Understanding the Hype: Memory Management and Error Handling in Rust

28.07.2025

Jakob Lambert-Hartmann

Contents

Introduction

In recent years the programming language Rust has been hard not to notice. Whether on StackOverflow surveys, where it has been the most loved language 9 years in a row [1], in the Linux Kernel, as the first language next to C and Assembler accepted into the codebase by Linus Torvalds [2], or on Github, where Rusts loyal fan-base seemingly rewrites every existing project with Rust [3]. The language appears to be everywhere.

While the popularity surrounding the language might be a good indicator, that Rust must make some very smart design decisions, for an outsider it might be difficult to see past the popularity and understand the reasons behind this love.

Rust has a few aspects that make it a great choice. It is a high performance language, whose runtime performance is on par with C/C++. Unlike C/C++ whose 40+ year legacy makes them cumbersome to use at times, Rust is a modern language with a streamlined experience. It incorporates functional paradigms and iterators that provide powerful functions such as `map()`, `filter()`, and `reduce()`. It also comes packaged with build tools, a test framework, and a package manager to allow for an expanding ecosystem similar to Pythons pip. But Rust does not just combine the best aspects of existing languages, with it's typing system it places a strong emphasis on memory safety and security. And it is those aspects which make Rust really stand out from other languages.

In this document we will introduce the key ideas of Rust memory management and error handling. With it, we hope to make the hype around Rust understandable. We also hope to show that writing safe and performant code can be a lot of fun.

Memory Management

In a running program, all class instances and variables need to be saved somewhere. In general there are two locations where programs save their data: The stack and the heap.

The stack usually contains variables local to a function and variables with a known size. By design the stack manages its memory automatically, meaning when calling a function, the function reserves as much space as it needs, and when the function returns, the space is automatically freed.

The heap on the other-hand contains global variables, and variables whose size is not known at compile time. In contrast to the stack, memory on the heap is not managed automatically. This means, that something needs to manage memory. Depending on the language, different entities are responsible for managing the heap. In most languages memory is either managed by the programmer (manual) or by the language runtime (garbage collector).

Manual Memory Management

The most popular languages to use manual memory management today are C and C++. In Manual memory management the programmer is responsible for managing heap data. When they want to create a new variable on the heap, they need to ask the operating system for some memory. When they are done with the variable, they need to tell the system that they no longer need it. How memory is allocated depends on the language, in C for instance it is done with the `malloc()` and `free()` functions provided by the standard C library:

```c
// Ask for enough memory to store DataType
DataType* data = malloc(sizeof(DataType));

// data no longer needed, ask to free data again
free(data);
```

With this power however, comes responsibility. Every `malloc()` needs exactly one matching `free()` call. At first this sounds very simple, but when introducing loops, if statements, nested function calls, and multiple pointers to the same data it becomes very difficult to perfectly match mallocs/frees for each variable. Mismanaging manual memory falls into the three following types of problem.

Memory Leak happens when the memory is not freed and the memory stays reserved. Over time a memory leak uses more and more space on the heap, leading to worse performance, and potentially even crashing the program or the system.

```c
// memory leak
DataType* data = malloc(sizeof(DataType));
```

Use After Free happens when the data is accessed after it is freed. This leads to undefined behavior, as it is possible that the operating system already reserved this space for some other variable.

```c
// use after free
DataType* data = malloc(sizeof(DataType));
free(data);
*data->member = 1;
```

Double Free happens when the same data is freed multiple times. This again leads to undefined behavior.

```c
// double free
DataType* data = malloc(sizeof(DataType));
free(data);
free(data);
```

Garbage Collector

To avoid the pitfalls of manual memory management most languages (Java, Python, JS, etc.) opt for a garbage collector. A garbage collector automates memory management at the cost of performance. When the programmer creates a variable which needs to be saved on heap, the programming language environment keeps track of it. In regular intervals a routine scans heap variables and checks if there are still active references in the program pointing towards this data. If nothing points to this data, it is freed. The benefit is that it is easier for the programmer, but the drawback is that the routine reduces the runtime performance. When measuring performance the garbage collector can be identified in spikes of CPU and Memory as seen in Figure 1.

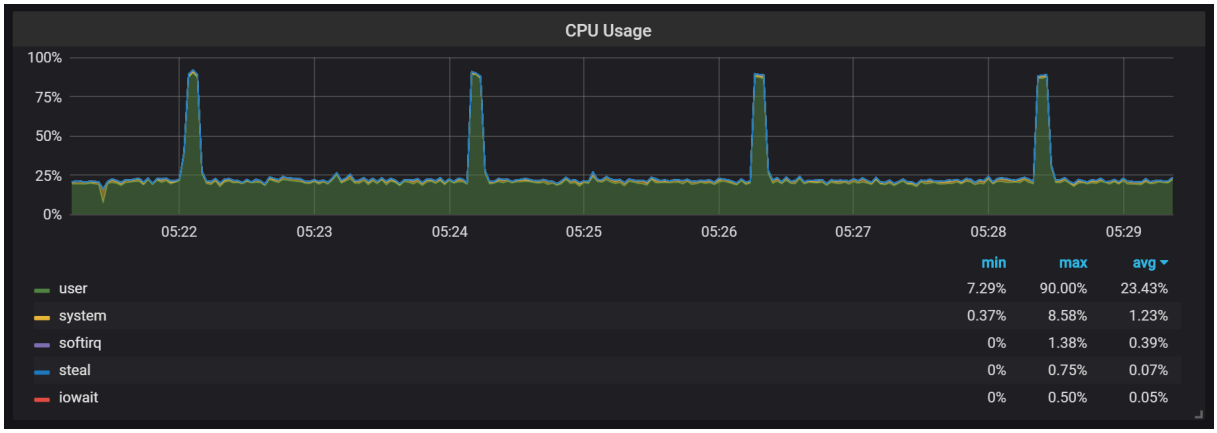

Figure 1: CPU usage spikes due to garbage collector. The CPU usage is low most of the time, there are however spikes at regular intervals. Those are caused by the garbage collector, which needs to go through the entire program memory checking for unused data [4]

Ownership

Rust takes a third approach, that is not revolutionary per se. Modern C++ already implements this way of managing memory in what is called Resource Acquisition is Initialization. The difference though, is that in C++ it is added to an existing language, while in Rust it is deeply integrated into the language.

In Rust this approach is called Ownership, and the idea is that neither the programming language environment nor the programmer need to manage memory. Instead, memory is managed by the data structures themselves. This is achieved with the following rules:

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

The following code snippet shows a very simple example of ownership, scope, and drop. The memory layout for this snippet can be seen in figure Figure 2.

```rust
fn main() {
    // create a string with content "hello" saved on the heap
    let s1 = String::from("hello")
    // s1 is owner of heap data: "hello"

}   // s1 goes out of scope: heap data "hello" is dropped
```

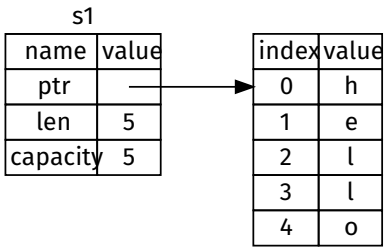

Figure 2: s1 is saved on the stack. It contains string metadata (length, capacity) and a pointer to the string data. This string data is saved on the heap. When s1 does out of scope, it drops (frees) the data on the heap. [5]

With only one variable ownership is quite straightforward. The following code snippet however, shows how ownership can become significantly more complex with just one more variable.

```rust
fn main() {
    let s1 = String::from("hello");
    // s1 is owner of heap value

    let s2 = s1;
    // who owns the heap value now?
}
```

Who owns the heap data containing “hello” now? And how does the memory layout look like? There are three possibilities how a language could handle such a case:

Shallow Copy A shallow copy would copy the data on the stack (string metadata). It would also copy the pointer to the heap data to point to the same location (see Figure 3). As mentioned in Rust a value can only have one owner, in this case however there would be two. This snippet also highlights why multiple owners are not allowed in Rust. When `s1` goes out of scope, it drops its value. When `s2` goes out of scope, then the value would already be dropped. Dropping it again would lead to a double free.

Deep Copy Another way to handle this situation would be a deep copy. With a deep copy, it would not only copy stack data, but also heap data (see Figure 4). With this approach there is no problem with Rusts ownership rules, as there is only one owner for both values. The problem with this approach starts during runtime. Depending on the data structure a deep copy might need a long time and take up a lot of memory. Imagine a string containing the content of an entire book, deep copying this string might take around a megabyte per copy. Of course there are situations where a deep copy is desired, but defaulting to it would be devastating for memory efficiency and runtime performance. Thus Rust allows for explicit deep copies, but does not perform deep copies implicitly.

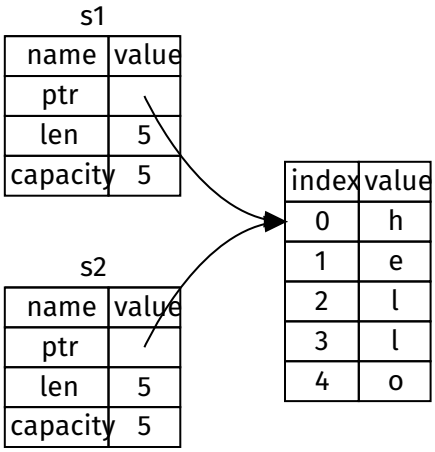

Figure 3: Shallow Copy: only data on the stack is copied (string metadata). the pointer to the heap value would still point to the same location. [5]

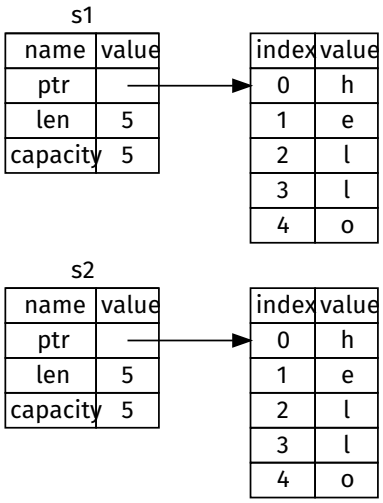

Figure 4: Deep Copy: both the data on the stack (string metadata) and the heap data (strring content) would be copied. [5]

Move Because a shallow copy does not comply with Rusts rule that there can only be one owner, and because a deep copy would severely impact performance, Rust defaults to moving values. In a move, there is no copying, and the ownership of the value is passed to the new variable. The memory layout can be seen in Figure 5. During a move, the original variable looses ownership of the value, this invalidates the variable. Using the variable after a move would cause a compile time error.

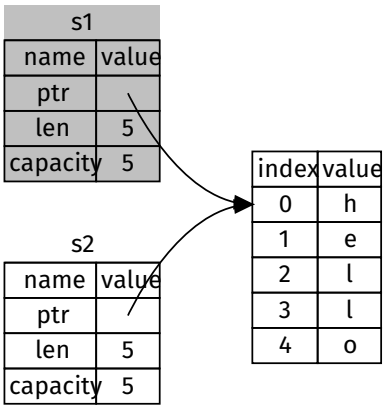

Figure 5: Move: s1 moves its ownership of “hello” to s2, becoming an invalid variable in the process. A compile error would occur when attempting to use s2 after moving ownership. [5]

Borrowing

One thing to note, is that passing variables as parameters implicitly performs a move (see Listing 1). To mitigate this, one could write functions, which also return ownership. This can be done by returning a tuple containing the return value, and the ownership as (see

Listing 2). Since this is a very common use case, Rust has a dedicated syntax for this. Rather than passing the value, only a reference is passed. This is called borrowing (see Listing 3).

```rust
fn compute_length(s: String) -> usize {
    // returns the length of a string
    return s.len();
}

fn main() {
    // s is owner of value "hello"
    let s = String::from("hello");

    // ownership of "hello" passed to get_string_length
    let len = compute_length(s);

    // Compile Error:
    // s is not valid anymore as its value has been moved
    let len2 = s.len();
}
```

Listing 1: Implicit move of function parameters. Variable cannot be used after function call because of the move.

```rust
fn compute_length(s: String) -> (usize, String) {
    // return length of string and ownership of string
    return (s.len(), s);
}

fn main() {
    // s is owner of value "hello"
    let s = String::from("hello");

    // ownership of "hello" passed to get_string_length
    let (len,s) = compute_length(s);

    // now this is possible because get_string_length
    // returned ownership to s
    let len2 = s.len();
}
```

Listing 2: Returning ownership after a function returns. This makes it possible to use the variable after it has been used as parameter.

```rust
fn compute_length(s: &String) -> usize {
    return s.len();
}

fn main(){
    // s is owner of value "hello"
    let s1 = String::from("hello");

    // value of s only borrowed to function
    let len = compute_length(&s1);

    // after get_string_length returns it also returns ownership of value
    let len2 = s.len();
}
```

Listing 3: Borrowing is a shorter syntax to return ownership to a variable after a function has returned

References

Borrowing works over references (see Listing 4). The memory layout for such references can be seen in figure Figure 6. Using references eliminates double frees, as references never drop values. With references however, use after frees could be possible, when the variable goes out of scope, while a reference to it is still in scope.

```rust
fn main() {
    // s1 is owner of value
    let s1 = String::from("hello");

    // s is a reference to s1
    let s = &s1;
}
```

Listing 4: Create references to a value

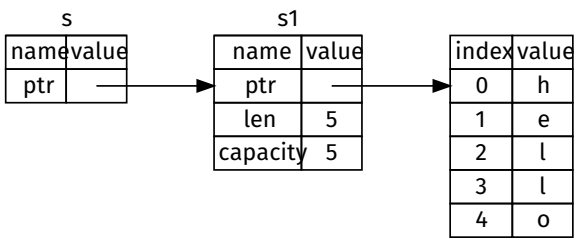

Figure 6: Reference: when s goes out of scope, the value is not dropped as s is not the owner. When s1 goes out of scope the value is dropped and s becomes invalid [5]

Lifetimes Rust is able to eliminate use after free errors with a feature called lifetimes. With Rust lifetimes, it is possible to bind the lifetime of two variables together to guarantee that a reference never lives longer than the value it is referencing. The syntax for lifetimes can be seen in Listing 5, which is unique to Rust may thus take some time to get familiar with it. In the most common use cases the compiler is able to deduce lifetimes automaticall (see Listing 5).

```rust
// both following functions are equivalent:
// with lifetime annotation
fn first_word<'a>(s: &'a str) -> &'a str {
    &s[0..1]
}

// without lifetime annotation
fn first_word(s: &str) -> &str {
    &s[0..1]
}
```

Listing 5: Lifetimes in Rust; `'a` indicates the lifetime of a variable. both the parameter `s` and the return value have this lifetime, guaranteeing that the returned reference will only be valid as long as `s` is valid. This code serves only as example, for this function lifetime annotations are not required, as the compiler is able to deduce them automatically.

There are cases however, where the compiler is not able to deduce the lifetimes automatically. Functions with multiple references (Listing 6) and Datastructures (Listing 7) for instance.

```rust
// does not compile, needs annotation
fn first(x: &i32, y: &i32) -> &i32 {
  x
}
```

Listing 6: This function does not compile without lifetime annotation, as the Rust compiler does not know whether the return value needs to live as long as `x`, or `y`.

```rust
// references x and y need to live longer than instance of DataType
struct DataType<'a> {
    x: &'a i32,
    y: &'a i32,
}
```

Listing 7: DataType instances have two `i32` references. The lifetimes are needed so that the compiler knows, that both the references `x` and the reference `y` need to live longer than the instance that is referencing them.

Further Guaranties

As shown in the previous sections Rust is able to eliminate the three issues with manual memory management: memory leak, double free, and use after free. However, it is also providing far more memory guaranties than this. To highlight the power of Rusts memory guaranties we will first introduce mutability.

Mutability In Rust all variables are immutable by default. To change variables, they need to be explicitly set as mutable (see Listing 8). The same goes for References (see Listing 9). It is possible to have as many immutable references to a variable in Rust, however, if there is one mutable reference to a variable, this becomes the only valid reference to it (see Listing 10).

```rust
// keyword mut specifies that the string value of s can be changed
let mut s = String::from("hello");

// changing the string data is only possible because s is mutable
s.clear();
```

Listing 8: Variables are immutable by default and can explicitly be set as mutable

```rust
let mut s = String::from("hello");

// borrowing s as a mutable reference
let s1 = &mut s;

// changing the string data only possible because both the string,
// and the reference are mutable
s1.clear();
```

Listing 9: References are immutable by default and can explicitly be set as mutable

```rust
let mut s = String::from("hello");

let r1 = &s;
let r2 = &mut s;

// compile error: r1 immutable reference and r2 mutable reference
// if r2 is a mutable reference it has to be the only reference
println!("{}, {}", r1, r2);
```

Listing 10: If there is a mutable reference it needs to be the only reference

Because there can only be one mutable reference at a time, this guarantees that references in Rust always point to valid data. In other languages where multiple references to the same value are allowed, access to invalid data can happen when one reference modifies the data in such a way that other references now point to invalid data. Those errors are especially

hard to debug, as they might not always cause faulty behavior (see Listing 11). In Rust such a usage invalidates previous references, leading to an error at compile time, rather than a runtime error (see Listing 12).

```cpp
int main() {
    std::string str = "Hello";

    // Get a pointer to string data
    const char* ptr = str.c_str();

    // Invalidate the pointer by clearing the string
    str.append(" World");
    // code compiles without any issues
    // BUT: using ptr after this line is undefined behavior

    // depending on capacity of str, ptr may still be valid.
    // if capacity large enough for " World"    -> ptr still valid
    // otherwise string moved to a new location -> ptr invalid
}
```

Listing 11: C++: multiple mutable references causing invalid data access

```rust
fn main() {
    let mut s = String::from("hello world");

    // immutable reference to the first 5 characters of string s
    let word = &s[..5];

    // mutably borrows s to clear content
    s.clear();

    // compile error:
    // word has become invalid, as it is an immutable reference to s
    // s.clear() mutably borrowed s invalidating other references
    println!("First word: {}", word);

}
```

Listing 12: Security guarantees from limiting mutable references

Error Handling

Error Handling, besides memory safety, is another standout feature of Rust. To understand Rusts approach, it is important to know that there are three types of errors that can occur when programming. Depending on the type, they might be easier or harder to detect and fix. The types of errors are:

Logic Error: Those happen in the brain of the programmer They occur when the problem statement is either misunderstood, or poorly communicated. These may be the hardest errors to detect, as they compile and run without any obvious failure, except for a wrong result. While the syntax of a language can help reduce those errors, this effect is rather subjective.

Compilation Errors: Those occur when attempting to compile bad code. The Language detects syntax errors or other instances with incorrect usage (such as calling a non-existent class method, using wrong parameters, ...). Those errors are easiest to detect, as the language detects them for the programmer.

Runtime Errors: Occur during program execution. Those errors are hard to detect, as the language does not detect them. Additionally, they might not necessarily occur on each program execution. There is a multitude of runtime errors, among the most common ones are:

1. Accessing invalid data (e.g. array out-of-bound, accessing nullpointers)
2. Calling a function that can fails (e.g. opening a nonexistent file)
3. performing invalid arithmetic (e.g. division by zero)

In this chapter we will describe how Rust helps with error handling, by turning runtime errors into compile time errors. Before moving to Rust, we will quickly highlight how other languages help programmers to avoid runtime errors, using the three common runtime errors (invalid data access, failing function calls, and invalid arithmetic) and C++ and Java as examples.

Error Handling in C

In C error handling is done by the programmer. There is little help, and programs can fail in multiple ways. Often, the language does not warn or fail to compile. The following snippets show how C fails to support programmers handling our example runtime errors, even in the simplest of cases:

Accessing invalid data By default, C does not help prevent access to invalid data. Some compilers have flags to protect against buffer overflow (e.g. `-fstack-protector` for `gcc`) to protect against attack writing into invalid data, but this is not on by default.

```c
int[2] arr = [1,2];
int el = arr[2];
// undefine behavior from here on out
```

Calling a function that fails Detecting a failed function call is not uniform in C. Some functions return `NULL` on failure (like `fopen()`), others return `0` on success (like `system()`), some set a global `errno` for extra info, and some encode the error in the return value itself. The programmer is responsible for reading how each function might fail.

```c
char* path = 'file/does/not/exist';
FILE* f = fopen(path);
// undefine behavior from here on out
```

Performing invalid arithmetic Division by zero is undefined behavior in C. The programmer is responsible for checking that the divisor is not zero.

```c
int a = 1;
int b = 0;
int c = a / b;
// undefine behavior from here on out
```

Error Handling in Java

In stark Contrast to C, Java helps programmers handling errors and provides a dedicated syntax and data type: Exceptions. Exceptions can be thrown in any part of the program. There two types of exceptions: checked exceptions, which need to be checked at compile time, and unchecked exceptions, which don't need to be checked to successfully compile. With our example runtime errors, the following exceptions would be thrown:

1. accessing an array out of bounds throws an `ArrayIndexOutOfBoundsException`
2. opening a nonexistent file throws a `FileNotFoundException`
3. dividing by zero throws an `ArithmeticException`

While much more helpful than C, Javas syntax still lacks in a few aspects:
1. All variables can be `nullpointers`, thus every variable needs to be checked
2. The syntax is cumbersome, and it is tempting to write empty `try/catch` blocks

Error Handling in Rust

Rust, similar to Java, helps developers to handle errors in a cleanly. But it does go further than Java, as almost all runtime errors are moved to compile time errors. Furthermore, it's typing system eliminates common pain points with Javas error handling. Before getting into our three runtime error examples, and how Rust deals with them, it is important to quickly introduce the `panic!()` function. Calling this function immediately exists the program safely. This is used in the case of unrecoverable errors.

Invalid Data Access

Part of how Rust helps with invalid data access was already discussed in the memory management chapter. In Rust, references always point to valid data. This is due to the restriction on mutable references. There are however also other scenarios which attempt to access invalid data, reading beyond the length of an array for instance. To exemplify why Rusts approach is so special however, we will discuss Rusts approach with one of the most common invalid data problems: null pointers.

In some scenarios it makes sense to have a special value to represent the absence of an object. For instance, a data structure containing information on a person might contain the field `home_phone_number`. Some people however don't have a phone at home. In this case it would make perfect sense to dedicate a special value, to specify that the person has no phone. Most other languages use `null` for this (C/C++/Java/...). The problem though, is that in those languages everything or next to everything can be `null`. In Java all possible variables, while in C/C++ all pointers can be `null`. Because all values can theoretically be `null`, but most practically can't, those languages usually don't enforce null checks, as it would be too much boilerplate code. In C/C++/Java you can access methods of a pointer/reference without first checking that this reference is valid.

In Rust on the other hand, no value can be empty unless explicitly allowed. With this approach only values that have a realistic chance of being empty can be empty. As such Rust forces the programmer to check for emptiness before accessing values. This is achieved with Enums, specifically with the Option Enum. Enums are structures in Rust, which can be in one of multiple named states (Variant). The Option Enum for instance can either contain a value `Some(value)` or be empty `None`.

```rust
// Option of type T
// can either be empty value
// or a value of T
enum Option<T> {
    None,
    Some(T),
}
```

To access data from an Enum, it needs to be unpacked into it's current Variant (see Listing 13). Because Options are commonly used, Rust implements methods to reduce boilerplate when working with Options (`unwrap()`, `unwrap_or_default()`, `expect()`. See Listing 14 for usage).

```rust
 fn main() {
     // call function which returns optional string
     let s:Option<String> = get_some_string();

     if Some(string) = s {
         // this code only runs if there is a string in s

         // only after unpacking to string
         // is it possible to use string methods
         let index:Option<usize> = string.find('hello');
     }

     if None = s {
         // this code only runs if there is no string in s
     }
}
```

Listing 13: Unpacking Option into different variants

```rust
fn main() {
    // call function which returns optional string
    let s:Option<String> = get_some_string();

    // if there is not string in s default to "hello"
    let string = s.unwrap_or("hello");

    // if there is not string in s exit
    let string = s.unwrap();

    // if there is not string in s exit with error message
    let string = s.expect("no string in s");
}
```

Listing 14: Methods implemented by Rust for the Option type to simplify common use cases and reduce boilerplate code

Rust does not only use `Option<T>` to remove the need for `NullpointerExceptions`, it also uses them for all other kinds of invalid data. The hash-map method `find()`, the Vector method `get()`, and the string method `find()` all return option types to indicate if the requested data even exists.

Failing Function Calls

Handling failing function calls is quite similar to accessing invalid data. However, with functions there is some additional complexity, as there are multiple possible reasons for a function to fail. Opening a file could fail because the file does not exist, or because the user does not have read access. To accommodate for this, Rust implements another enum `Result`. It is similar to `Option<T>` but allows for an error type.

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

The benefit over Option, is that the Error state can also carry information. Similar to exceptions in Java, this information can tell you why the function failed. Unlike Java though, there is no unchecked exception, all results need to be extracted into either a result on success, or an error on failure. Again Rust implements methods for common use-cases such as `unwrap()`, `unwrap_or_default()`, `expect()`, etc.

Invalid Arithmetic

The last runtime error consisted of invalid arithmetic. In contrast to the other two errors Rust trades off some security against convenience, for arithmetic operations. Just like in C and Java, it is possible to perform any operation without checking for edge cases.

```rust
let a = 1;
let b = 2;

// overflow/underflow possible without warning
let c = a + b;
let d = a - b;
let e = a * b;
// this panics if b == 0
let f = a / b;
```

However, while the default operations may cause a program to panic, Rust provides a range of methods for cases, where security is needed. For every arithmetic edge case Rust implements multiple wrappers. The following table shows a subset of addition methods for the `u32` type:

function	return	description
`checked_add()`	`Option<u32>`	returns `None` on overflow
`carrying_add()`	`(u32, bool)`	returns carry
`overflowing_add()`	`(u32, bool)`	returns wrapped value + overflow bit
`wrapping_add()`	`u32`	returns wrapped value
`strict_add()`	`u32`	panics on overflow

Conclusion

Rust is able to provide many security guarantees, both in memory safety, and in error handling. This is achieved for two reasons that work hand in hand. On the one hand Rust requires that the programmer checks for edge cases for everything that can go wrong. On the other hand Rusts syntax is very explicit, greatly the reducing the number of lines where something can go wrong (variables and references are immutable by default, empty values are not possible unless explicitly stated, functions cannot fail unless explicitly set in the return type, etc.)

In combination this makes for a language with great security, but also with great performance, as the compiler can make more assumptions during optimization.

Bibliography

[1] “Technology | 2024 Stack Overflow Developer Survey — survey.stackoverflow.co.” [Online]. Available: https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language-prof

[2] “Rust; The Linux Kernel documentation — docs.kernel.org.” [Online]. Available: https://docs.kernel.org/rust/index.html

[3] J. Neander, “Rewritten in Rust: Modern Alternatives of Command-Line Tools · Zaiste Programming — zaiste.net.” [Online]. Available: https://zaiste.net/posts/shell-commands-rust/

[4] P. Zaitsev, “Prometheus 2 Times Series Storage Performance Analyses — percona.com.” [Online]. Available: https://www.percona.com/blog/prometheus-2-times-series-storage-performance-analyses/

[5] “The Rust Programming Language - The Rust Programming Language — doc.rust-lang.org.” [Online]. Available: https://doc.rust-lang.org/stable/book/