Csc344 Problem Set: Memory Management / Perspectives on Rust

Task 1 - The Runtime Stack and the Heap

The stack and the heap are importance abstractions for computer scientists to understand, and understand well, as they are of critical importance when writing in languages like C or C++. Even experienced programmers in languages like Java that implement a garbage collection service benefit from a sound understanding of how memory management is executed. In the following paragraphs I will briefly define and explain Stack and Heap in a general sense. While various languages may handle some of the nuances differently, the overall abstraction remains relatively consistent.

The stack is a concept that is utilized for program memory management. The stack operates in separate sections called stack frames. When a function is called in the program that function gets its own stack frame added to the top of the stack with its local variables, and some other info, this is called allocation. Once that function completes the stack frame for that function is removed, or deallocated, and in turn all data inside is lost. Similar to a stack of dinner plates, the top stack frame must be removed before we can remove the next. This is referred to First in, Last out. The allocations and deallocations that happen on the stack are typically done automatically and not explicitly defined by the programmer.

Like the stack the heap is another abstraction utilized is program memory management. However, the heap differs in some keyways. What if you wanted to pass information from one function to another, or you wanted to keep data around after the function finished? This is where the heap comes in. If the programmer, or depending on the language – the function, wants to keep some info around for other parts of the program to reference they would allocate space for the data on the heap and add pointer to that space on the stack. Heaps, unlike stacks, are not contiguous sections of memory, they can end up with uneven holes as space is allocated and deallocated. They also don't follow a first in last out rule like the stack does.

Task 2 - Explicit Memory Allocation/Deallocation vs Garbage Collection

It's important for programmer to understand the innerworkings of the languages they write in. It is important as this deeper understand brings with it the knowledge require to make well informed decision as to what approaches are best for a given application. Will the application you are developing be running on hardware with limited resources? If so you may opt to utilize a language that supports direct memory management. Or if this is not a concern in the scope of your release, you may opt for the convenience of a language with some form of garbage collection. Over the course of the next couple paragraphs, I will briefly touch upon each of these two ideas.

Memory management is the process of explicit allocation and deallocation of memory to store data, run programs or systems. In languages that allow, or sometime require, the programmer to take an active role in this resource management have various instructions that the programmer use to facilitate such actions. For example, C uses Malloc() and Calloc() methods to allocate memory, and free() to deallocate memory. C++ also supports these functions but with the addition of New() and Delete(). Taking an active role in how your program manages the memory it uses can increase its performance and allow the program to run on hardware limited systems. However, explicit memory management is not without its draw backs, correct implementation can be complex or hard to follow and if not careful a developer could find themselves with errors from double freeing an allocation, dangling pointers created from a free,

and dreaded memory leaks. Memory leaks are particularly problematic if running on a small limited system and you don't have an OS to handle the mess for you.

With garbage collection, the programmer is totally separated from the nuisance of memory management. The programming language will dynamically handle everything the program needs to run during runtime. As the program runs, memory is allocated and reallocated accordingly for any data being used and once that data is no longer being referenced it is deallocated accordingly. While this is very convenient for the programmer, its is a slow process compared to explicit memory management described above.

Task 3 - Rust: Memory Management

1. 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++.
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 de-allocated. We'll see how this works; if anything, it's a bit easier than C++!
3. 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! This kind of "double delete" is a big problem that can crash your program and cause security problems.
6. 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! This kind of "double delete" is a big problem that can crash your program and cause security problems.
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**.
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. There's one big catch though! 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!

Task 4 - Paper Review: Secure PL Adoption and Rust

Memory safety and related incidents have always been a concern for computer scientists. Languages like Rust and Go were created to help ease the burden of lower-level developers trying to develop safe and efficient code. While adopting a new language in the work force can be an expensive and difficult task, its reported that 60-70% of critical vulnerabilities in Chrome, Microsoft products and in other large critical systems are owed to memory safety vulnerabilities. Therefore, investing in converting to a language like Rust could prove to be beneficial to some companies.

Rust is an open-source systems programming language created by Mozilla. Its creators claim the Rust can "help developers create fast, secure applications". They also argue that Rust can prevent seg. faults and protects thread safety. Rust has some additional salient features that might appeal to some developers. Rust draws elements from functional, imperative and object-oriented languages. Rusts variables are immutable by default and, once bound, can only be changed again by explicitly making them mutable. This does wonders for code safety and pairs nicely with how Rust handles ownership. Speaking of ownership Rust uses a programming discipline consisting of Ownership, Borrowing, and Lifetimes to ensure dangerous security related errors are avoided.

Rust does have its down sides though. Rust is slow to compile, has a steep learning curve, and can therefor be hard to hire for as few are well versed in the language. Rust also has a limited selection of libraries, which increases its dependencies. While C++ is portable into Rust it's a laborious process. These are some of the reasons a company my have its reservation about adopting Rust over its current language of choice.