Temitope Emokpae 5/12/2023

CSC 344

Csc344 Problem Set: Memory Management / Perspectives on Rust

Problem 1 - The Runtime Stack and the Heap

When managing memory for programming languages, two important data structures are the runtime stack and the heap. Although they are both involved in memory storage, they have significant differences in size and types of data stored that affect their roles in memory management. The following paragraphs will explain the functions of the runtime stack and the heap, highlighting their fundamental differences. Understanding these differences provides a foundation for improving memory management efficiency in multiple programming languages.

The runtime stack is a data structure mainly used for temporary storage. The call stack, a part of the runtime stack, handles memory allocation for variables associated with function calls. Once the execution of a function is complete, its variables get removed from the stack. Therefore, the stack plays a crucial role in determining the scope of variables in a program. Any variables that are within range can get accessed via the runtime stack. Another important aspect of this data structure is its limited space. Since the stack is smaller than the heap, it typically stores static data and utilizes pointers to reference more enormous data located in the heap.

The heap is a data structure that helps manage memory. It stores larger, dynamic data that the stack cannot handle. The stack keeps track of pointers to the heap. Proper memory allocation is crucial when using the heap to avoid memory leaks. Unlike the runtime stack, the heap has no built-in process for freeing up memory. The programmer is responsible for managing memory allocation in the heap.

Problem 2 - Explicit Memory Allocation / Deallocation vs Garbage Collection.

Understanding memory management in programming languages involves comprehending explicit memory allocation/deallocation and garbage collection. These concepts were considered competing entities, with programmers weighing the pros and cons of each approach. However, with the introduction of Rust, both

explicit memory allocation and garbage collection can coexist in a single language thanks to an optimized runtime stack and heap. The following paragraphs will dive into the inner workings of memory allocation/deallocation and garbage collection, highlighting their similarities and differences.

Explicit memory allocation/deallocation is typical in lower-level programming languages where memory management is essential. This method allows programmers to control the allocation of memory, which is beneficial in strict contexts. However, this approach is prone to human error, such as bugs like double-free, dangling pointers, and memory leaks. Double-free errors occur when two different code sections free the exact memory location. Dangling pointers happen when pointers do not point to anything. Memory leaks occur when the programmer forgets to reallocate and deallocate memory space. C and C++ are two programming languages that rely on explicit memory allocation and deallocation.

Garbage collection serves as an alternative to explicit memory allocation and deallocation. In this approach, the programming language manages memory during program runtime instead of the programmer. Garbage collection automatically classifies data usage and frees up memory accordingly. Unused data gets handled by garbage collection, which helps prevent bugs with explicit memory management. However, it is worth noting that garbage collection can take longer to run. Popular programming languages like Java and Python have built-in garbage collection features.

Task 3 - Rust: Memory Management

1: Rust's memory model differs from that of Haskell. Rust, like C++, is said to provide better control over memory utilization. In C++, we actively allocate heap memory with new and release it with delete.
2: In Rust, we allocate and deallocate memory at specific points in our program. Meaning that it doesn't have garbage collection like Haskell does.
3: In Rust, memory models such as heap memory are designed to have a single owner. Once that owner leaves the scope, the memory gets automatically deallocated.
4: In Rust, variables get declared within a specific scope, like a for-loop or function definition. Once that code block is complete, the variable is no longer accessible to us.
5: Rust lets you copy primitive types. However, modifying the copy will not affect the original and vice versa. It is recommendable to use the clone function instead.
6: In Rust, you cannot use string literals. Instead, using the string type for strings would be best. You can

use the methods provided within the string type to modify strings in Rust.

7: Programmers often underestimate the cost of deep copies. Therefore, Rust, a performance-focused language, avoids deep copying by default.

8: Like C++, Rust also can pass a variable by reference using the (&) operator. It enables another function to borrow ownership temporarily instead of taking it permanently.

9: It's important to note that you can only have one mutable reference to a variable at a time. If you have more than one, your code won't compile. This rule is in place to prevent so many bugs.

10: There are two types of slices: primitive data stored on the stack and references to other objects. Slices do not have ownership, so they do not deallocate memory when they go out of scope.

Task 4 - Paper Review: Secure PL Adoption and Rust

The library of programming languages is constantly expanding with new additions at an increasingly rapid rate. While languages like Python, Java, C, and C++ are currently the most widely used languages by different technology companies across various products, new languages like Rust are building on the concepts of these top languages and may one day reach the top.

Rust was published in 2014, significantly impacting the programming world. Unlike languages such as Java and C which have been around for decades, Rust offers solutions to memory faults commonly found in C and C++ by taking memory management out of the hands of the programmer. Additionally, Rust is more secure in terms of memory management than Python, which can be prone to leaks due to its loose treatment of variables.

Many companies are now adopting Rust as their preferred programming language due to its effortless ability to produce bug-free code efficiently. Although it has the drawbacks of having a slightly steeper learning curve and longer compile time than languages like C, Rust's high-quality error messages and faster program design and implementation make it easier to maintain. It's a language with a bright future, and it's worth it for a person to add to their skill set for career advancement.