Second Problem Set: Memory Management & Perspectives on Rust

Task 1 - The Runtime Stack and the Heap Essay

In computer science, memory management is crucial, especially when dealing with content heavy programs. The runtime stack and the heap are a few memory allocation tools that deal with the allocation and deallocation of memory in a program. In the next few paragraphs, the functionality of the runtime stack and heap will be discussed in order to better acquaint the reader with these ideas.

As the name suggests, the runtime stack occurs when a program is run. When a function is called from the main program, the parameters and local variables of the function are stored on the top of the stack. These variables are stored until the function terminates, and are then popped off the top of the stack. Any variables not on the top of the stack cannot be accessed. The program then returns to the main program, and continues the same process automatically until the whole program terminates.

Heaps are quite different from stacks in that they deal with memory manually rather than automatically. That is, the user has to explicitly allocate and deallocate memory in the heap, giving greater control to the user, but creating possible problems such as memory leaks. Since the heap memory is considered a more permanent space, variables stored in the stack can have pointers that reference the corresponding values that are stored in the heap.

Sources:
- https://courses.engr.illinois.edu/cs225/fa2021/resources/stack-heap/
- https://cathyatseneca.gitbooks.io/data-structures-and-algorithms/content/recursion/the_runtime_stack.html

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

---

The way memory is managed, whether manually or automatically, is a very important concept in computer science. Since there is a finite amount of information that can be stored in a program's memory, deallocating memory is needed for a smoother and faster runtime. In the following paragraphs, allocation and deallocation of memory will be discussed, as well as the process called garbage collection.

One of the most important aspects of computer science is how to deal with memory allocation and deallocation in a program. Memory allocation creates space in a program for variables to be stored until they are used by the program. Once these variables are no longer needed, they are deallocated in order to clear up space for new memory. Some languages do this in the background without the user having to manually manage the memory, while others, such as Rust and C++, allow the user to allocate and deallocate memory in a way that they see fit. This has a lot of advantages, such as an increase in memory safety, but can lead to problems, such as memory leaks, if the user does not correctly deallocate memory.

Garbage collection is a process where any section of memory that is no longer being used is automatically deleted in order to clear up space in a program. This means that the user does not need to worry about allocating or deallocating memory within their program. While this clears up time for the programmer when writing code, when it comes time to run the program, garbage collection can slow down the runtime. Whereas Rust is a language that has a more manual approach when it comes to memory, languages such as Java, Haskell, and Lisp rely on garbage collection for their memory management.

Sources:

- https://medium.com/@rabin_gaire/memory-management-rust-cf65c8465570
- https://www.cs.uah.edu/~rcoleman/Common/C_Reference/MemoryAlloc.html

Task 3 - Rust: Basic Syntax

Source: https://mmhaskell.com/rust/syntax

1. “One of the big changes is that Rust gives more control over the allocation of memory in one's program. Haskell is a garbage collected language. The programmer does not control when items get allocated or deallocated.”
2. “Both languages embrace strong type systems. They view the compiler as a key element in testing the correctness of our program. Both embrace useful syntactic features like sum types, typeclasses, polymorphism, and type inference. Both languages also use immutability to make it easier to write correct programs.”
3. “Rust distinguishes between primitive types and other more complicated types. We'll see that type names are a bit more abbreviated than in other languages.”
4. “We mentioned last time how memory matters more in Rust. The main distinction between primitives and other types is that primitives have a fixed size. This means they are always stored on the stack. Other types with variable size must go into heap memory.”
5. “While variables are statically typed, it is typically unnecessary to state the type of the variable. This is because Rust has type inference, like Haskell!”
6. “When writing a function, we specify parameters much like we would in C++. We have type signatures and variable names within the parentheses. Specifying the types of your signatures is required. This allows type inference to do its magic on almost everything else.”
7. “This is because a value should get returned through an expression, not a statement.”
8. “Rust has both these concepts. But it's a little more common to mix in statements with your expressions in Rust.”

9. "Like Haskell, Rust has simple compound types like tuples and arrays (vs. lists for Haskell). These arrays are more like static arrays in C++ though. This means they have a fixed size."
10. "Another concept relating to collections is the idea of a slice. This allows us to look at a contiguous portion of an array. Slices use the & operator though."

Task 4 - Rust: Memory Management

Source: https://mmhaskell.com/rust/memory

1. "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. "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."
3. "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."
4. "Instead, we can use the String type. This is a non-primitive object type that will allocate memory on the heap."
5. "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."
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."
7. "Memory can only have one owner."
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."
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. This means they do not have ownership and thus do not deallocate memory when they go out of scope."

Task 5 - Rust: Data Types

Source: https://mmhaskell.com/rust/data

1. “Rust is a little different in that it uses a few different terms to refer to new data types. These all correspond to particular Haskell structures. The first of these terms is struct.”
2. “When you're starting out, you shouldn't use references in your structs. Make them own all their data. It's possible to put references in a struct, but it makes things more complicated.”
3. “Rust also has the notion of a "tuple struct". These are like structs except they do not name their fields.”
4. “Rust also has the idea of a "unit struct". This is a type that has no data attached to it.”
5. “ But in Rust, an enum is the general term for a type with many constructors, no matter how much data each has.”
6. “Instead, Rust uses the match operator to allow us to sort through these.”
7. “But unlike Haskell, Rust allows us to attach implementations to structs and enums. These definitions can contain instance methods and other functions.”
8. “We can also create "associated functions" for our structs and enums. These are functions that don't take self as a parameter. They are like static functions in C++, or any function we would write for a type in Haskell.”
9. “As in Haskell, we can also use generic parameters for our types.”
10. “For the final topic of this article, we'll discuss traits. These are like type classes in Haskell, or interfaces in other languages. They allow us to define a set of functions.”

Task 6 - Paper Review: Secure PL Adoption and Rust

This paper touched on the advantages and disadvantages of the programming language Rust, as well as why companies might want to implement this language. In order to get a feel for how Rust users, both experienced and not experienced, view Rust, a survey was conducted. The results of the survey showed that some of the population viewed Rust as a language that is quite secure and easy to debug, while others thought it was hard to learn and sometimes slow.

One of the main advantages with Rust is that it tends to be more secure with its memory than say C and C++. This is in part due to the borrow checker, which is a compiler that helps to avoid memory management vulnerabilities such as dangling pointers. Many of the people who were surveyed agreed that one of the most appealing features of Rust is the increased security and safety. Others said that debugging a program in Rust is easy since the error messages are oftentimes very clear and point the user in the right direction. Rust's tools were also very liked amongst participants, with a good majority saying that they were good or very good compared to languages that they are well versed in. These advantages, along with others talked about in the paper, suggest that Rust might be a good language to implement in a company.

However, with its advantages, Rust also had some notable disadvantages amongst participants. One of the first drawbacks was that Rust was a very hard language to learn, with one of the biggest challenges being the borrow checker compiler. This learning curve along with the fact that Rust is not one of the most widely used languages (for example, Java) is what might make a company apprehensive about adopting Rust. However, some of the participants' companies still adopted Rust by highlighting the strengths of the language.

Source: https://obj.umiacs.umd.edu/securitypapers/Rust_as_a_Case_Study.pdf