CSC 344 – Memory Management/Perspectives on Rust by Aaradhya Acharya

Task 1 : The Memory Stack and the Heap

The concepts of runtime stack and heap are both very important ones that every budding programmer needs to know about. Both the runtime stack and heap are methods of memory management used by various programming languages.

The first concept is that of a runtime stack. When a program is executed, a stack is made to keep a track of everything that happens. A Runtime Stack deals with the temporary allocation of memory within a system. When a program is run, the compiler allocates some memory within the system for that program to use. Everything defined on a stack can only be defined by the following stack and stacks cannot be resized hence the data must have a definite length. There are three ways stacks can be implemented: arrays, dynamic memory, and linked lists. Stacks are small and do not take up a lot of data, which makes them fast and ideal for smaller programs.

The second concept is that of a heap. A heap is essentially a less organized form of a runtime stack. When something is put on a heap, a certain amount of memory is allocated for it and a pointer gets returned, which "points" to the location the associated data is stored in. Heaps can store a lot more data than stacks, but this also makes them run a lot slower than a stack. Heaps also do not deallocate their own memory, so it is very important to keep in mind when a heap is being designed in order to avoid situations of memory leak, dangling pointers, or having situations of double free bugs.

Task 2 : Explicit Memory Allocation/Deallocation vs. Garbage Collection

Memory is finite, there's only so much of it that can go around even in the beefiest of computer systems. Memory management is using the finite resource of memory as efficiently as possible. There are two main methods of memory management: (i)explicitly allocating and deallocating memory, and (ii)garbage collection. Both of these methods of memory management will be discussed in the next two paragraphs.

Languages like C/C++ are considered to be low-level of languages which require explicit memory allocation and deallocation. It is entirely up to the programmer on how the memory of the system is managed. The programmer would need to delete the memory manually as it is not automatically freed by the compiler with deallocation and the opposite would need to be done for allocation. This gives the programmer more control, but if not careful, memory can be eaten up and issues such as memory leaks can occur.

Garbage collection is a memory recovery feature generally present in higher level languages such as Java and Python. Garbage collectors automatically free up memory space allocated to objects that are no longer needed. First, the compiler allocates memory for the program to use, then once the program is finished the garbage collector comes in and deallocates the memory used by the

program automatically. It is, however, slower than explicitly doing so, as it takes time for the garbage collector to process and do its job. This is why it's not typically used in lower-level languages.

Task 3: Rust – Memory Management

- In Rust, we do allocate memory and de-allocate memory at specific points in our program. Thus it doesn't have garbage collection, as Haskell does.

- Heap memory always has one owner, and once that owner goes out of scope, the memory gets de-allocated.

- We don't need to call delete as we would in C++. We define memory cleanup for an object by declaring the drop function.

- In Rust, you can copy primitive types. But if you change the copy, the original will not change and vise versa. It's also just better to use the clone function.

- 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.

- Like in C++, we can pass a variable by reference. We use the ampersand operator (&) for this. It allows another function to "borrow" ownership, rather than "taking" ownership. When it's done, the original reference will still be valid.

- Rust uses the idea of ownership in memory. This means that memory can only have one owner.

- Rust is a performance-oriented language.

Task 4: Paper Review – Secure PL Adoption and Rust

Rust is an open-source systems programming language created by Mozilla, with its first stable release in 2014. It is a multi-paradigm language, with elements drawn from functional, imperative, and object-oriented languages. Rust does not use garbage collection, but the way ownership works in it is effective.

Rust was developed to combat memory and safety related vulnerabilities in a simple yet effective manner compared to the complex new languages. Based on the information gathered from surveys, companies that have implemented Rust as or senior software developers that had worked it reported mostly positive things. A key benefit mentioned by participants is that once Rust code compiles, developers can be fairly confident that the code is safe and correct. Also, while the initial time to design and develop a solution in Rust is sometimes long and/or hard to estimate due to unforeseen conflicts with the borrow checker, interview participants felt — and survey participants

agreed or strongly agreed — that Rust reduced development time overall, from the start of a project to shipping it, compared to other languages they were comfortable with. Most participants also reported that Rust has had at least a minor positive effect on their development in another language they're comfortable with.

However, Rust isn't all just sunshine and rainbows. The difficulty of learning Rust was among the biggest concerns participants encountered at their companies Many participants cited unfamiliarity with Rust as one reason people were worried about adopting or did not adopt Rust at their company. Any change to an unfamiliar language could create uncertainty or apprehension. Since Rust is relatively new, some participants cited company concerns about the maturity and maintenance of its tooling and ecosystem, as well as whether it would be around long-term. Like with most things, Rusts has its own pros and cons but there is a bright future ahead for Rust as a programming language and it wouldn't be too bad of an idea for new programmers to maybe look into it at some point down the road in their career path.