Advanced Systems Programming (H) 2021-2022 – Exercise 1

Yao Du, 2359451D

February 20, 2022

Part 1

There are several typical approaches of memory management, which can be categorized into two main kinds: manual and automatic memory management. Languages, for example, C requires programmer to manage the memory and resources manually but error-prone. JVM-based Languages like Java, Scala use garbage collectors which releases the heap space automatically, however, requires more runtime overheads. Specifically, many scripting languages (Python, Ruby) adopt an automatic management model called reference counting. It is surely predictable and understandable, but requires extra space to store the references. And it is not suitable for low-level code where performance is critical-concerned. Specifically, the automatic memory management of Rust is based on an approach of region-based lifetime tracking, offering more predictability and stronger compile-time behavior guarantees.

The region-based model is derived from the concept of stacked-based memory management. Every function call would push a new frame onto the stack, holding corresponding local variables and parameters. When current function finished, the frame with the contents would be then popped out from the stack. The scopes of functions are usually nested within the other, and the lifetime of the content stored on them is based on the scope of that frame. It can be assumed that every construct has its own scopes/regions. And every variable lives within a region, and is automatically allocated when entering that region, and deallocated when leaving that scope. Rust extends this model to manage the heap, tieing object lifetimes to stack frames. Allocate objects with lifetimes corresponding to regions. Also, track how objects are passed between stack frames i.e. ownership tracking. While the object has been passed to other regions, deallocating objects only when the lifetime of their owning region ends. And this is achieved by the destructor for the type i.e. `drop()` method is called.

The code snippet below shows an example of the smart pointer **Box**<T> which allows to store a value on the heap and returned a local variable `b` on the stack, pointing to the allocated heap memory. The lifetime of `b` is based of the stack frame it resides. The heap allocated object has lifetime matching that of the Box. So when `b` goes out of scope, its destructor is automatically called by the compiler and the corresponding heap memory is freed as well.

Listing 1: Using Box<T> to Point to Data on the Heap

```
fn main(){
    let b = Box::new(5);
    println!("b={}", b);
} // here b and the heap memory
// it points to would be deallocated
```

The key difference between Rust and other programming languages and the core of region-based memory management is the ownership tracking. And the ownership is a set of rules that governs how to manage Rust program memory. Also, it is requires that Rust compiler and type system tracks ownership (borrow checker) of all data in a program. The Rust compiler should know when a value has been consumed, when it's reached the end of its lifetime.

And there are three ownership rules:

- Every value in the program has a single owner at all times
- There can only be one owner at a time
- When the owner goes out of scope, the value should be dropped

Rust also defines rules about the transfer of ownership of data in function and method calls. The first case (Listing 2) is that, a function which passed a parameter by value would take ownership of that value. It can be seen that the string variable `s1` is passed to the function **fn** takes_ownership, i.e. the ownership would be taken. So if we attempt to copy the pointer, length, and the capacity of the String data that are on stack and assigns it to `s2` (namely, moves `s1` into `s2`) at the line 4, a compile time error is reported. Because Rust considers `s1` as no longer valid. This actually avoids the double free error where both `s1` and `s2` go out of scope, they will both try to free the same memory. And at this example, the memory which `s1` points to would be deallocated by the function **fn** takes_ownership and `s2` would never get a chance to free up the same memory in main().

Listing 2: Ownership Move and Stacked-based Value Copy

```
fn main(){
    let s1 = String::from("hello");
    takes_ownership(s1);
    let s2 = s1; // Error,
    //the ownership of s1 is moved

    let x =5;
    makes_copy(x); // x(primitives)
    // is copied into func
    // and x is still valid
}

fn takes_ownership(some_string: String){
    println!("{}", some_string);
}// drop some_string when ends

fn makes_copy(some_integer: i32){
    println!("{}", some_integer);
```

```
}// drop value copy here
```

It is also noticeable in Listing 2 that, for types which implements the `Copy` trait, Rust would perform value copy. Some simple types already implement it since this is cheap computationally. So we can still use the variable `x` after line `8`, as `x` itself is still valid in main().

In Rust, a function can give ownership of a value to its caller, making the caller responsible for freeing that resource. The function or method cannot retain any references to the resource when it gives up ownership in this way. (Listing 3)

Listing 3: Function Return Ownership

```
fn main(){
    // get the ownership from function
    let s1 = give_ownership();
} // main is responsible
// to deallocate s1

fn gives_ownership() -> String{
    let some_str = String::from("hello");
    // return the ownership of some_str
    // to caller
    some_str
}
```

In some cases, we just want to use value in function rather than get the ownership of that value, we need the concepts of borrowing reference. When a parameter is borrowed, ownership of the resource remains with the caller. The function can use the resource it's borrowed for the duration of the call, but no longer. And when the function returns, the caller still has access to and ownership of the resource. In default case, the borrowing reference defined by & symbol is immutable.

Listing 4 shows the default immutable reference of `s1` is passed to **fn** calc_len. And when `s` of **fn** calc_len goes out the scope, nothing happens as it is a reference to a String and does not have ownership of what it refers to. Additionally, we are not allowed to change the borrowed value in this case.

Listing 4: Borrowing Reference

```
fn main(){
    let s1 = String::from("hello");

    let len = calc_len(&s1);//borrow

    println!("len is {}", len);
}

fn calc_len(s: &String) -> usize{
    // Error, cannot change the value
    // via immutable reference
    // s.push_str(", world"),
    s.len()
}
```

In contrast to immutable reference, Rust provides the unique reference to mutable object which is labeled by &**mut**. And to change an object, we either own the object which is not marked as immutable or own the unique &**mut** reference to it. Distinguish with these two different references, we can avoid the data races and iterator invalidation in Rust, as the Rust compiler and runtime system work together to control how references can be used, and to track ownership of references and the referenced values. The restriction are:

- An object of type T can be referenced by multiple immutable references. Or it can be referenced by exactly one mutable reference
- It is not allowed to have both mutable and immutable references to the same object simultaneously
- An immutable object cannot be referenced as mutable
- A mutable object can be referenced as immutable which would make this object immutable during the referenced time

As the elements of iterators in Rust are immutable references, it guarantees that the element cannot be changed. Meanwhile, no mutable references to the object can exist in compile time. Thus, iterator invalidation problem could be detected.

This restriction however make it impossible to express certain cyclic data structures in Rust, e.g. doubly linked-list. It trades expressive power for safety. Luckily, Rust also has a third type of reference, known as the raw pointer (∗**const** T and ∗**mut** T) which works just like C pointers. And by using it, it is possible to write cyclic data structures in the same way as C, but is exposed to memory safety issues. Programmer effort is needed to manage it manually as raw pointers have no automatic clean-up and are not guaranteed to point to valid memory.

It is worth mentioning that, reference in Rust has its own lifetimes (except for raw pointers), which is the scope for which that reference is valid. And if the lifetime is not annotated explicitly, Rust has implicit rules to infer the lifetimes in general cases. This technique provides compiler with enough information to prevent us from unsafe memory behavior like dangling pointer.

Part 2

C takes stacked-based approach to manage the stack automatically which is effective. And the pattern of region-based memory management is derived from it, so both Rust and C can effectively manage the stack. However, C requires programmer to manage the heap memory manually. Using `malloc()` to allocate the heap memory and `free()` to free up the space. So it is common that memory leaks exist in C program if things related to heap are not processed successfully, and usually C compilers do not provide any approaches to report these errors before running the code. Once errors exist, it is hard to debug the root cause, especially in data races case. Also, managing memory manually has unpredictable overheads. As a programmer we have no idea how long those calls will take to execute. By contrast, Rust has the concepts of ownership tracking and lifetimes, enforcing many unsafe runtime behaviors to be checked efficiently in compile time. The data is clean-up automatically at the end of their lifetime when goes out of scope without the explicit syntax. And it offers predictable timing and has no run-time cost compared to manual heap memory management. Though, it loses generality, since the region-based management pattern ties the lifetime of heap allocation to that of stack frames.

Since C requires extra effort to manage the heap, there are several runtime bugs might happen: use-after-free, double free, iterator invalidation and race conditions. Type system of Rust tracks ownership and each value can only have one owner at the same scope so use-after-free or double free problems can be prevented at the compile time. The destructor of data would be called safely when it goes out of the scope which holds the ownership. As for data races with multiple threads, the restriction on how to use the borrowing references make it impossible for two threads to each have a mutable reference to the same object. As mutable and immutable reference to a mutable object cannot exist in the same scope simultaneously. It further prevents the iterator invalidation as the iterator in Rust requires the immutable reference of the object. So it is impossible to change the object being iterated over whilst the iterator exists. No dangling reference exists in Rust, as the borrow checker would check and infer the lifetimes of references. So any invalid reference will be detected before crashing at runtime. Consider that Rust compiler generates exactly the same code to allocate and free memory at the correct places as an equivalent C program using malloc and free. Consequentially, the timing and memory usage of Rust code is predictable.

However, the restriction rules about ownership and borrowing make it also impossible to express cyclic data structures in Rust safely. So at this point, programmer needs to use the raw pointer just like in C with care. Moreover, Rust cannot express shared ownership of mutable data while multiple references to point to immutable an object is fine. Because the shared ownership of mutable data might lead to race conditions. But Rust still provides the `RefCell` type, allows callers to borrow or mutably borrow the wrapped value, and enforces at run time that there can only be immutable borrows or a single mutable borrow, but not both.

Part 3

It is known that Rust takes region-based memory management pattern, offering high efficient and predictable behavior without extra runtime overheads. Common memory error can be avoided before runtime and this is guaranteed by the ownership tracking system and lifetimes. So the allocation and deallocation logics of objects can always be guaranteed to be inserted in the correct places, just like using malloc and free manually in C. Though generally we can write a safe Rust without extra effort, it is essential to understand the concepts of ownership, helping us better understand the program, especially with large system.

Similarly, Languages, for example, Java uses a garbage collector to manage the heap memory automatically. In this case, programmers do not need to worry too much about whether there are any inactive objects are not deallocated yet. Because the garbage collectors are responsible to deallocate them at the runtime. However, this pattern introduces high overheads at the runtime which is unpredictable and wasting memory. Even when a garbage collector is free, it is still CPU-consuming. Also, garbage collector might invalidate the CPU caches during context switching events. Java hides the complex pointers or references for us, to some extent, we are beneficial from it. While in terms of the underlying layer, we would not be able to know what is going on. We have no way of knowing when the garbage collector performs a deallocation or when it finishes. By using garbage collectors, it does not imply that programmers do not need to worry about objects processing at all. Sometimes, it is depends on programmers' experience or context, whether programmers have good coding style and knowledge of how to debug memory leaks or do things like JVM optimization to gain higher efficiency. Improper processing of objects would also lead to memory leaks in garbage collection language, resulting in unused objects not being released.

Code below is an example of temporary memory leaks in Java where the lifetime of static field vector is the same as the entire program. And objects are added into the vector in loop. As the vector hold the references of objects, so these objects cannot be released by the garbage collector.

Listing 5: Memory Leaks in Java

```
private static Vector v = new Vector(10);

public static void init(){
    for (int i = 0; i<100; i++){
        Object o = new Object();
        v.add(o);
        o = null;
    }
}
```

It is notable that new garbage collection algorithms have lower overheads and are more predictable. In the case of Java, for example, this is still acceptable with a combination of copy and mark-compact algorithms and a generation-based memory management model. Despite the benefits of using garbage collectors, it is still impossible to detect memory errors like iterator invalidation or data races at the compile time in Java. If shared resources and multiple threads involved, it is surely hard to debug at the runtime and sometimes require extra resources (locks) to avoid unexpected behaviors. However, Rust's clever use of borrowing rule restrictions allows data races and iterator invalidation to be checked in advance at compile time.

Rust is suitable for systems programming where the performance and safety is critical, since Rust can guarantee avoid common memory errors and data races at the compile time. Consider the performance is critical, so garbage collection languages of course will not work ideally in this context. Also, Rust can represent the state machine models in a clean way, such as network protocol, file systems. According to ownership rules, Rust can enforce state transition in specified manner, and any unexpected state transition would be reported at the compile time. While in some other languages, the system state is often encoded in mutable variables, hidden in a mass of conditional operations, and spread throughout the code. This makes it difficult to validate the code against the state machine specification.

There are two possible state machine implementation strategies that leverage these insights and can be used in Rust. The first is to use enumerated types to represent the states and the events and to use functions to represent state transitions and actions. Listing 6 shows an example of state machine implemented using `enum` and pattern matching. Note in `transit(...)`, the parameter is self rather than the reference, enforcing the old state is consumed and transition occurs.

Listing 6: Model State Machine - Enumerated Types

```
enum State{
```

```
    Ready,
    Running,
    Finish,
    ...
}

enum Event{
    Init,
    Processing,
    Done,
    ...
}

impl State{
    fn transit(self, event:) -> Self{
        ...
        match (self, event){
            (Ready, Init) => Running,
            (Running, Processing) => Finish,
            ... // Invalid (State, Event)
        }
    }
}

struct StateMachine{
    state: State
}

impl StateMachine{
    fn new() -> StateMachine{
        ...
    }

    fn run(&self) -> Event{
    // return event with current system state
        match self.state{
            Ready => ...
            Running => ...
            Finish => ...
        }
    }
}

fn main(){
    let mut sm =
     StateMachine::new();
    loop {
        let event = sm.run();
        // state transition
        sm.state = sm.state.next(event);
        if (sm.state==State.Finish){
            break;
        }
    }
}
```

Another method is using the struct type to represent the state. With ownership rules, methods on these structs will consume the state and return a new state, enforcing the state transition and ensure nothing from the previous state is accessible in the new state.

However, as already mentioned in Part 1 and 2, it is not possible to write cyclic data structures in safe Rust, due to the ownership rules and borrowing reference restrictions. The typical example is to write a doubly linked list. Since generally update the relationship between nodes requires specific nodes hold immutable or mutable reference with different purposes. However, with the restriction of borrowing reference, we cannot have both mutable and immutable references to the same object. Figure 1 below is a case where we cannot add an element `d` to the end. Because node `b` already holds the immutable reference to `c`, while we still need a mutable reference to `c` to add a reference to node `d`.

Figure 1: Fail to Update Relationship

These restrictions prevent race conditions, iterator invalidation. Though, we can still use the raw pointer like in C to circumvent the restrictions, implementing cyclic data structures in Rust if we want.