Background

In the APP C17-1 EXT assignment, you made a GUI in MATLAB to implement a **stack** data structure with **push** and **pop** commands. In this assignment, you will revisit this in C by simulating a **hardware stack**. A hardware stack is memory specifically assigned to be used as a stack for storing (pushing) and retrieving (popping) data before and after function calls. It is usually used for very low-level (assembly code level) operations and is not directly accessible by the programmer. Recall that you saw an example of assembly code and the push and pop instructions for a hardware stack when viewing the assembly code temporary file in the APP C22-1 EXT assignment.

A hardware stack has a **stack top** and a **stack bottom**, and a small number of data elements can be stored and retrieved in the memory spaces between these two spots (no data is stored at the stack top or at the stack bottom). A **stack pointer** hardware register holds the address of the current stack element being used, and this stack pointer is incremented and decremented as necessary to perform the push and pop commands. Commands are only executed at the stack pointer, and it is important to change the value of the stack pointer at the proper times. You will also use two **data registers**: **R1** and **R2**. These data registers are areas of memory set aside specifically to hold integer data which can be pushed onto the stack or popped off the stack. You will pass their data by reference into user-defined functions (this simulates how real hardware registers are accessed by using memory addresses).

For this program, the user will have four input choices: "`store R n`", "`push R`", "`pop R`", and "`end`". In these instruction commands, "`R`" can be either "`R1`" or "`R2`" and specifies which register will be used for the command, and "`n`" can be any integer value. Note that there must be spaces in between the commands, registers, and integers. If the user types "`store R n`", then the `store()` function is called to store the value of n into the register `R`. If the user types "`push R`", then the `push()` function is called to push the value in register `R` onto the stack. If the user types "`pop R`", then the `pop()` function is called to pop the value off the stack and into register `R`. If the user types "`end`", then the program ends. All other inputs will be ignored. You will have to determine which instruction command the user enters using simple string parsing.

The `store()` function will:

1. Scan the user input integer into the specified register.

The `push()` function will (in order):

1. Increment the stack pointer.
2. Store the value in the specified register into the new data element position pointed to by the stack pointer.

Note: The stack pointer will start at the stack bottom, but it cannot be moved to the stack top. So, if executing a push command would move the stack pointer to the stack top, then the stack pointer should not be incremented and the register value should not be moved onto the stack.

The `pop()` function will (in order):

1. Store the value in the data element position pointed to by the stack pointer into the specified register.
2. Decrement the stack pointer.

Note: The stack pointer can point to the stack bottom, but it cannot be moved beneath the stack bottom. So, if executing a pop command would move the stack pointer beneath the stack bottom, then the stack pointer should not be decremented and the register value should not change.

There is also a `show()` function which will be called every iteration through the main control loop to display the contents of the stack, the stack pointer, and the data registers.

This assignment also introduces a fun way of calling functions: by using a **pointer to a function**. Let's say we have the user-defined function prototype below:

```
void hello(int *);
```

Note that the return type of this function is `void`. If we want to declare and use a pointer to a function to call this function, we can write this:

```
void (*func_ptr)(int *);
int a, *int_ptr=&a;

func_ptr = hello;
func_ptr(int_ptr);
```

This works because the name of a function is a pointer to the beginning of the memory where the function is stored. Isn't that fun? For this assignment, you must use a pointer to a function to call the `store()`, `push()`, and `pop()` functions.

Lastly, this assignment also uses the `calloc()` function. This is very similar to `malloc()` but initializes all of the data to zero. This is used here to allocate memory for the stack and initialize its contents to zero, and also to initialize the data in the registers and initial input string.

Instructions

- Complete the C32-1 assignment which introduces user-defined functions in C.
- Copy `APP_C32_1_EXT_SKELETON.cpp` from:

  `/share/EED/class/engr1281/students/c/Class_32/Application`

- Begin by opening the skeleton code and adding your header information.
- Most of the code is provided for you, but you must write the code for the `store(), push()` and `pop()` functions as well as fill in the provided blanks in the `main()` and `show()` functions.
- A pre-compiled example output file is also provided for you to reference. `APP_C32_1_EXT_OUTPUT.out` can also be copied from the location above.