

Advanced Systems Programming (H/M) 2024-2025 – Exercise 1

2575686N

24 February 2025

Part 1:

Memory Management in Rust

Rust's memory management divides the general memory area into two separate regions: (1) the stack and (2) the heap [1], [2]. The stack follows a last-in-first-out (LIFO) principle, where the most recently added data is the one to be removed soonest. Rust typically uses this area for variables local to a single function. While the stack strictly follows a LIFO principle, memory on the other memory region, the heap, is allocated and deallocated more arbitrarily, depending on the existence of suitable memory locations with fitting sizes.

Allocation and deallocation are managed automatically for both areas and are done by identifying the parts of code a declared variable is valid and accessible for, known as the scope. To do so, Rust enforces strict rules on data ownership, as defined in its documentation [2]:

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Figure A shows a sample code of this. As variable *s* enters the scope, it takes ownership of the string "hello". Once it leaves the scope, the variable is removed from the stack. With the string losing its owner it is released, known as being dropped.

```
{  
    // s is not valid here, it's not yet declared  
    let s = "hello"; // s is valid from this point forward  
  
    // do stuff with s  
}  
// this scope is now over, and s is no longer valid
```

Figure A: Variable *s* takes ownership of "hello" in Rust [2]

If a variable is assigned to another variable, the new variable takes ownership of the data of the variable assigned to it. The example in Figure B shows, how *String* "hello" is first assigned to *s1*. As *s2* is set to be *s1*, the ownership of *String* "hello" moves from *s1* to *s2*. *s1* is now considered invalid, meaning it can no longer be used for statements such as *println*. Further, *s1*'s position in the scope has no longer any influence on the lifetime of *String* "hello" with *String* "hello" no longer freed if *s1* leaves the scope. The memory used for *String* "hello" will now be freed when *s2* leaves the scope.

```
let s1 = String::from("hello");  
let s2 = s1;
```

Figure B: Ownership moves from *s1* to *s2* in Rust.

Similarly, a function takes ownership of memory as it is passed to it as an argument. The function is then known to *consume* the data, dropping it as the function is executed fully (Figure C). Ownership can be returned to the function caller by using a return statement of the data.

```
fn main() {  
    let s = String::from("hello"); // s comes into scope  
  
    consume(s); // s's value moves into the function...  
               // ... and so is no longer valid here  
}  
  
fn consume(some_string: String) { // some_string comes into scope  
    println!("{}", some_string);  
} // Here, some_string goes out of scope and "drop" is called. The backing  
  // memory is freed. some_string was "consumed" by the function
```

Figure C: Rust variable "consumed" by a function, adapted from the Rust documentation [2]

Another approach to passing data into a function is doing so by using references, known as *borrowing* (Figure D).

```
fn main() {  
    let s1 = String::from("hello");  
  
    let len = borrow(&s1);  
  
    println!("{}", s1); // s1 is still valid here, as s1 was  
                       // passed as a reference (borrowed),  
                       // but not consumed  
}  
  
fn borrow(s: &String) -> usize { // function borrows a &String variable  
    s.len()  
}
```

Figure D: Borrowing of a variable in Rust, adapted from the Rust documentation [3]

Two types of references exist: (1) *&T*, a shared reference to an immutable object and (2) *&mut T*, a unique reference to a mutable object [3]. As their names suggest, immutable references (*&T*) allow functions to obtain data, without being able to change it. This represents a read-only behaviour. Mutable references (*&mut T*) on the other hand, allow for changes in the data's value, a read-write behaviour. Rust enforces strict rules on references [3], [4]:

- An object of type *T* can be referenced by one or more immutable references (*&T*) or by exactly one mutable reference (*&mut T*), but not both.
- A mutable reference (*&mut T*) cannot be obtained from immutable data of type *T*.
- As an immutable reference (*&T*) to mutable data exists, the mutable data becomes immutable.
- A reference must always point to a valid object and must never be null.

This prevents race conditions, cases where multiple functions try to access and change the same data simultaneously.

If memory is to be allocated on the heap, Rust enables smart pointers to access these values [5]. This might be required to change the sizes of elements during run-time, as they might not be known at compile time or to create recursive data structures (e.g. linked lists). Examples are *Box<T>* [6] and *Rc<T>* (reference counted smart pointer) [7]. *Box<T>* allocates data on the heap with the smart pointer, stored on the stack, pointing to it. It takes ownership of the heap memory location and if defined as mutable may change its values. As the box leaves the scope, the memory is deallocated. Similarly, *Rc<T>* also provides a smart pointer to a heap location. However, while *Box<T>* enforces single ownership, *Rc<T>* can allow for shared ownership by multiple smart pointers.

In enforcing the strict ownership and memory rules described above, rust guarantees safety measures resulting in error prevention. Memory is released automatically, as locations are freed as soon as the referring variables or references leave the scope, preventing memory leaks typically happening in manually managed memory. By requiring references and smart pointers to always refer to valid objects, possibilities of dangling pointers and use-after-free bugs can be excluded. Thread safety is improved, as mutable references to the same data cannot co-exist, eliminating the risk of race conditions.

Part 2:

Memory Management in C vs Rust

While stack memory management is automated in Rust and C, management of the heap poses significant differences between them. As explored in *Part 1*, Rust relies on an ownership system, allowing to automatically deallocate heap memory as references leave the current scope, with the compiler deeming data to be no longer required [2]. C on the other hand, relies on fully manual effort by the programmers to correctly allocate (*malloc()*) and deallocate (*free()*) heap memory [8].

By relying on this ownership system, Rust implements a variety of memory safety features, limiting or even entirely preventing many risks, some of which are:

- Dangling pointers
- Null-After-Free Use
- Double Free Error
- Memory Leaks

All these errors commonly happen in C, as the responsibility of preventing these bugs is carried by the programmer. Therefore, they must track the lifetime of allocated memory precisely, while understanding the flow of the program in the lifetime of variables required. Due to Rust's rules considering references (unique mutable references), it becomes impossible to establish race conditions, allowing Rust code to be easier handled by multiple threads while removing the risk of deadlocks, something C cannot guarantee and can hence only be achieved by effort invested by the programmer. While none of this may pose a problem if done right, research by Microsoft suggested in 2019 that about 70% of vulnerabilities of their systems are memory safety issues [9],

indicating the mental capacity required to do so, reaches the limits of programmers. In Rust, on the other hand, programmers are allowed to rely on the memory safety guarantees provided by the nature of the language, lowering the strain on their mental load required for keeping track of correct memory management. For example, as a developer allocates heap memory using *Box<T>::new()*, they can easily predict this memory location to be freed by the compiler, with the exact point deducible from the program flow. When accessing the allocated memory, they can further rely on Rust's guarantee of preventing pointers to be dangling or accessing unexpected data.

However, to achieve this Rust enforces a strict set of rules. As with most matters of life, established rules, while providing guidance and security, typically limit the capabilities of what is possible. This is no different in programming languages, causing limits in achievable performance and flexibility. In contrast to Rust's tight set of laws, C provides programmers with more freedom. Powerful features of C, such as pointer arithmetic or unchecked type casting, are not natively supported by Rust. This can limit the possibilities of solving coding problems in the latter language. C further allows mutable access to memory from various parts of code simultaneously, possibly improving runtime efficiency and performance. However, all these advantages require safety sacrifices of various degrees. While some of these limitations can be overcome by more complex code (e.g. double linked lists), others may only be implemented by utilising Rust's *unsafe block* feature, allowing programmers to write code that does not follow Rust's safety rules. While sometimes useful to do so, it is generally not recommended, as it removes the safety guarantees otherwise established by Rust.

In terms of run-time performance, both Rust and C are competitive and fast languages. As most safety guarantees of Rust are established during compile time, its run-time performance is not much different from C, sometimes outperforming C [10], other times being slightly slower [11]. Zhang et al. [11] further showed in their study, that most runtime losses of Rust are induced by run-time checks, which can be disabled to make Rust's runtime like the one of C (Figure E, [11]).

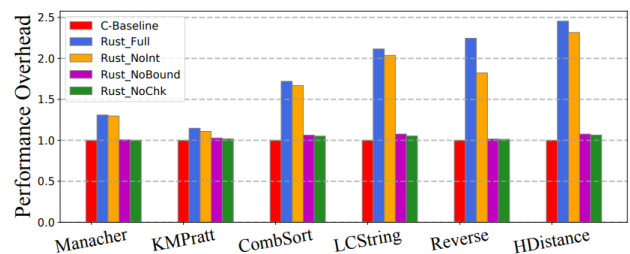


Figure E: Runtime comparison of Rust and C, with Rust ran with varying run-time checks enabled [11]

While C can generally improve run-time performance, it requires more careful software design, with much higher demands of the programmer to ensure memory safety. Rust might add additional overhead to runtime, however doing so guarantees a vast range of memory safety features. Either language might be the correct choice, heavily depending on

the requirements posed by the surrounding system. C may be a better choice for performance-critical systems, whether used for airbags or missile guidance systems. If problems caused by slow execution are more severe (e.g. an airbag does not open in time), than potential memory errors (e.g. the software of a car needs to restart as the airbag control encountered a bug), then C might be preferred. In many other systems, Rust may be a more suitable programming language. While pure execution speed might be slightly worse, advantages in terms of memory management might outweigh this. Rust is often used for Operating Systems, Parallel Computing applications or Embedded Systems, relying on Rust's predictability and safety measures. Modern systems carry processing power unheard of a few years ago. As a result of this, trade-offs in pure execution speed are more accepted nowadays, allowing for additional overheads required to improve error prevention, giving space for Rust to rise as a programming language.

Part 3:

Ownership Tracking vs Garbage Collectors

Rust's memory management system utilises an approach known as Ownership, relying on strict rules to track memory and automatically allocate and deallocate heap memory (further explored in *Part 1*). Those memory claims and releases can be deduced at compile time by following definite patterns of how the ownership model handles this. Another common practice of automated memory management is the use of *Garbage Collectors* (GC). Various algorithms of garbage collectors exist, such as Mark-and-Sweep, Incremental GC, Generational GC or Copying Collectors, with many popular languages such as Java or Python implementing them in pure or hybrid approaches. Generally speaking, garbage collectors "allow" programmers to fill memory with "garbage", unused data yet to be freed from memory. The garbage collection process is triggered as the memory is getting close to being filled. Depending on the algorithm used, the GC walks through memory, identifying used and unused data. The data no longer required is then released to make space for new structures to be held in memory. All of this happens during run-time.

A significant difference between both approaches, Memory management using ownership or using garbage collectors, is the transparency of how memory is released. As mentioned before, Rust allows deducing points of memory deallocation solely by examining the codebase. As the memory owner leaves the program scope, its claimed memory is released. The run-time overhead of memory release introduced is known to always happen at this point of the code, requiring equal time every time the program executes. This adds transparency to the code, with the programmer knowing exactly how the program will operate. Garbage collectors on the other hand, introduce some level of arbitrary. It is difficult to establish when exactly the garbage collector will decide to run, given it becomes almost

impossible to keep track of how full the memory is at any given point.

Furthermore, while the ownership-based model frequently releases small parts of memory, garbage collectors release substantial parts of memory all at once. The ownership model is required to run its release process more often but demands less time to do so. Garbage collection processes are triggered less often but require more time to execute. Also, while ownership models know at any given time which memory is still required, dependent on whether it belongs to some owner, garbage collectors must first iterate through memory to label data based on its requirements of existence. Again, adding some level of run-time overhead.

However, while Rust's ownership model improves the run-time overhead compared to languages relying on garbage collectors, it achieves this by increased demands from the programmer. While the rules of ownership allow to simplify the tracking of data requirements, they also add constraints. The code written must adhere to traits defined by Rust. This may make programming more difficult, prolonging the process of writing code. Code may also become more difficult to understand for less experienced programmers. The developers must know and understand all these rules of ownership to successfully write compiling code. These issues exist less in languages relying on garbage collectors. Code development is simplified, with the programmer allowed to focus solely on the actual algorithm implemented, rather than having to worry about the memory handling and implications underneath. One can rely on the garbage collector to do this correctly.

The rules introduced by ownership-based memory management models, further add constraints on "what is possible" in terms of code and data structures. A single owner of heap memory must be present at any given time. This constraint does not pose any problems for linear programs (Figure F) and data structures but poses significant problems for cyclic data structures.

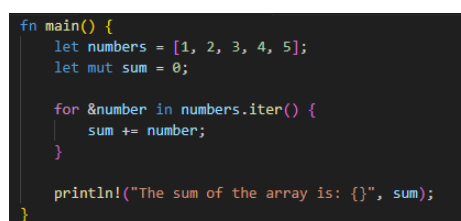


Figure F: Simple linear program in Rust. Ownership can easily be deduced from the program flow.

As multiple owners cannot co-exist, the creation of cyclic data structures is difficult or even impossible. In Rust, a doubly linked list, a cyclic data structure, can be implemented using the reference counted (*Rc*<*T*>) smart pointer in combination with reference cells (*RefCell*) as proposed by Claus Matzinger (Figure G, [12]). Reference counted pointers allow for multiple references to a single data element, while reference cells allow for mutation of data owned by an immutable reference. However, in doing so the program does not adhere to all typical rules applied by the rust ownership system and should hence be implemented with caution.

```

4  #[derive(Clone)]
5  struct Node {
6      value: String,
7      next: Link,
8      prev: Link,
9  }
10
11  type Link = Option<Rc<RefCell<Node>>>;
12
13  impl Node {
14      fn new(value: String) -> Rc<RefCell<Node>> {
15          Rc::new(RefCell::new(Node {
16              value: value,
17              next: None,
18              prev: None,
19          }))
20      }
21  }
22
23  #[derive(Clone)]
24  pub struct BetterTransactionLog {
25      head: Link,
26      tail: Link,
27      pub length: u64,
28  }

```

Figure G: Structures of a doubly linked list in Rust as proposed by Claus Matzinger [12]

Languages based on garbage collectors, do not require the implementation of ownership, allowing for more flexibility in the creation of complex data structures, with the creation of cyclic structures posing significantly fewer challenges (Figure H).

```

public class Node {
    int data;
    Node prev;
    Node next;

    public Node(int data)
    {
        this.data = data;
        this.prev = null;
        this.next = null;
    }
}

public class DoublyLinkedList {
    Node head;
    Node tail;

    public DoublyLinkedList()
    {
        this.head = null;
        this.tail = null;
    }
}

```

Figure H: Structures of a doubly linked list in Java [13]

The points raised above, make the ownership model of Rust an excellent choice for systems with strict demands in terms of execution time and predictability, such as Real-Time Systems or Embedded Systems. Languages based around garbage collectors are typically better suited for applications where flexibility and ease of development are of higher importance. These may include web pages, desktop applications or data analysis. However, solutions for more real-time focus exist, improving the responsiveness and predictability of garbage collectors [14].

References

- [1] 'The Rust Programming Language', https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/the-stack-and-the-heap.html.
- [2] 'The Rust Programming Language - What is Ownership?', <https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html>.
- [3] 'The Rust Programming Language - References and Borrowing', <https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html>.
- [4] C. Perkins, 'Advanced Systems Programming H/M (2024-2025)', <https://cspcrkins.org/teaching/2024-2025/adv-systems-programming/lecture03/>.
- [5] 'The Rust Programming Language - Smart Pointers', <https://doc.rust-lang.org/book/ch15-00-smart-pointers.html>.
- [6] 'The Rust Programming Language - Using Box<T> to Point to Data on the Heap', <https://doc.rust-lang.org/book/ch15-01-box.html>.
- [7] 'The Rust Programming Language - Rc<T>, the Reference Counted Smart Pointer', <https://doc.rust-lang.org/book/ch15-04-rc.html>.
- [8] W3Schools, 'C Memory Management - W3Schools', https://www.w3schools.com/c/c_memory_management.php.
- [9] MSRC, 'A proactive approach to more secure code', <https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/>.
- [10] W. Bugden and A. Alahmar, 'Rust: The Programming Language for Safety and Performance', <https://arxiv.org/pdf/2206.05503>.
- [11] Y. Zhan, Y. Zhang, G. Portokalidis, and J. Xu, 'Towards Understanding the Runtime Performance of Rust', <https://dl.acm.org/doi/pdf/10.1145/3551349.3559494>.
- [12] C. Matzinger, 'Hands On Data Structures and Algorithms with Rust - Chapter 4', <https://github.com/PacktPublishing/Hands-On-Data-Structures-and-Algorithms-with-Rust/tree/master>.
- [13] 'Introduction to Doubly Linked Lists in Java - GeeksforGeeks', <https://www.geeksforgeeks.org/introduction-to-doubly-linked-lists-in-java/>.
- [14] M. Kero and S. Aittamaa, 'Scheduling garbage collection in real-time systems in Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/- Software Codesign and System Synthesis, page 51-60', 2010.