Mahdeen Ahmed Khan Sameer
Professor Max Bender
CS333
23 March 2024

Project 04: C Semantics!

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

Abstract:

In this project, we explore the semantics of C and JavaScript programming languages. We explore comparator functions, function pointers, and handling integer overflow in C. We also implement a toString method for our stack data structure and analyze its memory allocation. In JavaScript, we examine control flow statements, implement a generic merge sort algorithm, and demonstrate it with numbers and strings. The project extensions showcase dynamic function definitions, variable scope, aggregate data types, and similar concepts in Lisp and Go. We also optimize a comparator function for efficiency in C.

Results:

Part 01:

We defined a C program that sorts an array of integers such that even numbers come first, sorted in descending order, followed by odd numbers, sorted in ascending order. The program defines a comparator function required by the qsort function to determine the sorting order based on the specified criteria. Inside the comparator, we first check if both numbers are even using the modulo operator (%). If they are, we return the difference between the second number and the first (b - a) to sort even numbers in descending order. If both numbers are odd, we return the difference between the first number and the second (a - b) to sort odd numbers in ascending order. Finally, if one number is even and the other is odd, we return the result of (a % 2) - (b % 2), which ensures that even numbers appear before odd numbers. In the main function, I create an array of random integers and determine its size. We utilize the qsort function to sort the array using the comparator function.

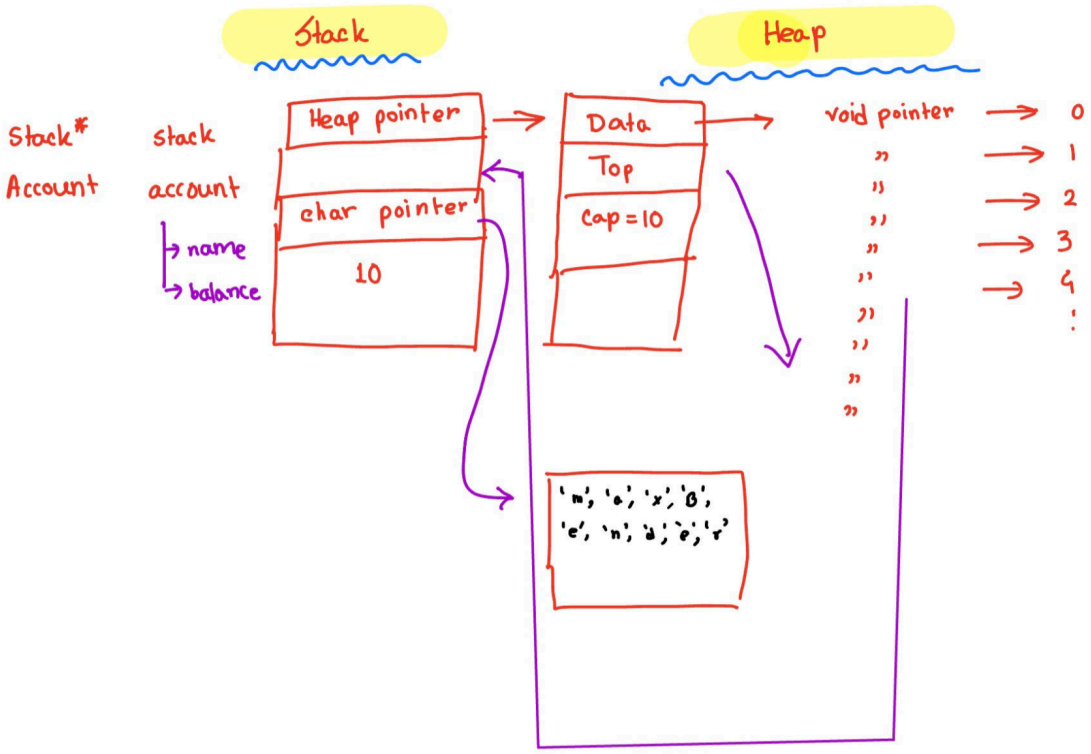
For task 2, we explore function pointers in C and how they allow functions to be treated as data types. We create a new file and define a function that calculates the factorial of an integer argument. In the main function, we declare a variable calc of type int (*)(const int), which is a function pointer that takes a constant integer argument and returns an integer. We then assign our factorial function to calc and use it to execute the function, printing out the return value. This showcases that functions in C can be assigned to variables and invoked through function pointers, highlighting the flexibility of treating

functions as data types. Next, we update the factorial function to accept an integer command-line argument N. We experiment with different values of N and observe the behavior. When calculating the factorial of numbers beyond 12, we encounter integer overflow. For example, 13! exceeds the range of a 32-bit integer (which is typically from -2,147,483,648 to 2,147,483,647) resulting in undefined behavior. This highlights the limitations of integer types in handling large factorial values.

In task 3, we update our stack data structure from the previous week to work with arbitrary data types. We are provided with an updated header file (cstk.h) and our goal is to implement the functions declared in the header file in a new cstk.c file. We created a stk_toString method to generate a string representation of the stack. It dynamically resizes the result string by doubling the capacity when necessary using realloc. This ensures efficient memory usage while maintaining linear time complexity. We also implement custom toString functions for different data types, such as intToString and accountToString. These functions take a void * parameter, cast it to the appropriate type, and convert the result to a string representation. For integers, we perform a straightforward conversion (e.g., 10 → "10"), while for custom types like Account, we format the string according to a specific pattern (e.g., "Max Bender: 10").

To test our stack implementation, we use the provided cstktest2.c file. We ensure that our toString method is memory-efficient and free of memory leaks. In the report, we include a diagram illustrating the memory contents at a specific point (mark 1) in the cstktest2.c file.

Mark 1:
The structure looks like this:



Part 02:

Here's a comparison of these control flow statements with their counterparts in the C programming language:

1. if statement:
    - In both JavaScript and C, the if statement is used to conditionally execute a block of code based on a boolean condition.
    - The syntax is similar in both languages.
2. if...else statement:
    - JavaScript and C both support the if...else statement to execute different blocks of code based on a condition.
    - The syntax is similar in both languages.
3. if...else if...else statement:
    - JavaScript and C both support the if...else if...else statement to test multiple conditions and execute different blocks of code accordingly.
    - The syntax is similar in both languages.
4. switch statement:

- JavaScript and C both have the switch statement for selecting one of many code blocks to be executed based on different cases.
- The syntax is similar in both languages, but JavaScript allows expressions as case labels, whereas C only allows constant expressions.

5. for loop:
    - JavaScript and C both have the for loop for iterating over a block of code a specified number of times.
    - The syntax is similar in both languages.
6. while loop:
    - JavaScript and C both have the while loop for iterating over a block of code as long as a specified condition is true.
    - The syntax is similar in both languages.
7. do...while loop:
    - JavaScript and C both have the do...while loop, which is similar to the while loop but executes the block of code at least once before checking the condition.
    - The syntax is similar in both languages.
8. continue statement:
    - JavaScript and C both have the continue statement, which is used to skip the current iteration of a loop and continue with the next iteration.
    - The syntax is similar in both languages.
9. try...catch...finally statement:
    - JavaScript has the try...catch...finally statement for exception handling, allowing you to catch and handle errors gracefully.
    - C does not have a built-in exception handling mechanism like try...catch...finally. Error handling in C is typically done using error codes and manual checks.

Additionally, we created a quicksort function in JavaScript that efficiently sorts arrays using the quicksort algorithm. The function is designed to work with any type of element, as long as a suitable comparator function is provided. The `quickSort` function takes two parameters: `arr`, which is the array to be sorted, and `comparator`, which is a function used to compare elements and determine their relative order.

The function starts by checking if the array has a length of 1 or less. If so, the array is already sorted, and it is returned as is. If the array has more than one element, the function selects a pivot element from the middle of the array. It then creates two empty arrays, `left` and `right`, to store elements smaller

than the pivot and elements larger than the pivot, respectively. The function iterates over each element in the array (excluding the pivot) and compares it with the pivot using the provided `comparator` function. If the `comparator` returns a negative value, indicating that the current element is smaller than the pivot, the element is added to the `left` array. If the `comparator` returns a positive value, indicating that the current element is larger than the pivot, the element is added to the `right` array. The `quickSort` function is then recursively called on the `left` and `right` arrays, using the same `comparator` function. This process continues until all subarrays are sorted. Finally, the sorted `left` array, the pivot element, and the sorted `right` array are concatenated to obtain the final sorted array.

To demonstrate the usage of the `quickSort` function, we provided two example comparator functions: `numberComparator` for sorting numbers and `stringComparator` for sorting strings. We tested the `quickSort` function with an array of numbers `[5, 3, 9, 1, 7]` using the `numberComparator`. The original array and the sorted array were logged to the console. We also tested the `quickSort` function with an array of strings `["banana", "apple", "orange", "grape"]` using the `stringComparator`. Again, the original array and the sorted array were logged to the console.

The quicksort algorithm implemented in our JavaScript function is efficient, with an average time complexity of O(n log n). It provides a flexible and reusable way to sort arrays of various types by allowing customization through the comparator function.

Please check Google Sites for Task 2.

Extensions:

Extension 01:

In this extension, we made a quicksort implementation in Common Lisp that efficiently sorts lists using the quicksort algorithm. The `quick-sort` function takes a list and a comparator function as parameters, recursively divides the list into sublists based on a pivot element, and sorts them independently. We provided example comparator functions for sorting numbers and strings and demonstrated the usage of the `quick-sort` function with both types of data. The code showcases the concise and expressive nature of Common Lisp, utilizing its functional programming features and built-in functions to implement the quicksort algorithm efficiently.

Extension 02:

In our second extension, we provided examples of dynamic function definition and function creation in JavaScript. In the first example, we defined a function `greet` that takes a `name` parameter

and logs a greeting message to the console. We then demonstrated how a function name can be stored in a variable and invoked dynamically using square bracket notation, allowing for flexible and dynamic function invocation. In the second example, we showcased the ability to dynamically create functions using a higher-order function `createGreetingFunction`. This function takes a `greeting` parameter and returns a new function that accepts a `name` parameter. The returned function logs a personalized greeting message to the console. By assigning the result of `createGreetingFunction` to a variable `customGreet`, we created a specialized greeting function that can be invoked with different names. These examples highlight the dynamic nature of JavaScript and its support for treating functions as first-class citizens, enabling powerful and flexible programming techniques.

Extension 03:

We provided examples of arrays, objects, and nested objects in JavaScript. In the first example, we created an array `numbers` containing five numeric elements and logged it to the console. Arrays in JavaScript are ordered collections of elements that can be of any data type. In the second example, we created an object `person` with properties `name`, `age`, and `city`, representing a person's details. Objects in JavaScript are unordered collections of key-value pairs, where the keys are strings and the values can be of any data type. We logged the `person` object to the console. In the third example, we demonstrated nested objects by creating an `employee` object that contains a nested object `department` and an array `skills`. The `department` object has its own properties `name` and `location`, representing the department details of the employee. The `skills` array contains a list of the employee's skills. We logged the `employee` object to the console. These examples showcase the flexibility and versatility of JavaScript in working with different data structures, allowing for the organization and representation of complex data in a structured manner.

Extension 04:

In this extension, we created a function called `display` that takes a single parameter `data` and demonstrates how to handle different data types in JavaScript. The function uses conditional statements to determine the type of the input `data` and performs different actions based on the data type. If the input is a string, the function logs a message to the console indicating that a string was given and includes the string value. If the input is a number, the function logs a message stating that a number was given along with the numeric value. If the input is an array, the function logs a message indicating that an array was given and uses `JSON.stringify` to convert the array to a string representation for display purposes. If the input is any other data type, such as an object, the function logs a message specifying the data type of the input using the `typeof` operator. We then demonstrated the usage of the `display` function by calling it

with different arguments: a string, a number, an array, and an object. The corresponding output showcases how the function handles each data type and provides appropriate messages in the console. This example highlights the importance of type checking and conditional handling in JavaScript to ensure proper treatment of different data types within a function.

Extension 05:

In this extension, we compared the performance of two different comparator functions used with the qsort function from the standard library. The purpose was to demonstrate how the efficiency of the comparator function can impact the overall performance of the sorting algorithm. We defined two comparator functions: efficientComparator and basicComparator. Both functions sort the array in a specific order: even numbers come before odd numbers, and within each group (even or odd), the numbers are sorted in ascending order.

The efficientComparator function optimizes the comparison logic by first checking if both numbers have the same parity (both even or both odd) using the modulo operator. If they have the same parity, it directly compares the numbers using subtraction. If they have different parities, it returns -1 if the first number is even and 1 if the first number is odd, ensuring that even numbers come before odd numbers. On the other hand, the basicComparator function uses a series of conditional statements to determine the order of the numbers based on their parity and value. It checks if one number is even and the other is odd, and returns -1 or 1 accordingly. If both numbers have the same parity, it compares them using subtraction. In the main function, we created two arrays (ary1 and ary2) of size 1,000,000 and filled them with the same set of random numbers. We then used the clock function to measure the execution time of sorting each array using qsort with the respective comparator functions.
After sorting, we calculated the time taken for each sorting operation and printed the results. The output shows that the efficient comparator function (efficientComparator) took slightly less time to sort the array compared to the basic comparator function (basicComparator).

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 in the form of a Haiku.

Extension 07:

In this JavaScript code, we demonstrated the concept of lexical scoping and closure by defining variables and functions at different scope levels. We have a global variable `x`, an outer function `outer()` with a local variable `y`, and an inner function `inner()` with its own local variable `z`. The `inner()`

function accesses variables from all three scopes and logs their sum to the console, showcasing closure. The `outer()` function invokes `inner()` and logs the sum of `x` and `y`, demonstrating its access to both global and local variables. Finally, we invoke `outer()` and log the value of `x` in the global scope. The output of this code would be 60, 30, and 10, respectively, illustrating how JavaScript functions can access variables from their outer lexical scopes and retain access to those variables even after the outer functions have finished executing.

Extension 08:

In this Lisp code, we explored the concept of treating functions as first-class citizens. We demonstrated assigning functions to variables, passing functions as arguments to other functions, and defining and calling functions directly as arguments. First, we assigned a lambda function to the variable `greet`, which takes a `name` parameter and prints a greeting message using `format`. We then called the function assigned to `greet` using `funcall`, passing the argument "John". Next, we defined a function `say-something` that takes a function `func` and a `name` as parameters and calls `func` with `name` using `funcall`. We passed the `greet` function to `say-something` along with the argument "Alice". Finally, we demonstrated defining and calling a lambda function directly as an argument to `say-something`, which prints a different greeting message with the name "Bob". This code showcases the flexibility and power of treating functions as first-class objects in Lisp, enabling functional programming techniques and higher-order functions.

Extension 09:

In this extension, we made a general sorting algorithm in Go. In this Go code, we implemented the quicksort algorithm using function types and interface{}. We defined a type Comparator as a function that takes two interface{} parameters and returns a bool, allowing for custom comparison logic. The QuickSort function takes a slice of interface{} and a Comparator function as parameters. It recursively sorts the slice using the quicksort algorithm, utilizing the provided comparator function to compare elements. The algorithm selects a pivot element, partitions the slice into left and right sub-slices based on the pivot, recursively sorts the sub-slices, and combines them to obtain the final sorted slice. We provided two example comparator functions: intComparator for comparing integers and stringComparator for comparing strings. In the main function, we demonstrated the usage of QuickSort with both integers and strings. We created slices of interface{} containing the elements to be sorted, printed the original slices, called QuickSort with the appropriate comparator function, and printed the sorted slices. This code showcases Go's support for function types, interface{}, and the ability to write generic code that can work with different data types.

Extension 10:

In this Go code, we explored the concept of treating functions as first-class citizens. We defined a function type `greeterFunc` that represents a function taking a string parameter and returning a string. We then demonstrated various ways to work with functions in Go. First, we defined two functions, `sayHello` and `sayHi`, that take a name as a parameter and return a greeting string. We also defined a function `greet` that takes a `greeterFunc` and a name as parameters and calls the `greeterFunc` with the given name, printing the result. In the `main` function, we assigned the `sayHello` function to a variable `greetFunc` of type `greeterFunc` and called it with the argument "John". We then passed the `sayHi` function directly to the `greet` function along with the argument "Alice". Finally, we demonstrated defining and calling an anonymous function directly as an argument to `greet`, which returned a greeting in Spanish for the name "Bob". This code showcases Go's support for treating functions as first-class objects, enabling functional programming techniques such as assigning functions to variables, passing functions as arguments, and defining anonymous functions on the fly.

Acknowledgements:

Professor Bender and Nafis