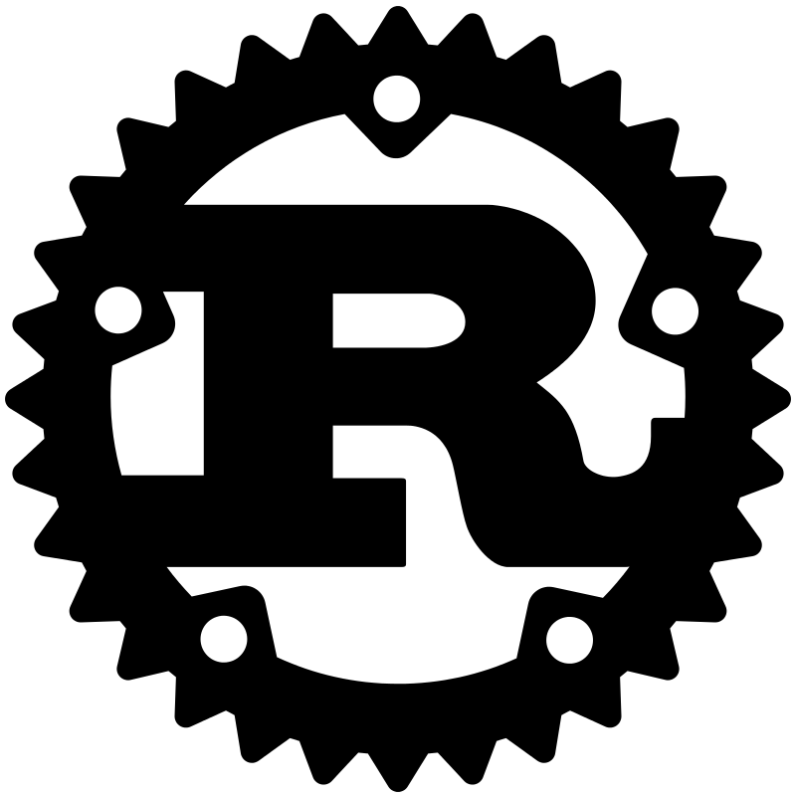
CAB402 Programming Paradigms

Introduction to Rust



Student Name: Kwun Hyo Lee

Student Number: N9748482

Due Date: 12/04/2019

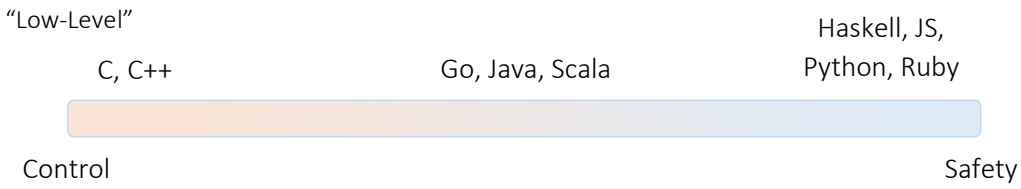
Table of Contents

1.0 Introduction

"Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety."

- https://www.rust-lang.org/

Why another one? Many might question why another programming language is necessary when we have over 700 notable languages to choose from [1]. From all these languages, some opt for more control over safety, and others prioritise more safety over control.



The spectrum above shows an approximation of the control versus safety of several programming languages. All the languages above prioritise one over another or neglect some aspects of one for the other.

For example, C, C++, and other *"low-level"* languages have stronger control over the hardware which they run on and can be optimised and translated to assembly code. It can be easily determined exactly what they will do. However, it is not "*safe"* as it is prone to segfaults and is easy to *"shoot yourself in the foot"*. While modern C++ is significantly safer than it previously was at release in 1985, there are fundamental aspects which do not allow the language to ever be truly safe [2].

Rust is a programming language which disregards the spectrum and simply provides both control and safety without compromising on any. It is a systems programming language like C and C++, and thus provides fine-grained control over allocations, does not require a garbage collector, has minimal runtime, and is close to the hardware. It is *"blazingly fast"* as it compiles to an executable binary, has an LLVL backend, and an LLVM's suite of optimisation. It also *"prevents segfaults"*, which makes the language safe by default. There are no segfaults, null pointers, or dangling pointers. Rust also *"guarantees thread safety"* by *ownership*, *borrowing*, and strong and safe abstractions. Today, we have highly concurrent computers with numerous cores and many threads running in parallel, but it is difficult to write correct parallel code. Rust guarantees that programmers cannot write wrong parallel code. Programmers may find logical errors, but they cannot segfault or find any undefined behaviours [3].

This report will aim to define the Rust programming language, its syntax, the design features of the language, and performance compared to other low-level languages.

2.0 Background

Rust is a general-purpose, multiparadigm systems programming language [4] originally designed by Graydon Hoare at Mozilla Research with contributions from Dave Herman, Brendan Eich, and others [5]. It was designed to pursue better memory safety without disregarding high performance.

The language strongly focuses on its design to be memory safe, its memory management, the ownership system, type system, and safe concurrency [4]. The Rust Programming Language Book [2] features the following features as below,

- Pattern matching and algebraic data types (enums)
- Task-based concurrency. Lightweight tasks can run in parallel without sharing any memory
- Higher-order functions (closures)
- Polymorphism, combining Java-like interfaces and Haskell-like type classes
- Generics
- No buffer overflow
- Immutable by default
- A non-blocking garbage collector

The first appearance of Rust was on July 7, 2010, and the current stable release is version 1.35.0 on May 23, 2019 [6]. Throughout the years, Rust was considered the "most loved programming language" in the Stack Overflow Developer Survey for 2016 [7], 2017 [8], 2018 [9], and 2019 [10].

2.1 What is Control and Safety?

System programming languages are languages which are designed to operate and control the computer hardware by most efficient means possible. Control in programming languages achieves highly performant programs via more direct access to the physical hardware.

Languages such as C and C++ lack a crucial feature for building secure and reliable infrastructural software systems. While both languages have strong control over hardware, they both lack safety as all program errors must either be detected before execution or be gracefully handled during execution; so that such errors shall not lead to unpredictable system behaviours [11].

Even if safety does not propose a large concern during development, it can become largely problematic when computers connect to a network such as the *Internet* where security is crucial. Safety violations such as buffer overflows, out-of-bound array accesses, and dangling pointers accesses of software systems can be exploited by hackers to subvert systems and gain unauthorised control [11].

A C++ implementation of such security vulnerabilities can be seen below,

```
// Aliasing and Dangling Pointer Example
void example() {
    vector<string> vector;
    …
    auto& elem = vector[0];
    vector.push_back(some_string);
    cout << elem;
}
```

Figure 1: Aliasing and Dangling Pointer Example in C++

The code snippet above proposes an issue as it pushes new data and deallocates the content of the previous data pointer, creating a dangling pointer. When the code prints out *elem* to the standard output, it returns an undefined behaviour in the form of most likely a segfault.

While C and C++ today can achieve extreme reliability through using mature software engineering techniques and experienced developers, such methods cannot guarantee an absolute degree of safety. This risk can only increase with larger projects involving over millions of lines of code. Figure 1 shows that these safety violations are usually not due to the inexperience of developers, but the root issue lies in the inadequacies of the programming language itself.

2.1.1 How is Rust *"Safe"*?

The Rust Programming Language was designed with an emphasis on safety, more specifically, safety in memory management, and concurrency [4].

While Rust has subtle differences such as immutable variables and the lack of the *null* value [12] (eliminating *NullReferenceException*s), it strongly focuses on safety for memory.

Higher control over the memory of a system allows for more performant software. However, this may lead to a higher risk of making mistakes and errors may be introduced.

Higher level languages such as Java essentially eliminates the risk of such errors by automatically managing memory with the use of an automatic garbage collector. Garbage collectors can be extremely efficient when doing exactly what was intended. However, it significantly loses efficiency when it is unable to do so [13].

Thus, Rust meets both grounds between the manual memory management of low-level languages, and the automatic garbage collection of higher-level languages. This solution is referred to as *Ownership* [13], which will be further discussed below (refer to Section 3.2).

Zero Cost Abstraction is also a strong principle of Rust and is defined as abstractions that do not impose a cost over the optimal implementation of the task it is abstracting. Rust allows programmers to add structure and readability to their code without suffering in runtime or memory performance [14] (further discussed in Section 3.4.1).

3.0 Design

The following section of the report will discuss Rust's syntax and key aspects which are unique to the Rust language.

3.1 Syntax

Rust aims to achieve the readability and appearance of high-level languages and is thus, syntactically similar to the C and C++ Programming Languages. As shown below, code blocks are delimited by curly brackets.

```
// Simple Hello, world! Program in Rust
fn main() {
    println!("Hello, world!");
}
```

Figure 2: Hello, World! Example in Rust

Control flow statements such as *if*, *else*, *while*, and *for* is available in Rust, alongside additional Rust functions such as the keyword *match* for pattern matching.

However, despite the similarities with C and C++ in syntax, Rust is considered in a deeper sense to be closer to that of the ML family of languages and the Haskell language. Almost all parts of a function body in Rust is an expression, including control flow operators. For example, the *if* expression also takes the place of C's ternary conditional.

Functions in Rust also are not required to end with a *return* expression as the last given expression in the function is simply returned.

3.1.1 Variables and Mutability

As discussed in section 2.2.1, default variables are immutable, which contributes to the safety and easy concurrency which Rust provides.

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

Figure 3: Variables and Immutability Example in Rust

The code snippet above should return an error as an immutable value, *x*, cannot be assigned twice. Mutable variables must be declared as shown below,

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

Figure 4: Mutable Variables Example in Rust

3.2 Ownership and Borrowing

Rust's most notable and central feature is *Ownership*, which enables Rust to guarantee memory safety without the use of a garbage collector.

All programs must decide how they use a computer's memory during execution. Some languages have a garbage collection which constantly searches for memory that is no longer in use. Other languages require the programmer to explicitly allocate and free the memory [13].

Rust provides a different solution; it manages memory through an *ownership* system with a set of rules which the compiler checks at compile time.

3.2.1 Stacks and Heaps

Before delving further into the *ownership* system, it is important to understand the *stack* and the *heap*. While it is not necessary to remind yourself of stacks and heaps in most languages, *Rust* is a systems programming language and is essential to know whether a value exists on the stack or the heap.

Stacks are a linear data structure which stores values *last-in-first-out* (LIFO), meaning that they store values in the order in which they are given. They also remove values in the opposite order. Stacks are required to have a known, fixed size and thus, when data has a mutable size or a size unknown at compile time, it must be stored on the heap [15].

Heaps are usually less organised, and when data is inserted into a heap, it requests for an amount of space. The operating system searches for an empty space on the heap which meets the size requirements, marks it as being in use, then returns a pointer to the space. This process is referred as *allocating on the heap* or simply *allocating*. Pushing data onto the stack is not considered *allocating*. As the pointer is a known and fixed size, it can be stored on the stack. However, to retrieve the actual data, it is required to follow the stored pointer [15].

Thus, pushing data onto a stack does not require as much time as *allocating on the heap*, as it does not need to find space to store the new data; it is simply pushed on the top of the stack. Comparatively, *allocating on the heap* requires more effort as the operating system must search for a space which meets the size requirement of the data, and then perform bookkeeping to prepare for the next allocation [15].

Likewise, accessing data in the heap is slower than accessing data on the stack as it is required to follow a pointer to access the heap data.

When a function is called, the arguments passed into the function and the function's local variables get pushed onto the stack. When the function is over, the values are then popped off [15].

Ownership addresses the issues [13] given below:

- Keeping track of what parts of code are using what data on the heap
- Minimising the amount of duplicate data on the heap
- Cleaning up unused data on the heap so that you don't run out of space

As given in the *Rust Programming Language Book*, *ownership* consists of three rules,

- Each value in Rust has a variable that's its *owner*

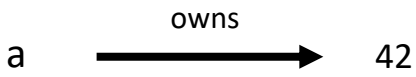


Figure 5: Variable *a* owns the value 42

- There can only be one owner at a time

Figure 6: Variable *a* and *b* cannot both own the same value, 42

- When the owner goes out of scope, the value will be dropped

Before further exploring the last rule given above, it is important to know how variables and data interact and how ownership is assigned to values in Rust. Take a look at the example below,

```
let x = 5;
let y = x;
```

Figure 7: Binding Integers to Variables Example in Rust

By observation, we can assume that this code simply binds the value 5 to *x*; then makes a copy of the value in *x* and binds it to *y*. And this assumption would be correct, because integers are simple values with a known, fixed size, and these two 5 values are pushed onto the stack.

The code below shows an example where the size of the data is unknown as strings can vary in data size.

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

Figure 8: Binding a Copy of a String from *s1* to *s2* (*Moving* the Value of *s1* to *s2*)

The composition of the string of *s1* [15] can be seen below,

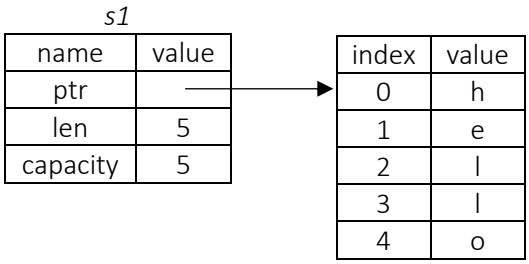


Figure 9: *s1* String Value Composition

We might assume again that the code binds a copy of the value of *s1* to *s2* as shown below [15], however, this is incorrect.

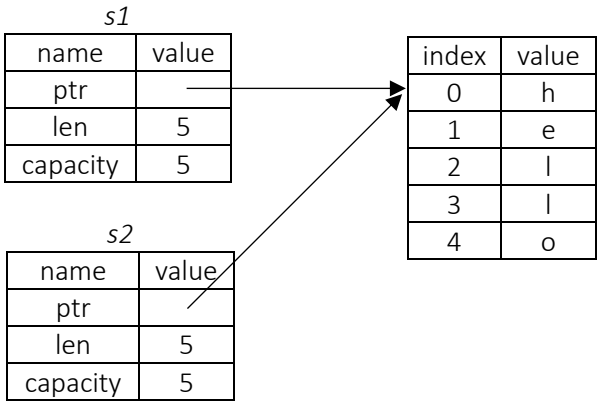


Figure 9: Incorrect Assumption of the Code in Figure 8

The last rule states when the owner goes out of scope, the value is dropped. Rust automatically calls the drop function and cleans up the heap memory for that variable. When there are two data pointers pointing to the same location, this proposes an issue as when *s1* and *s2* go out of scope, they will both try to free the same memory. This is known as a double free error and is a memory safety bug which can lead to memory corruption and potentially lead to security vulnerabilities. To ensure memory safety, Rust considers *s1* to no longer be valid, thus, Rust does not need to free anything when *s1* goes out of scope. This process is called a *move* [15].

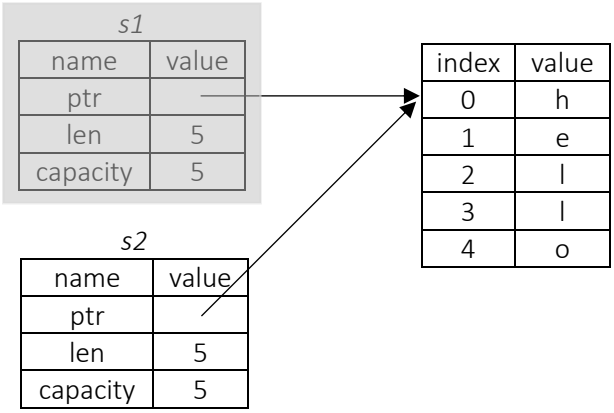


Figure 10: Accurate Representation of the Code Output in Figure 8

To deeply copy the heap data of the String, not just the stack data, Rust provides a common method called *clone* as shown below,

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

Figure 11: Clone Functionality in Rust

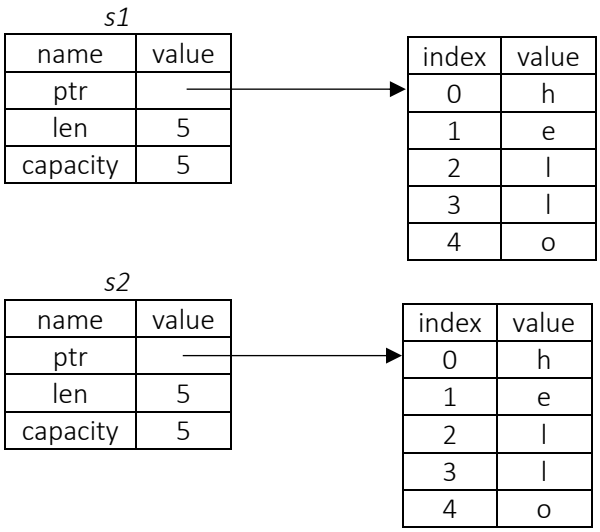


Figure 12: Output of the Code in Figure 11

However, *cloning* is very expensive in terms of runtime performance, especially if the data on the heap were large [15].

Function arguments are similar to variable assignment and can also take ownership as shown below.

```
fn main() {
    let x = String::from("hello");
    drop_string(x);
}

fn drop_string(s: String) {
}
```

Figure 13: Taking Ownership of a Variable *x* in Rust

The code above presents a function which takes ownership of a string and drops it.

First, variable *x* owns the string value, "hello". The value *x* is then passed in the drop_string function. The string's ownership is moved to the argument *s* and *x* loses ownership. Then the function ends, ending the scope, and the stack frame is thrown away. Thus, string *s* is dropped.

Ownership of this value can also be returned from the function as shown below, however, it is not ideal to always pass around ownership.

```
fn main() {
    let x = String::from("hello");
    return_ownership(x);
}

fn return_ownership(s: String) -> String {
    s
}
```

Figure 14: Passing Ownership in Rust

Thus, Rust provides a solution called borrowing. Borrowing, however, is bound by several rules. Rust allows unrestricted instantiations of read-only accesses of a variable. This borrowing is achieved by using the ampersand as shown below,

```
let mut x = String::from("hello");
let z = &x;
let y = &x;
```

Figure 15: Borrowing Ownership in Rust

While *x* is borrowed by *z* and *y*, the value is immutable. The value is made mutable when the scope closes as shown below.

```
let mut x = String::from("hello");
{
    let z = &x; // x is immutable
    let y = &x;
}
// x is mutable
```

Figure 16: Showing Mutability Changes During Borrowing in Rust

Rust also allows for mutable borrows by using the keyword *&mut* as shown below. However, this only allows for a single instantiation of a mutable borrow at a time.

```
let mut x = String::from("hello");
{
    let z = &mut x; // x is immutable
} // x is mutable
```

Figure 17: Mutable Borrow in Rust

In Rust, the caller of a function can see which arguments take ownership, which are read-only borrows, and which are mutable borrows from the type signature as shown below.

```
fn example_func(a: String, b: &str, c: &mut str) {
}
```

Figure 18: Recognizing Function Argument Ownership and Borrows in Rust

3.3 Memory Safety

The Rust Programming Language was designed prioritising memory safety. This resulted in a language which can effectively eliminate all cases of null pointers, dangling pointers, and data races in safe Rust.

The Rust core library provides an *option* type, which allows for testing to determine whether a pointer has *Some* value or *None*. Doing so eliminates the need for a *null* value and facilitated the removal of *null* pointers.

Dangling pointers were also eliminated as discussed in Section 3.2 as the *ownership* and *borrowing* system requires that there “can only be one owner at a time”.

3.4 Memory Management

3.4.1 Zero-Cost Abstractions

The Rust Programming Language strives for the capabilities of low-level languages while providing the readability and manageability of high-level languages. This is achieved via zero-cost abstractions for Rust’s memory management. Like the C Programming Language, Rust uses manual memory management, meaning that the programmer is in complete control of which memory is being allocated where. However, unlike C, Rust automatically knows when to allocate and free memory [16]. Memory management has been abstracted away from the view of the programmer to achieve the manageability of higher-level languages [17]. This is accomplished by the *ownership* and *borrowing* system as discussed in Section 3.2.

3.5 Types and Polymorphism

In the Rust Programming Language, all variables, items, and values have a type which defines the attribute of the data and the operations which may be performed with the data [18].

3.5.1 Type Inference

Like many other languages, Rust features *type inference* for variables declared with the keyword *let*. This means that while all variables have a type in Rust, it is not required to initially assign a type to a variable. Rust can simply and automatically detect the data type of data by their use in an expression. If any branch of code within the program fails to assign a value to a variable, Rust results with an error at compile time [19]. Thus, Rust's type system does not affect the end-user beyond compile-time errors, which facilitates faster and more efficient development.

3.5.2 Function Generic Parameters

In Rust, functions arguments can contain generic types, which are often accompanied by trait(s) implemented by the generic value. The generic value can only be used within the scope of the given traits and thus, a generic function can be type-checked after being defined. While this is opposed to C++ templates, the implementation of Rust generics is alike C++ templates in its monomorphization, meaning that another copy of code is generated with each instantiation.

3.5.3 Object System and Object-Oriented Programming in Rust

The chances are high that most programmers have previously implemented Object-Oriented Programming (OOP) in one way or another. In OOP, there are *classes* which act as a blueprint for creating *objects* defining unique types. *Classes* can inherit from other *classes* known as their *parent class*, inheriting the class' data (*fields*) and behaviours (*methods*). The concept of *subtyping* also allows the logic that if B inherits from A, then an instance of B can be passed to something expecting an A. Also, an object can often hide its data through *encapsulation*, which only allows modification and retrieval of its data via methods.

As a multi-paradigm programming language, Rust is capable of implementing an object system, however, it is founded around implementations, traits, and structured types. Implementations act as *classes* from other OOP languages and can be defined by the keyword *impl* [20]. Traits are used to achieve inheritance and polymorphism as given in OOP, where they allow methods to be defined and integrated into implementations [21]. However, both implementations and traits cannot define fields within themselves, and require structured types to do so [22]. In addition, only traits are given the ability to provide inheritance. An example of these features is seen below.

```
// The Animal trait provide functionalities (in this case,
// do_sound). And traits itself, cannot contain data like classes would.
trait Animal {
    fn noise(&self);
}

// The Dog struct contains fields as a class would in other languages.
struct Dog {
    name: String,
}

// Methods are implemented in the implementation and are not in the
// struct
impl Dog {
    // Rust only provides one constructor but it is in common practice to
    // create a function to serve the purpose of a new constructor
    fn new(name: &str) -> Self {
        // to_owned clones the borrowed string (&str)
        Dog{ name: name.to_owned() }
    }

    fn wag(&self) {
        println!("{} (the dog) wagged!", self.name);
    }
}
// The snippet below declares that Dog is an Animal by implementing the
// Animal trait.
impl Animal for Dog {
    fn noise(&self)
    {
        println!("{} (the dog) said: Can I pass this unit?", self.name);
    }
}
```

Figure 19: Animal Object-Oriented Programming Example in Rust

While programmers new to the Rust Programming Language may struggle with grasping concepts of OOP once again, Rust has been cleverly designed as such to avoid mistakes recognisable in the OOP paradigm, such as the diamond problem of multiple inheritance in C++, and more.

3.6 Concurrency

Concurrency is the ability to allow the execution of partial sections of a program with disregard for execution order, without affecting the final outcome. Rust was designed to prioritise memory safety via its *ownership* and *borrowing* system. However, the *ownership* system did not only aim to eliminate memory errors, but to achieve safe concurrency. Through Rust, many concurrency errors were made into compile-time errors instead of runtime errors [17].

Performing computation of a program in multiple threads improve performance as the program does multiple tasks at a time. However, running threads simultaneously can propose complexity and potential issues such as race conditions, deadlocks, and other bugs that could be hard to reproduce and fix reliably. Data races are another issue for many system programming languages and is defined as two accesses to a shared memory location without synchronisation, and at least one of the accesses being a write.

The Rust Programming Language eliminates data races by leveraging the *ownership* and *borrowing* system and *type checking*, as it makes it impossible to alias a mutable reference.

4.0 Comparing Rust and C++

A comparison between Rust and C++ may almost seem unreasonable as Rust is still less than a decade old, compared to the release of the first edition of C++ in 1985. The Google Trends graph below can even show the maturity gap between both languages [23].

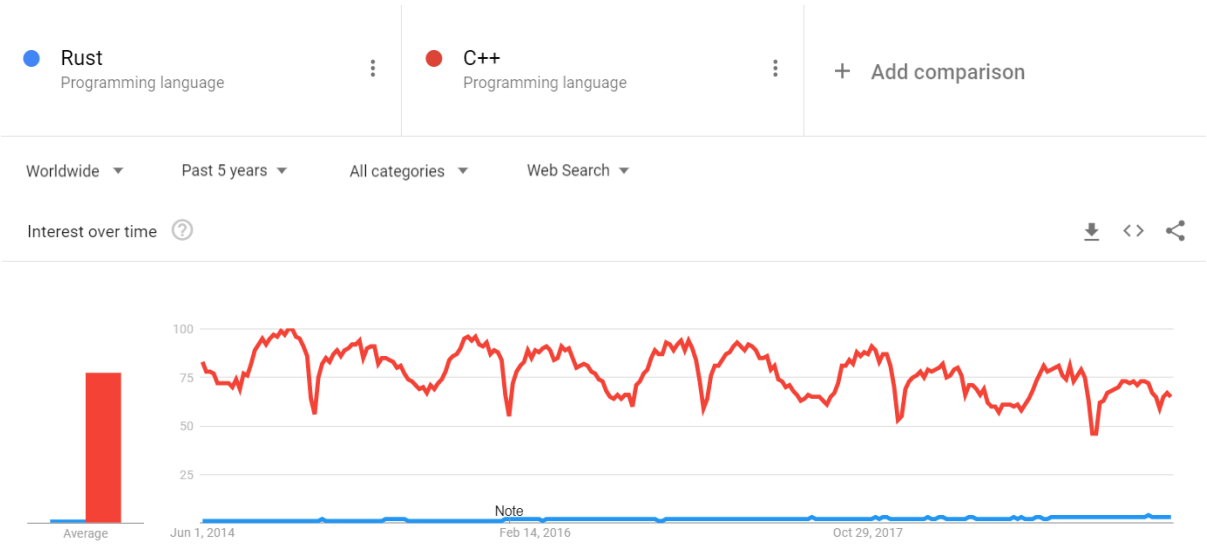


Figure 20: Rust vs. C++ in Web Search (Google Trends)

C++ had maintained a consistent trend for the past 5 years, only showing a very slight decline, perhaps due to the other new contending languages such as Rust and Go. However, compared to Rust, it shows much more promise in popularity and usage.

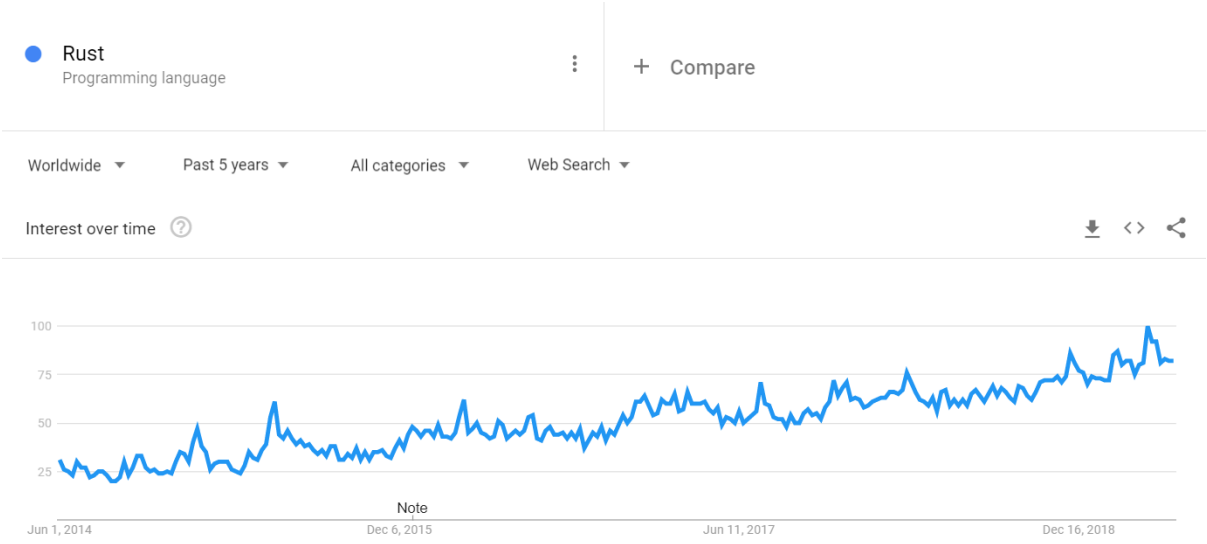


Figure 21: Rust Web Search Google Trends

Though, when Rust is examined by itself, it shows a peculiar trend [24]. Unlike C++, Rust has not faced a significant or consistent fall in usage and popularity. Instead, the trend for Rust shows a promising and approximately linear growth in fame.

Rust had also maintained its name as the “most loved programming language” in the Stack Overflow Developer Survey for four consecutive years from 2016 to 2019 [7] [8] [9] [10] (refer to appendix 1), compared to C++ which is the fourth from the last on the list.

Many might question why is the love so great for Rust when C++ can deliver high performance with its rich function library? C++ has a wide use of applications and is primarily used today, from GUI applications to games, desktop applications, as well as operating systems.

When presented in comparison with C++, Rust is loved for its safety. Not only it can measure up to the performance of C++ (refer to Section 4.1), but it protects its own abstractions and also allows programmers to protect theirs too. Rust will never result with any arbitrary behaviour so that programmers can concentrate on solving programming problems.

4.1 Performance Benchmarks

Benchmark comparisons between Rust and C++ g++ is provided fairly, using the reverse-complement, n-body, binary-trees, pidigits, spectral-norm, fannkuch-redux, fasta, regex-redux, mandelbrot, and the k-nucleotide algorithms. The specifications for the algorithm benchmarks are as below [25].

Rust	rustc 1.35.0 (3c235d560 2019-05-20) LLVM version 8.0
C++ g++	g++ (Ubuntu 8.3.0-6ubuntu1) 8.3.0

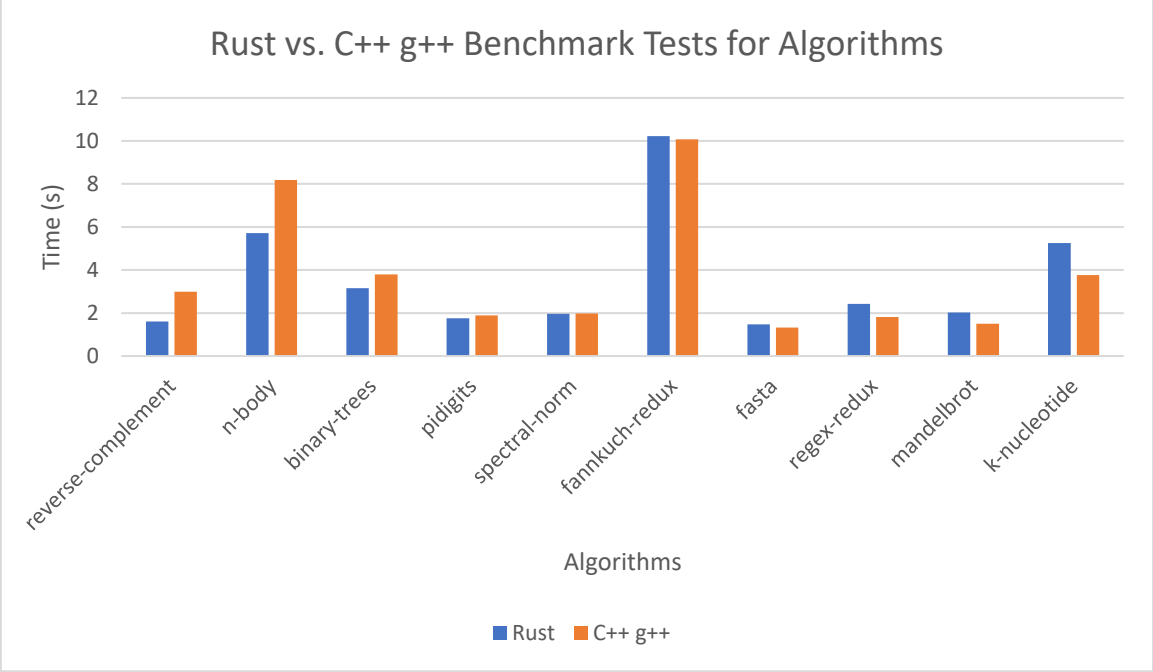


Figure 22: Rust vs. C++ g++ Benchmark Tests for Specific Algorithms Graph

It can be observed from the data above, that while Rust is a language still in its early stages, it can already parallel the performance of C++. While the algorithms above may not provide insight in the languages and their approach to memory management, parallel programming, and implementation technique, this data was created with interest in measuring the quality of generating code when both compilers are presented with what amounts to the same program.

5.0 Personal Comments

Learning about the Rust Programming Language and its ambitions have been a pleasant and insightful surprise. Special features such as the *ownership* system provided Rust with a learning curve steeper than many other languages.

However, while it was embarrassing to be unfamiliar with system programming despite my years studying computer science, researching Rust had not only introduced me to low-level languages but to an aspect of programming which I had not considered too deeply. Learning Rust brought to my attention, the issues regarding system programming languages and what they compromise to achieve performance. It was astonishing to see a new programming language which aimed to not be between the spectrum of performance and safety, but in both ends of being fully performant and completely safe.

Learning about another programming language while developing in F# had also led me to question these languages. For example, both Rust and F# share the feature of having immutable default values and requires the declaration of a mutable value to mutate data. As I had only experienced high-level programming languages, such functional properties were a foreign concept. This concept, however, seemed to be logical as opposed to the standard in C and C++. After all, it seemed best to work with immutable values throughout a program to stay consistent and to provide stronger readability. Through research, Rust had not only seemed to provide a safer experience to program but also improved many faults in C and C++ which were inapparent with the lack of programming knowledge at the given time period.

Rust had also helped me to delve further into language typing as the typing was a core functionality of the language. It brought me to language design principles and had me question the design principles of Rust. Rust allowed for most principles from efficiency, simplicity, extensibility, preciseness, and expressiveness through its design features discussed in Section 3.0.

Initially, I did delve into a research topic which I was unfamiliar with, hoping to learn about the recent fame and popularity of Rust. Rust was simply another one of those languages which I've vaguely heard of but never piqued my interest as I did not involve myself with systems programming. However, I have come to understand the ambitions of the Rust Programming Language and am excited to see the course of Rust in the future.

6.0 Bibliography

[1] "How Many Programming Languages are there in the World?," 17 November 2017. [Online]. Available: http://codelani.com/posts/how-many-programming-languages-are-there-in-the-world.html.

[2] rust-lang.org, "The Rust Programming Language Book," 2018.

[3] A. Crichton, Interviewee, *Rust: Safe and Scalable Systems Programming.* [Interview]. 28 August 2016.

[4] rust-lang.org, "Rust," rust-lang.org, 2019. [Online]. Available: https://www.rust-lang.org/. [Accessed 11 May 2019].

[5] rust-lang, "Contributors," GitHub, 11 May 2019. [Online]. Available: https://github.com/rust-lang/rust/graphs/contributors. [Accessed 11 May 2019].

[6] rust-lang.org, "Announcing Rust 1.35.0," Rust Blog, 23 May 2019. [Online]. Available: https://blog.rust-lang.org/2019/05/23/Rust-1.35.0.html. [Accessed 24 May 2019].

[7] StackOverflow, "Developer Survey Results 2016," StackOverflow, 2016. [Online]. Available: https://insights.stackoverflow.com/survey/2016#technology-most-loved-dreaded-and-wanted. [Accessed 11 May 2019].

[8] StackOverflow, "Developer Survey Results 2017," StackOverflow, 2017. [Online]. Available: https://insights.stackoverflow.com/survey/2017#most-loved-dreaded-and-wanted. [Accessed 11 May 2019].

[9] StackOverflow, "Developer Survey Results 2018," StackOverflow, 2018. [Online]. Available: https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted. [Accessed 11 May 2019].

[10] StackOverflow, "Developer Survey Results 2019," StackOVerflow, 2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019#technology-_-most-loved-dreaded-and-wanted-languages. [Accessed 12 May 2019].

[11] P. Li, "Safe System Programming Languages," University of Pennsylvania, 2004.

[12] N. Brown, "A Taste of Rust," LWN, 17 April 2013. [Online]. Available: https://lwn.net/Articles/547145/. [Accessed 13 May 2019].

[13] rust-lang.org, "Rust," in *The Rust Programming Language Book*, 2018, p. Ownership and Lifetimes.

[14] rust-lang, "Abstraction Without Overhead: Traits in Rust," Rust Blog, 11 May 2015. [Online]. Available: https://blog.rust-lang.org/2015/05/11/traits.html. [Accessed 12 May 2019].

[15] rust-lang, "What is Ownership," rust-lang.org, 2018. [Online]. Available: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html. [Accessed 7 May 2019].

[16] rust-lang, “Fearless Concurrency,” rust-lang, 2018. [Online]. Available: https://doc.rust-lang.org/book/ch16-00-concurrency.html. [Accessed 7 May 2019].

[17] S. Shanker, “Safe Concurrency with Rust,” squidarth, 4 June 2018. [Online]. Available: http://squidarth.com/rc/rust/2018/06/04/rust-concurrency.html. [Accessed 7 May 2019].

[18] rust-lang, “Types,” rust-lang, 2017. [Online]. Available: https://doc.rust-lang.org/reference/types.html. [Accessed 7 May 2019].

[19] “Rust Features I: Type Inference,” ~pcwalton, 1 October 2010. [Online]. Available: http://pcwalton.blogspot.com/2010/10/rust-features-i-type-inference.html. [Accessed 7 May 2019].

[20] rust-lang, “Implementations,” rust-lang, 2017. [Online]. Available: https://doc.rust-lang.org/reference/items/implementations.html. [Accessed 7 May 2019].

[21] rust-lang, “Traits,” rust-lang, 2018. [Online]. Available: https://doc.rust-lang.org/rust-by-example/trait.html. [Accessed 7 May 2019].

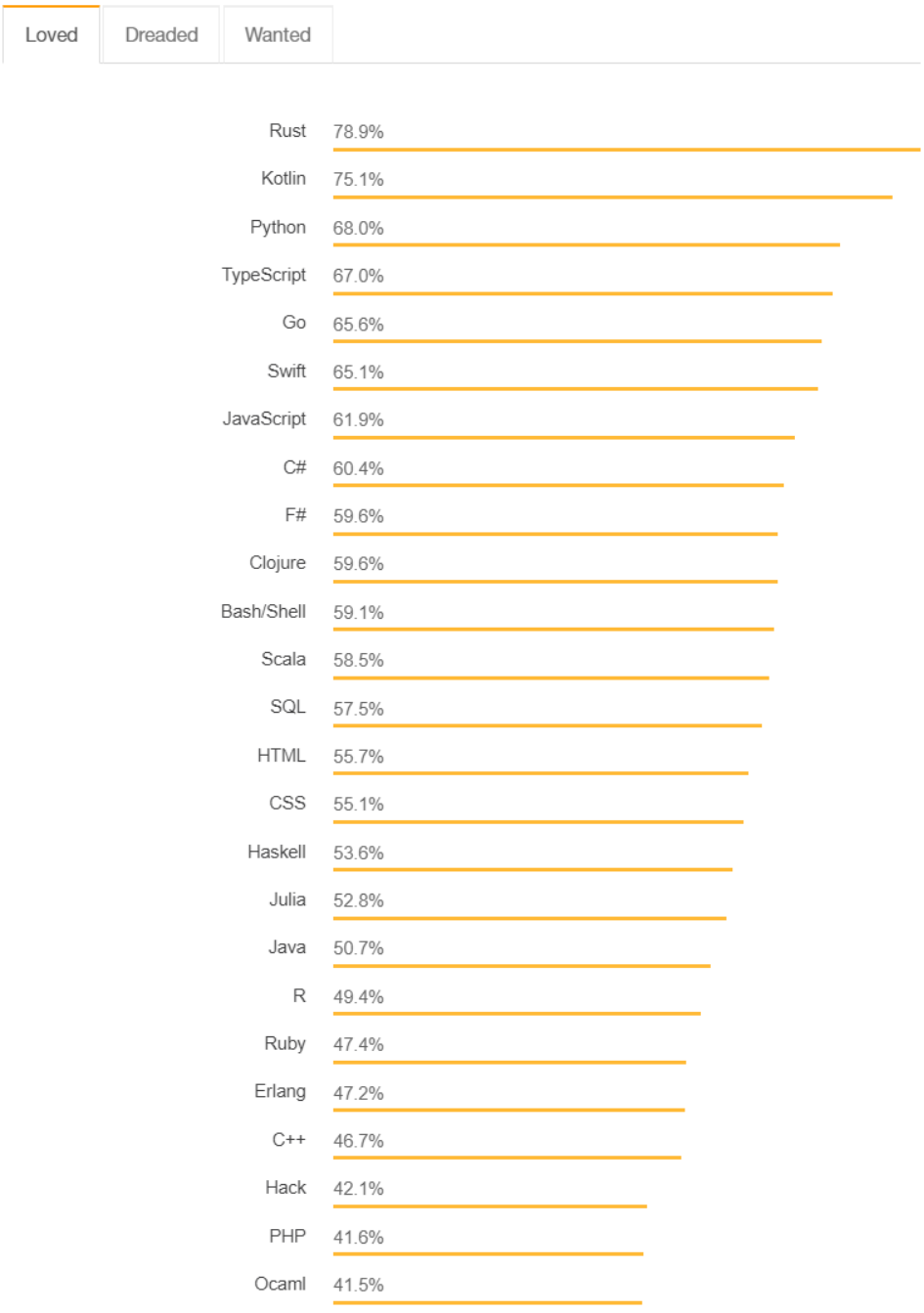
[22] rust-lang, “Structures,” rust-lang, 2018. [Online]. Available: https://doc.rust-lang.org/rust-by-example/custom_types/structs.html. [Accessed 7 May 2019].

[23] G. Trends, “Compare Rust vs. C++ by Web Search - 5 Years,” Google Trends, 2019. [Online]. Available: https://trends.google.com/trends/explore?hl=en-US&tz=-600&date=today+5-y&q=%2Fm%2F0dsbpg6,%2Fm%2F0jgqg&sni=3. [Accessed 10 May 2019].

[24] G. Trends, “Compare Rust by Web Search - 5 Years,” Google Trends, 2019. [Online]. Available: https://trends.google.com/trends/explore?hl=en-US&tz=-600&date=today+5-y&hl=en-US&q=%2Fm%2F0dsbpg6&tz=-600&sni=3. [Accessed 10 May 2019].

[25] “Rust versus C++ g++ fastest programs,” The Computer Language Benchmarks Games, 20 May 2019. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/rust-gpp.html. [Accessed 21 May 2019].

7.0 Appendices



Appendix 1: Stack Overflow Developer Survey Results 2019