Joseph M. Scollo

Memory Management / Perspectives on Rust:

The Run Time Stack and the Heap

Memory management is one of the most, if not the most paramount concern when it comes to computation systems. Different programming languages attempt management in various approaches. A very common approach is a run time stack and heap.

A run time stack is an array of memory addresses that is directly accessed by the CPU. Among these array of registers there I a reserved one called the ESP. This register is a stack pointer that points the next operation on the stack. Each register is 32 bits an manipulated by push and pop commands.

Heap memory, also known a dynamic memory differs from stack memory. Stack memory is allocated and deallocated automatically. Heap memory is dynamically allocated by the user and must be deallocated by the user if no garbage collector is present.

Explicit Memory Allocation/Deallocation vs Garbage Collection

Memory management, as mentioned previously, is often done with a stack and a heap. Since memory on the stack is trivial, when we say management we mostly are referring to the heap. When the program asks for a memory block to allocated or freed this explicit. Implicitly allocated or deallocated memory happen automatically and is managed by a garbage collector.

Programs like C and C++ explicitly allocate and deallocate memory. If a programmer just wanted to keep track of a few variables in a small array then a the stack will suffice. However, large data collections will be to large for the stack and likely cause a stack overflow. This is where explicit memory allocation is so powerful. If you know the size of the data you have then that

exact amount can be allocated to the heap. It must also be explicitly deallocated or freed. Since there is no garbage collector for these languages it is crucial to free blocks of memory to avoid memory leaks.

Languages like Java and C# implicitly manage memory, since garbage collectors automatically free up memory blocks, you could say the work implicitly as well. While it has some drawbacks, garbage collection relieves the programmer of having to manage his or her own memory themelves. The garbage collector automatically frees up memory that is know longer referenced.

Rust Memory Management.

1. Out of scope variables in rust get automatically freed. In java garbage collector would have to handle this and in C or C++ the programmer would. A unique and convenient feature. It certainly promotes cleaner code.

2. In certain functions, it may be difficult to free certain memory. If you need an object all the way until the end of the scope you may not be able to free it. Hence a memory leak. In rust once you are out o the function you are out of scope and rust frees the block for you.

3. Rust is a performance oriented language. Deep copies of variables can get quite expensive. Rust prevents this by only allowing one owner or ownership of memory. If two variables point to the same place in memory of those references will be invalid. Passing to a function gives the function ownership.

4. Rust still, however, like C and C++ allow you to pass variables by reference. Rust allows you to borrow ownership.

5. This can be done with mutable references. But rust only allows for one mutable reference at a time.

6. Because Rust has a one owner policy on memory, if to variables were to point to the same block, once both those references were out of scope that

memory would be freed twice which would likely crash the program. Like all dynamic memory, programmers should code carefully.

7. Rust allows copying of primitive types on the stack. Primitive types are fixed size so coppying is inexpensive

8. By controlling Memory ownership, it prevents rust from having to clean up too many out of scope variables which in turn makes it perform better.

9.While rust uses the stack and the heap similar to C and C++, it makes certain trade offs that make rust a worthy rival to programmers that want more performance.