International Journal of Computer Applications (0975 - 8887)
Volume 183 - No.22, August 2021

# Memory Management via Ownership Concept Rust and Swift: Experimental Study

Elaf Alhazmi
Computer Science Department
Umm Al-Qura University
Al-leith, Saudi Arabia

Abdulwahab Aljubairy
Computer Science Department
Umm Al-Qura University
Al-leith, Saudi Arabia

Ahoud Alhazmi
Computer Science Department
Umm Al-Qura University
Al-leith, Saudi Arabia

## ABSTRACT

One of the most important features to help facilitate reliable design in a programming language is memory management design. There are two wide-spread approaches: manual and automatic memory management, known as garbage collection (GC). Recently, a third approach which is ownership design has been fully adapted in new modern programming languages such as Rust and Swift. Rust uses ownership to eliminate high degree memory problems such as memory leak, dangling pointer, and use after free. Rust follows deterministic syntax-driven memory management depending on static ownership rules implemented and enforced by the rustc compiler. Swift also implements ownership concept in automatic reference counting (ARC). Though the ownership concept is adapted in Swift, it is not a memory-safe language because of the possibility of strong reference cycles. In this paper, we will illustrate the fundamental of ownership and the consequences of memory safety guarantees and issues related to Rust and Swift. We also conducted an experiment to compare the elapsed time binary tree allocation and deallocation in five programming languages C, C++, Java, Swift and Rust.

## General Terms

Memory Management, Programming Languages

## Keywords

Ownership, Manual Memory Management, Garbage Collection

## 1. INTRODUCTION

There are several programming languages designed with different capabilities and features that make programmers focusing more on reliability of coding [1, 2, 8, 24, 26]. Some programmers select a programming language that supports automatic management of memory resources instead of taking the risk of manual management but others prefer to use the conventional programming languages that depend on manual management of memory resources such as C and C++ that have been used for decades.

Memory management - the allocation and deallocation of dynamic memory during the execution of a program - is one of the most important aspects of programming languages. The most serious memory issues such as memory leak, use after free are common on manual memory management even though high performance and low-level control are satisfied. On the other hand, automatic memory management known as garbage collection eliminates a number of memory issues but still has visible overhead at run-time. The third approach of memory management known as ownership supported partially in a conventional programming language such as C++ by using std :: unique_ptr and fully in new modern programming languages such as Rust and Swift. Ownership in modern programming languages has significant points in overcoming a number of memory issues deterministically.

The two modern programming languages that support ownership memory management approach are Rust and Swift. The former - a system programming language - is a strongly statically typed language where types are checked at compile time [4]. Rust compiler - rustc - eliminates a number of memory issues and satisfies deterministic memory management without garbage collection. The compiler has strict rules of checking related to the ownership approach. Rust eliminates data races and has low-level control of resources comparable to C and C++ but it has a steep learning curve and new programmers will have to adopt new cognitive programming skills to get successful compiled source codes by strongly taking into consideration the ownership strict rules. The latter is a compiled general-purpose programming language that supports the concept of ownership to manage its dynamic reference types by using Automatic Reference Counting - ARC. Swift developers emphasize that Swift is enjoyable, expressive, and safe language but novice programmers need to be aware of strong reference cycle in order to avoid memory leak which is the main issue of ARC [15].

This paper aims to focus on the fundamentals and the efficiency of the ownership concept in modern programming languages Rust and Swift. We conduct an experiment to compare and evaluate the performance of Rust and Swift with other conventional programming languages: C, C++ and Java. We select a task requiring a frequent allocation and deallocation such as binary tree data structure to measure the elapsed time of dynamic allocation and deallocation. The performance of binary tree shows a remarkable significant comparison between conventional and modern programming languages: *(i)* allocation in Rust is noticeably

faster than Swift, but Swift is faster than Rust in deallocation even with a large number of nodes, *(ii)* Java has a better performance with a restricted number of nodes, whereas Swift is capable of working with a large number of nodes even though both languages incur runtime costs, and *(iii)* modern programming languages compare favorably in a large number of nodes to the conventional programming languages C, C++ and Java. The remaining sections in this paper are organized with the following orders: section 2 discusses some of the related works, section 3 explains the fundamentals and memory safety of ownership approach in Rust and Swift, section 4 presents the conducted experiment and section 5 concludes the paper.

## 2. RELATED WORKS

There are a number of studies focused on comparing programming languages with different criteria [24, 25]. In programming languages, memory management has two common fundamental approaches: manual and automatic memory management. The first approach which is manual or explicit memory management requires programmers to allocate and free memory manually such as a programmer who uses the C language. The second approach is automatic memory management which emerged in order to overcome the issues of the first type approach [28]. In automatic memory management, a programmer who uses programming languages such as Java, C#, and Caml is not obliged to manage the memory manually because these languages offer the capability of managing the memory automatically via garbage collection.

Undoubtedly, the benefits of automatic memory management are indisputable but the costs are high in runtime and space efficiency [24, 29]. The authors of [23] emphasized that object-oriented programming significantly increases dynamic memory usage, thereby causing overhead in terms of memory, performance, and power. Due to various reasons, there are only a handful of documented bad experiences of object-oriented technology.

Nowadays, there are several new programming languages designed to improve the weaknesses of the previous programming languages. In this research, two recent languages Rust and Swift have been studied. The former was developed by Mozilla [22] and it was designed to overcome two obstacles in the previous languages to offer programmers both high-level safety and low-level control [19]. Swift was developed and introduced at Apple Worldwide Developers Conference (WWDC) in 2014. [11]. In comparison to Rust, this language is easier and more flexible to use for new programmers. In particular, Swift is designed to be faster and safer than Objective-C [30] that inspired it.

Comparing Rust with other languages, the research [20] discusses the benefits of Rust. They found Rust promising both better performance and safety in comparison to C-like languages. Also, their results show Rust's safety features do not create significant barriers to implementing a high-performance collector. In terms of Rust's safety, the authors of [18] demonstrated how Rust utilizes the ownership-based type system. However, this core type system uses libraries that internally use unsafe features. Furthermore, the authors of [3] investigate the strengths of Rust in comparison with conventional languages such as C. They found Rust has the ability to implement powerful security and reliability mechanisms like Software fault isolation (SFI), Information flow control (IFC), and Automatic checkpointing more efficiently than any conventional language. However, that is not a sufficient reason to enforce using Rust and abandon other languages such as C. The efficiency and performance of the current languages like C encourage the use of these languages[5].

Regarding conventional languages weaknesses like lack of memory safety and integrated support for concurrency, these languages force developers to deal with memory and thread safety[6]. There are attempts for solving these problems by designing some methods in languages. For example, the authors of [7] present a static type system called Real-Time Specification for Java (RTSJ) in order to ensure memory safety. Their work was the first work that combined the benefits of region types and ownership types. However, in this research, we do not focus on studying one language and how researchers try to solve its weakness.

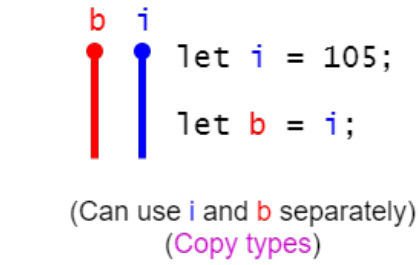
## 3. OWNERSHIP: RUST AND SWIFT

**Rust** - system programming language - is a strong type compiled language. It is designed to eliminate a number of memory issues and problems such as dangling pointers, buffer overflows, null pointers, segmentation faults, and data races statically at compile time. Rust not only has a high safety because of its strong type system but also has a low level control of resources comparable to C and C++. Rust satisfies deterministic memory management without garbage collection by using *rustc* compiler that has strict rules of checking the ownership. The compiler affects the following aspects: ownership, mutability, borrowing, lifetime and scope. The strict rules of ownership in rustc enforce the following: *(i)* each variable in Rust has only one owner at a time, and *(ii)* a variable will be deallocated when the owner goes out of the scope. Even though the concept of ownership is not unique in programming languages, the feature when a variable lifetime end goes out of scope is unique to Rust. However; Rust has steep learning cost as reported in Rust documentation [10]. A community survey in 2017 revealed that 25% of the people who tried Rust and dropped it felt the language was "too intimidating"; in other words, it is too hard and complicated to learn the language [31].

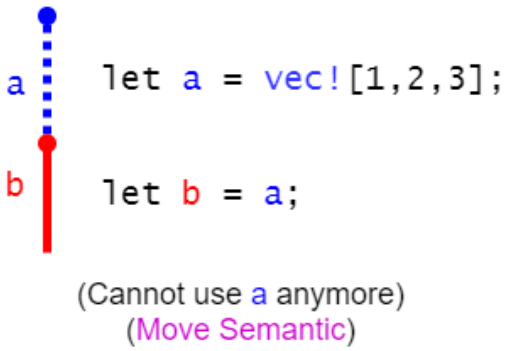
**Swift** - general-purpose programming language - is a strong type compiled language. It supports the concept of ownership to manage its dynamic reference types by using automatic reference counting-ARC. In Swift, programmers do not have to think about retain and release operations that are required in manual retain release -MRR- supported in Objective-C [12, 13]. ARC compiler will manage the lifetime of objects by inserting appropriate memory management method calls to perform object destruction automatically once the object is no more needed. When an object is created and assigned to a property, constant, or variable, the integer associated with the object will be increased by the number of properties, constants, or variables that hold a reference to the object. This integer associated with the object is called the reference count and the object will be reclaimed when its reference count reaches zero [21]. As reported by Apple developers, Swift is a safe, fast, and interactive programming language [15] but writing a safe code mostly means avoiding memory leaks in Swift.

### 3.1 Rust Fundamentals

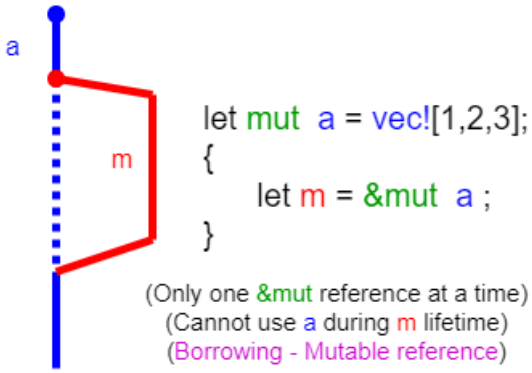
To start programming in Rust, new programmers need to know the Rust fundamentals such as ownership and borrowing. The concept of ownership is simple, but it affects all aspects of Rust.



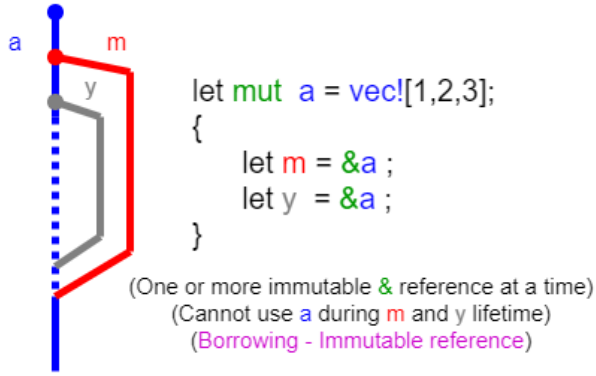

(a) Ownership: copy types




(b) Ownership: move semantic




(c) Borrowing - Mutable reference




(d) Borrowing - Immutable reference

Fig. 1: Borrowing - Mutable and Immutable references [27]

**Ownership & Scope**: rustc compiler enforces the following ownership rules: *(i)* each variable binding has a single owner at any time, and *(ii)* a variable is dropped when the owner goes out of scope. Rust also uses static, or equivalently, lexical scoping which defines a range of program source code where an item such as a variable is valid.

**Borrowing & Borrow Checker**: passing a variable binding in Rust needs to be borrowed with two special types of references that depends on read-write lock pattern: *(i)* immutable reference type $\&T$ for only reading data and *(ii)* mutable reference type $\&mutT$ for writing data as shown in Figure (1d), and Figure (1c) respectively.

In terms of borrow checker, borrowing rules are strictly enforced at compile time, and allow only *(i)* one or more immutable references to a resource, or *(ii)* exactly one mutable reference at a time with taking into consideration that borrow mutable reference requires the owner to be mutable.

**Lifetime**: a lifetime is a construct the borrow checker of rustc uses to ensure that all borrows are valid. A variable's lifetime begins when it is created and ends when it is destroyed. The borrow checker uses symbolic names representing lifetimes to determine how long a particular reference should be valid.

**Smart Pointers**: Rust supports several smart pointers with different implementations and guarantees. The following are the pointers that used in the conducted experiment:

—$Box < T >$ is the simplest smart pointer that uniquely owns a piece of heap-allocated data that will be deallocated when it goes out of scope.

—$Rc < T >$ is a single-threaded reference-counting pointer that supports multiple owners. The reference-counting pointer will be deallocated when the reference count reaches zero, and its internal data is immutable. If a cycle of references is created by using RefCell<T>, the memory will be leaked [9].

—$Weak < T >$ is similar to reference-counting pointer, it is a non-owning and non-borrowed reference counted smart pointer and it is useful for cyclic data structures.

—$RefCell < T >$ are shareable mutable containers. It provides a mutable memory location with dynamically checked borrow rules by using borrow() and borrow_mut() functions; however, it does not move data in and out of the cell.

### 3.2 Swift Fundamentals

To program safe code in Swift, new programmers need to know how ownership memory management approach is adapted in Swift by understanding the following fundamentals.

**Optional Types**: is an enumeration with two cases: nil or value. $Optional.none$ is equivalent to the nil literal, and $Optional.some(T)$ stores a T value. Swift supports the operation of optional chaining symbolized by (?) which accepts nil value. To forcefully access optional types, the operation (!) is required. Optional values will be used extensively when lifetime qualifiers are needed.

**Classes**: provide reference types that required to be handled by ARC. Even though there are significant similarities between classes and structures but structures are value types. Swift provides two special methods: *init()* and *deinit()*. The former does not return value and uses for initial value of properties, and the latter is provided to execute a custom code before an instance destruction[16, 17].

**Strong Reference**: is a reference between a class instance to a property, constant, or variable. In fact, a strong reference cycle may occur when two class instances hold a strong reference to each other, such that each instance keeps the other alive. To avoid strong reference cycle that causes memory leak, a lifetime qualifier such as weak or unowned references must be used.

**Automatic Reference Counting - ARC**: compiler will manage the lifetime of objects by inserting appropriate memory management method calls to perform object destruction automatically once the object is no more needed. Reference counting is the mechanism that has been used in Objective-C and is so supported by Apple, and ARC is the official standard adopted in Swift. Ownership and lifetime concepts are strongly required to manage this automatic handling. In fact, when an object is owned by a property, constant, or variable, that object will be alive as long as the object is needed. When an object is created and assigned to a property, constant, or variable, the integer that is associated with the object will be increased by the number of properties, constants, or variables that hold a reference to the object. This integer associated with the object is called the reference count and the object will be reclaimed when its reference count reaches zero [21].

**Lifetime Qualifiers**: will be used to avoid memory leak which is the main memory issue comes from strong reference cycle. There are three scenarios that can summarize the solution of strong reference cycle in Swift [14]. The first lifetime qualifier is - *Weak* - lifetime. It can be used in a relationship that describes two independent lifetimes which could be described as association relationship. The weak reference will not keep a strong reference on the class instance it refers and it will not stop ARC from reclaiming the reference. ARC automatically sets a weak reference to nil when the instance that it refers to is deallocated, and weak references are always declared as variables of optional type, rather than constants in order to change the value to nil at runtime. The second qualifier is - *Unowned* - lifetime. It can be used in a relationship that describes two dependent lifetimes as aggregation relationship. In fact, unowned reference type where should be used when another instance has the same lifetime or longer lifetime. Unowned reference is expected also to have a value during the object's lifetime because it should not be defined as an optional type; in other words, unowned reference should be used when it always refers to an instance that has not been deallocated.

It must also be noted that a runtime error will be executed when unowned reference refers to an instance that has been deallocated. The third type of relationship defines strong dependent relationship as composition relationship that describes in Swift by combining *unowned reference* and *unwrapped optional property* satisfying the guarantee that both properties have values instead of being nil. The third type of relationship defines not only how Swift developers had overcome strong reference cycle, but also how objects relationship is composed. Swift programmers need to be aware of memory management and class objects relationship. Even though Swift supports ARC to implicitly destruct objects in a deterministic way, the programmer still has the responsibility to provide a reliable design.

### 3.3 Safety Guarantees: Rust and Swift

The concept of ownership is presented in both Rust and Swift, but Rust ownership system is wider and more solid than Swift. Rust provides a challenging rustc compiler that eliminates a number of common errors at compile time. The following are basic errors will be countered particularly by new programmers in Rust:

**error[E0384]**: re-assignment of immutable variable 'x', which explicitly required to declare the state of mutability because all variable binding is immutable by default in Rust.

```
fn main() {
    let x = 2018;
    x += 1; // error
    println!("x = {}", x);
}
```

**error[E0382]**: use of moved value:'x', occurs when a dynamic data size reassigned to new binding which is known as move ownership and results one of Rust ownership violations which is aliasing.

```
fn main() {
    let x = vec![1,2,3];
    let y = x;
    println!("x[0] = {}", x[0]); // error
}
```

**error[E0597]**: 'x' does not live long enough, will be caused if a reference lives longer than the resource it refers to.

```
fn main() {
    let y : &i32;
    let x = 5;
    y = &x;
    println!("{} ", y);
}
```

**error[E0597]**: 'i' does not live long enough, will be caused if a resource points to invalid resource. The variable r will live longer than i, but after i is deallocated, r would be dangling.

```
fn main() {
    let r;
    {
```

```
        let i = 1;
        // i does not live long enough
        r = &i;
    }
        // i dropped while still borrowed
    println!("{}", r);
}
```

Rust has fixed and dynamic size types. As shown in Figure (1a) and Figure (1b). Dynamic size binding such as vector - allocated on heap - uses move semantic in order to keep only one owner as a reference to the dynamic memory resource. In Rust, there is a difference between move semantic approach which is used with dynamic size binding, and copy type approach which is used with fixed size binding. Rust also follows strict rules in borrowing known as borrow checker which depends on read-write lock pattern as illustrated in Figure (1d) and Figure (1c). A majority of memory management issues are prevented by rustc compiler statically at compile time but strong reference cycle may be caused when $RefCell < T >$ used with $RC < T >$ as shown in Appendix A.1 and illustrated in Figure (2).

To create strong reference cycle in Rust, novice programmers need to know how to use smart pointers and how to mutate the content with $RefCell < T >$ that put novice programmers in challenge to learn Rust. Unlike Rust, Swift novice programmers will create strong reference cycle easily with less effort of notice as shown in Appendix A.2. Swift developers claim that Swift is a safe language but Swift has memory issues and programmers are responsible to avoid them. Swift does not support all memory safety guarantees even though Swift is a strongly typed language. In fact, dangling pointers, memory leak, use after free are serious memory issues that still exist in Swift as strong reference cycle will be caused easily by novice programmers without using the appropriate lifetime qualifiers. Besides ownership and lifetime qualifiers, Swift needs programmers to be precise about optional types. Swift supports mutability in a naive, simple way providing constants, variable or optional types. Even though mutability in Swift will contribute to declaring objects lifetimes, it does not prevent undesirable behavior. Swift has rules to combine these types with lifetime qualifiers; for example, an unowned reference needs to have a valid value and declaring an unowned reference as optional type will not be accepted by the compiler. In addition, the mutability of types by using let and var is still designed in a naive way.

## 4. THE EXPERIMENT

In this section, we describe how to use Rust and Swift fundamentals to build an experiment that measures the performance of dynamic allocation and deallocation by selecting a task requiring frequent allocation and deallocation. In the conducted experiment, unsafe Rust pointers have not been used and the risk of strong reference cycle has been avoided. The experiment will be evaluated with mature programming languages such as C, C++ and Java. Two approaches were selected for measuring the performance of the algorithms using a simple data structure Binary Tree. The first approach was insertion – number of – nodes. The second approach focused on measuring the binary search node replacement by recreating an existing node in order to satisfy deallocation purposes in all five selected languages. Two initial primary tasks-generating permutation numbers - and shuffle the numbers with the Fisher-Yates algorithm performed to prepare a random number of nodes. The purpose of the experiment was not any strict benchmarking, but an illustration of the efficiency of the respective memory management approaches.

### 4.1 Experimental Setting

In order to compare the performance of different memory management designs, three well-known programming languages: C, C++ (manual memory management), Java (automatic memory management), Rust (Ownership memory management), Swift (automatic reference counting) were selected. The following environment and platform operating system used for measurements: macOS Sierra version 10.12.6 with processor: 2.9 GHz Intel Core i5.

We used the following compilers for the selected languages: javac 1.8.0-161 for Java, Clang 802.0.42 with LLVM as back-end for C and C++, rustc 1.23.0 for Rust, and swift 3.1 for Swift. For elapsed time measurement, we used the following functions: *mach_absolute_time()* works in OSx,(C, C++, Swift).*System.nanoTime()* works in Java, and *std::time::Instant* works in Rust. All-time results are unified in milliseconds (ms) and the average time is computed of 28 runs for 10-million nodes. Based on the experiment results for 50-million and 100-million nodes, number of runs reduced to 5 attempts. In Table (3) and Table (4), only five attempts showed out of 28 attempts for measuring 10-million nodes of allocation and deallocation.

### 4.2 Experiment Implementation

The experiment[1] implemented by two algorithms insert and replace a node for heap-allocation and deallocation respectively. The measurement for allocation includes three stages: inserting 10-million random numbers then gradually increases the number of allocations to 50-million and 100-million nodes. Like allocation, deallocation is forced by replacing an existing node with a new one. We measured the deallocation with 10-million, 50-million and 100 million random numbers.

*4.2.1 Allocation.* The process of allocation in all five selected languages will be a pointer-reference- to a heap resource. Satisfying allocation on dynamic memory location in C and C++ addressed by using $malloc$ and $new$ functions respectively. On the other hand, Java and Swift allocation approach are satisfied by using object classes while $Box < T >$ smart pointer was used in Rust. with smart pointer, optional values supported by $Option < T >$ module was used because null value is not supported in Rust for memory safety reasons.

*4.2.2 Deallocation.* The process of deallocation in the selected languages will be handled with three different perspectives. In conventional programming languages: C and C++, we have to deallocate a memory resource manually by $free$ or $delete$ functions while in Java the old nodes will be collected by the garbage collection which is implicit non-deterministic memory management; in other words, when the garbage collator starts working is unknown. In Rust, $Box < T >$ smart pointer will be deallocated when it goes out of scope as well as when no owner points to a resource. Swift deallocation will be handled by ARC

[1] https://github.com/E-Alhazmi/Memory-Management-via-Ownership-Concept-Rust-and-Swift-Experimental-Study

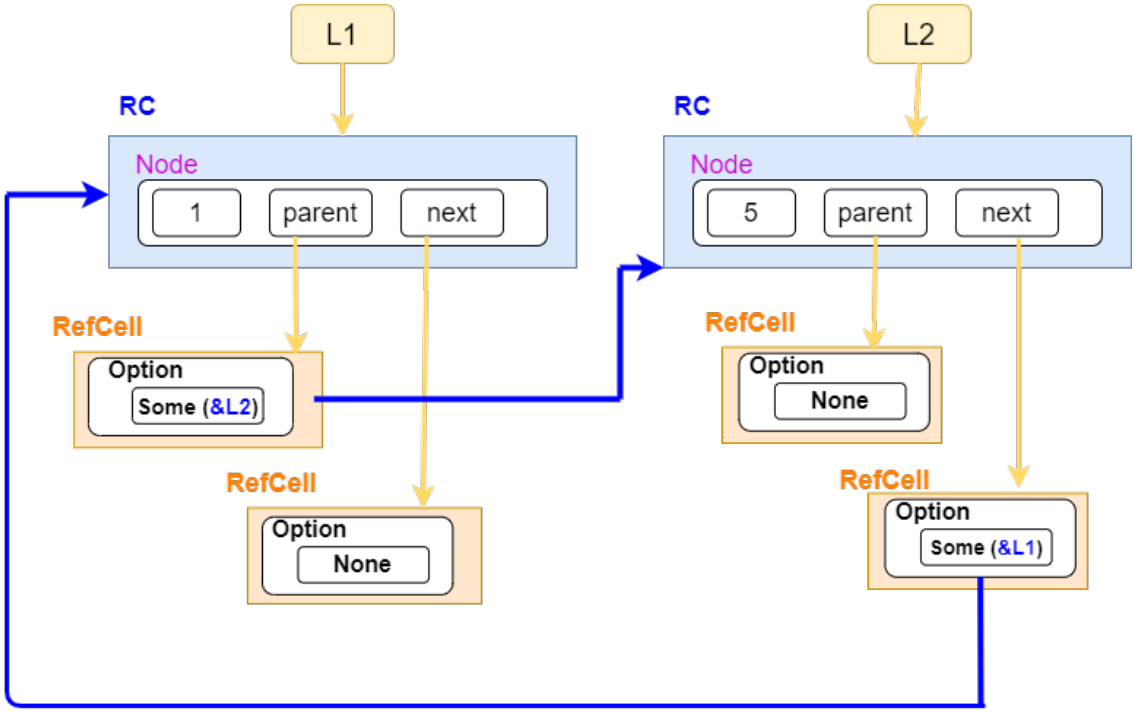



Fig. 2: Rust- Strong Reference Cycle

that a node will be deallocated when reference counter of a resource reaches zero.

### 4.3 Experimental Results

The experiment started by observing the elapsed time of inserting 10M nodes Table (1) presents the average performance of 28 runs. The number of insertions increased to 50M nodes then to 100M nodes. The performance of inserting 100M nodes showed in Table (1) has been completed successfully in Rust and Swift whereas other programming languages had significant problems in completing the task that make the attempts reduced to 5 times. In deallocation, Table (2) shows the average replacement of 10M, 50M and 100M random numbers. There was a significant delay in replacing 50M nodes in C++. In replacing 100M nodes, significant problems showed in traditional programming languages (C, C++, and Java).

Table 1. : Average elapsed time for allocation

| Language | 10M (s) | 50M (m) | 100M (m) |
|---|---|---|---|
| Swift | 114.67 | 10.97 | 23.38 |
| Rust | 25.13 | 2.73 | 6.36 |
| Java | 8.59 | 1.36 | NA |
| C | 19.93 | 16.84 | NA |
| C++ | 20.54 | 67.83 | NA |

All time unified with millisecond and converted to (s) second or (m) minutes and (NA) means not applicable.

Table 2. : Average elapsed time for deallocation

| Language | 10M (s) | 50M (m) | 100M (m) |
|---|---|---|---|
| Swift | 38.86 | 3.63 | 7.26 |
| Rust | 48.92 | 5.39 | 12.11 |
| Java | 14.79 | 3.17 | NA |
| C | 36.97 | 50.12 | NA |
| C++ | 45.14 | NA | NA |

All time unified with millisecond and converted to (s) second or (m) minutes and (NA) means not applicable.

### 4.4 Experimental Evaluation

The experiment shows significant results in both performance comparison and coding challenges. The experiment evaluation is divided into two sections: allocation and deallocation results. Conventional programming languages show a remarkable number of issues and restrictions related to memory management, while the recent programming languages – Rust and Swift – were able to manage a number of these issues, partially in Swift and fully in Rust, with the ability to handle large numbers of allocation and deallocation.

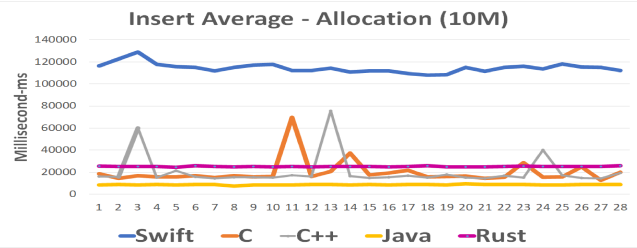
*4.4.1 Allocation.* The overhead in Swift is visible in allocation. Even though Java program has much better performance than Swift program, Java is a non-deterministic language and when the garbage collector will engage is not possible to be known. In modern programming languages, Rust is faster to accomplish the task than Swift. Figure (3a) shows the results of inserting 10M nodes along with a line graph indicating the time in milliseconds. In 50-million random number allocation, several errors and issues were observed in manual memory management. For

Table 3. : Insert- millisecond elapsed time

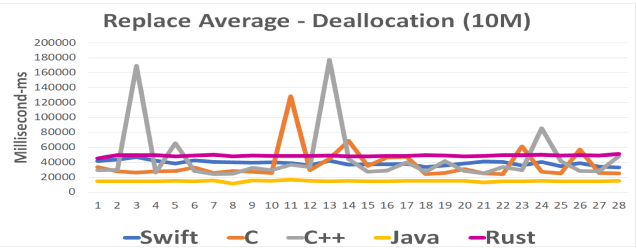
| 10-million allocation | | | | | |
|---|---|---|---|---|---|
| Language | T1 | T2 | T3 | T4 | T5 |
| Swift | 116439.16 | 122828.51 | 129004.66 | 117719.71 | 115762.35 |
| Rust | 25409.03 | 25271.72 | 25255.32 | 25192.32 | 24599.76 |
| Java | 8508.74 | 8628.33 | 8508.84 | 8703.05 | 8512.28 |
| C | 18387.34 | 14469.50 | 16754.22 | 20527.44 | 15607.92 |
| C++ | 15931.9 | 15600.1 | 59991.9 | 15031.5 | 21243.9 |
| 50-million allocation | | | | | |
| Swift | 658737.42 | 641049.96 | 634246.88 | 631413.36 | 639557.98 |
| Rust | 163998.48 | 170169.23 | 167847.49 | 167994.33 | 166003.74 |
| Java | 81814.69 | 88284.90 | 83220.19 | 81848.39 | 88617.43 |
| 100-million allocation | | | | | |
| Swift | 1383925.91 | 1383426.25 | 1447623.83 | 1443952.84 | 1355973.74 |
| Rust | 381866.52 | 379781.32 | 375887.08 | 386155.64 | 384570.86 |

Table 4. : Replace- millisecond elapsed time

| 10-million deallocation | | | | | |
|---|---|---|---|---|---|
| Language | T1 | T2 | T3 | T4 | T5 |
| Swift | 41781.96 | 43838.74 | 47114.82 | 42137.09 | 38541.27 |
| Rust | 45312.51 | 49641.21 | 49537.81 | 49791.45 | 48122.95 |
| Java | 14829.36 | 14839.01 | 14606.18 | 14595.03 | 15023.66 |
| C | 33682.84 | 27697.70 | 26497.83 | 27769.41 | 28382.32 |
| C++ | 29153.3 | 24099.2 | 24697.9 | 36855.8 | 33514.4 |
| 50-million deallocation | | | | | |
| Swift | 218031.83 | 199452.94 | 212042.12 | 205188.67 | 219376.88 |
| Rust | 323532.76 | 325573.52 | 321051.45 | 321864.08 | 324273.76 |
| Java | 190440.67 | 213775.20 | 191849.18 | 174916.81 | 208183.29 |
| 100-million deallocation | | | | | |
| Swift | 460675.31 | 420665.13 | 425775.80 | 433138.88 | 440672.59 |
| Rust | 710868.28 | 736143.10 | 723944.91 | 730938.40 | 733761.40 |

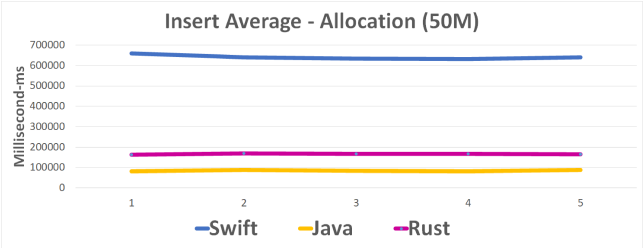



(a) Insert 10M nodes - five languages

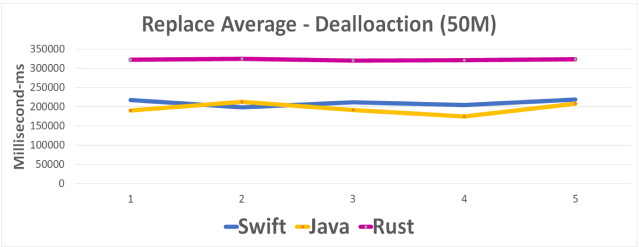



(b) Replace 10M - five languages

Fig. 3: Allocation and Deallocation 10M nodes




(a) Insert 50M nodes - three languages




(b) Replace 50M nodes - three languages

Fig. 4: Allocation and Deallocation 50M nodes

example, C language gave *"Segmentation fault:11"* run-time error approximately eight times out of ten attempts. One successful attempt shows approximately 17 minutes to insert the desired number of nodes whereas Swift and Rust need about 11 minutes and 3 minutes respectively. In fact, Rust allocation is much more significant than Swift in this part. Java still shows better performance which is approximately less than 2 minutes. On the other hand, C++ average result shows sharply slow performance in inserting 50 million about 68 minutes which is more than an hour. Figure (4a) shows the results of inserting 50M nodes in three different languages (Swift, Java and Rust) along with a line graph indicating the time in millisecond.

The influence of increasing the volume of the dynamic allocation by increasing numbers to 100-million numbers was obvious. Allocated this number of nodes became difficult to measure in C and C++. In addition, Java produced the error *"Exception in thread "main" java.lang.OutOfMemoryError: Java heap space"*. We succeeded only in the measurement for Swift and Rust. In 5 attempts, Swift performance average time allocation needed about 23 minutes, while Rust needed about 6 minutes; in other words, allocation in Rust is approximately four times faster than Swift.

*4.4.2 Deallocation.* Java shows the best performance among the other four selected languages whereas Rust has the slowest performance of deallocating as showed in Figure (3b). It should be noted that Java is a non-deterministic language and when the garbage collector will engage is not possible to be known. Swift is performing faster than Rust and close to C performance while C++ is slower than C and Swift as well as Java. Manual memory management shows a noticeable gap between minimum and maximum results whereas Swift, Rust, and Java are more stabilized.

In deallocating 50-million random numbers, C++ performance is unspecified as the programs would not terminate correctly. Indeed, C and C++ show a sharp delay in – replace algorithm – when the number of nodes is increased. C has been completed in about 50 minutes. Swift and Java show approximately the same performance about 4 minutes and 3 minutes respectively. Swift shows slightly better performance than Rust. Rust needs about 5 minutes, while Swift needs about 4 minutes to replace 50 million nodes randomly. The line graph in Figure (4b) shows the result of replacing 50-million nodes in Swift, Java, and Rust.

In 100-million random number deallocation, the conventional programming languages Java, C, and C++ did not complete the experiment for the same reasons as when inserting 100 million nodes. The average performance of 28 times of – replace algorithm completes the task with approximately 7 minutes in Swift and 12 minutes in Rust.

## 5. CONCLUSION

This paper introduces the ownership memory management approach adopted by modern programming languages and compares it with conventional approaches known as manual and automatic memory management. We summarized the fundamentals of ownership, memory safety, and issues related to Rust and Swift. We proposed an experiment to compare the elapsed time of binary tree nodes allocation and deallocation in five programming languages (C, C++, Java, Rust, and Swift). The experiment has remarkable success in handling a large number of nodes in ownership design. Rust and Swift have fewer run-time errors comparing to mature programming languages such as C, C++, and Java. The experiment not only shows the performance of the selected languages but also evaluates the ownership memory safety in Rust and Swift. Rust ownership design is more solid than Swift in terms of preventing memory issues such as a dangling pointer, memory leakage, and use after free statically at compile-time with zero or negligible run-time costs; on the other hand, Rust has steep learning cost as system programming for novice programmers. Comparing elapsed performance programs written in five different programming languages deserve to be taken seriously as a valuable contribution for comparison of the various language designs.

## 6. ACKNOWLEDGEMENTS

We would like to thank and express our gratitude to Dr. Frantisek Franek and Dr. Emil Sekerinski for their helpful discussions and guidance. We would like to thank the Saudi Arabian Cultural Bureau in Canada for their support. We would also like to acknowledge the scholarship support from Umm Al-Qura University.

## 7. REFERENCES

[1] Sultan S Al-Qahtani, Pawel Pietrzynski, Luis F Guzman, Rafik Arif, and Adrien Tevoedjre. Comparing selected criteria of programming languages java, php, c++, perl, haskell, aspectj, ruby, cobol, bash scripts and scheme revision 1.0-a team cplgroup comp6411-s10 term report. *arXiv preprint arXiv:1008.3434*, 2010.

[2] Alexandra Back and Emma Westman. Comparing programming languages in google code jam. Master's thesis, 2017.

[3] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 156–161, 2017.

[4] Ivo Balbaert. *Rust Essentials: A quick guide to writing fast, safe, and concurrent systems and applications*. Packt Publishing Ltd, 2017.

[5] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. Oil and water? high performance garbage collection in java with mmtk. In *Proceedings. 26th International Conference on Software Engineering*, pages 137–146. IEEE, 2004.

[6] Hans-J Boehm. Threads cannot be implemented as a library. *ACM Sigplan Notices*, 40(6):261–268, 2005.

[7] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee Jr, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 324–337, 2003.

[8] Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. Towards a green ranking for programming languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages*, pages 1–8, 2017.

[9] The Rust Project Developers. Reference cycles can leak memory. `https://doc.rust-lang.org/book/second-edition/ch15-06-reference-cycles.html`, 2011 (accessed April 2018).

[10] The Rust Project Developers. *Ownership*, 2018.

[11] James Goodwill, Wesley Matlock, and Bruce Wade. *Beginning Swift games development for iOS*. Springer, 2015.

[12] Apple Inc. Advanced memory management programming guide. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html#//apple_ref/doc/uid/10000011i, 2012 (accessed March 2018).

[13] Apple Inc. Transitioning to arc release notes. https://developer.apple.com/library/archive/releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011226, 2012 (accessed March 2018).

[14] Apple Inc. Automatic reference counting. https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html, 2018 (accessed March 2018).

[15] Apple Inc. *Collection types*, 2018 (accessed March 2018).

[16] Apple Inc. Deinitialization. https://docs.swift.org/swift-book/LanguageGuide/Deinitialization.html, 2018 (accessed March 2018).

[17] Apple Inc. Initialization. https://docs.swift.org/swift-book/LanguageGuide/Initialization.html, 2018 (accessed March 2018).

[18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[19] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[20] Yi Lin, Stephen M Blackburn, Antony L Hosking, and Michael Norrish. Rust as a language for high performance gc implementation. *ACM SIGPLAN Notices*, 51(11):89–98, 2016.

[21] Waqar Malik. *Learn Swift 2 on the Mac: For OS X and IOS*. Springer, 2015.

[22] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[23] Julio CB Mattos and Luigi Carro. Object and method exploration for embedded systems applications. In *Proceedings of the 20th annual conference on Integrated circuits and systems design*, pages 318–323, 2007.

[24] Sebastian Nanz and Carlo A Furia. A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788. IEEE, 2015.

[25] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[26] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165, 2014.

[27] Rufflewind. Graphical depiction of ownership and borrowing in rust. https://rufflewind.com/2017-02-15/rust-move-copy-borrow, 2017 (accessed June 2018).

[28] Narendran Sachindran and J Eliot B Moss. Mark-copy: Fast copying gc with less space overhead. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 326–343, 2003.

[29] Sudhir Sangappa, K Palaniappan, and Richard Tollerton. Benchmarking java against c/c++ for interactive scientific visualization. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 236–236, 2002.

[30] Harwinder Singh. Speed performance between swift and objective-c. 2016.

[31] Jonathan Turner. Rust 2017 survey results. https://blog.rust-lang.org/2017/09/05/Rust-2017-Survey-Results.html, Sep 2017.

# Appendices

## A. STRONG REFERENCE CYCLE

### A.1 Rust

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};
use std::fmt::Display;
#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Option<Rc<Node>>>,
    next: RefCell<Option<Rc<Node>>>,
}
impl Drop for Node {
    fn drop(&mut self) {
    println!("Dropping! Node value {}",
    self.value);
    }
}
fn main() {
  let L1 = Rc::new(Node {value: 1,
    parent: RefCell::new(None), next:
    RefCell::new(None),});
  let L2 = Rc::new(Node {value: 5,
    parent: RefCell::new(None), next:
    RefCell::new(Some(Rc::clone(&L1))),
    });
  //Stack overflow L2 points to L1 parent
  *L1.parent.borrow_mut() =
    Some(Rc::clone(&L2));
}
```

### A.2 Swift

```
class Phone {
    var number : UInt
    var customer : Customer?= nil
    init(n:UInt) {
        number = n
        print("\(number) has been
    created!")
    }
```

```
    deinit {
        print("Phone \(number) deleted!")
    }
}
class Customer {
    var name : String
    var phone: Phone? = nil
    init(n: String) {
        name = n
        print("Customer \(name) has been
    created!")
    }
    deinit {
        print("Customer \(name) deleted!")
    }
}
customer1!.phone = phone1
phone1!.customer = customer1
customer1 = nil;
phone1 = nil
```