Csc344 Problem Set: Memory Management / Perspectives on Rust:

Cameron Francois
May 12th, 2023
CSC 344

Task 1 - The Runtime Stack and the Heap:

Stack and Heap are crucial components that programmers should take the time to gain the proper knowledge on. Especially when writing in languages like C, C++, and programmers who have been around with java could gain more insight on how memory management is functioning. In the next two paragraphs we will go more in depth on Stack and Heap, this will give more insight on both of them. Overall, both of these are important to know for programmers and will only expand their knowledge as a computer scientist.

Stack is a component or concept that is generally used for a program's memory management and stack includes sections called stack frames. Those stack frames are created before the function is performed and will be deleted once the function execution is completed. Basically it follows the rule of fulfilling the requirement of giving space for the function to perform. Like most things before we are able to remove or delete the next stack frame we must remove the top stack before the next one. A lot of the stack function will automatically perform rather than having the programmer define otherwise.

Heap is also a component or concept that is generally used for a program's memory management. Although, there are key differences in Heap comparing it with stack. Heap specializes when a programmer wants to keep data or just pass the data along from function to function. This will obviously depend on what language is being used but when keeping data it will need to find/allocate space for this data. SOme differences is Stack generally has various sections of memory while Heap does not need to do this and also does not one of the crucial ones is the first in last out rule followed by Stack.

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

For a programmer there's key concepts that can be really useful for whatever language the programmer is using. Having deeper understanding only furthers the proper information needed when faced with various decisions in the program. Within the next two paragraphs we will discuss explicit memory allocation and deallocation versus garbage collection. Both of these topics are crucial for programmers to fully grasp and both could prove beneficial to have this knowledge depending on the language that is being used. So both of these will be discussed to more of an understanding that could provide more insight.

Memory management focuses on the process of both explicit allocation and deallocation of the memory in which to store the data. This of course helps to run any program and this will vary on the language that is being used in some cases how it functions. In some cases like C++ the programmer will need to be a little more proactive with the management of the program by using functions like New() and Delete() which is also similar in C which uses Calloc() and Malloc() both to help allocate memory. There are also methods to deallocate memory like free(). The benefit of the programmer being more of an active role will help how the program functions and even can run on hardware limited systems. But despite some of the positives of explicit memory management it's worth mentioning that it does have some downfalls like anything else. One of the main issues is it could be hard to follow or tough to set up and there could also be memory leaks without a proper system that could get messy.

Although with garbage collection the programmer will be disconnected from the memory management system. So as we somewhat discussed in the previous task, garbage collection will allocate memory once the program starts running and will also reallocate. It does this when the data is being used the memory will be allocated and once it stops then it will then reallocate. This is a lot better for a programmer in the sense that it is more convenient but it's generally a slow function compared to the explicit memory management. Overall, a programmer could prefer this due to the low effort needed from them but as mentioned above its not necessarily the ideal process.

Task 3 - Rust: Memory Management:

1. The suggestion was that Rust allows more control over memory usage, like C++. In C++, we explicitly allocate memory on the heap with **new** and deallocate it with **delete**. In Rust, we do allocate memory and deallocate memory at specific points in our program. Thus it doesn't have garbage collection, as Haskell does.

2. In this part, we'll discuss the notion of **ownership**. This is the main concept governing Rust's memory model. Heap memory always has **one owner**, and once that owner goes out of scope, the memory gets deallocated.

3. When that block of code ends, the variable is **out of scope**. We can no longer access it. Rust works the same way. When we declare a variable within a block, we cannot access it after the block ends.

4. 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.

5. Deep copies are often much more expensive than the programmer intends. So a performance-oriented language like Rust avoids using deep copying by default. But let's think about what will happen if the example above is a simple shallow copy. When **s1** and **s2** go out of scope, Rust will call **drop** on both of them. And they will free the same memory!

6. In Rust, here's what would happen with the above code. Using **let s2 = s1** will do a shallow copy. So **s2** will point to the same heap memory. But at the same time, it will **invalidate** the **s1** variable. Thus when we try to push values to **s1**, we'll be using an invalid reference.

7. **Memory can only have one owner.** This is the main idea to get familiar with.

8. In general, **passing variables to a function gives up ownership**. In this example, after we pass **s1** over to **add_to_len**, we can no longer use it.

9. 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.

10. This works like a **const** reference in C++. If you want a *mutable* reference, you can do this as well. The original variable must be mutable, and then you specify **mut** in the type signature.

Task 4 - Paper Review: Secure PL Adoption and Rust:

So languages like Rust and Go were initially developed to fight against the crucial memory safety-related weak points. While it's important knowing that getting secure software development is a tough task with weak points in code generally on a daily basis. Back to Rust and Go being created these were helpful to negate the burden or hassle the lower end developers had to deal with regarding developing safe and impactful code. Since memory weak points are so critical, most would invest in helping convert a language like Rust into a helpful language dealing with these weak points despite the expensive price tag that comes along with it.

Some background knowledge of Rust is it was created by Mozilla in 2014 with its first proper release. It is an open-source system programming language that's goal is to help developers create secure applications and prevent the typical faults you might see. Most of the languages that promote safe typing often use a garbage collection, although with Rust it has a strict ownership with three main rules it follows. So in Rust each variable will have an owner but there can only be one owner per value and the value must be dropped once it reaches out of the scope. There are other key items in Rust like borrowing and lifetime that go along with the ownership aspect. All of their goals are simple to ensure or help there is safety with the security and avoiding any downfalls that are critical.

Along with any benefits or plus sides to Rust there are also drawbacks that it faced as well. Some of these drawbacks are that there are some areas that make it unsafe that a programmer can use these unsafe blocks. Along with these functions and methods having the possibility of being unsafe there's the fact Rust as a language can be difficult. It's generally challenging than most languages making it tough to want to learn at first but has a lot of neat features that make it worth it. Overall, Rust has a lot to learn about and afford to developers, if they are willing to take the time and learn Rust it will surely be worth the time and effort.