CSC344 Problem Set: Memory Management / Perspectives on Rust

written by **David Hennigan**

Task 1 – The Runtime Stack and the Heap

The runtime stack and heap memory space are two very important concepts in computer science. The runtime stack and heap allow for the operation of many programming languages we use today. Understanding the runtime stack and heap will allow you to write more efficient code, utilize recursion, and avoid memory issues(if using a language like c). The runtime stack and heap are outlined in the following paragraphs.

A stack is a data structure that is typically used for program memory management aka the runtime stack(or call stack). This stack houses all of the functions called in the program and creates different levels of scope where local variables exist. The most recent function call will be at the top of the stack until it returns and pops off the stack, this stack system is a last in first out system. A notable quality of the runtime stack is its lack of space for data, typically only static data can be stored on the stack. The issue of dynamic data types arises with the stack but it is overcome with the use of pointers. Pointers allow the stack to reference dynamic data types that are stored in the larger heap space, that is also designed for dynamic data types and global variables.

As mentioned previously, the heap is where dynamic memory is stored. In most languages global variables are stored by default in the heap space. The heap space is not automatically managed for you (unless the language has garbage collection), memory must be allocated and deallocated in this space. The access time of the heap tends to be slower than that of the runtime stack, but a huge advantage of the heap is it allows for the resizing of variables.

Resources read before the creation of the previous paragraphs include:
https://rabingaire.medium.com/memory-management-rust-cf65c8465570
https://www.guru99.com/stack-vs-heap.html#:~:text=The%20heap%20is%20a%20memory,tightly%20managed%20by%20the%20CPU

Task 2 – Explicit Memory Allocation/Deallocation vs Garbage Collection

Memory management plays a vital role in the production of a working program. It is extremely important the developer knows how their tool(programming language) handles memory management whether it is manual or garbage collected. Memory management when handled poorly can cause a program to run significantly slower or in worse cases crash. Understanding memory allocation, deallocation, and garbage collection will allow developers to fully utilize their tool to accomplish any task(or to decide on which tool to use based on memory management).

The allocation and deallocation of memory is a fundamental concept in programming, all languages may not require knowledge of explicit allocation and deallocation of memory but as a programmer matures these concepts become far more important especially when more control is desired. The explicit allocation of memory is where a partition of the heap is used to house some information, typically house-keeping(meta) data is stored like the size of the partition and a boolean flag that is used to determine if the partition is free or in use. When it comes to memory deallocation, the memory is not returned to the system, only the house-keeping data is altered to reflect its new state of being free for use. Two programming languages come to mind when discussing memory allocation and deallocation, C and C++ as they both give the user direct control over the heap space.

Garbage collection is the handling of memory allocation and deallocation by the programming language and not the user. A system collects garbage by deallocating all of the heap space partitions that are no longer in use. Garbage collection is incredibly useful when it comes to development speed of a product, all of the errors that come with manual memory management are no longer a sweat but it comes at the cost of efficiency. Garbage collection is notorious for causing programs to “freeze up”. Two well-known programming languages that utilize garbage collection are Java and Python; Both languages actually use very different forms of garbage collection. Python uses a form of garbage collection that relies on reference counting while Java’s default garbage collection uses a mark and sweep system that can run on multiple cores.

Resources read before the creation of the previous paragraphs include:
https://towardsdatascience.com/c-memory-allocation-deallocation-for-data-processing-1b204fb8a9c
https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

Task 3 – Rust: Memory Management

All of the following salient sentences are taken from the article https://mmhaskell.com/rust/memory:

1. “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. Rust works the same way.”
2. “Another important thing to understand about primitive types is that we can copy them. Since they have a fixed size, and live on the stack, copying should be inexpensive.”
3. “What's cool is that once our string does go out of scope, Rust handles cleaning up the heap memory for it!”
4. “Deep copies are often much more expensive than the programmer intends. So a performance-oriented language like Rust avoids using deep copying by default.”
5. “Memory can only have one owner.”
6. “In general, passing variables to a function gives up ownership.”
7. “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.”
8. “You can only have a single mutable reference to a variable at a time! Otherwise your code won't compile!”
9. “As a final note, if you want to do a true deep copy of an object, you should use the clone function.”
10. “Slices give us an immutable, fixed-size reference to a continuous part of an array. Often, we can use the string literal type str as a slice of an object String. Slices are either primitive data, stored on the stack, or they refer to another object.”

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, contains many key points that computer scientists caring about the future of software development should take note of, two points will be discussed. The first being that Rust is a noteworthy programming language that developers tend to enjoy using. The second key point being product development time decreases when Rust is utilized.

Rust is a noteworthy language because it tackles memory management in a unique way and offers the efficiency of low-level programming. Rust’s ownership model makes it a popular candidate for databases, low-level systems, virtual machine management, and monitoring resource usage. Developers should probably avoid Rust if they are developing mobile apps, web apps, or GUIs due to lack of flexibility which is typically required with these applications.

Rust is not only a popular candidate for systems management and low-level applications because of its secure memory management but also because it easier to implement bug free code than in other low-level languages. Programmers who develop products in Rust tend to be more confident their code is bug free than when developing with other languages. Since Rust code tends to be more likely to contain less bugs, development tends to be faster. Due to the reasons previously described, Rust tends to be a great choice for a programming language if you are interested in lower level programs.

Paper reviewed: https://obj.umiacs.umd.edu/securitypapers/Rust_as_a_Case_Study.pdf