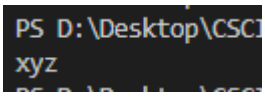
Version 1 - Basic Implementation

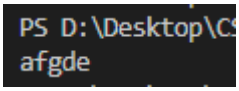
1. The program prints the Store array's contents one after another with no spaces in between them. Even after popping one element, 'z', it was still printed since the pop() method returns the element pointed at by the top variable and decrements it. It does not remove the popped element and it retains in the array. It only gets "erased" after pushing a new element, overwriting it in the process.

Output:



2. What's happening is that the pop() function only adjusts the top index and returns the popped value. It does not actually erase the element from the underlying Store array. This means the old value remains in memory, but it is considered "invalid" because top no longer points to it. When you later push() again, the new element overwrites the stale value at the same index. That's why the program output looked like 'afgde': the top variable moved back down after several pops, and when new elements were pushed, they reused those freed slots in the array

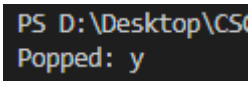
Output:



Version 2 - Encapsulation and Modular Programming

1. To be able to manipulate the stack inside the application function and outside of stack.c, I removed the "static" keyword from the top variable in stack.c, and added another top variable in app.c with the same signature and the keyword "extern". I then set the value of the top variable to be 2 so that I can manipulate the program to pop the 'y' element instead of the intended one, 'z'.

Output:



Version 3 and Version 4 - Structs

1. Structs with Pointers

In this implementation, the application program has access to the pointer to the stack itself. This means that it'll be easy to manipulate the stack from the application since we can simply dereference the pointer and get the contents of the stack. Here, I was able to print out all of the contents of both of the stacks, showcasing a weakness in using structs with pointers as an approach.

Output:

```
PS D:\Desktop\CSCI-
Stack s0: xa
Stack s1: yz
Popped from s1: z
```

2. Structs with Handlers

The implementation of stacks using structs with handlers show promise but it can easily be broken by pushing or popping to a stack that does not fit within the intended input. Here, I tried to push to stack 50 and pop at stack 100, but both attempts were not successful due to the implementation only having 10 as the max number of stacks. This also would mean that using a handler, would by default give you a MAX_STACKS number of stacks, whether you use all of them or not. Meaning, you could have been allocating memory for stacks that you are not going to be using anyway, wasting resources.

Output:

```
PS D:\Desktop\CSCI-70\HW1-
Invalid stack number 50
Popped from stack 0: z
Invalid stack number 100
```

Version 5 - OOP

Using OOP for implementing a stack in C is not really preferred because of the following reasons: **C isn't an object-oriented focused language** and **performance and overhead**

A stack is conceptually very simple: an array + a top pointer. Implementing a stack in C by wrapping it in OOP-like abstractions (such as structs with function pointers, "methods", etc.) adds layers of complexity for something that doesn't really need it.

C is often used in systems where efficiency in performance and memory matter. Adding OOP-style constructs in C increases memory usage for something that can be implemented in more efficient ways. Plus, C doesn't have built-in OOP support. No classes, no inheritance, no polymorphism. You can fake it with structs + function pointers, etc. but it's generally not ideal. Other languages like Java, C++, Python are more suited for that since OOP is built in and natural.