Task 1- The Runtime Stack and the Heap

Memory management is an important part of any computer program. The runtime stack and the heap are two of the most fundamental components of this system, so it is important to understand the differences between them. In short, the stack is used for short-term operations with a fixed size, while the heap allows for dynamic memory allocation, but there is plenty more behind this.

The runtime stack is a block of memory used to store local variables and function parameters when a program is running. It operates using a Last-In-First-Out (LIFO) algorithm, meaning that the last item pushed onto the stack is the first one taken out. This makes the stack an efficient memory usage system for short-term operations, allowing the program to store a large number of variables quickly and efficiently.

The heap, on the other hand, is used for dynamic memory allocation. Rather than a fixed structure like the stack, the heap reserves an area of memory for each variable as it is needed. This allows the program to handle more complex operations as the need arises without taking up too much memory, making it ideal for long-term operations. Furthermore, the heap does not have a fixed size, meaning that it can grow and shrink depending on the amount of memory required.

Task 2- Explicit Memory Allocation/Deallocation vs Garbage Collection

Memory allocation and deallocation are important topics for programmers to understand. In computer programming, memory management is the process of allocating and freeing memory in a way that prevents errors such as unwanted overwrites or unexpected access violations. It is especially important for more complex languages, where explicit allocation and deallocation of memory is required.

Explicit memory allocation and deallocation requires that the programmer manually allocate and deallocate memory before and after use. An example of a language that requires explicit allocation and deallocation of memory is C++. In this language, the programmer must allocate memory manually using “new” and free the memory after.

Garbage collection is an alternative to manual memory management. In this system, a program will automatically detect and free any memory that is no longer being used. Since garbage collection is done manually, it eliminates any possible errors with freeing memory. A language like Java, utilizes garbage collection. Java will mark objects eligible for garbage collection, which is then free’d when the program has no more use for them.

Task 3- Rust: Memory Management

1) Heap memory always has **one owner**, and once that owner goes out of scope, the memory gets de-allocated.
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. Rust works the same way.
3) …string literals don't give us a complete string type. They have a fixed size. So even if we declare them as mutable, we can't do certain operations like append another string. This would change how much memory they use! Instead, we can use the String type.
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) …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) **Memory can only have one owner.**
7) **passing variables to a function gives up ownership**.
8) 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.
9) 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!
10) if you want to do a true deep copy of an object, you should use the clone function.

Task 4- Paper review: Secure PL Adoption and Rust

Rust is an important language for any new developer to learn, not just for the safety reasons such as automatic garbage collection, but also because Rust is said to “help developers create fast, secure applications”, and “prevents segmentation faults and guarantees thread safety”. These reasons alone already make Rust seem pretty great, but on top of that, Rust is an open-source systems programming language created by Modzilla, and has similar abstract types to that of Java and Haskell, so with some experience in those, Rust shouldn't be too hard to learn even if it’s just to throw it on your resume.

Rust also has “ownership”, which is that each value has a variable that is its owner. There can only be one owner per value, so the only way to use the value is by borrowing a reference to it. You may be thinking, “Won’t all these security features sometimes get in the way?” and the answer is, No- because Rust provides escape hatches that allows the developer to deactivate some of the security measures and safety checks of Rust. Things like dereferencing a raw pointer, accessing or modifying a mutable global variable, implementing an unsafe trait, and more are all made possible from this.

Rust is said to be used in a variety of areas including, databases, low-level systems, device drivers, virtual machine management, and kernel applications, so you never know when it could come in handy at some point in your career. Although it is frequently used, Rust isn’t some new groundbreaking programming language, one downside is since Rust is still quite new, it will often require some porting from other older languages. But, once you get that thing to compile, you’re pretty much guaranteed it will be a safe, fast, and reliable piece of code.