CSC344 Problem Set: Memory Management / Perspectives on Rust

Learning Abstract

Task 1: The Runtime Stack and the Heap

Memory management is done by all programming languages whether it be automatically through garbage collection or done manually by the programmer. The purpose is to provide different ways to dynamically allocate portions of memory to programs at their request, and free it for reuse when no longer needed. The 2 main ways of doing this are Stacks and Heaps.

The runtime stack, or just stack, is the first memory management method I'll be discussing. Using the stack for memory management is typically safer than using the heap. The data is can only be accessed by the owner and it is the faster choice. You can think of the stack as an actual stack of memory that follows the last in first out principle. The compiler will place the memory on top of the stack and when the function call is over the memory is automatically de-allocated, and then the memory underneath can be accessed. Benefits of the stack are that it is easier, faster, and safer to implement. The downsides are that they are fixed sizes and can cause a shortage of memory, or stack overflow.

Heap allocation is much different than a runtime stack as the memory size is dynamic. The allocation and deallocation must be done by the programmer and is not as safe because the data stored is accessible by other threads not just the method running on it. The heap can be thought of as a pile of memory and can have memory allocated and deallocated at any time and any place in a program (therefore it's not as safe, but also dynamic). However, if you are not careful you can memory leaks, where memory has been designated for a task that is over and no longer needed but never deallocated and therefore wasted. The downsides are that its less safe than runtime stack, slower, and more difficult to implement for programmers. The benefit of course is that resizing is possible, and the memory can grow and shrink as the programmer reallocates memory.

Task 2: Explicit Memory Allocation/Deallocation vs Garbage Collection

I've covered two main tuples of memory management: The runtime stack and the heap. I briefly touched on will adding and removing memory from stacks and heaps through allocation/deallocation and garbage collection, but I'd like to dive more into them. To explain better I will reference a few programming languages that I've used personally that use explicit memory allocation and another that uses garbage collection.

Low level languages such as C and C++ are the most common that require explicit memory allocation and deallocation. Memory control in these low-level languages make them very efficient but can also be a pain to implement, and if done improperly, can cause memory leaks. As described earlier, memory leaks occur when we are done with a runtime process but fail to deallocate the memory and it is not freed up for other uses. As functions are declared it is important to use pointers to point to memory addresses that information is stored for use during runtime. It is also important at completed to make sure that memory is explicitly deallocated.

High level languages such as Java and Python no longer require that programmers explicitly declare memory allocation. Instead, they have built in process called garbage collection. The garbage collection will automatically detect when a variable or object is no longer using memory and will deallocate the memory for said object/variable. This is why the garbage collector is much easier to work with and less likely to cause memory leaks. But, with an added process to constantly be searching for memory to be freed up, the programs are slightly less efficient as the programmer has less control over how this is done.

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.”

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. “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.”

4. “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.”

5. “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. This causes the compiler error.”

6. “At first, s1 "owns" the heap memory. So when s1 goes out of scope, it will free the memory. But declaring s2 gives over ownership of that memory to the s2 reference. So s1 is now invalid. Memory can only have one owner. This is the main idea to get familiar with.”

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. When it's done, the original reference will still be valid.”

8. “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!”

9. “As a final note, if you want to do a true deep copy of an object, you should use the clone function.”

10. “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 de-allocate memory when they go out of scope.”

Task 4: Paper Review Secure PL Adoption and Rust

Based on information gathered by the article, Rust is an upcoming programming language, comparable to Google's Go, designed and created by Mozilla. The purpose of Rust to is to be a replacement for C and C++ that aim to be "fast, low-level, AND type- and memory-safe". The memory safe portion seems to be of utmost important. C and C++ do not enforce memory safety automatically unlike their high-level counterparts. The article interviews 16 senior developers who have actively worked with Rust on their teams and have attempted to get their companies to adopt Rust in order to take advantage of it's benefits.

The benefits of Rust far outweigh its disadvantages, according to a survey of 178 participants where 74% have used the language for more than a year. Some of the most important strengths include its success at maintaining security and performance – as a low-level language that is driven on preventing memory leaks. Programmers are probably most enthralled by it's use of memory "ownership", in which only one variable can point to a memory location at a time. Rust enforces strict rules on how to borrow that ownership, how to clone it, and how it maintains memory allocation when ownership is changed. While it does not have the common garbage-collection we are used to, it has it's own version that allows it continue running smoothly safely and quickly without the need to constantly stop and check for memory to collect as high-level languages would. Rust also makes it easy to find solutions to problems because of the high-quality documentation and clear error messages.

The drawbacks seem slim, as it has a steep learning curve to learn the paradigms that enforce security guarantees. But of course, are outweighed by the benefits listed above. Programmers are concerned about it's limited library support, future stability and maintenance. However, I believe if Rust continues to climb the ladder of widely used languages these concerns will be invalidated. The article overall presents excellent detailed results from its participants in the survey, with detailed graphs and numbers outlining benefits and cons such as time to compile, is it easier to maintain, is it quick to implement? Overall, I believe the article makes a good argument on the adoption of Rust and why it could eventually become a competitor for the top spots in programming languages. Any senior looking for a job out of college would benefit from taking some time to learn rust and may even be intrigued enough to find an employer that has already adopted the language or is interested in doing so.