Csc344 Problem Set: Memory Management / Perspectives on Rust

Task 1: The Runtime Stack and the Heap

The runtime stack and the heap are essential to Rust programming because they can effectively and flexibly manage memory allocation and deallocation. Values with a defined size at compile time and a lifetime determined by the function call stack are stored in the runtime stack. On the other hand, values whose size and lifetime cannot be known at build time are stored in the heap.

A data structure called the runtime stack is used to keep track of function calls and the variables they use. The values of these variables are kept in the frame during function execution, and any additional function calls add new frames to the stack. When a function completes, its frame is removed from the stack, and control is given back to the function that called it. The runtime stack is a fundamental idea in computer science and is essential for controlling the flow of control and data in a program.

The heap is a portion of memory used in Rust programming that enables persistent value storage across function calls. Memory is allocated for items whose size cannot be known at compile time or whose lifespan spans outside of a certain function or block using the heap.
The memory allocator, which is in charge of allocating and dealinglocating memory on the heap in Rust, is in charge of managing the heap.

Task 2: Explicit Memory Allocation / Deallocation vs Garbage Collection

Programmers require a firm grasp of memory management strategies to write high-quality code that performs effectively and is free from memory-related mistakes. Effective memory management is essential for creating efficient and dependable software programs. Among many memory management techniques, I'm going to briefly cover two of the well-known techniques; manual memory management, and garbage collection.

Memory requests and releases that are made directly from the computer's memory system using explicit instructions found in the program code are referred to as explicit allocation and deallocation of memory. This is manual memory management. One of the languages that use this explicit allocation and deallocation of memory is Rust. Although its ownership and borrowing mechanism does offer automatic memory management, it also enables explicit allocation and deallocation. The other one is C. When memory on the heap needs to be explicitly allocated in C, the malloc() function is used to explicitly allocate memory on the heap, and the free() function is used to release memory when it is no longer required.

Many programming languages employ garbage collection, an automatic memory management approach, to handle memory allocation and deallocation without necessitating explicit allocation and deallocation by the programmer. The specifics of memory allocation and deallocation are abstracted away from programmers by the use of automatic memory management techniques like garbage collection in several contemporary programming languages, including Java, Python, and C#.

Task 3: Rust: Memory Management

1. In Rust, we do allocate memory and de-allocate memory at specific points in our program.
2. Thus it doesn't have garbage collection, as Haskell does.
3. This(ownership) is the main concept governing Rust's memory model.
4. Heap memory always has **one owner**, and once that owner goes out of scope, the memory gets de-allocated.
5. We declare variables within a certain scope, like a for-loop or a function definition. When that block of code ends, the variable is **out of scope**. We can no longer access it.
6. Rust works the same way. When we declare a variable within a block, we cannot access it after the block ends. (In a language like Python, this is actually not the case!)
7. Another important thing to understand about primitive types is that we can copy them.
8. Since they have a fixed size and live on the stack, copying should be inexpensive.
9. This(String type) is a non-primitive object type that will allocate memory on the heap.
10. In general, **passing variables to a function gives up ownership**.

Task 4: Paper Review: Secure PL Adoption and Rust

The paper "Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study" is a research paper that explores the benefits and drawbacks of using the programming language Rust for developing secure software. Rust is a systems programming language that is known for its emphasis on safety and security. The paper likely evaluates Rust's security features and how they compare to other programming languages, as well as discusses the practical considerations of adopting Rust for software development.

One of the contents that I find most interesting about the paper is Core features and ecosystem. Rust is a multi-paradigm language that supports a range of programming language styles and can be sued for a variety of purposes. The usage of traits, which are akin to Java's interfaces or Haskell's type classes in that they enable developers to specify behavior that types can share, is one of Rust's fundamental characteristics. Any type can use traits, and structures and traits can both be used to encode objects.

Additionally, Rust provides generics and modules, which offer a mechanism to create flexible, reusable code. The macro system in Rust enables programmers to create code that creates additional code, which can be helpful for streamlining time-consuming or challenging operations.