Csc344 Problem Set: Memory Management / Perspectives on Rust

By Miguel Cruz

Task 1 - The Runtime Stack and the Heap

Rust was designed to be a safe language and it does not require a garbage collector because of that, it is efficient and fast because of the Stack and Heap memory it uses while the program is running. The Stack is significant because it keeps track of everything that is happening inside the program but in a small degree because it is limited, the information is then stored in the Heap which is limited as well, but it has a larger size than the Stack and utilizes a pointer that references the Stack. In the following paragraphs we will dive deeper into these two components.

The runtime stack is a data structure that stores the call stack, which is a list of the currently executing functions and the arguments to those functions. A stack is similar to the piles of books in a library. The last book you look at from the top of the pile is the first book you put back. When a function is called, a new book is added to the top of the pile. When a function returns, the last book on the pile is taken off and placed on the top of the pile of books waiting to be read. The call stack lists all of the currently active functions with their arguments. When a function is activated, the runtime stack stores that function's return address and its local variables on the stack. The stack also stores the memory location of the function that called the current function When a function completes, the runtime stack removes its return address and local variables from the stack. It then stores the memory location of the function that called the now-completed function in the stack. The stack continues to store the memory locations of the functions that called the previous function, until the original function is activated again.

The runtime heap is a data structure that stores dynamic memory. Dynamic memory is memory that is allocated and deallocated at run time. The runtime heap contains a list of free memory blocks. Each free memory block has a header that contains the size of the block and a field that stores the memory address of the next free block. Each memory block contains a field that stores the memory address of the next memory block after it. This field forms a linked list of free memory blocks. When the runtime heap has free memory blocks, it allocates a new memory block and assigns it to the next free memory block in the list. When the runtime heap is out of free memory, the runtime heap searches the list for the largest memory block. It then allocates a new memory block and assigns it to the next free memory block in the list. When a memory block is no longer needed, the runtime heap frees the memory block and assigns it to the next free memory block in the list.

Task 2 - Explicit Memory Allocation/Deallocation vs Garbage Collection

Explicit memory allocation and deallocation is when the programmer manually allocates space in memory and manually deallocates that allocated memory when it is no longer needed. The following is a simple example of explicit memory allocation and deallocation Implicit memory deallocation is when the programmer does not manually allocate or deallocate memory, but rather the operating system does it automatically. In the following below we will discuss methods of implicit memory management like garbage collection and explicit memory management.

The meaning of explicit allocation/deallocation of memory is that when a programmer wants to allocate memory for a variable, the programmer must explicitly tell the compiler to allocate memory for the variable, and when a programmer wants to deallocate memory, the programmer must explicitly tell the compiler to deallocate the memory. Some languages require the programmer to allocate and deallocate memory explicitly. C and C++ are two examples of languages that require the programmer to explicitly allocate and deallocate memory. In C, a programmer can explicitly allocate memory using the malloc() function. The malloc() function takes two arguments: the size of the memory to be allocated, and the address of the location to be allocated in. The malloc() function returns a pointer to the allocated memory. The programmer can explicitly deallocate memory by using the free() function. The free() function takes one argument: the pointer to the memory to be deallocated. The free() function returns void. In C++ a program can explicitly allocate memory using the new operator. The new operator takes two arguments: the size of the memory to be allocated, and the address of the location to be allocated in. The new operator returns a pointer to the allocated memory. The programmer can explicitly deallocate memory by using the delete operator. The delete operator takes one argument: the pointer to the memory to be deallocated. The delete operator returns void.

The meaning of garbage collection is the automatic reclamation of memory in a computer program when it is no longer needed. Garbage collection is mostly used in a high-level programming language to avoid manual memory management. It is a form of automatic memory management. Some implementations of garbage collection are specific to a particular programming language, while others are more general or portable. Generally, garbage collection is implemented by a dedicated program running in the background, or at regular intervals, that checks for memory that is no longer in use by the program and reclaims it for re-use. In C and C++, the programmer is responsible for implementing manual memory management and can choose to explicitly free allocated memory. Memory that is allocated but not explicitly freed is lost and cannot be recovered. The C and C++ languages do not include garbage collection. In C# and Java, the garbage collector is a part of the runtime system. The programmer does not have to write code to free memory that is no longer in use. The programming language Lisp was one of the first programming languages to provide garbage collection, where objects are allocated and deallocated without explicit reference counting or pointer manipulation. Garbage collection in Lisp is performed by a garbage collector running in the background, freeing memory when it determines that the memory is unreachable by any program variables.

Task 3 - Rust: Memory Management

1.“But let’s think about what will happen if the example 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!”

2.“Rust works the same way. When we declare a variable within a block, we cannot access it after the block ends. (In a language like Python, this is actually not the case!)”

3.“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.”

4.“You can only have a single mutable reference to a variable at a time!”

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.“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.”

7.“In Rust ownership) Heap memory always has one owner, and once that owner goes out of scope, the memory gets deallocated”

8.“Certain primitive data known as “slices” have no ownership and thus do not free up memory when they are out of scope.”

9.“Another important thing to understand about primitive types is that we can copy them. Since they have a fixed size, and live on the stack, copying should be inexpensive.”

10.“In Rust, we do allocate memory and de-allocate memory at specific points in our program.”

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`

The Rust programming language, which is a new programming language launched in 2010. It is similar to C when it comes to memory allocation, but it is smarter by utilizing smart pointers compared to C that you need to manually allocate and free memory. In C, it is difficult to manage memory, since sometimes it does not free properly, therefore it is known for having memory leaks if not careful. Rust is able to perform the same job but being much more secure since it tries to take care of these memory leaks. Rust also has the ability to download library packages, which keeps the program updated to increase memory security as much as possible. It has a feature of using different types of compilers and can be used to compile the program on any operating system. The program is written in such a way that it can be run on multiple platforms without any changes in the code.

This programming language has its own syntax and uses concepts of having no pointer arithmetic. It also has the concept of ownership and does not have the concept of a global variable. The variable which is created is local to its scope. Also, it uses the concept of borrowing as well as a few other concepts which are new but similar to other languages. Rust has a package manager which is used for managing dependencies and it is similar to the apt-get system. Rust also has a version manager for managing the versions of libraries. It also has a package manager for distributing packages and installing them. The compiler of Rust is written in Rust itself and it supports multiple platforms, which makes it cross-platform, so that one code will work on all platforms without any modification. Rust compiles to native machine language, so that it produces native binaries, unlike C/C++ which produces executables in bytecode which needs to be translated into machine code before being executed, resulting in slower execution time. Rust code runs at least 2× slower than C. Therefore, the main aim of this programming language is to increase security without sacrificing the performance.

Overall, the point in the paper that Rust sold me to using rust was the detail that it combines advanced type systems with low level control over memory layout and performance characteristics it is able to eliminate entire classes of bugs found in most other programming languages including null dereferencing, buffer overflows, use after free, double free or delete as well as race conditions while providing as fast or faster execution times than comparable C implementations – or faster than C++ implementations when garbage collection is employed instead of manual memory management.