# Advanced Systems Programming - Exercise 1: Memory Management

David Wood (2198230W)

February 2020

## Contents

**1 Introduction** **1**

**2 Rust and Region-based Memory Management** **1**

**3 In contrast with Manual Memory Management** **2**
3.1 Programmer effort . . . . . 2
3.2 Performance . . . . . 2
3.3 Safety . . . . . 3
3.4 Flexibility . . . . . 3

**4 In contrast with Garbage Collection** **4**
4.1 Advantages and disadvantages . . . . . 4
4.2 Ergonomics of ownership and borrowing . . . . 4

**5 Conclusion** **5**

## 1 Introduction

Rust [4] is a multi-paradigm systems programming language, designed to be syntactically similar to C++ but with an explicit goal of providing memory safety and safe concurrency while maintaining high performance. Development was started by Mozilla Research in 2010, with the goal of being used in the development of a parallel browser engine, Servo, and being self-hosting.

Rust achieves its goal of providing memory safety through region-based memory management, in Section 2, Rust's approach to memory management is described in detail.

Region-based memory management techniques enable Rust to be a viable alternative to traditional systems programming languages which require manual memory management. In Section 3, region-based memory management in Rust is contrasted with C's manual memory management, discussing the effect of each on programmer effort, efficiency, safety and flexibility.

Many popular high-level programming languages are garbage collected, in Section 4, the advantages and disadvantages of region-based memory management are compared to those of garbage collection, and examples are presented of where region-based memory management makes certain programs easier or more challenging to write.

## 2 Rust and Region-based Memory Management

Rust's approach to memory management is its defining feature. Region-based memory management allows Rust to be both memory-safe and efficient whilst avoiding the performance implications associated with garbage collection.

Rust's key idea is the concept of ownership. All data in Rust has a variable which is its owner. There can only be one owner for data at any given time and when the owner goes out of scope, the data is dropped. Variables are dropped in reverse order of declaration.

By introducing the concept of ownership, Rust's compiler is able to track the lifetime of data and ensure that data is not referenced after deallocation. Ownership of data can change throughout of the duration of a program's execution, as data is passed to and returned from functions.

It isn't always desirable for the ownership of data to change. In traditional systems programming languages, pointers provide a mechanism to refer to other data. Pointers are variables which contain an address to memory. References are the most common pointer type used in Rust. References "borrow" the data that they point to (ownership of the data does not change) and don't have any overhead. Rust defines two kinds of reference, shared references (`&`) and mutable references (`&mut`).

Rust enforces that references cannot live longer than the data they refer to and this allows them to always be safe to use. It is this property that guarantees that use-after-free bugs and dangling references are statically impossible.

Data behind references can only be mutated through a mutable reference. In order to take a mutable reference to some data, there must be no other live references to that data. This restriction enables Rust to prevent iterator invalidation and race conditions.

References are always dropped when they go out of scope but might be considered live for a shorter duration. Before Rust 1.31, a reference was considered "live" until it went out of scope. It was thought that this would be intuitive as programmers were familiar with the concept of scope.

However, in Rust 1.31, non-lexical lifetimes were stabilised, this language feature meant that a reference would now be considered "live" only until its last use. NLL made lifetimes more intuitive for newcomers to the language, enabling the code in Listing 1 to compile where it previously did not.

All local variables in Rust are allocated on the stack. Data can be stored on the heap through use of the smart pointer type `std::boxed::Box`. `let y = Box::new(x);` allocates on the heap and stores `x` into it, `y` is still a stack variable which contains a pointer to the heap.

Smart pointer types are data structures which act like pointers. While references only borrow data, smart pointers generally own the data they point to. Smart pointer types often implement the `std::ops::Drop` trait, which allows them to implement custom destructor logic which is run when a value of the type is dropped. The `Box` type implements `Drop` to deallocate the

```rust
fn main() {
  let mut data = vec![1, 2, 3];
  let x = &data[0];
  println!("{}", x); // NLL: `x` live until here

  // Before NLL, `x` would be live until the end
  // of the block. `Vec::push` takes `data` by
  // mutable reference, which is not possible
  // when the shared reference, `x`, to `data`
  // is live.
  data.push(4);
}
```

Listing 1: Lifetimes and NLL

heap data that it points to.

Rust provides a variety of other smart pointer types in the standard library. For example, `std::rc::Rc` is a reference-counted smart pointer which stores data in the heap. When cloned, `Rc` produces a new pointer to the same heap allocation. When the last `Rc` pointer to an allocation is dropped, then the data is deallocated. `std::sync::Arc` is also a reference-counted smart pointer but uses atomic operations, making it safe to use across threads.

`std::cell::RefCell` is a smart-pointer which owns data and has interior mutability. Interior mutability allows mutation of data even shared references to that data exist. `RefCell` enforces Rust's borrowing rules at runtime and can be used to create safe abstractions that wouldn't normally be possible with Rust's borrowing rules.

There exists another type of pointer in Rust (besides references and smart pointers): raw pointers. Raw pointers are used in unsafe Rust. Unsafe Rust is written in blocks or functions annotated as `unsafe` and can use features of the language that the compiler cannot check for memory safety.

Raw pointers can ignore the language's borrowing rules, aren't guaranteed to point to valid memory, can be `null` and don't implement any automatic cleanup. Raw pointers can exist and be created outwith unsafe code, but dereferencing raw pointers can only happen in unsafe code. Unsafe Rust exists to enable interop with hardware and other languages.

# 3 In contrast with Manual Memory Management

Traditional systems programming languages require manual memory management. Before automatic memory management techniques became practical, manual memory management was the only way to guarantee efficient runtime behaviour, an essential requirement for systems programming. Manual memory management requires that the programmer decide when memory is allocated and deallocated.

In C, there is no concept of ownership within the language and it is the programmers responsibility to ensure that no pointers refer to memory (on the stack or heap) that has been deallocated. Pointers in C are most analogous to Rust's raw pointers, there are no language guarantees that a pointer refers to valid memory, isn't `NULL` or aliased.

Heap allocation occurs through use of the `malloc` function, provided by the C standard library. `malloc` takes a size and returns a `void` pointer to allocated memory of that size. It is the programmer's responsibility to cast the returned pointer to the appropriate type, check that allocation succeeded (that the returned pointer is not `NULL`) and ensure that only memory within the allocated range is accessed/written.

Deallocation is also the responsibility of the programmer, using `free`, who must ensure that no memory is used once deallocated.

## 3.1 Programmer effort

Rust is often criticised for its learning curve. Region-based memory management is newer than other approaches to memory management and thus unfamiliar to many programmers.

It should then be no surprise that foremost amongst the stumbling blocks cited by beginners to the language is "fighting the borrow checker" - learning Rust's borrowing rules and region-based memory management.

This is a entirely valid criticism, and whilst the Rust project has invested significant energy into improving diagnostics and writing excellent documentation, it is an inherent property of Rust's increased compile-time scrutiny that the language will be more challenging to learn.

However, many users of the language find that the increased compile-time scrutiny results in an elimination of the time previously spent resolving memory safety bugs at runtime. Rust forces memory safety bugs to be "debugged at compile-time" and when viewed through that lens, could actually result in less time spent debugging overall.

## 3.2 Performance

With respect to performance and runtime efficiency, Rust has the potential to be as fast (if not faster) than C.

Region-based memory management, like manual memory management, but unlike garbage collection, happens entirely at compilation time - there is no inherent overhead as a result of using region-based memory management (such as having a runtime).

Further, region-based memory management techniques can generate code which allocates and deallocates memory as efficiently as is possible with manual memory management. Therefore, performance of code written in Rust is not limited by Rust's approach to memory management when compared to C.

Due to Rust's borrowing rules, the language can provide stronger guarantees about aliasing - when multiple pointers may refer to the same memory - than C. These guarantees can be leveraged in compiler optimisations to produce faster code than would be possible with C.

For example, consider the function `compute`, shown written in Rust in Listing 2 and written in C in Listing 3.

In Listing 4, `compute` is optimised to cache the result of dereferencing `input` (a memory access) and avoid computation in the first branch (when `input > 10`) by exiting early with the value `2`. This optimisation is only possible due to Rust's aliasing guarantees.

In the equivalent C function, `input` could refer to the same memory as `output` and both conditionals could be entered (likely unanticipated and a bug). However, in Rust, as `output`

```rust
fn compute(input: &u32, output: &mut u32) {
    if *input > 10 {
        *output = 1;
    }
    if *input > 5 {
        *output *= 2;
    }
}
```

Listing 2: Aliasing (Rust)

```c
void compute(int* input, int* output) {
    if (*input > 10) {
        *output = 1;
    }
    if (*input > 5) {
        *output *= 2;
    }
}
```

Listing 3: Aliasing (C)

is a mutable reference, it is not possible for a shared reference (`input`) to exist which refers to the same memory.

```rust
fn compute(input: &u32, output: &mut u32) {
    let cached_input = *input;
    // Optimisation: keep `*input` in a register.
    if cached_input > 10 {
        // Optimisation: `x > 10` implies `x > 5`,
        // so double and exit immediately.
        *output = 2;
    } else if cached_input > 5 {
        *output *= 2;
    }
}
```

Listing 4: Aliasing (Optimised)

There are no other design decisions (unrelated to memory management) in the Rust language which would prohibit Rust from equalling or surpassing C's performance.

### 3.3 Safety

According to Microsoft's Security Response Center, 70% of the vulnerabilities that Microsoft assign a CVE each year are memory safety issues [2].

Memory safety bugs result in security vulnerabilities. Leading technology companies hire teams dedicated to finding vulnerabilities and invest heavily in creating and deploying static analysis tools to find memory safety bugs in their software before it is released and yet still suffer from vulnerabilities as a result of memory unsafety. There is an abundance of evidence that it is not possible to write safe software in programming languages, like C, with a fundamentally unsafe memory model.

Region-based memory management in Rust makes it a compile-time error to have a dangling pointer (Listing 5) or use-after-free (Listing 6) - the equivalent C for both examples would compile and fail at runtime.

```rust
fn foo(data: u32) -> &str {
  let data_as_str = format!("{}", data);
  &data_as_str
}
```

Listing 5: Dangling pointer

```rust
fn main() {
  let mut data = vec![1, 2, 3];
  std::mem::drop(data);
  println!("{}", data);
}
```

Listing 6: Use after free

Consider the example shown in Listing 7. The equivalent function for `foo` in Rust would fail to compile because it is ambiguous which lifetime the reference being returned should have - does the return value live as long as `x` or `y`? It is easy to miss that depending on the value of `x`, `bar` can return a pointer to a stack variable.

```c
int* foo(int* x, int* y) {
    if (*x > 3) {
        return y;
    } else {
        return x;
    }
}

int* bar(int* x) {
    int y = 5;
    return foo(x, &y);
}
```

Listing 7: Lifetime ambiguity

It has been argued that Rust's memory safety guarantees are rendered moot by the need for unsafe Rust in order to interop with hardware and other languages. However, this is still an improvement over C! To successfully audit code for potential memory safety vulnerabilities in C, an auditor would need to consider every line. This isn't true of Rust, where only unsafe code would need audited. As the vast majority of Rust code does not require or use unsafe, this makes manual verification of safety invariants significantly more feasible.

### 3.4 Flexibility

Rust's borrowing and ownership inherently limit the domain of valid code that can be written. Unfortunately, Rust's rules don't just prohibit writing memory safety violations. When compared to C, Rust is strictly less flexible. Through use of `unsafe` and smart pointer types from the standard library, it is often possible to achieve a degree of flexibility closer to what is possible in C.

# 4 In contrast with Garbage Collection

In the last two decades, almost every high-level language that has grown in popularity has used garbage collection techniques to manage memory. Garbage collected languages are popular because they allow the programmer to focus on implementing their application without worrying about the details of memory management.

## 4.1 Advantages and disadvantages

By 1960, two fundamental approaches to garbage collection had been developed - tracing and reference counting - and both have seen a multitude of refinements and enhancements in the years since.

Tracing garbage collectors [6] determine which objects are reachable from a set of "root" objects and consider the remaining objects as garbage, collecting them. Reference counting collectors [3] keep a count of references to each object, and when the reference count drops to zero, that object is collected.

Tracing garbage collectors scan the entire heap at once, incurring long pauses during these collection cycles. In contrast, reference counting is inherently incremental - only updating reference counts on pointer writes. This property makes tracing garbage collectors less suitable for real-time systems.

However, reference counting garbage collectors incur a cost on mutation as reference counts are updated. Tracing garbage collectors demonstrate no such penalty on mutation and have higher throughput as a result. Reference counting garbage collectors are unable to collect cycles, so often require backup tracing collection or trial deletion algorithms.

As discussed by Bacon et al [1], tracing and reference counted garbage collection techniques can be considered algorithmic duels and high performance garbage collectors are actually hybrids of tracing and reference counting techniques which have surprisingly similar performance characteristics.

Both tracing and reference counted garbage collectors incur a runtime performance penalty that region-based memory management does not. Region-based memory management in Rust generates code which performs allocations and deallocations for all non-static data, therefore there is no need for a language runtime to perform garbage collection, and thus there are no collection cycles or overhead on pointer writes.

Recently, Discord announced that they switched from Go to Rust for one of their production services [5]. In their article, they cited the performance of Go's garbage collection.

In some domains, language runtimes required for garbage collection can introduce latency which is unacceptable, or require an operating system (which isn't possible on embedded micro-controllers).

Garbage collectors implemented in popular programming languages are very advanced and tuned to have as little performance impact as possible. Performance of garbage collectors in mainstream languages is often acceptable for a majority of applications and thus the increased flexibility enabled by garbage collectors in these languages can make them an appropriate and appealing choice over Rust for many projects.

```
help: insert indirection (e.g., a Box, Rc, or &) at some
point to make `List` representable
```

Figure 1: Compiler error from Listing 8

## 4.2 Ergonomics of ownership and borrowing

Garbage collected languages are typically cited as being more productive and ergonomic than Rust due to the constraints imposed by ownership and borrowing.

It is common for programmers to implement simple data structures and algorithms that they are familiar with when learning new programming languages. Unfortunately for Rust, one of these data structures is the linked list.

```
pub enum List<T> {
    Empty,
    Elem(T, List),
}
```

Listing 8: Naive linked list

Garbage collected languages make implementation of linked lists trivial - when a node is no longer referenced, then it will be garbage collected!

Listing 8 shows a naive implementation of a linked list in Rust, it doesn't compile. In Figure 1, the help message from the compiler error is shown. Linked lists are such a common stumbling block that if a user proceeds to open the documentation for `Box`, then they will quickly find the code shown in Listing 9 - a working linked list!

```
#[derive(Debug)]
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

fn main() {
    let list: List<i32> = List::Cons(1,
        Box::new(List::Cons(2,
            Box::new(List::Nil))));
    println!("{:?}", list);
}
```

Listing 9: Linked list from documentation

In addition to the documentation, there's an entire book dedicated to it - "Learning Rust With Entirely Too Many Linked Lists" [7].

Linked lists are challenging in Rust because recursive types aren't supported without indirection, such as heap allocation (with `Box`). Most garbage collected languages don't have this problem by storing *everything* on the heap (at the cost of performance). Doubly-linked lists in Rust get even more tricky because of the language's ownership rules demanding that every object have a single owner.

However, Rust's safety guarantees can also make writing some code easier! For example, writing code that uses iterators is safer and easier in Rust than in other languages. Due

to Rust's ownership and borrowing rules, Listing 10 doesn't compile! `Vec::push` takes `xs` by mutable reference, which is impossible as a shared reference into `xs` is live with the call to `Vec::iter`.

```
fn main() {
    let mut xs: Vec<u32> = random_vector();
    for x in xs.iter() {
        xs.push(x + 2);
    }
}
```

Listing 10: Iterator invalidation (Rust)

In contrast, Listing 11 demonstrates that Java, a garbage collected language, will throw a runtime exception if a collection is modified during iteration.

```
import java.util.*;

class Main {
  public static void main(String[] args) {
    List<String> list = new ArrayList<String>();
    list.add("A");
    list.add("B");

    Iterator<String> it = list.iterator();
    String s = it.next();
    list.add("C");
    s = it.next();
    // ^ ConcurrentModificationException
  }
}
```

Listing 11: Iterator invalidation (Java)

Worse yet, Listing 12 presents an example of iterator invalidation in C++, a language with manual memory management, which produces a incorrect results!

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  vector <int> v{1, 5, 10, 15, 20};

  for (auto it = v.begin(); it != v.end(); it++)
    if ((*it) == 5)
      v.push_back(-1);

  for (auto it = v.begin(); it != v.end(); it++)
    cout << (*it) << " "; // 1 5 10 15 20 -1 -1

  return 0;
}
```

Listing 12: Iterator invalidation (C++)

In general, Rust's ownership and borrowing make it easier to write programs with data structures which have clear trees of ownership and harder otherwise. In languages with manually managed memory or garbage collected memory, it may have been more natural to have a single, more complicated data structure with graphs of multiple ownership.

# 5 Conclusion

Rust successfully employs region-based memory management to create a practical systems programming language which maintains high performance while providing memory safety and safe concurrency.

It is unclear whether applications programming would benefit more from Rust's safety guarantees than the productivity of today's high-level, garbage collected programming languages, as described in Section 4.

However, through Rust's modern ownership and borrowing rules, described in Section 2, writing correct and safe systems software is not only possible but practical and, as shown in Section 3, a clear, demonstrable improvement over traditional systems programming languages with unsafe memory models. It is therefore imperative that future core systems and infrastructure are written in languages like Rust.

# References


[1] D. F. Bacon, P. Cheng, and V. T. Rajan. A unified theory of garbage collection. *SIGPLAN Not.*, 39(10):50–68, Oct. 2004.

[2] M. S. R. Center. A proactive approach to more secure code, 2019 (accessed February 9, 2020). https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/.

[3] G. E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, Dec. 1960.

[4] T. R. P. Developers. Rust, 2020 (accessed February 8, 2020). https://www.rust-lang.org.

[5] Discord. Why discord is switching from go to rust, 2020 (accessed February 9, 2020). https://blog.discordapp.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f.

[6] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, Apr. 1960.

[7] R. Unofficial. Learning rust with entirely too many linked lists, 2020 (accessed February 9, 2020). https://rust-unofficial.github.io/too-many-lists/.