# About the project

We aim at creating a substantial program in a programming language that we have not already studied, with focus on features that distinguish the specific programming language from other programming languages. The programming language in question will be Rust which seems to be a serious alternative to C/C++. Above all, it is promoted for guaranteeing memory and thread safety, thus preventing pitfalls that have been common in C/C++ [1].

One aspect of our project will be comparing Rust features to some characteristics of the C language. In doing so, we want to find out how specific features have been realized and why those design decisions have been made. Therefore, we plan to rewrite the GNU sort utility by employing Rust’s distinct features while comparing it to the baseline utility.

# The GNU sort utility

# Context of Rust

The Rust developers categorize Rust as a systems programming languages [1] that is most significantly inspired by C and ML languages. Thus, it is also influenced by concepts originating from the functional programming community which results in a mixture of low level programming with high level features.

Similarly to C, Rust provides “fine control over memory layout and predictable performance” [Mozilla Paper]. However, the need for a language like Rust has arisen from observing the behaviour of applications written in older systems programming languages: As an informal study of the Mozilla Research team shows, use after free, out of range access or related to integer overflow constitute half of the bugs in their Firefox engine that is written in C++. [Mozilla Paper] Another source, the Common Vulnerabilities and Exposures database, reports “217 vulnerabilities […] in the Linux Kernel, two-thirds of which can be attributed to the use of an unsafe language” [Beyond Safety]. Unsafe programming languages like C or C++ are still widely used because they hold the advantage of great performance that cannot be achieved with safe programming languages that rely on garbage collection. Latter programming languages like Java can only determine at runtime when data can be freed plus passing references to a function holds the risk of unforeseen changes to the data, especially when threading is involved [intorust, ownership.

Due to these findings, Rust tackles the issues concerning memory safety, mostly by following the concept of ownership (also known as affine type system). This static type system enables the compiler to recognize common pitfalls that could lead to “data races, buffer overflows, stack overflows, and accesses to uninitialized or deallocated memory” [the rust language] or other memory errors like dangling pointers or double frees. Thus, safe memory management is guaranteed without the overhead of garbage collection. As another consequence, Rust provides a safer environment for concurrency. [Mozilla Paper] In addition to that, the design of Rust facilitates prediction and reasoning about performance. [The Rust Language]

However, the ownership model also holds disadvantages such as prohibiting the implementation of data structures like doubly-linked lists. Therefore, an unsafe subset of Rust has been introduced that does not strictly adhere to the ownership rules. However, unsafe operations have to be explicitly indicated in designated blocks so that programmers have to be aware of producing unsafe code.

Another downside is that Rust exhibits some weaknesses inherent to new programming languages that might let it seem less attractive than established programming languages, for example no/little IDE support, a small ecosystem or a high learning curve. The Rust community has recognized these deficiencies and intends to tackle them [https://internals.rust-lang.org/t/setting-our-vision-for-the-2017-cycle/3958]. Nonetheless, the user base of Rust appears to have been growing fast since its first stable release in 2015 [wikipedia]. Recent application fields encompass Mozilla’s browser engine Servo, cloud services or backend tasks among many others [FoR/2].

# Technical aspects

## Memory management

As indicated above, the ownership model constitutes the core of Rust’s safe memory management. Ownership means the “unique access and control of data” [intorust, Ownership]. Whenever an object is created, the creator becomes the unique owner of the object. Once allocated, the resource can be moved to another owner, whereby the former owner loses the access rights to this object. Once the owner of an object returns and the object has not been moved or used otherwise, the data gets dropped (thus deallocated from the stack or heap) automatically.

Rust further distinguishes between copy types and non-copyable types. The former is always implicitly copied when referenced whereas the latter is only handed from one owner to another. To overcome this restriction, the concept of borrowing has been introduced that denotes passing around references which also belong to the copy types.

By default, variables in Rust are immutable and have to be declared mutable otherwise. Accordingly, references are also immutable by default, thus they are only readable not writeable. This variant is also called a “shared borrow”. This enables creating multiple references to the same data. However, both sides, the owning as well as the borrowing instance have to indicate who is owning the data and who is borrowing. If a variable has been declared mutable and a callee borrows that data, the variable becomes immutable until the reference goes out of scope again.

Nonetheless, mutable borrows are possible as well. Those references allow no other readers and only one writer. That means while being in scope, a mutable reference can exclusively access the data of the variables they are referring to. Thus, it is often helpful to only borrow anonymous references to helper function, so that the reference is just in scope for that call. Thus, the original variable can be used further in the course of the program.

## Enumeration and Pattern matching

Another powerful and handy tool that Rust provides are enumerations and pattern matching whereby the latter is yet another example for the inspiration taken from the functional programming languages. Whereas an if-condition needs to evaluate to a Boolean value, any type can serve as a pattern. Several match arms – each with a pattern and some code – make up a match expression. The first match arm where its pattern matches the given expression is executed. This functionality is especially useful in combination with enumerations so that different variants of an enumeration can lead to different consequences for the execution of the program. Furthermore, variables can be bound to the data inside a variant. This functionality, especially in combination with Enums, seems to resemble the define-type and type-case functionality in Racket.

## Error Handling

Whereas C does not provide error handling, Rust does just as other modern high-level programming languages do. Errors are divided into two categories: recoverable and unrecoverable errors. The former can be solved by the user, whereas the latter indicates a bug. Instead of exceptions though, Rust provides the type Result <T,E> for recoverable errors and the panic! macros for its counterpart.

# Potential project + value/importance