Mahdeen Ahmed Khan Sameer
Professor Max Bender
CS333
10 April 2024

Project 05

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

Abstract:

In our project, we explored polymorphism and generic data structures across multiple programming languages including C, JavaScript, and Scala. I found it fascinating to explore how these concepts apply differently in each language. Starting with C, I worked on creating a generic linked list capable of handling any data type. By using function pointers, I enabled specialized data handling and comparison operations. This was particularly crucial for operations like squaring values and managing dynamic memory allocations for complex data types, such as strings. The focus was always on maintaining versatility and ensuring an error-free environment. In JavaScript, I implemented polymorphism through a dynamic linked list and various shape objects. This allowed me to demonstrate how methods like push, pop, and map can be universally applied. I also paid special attention to error handling and preventing memory leaks to boost robustness and reliability during runtime operations. Moving over to Scala, my approach included using traits and class-based implementations for stacks. This highlighted Scala's functional programming capabilities intertwined with object-oriented paradigms. It was rewarding to provide efficient and encapsulated data handling through list-based stack operations. Each language's implementation was carefully designed to optimize for the specific traits of the environment it operates in—such as meticulous memory management in C and the functional flexibility in Scala. This not only served as a practical utility toolkit for data management but also enriched my understanding of polymorphic and generic programming.

Results:

Part 1 – Polymorphism in C:

In this task, I created a generic linked list class in C, adhering to the provided requirements and function specifications. I started by designing a header file that comprises the Node struct, LinkedList struct, and all the necessary function prototypes. Following that, I developed a C file to implement all these functions. My implementation made the linked list capable of storing any data type. I utilized function pointers to customize operations such as data comparison or specific data manipulations, which

are essential for handling different data types effectively. This approach not only tailored the linked list to be versatile but also ensured that its memory management was robust, preventing any memory leaks. After extending the linked list to accommodate a second data type, the final results showcased the flexibility and effectiveness of this implementation in managing multiple data types dynamically. This modification further enhanced the functionality and applicability of the linked list in various data handling scenarios.

Results

After initialization

value: 18
value: 16
value: 14
value: 12
value: 10
value: 8
value: 6
value: 4
value: 2
value: 0

After squaring

value: 324
value: 256
value: 196
value: 144
value: 100
value: 64
value: 36
value: 16
value: 4
value: 0

removed: 16
No instance of 11
No instance of 11

After removals

value: 324
value: 256

value: 196
value: 144
value: 100
value: 64
value: 36
value: 4
value: 0

After append
value: 324
value: 256
value: 196
value: 144
value: 100
value: 64
value: 36
value: 4
value: 0
value: 11

After clear

After appending
value: 0
value: 1
value: 2
value: 3
value: 4

popped: 0
popped: 1

After popping
value: 2
value: 3
value: 4

List size: 3

String list after initialization
value: CS333 Project 5
value: World
value: Hello

Found: World
Removed: World

String list after removal
value: CS333 Project 5
value: Hello

Additionally, the “freefunc” parameter is crucial for the “ll_clear” function because it allows the user to define how to properly free the memory associated with the data stored in each node of the linked list. This aspect is vital given that the linked list is generic and designed to store arbitrary data types, which may require specific deallocation strategies. For instance, consider a linked list where each node contains a pointer to a dynamically allocated string (allocated using `malloc`). The appropriate method to free the memory for each string is to use the “free” function. If the `freefunc` provided by the caller is incorrect, or if “ll_clear” does not utilize the supplied “freefunc”, the memory allocated for the strings would not be adequately freed, leading to memory leaks. This functionality underscores the need for careful memory management in generic data structures, particularly when dealing with various data types that necessitate distinct approaches for memory allocation and deallocation.

Part 2 – Polymorphism in JS:

I implemented a linked list in JavaScript by creating two classes: “Node” and “LinkedList”. The “Node” class forms the building blocks of our list, with each node containing data and a reference to the next node. The “LinkedList” class manages the list with methods for various operations: “push” to add elements at the front, “pop” to remove elements from the front, “append” to add to the end, `remove` to delete a specific element, “find” to locate an element, and “clear” to empty the list entirely. This structure ensures efficient element access and manipulation, leveraging dynamic array characteristics similar to Python’s built-in list, where index-based access is fast (O(1)), but element searches and resizing may take linear time (O(N)).

Results:

After initialization

value: 18

value: 16

value: 14

value: 12

value: 10

value: 8

value: 6

value: 4

value: 2

value: 0

After squaring

value: 18

value: 16

value: 14

value: 12

value: 10

value: 8

value: 6

value: 4

value: 2

value: 0

removed: 16

No instance of 11

No instance of 11

After removals

value: 18

value: 14

value: 12

value: 10

value: 8

value: 6

value: 4

value: 2

value: 0

After append

value: 18

value: 14

value: 12

value: 10

value: 8

value: 6

value: 4

value: 2

value: 0

value: 11

After clear

After appending

value: 0

value: 1

value: 2

value: 3

value: 4

popped: 0

popped: 1

After popping

value: 2

value: 3

value: 4

List size: 3

String list after initialization

value: CS333 Project 5 in JavaScript now!

value: World

value: Hello

Found: World

Removed: World

String list after removal

value: CS333 Project 5 in JavaScript now!

value: Hello

Task 2: (on Google Sites)

Extensions

Extension 1:

In this extension, I implemented a "LinkedList" class in Scala, which involved defining a "Node" class as the fundamental component of our linked list. The "Node" class is generic, parameterized by type "T", and each node holds a data element of type "T" along with an optional reference to the next node, also typed as "Node[T]". This setup allows for the linked list to be versatile and adaptable to various data types. The "LinkedList" class itself is similarly generic, making it capable of handling any type "T". This design not only facilitates type safety and flexibility but also aligns with Scala's robust type-inference system, making the list particularly powerful for a wide range of programming needs.

Extension 2:

In this extension, I explored the concept of polymorphism in JavaScript through a practical example involving geometric shapes. I defined a base class called "Shape" along with two derived classes: "Circle" and "Square". The "Shape" class has a constructor that accepts a "name" parameter and includes an "area()" method, which returns a default message indicating that the area is not computed. Both "Circle" and "Square" classes inherit from "Shape" and override the "area()" method to provide specific implementations: using "radius" for "Circle" and "side" for "Square" to calculate their respective areas. The demonstration of polymorphism is further emphasized through a function called "haiku", which accepts a "shape" object and prints a haiku-style message alongside the shape's area. This function illustrates polymorphism by handling objects of "Circle" and "Square" as instances of their common base class, "Shape". This approach showcases how JavaScript can effectively use object-oriented principles to treat different objects through a uniform interface, highlighting the flexibility and dynamic capabilities of polymorphism in managing diverse object types with common methods.

Extension 3:

In this extension, I focused on refining the linked list implementation to ensure it executes without memory leaks and to enhance its testing functionality. I incorporated comprehensive memory allocation error handling to prevent crashes and leaks due to improper

memory management. To facilitate more robust testing, I separated the test functions for integers and strings, which allowed me to validate the functionality specifically tailored to these data types. The testing framework now comprehensively demonstrates the capability of the linked list to handle various operations. These operations include appending elements to the end, pushing elements to the front, removing specific elements, finding elements by their value, and clearing the entire list. This extended testing is conducted separately for both integer and string data types to ensure the linked list accurately manages memory allocation and deallocation, which is crucial for maintaining the integrity and performance of data structures in dynamic programming environments. These improvements not only enhance the utility and reliability of the linked list but also ensure it operates efficiently under different data handling scenarios.

Extension 4:

In this extension, I explored two distinct approaches to implement a generic stack data structure in JavaScript, catering to different programming needs and styles. The first method is a class-based implementation where I defined a "Stack" class complete with methods such as "push", "pop", "isEmpty", and "size". This class internally uses an array to manage the stack's elements, providing a clear and traditional object-oriented way to manipulate the data structure. The second method is a closure-based implementation, which leverages JavaScript's functional capabilities. I created a "createStack" function that returns an object with methods including "push", "pop", "peek", "isEmpty", and "size". Here, the stack's items are kept in a private array within the closure. This encapsulation prevents external access to the stack's elements, enhancing security and data integrity. Both implementations are generic, capable of handling elements of any type. The class-based approach offers structure and reusability, making it easy to create multiple instances of the stack with a clear, template-like format. In contrast, the closure-based approach provides better encapsulation and privacy, securing the data within and preventing unintended interactions from outside the stack's scope. The choice between these two depends largely on the specific needs for encapsulation and structural organization in the project, as well as the developer's preference for programming paradigms—object-oriented vs. functional programming.

Extension 5 & 6: Generic Stack

In these extensions, I implemented two different approaches to creating a generic stack data structure. The first approach utilizes a trait-based implementation, where the Stack trait defines the contract for the stack operations, including push, pop, peek, isEmpty, and size. The trait is parameterized with a type parameter T, allowing it to handle elements of any type. The StackImpl class extends the Stack trait and provides the actual implementation of the stack operations. Internally, it uses a List to store the stack elements, with the head of the list representing the top of the stack. The second approach directly defines a Stack class parameterized with a type parameter T. The class encapsulates the stack operations and the internal storage of elements using a List. The push operation prepends an item to the list, while the pop operation removes and returns the head of the list. The peek operation returns the head of the list without removing it, and the isEmpty and size operations provide information about the stack's state.

Extension 7:

In this extension, I introduced a new function called "ll_delete" to the generic linked list implementation, which allows for the removal of nodes at specific positions using 0-indexing (where the head node is position 0). The function robustly handles invalid inputs by doing nothing if the position is out of bounds (less than 0 or greater than or equal to the size of the list). It efficiently manages special cases, such as removing the head node or the only node in the list, and ensures no memory leaks by properly deallocating memory for the removed node after extraction. This addition not only enhances the functionality of the linked list, allowing for precise structural modifications, but also maintains the system’s integrity and performance over time.

Collaborators & Acknowledgements:

Professor Bendor & TA Nafis