

INSTITUT FÜR INFORMATIK  
der Ludwig-Maximilians-Universität München

# MEMORY MANAGEMENT IN RUST

---

Principles and Comparison with C  
and C++

Claudio Zeeb

## Bachelor Thesis

Supervisor  
Mentor

Prof. Dr. François Bry  
Dipl.-Ing. Thomas Prokosch

Submission date 07/07/2022



---

## Erklärung

---

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 07.07.20202



Claudio Zeeb



---

## Abstract

---

Rust is a new general purpose programming language which promises thread and memory safety without relying on a tracing garbage collector. To achieve this, the language utilizes a memory management concept called ownership. This concept validates memory accesses and references through compile time checks. With this approach, Rust is able to offer performance similar to those of C and C++. The combination of these features makes the language especially suitable for system programming – a field traditionally dominated by C and C++.

This thesis describes the memory management concepts of Rust and compares them to those of C and C++. This is done by providing exemplary code listings to analyze the semantics of the memory management approaches of each language. By further comparing the memory management of Rust to the traditional approaches of garbage collection and manual memory allocation, this thesis evaluates the advantages and disadvantages of the ownership model employed in Rust.



---

## Zusammenfassung

---

Rust ist eine neue Allzweckprogrammiersprache, die Thread- und Speichersicherheit verspricht, ohne sich dabei auf einen Speicherbereiniger zu verlassen. Um dies zu erreichen, verwendet die Sprache ein Speicherverwaltungskonzept namens Ownership. Dieses Konzept validiert Speicherzugriffe und Referenzen durch Überprüfungen während der Übersetzung. Mit diesem Ansatz ist Rust in der Lage, eine Leistung vergleichbar zu C und C++ zu bieten. Die Kombination dieser Merkmale macht Rust insbesondere geeignet für die Systemprogrammierung – eine Domäne, welche traditionell von C und C++ dominiert wird. Diese Arbeit beschreibt die Speicherverwaltungskonzepte von Rust und vergleicht sie mit jenen von C und C++. Dies geschieht durch beispielhafte Quelltextausschnitte, um die Semantik der Speicherverwaltungsansätze der genannten Sprachen zu analysieren. Durch den anschließenden Vergleich der Speicherverwaltung von Rust mit den traditionellen Ansätzen der Garbage Collection und der manuellen Speicherverwaltung werden in dieser Arbeit die Vor- und Nachteile des in Rust verwendeten Ownership-Modells bewertet.





---

## Acknowledgments

---

First and foremost I would like to thank Prof. Dr. François Bry for giving me the opportunity to write this thesis, thereby allowing me to dive into a new (for me, at least) programming language and its internals.

I am also extremely grateful for the continued support from my mentor Dipl.-Ing. Thomas Prokosch. He provided me with suggestions and constructive feedback on many occasions, inspiring me to challenge myself and dig deeper.

A big thank you also goes to the Rust community, in particular the contributors of the official Discord server. The server has allowed me to both gain insights by reading past conversations as well as get answers to my own questions regarding the language.

Finally, I would like to thank my family and friends, who have supported and encouraged me a great deal during this thesis and my bachelor studies in general.



---

## Contents

---

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Aim of this thesis . . . . .	3
1.2	Related work . . . . .	4
1.2.1	Academic research . . . . .	4
1.2.2	Blog posts . . . . .	6
1.2.3	Official Rust books . . . . .	6
<b>2</b>	<b>Syntax and semantics</b>	<b>7</b>
2.1	Syntax and language features . . . . .	7
2.1.1	Variables and values . . . . .	7
2.1.2	Types . . . . .	8
2.1.3	Functions and methods . . . . .	11
2.1.4	Macros . . . . .	13
2.1.5	Traits . . . . .	14
2.2	Ownership . . . . .	16
2.2.1	Variable scope and access . . . . .	16
2.2.2	Moving of ownership versus copying values . . . . .	17
2.2.3	Clone . . . . .	19
2.2.4	Passing ownership across function boundaries . . . . .	19
2.2.5	Ownership in C and C++ . . . . .	21
2.3	Borrowing . . . . .	22
2.3.1	Mutable references . . . . .	22
2.3.2	References and mutability in C and C++ . . . . .	23
2.3.3	Dangling references . . . . .	23
2.4	Lifetimes . . . . .	25
2.4.1	Validating lifetimes with the <i>borrow checker</i> . . . . .	25
2.4.2	Rust code with lifetime annotations . . . . .	26
2.4.3	Lifetime elision . . . . .	28
2.4.4	Non-lexical lifetimes . . . . .	29
<b>3</b>	<b>Garbage collection in Rust</b>	<b>31</b>
3.1	Automatic memory management . . . . .	31
3.1.1	Tracing garbage collection . . . . .	31
3.1.2	Reference counting . . . . .	32
3.1.3	Problems with automatic memory management . . . . .	32
3.2	Garbage collection in early Rust versions . . . . .	32
3.3	Garbage collection as a library . . . . .	33

3.3.1	Motivation for garbage collection in systems programming . . . . .	33
3.3.2	Rust garbage collection libraries . . . . .	34
3.3.3	C and C++ garbage collection libraries . . . . .	35
3.4	Extending Rust's memory management through <b>unsafe</b> calls . . . . .	36
3.5	Rust's solution to memory management . . . . .	38
3.5.1	Reclaiming memory of non-cyclic data structures . . . . .	38
3.5.2	Reclaiming memory of cyclic data structures . . . . .	39
<b>4</b>	<b>Memory management: Rust and C/C++ compared</b>	<b>41</b>
4.1	Movable versus fixed memory locations . . . . .	41
4.2	Memory management in asynchronous Rust programming . . . . .	42
<b>5</b>	<b>Conclusion</b>	<b>45</b>
	<b>Bibliography</b>	<b>47</b>

---

## List of source code listings

---

1	Declaring a variable and assigning a value to it . . . . .	7
2	Declaring a mutable variable . . . . .	7
3	Function definition . . . . .	8
4	Calculating a value with type <code>Option&lt;i32&gt;</code> . . . . .	9
5	Use of tuples . . . . .	9
6	Use of arrays . . . . .	10
7	Creating a vector using <code>new</code> . . . . .	10
8	Creating a vector using <code>from</code> . . . . .	11
9	Adding and removing elements from a <code>Vec</code> . . . . .	11
10	Function definition . . . . .	12
11	Function from Listing 10, but with elided return type . . . . .	12
12	Calling the function from Listing 10 . . . . .	12
13	Definition of a function with multiple parameters . . . . .	12
14	Adding a return value to the function from 13 . . . . .	13
15	Defining a function using generics . . . . .	13
16	Using the <code>println!</code> macro to print variable values . . . . .	14
17	Function with trait bounds . . . . .	14
18	Variable scopes . . . . .	16
19	Accessing an uninitialized variable . . . . .	17
20	Compiler output generated by compiling Listing 19 . . . . .	17
21	Move . . . . .	18
22	Copy . . . . .	18
23	Cloning values . . . . .	19
24	Passing ownership . . . . .	19
25	Passing ownership through return values . . . . .	20
26	Returning ownership and additional return value . . . . .	21
27	Computing length with reference to vector . . . . .	22
28	Appending to a <code>String</code> with a mutable reference . . . . .	23
29	Attempting to create multiple mutable references . . . . .	23
30	Creating a dangling pointer in C . . . . .	24
31	Attempting to create a dangling reference in Rust . . . . .	24
32	Creating an invalid reference . . . . .	25
33	Error message produced by compiling Listing 32 . . . . .	26
34	Visualizing lifetimes from Listing 32 . . . . .	26
35	Attempting to create a function without lifetime annotations . . . . .	27
36	Expanding the function definition from Listing 35 with lifetime annotations . . . . .	27
37	Calling the <code>shorter_vec</code> function . . . . .	28
38	Safe code rejected by compiler: Mutable reference assigned to a variable . . . . .	29

39	Dereferencing a raw pointer . . . . .	37
40	Defining and calling an <b>unsafe</b> function . . . . .	37
41	Implementing a Rust <code>malloc</code> -based memory allocator using <b>unsafe</b> code .	37
42	Moving a value in memory by transferring ownership . . . . .	42

### 1.1 Aim of this thesis

Rust has been named the “most loved programming language” for the sixth year in a row by participants in the annual Stack Overflow Developer Survey [85]. For many, this does not come as a surprise: As a general purpose language, Rust promises type, thread and memory safety while giving the programmer control over memory management and system resources. It offers a desirable combination of safety, performance and flexibility.

Systems and embedded systems programmers often have to consider memory and processing power constraints of their target systems [15]. These systems have traditionally been developed with C and C++ due to the ability of these languages to directly manage system resources. This includes manual allocation and deallocation of memory as needed by the program. While this can be efficient, it is also complicated: Memory errors, such as *use-after-free* and *double free* errors, as well as *race conditions* can occur. These errors have the potential to crash programs. The conditions under which they occur can be hard to reproduce, making them difficult to debug and resolve. Teams at Apple, Microsoft and Google have discovered that over two thirds of security vulnerabilities in their products are attributable to memory unsafety, primarily introduced by developers in C and C++ code [20, 38, 53, 86].

Rust aims to avoid these issues through its innovative concept of ownership and the strict separation of safe and unsafe code. With this approach, Rust guarantees memory safety, making the errors described above impossible. At the same time, Rust does not compromise its performance to achieve this. Many checks ensuring memory safety are performed at compile time.

Rust and C/C++ intentionally share the same abstract machine model [52]. This allows systems programmers to switch to Rust without having to leave behind their intuition and knowledge on how the language interacts with hardware.

This thesis aims to lay out the principles of memory management in Rust and highlights how the language achieves memory safety without using a garbage collector. Throughout this work, the concepts presented are viewed from both perspectives, the one of Rust, and the one of C and C++. This thesis aims to distinguish itself from other publications on Rust by giving an in-depth overview of all memory related topics, while assuming no prior knowledge of the language. This closes a gap in existing publications between blog

posts and tutorials relating to single topics regarding Rust, and scientific publications laying out the theoretical foundations for the concepts of memory management, both general concepts and concepts specific to Rust.

## 1.2 Related work

Since Rust is a relatively new language (the first version 1.0 of which was released in 2015), the amount of academic publications covering the language is limited. However, the developer community around the Rust language is quite active, which leads to many discussions, blog posts and tutorials being published online. This section aims to give an overview on both the academic publications as well as categorize the broad landscape of blog posts and official Rust materials.

### 1.2.1 Academic research

Many scientific Rust-related articles have been published in the context of the “Principles of Programming Languages (POPL)” conference and the RustBelt project [21]. With the more widespread adoption of the language in recent years, additional research has also appeared in other venues.

#### Static memory safety before Rust

The concept of memory safety for programs without the need for runtime checks or garbage collection employed in Rust is not new: A compiler technique to achieve this for a subclass of C programs was proposed by Dhurjati et al. already in 2003 [20]. The ownership concept used in Rust is based on substructural typing, as described by Walker in 2005 [19].

#### Rust and the conference “Principles of Programming Languages”

The development of Rust eagerly considers and contributes to the publications made in the conference “Principles of Programming Languages” (POPL). The conference has produced numerous papers on theoretical programming language foundations, some of which have served as inspiration for the design of the Rust language. Also, many of the developments in Rust have been published in the proceedings of the conference. In 2017, then Rust core team member Aaron Turon was invited to give a keynote speech on transitioning the Rust language from a theoretical concept to a functioning language in practice which described *ownership* and *traits* as Rusts core and most distinguishing features [89].

Work stemming from the development of Rust is being published regularly, such as the formalization of the borrow checking model by Weiss et al. (2021) [94].

#### RustBelt

*RustBelt* is a project started in 2015 with the goal of formally proving the claimed memory and thread safety of the Rust language and its standard library. Within the scope of the project, Jung et al. (2017) have developed a formal representation of the Rust language allowing to prove the safety of a realistic subset of Rust. The model is extensible: New libraries using unsafe code can be included. The model allows the specification of verification conditions for new libraries utilizing unsafe code. When these conditions are met, the library can be considered a safe language extension [37]. In 2019, Jung et al. have also proposed an aliasing model called *Stacked Borrows* for Rust, defining aliasing rules for unsafe code that allow for compiler optimizations while preserving the semantics of the code [36].



Additional work has been done by Dang et al. (2019) on extending the previously mentioned models to concurrent libraries [18] as well as developing methods to relax the constraints placed on mutable aliased state, which is necessary for the implementation of certain data structures [97].

The RustBelt project has also produced work having implications for other languages than Rust: Lee et al. (2018) have implemented a new version of the memory model of LLVM to allow for more optimizations of low-level programs written in C, C++, and Rust (LLVM is the compiler framework used to build components of a variety of compilers, including Rust) [44].

### Usage of Rust

More practical aspects of the Rust language have also been investigated scientifically. An experience report on developing the browser engine for Mozilla’s Firefox browser in Rust has been published by Anderson et al. in 2020 [2]. Evans et al. (2020) evaluated the use of unsafe code and proposed changes to the compiler to signal to developers that their code may have unsafe features [24]. Emre et al. (2021) investigated the automatic translation of C code to Rust code with the goal of improving safety [22].

Xu et al. (2020) studied over 180 real-world Rust common vulnerabilities and exposures (CVEs) and found that all memory safety bugs required unsafe code, backing up Rusts claim of memory safety [96].

In 2022, Coblenz et al. found that using the garbage collection library *Bronze* (which is discussed in chapter 3 of this thesis) significantly sped up the time subjects took to complete a programming task. They found ownership, borrowing and lifetimes to be the main areas users struggled with [17].

### Garbage collection and memory management

To be able to compare Rusts approach to memory management to the approach of other languages, a look at garbage collection is necessary. The concept of garbage collection was introduced by John McCarthy around 1960 to simplify memory management in the LISP programming language [48]. Subsequently, Donald Knuth (1968) discussed differences between reference counting and garbage collection and pointed out limitations of both approaches in volume 1 of his standard work *The Art of Computer Programming* [43]. Since then, a large number of research on a variety of garbage collection approaches have been published: While listing all of them is beyond the scope of this thesis, the following are worth mentioning: Wilson et al. (1992) describe different garbage collection techniques for uniprocessor systems [95], serving as a well organized summary of approaches. In 2004, Detlefs et al. introduced the *Garbage First Garbage Collector* [33], a high throughput collector optimized for multiprocessor systems. It has since served as a reference for concurrent collectors and has been the default garbage collector of the Java Virtual Machine (JVM) reference implementation *OpenJDK* since version 9.

In 2004, Bacon et al. presented their “Unified Theory of Garbage Collection”, which shows that the two core approaches to automatic memory management – reference counting and tracing collection – are algorithmic duals of each other, and that all other commonly used techniques are hybrids of these [5].

Rust does not include a garbage collector for performance reasons, which is why the performance of garbage collectors also need to be looked at. Benjamin Zorn published an analysis of the cost of conservative garbage collection in 1993 [98], Hertz and Berger (2004) aimed at settling the debate of performance of automatic versus manual memory management. They found that under certain conditions, garbage collection can outperform manual memory management however only at a significant memory overhead [30]. The conclusions on

the performance of garbage collection diverge considerably and appear to be dependent on the specific use case indicating that garbage collection is still not a universal solution.

### 1.2.2 Blog posts

Over the past years, a large number of blog posts on the Rust language have been published online. Some Rust team members give insight on the development and technical issues, such as Aria Beingessner pointing out problems with unsafe pointer types [8]. Other core members have shared their visions of what Rusts use case, its target audience, and even its marketing strategy should be [31,39].

Several authors not part of Rust team have written blog posts and tutorials about the language as well. Most of them have the goal of explaining a concept or even to solve a specific problem in Rust. An example of this is Gui Andrade talking about storing unboxed trait objects [3].

Another category of blog posts commonly found is the comparison to other languages, often with the aim of introducing Rust to developers with a certain background. Comparisons to a variety of languages exist, among those are OCaml [23], C and C++ [23,91], and Haskell [56].

Unfortunately, not all blog posts on Rust are completely accurate. Due to the often anonymous nature of blogs, it is not always possible to identify the author. Also, as opposed to the academic research presented above, blog posts are generally not peer-reviewed, though some blogs provide a feedback mechanism through comment fields. A blog post on the website of a consultancy comes to the conclusion that “[..] Rust has Garbage Collection, and a Fast One” [87]. This is – as will be shown in this thesis – not an accurate description of Rust, which does not have a garbage collector. Another blog post compares execution speed of a Kotlin and a Rust program, coming to the conclusion that Kotlin is faster than an optimized Rust program [49]. However, a closer look shows that the methodology is not sound which is pointed out by users in the comments.

### 1.2.3 Official Rust books

In addition to the documentation and the source code of the Rust language, the Rust team publishes and maintains official books online. The most notable one is by Steve Klabnik and Carol Nichols, titled “The Rust Programming Language” [42]. It was also published in print and serves as an introduction to Rust, describing aspects of the language [59].

The official Rust reference is available as an online book [73]. More official books are available digitally on the topics of compiler development, asynchronous programming, unsafe Rust, and more [61,78,79].

### 2.1 Syntax and language features

This section aims to give an overview of the syntax of Rust. The focus is on concepts that are used in the following chapters and relevant to the topic of memory management. To keep examples short, the code listings only show fragments of full programs. In particular, the `main()` function is omitted when not needed for the example.

#### 2.1.1 Variables and values

In Rust, values are accessed through variables. A variable is declared using the `let` keyword, followed by the name of the variable. The equal sign (`=`) serves as the assignment operator. On its right the desired value for the variable is stated. Listing 1 creates a variable named `var` and assigns the value `42` to it. The semicolon (`;`) concludes the assignment statement.

```
1 let var = 42;
```

Listing 1: Declaring a variable and assigning a value to it

Variables are immutable by default. Once a value is bound to a variable, it cannot be changed. In many programs, however, it can be useful to have a variable that is capable of holding different values over time. To enable this, Rust offers mutable variables. A variable is declared as mutable by adding the `mut` keyword as shown in Listing 2.

```
1 let mut x = true;
```

Listing 2: Declaring a mutable variable

Listing 2 creates a mutable variable `x` and assigns the value `true` to it.

## 2.1.2 Types

### Type inference

Rust is *statically typed*, meaning that the type of a variable has to be known at compile time. The compiler is able to infer the type of most variables. In Listing 2, even though it was not explicitly stated, variable `x` is of type `bool`, the Boolean type. It is possible (and sometimes even necessary, as shown later on) to explicitly annotate the type of a declared variable, as is shown in listing 3.

```
1 let x: i32 = 21;
2 let y = 42i64;
```

Listing 3: Function definition

Two different notations for type annotations exist: The type of variable `x` is specified by adding a colon (`:`) after the variable name, followed by the type. Variable `y` uses a postfix notation to specify the type, in this case the 64-bit integer type `i64`.

### Scalar types

Scalar types, sometimes called *primitive types*, are types representing only a single value [59, p. 36]. In Rust, four scalar types are implemented: Integers, floating-point numbers, Boolean values, and characters.

The integer and floating-point types are provided in different sizes. Integers exist both signed and unsigned, with sizes ranging from 8 to 128 bit, as well as with the `usize` type having system-dependent size. Two floating point types are provided: `f32` and `f64` with 32 and 64 bit size, respectively. All basic mathematical operations for number types are supported: Addition, subtraction, multiplication, division, and the remainder operation.

The character type `char` serves to represent a single character. Character literals are indicated with opening and closing single quotes (`'`). Rust's `char` type uses 4 bytes of memory. In comparison, a `char` in C takes up only one byte in order to be able to represent ASCII-encoded characters. By using 4 bytes, Rust's character type is able to represent any Unicode scalar value. Because of this, symbols of the Chinese or Japanese language, accented characters, and even emojis are valid character values in Rust.

### Generic data types

Generic data types are data types which are not fully specified but are represented in parts or in full by type variables. These type variables may be required to provide certain operations. An example of a generic data type is the `Option<T>` type, where `T` is called a *type variable*. The type `Option<T>` is defined with two constructors, `Some(T)` and `None`, which provide the possibility to store a value of type `T`, or no value, if no value exists. Similar to the `Maybe` a type found in Haskell it can be used as a return type for a function or computation, where – if the computation fails – no value will be returned. Listing 4 shows an example use of the type `Option<T>`.

```

1  let mut rng = rand::thread_rng();
2  let num: i32 = rng.gen::<i32>();
3
4  let half: Option<i32> =
5      if num % 2 == 0 {
6          Some(num / 2)
7      } else {
8          None
9      };

```

Listing 4: Calculating a value with type `Option<i32>`

The first two lines of Listing 4 compute a random integer value which is stored in the variable `num`. In line 4, a new variable named `half` of type `Option<i32>` is declared. It is then initialized with the result of a computation: If `num` is even, it is divided by two, and the result of this operation is returned, wrapped in the `Some` constructor of the `Option` type. If `num` is odd, the constructor `None` is used, no value is being returned. Other generic types include `Result<T, E>`, which is used to hold a value or an error, or the vector type `Vec<T>`. The vector type is discussed in detail in section 2.1.2.

### Tuples and arrays

Tuples and arrays are compound data types. They are used to combine several values into a single data type.

A tuple serves as a collection of values of different types. It can hold any number of values, the length, however, is fixed. Tuples are created using parentheses: An opening parenthesis is followed by a comma-separated list of values, followed by a closing parenthesis. The individual values can be accessed using a period (`.`), followed by the index of the element to be accessed. Alternatively, a tuple can be *destructured* using pattern matching, breaking it up into its individual parts. Listing 5 illustrates the use of tuples.

```

1  let tuple = (42, 'x', true, 12.0);
2  let fortytwo = tuple.0;
3  let (a, b, c, d) = tuple;
4  println!("{}", {}, fortytwo, b);

```

Listing 5: Use of tuples

In line 1, a variable called `tuple` is declared and initialized with a tuple made up of four values of different types. In line 2, the period notation is used to access the first value of the tuple and assign it to a new variable `fortytwo`. Line 3 shows an example of *destructuring*: Four variables `a`, `b`, `c`, and `d` are declared and initialized with the individual values of the tuple. Their types are inferred by the compiler. The `println!` instruction used here is a *macro* and will be explained in section 2.1.4. For this example it suffices to know that it prints the variable's values to the console. The output of this program is `42, x`.

A particularly interesting tuple is the empty tuple `()`. It is also called the *unit* type [57]. Only one value – the empty value `()` – of it exists. Therefore, the type carries no information and can be used as a placeholder when no meaningful value exists, for example as a return value. This will be described in section 2.1.3 [82].

Arrays are another way of storing multiple values in a single variable. The length of an array is fixed. Arrays are zero-indexed, meaning that the first element of an array has index

0. As opposed to tuples, all values in an array have to be of the same type. To create an array, the list of values separated by commas, is placed between a pair of square brackets (`[]`). Elements in an array are accessed using `a[i]`, where `a` denotes the array, and `i` denotes the desired index. Listing 6 shows an exemplary use of arrays.

```
1 let arr = [4, 8, 15, 16, 23, 42];
2 let x = arr[3];
3 println!("{}", x);
```

Listing 6: Use of arrays

In Listing 6, an array named `arr` consisting of six integers is created. In line 2, a new variable `x` is created and initialized with the value at index 3 of `arr`. When executed, this program prints 16 to the console, as indices of arrays are zero-based.

## Vectors

The standard library contains implementations of commonly used data structures each of which can carry multiple values and offer additional functionality over tuples and arrays, such as dynamic resizing. These data structures are called collections. The standard library contains a number of collections. In the scope of this thesis, however, only an understanding of vectors and strings is required.

Arrays can only be used when the amount and size of their contents are known at compile time, or when a collection of a fixed size is desired. In many programs, though, the number of elements to store can only be determined at runtime, e.g. when user input is processed. Rust provides a vector type that is resizable at runtime, allowing the programmer to store a dynamic number of values in a single variable. This vector type is called `Vec<T>`. Vectors are *generic types*, with `T` being a type variable. This means that the `Vec` type is capable of holding elements of any type `T`. When referring to a specific vector, the `T` is replaced by the type of the elements of that given vector.

An empty vector is created using the `new()` function provided by the type. Listing 7 shows how it is used:

```
1 let v: Vec<i32> = Vec::new();
```

Listing 7: Creating a vector using `new`

Two things from Listing 7 stand out: The type annotation in angled brackets (`<>`), as well as the double colon (`::`) in front of the `new()` function.

The angled brackets are necessary because the compiler cannot infer from the call to `new` which type the elements of the vector will be. Hence, explicit type annotation is required which is placed inside the angled brackets.

The double colon (`::`) indicates a *type-associated function* [59, p. 16]. This means that the function is implemented by the type itself, as opposed to a particular instance of a `Vec`. This is similar to static member functions in C++, which also do not require an object of a class to be instantiated in order to call the method [64].

Similar to `new`, `Vec` also provides a `from` function, which allows the programmer to create a vector from an array of elements. In this case, the type of the elements can be inferred. Listing 8 shows the creation of a vector from an integer array.

```
1 let w = Vec::from([4, 8, 15, 16, 23, 42]);
```

Listing 8: Creating a vector using `from`

Vectors can grow: Elements can be added to a `Vec` by calling its `push` method which adds the passed value to the end of the vector. Methods to remove elements exist as well: The function `remove()` takes an index as its argument and removes the element at the given position, moving all following elements to the front. The vector has to be declared mutable for these modifications to be possible. Listing 9 shows the mutability of vectors:

```
1 let mut v = Vec::from([4, 8, 15, 16, 23, 42]);
2 v.push(4);
3 v.push(8);
4 v.remove(4);
5 println!("{:?}", v);
```

Listing 9: Adding and removing elements from a `Vec`

In line 1, a vector named `v` is created from an array. The two integers 4 and 8 are pushed and thereby added to the back of the vector. In line 4, the element at index 4 is removed. Note that `remove` does not remove the integer 4, but the element at index 4, in this case the integer 16. The array is printed to the console in line 5, resulting in the output `[4, 8, 15, 16, 42, 4, 8]`.

### String types

The core Rust language comes with the string slice type `str`. It is an immutable, static string of UTF-8 encoded characters. In the program code, a value of `str` is delimited using double quotes (`"`). This type lacks flexibility in practice, as the length of a string is often not known at compile time. The standard library of Rust offers another string type: `String`. It can be mutable (provided that the respective variable is mutable) and it is dynamically resizable, similar to the `Vec` type. A `from` function exists for `String`, allowing the creation of a `String` variable from a string slice (`str`). Appending to a string is done using the `push_str` method.

### 2.1.3 Functions and methods

All executable Rust code must be placed inside functions and methods. They are used as building blocks to make code more readable, maintainable, and modular. Some programming languages distinguish between routines that return a value (often called *functions*) and routines that do not return a value (often called *procedures*). Rust does not make this distinction. It does, however, distinguish between functions and methods. Functions can take parameters as input, and return values to the caller. Methods are functions which are defined in the context of a data structure introduced by the keywords `struct` or `enum`. They always operate on a specific instance of a data object. The first parameter of a method is always a reference to the object whose value is being used. That parameter is called `self`. Apart from that, methods work exactly like functions.

#### Function definition

A function definition in Rust begins with the keyword `fn` which is followed by the function name and a set of parentheses, within which the function parameters are given. Function

parameters are optional, therefore the parentheses may be empty. After the parentheses, an arrow `->` is used to prefix the return type. Curly braces delimit the function body. Listing 10 shows a function definition:

```
1 fn a_function() -> () {  
2     println!("Hello, Rust!");  
3 }
```

Listing 10: Function definition

The function `a_function` has the return type `()`, also called the *unit* type. It is used here because the function does not have any meaningful value to return to the caller. The explicit indication that a function's return type is `unit` can be elided, as is shown in Listing 11:

```
1 fn a_function() {  
2     println!("Hello, Rust!");  
3 }
```

Listing 11: Function from Listing 10, but with elided return type

### Calling functions

A function is called by stating its name, followed by a set of parentheses and a semicolon. The entry point for a program is the `main()` function. Listing 12 shows calling the function defined in Listing 10 from inside the `main()` function.

```
1 fn main() {  
2     a_function();  
3 }
```

Listing 12: Calling the function from Listing 10

Function parameters are variables in the function signature. When a function is called, values for these variables are passed to it, and they can be used in the function body. Once passed, the concrete values are called *arguments*. The types of the function parameters have to be declared in the signature: Within the parentheses after the function name, the name and type of each parameter need to be stated, separated by colons. Multiple parameters can be defined, their declarations are separated by commas. Listing 13 shows the signature of a function with two arguments.

```
1 fn add_int(x: i32, y: i32) -> i32
```

Listing 13: Definition of a function with multiple parameters

The signature of the function `add_int` from Listing 13 indicates that the function takes two arguments of type `i32` named `x` and `y`, and returns a value of type `i32`. To understand how values are returned a look at the body of a function is necessary: The body consists of a set of statements with an optional expression at the end [59].

A statement represents code that performs an action but does not return a value. Contrary,



an expression is evaluated to a value. Assignments, as seen in Listing 1, are statements. Calling functions and macros are expressions, as are code blocks. The last expression inside a code block is evaluated, its result is considered the value of that block, and is returned to the caller.

Listing 14 completes the function `add_int` introduced in Listing 13 with a return value.

```
1 fn add_int(x: i32, y: i32) -> i32 {
2     x + y
3 }
```

Listing 14: Adding a return value to the function from 13

In line 2 of function `add_int`, variables `x` and `y` are added. This is an expression and evaluates to a single value which is then returned by the function. Note that line 2 is not concluded with a semicolon. Adding this semicolon would change the expression to a statement, therefore not returning the intended value from the function. The semicolon `;` discards the value returning the `unit` type `()` instead.

Another way to return values in Rust is using the `return` keyword. Explicit returning follows the syntax `return e` where `e` is an expression. The value of this expression will be returned. Explicit return statements can be placed after statements at any point within the function body, causing the function to immediately return from that point with the desired value. Any code following it will not be executed. By convention, explicit `return` is used for early returns from functions in cases of errors or unexpected results [9].

### Generic types in function signatures

In order to reduce the duplications of code, it is possible to write functions that are capable of taking arguments of any type. To do this, a generic type parameter replaces the data types of the arguments or the return value of the function. Additionally, after the function identifier the type parameter has to be specified inside angled brackets `<>`. Listing 15 shows the signature of a function using generic data types:

```
1 fn largest_element<T>(v: Vec<T>) -> T
```

Listing 15: Defining a function using generics

The function `largest_element` from Listing 15 takes a vector containing elements of an arbitrary type `T` as its argument and returns a value of the same type `T`.

#### 2.1.4 Macros

Rust offers a sophisticated macro system. Klabnik (2019) describes macros as “code that writes other code” [59]. Macros are a way of providing convenience functionality to the programmer while hiding complexity. A macro looks similar to a function, but its name ends with an exclamation mark `!`. Macros allow the creation of *variadic interfaces*. A variadic interface can take any number of arguments, as opposed to a function, where the number of arguments is defined through the function signature. An example of a macro with an arbitrary number of arguments is the `println!` macro. The first argument passed to `println!` is a string literal containing at least one set of curly braces `{ }`. This string is called a *format string*. Additional arguments are then used to replace the curly braces in the

format string. The resulting string is then printed to the console. Listing 16 shows the use of the `println!` macro:

```
1 let i = 42;
2 let c = 'x';
3 let s = "Rust";
4
5 println!("The values are {}, {}, and {}.", i, c, s);
```

Listing 16: Using the `println!` macro to print variable values

In Listing 16, three variables are declared and initialized with arbitrary values. In line 5, the `println!` is used to print the value of these variables at desired positions of the format string passed to the macro. The output of this code is `The values are 42, x, and Rust.`

### 2.1.5 Traits

Most programming languages offer a mechanism to define shared behaviour between types in order to avoid the repetition of code. In Rust, this mechanism is called traits. It allows providing a general type signature for the implementing data types. The trait system was inspired by type classes as found in the Haskell programming language [89]. The discussion of traits focuses on the aspects relevant for understanding memory management in Rust. A trait abstractly defines common behaviour that all implementing types must provide. This way, the same methods can be called on different types.

In Rust, traits serve two main purposes. The first one is the grouping of data types by their functionality. The other purpose is to provide the compiler with additional information for optimizations. This aspect is especially relevant for traits that can not be implemented by the user, but are usually implemented by the compiler, such as the `Copy`, `Sized`, `Unpin` and other “marker” traits.

In function signatures, traits serve as type constraints. In a function signature using traits, no information about the specific type is available, except for the fact that it implements methods defined by the trait. In Rust, these constraints defined through traits are called *trait bounds*. Trait bounds can be used to ensure that arguments of a function have a certain functionality that is required to execute the function.

Listing 17 shows an example of trait bounds. For this example, consider a trait called `Metadata` to exist. It provides a method called `meta()` that returns a human-readable string representing the metadata of an object. For example, metadata of a picture could be the time and location where the picture was taken.

```
1 fn print_date<T: Metadata>(object: T) {
2     let meta_string = object.meta();
3     // ...
4 }
```

Listing 17: Function with trait bounds

The example in Listing 17 shows the signature and parts of the body of a function called `print_date`. It takes an object of a generic type `T` that implements the `Metadata` trait as its argument. The trait bound is placed in the angle brackets after the declaration of the

type variable  $\mathbb{T}$ , separated from it by a colon. Through the trait bound the compiler knows that the object passed must provide an implementation of the method `meta()` which can therefore be called in line 2.

## 2.2 Ownership

Ownership is often described as one of the core principles of Rust [59, 89]. Ownership is built upon three rules, as specified in the official Rust programming language book [59, p. 61]:

- Each value in memory is owned by a variable which is called its *owner*.
- At a given time, every value has exactly one owner.
- If the owner goes out of scope, the value is dropped and its backing memory is released.

These three rules are checked at compile time. The owner of a value may change during the execution of the program. This is called moving of ownership. The ownership of a value is tightly coupled with the scope of its owning variable.

### 2.2.1 Variable scope and access

The scope of a variable describes a range within the code of a program where its identifier is valid [59, p. 62]. The scope of a variable in Rust starts with its name declaration and ends at the end of the declaring block which is characterized with a closing curly brace. Rust uses *lexical scoping*, also referred to as *static scoping* [93]. Brooks et al. (1982) describe lexical scoping as “ALGOL-style” scoping [13]. Lexical scoping can be checked statically, i.e. at compile time. Variables can be *shadowed*, i.e. a variable can have the same name as a variable declared in the same or an outer scope. An example of shadowing is shown in Listing 18. Values associated with a shadowed variable cannot be accessed through the shadowed variable name [90]. Variables declared in outer blocks can however be accessed from inner blocks by other means [58].

Before a variable goes out of scope, Rust implicitly calls a function called `drop` with the variable as its only parameter. The actions required to return the memory allocated to that variable are implemented in this function. [59]. Apart from freeing memory, `drop` can also be used to release other resources such as network connections or files. This approach of deallocating memory at the end of its lifespan is similar to the *Resource Acquisition is Initialization (RAII)* pattern made popular by the programming language C++ [60, 77].

Listing 18 shows how variables can be scoped, shadowed and dropped based on the two variables `x` and `y`. The curly braces in line 1 and 9 are deliberately included in this example to illustrate the start and the end of the outermost scope.

```
1  {  
2      let x = 42;  
3      {  
4          let x = 21;  
5          let y = 0;  
6      }  
7      println!("x: {}", x);  
8  }
```

Listing 18: Variable scopes

Variable `x` comes into scope with its name declaration in line 2 and is valid from thereon. It is initialized with value 42. In line 4, within a separate code block, `x` is shadowed, i.e. a new variable with the same name is declared, and initialized with value 21. This inner scope ends with the closing curly brace in line 6. Within the entirety of this scope, the value associated with variable `x` is 21. At the end of the inner scope, value 21 is dropped. Back in the outer scope, the name `x` is still valid and associated with value 42. It can therefore

be used in the `println!` macro in line 7. Variable `x` goes out of scope with the end of the outer scope in line 9.

Variable `y` is declared and initialized in the inner scope at line 5. This variable is only valid in this block, so it goes out of scope in line 6 and its value is dropped. Attempting to read its value after line 6 would result in a compiler error.

For reading the value of a variable, equally important to the variable being in scope is that it is actually holding a value. This is not the case if it has never been initialized, or if its value has been moved. These two possibilities shall be illustrated by examples. Listing 19 shows an example of an uninitialized variable.

```
1 {
2     let x: i32;
3     println!("{}", x); // error
4 }
```

Listing 19: Accessing an uninitialized variable

```
error[E0381]: borrow of possibly-uninitialized variable: `x`
--> uninit.rs:3:19
|
3 |     println!("{}", x);
|                   ^ use of possibly-uninitialized `x`
```

Listing 20: Compiler output generated by compiling Listing 19

Listing 20 shows the compiler error that arises when trying to compile the code from Listing 19. The compiler does not allow the use of the uninitialized variables, as accessing them could lead to undefined behaviour [70]. This way the compiler enforces that any valid variable to be read has been initialized according to the specific needs of its type prior to using it.

The second reason why a variable might not have a value associated with it is because its value has been moved. Move semantics are the topic of the next section.

C and C++ have no builtin way of tracking whether a variable has been initialized and thus allow to read the values of uninitialized variables possibly resulting in undefined behaviour.

### 2.2.2 Moving of ownership versus copying values

Rust differentiates between statically and dynamically sized types. Statically sized types have a fixed size in the sense that the size is already known at compile time. In contrast, the size of a dynamically sized type can only be determined at runtime, as it is dependent on parameters specific to the type such as the length of a string of characters.

As most other languages, Rust has two areas of memory: Stack and heap. Values on the stack can only be of fixed size. Values on the heap may be of either variable or fixed size. Storing values on the heap is considered more expensive than storing values on the stack since in order to allocate memory on the heap the use of the memory allocator of the operating system is necessary. This is why Rust disallows copying data which is of variable size. However, instead of resorting to the use of pointers, as C and C++ do, Rust has the concept of *moving ownership* which means that the value to be moved is disassociated from its previously owning variable and associated with a new variable.

Listing 21 makes use of the `String` type to illustrate moving of ownership. The dynamically sized type `String` is not copied for efficiency reasons, very much like in C. The approach in C differs from the one in Rust, as Rust does not solely rely upon references but instead moves ownership of values.

```

1  let s1 = String::from("Rust");
2  let s2 = s1;
3  println!("{}", s2);
4  println!("{}", s1); // error

```

Listing 21: Move

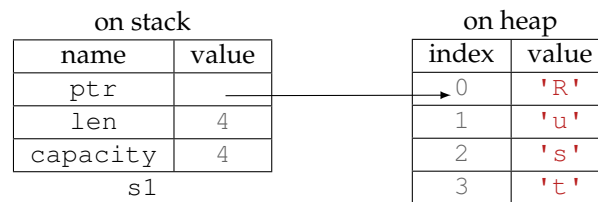
```

1  let x = 42;
2  let y = x;
3  println!("{}", y);
4  println!("{}", x);

```

Listing 22: Copy

When `String` `s1` is created in line 1 of Listing 21, memory is allocated on the heap. However, not the entire data structure is stored on the heap: For a variable of the `String` type, additional metadata – its length and its capacity – is stored on the stack along with a pointer to the memory address of the segment allocated on the heap. Figure 2.1 shows the representation of the `String` variable `s1` with value `"Rust"` from Listing 21 in memory. The metadata, along with the pointer to the memory address returned by the allocator are stored on the stack, as depicted on the left side of the figure. The actual characters making up the string are stored on the heap (right side of the figure) [81].

Figure 2.1: Memory representation of the `String` `s1` from Listing 21

When `s1` is assigned to `s2` in line 2 of Listing 21, Rust does not copy data, as this is could potentially be expensive at runtime if the heap data was large. Instead, `s2` refers to the memory location of `s1` while `s1` is considered unbound from thereon. The Rust compiler considers `s1` to be no longer accessible after assigning it to `s2` [59], as the value of `s1` was moved to `s2`. The motivation behind the move semantics is the prevention of invalid or dangling references [75].

Listing 22 contains the same statements as Listing 21 but operates on statically sized data. Hence, the semantics are different: For the integer type `i32`, both variables `x` and `y` can be used after copying the value of `x` to `y`. The reason for this is that pushing values onto the stack is not considered an allocation, as stack memory is allocated to the process by the operating system as soon as the process is created [59, p. 62]. As a consequence, values on the stack can be copied freely without necessitating a call to the memory allocator of the system.

The compiler determines whether a type can be copied by checking if it implements the `Copy` trait. Examples of types implementing the `Copy` trait are all primitive data types, such as integers (`i32`), floating point types such as `f64` and the Boolean type `bool`. It is not possible for the programmer to implement the `Copy` trait for an arbitrary type. A user-defined, compound type can only implement `Copy` if it is made up solely of `Copy` types, i.e. primitive types [69].

A rule of thumb regarding whether a type implements the `Copy` trait is the size of a pointer:

If the type requires more space than a pointer, it usually does not implement the `Copy` trait. If an object takes up less space than a pointer, copying the object can be implemented in an efficient way, and so the `Copy` trait usually is implemented for these types. This consideration is part of the design decisions taken by the Rust team on whether a type should implement the `Copy` trait.

In future examples integers are used to represent `Copy` types and strings are used to represent non-`Copy` types.

### 2.2.3 Clone

In some programs it is desirable to have the same value assigned to more than just one variable. The Rust developers have anticipated this need and therefore provide a function called `clone` for most builtin types [68] which do not implement `Copy`. Listing 23 shows the use of this function: A `String` variable `s1` is declared and initialized. A second `String` variable `s2` is created, and initialized with a copy of the original string, returned by the call to `s1.clone()`. The two strings are stored in different memory areas and are completely independent from each other.

```
1 let s1 = String::from("Rust");
2 let s2 = s1.clone();
3 println!("{}", s2);
4 println!("{}", s1);
```

Listing 23: Cloning values

The above set of statements prints `"RustRust"` to the console. Since cloning can negatively affect the performance of a program, it is never done implicitly and is always visible in code with the `clone` function [35].

### 2.2.4 Passing ownership across function boundaries

Passing a variable to a function moves ownership into the function. Listing 24 shows how ownership is transferred through function calls.

```
1 fn take_ownership(s: String) {
2     println!("{}", s);
3 }
4
5 fn makes_copy(i: i32) {
6     println!("{}", i);
7 }
8
9 fn main() {
10    let s = String::from("Rust");
11    take_ownership(s);
12
13    let x = 5;
14    makes_copy(x);
15 }
```

Listing 24: Passing ownership

Both the functions `take_ownership` and `makes_copy` have only one function parameter, an `i32` and a `String`, respectively. They both print the value of the variable to the console and return nothing.

A variable passed into a function comes into scope the moment the function body is entered, and is valid until the closing brace. At the end of the scope the respective value is dropped and from thereon unavailable for reading. In the case of the function `makes_copy`, a copy of the value assigned to variable `x` is made and passed to the function when it is called in line 14 therefore leaving `x` accessible for reading after the function has been executed. Since `String` does not implement `Copy`, the function `take_ownership` does not create a copy of the value passed to it when it is called in line 11. Ownership is moved, and therefore the compiler marks variable `s` as unbound and does not permit read access to it afterwards.

In a similar manner, ownership can be transferred by using return values. When a function returns a value, the ownership of that value is then transferred to the caller of the function. Listing 25 illustrates this:

```

1  fn gives_ownership() -> String {
2      let s = String::from("Rust");
3      return s
4  }
5
6  fn reverse(s: String) -> String {
7      let reversed = s.chars().rev().collect();
8      reversed
9  }
10
11 fn main() {
12     let string1 = gives_ownership();
13     let string2 = reverse(string1);
14     println!("{}", string2);
15     println!("{}", string1); // error
16 }
```

Listing 25: Passing ownership through return values

The function `gives_ownership` takes no arguments and returns a `String`. In the body of the function, a `String` variable is created and subsequently returned. When the function is called in function `main` in line 12, the variable `string2` takes ownership of the value of the `String` returned by the function `gives_ownership`. It is important to note that the practice of declaring a variable and subsequently returning it in the following line is generally not considered to be good Rust style. It is deliberately being used in this example to make the code clearer.

The function `reverse` takes a `String` as its only argument and returns a `String` with the characters in reversed order. In the body, the `String` passed to the function is reversed by calling `s.chars().rev().collect()` which simply is a chain of method calls. Its result is assigned to a new variable named `reversed` which is subsequently returned. The function `reverse` is called in line 13 with the previously initialized `String` `string1`. The return value is assigned to a new variable `string2`, which takes ownership of the value returned by the function. Because of the move semantics described above, `string1` is considered unbound after line 13 and can not be accessed afterwards which is why accessing it in the `println!` macro in line 15 will result in a compiler error.

Looking at the example in Listing 25 it becomes clear that in order to use a value after a



function takes ownership of it, the function has to pass back ownership by returning the value. Passing ownership twice clutters the code and becomes cumbersome if a function needs to change the value of more than one variable. Listing 26 shows how this is achieved by returning tuples:

```
1 fn main() {
2     let vector = Vec::from([1, 2, 3, 4]);
3
4     let (returned_vec, computed_len) = compute_length(vector);
5     println!("The length of the vector is {}.", computed_len);
6 }
7
8 fn compute_length<T>(v: Vec<T>) -> (Vec<T>, usize) {
9     let length = v.len();
10    (v, length)
11 }
```

Listing 26: Returning ownership and additional return value

The function `compute_length` is being called from the function `main` which subsequently uses its return value in line 5 in order to print the length of the vector to the console. `compute_length` takes ownership of the vector, and returns a tuple of the computed length and the vector itself to ensure the vector is not dropped and can be used afterwards.

### 2.2.5 Ownership in C and C++

Both C and C++ have no concept similar to ownership. They both require the programmer to allocate and free memory by hand and allow for freely making copies of values, possibly leading to bugs [1].

## 2.3 Borrowing

Rust also offers a mechanism of using values without taking ownership of them. This mechanism is called *borrowing* [59, p. 71]. Borrowing augments the ownership model by introducing references to values. These references are tied to the values themselves and as such may only exist as long as the values they are referring to. The *borrow checker*, part of the Rust compiler, is tasked with validating these references: Every reference is associated with a *lifetime* which will be discussed in detail in Chapter 2.4. Until then, lifetimes will be written as `'a` and can be ignored.

Listing 27 modifies the signature of the function `compute_length` from Listing 26 to accept a reference to a `Vec`, denoted by the ampersand (`&`) in front of the argument type. The function now only returns the computed length; ownership of the vector remains with the calling function.

```

1 fn main() {
2     let vector = Vec::from([1,2,3,4,5,6]);
3     let length = compute_length(&vector);
4     println!("The length of the vector is {}", length);
5 }
6
7 fn compute_length<'a,T>(v: &'a Vec<T>) -> usize {
8     v.len()
9 }
```

Listing 27: Computing length with reference to vector

Function `compute_length` is called in line 3. Instead of passing the vector as the function argument, a reference to it is passed, indicated by the ampersand (`&`). Because of this, the value of `vector` can be used without taking ownership of it. When variable `v` goes out of scope in line 9 its memory is released. Since this variable only held a reference to the vector, the memory of the vector will not be released. The ownership of `vector` always remained with the `main` function, hence the function `compute_length` is not required to give ownership of the value back to the caller.

### 2.3.1 Mutable references

Just as variables, references are immutable by default: Attempting to modify a borrowed value results in a compiler error. Rust allows references to be mutable, with one key restriction: Only one mutable reference to a particular value is allowed at any time. This restriction serves to rule out the possibility of *data races*. A data race occurs when the following three conditions are met [59, p. 72]:

- At least two references access the same memory at the same time.
- At least one of the memory accesses is a write access.
- No synchronization mechanism is in place to manage access to the data.

As soon as one of these conditions is removed, a data race is no longer possible. Data races can cause undefined behaviour and may be difficult to debug. Rust prevents data races by not allowing code with multiple mutable references to compile in the first place. Having a mutable and an immutable reference to a value at the same time is also prohibited by the compiler.

Listing 28 shows the use of a mutable reference:

```

1 fn main() {
2     let mut s = String::from("Hello");
3     append_newline(&mut s);
4 }
5
6 fn append_newline<'a>(s: &'a mut String) {
7     s.push_str("\n");
8 }

```

Listing 28: Appending to a `String` with a mutable reference

The function `append_newline` takes a reference to a `String`, and appends the new-line character `"\n"` to it. This is done with the `push_str` method which is provided by the `String` type. In the `main` function, a mutable string `s` is created and a reference to it is passed to the `append_newline` function.

Creating two mutable references to the same value is not allowed by Rust per the above rules which is why compiling the code in Listing 29 results in a compiler error:

```

1 let mut s = String::from("Rust");
2
3 let reference1 = &mut s;
4 let reference2 = &mut s; // error

```

Listing 29: Attempting to create multiple mutable references

### 2.3.2 References and mutability in C and C++

Variables and references in C and C++ are mutable by default. Both C and C++ specify a `const` keyword serving as a type classifier. A variable that is declared as `const` cannot be changed by the programmer after initialization [62].

The C standard does not require compiler implementations to maintain a record of the number of pointers to an area of memory. Furthermore, by allowing pointer aliasing, i.e. having multiple pointers pointing to overlapping parts of the same buffer [35, p. 2], the arbitrary creation of copies, and the use of pointer arithmetic it becomes impossible to keep track of pointers.

The combination of variables being mutable by default, the lack of stringent rules for the creation of copies as well as allowing pointer aliasing make C and C++ susceptible to data races.

### 2.3.3 Dangling references

Languages with manual memory management such as C and C++ allow the programmer to create *dangling pointers*. A dangling pointer is a pointer that references a location in memory that is no longer valid. This occurs when memory is freed but its corresponding pointer is not removed. Listing 30 – written in C – creates a dangling pointer. This is not something one would deliberately attempt, the example serves to show that creating a dangling pointer in C can be easily done. Such a bug is not prevented by the compiler and may happen accidentally.

```
1 char *s1 = malloc(BUF_SIZE);
2 char *s2 = s1;
3
4 free(s1);
5 printf("Value: %d\n", s2); // error, undetected
```

Listing 30: Creating a dangling pointer in C

In line 1 of Listing 30 a pointer to a character is declared and initialized. The `malloc` function, provided by the C standard library, is called to allocate memory. It takes the number of bytes to allocate as its argument. Here, it is called with the constant `BUF_SIZE` which for the sake of this example we assume to be an arbitrary integer  $> 0$ . `malloc` requests memory from the operating system and returns a pointer to that memory block. In line 2, a second character pointer is declared, and is initialized with a copy of the pointer `s1`. The pointers `s1` and `s2` now point to the same portion of memory. In line 3, `free` is called and releases the memory currently pointed to by `s1`. Variable `s2` still points into the memory originally allocated in line 1, which has just been returned to the operating system. Therefore, accessing `s2` in line 4 leads to an error when the program is executed. This error, however, is not detected by the compiler. It can be avoided by, for example by assigning `NULL` to `s2`. However, it is up to the programmer to do so. In Rust, such errors are not possible, as the example in Listing 31 shows.

```
1 fn dangling_reference<'a>() -> &'a String {
2     let s = String::from("Rust");
3     &'a s // error
4 }
5
6 let s = dangling_reference();
```

Listing 31: Attempting to create a dangling reference in Rust

Listing 31 attempts to create a dangling reference by creating a variable with a value and a subsequent reference to that value. At the end of the scope, the variable is dropped by regular means but the reference is attempted to be returned from the function. The function `dangling_reference` has return type `&String`, that is a reference to a `String`. In line 2, a `String` variable `s` is created. Returning a reference to it in line 3 fails: Since in line 4, `s` goes out of scope its value is dropped. Therefore, the reference to it is invalid afterwards.

## 2.4 Lifetimes

Lifetimes augment references, describing the region of program code where a reference can be used. Therefore, they are essential for validating references at compile time. When a value is borrowed, a reference to that value is created. This reference can never be valid for longer than the value itself. In Rust, the region where a variable is valid is called its scope as described in section 2.2.1. The scope of a reference is called its *lifetime*.

Lifetimes are a feature not commonly found in other languages. In particular, neither C nor C++ have a similar feature (or any other way of verifying the validity of pointers or references). Due to their unique character, their semantics have been shown to cause programmers new to Rust problems at the beginning [17].

### 2.4.1 Validating lifetimes with the *borrow checker*

In order to guarantee memory safety, the Rust compiler needs to verify that any access to a reference is valid in the sense that the data it references still exists. If a variable references a location in memory that has been returned to the operating system, attempting to access it would result in undefined behaviour. Hence, every reference in Rust is associated with a lifetime, which indicates for how long a reference is valid. The compiler makes use of these lifetimes in order to guarantee that all accesses to references in the code are safe. The part of the compiler responsible for this is called the *borrow checker*.

Listing 32 – adapted from the official Rust book [59, p. 194] – shows an example of an invalid reference. Curly braces are added to explicitly show the end of the outer scope.

```
1 {  
2     let r: &i32;  
3     {  
4         let val = 42;  
5         r = &val; // error  
6     }  
7     println!("{}", r);  
8 }
```

Listing 32: Creating an invalid reference

Line 2 in Listing 32 declares a variable `r` which can hold a reference to an integer value. Initially, no value is assigned to this variable. In a new block, a new variable `val` is declared and initialized with value `42`. In line 5, attempting to assign a reference to `val` to the variable `r` fails. Back in the outer scope, in line 7, the value of `r` is printed to the console. The error message gives a hint why this code is rejected:

```

error[E0597]: `val` does not live long enough
--> invalid_reference.rs:5:17
|
4 |         r = &val;
|           ^^^^^^ borrowed value does not live long enough
5 |     }
|     - `val` dropped here while still borrowed
6 |     println!("{}", r);
|                       ----- borrow later used here

```

Listing 33: Error message produced by compiling Listing 32

The compiler indicates that the “borrowed value does not live long enough”. To aid understanding this error, the lifetimes of the references are visualized in Listing 34.

```

1 {
2     let r: &i32;           // -----+--- 'a
3     {                     //          |
4         let val = 42;      //          |
5         r = &val;          // ----+--- 'b |
6     }                     // ----+   |
7     println!("{}", r);    //          |
8 }                         // -----+

```

Listing 34: Visualizing lifetimes from Listing 32

Listing 34 includes comments to visualize the lifetime of references in a notation used in Klabnik’s official Rust book [59]. The lifetime of the reference held by `r` is annotated with the lifetime identifier `'a`, while the lifetime of reference `&val` is described by `'b`. While these letters seem arbitrarily chosen, it is convention to use `'a`, `'b`, and so forth as lifetime specifiers.

The visualization shows the reason why the code was rejected: The scope of `r` is larger than the reference to `val`. With the closing brace in line 5, variable `val` goes out of scope, dropping the value. As a consequence, any reference to it cannot be valid afterwards.

## 2.4.2 Rust code with lifetime annotations

While Listing 32 uses lifetime annotations in comments for the purpose of visualization, annotating the code directly with lifetimes would not have been valid: Before lifetime identifiers can be used, they have to be declared. This is done in function signatures with the same syntax with which trait constraints are expressed. Signatures of functions that take references as arguments or return references are required to carry *lifetime annotations*. They serve to indicate to the compiler how lifetime parameters of references relate to one another. The following listing attempts to create a function without specifying lifetime parameters for the references:

```

1 fn shorter_vec<T> (x: &Vec<T>, y: &Vec<T>) -> &Vec<T> {
2     if x.len() < y.len() {
3         x
4     } else {
5         y
6     }
7 }

```

Listing 35: Attempting to create a function without lifetime annotations

In Listing 35, a function named `shorter_vec` is defined which takes two references each of data type `Vec<T>` and returns a reference to the shorter one of those vectors. The compiler rejects this function definition. The error message again gives more insight:

```

this function's return type contains a borrowed value, but the
→ signature does not say whether it is borrowed from `x` or `y`

```

Since the execution of this function is dependent on the concrete values passed – particularly their length – the compiler cannot statically determine which of the two references will be returned. To make this example work, generic lifetime annotations need to be added to the function. Lifetime annotations do not change for how long a reference is valid [40]. Lifetimes are used as follows: The called function specifies the relationship of the lifetimes to each other, i.e. which lifetime must be greater or less, and which lifetimes must be identical. The lifetime annotations are written in angle brackets after the function name. The calling function determines the concrete value of the lifetimes of the passed references. In this respect, lifetime annotations serve as a constraint: The compiler receives additional information about the relationships between the lifetimes of the different arguments. By checking whether the lifetimes of the passed arguments match this specification, the borrow checker can ensure the validity of the references.

In the case of the `shorter_vec` function from Listing 35, the relationship of the lifetimes of the parameters and the return value is as follows: In order to be valid, the reference returned needs to be valid for at least as long as the passed parameters `x` and `y`. Therefore, it suffices to require all three references to have the same generic lifetime. As such, Listing 36 adds the lifetime annotation `'a` to all parameter and return types of the function signature. The function body can remain unaltered.

```

1 fn shorter_vec<'a, T> (x: &'a Vec<T>, y: &'a Vec<T>) -> &'a Vec<T> {
2     if x.len() < y.len() {
3         x
4     } else {
5         y
6     }
7 }

```

Listing 36: Expanding the function definition from Listing 35 with lifetime annotations

The lifetime annotation `'a` tells the compiler that the function `shorter_vec` demands that it is called with reference arguments whose values are valid for at least as long as the generic lifetime `'a`. When concrete references are passed to the function, the generic

lifetime `'a` is instantiated as the smaller of the lifetimes of the arguments `x` and `y` [59]. In consequence, as the return value is also annotated with lifetime `'a`, the compiler will then enforce that this returned reference is valid for at least the smaller of the lifetimes of the arguments. If a function returns a reference, the lifetime parameter of that reference needs to match the lifetime of at least one of the function parameters.

Listing 37 calls the `shorter_vec` function of Listing 36 with two references each having a different lifetime:

```

1 {
2     let long = &vec![1,2,3,4];           // -----+'a
3                                         //          |
4     let short = &vec![5,6,7];           // ----+'b  |
5                                         //         |  |
6     let result = shorter_vec(long, short); //         |  |
7     println!("{}", result);             //         |  |
8 }                                         // ----+-----+
```

Listing 37: Calling the `shorter_vec` function

In Listing 37 lines 2 and 4, two references to values of type `Vec` are created. The vectors are of different lengths. In line 6, `shorter_vec` is called with these references. The return value, a reference to the shorter vector, is assigned to a new variable called `result` and printed to the console. The output of the program is `[5, 6, 7]`. The format string of the `println!` macro makes use of the character sequence `{:?}`, which serves a similar purpose as the sequence `{}` used so far; the differences between these notations are irrelevant for the understanding of lifetime annotations.

The code is accepted by the compiler, since the lifetime of both vectors from Listing 37 `long` – as indicated by the lifetime `'a` – and `short` – indicated by `'b` is the same. Their scope ends with the closing brace in line 8.

One special lifetime exists in Rust: The `'static` lifetime. It denotes that the reference is valid for the entire program. It is usually used for the string slice type (`str`) or for globals, since their values are hardcoded into the application binary [40].

### 2.4.3 Lifetime elision

In the early days of Rust, programmers often found themselves using the same lifetime annotation patterns over and over when writing code that deals with references. These patterns were deterministic and predictable in nature, which lead the Rust team to extend the borrow checker in a way that allows programmers to elide specific lifetime annotations [59]. Currently, there are three rules that define when lifetime annotations can be omitted [40]:

- Each reference in the arguments of a function gets a distinct lifetime parameter.
- If there is exactly one input lifetime, that lifetime is assigned to all elided lifetimes in the return values of that function.
- If there are multiple input lifetimes, but one of the input references is `&self` or `&mut self`, the lifetime of `&self` is assigned to all elided output lifetimes.

These rules allow programmers to elide the most commonly found lifetime patterns, therefore making Rust code more readable. There is potential for additional lifetime elision rules to be implemented in the future, in order to require even less explicit lifetime annotations [59].



### 2.4.4 Non-lexical lifetimes

Non-lexical lifetimes are lifetimes that are not based on the lexical scope of a variable but are instead based on the control-flow graph [47]. By using non-lexical lifetimes, it is possible to determine through syntactic analysis the range in the program code where a reference is “live”, in the sense that it is being accessed.

The support for non-lexical lifetimes is a recent addition to the Rust borrow checker. The concept was introduced in 2018 with Rust version 1.31 [65]. This addition addresses two problems programmers had to face with lexical scopes: First, the compiler previously rejected code that was safe, that is, the borrow checker was too strict. Second, program code often became unnecessarily cluttered: In order to get the borrow checker to approve of their programs, programmers were sometimes forced to write more complex code than what their application logic would have required.

When using lexical lifetimes, the borrow checker requires a reference to be valid until the end of the lexical scope of the variable holding that reference. In contrast, when using non-lexical lifetimes, the borrow checker performs additional syntactic analysis to determine where variables are accessed. If, for example, a mutable reference to a value exists, no additional mutable references to that value can be created, as described in section 2.3. If, however, that original mutable reference never accesses the value, it would be safe to allow the creation of a new mutable reference. In many cases, the range where accesses to a variable occur is smaller than the lexical scope of the referenced variable. Requiring a reference to only be valid for this reduced source code range allows for more flexible writing of safe code.

Listing 38 – adapted from the RFC regarding non-lexical lifetimes – illustrates this need for non-lexical lifetimes [47]. Attempting to compile this code with an old version of the Rust compiler (versions before version 1.31) results in an error.

```
1 fn main() {
2     let mut data = vec!['a', 'b', 'c'];
3     let slice = &mut data[..];
4     capitalize(slice);
5     data.push('d'); // error with old compiler
6 }
7
8 fn capitalize(data: &mut [char]) {
9     // ..
10 }
```

Listing 38: Safe code rejected by compiler: Mutable reference assigned to a variable

In line 2, a mutable vector called `data` is declared and initialized. In line 3, a mutable reference to the vector is created and stored in the variable `slice`. In line 4, this variable `slice` is passed to the function `capitalize`, which takes a mutable reference to a **char** array. After the function returns, the reference to `data` stored in variable `slice` is still valid until the scope ends. Finally, in line 5, `push()` is called in order to append a character to the vector `data`, resulting in a compiler error with the old Rust compiler.

The problem lies in the fact that `push()` – a method defined in the context of the `Vec` type – takes a mutable reference to the object that is called upon, in this case `data`. Since a mutable reference to `data` already exists which is stored in the variable `slice`, old compiler versions reject this code. However, variable `slice` is never used after line 4. Therefore, the code is safe to compile. With a modern version of Rust, this example compiles successfully because not only the lexical scope is taken into consideration, but also the data

flow. The example above is just one concise examples of an issue that non-lexical lifetimes are trying to solve. More problems existed, for example with conditional control flow or nested functions [47]. Through the introduction of non-lexical lifetimes as the default in 2018, these issues have been mostly solved.

---

## Garbage collection in Rust

---

### 3.1 Automatic memory management

Automatic memory management refers to the automatic allocation and reclamation of memory. Memory that was allocated to the program but is no longer referenced is reclaimed automatically. This unused memory is called *garbage*. In academic publications, the terms “garbage collection” and “automatic memory management” are occasionally used interchangeably [95]. Automatic memory management, however, is a broader term, which includes both *reference counting* and *garbage collection*. Rust core developer Manish Goregaokar proposed in a blog post [26] to use the term garbage collection to refer to *tracing garbage collection*, as this is more aligned with the colloquial use of the term.

In this section, the two main approaches to automatic memory management as well as their advantages and disadvantages are presented.

#### 3.1.1 Tracing garbage collection

Garbage collection is used to automatically manage memory. Manual memory allocation as used in C and C++ is error-prone and can lead to critical memory errors [1]. According to Wilson [95], garbage collection is also necessary to allow for full modularization of programs – using garbage collection abstracts the process of memory allocation and deallocation away from the programmer, thereby simplifying the program code. It also allows programming languages to guarantee memory safety, as memory management duties are fully taken over by the garbage collector.

A tracing garbage collector determines whether a data object is “live” by traversing pointers in order to find all data objects that are reachable by the program [95]. A well known algorithm for tracing garbage collection is called *mark and sweep* and is divided into two phases: The marking phase, and the sweeping phase. Every data object is given a *mark bit*, which is used to indicate whether this object is reachable. When an object is created this bit is set to 0 (for false). A non-empty set of *root objects* is defined (this is implementation specific but usually includes objects like global and static variables, variables on the stack, and register variables).

During the marking phase of the garbage collector a graph traversal of the application data is performed, starting at the roots of each of these data objects: Data objects are considered

nodes, and every visited node is marked as reachable, i.e. its mark bit is set to 1. When all reachable objects have been traversed the algorithm moves to the sweeping phase. If more than one root object exists the marking algorithm is performed for each of them.

In the sweeping phase, the garbage collector checks for each heap-allocated object whether its mark bit is set to 0 (for unreachable) and releases the memory for those objects.

Tracing garbage collection is usually executed in fixed intervals during the execution of the program. This leads to pause times for the application as the program is effectively halted during the garbage collection period [6]. This halting approach is called *stop-the-world* mark and sweep [45].

### 3.1.2 Reference counting

The fundamental idea to reference counting is to keep track of the number of references to any data object [43]. When no references to an object exist the object is deallocated. Therefore, every object maintains a reference count field. Whenever a reference to the object is created, that field is incremented. Similarly, the field is decremented as soon as a reference is removed. On reaching a zero counter, the object is freed and its memory is returned to the operating system [34].

### 3.1.3 Problems with automatic memory management

While automatic memory management provides many advantages over manual memory management, it is not without issues of its own. Since additional memory and CPU cycles are used, automatic memory management requires a significant resource overhead compared to manual memory management, thereby making garbage collection unsuitable for high-performance and embedded applications [20, 95]. In addition to that, use cases exist where neither reference counting nor tracing garbage collection is suitable. Reference counting, for example, is unable to handle cyclic data structures like graphs, and is also not suited for high-throughput applications: Since with every addition or removal of a reference a counter has to be updated, a large number of writes is necessary [6]. Tracing garbage collection leads to potentially high pause times for the user application, especially with large heap sizes. In real-time applications usually implemented in a system language, such behaviour is typically not acceptable.

In practice, the issues described above make it difficult to pick a universally acceptable solution. A large number of specialized collectors have been developed to solve these issues for particular use cases. However, Bacon et al. have shown that all subsequent approaches are hybrids of tracing garbage collection and reference counting [6].

## 3.2 Garbage collection in early Rust versions

The initial design of Rust – which was announced in 2010 – included a garbage collector. In 2013, a proposal by a group of Rust developers around Patrick Walton [92] was made to remove this garbage collector from the language and instead move it to a library. The proposal lists three key reasons why a garbage collector is not necessary in the Rust language: Lack of familiarity with its semantics, unnecessary complexity, and lack of flexibility.

Along with a reference type with similar semantics to today’s implementation, the initial versions of Rust included two different pointer types, an *owned* pointer type, indicated by a tilde (~), as well as a *managed* pointer type, annotated with an @ symbol. Both of these types were *smart pointers* meaning they are an abstract data type simulating a pointer, but with added functionality. In particular, they both deallocated memory when it was no longer in use.

The owned pointer type, sometimes called “unique smart pointer”, only allowed a single reference to a value. In contrast, the managed pointer type allowed several references to a value and used garbage collection to automatically free memory once all references were dismissed. According to Patrick Walton, deciding which pointer type to use proved difficult for many programmers [92]. Over time, it became clear that the owned pointer type was the prevalent one, and the managed pointer only played a minor role. Walton therefore proposed that the managed pointer type `@` should be removed from the language. By having several different pointer and reference types the language also had an unnecessary runtime overhead. As it became clear that Rust was suited to do low-level systems programming and even kernel development, it became a goal to remove Rust’s runtime environment altogether. In order to be able to compete with C and C++, the performance level of Rust needs to match the one of C and C++ and reducing unnecessary overhead was a way to improve performance in Rust.

Implementing a universal garbage collector that handles all possible use cases efficiently is a difficult task and hinders the flexible use of a language. That said, a small class of applications exists that works better with tracing garbage collection due to the higher throughput as well as the handling of cyclic data structures. For other applications, reference counting is better suited. Additionally, in order to interact with garbage collected languages or operating systems, a flexible memory management is required.

A possible solution to these issues would be to implement several automatic memory management systems into the core language. However, this would further increase the overhead and complexity of the language. The other approach – as proposed by Walton – would be removing garbage collection from the language entirely, and replacing it with functionality that would allow recreating its features as a library. Doing this would also allow users to create garbage collection libraries tailored to their needs in the future. This idea resonated strongly in the Rust community [29]. One user noted that he believes that “garbage collection has no place in the core definition of any language that targets [...] high-performance applications” [28]. The proposal to remove garbage collection from the Rust language was eventually accepted: The first stable version of Rust, version 1.0, did not include a garbage collector [66].

### 3.3 Garbage collection as a library

This section aims to give an introduction to selected garbage collection libraries in both Rust and C++. The general motivation of garbage collection in system programming languages is highlighted, as well as the individual use cases of specific libraries.

#### 3.3.1 Motivation for garbage collection in systems programming

Using garbage collection in systems programming languages that are focused on performance and flexibility may seem counter-intuitive at first. According to Rust core developer Manish Goregaokar (2021), there are two major motivations to use garbage collection in systems programming languages: Managing cyclic data structures, and the integration with (or implementation of) garbage collected languages or systems [26].

Cyclic data structures such as graphs or linked lists cannot be handled by conventional reference counting mechanisms, as the reference count field can never be decreased to zero, even if the entire list has become unreachable [14, p. 25]. Both Rust and C++ come with a reference counting mechanism that can be utilized by the programmer if desired. For cyclic data structures, however, more sophisticated garbage collection algorithms are required.

In large software systems, interaction with components written in other languages is often inevitable. These components may have specific requirements for client software in order

to use certain functions or Application Programming Interfaces (APIs) of the system [92]. To be able to achieve this, a use-case specific garbage collection library may be necessary to accomodate the needs of the host system.

### 3.3.2 Rust garbage collection libraries

All of the garbage collectors listed in this section are open-source software and available through the Rust package management system `cargo`. The respective GitHub repositories are referenced as well.

#### **rust-gc**

According to its author Manish Goregaokar, `rust-gc` [27] is a general purpose mark-and-sweep garbage collector. In a 2015 blog post, he describes the motivation behind the `rust-gc` garbage collector: “In order to implement an interpreter for a garbage collected language, a garbage collector in Rust would be helpful” [46]. Also, a garbage collector would simplify handling complicated, dynamic graph data structures.

Since the Rust language was built without a garbage collector, there is no designated language support for the detection of root objects on the stack. To avoid scanning the entire stack for root objects, the `rust-gc` garbage collector uses an approach comparable to reference counting in order to track the root objects. The actual collection is based on the mark-and-sweep approach described above.

`rust-gc` provides a type called `Gc<T>`, which enables the user to wrap a type `T` into a garbage collected object. The `Gc` type is designed to provide an interface similar to the one of the `Rc` type to the user, but additionally allows mutability [27]. The developers do not recommend the pervasive use of this collector but instead encourage to only use it when it is required.

The `rust-gc` garbage collector currently does not support concurrency [27]. While there have been efforts to add concurrency support, the corresponding development branch has been stale since 2018 [51]. Before this, the garbage collector has been receiving continuous updates.

#### **bacon-rajana-cc**

Bacon et al. have developed a cycle collection algorithm that is suitable for concurrent applications [7]. Nick Fitzgerald implemented a cycle collector called `bacon-rajana-cc` [25] based on this algorithm. It was first published in Rust’s package management system `crates.io` in 2015 and has received periodical updates ever since.

A key observation regarding garbage cycles is that they can only be created when a reference count is decremented to a non-zero value, since incrementing a reference counter cannot create garbage, and decrementing the counter to zero means that the garbage has already been detected and is being freed [7]. The `bacon-rajana-cc` makes use of this observation: Whenever a reference count is decremented to a non-zero value, the object is added to a list of “potential cycle roots” [26]. This list of potential roots is then traversed: The collector decrements the reference count of every object it encounters, cleaning up any object whose reference count is decremented to zero. In another pass, the original reference count of the elements is restored.

The upside to cycle collectors is that their performance is dependent on the amount of actual garbage, as opposed to the size of the heap, as is the case with mark-and-sweep tracing garbage collectors [26].

### **josephine**

`josephine` [32] is an experimental garbage collection library developed to work with the likewise experimental Servo browser engine which is being used in Mozilla Firefox. In particular, it allows Rust data to be attached to JavaScript objects, whereas the JavaScript execution is then managing the lifetime of the Rust data. References to data that is managed by JavaScript can then be copied and discarded freely, while the garbage collector cleans up any unused data.

`josephine` was designed specifically to be used with the Servo browser engine, and therefore has implementation details very specific to this particular use case. Hence, `josephine` has little relevance outside of the Servo project.

### **Bronze**

The `Bronze` [16] garbage collector serves as an experimental library for language design research purposes. It was developed by Michael Coblenz in 2020, and has been used as a tool to conduct studies on the usability costs of Rust's restrictions [17]. The author has explicitly stated that the garbage collector is not stable and should not be used in production systems [88].

All of the libraries listed above serve very specific purposes and should not be considered as universal garbage collection solutions for the Rust language. All of the libraries listed above currently lack support for concurrency therefore limiting their potential applications. While this list highlights only a selection of garbage collection libraries, at the time of writing, no universally usable garbage collection implementation is universally accepted. A comprehensive list of garbage collectors with more implementation details is provided in a blog post by Manish Goregaokar (2021) [26].

### **3.3.3 C and C++ garbage collection libraries**

Since neither C nor C++ come with a builtin garbage collector, libraries to support garbage collection exist for these languages as well.

#### **Boehm garbage collector**

The Boehm garbage collector is a conservative mark-and-sweep garbage collector proposed by Boehm and Spertus in 2007 [11]. Conservative garbage collection describes the approach of considering all stack objects to be pointers (thereby increasing the amount of potential garbage) in the absence of information about the types of elements at runtime [12]. Most implementations of C and C++ do not provide any runtime pointer information, hence making this technique necessary [55]. This approach is called *conservative*, as it does not use any heuristics and may not always detect all garbage [12].

The Boehm collector serves as a replacement for C's `malloc` memory allocator and makes explicit calls to `free` optional. It supports both C and C++ and can even be used to add garbage collection to the runtimes of other languages [10]. It includes alterations to the naïve mark-and-sweep algorithm to improve the performance.

#### **Oilpan**

Oilpan is a mark-and-sweep garbage collector developed by the Chromium project as a library for the JavaScript engine V8 which is written in C++ [4]. While designed for use

in combination with V8, the garbage collector can also be used separately, allowing it to be integrated into other C++ projects as well. Oilpan supports atomic, incremental and concurrent garbage collection, making it a flexible solution [63]. The project is under active development and is continued to be extended [4].

### 3.4 Extending Rust's memory management through **unsafe** calls

As described in the last chapter, the Rust language can be augmented with memory management tools targeting specific use cases. As the access to memory is generally managed by the operating system, interaction with interfaces provided by the respective operating system is required to build these tools: In the case of Linux, one might want to use the memory allocator `malloc` which is defined in the C standard library `libc`, or even directly use system calls to interact with the operating system. However, both the C standard library and most modern operating systems only provide a C interface [54].

To enable interaction of Rust with C a Foreign Function Interface (FFI) exists that allows Rust to call functions written in C. However, since C is an inherently unsafe language in terms of memory management, Rust is unable to uphold its memory safety guarantees which is why Rust requires external function calls to be wrapped in an **unsafe** code block [74].

Unsafe code serves as an extension to the Rust programming language. Rust can essentially be seen as the union of two languages: *Safe Rust* and *Unsafe Rust* [74]. Safe Rust is the language that is colloquially meant by “Rust”. While in safe Rust code memory safety guarantees are enforced by the compiler, these guarantees are not enforced in unsafe Rust code [59, p. 418]. It becomes the responsibility of the programmer to encapsulate unsafe code in an API that is safe to use. The Rust standard library makes use of unsafe code as well but this code has been verified to be safe to use [59]. A programmer making use of unsafe code essentially tells the compiler that all invariants of the language are upheld manually.

Common use cases for unsafe Rust are the interaction with operating systems or hardware, low-level performance improvements, or the implementation of data structures and functions that internally require operations that are not deemed safe [74]. By separating safe from unsafe code programmers can gain control over low-level implementation details where needed while maintaining thread and memory safety in all other portions of the program.

The rules and semantics of unsafe Rust are the same as those of safe Rust while allowing additional operations which include the dereferencing of raw pointers and calling external functions [83]. Apart from these additional operations, Rust code has the same semantics and it is equally checked by the compiler and memory safety rules are enforced.

Unsafe Rust introduces a raw pointer type `*T`, where `T` denotes a generic type. Similar to a C pointer, the raw pointer points to an address in memory. The value at that address can be accessed using the dereference operator which is also denoted by an asterisk. Raw pointers can be declared in safe Rust but dereferenced in an **unsafe** block only. The raw pointer type allows multiple mutable pointers to the same memory location. Listing 39 shows an exemplary use of raw pointers:



```

1 let mut i = 42;
2 let r = &mut i as *mut i32;
3 unsafe {
4     println!("Value of r is: {}", *r);
5 }

```

Listing 39: Dereferencing a raw pointer

In line 1, a mutable variable `i` is declared and initialized with value 42. In line 2, a new variable `r` is declared. It is initialized with a raw pointer of type `*mut i32` which is created by typecasting the mutable reference to `&i` to the `*mut i32` type using the `as` keyword. In order to dereference the raw pointer `r`, an `unsafe` block is required. Attempting to dereference the raw pointer outside of such a block results in a compiler error.

If a programmer wants to use any unsafe feature outside of an `unsafe` block, the function itself has to be declared as `unsafe`. An `unsafe` function can only be called from within an `unsafe` block. The following listing declares and uses an unsafe function:

```

1 unsafe fn unsafe_function() {
2     println!("This is unsafe!");
3 }
4
5 unsafe {
6     unsafe_function();
7 }

```

Listing 40: Defining and calling an `unsafe` function

In line 1 of Listing 40, the `unsafe` modifier is used to declare the function `unsafe_function` as unsafe. In the function body unsafe features can be used. Note that this function only prints to the console and does not actually perform any unsafe actions. However, since the function was declared as `unsafe` it can only be called from an `unsafe` block: In line 4, an `unsafe` block is entered. In line 5, the function `unsafe_function()` is then called. The `unsafe` block ends with the closing brace in line 6.

Calling an external function such as a C function is generally deemed unsafe and as such has to be wrapped in an `unsafe` block. The example in Listing 41 shows how the `malloc` memory allocator from the C standard library is called in Rust in order to implement an allocator to be used in Rust:

```

1 struct MinimalAllocator;
2
3 unsafe impl GlobalAlloc for MinimalAllocator {
4     unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
5         libc::malloc(layout.size()) as *mut u8
6     }
7
8     unsafe fn dealloc(&self, ptr: *mut u8, _layout: Layout) {
9         libc::free(ptr as *mut libc::c_void);
10    }
11 }

```

Listing 41: Implementing a Rust `malloc`-based memory allocator using `unsafe` code

In line 1 of Listing 41 a `struct` named `MinimalAllocator` is defined. In Rust, a `struct` is used to define compound data types, whereby in this case the `struct` does not hold any data but is instead used as a marker to allow implementing a trait. In line 3, the trait `GlobalAlloc` is implemented on the `MinimalAllocator` type using the `impl`: The trait requires implementations of the methods `alloc` and `dealloc`.

In line 4, the method `alloc` is implemented: It takes a `Layout` value as its argument. The `Layout` value represents a block of memory and consists of the size of the block and its alignment. The function returns a raw pointer to the allocated memory. In the function body of `alloc` the `libc` crate – used to interact with the C standard library – is used to call the memory allocator `malloc`. Since `malloc` is defined in the C standard library, its call has to be wrapped in an `unsafe` function. In line 5, `malloc` is called with the size of the `Layout` object passed to the `alloc` method as its argument. The result of this call is typecast to a `*mut u8` using the `as` keyword and returned from the function.

The implementation of `dealloc` in line 8 takes a raw pointer to a memory segment and a `Layout` as its arguments. The raw pointer is passed to the `free` function from the C standard library after being typecast to a `C void` type. This call returns the previously allocated memory to the operating system.

By implementing the `GlobalAlloc` trait, `MinimalAllocator` can be set as the global memory allocator and is then used for all memory allocations in the program.

This minimal example shows how `unsafe` features such as raw pointers and external function calls can be used to implement a new memory allocator. The Foreign Function Interface could also be used to interact with the operating system directly using system calls for tunability or performance reasons.

## 3.5 Rust’s solution to memory management

Rust’s solution to memory management can be divided into two categories: The handling of non-cyclic and cyclic data structures. Memory of non-cyclic data structures is reclaimed using the `Drop` function which follows the *Resource Acquisition is Initialization (RAII)* pattern. The use of `drop` has already been explained in Chapter 2.2.1 but its implementation has not been considered so far. The same is true for reference counting which has been investigated on a very high level in Chapter 3.1.2 but not described in the context of Rust. These important aspects are the focus of the next two chapters.

### 3.5.1 Reclaiming memory of non-cyclic data structures

For non-cyclic data structures Rust makes use of the *RAII* pattern. Introduced in the C++ programming language, the fundamental idea of *RAII* is that objects own their resources – including heap memory – and they are solely responsible for managing them [60]. The memory for an object is allocated when the object is created. As soon as an object goes out of scope, its *destructor* is invoked, where all operations required to release the resources of the object are implemented.

In Rust, this destructor is realized using the `Drop` trait: Any type implementing the `Drop` trait is required to provide an implementation of the function `drop(&mut self)`. This method is called when an object of this type goes out of scope [72]. An implementation of `drop` is provided by Rust for all standard data types, and this function is called recursively for all fields of compound data types [71] which is why programmers rarely need to implement the trait themselves.

The `drop` method is also called in some other use cases where resources need to be freed: Assignment, for example, calls the destructor of the left-hand side operand if it was previously initialized [71].

It is not possible for types to implement both the `Copy` and the `Drop` trait at the same time. Types implementing `Copy` usually do not use heap-allocated memory but are rather stored on the stack. As stack memory is pre-allocated to the program by the operating when its process is created and freed when the process exits, no special treatment of values stored on the stack is possible.

### 3.5.2 Reclaiming memory of cyclic data structures

The RAII approach has practical limitations: In order to prevent *double free* errors (an error where a memory region is freed more than once, leading to undefined behaviour) it is necessary to restrict the management of a memory segment to a single owner. This, however, limits the data structures that can be implemented: Cyclic data structures cannot be represented with this restriction in place. To enable the implementation of cyclic data structures, Rust provides *smart pointers*. Smart pointers provide an abstraction level from raw pointers and implement additional features such as metadata or memory management capabilities. Additionally, some smart pointer types allow for a value to be owned by multiple owners [59, p. 312]. Rust includes several smart pointer types to extend the functionality of the language and users of external libraries may implement their own. This chapter focuses on the reference counted smart pointer `Rc<T>` due to its relevance to memory management. `Rc<T>` works as outlined in Chapter 3.1.2. The `clone` function is used on an `Rc<T>` object in order to create a new pointer to the same memory location [80] and to increment the reference count. It is not possible to manually decrement the reference count as the implementation of the `Drop` trait already takes care of decrementing the reference count at the end of a scope.

Simple reference counting, however, is not sufficient to reclaim memory used by cyclic data structures. To be able to correctly deallocate memory of cyclic data structures Rust extends the reference counting algorithm described in Section 3.1.2 by an additional field that keeps track of *weak references*. An additional smart pointer type called `Weak<T>` is added. While the `Rc<T>` type allows expressing ownership relationships where more than one owner to a value exists, the type `Weak<T>` allows creating a reference that does not express an ownership relationship and only exists for the implementation of the data structure [59, p. 341]. A reference counted value now has two fields to keep track of: The number of strong references, and the number of weak references. The memory is deallocated once the strong reference count reaches zero whilst discarding the number of weak references.

By introducing reference counting as a mechanism to automatically reclaim memory for more complex data structures and providing the necessary tools to even handle cyclic data structures Rust gives programmers flexibility while maintaining memory safety.

The combination of `Drop` and `Rc<T>` enables Rust to make use of static optimizations wherever possible but also allows for runtime memory management for increased flexibility. Programmers do not have to perform manual memory allocations and deallocations, thereby reducing the complexity and susceptibility to errors. Rust also offers high performance by avoiding a runtime tracing garbage collector, yet the possibility to reclaim memory of cyclic data structures remains.



---

### Memory management: Rust and C/C++ compared

---

This chapter presents the differences in memory management at the system level of the languages Rust as well as C and C++. While in C and C++, values are tied to a specific memory location and are only moved if explicitly instructed to do so by the programmer, values in Rust may move in memory unbeknownst to the programmer. This requires special handling in certain use cases.

#### 4.1 Movable versus fixed memory locations

In C and C++, every data object has a fixed location in memory that does not implicitly change. Pointers are then used to access this memory location. Any number of pointers to a memory location may be created to enable multiple concurrent accesses to the stored value; for this to work, however, the target address of the value referenced by the pointers must not change. C and C++ do not guarantee that a pointer refers to a valid memory location, the pointer may be uninitialized (leaving its value indeterminate), could have been freed, or be `NULL`. Accessing the memory in any these cases can lead to runtime errors [1].

Rust is different in this respect: The location of a value in memory may change without notice. This is closely tied to the concept of *moving ownership*, as explained in chapter 2.2.2. When, for example, a function takes ownership of a value (when that value is passed to the function as an argument), the value may be moved to the stack frame of that function. A stack frame is a region of memory allocated to a function when it is called. It contains space for both the function arguments and local variables, as well as additional metadata [41]. Listing 42 shows that moving ownership can result in a value being moved in memory as well:

```

1 fn take_ownership(s: String) {
2     println!("Memory address of s: {:p}", &s);
3 }
4
5 fn main() {
6     let s = String::from("Rust");
7     println!("Memory address of s: {:p}", &s);
8     take_ownership(s);
9 }

```

Listing 42: Moving a value in memory by transferring ownership

Listing 42 revisits the function `take_ownership` introduced in Listing 24. The function takes a `String` as its argument and was modified to print the memory address of the `String` instead of printing its value. Since the `String` does not implement the `Copy` trait, ownership of the argument `s` is moved when the string is passed to the function. In line 6, within the `main` function, a `String` variable `s` is created. Subsequently, the memory address of variable `s` is printed. In line 8, function `take_ownership` is called with `s` as its argument, thereby moving ownership to the function. Running this program results in two different memory addresses for the same value (the `String "Rust"`) being printed to the console. This shows that the value was in fact moved in memory.

An objects location in memory may change in non-obvious, complicated ways through several layers of function calls and dependent on conditional control flow. Wherever possible, the compiler will avoid redundant moves for efficiency reasons, however, no guarantees about the absence of moves are made at compile time [50].

## 4.2 Memory management in asynchronous Rust programming

In many programs, it is desirable to have more than one task being executed concurrently. Rust supports concurrency with the *asynchronous programming* model and the `async` keyword. It allows users to run a large amount of tasks in a small number of operating system threads, thereby reducing overhead whilst the synchronous programming style can mostly be maintained [84]. Asynchronous computations are represented by the `Future` trait. Futures are *polled*, advancing the computation as far as possible. In a synchronous context, a blocking function will block an entire thread. With blocked Futures, however, control is returned to the thread thereby allowing other operations to be ran [67]. A simple example of a use case of asynchronous programming is reading data from a socket: The socket may or may not have data available. When polled, the `Future` either returns `Pending` if no data is available, or `Ready(data)` where `data` represents the data received from the socket.

Problems can arise if an `async` block makes use of references: If a reference to a field of a `Future` exists, and that `Future` is moved in memory, for example due to the move semantics described above, the referenced field is moved as well, resulting in an invalid reference. To avoid this, Rust requires a `Future` to be *pinned* to a specific memory location. This is done using the `Pin` type which wraps pointer types, ensuring that referenced values will not be moved [76]. For most types, this is not necessary: These types implement a trait called `Unpin` indicating to the compiler that they may always be moved in memory, even when wrapped by the `Pin` type. Types that cannot be moved because they contain pointers or references that could otherwise be invalidated contain a *marker* called `!Unpin`, indicat-

ing to the compiler that these types must never be moved after being pinned [76]. Values can be pinned to both the stack and the heap. Pinning a value on the heap ensures that it remains reachable at the same memory address for the entirety of its lifetime. Pinning values to the stack is slightly more complicated, as it requires the use of **unsafe** code and places the responsibility of maintaining the *pin contract* on the programmer [76]. References to pinned values on both stack and heap may still be obtained using **unsafe**. With the pinning mechanism, Rust ensures that memory safety guarantees can be upheld in asynchronous programming.





## CHAPTER 5

---

### Conclusion

---

This thesis has laid out the fundamental concepts of memory management in Rust. It also provided an overview of memory management concepts in general. It has done so by evaluating the current state of research regarding theoretical foundations of memory management and garbage collection, followed by a brief introduction to the syntax of the Rust language. The semantics of Rust's memory management concepts were illustrated by code listings, highlighting relevant aspects and allowing for comparisons.

The ownership model allows the Rust language to guarantee memory safety by performing compile time checks. References to values are validated through borrow checking, preventing dangling references. No tracing garbage collector is included in Rust, although the language provides a reference counting mechanism to allow the implementation of complex data structures. Additionally, garbage collectors for specific use cases are available as libraries. Programmers also have the option to choose to elide some of the compile time checks by declaring `unsafe` blocks. Doing so adds additional features to the language, allowing for more low-level access to hardware and operating system resources, however, it requires the programmer to uphold invariants themselves.

Through the combination of memory safety without tracing garbage collection and access to low-level resources Rust is especially suited to system programming and constitutes a real alternative to C and C++.



---

## Bibliography

---

- [1] PERIKLIS AKRITIDIS. **Practical memory safety for C**. Technical report, University of Cambridge, Computer Laboratory, 2011. Available from: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-798.pdf>.
- [2] BRIAN ANDERSON, LARS BERGSTROM, DAVID HERMAN, JOSH MATTHEWS, KEEGAN MCALLISTER, MANISH GOREGAOKAR, JACK MOFFITT, AND SIMON SAPIN. **Experience Report: Developing the Servo Web Browser Engine using Rust**. *arXiv:1505.07383 [cs]*, May 2015. arXiv: 1505.07383. Available from: <http://arxiv.org/abs/1505.07383>.
- [3] GUI ANDRADE. **Storing unboxed trait objects in Rust**, December 2018. Available from: <https://guiand.xyz/blog-posts/unboxed-trait-objects.html>.
- [4] ANTON BIKINEEV, OMER KATZ, AND MICHAEL LIPPAUTZ. **Oilpan library · V8**, November 2021. Available from: <https://v8.dev/blog/oilpan-library>.
- [5] DAVID F. BACON, PERRY CHENG, AND V. T. RAJAN. **A Unified Theory of Garbage Collection**. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '04*, pages 50–68, New York, NY, USA, 2004. Association for Computing Machinery. event-place: Vancouver, BC, Canada. Available from: <https://doi.org/10.1145/1028976.1028982>.
- [6] DAVID F. BACON, PERRY CHENG, AND V. T. RAJAN. **A Unified Theory of Garbage Collection**. *SIGPLAN Not.*, **39**(10):50–68, October 2004. Place: New York, NY, USA Publisher: Association for Computing Machinery. Available from: <https://doi.org/10.1145/1035292.1028982>.
- [7] DAVID F. BACON AND V. T. RAJAN. **Concurrent Cycle Collection in Reference Counted Systems**. In *Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP '01*, pages 207–235, Berlin, Heidelberg, 2001. Springer-Verlag. Available from: <https://pages.cs.wisc.edu/~cymen/misc/interests/Bacon01Concurrent.pdf>.
- [8] ARIA BEINGESSNER. **Rust's Unsafe Pointer Types Need An Overhaul - Faultlore**, March 2022. Available from: <https://gankra.github.io/blah/fix-rust-pointers>.
- [9] CHRIS BISCARDI. **Why can't I early return in an if statement in Rust?**, May 2021. Available from: <https://www.christopherbiscardi.com/why-can-t-i-early-return-in-an-if-statement-in-rust>.

- [10] HANS-J BOEHM. **The “Boehm-Demers-Weiser” Conservative Garbage Collector.** Presentation slides, 2004. Available from: <https://hboehm.info/gc/04tutorial.pdf>.
- [11] HANS-J BOEHM AND MICHAEL SPERTUS. **Transparent Programmer-Directed Garbage Collection for C++.** page 29, June 2007. Available from: <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2310.pdf>.
- [12] HANS-JUERGEN BOEHM AND MARK WEISER. **Garbage collection in an uncooperative environment.** *Software: Practice and Experience*, **18**(9):807–820, 1988. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380180902>. Available from: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380180902>.
- [13] RODNEY A. BROOKS, RICHARD P. GABRIEL, AND GUY L. STEELE. **An Optimizing Compiler for Lexically Scoped LISP.** In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, SIGPLAN ’82*, pages 261–275, New York, NY, USA, 1982. Association for Computing Machinery. event-place: Boston, Massachusetts, USA. Available from: <https://doi.org/10.1145/800230.807000>.
- [14] K G CASSIDY. **Feasibility of automatic storage reclamation with concurrent program execution in a LISP environment.** Master’s thesis. December 1985. Available from: <https://www.osti.gov/biblio/5792943>.
- [15] YANG CHANG AND ANDY WELLINGS. **Low Memory Overhead Real-Time Garbage Collection for Java.** In *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES ’06*, pages 85–94, New York, NY, USA, 2006. Association for Computing Machinery. event-place: Paris, France. Available from: <https://doi.org/10.1145/1167999.1168014>.
- [16] MICHAEL COBLENZ. **Bronze**, November 2020. original-date: 2020-11-22T14:41:32Z. Available from: <https://github.com/mcoblenz/Bronze>.
- [17] MICHAEL COBLENZ, MICHELLE MAZUREK, AND MICHAEL HICKS. **Garbage Collection Makes Rust Easier to Use: A Randomized Controlled Trial of the Bronze Garbage Collector.** *arXiv:2110.01098 [cs]*, February 2022. arXiv: 2110.01098. Available from: <http://arxiv.org/abs/2110.01098>.
- [18] HOANG-HAI DANG, JACQUES-HENRI JOURDAN, JAN-OLIVER KAISER, AND DEREK DREYER. **RustBelt Meets Relaxed Memory.** *Proc. ACM Program. Lang.*, **4**(POPL), December 2019. Place: New York, NY, USA Publisher: Association for Computing Machinery. Available from: <https://doi.org/10.1145/3371102>.
- [19] DAVID WALKER. **Substructural Type Systems.** In *Advanced Topics in Types and Programming Languages.*, pages 3–43. MIT Press, 2005.
- [20] DINAKAR DHURJATI, SUMANT KOWSHIK, VIKRAM ADVE, AND CHRIS LATTNER. **Memory Safety without Runtime Checks or Garbage Collection.** *SIGPLAN Not.*, **38**(7):69–80, June 2003. Place: New York, NY, USA Publisher: Association for Computing Machinery. Available from: <https://doi.org/10.1145/780731.780743>.
- [21] DEREK DREYER. **RustBelt**, April 2021. Available from: <https://plv.mpi-sws.org/rustbelt/>.
- [22] MEHMET EMRE, RYAN SCHROEDER, KYLE DEWEY, AND BEN HARDEKOPF. **Translating C to safer Rust.** *Proceedings of the ACM on Programming Languages*, **5**(OOPSLA):1–29, October 2021. Available from: <https://dl.acm.org/doi/10.1145/3485498>.

- [23] GUILLAUME ENDIGNOUX. **Rust from a C++ and OCaml programmer's perspective (Part 2) | Blog | Guillaume Endignoux**, 2017. Available from: <https://gendignoux.com/blog/2017/09/29/rust-vs-cpp-ocaml-part2.html>.
- [24] ANA NORA EVANS, BRADFORD CAMPBELL, AND MARY LOU SOFFA. **Is rust used safely by software developers?** In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 246–257, Seoul South Korea, June 2020. ACM. Available from: <https://dl.acm.org/doi/10.1145/3377811.3380413>.
- [25] NICK FITZGERALD. **bacon.rajan.cc**, May 2022. original-date: 2015-05-19T07:10:44Z. Available from: <https://github.com/fitzgen/bacon-rajan-cc>.
- [26] MANISH GOREGAOKAR. **A Tour of Safe Tracing GC Designs in Rust - In Pursuit of Laziness**, April 2021. Available from: <https://manishearth.github.io/blog/2021/04/05/a-tour-of-safe-tracing-gc-designs-in-rust/>.
- [27] MANISH GOREGAOKAR. **rust-gc**, May 2022. original-date: 2015-05-17T03:31:43Z. Available from: <https://github.com/Manishearth/rust-gc>.
- [28] HACKERNEWS USER AMBROP7. **I'm completely in favor of this. Garbage collection has no place in the core def...** | **Hacker News**, June 2013. Available from: <https://news.ycombinator.com/item?id=5814390>.
- [29] HACKERNEWS USER GRAUE. **Removing garbage collection from the Rust language**, June 2013. Available from: <https://news.ycombinator.com/item?id=5811854>.
- [30] MATTHEW HERTZ AND EMERY D. BERGER. **Automatic vs. Explicit Memory Management: Settling the Performance Debate**, 2004.
- [31] GRAYDON HOARE. **graydon2 | Rust is mostly safety**, December 2016. Available from: <https://web.archive.org/web/20190502181357/https://graydon2.dreamwidth.org/247406.html>.
- [32] ALAN JEFFREY. **Josephine: using JavaScript to safely manage the lifetimes of Rust data**, April 2022. original-date: 2017-05-23T15:35:25Z. Available from: <https://github.com/asajeffrey/josephine>.
- [33] STEFAN JOHANSSON. **JEP 248: Make G1 the Default Garbage Collector**, February 2015. Available from: <http://openjdk.java.net/jeps/248>.
- [34] RICHARD JONES, ANTONY HOSKING, AND ELIOT MOSS. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman and Hall/CRC Press, first edition, August 2011. Available from: <http://gchandbook.org>.
- [35] RALF JUNG. *Understanding and evolving the Rust programming language*. PhD thesis, Universität des Saarlandes, Saarbrücken, 2020. Publisher: Universität des Saarlandes. Available from: <https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647>.
- [36] RALF JUNG, HOANG-HAI DANG, JEEHOON KANG, AND DEREK DREYER. **Stacked Borrows: An Aliasing Model for Rust**. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. Place: New York, NY, USA Publisher: Association for Computing Machinery. Available from: <https://doi.org/10.1145/3371109>.

- [37] RALF JUNG, JACQUES-HENRI JOURDAN, ROBBERT KREBBERS, AND DEREK DREYER. **RustBelt: Securing the Foundations of the Rust Programming Language**. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. Place: New York, NY, USA Publisher: Association for Computing Machinery. Available from: <https://doi.org/10.1145/3158154>.
- [38] PAUL KEHRER. **Memory Unsafety in Apple’s Operating Systems**, July 2019. Available from: <https://langui.sh/2019/07/23/apple-memory-safety/>.
- [39] STEVE KLABNIK. **Rust is more than safety | Next.js Blog Example with Markdown**, December 2016. Available from: <https://steveklabnik.com/writing/rust-is-more-than-safety>.
- [40] STEVE KLABNIK AND CAROL NICHOLS. **Lifetimes**, 2018. Available from: [https://web.mit.edu/rust-lang\\_v1.25/arch/amd64\\_ubuntu1404/share/doc/rust/html/book/first-edition/lifetimes.html](https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/lifetimes.html).
- [41] STEVE KLABNIK AND CAROL NICHOLS. **The Stack and the Heap**, April 2018. Available from: [https://web.mit.edu/rust-lang\\_v1.25/arch/amd64\\_ubuntu1404/share/doc/rust/html/book/first-edition/the-stack-and-the-heap.html](https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/the-stack-and-the-heap.html).
- [42] STEVE KLABNIK AND CAROL NICHOLS. **The Rust Programming Language**, 2022. Available from: <https://doc.rust-lang.org/book/>.
- [43] DONALD E. KNUTH. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., third edition, 1997.
- [44] JUNEYOUNG LEE, CHUNG-KIL HUR, RALF JUNG, ZHENGYANG LIU, JOHN REGEHR, AND NUNO P. LOPES. **Reconciling high-level optimizations and low-level code in LLVM**. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, October 2018. Available from: <https://dl.acm.org/doi/10.1145/3276495>.
- [45] YOSSI LEVANONI AND EREZ PETRANK. **An On-the-Fly Reference-Counting Garbage Collector for Java**. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, January 2006. Place: New York, NY, USA Publisher: Association for Computing Machinery. Available from: <https://doi.org/10.1145/1111596.1111597>.
- [46] MANISH GOREGAOKAR. **Designing a GC in Rust**, September 2015. Available from: <https://manishearth.github.io/blog/2015/09/01/designing-a-gc-in-rust/>.
- [47] NIKO MATSAKIS. **2094-nll - The Rust RFC Book**, September 2017. Available from: <https://rust-lang.github.io/rfcs/2094-nll.html>.
- [48] JOHN MCCARTHY. **Recursive functions of symbolic expressions and their computation by machine, Part I**. *Communications of the ACM*, 3(4):184–195, April 1960. Available from: <https://dl.acm.org/doi/10.1145/367177.367199>.
- [49] DANYAL MH. **Java is faster than the optimized Rust program**, February 2022. Available from: <https://towardsdev.com/java-is-faster-than-optimize-rust-program-bd0d1720bab2>.
- [50] CLÉMENT NERMA. **Why do values need to be moved?**, February 2021. Available from: <https://users.rust-lang.org/t/why-do-values-need-to-be-moved/55971/18>.

- [51] NIKA LAYZELL. **Implement Cgc<T> - a concurrent threadsafe garbage collector by mystor · Pull Request #6 · Manishearth/rust-gc**, September 2018. Available from: <https://github.com/Manishearth/rust-gc/pull/6>.
- [52] RAPHAEL POSS. **Abstract Machine Models - Also: what Rust got particularly right**, February 2022. Available from: <https://dr-knz.net/abstract-machine-models.html>.
- [53] THE CHROMIUM PROJECT. **Chromium security: Memory safety**. Accessed: 2022-03-21. Available from: <https://www.chromium.org/Home/chromium-security/memory-safety/>.
- [54] LIAM PROVEN. **The weird world of non-C operating systems**, March 2022. Available from: [https://www.theregister.com/2022/03/29/non\\_c\\_operating\\_systems/](https://www.theregister.com/2022/03/29/non_c_operating_systems/).
- [55] GUSTAVO RODRIGUEZ-RIVERA. **Conservative Garbage Collection for General Memory Allocators**. *SIGPLAN Not.*, 36(1):71–79, October 2000. Place: New York, NY, USA Publisher: Association for Computing Machinery. Available from: <https://doi.org/10.1145/362426.362464>.
- [56] MICHAEL SNOYMAN. **Is Rust functional?**, October 2018. Available from: <https://www.fpcomplete.com/blog/2018/10/is-rust-functional/>.
- [57] RYAN JAMES SPENCER. **The Many Uses Of The Empty Tuple**. Available from: <https://justanotherdot.com>.
- [58] PIERRE SPRING. **What is lexical scope?**, October 2010. Available from: <https://stackoverflow.com/a/2896899>.
- [59] STEVE KLABNIK AND CAROL NICHOLS. *The Rust programming language*. No Starch Press, Inc., San Francisco, CA, 5 edition, 2019.
- [60] B. STROUSTRUP. *The Design and Evolution of C++*. Programming languages/C+. Addison-Wesley, 1994. Available from: <https://books.google.de/books?id=GvivU9kGIoC>.
- [61] THE RUST TEAM. **Asynchronous Programming in Rust**, 2022. Available from: <https://rust-lang.github.io/async-book/>.
- [62] THE C++ TEAM. **const (C++)**, 2022. Available from: <https://docs.microsoft.com/en-us/cpp/cpp/const-cpp>.
- [63] THE CHROMIUM PROJECT. **Oilpan: C++ Garbage Collection**, 2021. Available from: <https://chromium.googlesource.com/v8/v8/+main/include/cppgc/README.md>.
- [64] THE CPPREFERENCE TEAM. **static members**. Available from: <https://en.cppreference.com/w/cpp/language/static>.
- [65] THE RUST CORE TEAM. **Announcing Rust 1.31 and Rust 2018**, December 2018. Available from: <https://blog.rust-lang.org/2018/12/06/Rust-1.31-and-rust-2018.html#non-lexical-lifetimes>.
- [66] THE RUST TEAM. **Announcing Rust 1.0 | Rust Blog**, May 2015. Available from: <https://blog.rust-lang.org/2015/05/15/Rust-1.0.html>.

- [67] THE RUST TEAM. **async/await Primer**, June 2022. Available from: [https://rust-lang.github.io/async-book/01\\_getting\\_started/04\\_async\\_await\\_primer.html](https://rust-lang.github.io/async-book/01_getting_started/04_async_await_primer.html).
- [68] THE RUST TEAM. **Clone in std::clone - Rust**, 2022. Available from: <https://doc.rust-lang.org/std/clone/trait.Clone.html>.
- [69] THE RUST TEAM. **Copy in std::marker - Rust**, 2022. Available from: <https://doc.rust-lang.org/std/marker/trait.Copy.html>.
- [70] THE RUST TEAM. **Declare first - Rust By Example**, 2022. Available from: [https://doc.rust-lang.org/rust-by-example/variable\\_bindings/declare.html](https://doc.rust-lang.org/rust-by-example/variable_bindings/declare.html).
- [71] THE RUST TEAM. **Destructors - The Rust Reference**, 2022. Available from: <https://doc.rust-lang.org/reference/destructors.html>.
- [72] THE RUST TEAM. **Drop in std::ops - Rust**, 2022. Available from: <https://doc.rust-lang.org/std/ops/trait.Drop.html>.
- [73] THE RUST TEAM. **Introduction - The Rust Reference**, 2022. Available from: <https://doc.rust-lang.org/reference/>.
- [74] THE RUST TEAM. **Meet Safe and Unsafe - The Rustonomicon**, 2022. Available from: <https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html>.
- [75] THE RUST TEAM. **Ownership and moves - Rust By Example**, 2022. Available from: <https://doc.rust-lang.org/rust-by-example/scope/move.html>.
- [76] THE RUST TEAM. **Pinning**, June 2022. Available from: [https://rust-lang.github.io/async-book/04\\_pinning/01\\_chapter.html](https://rust-lang.github.io/async-book/04_pinning/01_chapter.html).
- [77] THE RUST TEAM. **RAII - Rust By Example**, 2022. Available from: <https://doc.rust-lang.org/rust-by-example/scope/raii.html>.
- [78] THE RUST TEAM. **The rustc book**, 2022. Available from: <https://doc.rust-lang.org/rustc/index.html>.
- [79] THE RUST TEAM. **The Rustonomicon**, 2022. Available from: <https://doc.rust-lang.org/nomicon/>.
- [80] THE RUST TEAM. **std::rc - Rust**, 2022. Available from: <https://doc.rust-lang.org/stable/std/rc/index.html>.
- [81] THE RUST TEAM. **String in std::string - Rust**, 2022. Available from: <https://doc.rust-lang.org/std/string/struct.String.html#representation>.
- [82] THE RUST TEAM. **unit - Rust**, 2022. Available from: <https://doc.rust-lang.org/std/primitive.unit.html>.
- [83] THE RUST TEAM. **What Unsafe Can Do - The Rustonomicon**, 2022. Available from: <https://doc.rust-lang.org/nomicon/what-unsafe-does.html>.
- [84] THE RUST TEAM. **Why Async?**, June 2022. Available from: [https://rust-lang.github.io/async-book/01\\_getting\\_started/02\\_why\\_async.html](https://rust-lang.github.io/async-book/01_getting_started/02_why_async.html).



- [85] THE STACK OVERFLOW TEAM. **Stack Overflow Developer Survey 2021**, August 2021. Available from: [https://insights.stackoverflow.com/survey/2021/?utm\\_source=social-share&utm\\_medium=social&utm\\_campaign=dev-survey-2021](https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021).
- [86] GAVIN THOMAS. **A proactive approach to more secure code – Microsoft Security Response Center**, 2019. Available from: <https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/>.
- [87] TNFINK. **Yes, Rust has Garbage Collection, and a Fast One**, October 2020. Available from: <https://blog.akquinet.de/2020/10/09/yes-rust-has-garbage-collection-and-a-fast-one/>.
- [88] TOMÁŠ GAVENČIAK. **Multiple mutable references and memory bugs · Issue # 2 · mcoblentz/bronze**, October 2021. Available from: <https://github.com/mcoblentz/Bronze>.
- [89] AARON TURON. **Rust: From POPL to Practice (Keynote)**. *SIGPLAN Not.*, 52(1):2, January 2017. Place: New York, NY, USA Publisher: Association for Computing Machinery. Available from: <https://doi.org/10.1145/3093333.3011999>.
- [90] UNISYS CORPORATION. **ALGOL Programming Reference Manual, Volume 1: Basic Implementation**, 2017. Available from: <https://public.support.unisys.com/aseries/docs/clearpath-mcp-18.0/86000098-516/section-000026712.html>.
- [91] GITHUB USER LOCKA99. **Memory Management · A Guide to Porting C and C++ code to Rust**. Available from: [https://locka99.gitbooks.io/a-guide-to-porting-c-to-rust/content/memory\\_management/](https://locka99.gitbooks.io/a-guide-to-porting-c-to-rust/content/memory_management/).
- [92] PATRICK WALTON. **Removing Garbage Collection From the Rust Language**, June 2013. Available from: <https://pcwalton.github.io/2013/06/02/removing-garbage-collection-from-the-rust-language.html>.
- [93] JOSH WATZMAN. **scope - Static (Lexical Scoping)**, March 2014. Available from: <https://stackoverflow.com/a/22395580>.
- [94] AARON WEISS, OLEK GIERCZAK, DANIEL PATTERSON, AND AMAL AHMED. **Oxide: The Essence of Rust**. *arXiv:1903.00982 [cs]*, October 2021. arXiv: 1903.00982. Available from: <http://arxiv.org/abs/1903.00982>.
- [95] PAUL R. WILSON. **Uniprocessor garbage collection techniques**. In YVES BEKKERS AND JACQUES COHEN, editors, *Memory Management*, pages 1–42, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.
- [96] HUI XU, ZHUANGBIN CHEN, MINGSHEN SUN, YANGFAN ZHOU, AND MICHAEL R. LYU. **Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs**. *ACM Trans. Softw. Eng. Methodol.*, 31(1), September 2021. Place: New York, NY, USA Publisher: Association for Computing Machinery. Available from: <https://doi.org/10.1145/3466642>.
- [97] JOSHUA YANOVSKI, HOANG-HAI DANG, RALF JUNG, AND DEREK DREYER. **Ghost-Cell: Separating Permissions from Data in Rust**. *Proc. ACM Program. Lang.*, 5(ICFP), August 2021. Place: New York, NY, USA Publisher: Association for Computing Machinery. Available from: <https://doi.org/10.1145/3473597>.
- [98] BENJAMIN ZORN. **The Measured Cost of Conservative Garbage Collection**. *Software - Practice and Experience*, 23, July 1993.