Mahdeen Ahmed Khan Sameer
Professor Max Bender
CS333
09 March 2024

Project 03: C Syntax and More!

Google site: https://sites.google.com/colby.edu/sameer-again/home

Abstract:

In this project, we explore the basics of C programming and the syntax of a chosen programming language. We start by creating a stack in C, using arrays to manage a list of integers, and implement functions to manipulate this stack. This includes creating, checking if it's empty or full, adding and removing elements, and displaying the stack in both original and reversed order. We also delve into memory management by demonstrating how to prevent memory leaks with our stack implementation. In the second part, we focus on learning a new programming language by writing sample programs. These programs illustrate naming rules, variable declarations, scope, binary search, and the use of built-in and aggregate types.

Results:

Part 01:

First, we focused on implementing a stack data structure using arrays. We began by defining the structure of our stack, which included pointers to the stack's data and its top element, along with its capacity. Following the structure definition, we created a header file, `cstk.h`, containing declarations for essential stack operations such as creating a stack, checking if it's empty or full, adding and removing elements, displaying the stack, destroying the stack to free memory, and copying the stack. We then implemented these functions in a separate `cstk.c` file. To ensure our stack implementation worked correctly, we used a provided test file, `cstktest.c`, which required us to compile and execute it along with our implementation. The test file helped us identify and fix issues until our stack passed all tests successfully. Additionally, we explored memory management by modifying the test file to repeatedly create and destroy stacks, ensuring our `stk_destroy` function effectively prevented memory leaks.

This aspect was demonstrated through a video showing that our program did not consume additional memory despite the continuous creation and destruction of stack instances, highlighting efficient memory usage.

Please check the video in the project folder called: video.mov. Now, Let’s draw the stack and heap (with address labels) for the code at Mark 1 and Mark 2 in this c file and include text explaining our drawings. To do this, we will need to add code to print out the addresses of the pointers. Here is a model of what we are looking for, but with a simpler example.

Mark 01:

Our stack is characterized by two primary variables: a pointer designated as 's' and an integer labeled 'i'. The pointer 's' is assigned the memory address of our Stack's structure, a location secured within the heap's territory, identifiable by the address 0x16f0ff068. Meanwhile, 'i' functions as a loop counter with its own allocated space at the memory address 0x16f0ff064. The term 'Top' within our structure corresponds to an integer that marks the position of the stack's uppermost element, whereas 'capacity' is another integer that denotes the stack's limit, which we have defined as 20 elements. The pointer named 'data' serves as a reference to a sequence of integers forming the stack, its address being 0x11ce06ba8, which directs us to the initial element of the array resting on the heap.

Furthermore, this array within the heap is populated by integers ranging from 1 to 10, sequenced through the execution of the 'stk_push' operation within our loop. At the juncture known as Mark 1, the intricate relationship between the stack—with its pointers—and the heap—housing the actual data arrays—is made evident. Notably, the memory addresses we observe in the visual representation, such as 0x16f0ff068, 0x16f0ff064, and 0x11ce06ba8, are instance-specific to this particular run of the program and are subject to change with each execution cycle, underlining the dynamic nature of memory allocation within such programming environments.

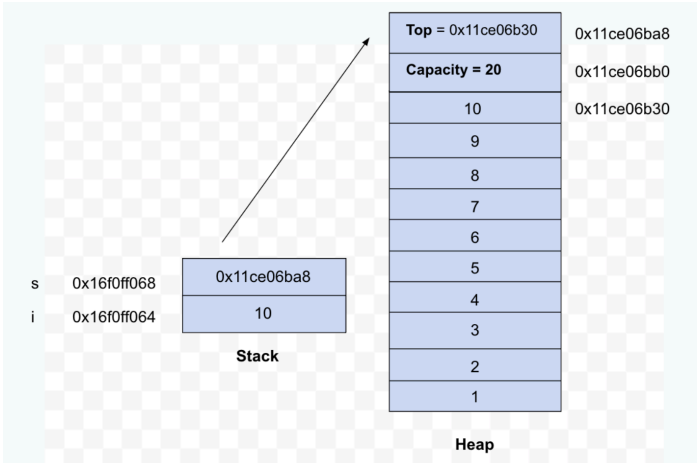

Figure: Mark 01 stack

Mark 02:

Following the execution of the stk_destroy() function, the heap has been cleared. The space that was once taken up by the Stack structure and its related data is now released. Consequently, the memory address 0x11ce06ba8, which used to refer to the 'data' array, is now invalid. This freed-up space in the heap is now accessible for future malloc calls.

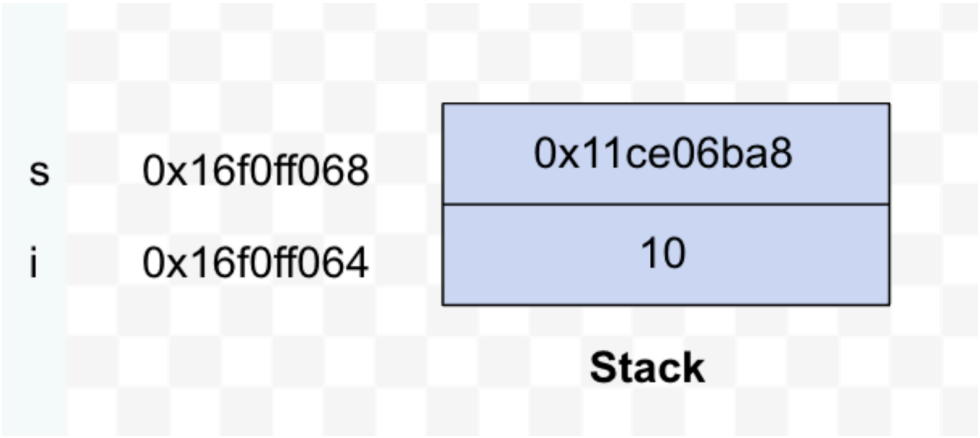

Figure: Mark 02 Stack

Part 02:

Please check the Google Sites.

Extensions:

Extension 01:

In this extension, we made the classic Stack data structure by implementing it in JavaScript, drawing inspiration from a similar structure originally implemented in C. Our adaptation maintains the fundamental operations of the Stack, such as push, pop, peek, and checking if the stack is empty or full, while introducing additional functionalities to enhance its usability and efficiency. Notably, we implemented a dynamic resizing method, allowing the stack to automatically expand its capacity when needed, thereby overcoming the limitations of a fixed-size stack. This feature ensures that the stack can adapt to varying data volumes without manual intervention. Furthermore, we included methods to display the stack's contents, either from the bottom up or in reverse order, copy the stack, and clear its contents, enhancing its versatility.

Extension 02:

In our second extension, we started with some of the distinctive features and syntactical elements that set JavaScript apart as a flexible and highly popular programming language. By touching on a variety of concepts including template literals, destructuring assignment, arrow functions, higher-order functions, closures, immediately invoked function expressions (IIFE), prototypal inheritance, object literals, and the asynchronous programming paradigm through Promises and async/await, we shine a light on JavaScript's rich expressive capabilities and adaptability. The examples span from straightforward string manipulation and array processing to intricate patterns like closures for data encapsulation and async programming for handling time-consuming operations. The first example makes use of template literals for seamless string construction and variable integration. The second example emphasizes the streamlined syntax of arrow functions and leverages higher-order functions such as map and reduce for efficient array operations. The third example demonstrates the use of closures and IIFEs to safeguard data within private scopes. The fourth example explores the core of JavaScript's object-oriented approach through prototypal inheritance and the use of object literals for defining and augmenting objects. The final example ventures into the realm of asynchronous programming, illustrating how Promises and async/await syntax can simplify the handling of operations that occur over time, making the code more intuitive and straightforward to follow.

Extension 03:

We also tried to use Python and JavaScript, offering similar features in terms of conditional statements and comparison mechanisms. Yet, they diverge significantly in syntax and unique language characteristics. Python employs indentation to structure code blocks, utilizes "def" for function definitions, and "elif" for additional conditional branches. On the other hand, JavaScript distinguishes code blocks with curly braces {}, defines functions with the "function" keyword, and employs "else if" for further conditional statements. Variable declaration in Python is straightforward, involving direct assignment without the need for specifying a type. Conversely, JavaScript requires the declaration of variables using "let," "const," or "var," also without type specification. In string manipulation, Python uses f-strings, marked by an "f" and curly braces {} for incorporating variables into strings. JavaScript, however, opts for template literals indicated by backticks and the ${} syntax for similar purposes. To output data, Python relies on the "print()" function, whereas JavaScript uses "console.log()." Both languages share common comparison operators like ">", "<", ">=", "<=", "==", and "!=". However, for checking equality and identity, Python distinguishes between "==" (equality) and "is" (identity), while JavaScript differentiates between "===" (strict equality, including type) and "==" (equality with type

coercion). Execution environments also differ: Python code runs on interpreters like CPython, and JavaScript code executes within web browsers or through runtime environments such as Node.js.

Extension 04:

In this extension, we demonstrate the use of JavaScript's native `indexOf()` method as a practical approach to performing linear searches within arrays. The `indexOf()` function is a powerful, built-in JavaScript method designed to search through an array sequentially from the beginning to find the first occurrence of a specified element. Upon finding the element, it returns the index at which the element is located. Conversely, if the element does not exist within the array, the method returns `-1`, signaling the absence of the target element. Here, we've prepared an array named `numbers`, populated with a series of even integers ranging from 2 to 20. Our goal is to locate the position of a target value, in this case, `12`, within the `numbers` array. By calling `indexOf(target)` on our array and passing in the target value as the argument, we initiate a search for the value `12`. The outcome of this operation is then stored in a variable named `index`. Depending on the result stored in `index`, we employ a conditional statement to determine the output. If `index` is not equal to `-1`, indicating that the target has been found, we print a message stating the target's value along with its index position in the array. If `index` equals `-1`, we convey through a message that the target value is not present in the array. This example illustrates not only the functionality of the `indexOf()` method in searching for array elements but also emphasizes JavaScript's capability to facilitate straightforward and efficient data search operations.

Extension 05:

In this extension, we took on the challenge of showing how JavaScript can be used to write very complex and hard-to-understand code, on purpose. Our goal was to highlight the importance of clear and easy-to-read code. We chose to create a binary search function, which is a way to find a specific item in a sorted list by repeatedly dividing the search interval in half. However, we wrote it in a way that makes it really hard to figure out what's happening.

First, we used very short and vague names for variables and functions. For example, we used `b` for the binary search function, `a` for the list it searches through, and `t` for the item it's looking for. These names don't tell you anything about what they're for, making the code harder to follow. Then, we squeezed the whole function into one line. Normally, code is spread out over several lines with spaces and indentations that make it easy to see the structure and flow. We skipped all that, packing everything tightly together. This makes it hard to see where one part of the code ends and another begins. We also relied heavily on the ternary operator, a shorthand way of writing if-else statements. While it can make code shorter, using it too much, especially in complicated situations, can make it very difficult to understand what conditions are being checked and what happens as a result. We didn't include any

comments in the code. Lastly, we ignored common practices that help keep code readable, like putting spaces around operators and using meaningful names for variables. This goes against the recommendations for making code that's easy for others to read and work with.

Extension 06:

In this extension, we explore the artistic intersection of poetry and programming by crafting a unique piece that functions both as a runnable JavaScript code and a poetic expression. The script begins with a haiku dedicated to the changing skies over Waterville, Maine, capturing the essence of nature's cyclical drama through concise and vivid imagery, printed line by line using `console.log()` statements. Following the haiku, we introduce a function named `generateWeatherPoem()`, designed to dynamically generate a random weather-themed poem each time the script runs. This function utilizes four arrays containing words associated with the seasons (`seasons`), descriptive adjectives (`adjectives`), action verbs (`verbs`), and natural elements (`objects`). By randomly selecting one element from each array, we construct three lines of poetry that collectively depict a scene reflective of Waterville's weather, leveraging template literals for seamless word integration.

Executing this script first presents the reader with the haiku, immediately followed by a randomly generated poem. The output showcases not only the beauty of Waterville's landscapes across different seasons but also demonstrates the playful potential of code to generate ever-new poetic imagery. Each execution promises a fresh poetic creation, inviting the reader to experience the unpredictable beauty of nature as mirrored in the unpredictability of the poem's composition.

Extension 07:

In JavaScript, functions are treated as first-class citizens, meaning they can be manipulated in ways similar to objects. This includes assigning them to variables, passing them as arguments to other functions, and returning them as values from functions. Although functions aren't classified as a "basic data type" in the traditional sense, their flexibility allows them to be stored in variables and treated much like any other data type. This capability is exemplified in several ways. A function can be declared and then assigned to a variable, as shown with the `greet` function being stored in `sayHello`. This variable can then be used to invoke the function. Similarly, JavaScript supports the assignment of anonymous functions to variables, such as the `multiply` function, and the use of arrow functions assigned to variables, like the `square` function. These examples highlight JavaScript's dynamic nature, where functions can be passed around and invoked through variables. The ability to assign functions to variables underscores JavaScript's embrace of functional programming concepts, allowing for higher-order functions, closures, and function composition. This approach enables a level of abstraction and reusability

not readily available in languages that do not treat functions as first-class citizens. Therefore, while functions may not be a "basic data type" in the strictest definition, their first-class status in JavaScript.

Extension 08:

In extension 08, we aimed to enhance the robustness of the stack data structure, originally implemented in C, to efficiently manage overflow errors and enable automatic expansion when its capacity is reached. To achieve this, we introduced a dynamic resizing mechanism within the `stk_push` function, allowing the stack to grow in size dynamically, effectively mitigating the risk of overflow errors that could occur with a fixed-size stack. Additionally, we expanded the stack library by adding a variety of functions to increase its functionality and usability. For instance, the `stk_create` function initializes a new stack with a specified capacity, allocating memory for its elements. The `stk_empty` and `stk_full` functions check whether the stack is empty or full, respectively, providing crucial checks that aid in avoiding underflow and overflow conditions. To address the need for dynamic resizing, the `stk_resize` function doubles the stack's capacity and relocates its elements to a new, larger memory area, ensuring the stack's capacity can adjust according to its usage. The `stk_push`, `stk_peek`, and `stk_pop` functions are fundamental operations allowing for adding to, viewing, and removing from the top of the stack, while ensuring safety checks are in place to handle empty stack conditions gracefully. Further enhancing the stack's utility, the `stk_display` function offers a way to print the stack's contents, optionally in reverse order, providing a visual representation of its current state. The `stk_destroy` function ensures proper memory management by freeing the allocated memory when the stack is no longer needed. For more advanced operations, the `stk_copy` function creates a duplicate of the stack, and `stk_size` and `stk_capacity` functions provide insights into the stack's current usage and limits. Lastly, the `stk_clear` function resets the stack, allowing for reuse without reallocating memory.

Extension 09:

We converted the whole task 1, 2, 3 into LISP.

Task 1: We started by defining valid identifier names and declaring variables in Lisp. Lisp allows various characters, including symbols like hyphens and dollar signs, making it flexible for naming conventions. We demonstrated valid identifiers like `myVariable`, `MY_CONSTANT`, `_privateVar`, and `$specialVar`. Additionally, we introduced variable declarations using the `def` keyword, such as `(def x 5)`, `(def y 10.5)`, and `(def z "Hello, LISP!")`.

Task 2: Lisp supports block scoping using `let` expressions and conditional statements like `if`. We illustrated block scoping by defining variables within a block using `let` and printing their values. Similarly, we showcased function scoping by defining a function using `defun` and encapsulating

variables within it. Furthermore, we exemplified module scoping by encapsulating variables and functions within a module using `defmodule`. Lastly, we demonstrated global scoping by defining a global variable using `defparameter`.

Task 3: We explored basic built-in types such as integers, floating-point numbers, characters, strings, and boolean values in Lisp. Additionally, we introduced tuples, arrays, and dictionaries (association lists) as aggregate data structures. Lisp provides convenient ways to work with these data structures, making it suitable for various computational tasks.

Extension 10:

We converted the whole task 1, 2, 3 into VHDL.

Task 1: VHDL follows strict rules for naming identifiers and declaring variables. We adhered to these rules by defining valid identifier names such as signals, constants, and variables. Variable declarations were demonstrated using signals and variables with appropriate data types, including integer and floating-point types.

Task 2: VHDL uses concurrent and sequential statements for describing behavior and control structures in digital systems. We illustrated block scoping using concurrent signal assignments within processes and conditional statements like `if` for controlling the flow of execution. VHDL also supports modular design through entities and architectures, allowing hierarchical organization of designs.

Task 3: In VHDL, data types are crucial for representing signals and variables in digital circuits. We introduced basic data types such as integers, floating-point numbers, and characters, which are commonly used in VHDL designs. Additionally, we demonstrated aggregate data structures like arrays and records for organizing and representing complex data in VHDL designs.

Acknowledgements:

Professor Bender and Nafis