Nov 27, 2022

Task 1:

The Run Time Stack and the Heap

Most popular languages in the coding society tend to fall into one of two categories they are guiding towards safety or they allow low-level control. Languages like Haskell, ruby, and python were safe but did not allow low-level control whereas languages such as C++ and C fell onto the opposite side of the spectrum. Rust is like a middle ground here in which it completely disregards the trade off and allows you to have both low-level control and safety. The problem with the above languages is in how they attempt to cover up their fault and in attempting to do so the programmer can make even bigger problems later on if not careful. Rust uses a concept called ownership which used both stacks and heaps and is how it offers both low-level memory control and safety.

When a rust program is executed a stack is made to track everything happening. The reason a stack is being used is because a stack is a special area of memory used to store temporary variable by a function. After the computing task is complete the memory with be automatically erased. Everything defined on a stack can only be defined by the following stack and stacks cannot be resized hence the data must be have a definite length. Stacks can be implemented in three ways which are arrays, dynamic memory and linked lists, their biggest problem is the shortage of memory but they make up for it with high access speed and efficiently managed space.

When stacks cannot be used because we need to hold a large amount of data we use a heap memory. The memory is not unlimited but we have more leeway than we did if we were to use a stack. Heaps are hierarchical data structures that allow you to access variables globally. The memory is allocated in any random order de-allocation is necessary. Allocation and deallocation are done by programmer instead of the compiler instructions. Heaps can be implemented using arrays and trees and their main issue is that the memory can become fragmented.

Task 2:

Explicit Memory Allocation / Deallocation vs Garbage Collection

Memory management is using the finite resource of memory as efficiently as possible. When you allocate memory you must always be careful to free that same memory because if you have a complicated system and you need to allocate several small pieces of memory you must remember to free those too. Implicit memory allocation is memory that is allocated usually on the system stack by the compiler and explicit memory allocation occurs through pointers. If you were to be using either of these in a dynamic situation where memory is being allocated and deallocated repeatedly and the chunks must be available at unknown times for an unknown length you'll have many complications. To deal with those problems there are automatic memory manager

systems better yet known as garbage collectors and they can be used to automate the process.

Memory allocation and deallocation can be either explicit or implicit and the difference is that deallocation is considered explicit when the program has to ask for a block to be freed. We would need to delete the memory manually as it is not automatically freed by the compiler with deallocation and we would need to do the opposite for allocation. Explicit allocation is useful for when we are unsure of the amount of memory we'd be using beforehand, when we want data structures without any upper limit of memory space, and when you want to use your memory space more efficiently. C is a good example of a language that require dynamic memory allocation / deallocation.

Garbage collection is a memory recovery feature built into some programming languages such as Java and C#. Garbage collectors automatically free up memory space that has been allocated to objects no longer needed. Garbage collecting makes sure that the memory limit is not exceeded or that the program reaches a point where it can no longer function. It frees developers from the responsibility of having to manually manage the memory which reduces the chances for bugs drastically.

Task 3:

Rust Memory Management

1. The suggestion was that Rust allows more control over memory usage, like C++. 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. But it doesn't work quite the same way as C++.
2. Heap memory always has **one owner**, and once that owner goes out of scope, the memory gets de-allocated.
3. 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. C++ doesn't automatically de-allocate for us! In this example, we **must delete** `myObject` at the end of the `for` loop block. We can't de-allocate it after, so it will leak memory!
5. 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. 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.

7. Another important thing to understand about primitive types is that we can copy them.
8. We've dealt with strings a little by using string literals. But 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.
9. Deep copies are often much more expensive than the programmer intends. So a performance-oriented language like Rust avoids using deep copying by default.
10. 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.

Task 4:

Paper Review: Secure PL Adoption and Rust

Rust is a multi-paradigm language with elements drawn from functional, imperative, and object oriented languages. Its traits abstract behavior that types can have in common. It is an open-source language created by Mozilla and it focused on helping developers create fast and secure applications. Rust though not a very popular coding language to those around our age is being shown to have quite the importance in the computer science industry. Rust was developed by Mozilla to

combat memory and safety related vulnerabilities in a simple yet effective manner compared to the complex new languages. Based on the information gathered from our surveys companies that have implemented Rust as or senior software devs that had worked it reported mostly positive things. Secure software development is such an important issue because we will always be striving to find new ways to protect our data so learning Rust or gaining some insight on the language would increase your chances of being hired. If not rust Go is a language developed by Google for the very same purpose as Rust so adding either of these skills to your repertoire is greatly encouraged.

In a next set of interviews conducted we asked our participants that were learning Rust why they chose to start and most of their responses fell along the lines of "because it is marketable or a good job skill." Don't become confused Rust is by no means an easy language to learn. The learning curve on this language is quite tough and that's me putting it lightly. Some people have said it takes them up to a month to write a program without resorting to unsafe blocks. Six months in to learning Rust and many reported that they weren't too comfortable with the language. One of the participants commented, "You spend 3–6 months in a cave, breathing, eating and sleeping Rust. Then find like-minded advocates who are prepared to sacrifice their first born to perpetuate the unfortunate sentiment that Rust is the future, while spending hours/days/weeks getting a program to compile and run what would take many other 'lesser' languages a fraction of the time."

The biggest challenges reported were the borrow checker and the overall shift in the programming paradigm.

Rust isn't all thorny branches and rocky mountains though. Rust improves confidence in code because once your code does compile many have reported confidence in the fact that their code is safe and correct. Debugging in Rust is supposedly shorter in Rust than in many other languages and Rust makes most interviewees feel as if their code is bug-free. While the initial time to design and develop a program in Rust is sometimes longer due to the borrow checker our participants from this survey have strongly stated that Rust reduced the overall development time from start to finish.