Assignment 5

DTU Compute
Technical University of Denmark
02102 Introductory Programming
08/05/2022

Group 34

Polly Clémentine Nielsen Boutet-Livoff - 214424

As I am the only member of the group, all the work was done by me.

1 Letters

In order to count the amount of "letters"[1] using strlen, we simply have to call it on every argument passed to our program. There are many edge cases when counting characters.

```
$ letters Hello World
10
$ letters "Hello World"
11
$ letters a
1
$ letters
0
$ letters Punctuation!
12
$ letters Punctuation
11
$ letters ∅
2
```

As we can see, parsing `Hello World` as two arguments or one argument surrounded by quotes changes how it's interpreted [2]. Not giving the program any characters results in it simply counting to 0 (We might prefer an error, depending on our use case). We note that the program counts characters, including spaces or exclamation points, not letters. Except that it doesn't quite count characters, it counts chars. Which just means that it counts how many bytes the strings we give are encoded as, which is not the same as what a user might expect a character to be. You might think only giving it ASCII would resolve this as we don't run into emoji, Latin (but non-English) letters which are all encoded as more than 1 byte, however even ASCII has some spooky characters. All of `0x00-0x1F` in ASCII are control characters, which are valid ASCII but nonetheless will not be understood by users.

1.1 Stack

The stack is a simple and fundamental data structure in computer science. Most of the methods are trivial to implement, but `push` is a bit harder. `empty` simply returns whether or not the stack is empty. Since we store the size we can simply check if the size is zero and return the appropriate "boolean" value. The function `top` returns a pointer to the top element in the stack. This function has two pitfalls, the edge condition of an empty stack and a potential off-by-one error. We decide to crash if the function is run on an empty stack [3]. and make sure that the array arithmetic is correct. `pop` is the same as top except we "remove" the top element, which is effectively just decrementing the size. `newStack` initializes the stack and allocates the required heap memory. The last function `push` first checks if we have enough space to store the new integer. If we don't we double the capacity and reallocate (and crash if we can't reallocate). To store the new int we simply increment the size and put it and the first unused position of the array.

[1] The assignment specifies that we're counting characters actually.

[2] Arguably, this behavior stems from bash, but similar noteworthy edge cases occur in almost every command line interface to run executables.

[3] A better design might be to return an error, however since the function signatures have already been decided it's difficult to do.

The data structure does not deallocate itself and has no helper function to deallocate it. In order to do so you must first free the array and then the stack struct. It's easy for a user of our library to accidentally leak or double free memory. Additionally, the stack only works for integers. It would be nice to have it work for

1.2 The C University Data Base

When the program is run, it displays a user friendly interface that prompts one of 3 options; Quit, list all students in the database or add a new student to the database. The first two options are self-exploratory. The third option asks our user for a name, start year, start semester and GPA before (permanently) adding them to the database. There is no delete or modify operation. All inputs are validated before we attempt to store them in the database. There are helpful messages for invalid input, or input that makes little sense (like listing all students when there are no students). The following is a demonstration of the software.

```
$ cudb
Welcome to CUDB - The C University Data Base
0: Halt
1: List all students
2: Add a new student

Enter action: 1

There are no students in the database.

Enter action: 2
Enter a name (4 characters only): Polly
The students name has to be 4 characters or shorter.

Enter action: 2
Enter a name (4 characters only): Poly
Enter start year (2009-2040): 2021
Enter start semester (0=Autumn/1=Spring): 0
Enter GPA (0-255): 127
Student added.

Enter action: 2
Enter a name (4 characters only): Niko
Enter start year (2009-2040): 2021
Enter start semester (0=Autumn/1=Spring): 1
Enter GPA (0-255): 420
GPA must be between 0 and 255 (inclusive).

Enter action: 2
Enter a name (4 characters only): Niko
Enter start year (2009-2040): 2021
Enter start semester (0=Autumn/1=Spring): 1
Enter GPA (0-255): 200
```

```
Student added.

Enter action: 1

Number Name Year Start  GPA
s0000  Poly 2021 Autumn 127
s0001  Niko 2021 Spring 200

Enter action: 4
Invalid action. Select a valid option:

0: Halt
1: List all students
2: Add a new student

Enter action: -1
Invalid action. Select a valid option:

0: Halt
1: List all students
2: Add a new student

Enter action: 0
Dropping the database, persistence is for chumps anyway.
```

The demonstration is not complete, some of the error handling in the `add_student` function has not been shown, however it is documented in the code and has been thoroughly tested on my own. I decided to deviate from the interface given in the assignment a little as I wanted to provide a better UX. With regards to input I had some struggles. Simply using `scanf("%d", ...)` would read an integer from the user input correctly but also wouldn't read a whole line, leaving a newline in the input buffer [4]. I tried dumping the input buffer after using `scanf` but that would result in the program hanging since a `getchar` or similar function call would wait for input if the input buffer was empty. Additionally, the design seemed unlikely to function correctly in different environments. I decided instead to read up to 100 characters into a buffer, and the parse on the buffer. This would allow me to fully empty[5] the input buffer while also not hanging after user input. Additionally, this let me handle cases where I got too much input. If the user types in a name that is too long (over 4 characters) the program will recognize it and notify the user, instead of silently trimming it (which might lead to undesirable results).

Most database have many features that this one doesn't have: persistence, updating items, deleting items, filtering, more than 10 thousand rows, etc. The program I have developed lives up to the requirements for the assignment, but it is a very bad database.

[4] You can end up with a lot more than just a newline in your buffer, but it doesn't matter

[5] as long as no one types in more than 99 bytes