Problem Set: Memory Management / Perspectives on Rust

Brandon LaPointe

Learning Abstract

This assignment's first task is a writing piece which focuses on the runtime stack and the heap using information from any material found online or offline pertaining to the runtime stack and the heap as a source of background information to acquaint oneself with these component systems of a program's memory. The second task is a writing piece which involves explicit memory allocation/deallocation versus garbage collection within programming languages using examples from C and Java. The third task pertains to a blog entry from "Monday Morning Haskell" and contains ten salient sentence sequences which revolve around the idea of memory management. The final task is a three-paragraph review on the paper titled, *Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study*, that I believe would resonate with a senior computer science major who is about to graduate and commence a search for an entry level software developer position.

Task 1: The Runtime Stack and the Heap

Within programming languages, memory used within each function of the program must be stored somewhere and kept while in use. For this reason, the stack and heap exist in order for functions to properly store their temporary data while in use. With the stack and heap come in handy for functional programming but also come with drawbacks if not used carefully.

For every call of a function within a program, a stack frame is pushed onto the runtime stack which contains the memory used for the function's local variables, the function's parameters, and all of the data the function uses. We can think of this like a stack of dishes, where the last one in is the first one out. When the program is done with the function, the stack frame is removed, or popped, off the top of the stack. This releases the memory that was reserved for the original stack frame in order to make room for another function call. The issue with this is when the stack frame fills up, this results in a stack overflow error.

In certain programming languages, such as C and Rust, a heap is a certain data structure used like a stack but with the ability to hold more information than the stack. The heap is used in conjunction with the stack to hold larger sets of data than the stack would be able to contain. The stack simply contains a pointer to the data within the heap, but this can open up the issues of double free, when a program attempts to free an already freed memory address, and dangling pointers, if one of the pointers frees the memory that another pointer is pointing to. These issues arise when we have several pointers all pointing to the same memory location and the programmer isn't careful about their use of deallocating the memory. Within Rust, the concept of ownership comes into play where the stack is used to keep track of the program and the variables stored are moved from owner to owner and effectively are kept until all functions which own the data have finished using the data, eliminating the need for a garbage collection system.

Task 2: Explicit Memory Allocation/Deallocation vs Garbage Collection

Programming languages require specific sizes of memory for different data types. In order for the program to temporarily store this information, a memory block of a certain size must be reserved for the data to be saved to. These are known as the processes of allocation and deallocation of memory.

Within some programming languages, such as the C and C++ programming languages, they require the programmer to manually reserve space for blocks of information and then manually release the reserved memory block within the code once the program is done using the information. The process of reserving space is called allocating memory; within C, this is done with the malloc() and calloc() functions. The process of releasing memory is called deallocating memory; within C, this is done using the free() function using the pointer to the data block being released as the sole argument.

Within many higher-level programming languages, the inclusion of a feature called garbage collection is implemented. Unlike C, where explicit allocation and deallocation are used to manually reserve and free memory, languages with garbage collection, such as Java, automatically identify memory locations which are not in use and frees up the memory by itself, without the programmer having to code it in. This is beneficial for obvious reasons, but the main downside is that garbage collection is a process that requires extra time to analyze and clean up the data blocks which are no longer in use during run time. This results in longer execution times than other languages which use explicit allocation and deallocation of memory.

Task 3: Rust: Memory Management

“In C++, we explicitly allocate memory on the heap with new and de-allocate it with `delete`. 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. But it doesn't work quite the same way as C++.”

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

“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.”

“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.”

“What's cool is that once our string does go out of scope, Rust handles cleaning up the heap memory for it! We don't need to call `delete` as we would in C++. We define memory cleanup for an object by declaring the `drop` function.”

“Deep copies are often much more expensive than the programmer intends. So, a performance-oriented language like Rust avoids using deep copying by default.”

“Memory can only have one owner. This is the main idea to get familiar with. Here's an important implication of this. In general, passing variables to a function gives up ownership**.”**

“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.”

“You can only have a single mutable reference to a variable at a time! Otherwise, your code won't compile! This helps prevent a large category of bugs!”

“Slices are either primitive data, stored on the stack, or they refer to another object. This means they do not have ownership and thus do not de-allocate memory when they go out of scope.”

Task 4: Paper Review: Secure PL Adoption and Rust

When it comes to programming languages, safety and performance are extremely important when it comes to memory management. That's why programming languages like Rust, which handle safety and performance exceptionally well by incorporating the ideas of ownership, borrowing, and lifetimes, have been slowly growing in popularity within the workplace. Within the paper titled, *Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study*, the attributes of ownership, borrowing, and lifetimes are brought up and discussed in detail and the following pertains to the discussion of the positives and negatives of including these features.

The benefits that the Rust programming language provides offer a safety net from common memory management issues present in languages where explicit allocation and deallocation are primarily used. The programming disciplines known as ownership, borrowing, and lifetimes assist Rust programmers to avoid the common memory errors that many languages face. The idea of ownership makes it so each value within Rust has a variable which is its owner, only allowing one owner at any time for a given variable, and automatically deallocating the memory once it is out of scope of being owned. The idea of borrowing allows Rust to transfer ownership of variables between different owners, making it so the threat of dangling pointers and double frees are practically eliminated. The downside to this is that it prevents Rust programmers from creating doubly linked lists as well as graph data structures without the use of aliasing utilized through external libraries. The idea of lifetimes in Rust makes it so functions which borrow a reference to the allocated memory won't have the memory deallocated by the previous function which sent the reference over. The combination of these three concepts creates a virtual safety net within the Rust programming language which can be extremely beneficial.

Along with the Rust language benefits, come downfalls as well. With many reporting that Rust comes with a steep learning curve to learn exactly how the safety precautions that Rust implements need to be worked with and worked around. As well, many users of Rust reported other issues pertaining to the programming language including, but not limited to, limitations of library support, length of compile times, and worry of future stability and maintenance within the language. Many users also reported that, even with the quality of available tools and libraries within Rust, the language still lacks some very critical libraries and infrastructure.