2, and 3, and the given mode is `DEC_MODE`, `printStack` should print:

```
[1, 2, 3]
```

. . . if the given stack contains 15, 7, 0, and 32, and the given mode is `HEX_MODE`, `printStack` should print:

```
[0xF, 0x7, 0x0, 0x20]
```

. . . if the given stack contains 65, 115, and 38, and the given mode is `CHAR_MODE`, `printStack` should print:

```
['A', 's', '&']
```

. . . if the given stack is empty, `printStack` should print:

```
[]
```

Requirements:

- Once again, do not alter the existing prototypes or `stack.h`, nor use functions defined in other files. Declare prototypes for any helper functions at the top of `stackFuncs.c`, not in `stack.h`.
- The elements should be printed in the order they appear in the array, from low indices to high indices.
- `printStack` should only print the stack; it should not manipulate it.

Part 4: Using the Stack

Develop a driver program to use your stack functions. This driver program will declare the necessary array and "size" variables to represent a stack, then read user commands and use the stack functions to manipulate the stack accordingly.

Requirements:

- Write your driver in a file named "`stackDriver.c`."
- Your driver may only interact with the stack using the `push`, `pop`, and `printStack` functions defined in `stackFuncs.c`.
- The driver must support the following single-character commands:
    - `+` — Pushes an integer onto the stack.
    - `-` — Pops an integer from the stack.
    - `d` — Switch to printing the stack as decimal integers.
    - `x` — Switch to printing the stack as hexadecimal integers.
    - `c` — Switch to printing the stack as ASCII characters.
    - `q` — Quit.
- You may assume that the user will hit "Enter" after typing each input, even if the expected input is only a single character long.
- The stack should be printed after every command except "`q`".
- If an unknown command is entered, do nothing; simply print the stack.

Sample Run:

```
Welcome to the stack program.

Enter option: +
What number? 1
Stack: [1]

Enter option: +
What number? 2
Stack: [1, 2]

Enter option: +
What number? 4
Stack: [1, 2, 4]

Enter option: x
Stack: [0x1, 0x2, 0x4]

Enter option: -
Popped 4.
Stack: [0x1, 0x2]
```

```
Enter option: -
Popped 2.
Stack: [0x1]

Enter option: -
Popped 1.
Stack: []

Enter option: -
Error: Stack underflow!
Stack: []

Enter option: +
What number? 97
Stack: [0x61]

Enter option: c
Stack: ['a']

Enter option: q
Goodbye!
```

(Where italicized characters are typed by the user. Your program will be tested using `diff`, so your output must match *exactly.*)