Task 1 The Runtime Stack and the Heap:

To be a good programmer, it is vital to understand the differences between the runtime stack and the heap as it pertains to memory allocation/deallocation. Because of this, I would like to research this topic and explain it briefly to you in two short but informative paragraphs.

The stack is a linear data structure that is similar to a stack that you can declare in many different programming languages like Java. This is because it is a LIFO (last in first out) data structure. The stack contains only primitive variables and references to objects that are in the heap, and memory stored in the stack is used only by one thread of execution (An example of a thread would be a method like the main method in Java).

The heap is a hierarchal data structure. The heap is used by the entire program and memory inside of it can be accessed by other threads. It contains all objects in a program. Heap data is less safe than stack data because all threads can access it. This can result in memory leaks. Heap memory takes longer to access than stack memory.

Task 2 Explicit Memory Allocation/Deallocation vs Garbage Collection

Memory allocation/deallocation is a vital process that occurs in programming that often goes unnoticed because of the ingenious ways in which it has been made automatic in many different programming languages. This does not mean we should overlook it, because often understanding what goes on behind the scenes can make one a much better programmer. For this reason, I would like to explain the two main forms of memory allocation/deallocation.

Explicit memory allocation/deallocation is a form of memory management used in languages like C and C++ that requires the programmer to manually allocate and deallocate memory for a program using certain functions. This allows for more optimized programs and therefore makes languages that follow this paradigm very fast. However, this is not without its drawbacks. Explicit memory allocation/deallocation introduces many different ways in which a programmer can make a mistake, and this can cause many bugs in a program.

Garbage collection is a system of memory management that makes allocation and deallocation much easier on the programmer. Garbage collectors automatically detect when an object is no longer needed in a program and removes it using advanced garbage collecting algorithms. This results in much less opportunity for a programmer to make a mistake and to introduce bugs into a program. Garbage collection can be introduced through a library or module in almost any programming language, but it is natively supported in languages like Java and C#

Task 3 Rust: Memory Management:

1: 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.

2: Rust works the same way. When we declare a variable within a block, we cannot access it after the block ends.

3: Instead, we can use the String type. This is a non-primitive object type that will allocate memory on the heap. Here's how we can use it and append to one:

```
let mut my_string = String::from("Hello");
```

```
my_string.push_str(" World!");
```

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++.

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.

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.

7: Here's an important implication of this. In general, 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: 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!

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 de-allocate memory when they go out of scope.

Task 4 Paper Review: Secure PL Adoption and Rust

Review of “Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study”.

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

In a sentence, I enjoyed reading through some of this paper. It begins by introducing why Go and Rust have become popular recently and what they offer to programmers. It seems that their main benefits are that they are fast low-level languages, and that they are much more secure than C and C++ when it comes to memory management.

The paper also conducts a study on senior software software developers and the Rust community to get a gauge on the benefits and drawbacks of Rust. It also describes how Rust handles mutability, types, and many other programming elements. It nicely shows how Rust keeps its memory safe through variable-value ownership. It also provides actual code examples to show what exactly it means when describing these things.

The study itself is very detailed. It provides graphs that pertain to how much Rust is used in the industry and the opinions on the language from those who use it. The study revealed that many are confident it produces bug-free code, takes less time to debug, and that it is easier to maintain, but that it also takes more time to compile and more time to prototype. In conclusion, I think this article provides a very detailed description of the benefits and drawbacks of Rust and languages that are similar to it.