Introduction to Rust

Alastair Droop, 2023-02-01

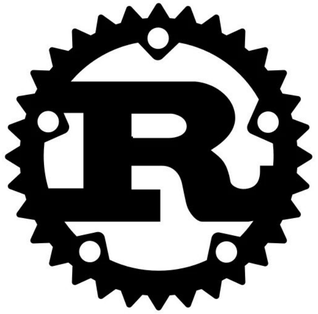
“Rust is a modern systems programming language focusing on safety, speed, and concurrency. It accomplishes these goals by being memory safe without using garbage collection.”

Created in 2006 by Graydon Hoare whilst at Mozilla (used for Mozilla servo)

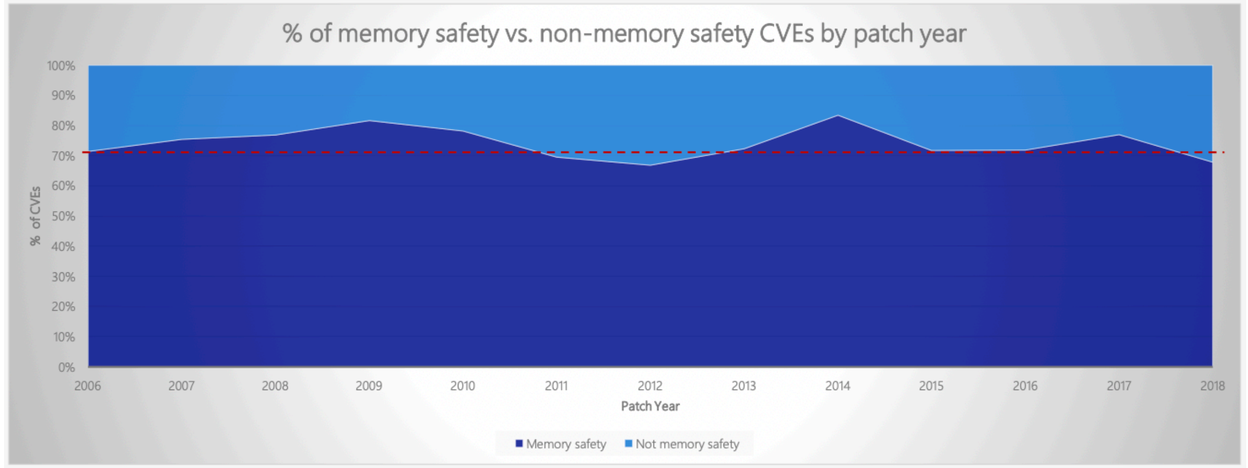
Based strongly on older languages (Nil, Erlang, Limbo, etc…) (“Nothing New”)

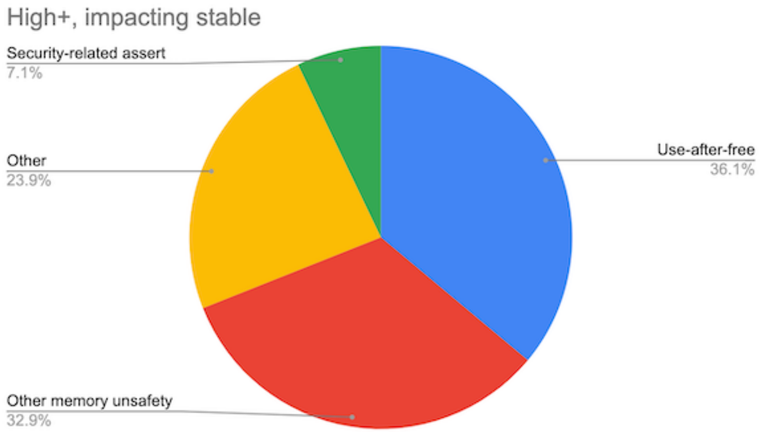
Stack Overflow developer survey “most loved language” every year since 2016

Second language adopted for writing the Linux kernel (v6.1, in October 2022)



Memory Safety is A Big Deal





Memory safety bugs account for

- ~70% of all vulnerabilities addressed through a Microsoft security update each year [1]
- ~70% of all serious bugs in the Google Chromium project [2]

Memory safety is a legacy of C (well, ALGOL)'s "Billion dollar mistake" [3]

[1]: https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/
[2]: https://www.chromium.org/Home/chromium-security/memory-safety
[3]: http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

Rust's Type System

Rust is *statically typed*, so the compiler must know what type al variables are at compile time

- However, the compiler is very good at inferring types, so often you don't need to worry
- For example, a vector of 64-bit floats would be `Vec<f64>`
- A hash map mapping string keys to 64-bit unsigned integers would be `HashMap<String, u64>`

Collections have powerful & sensible methods that allow "pythonic" manipulations (iterators, *etc*)Strings (`String`) are totally different to vectors of characters, and are separate to string references (`&str`):

```
let a:String = String::from("Hello, World!");
let b:&str = &a;
```

Generic types allow us to write code that can work for multiple types

- We can set boundaries on which types can be used by specifying which traits a generic type must possess

Structs & Enumerations

Structs allow related values to be packaged together into a meaningful group:

```
struct Read {
    header: String,
    sequence: String,
    quality: Vec<f32>,
}
```

Enums list *all* possible variants of a value:

```
enum MessageResult {
    Success,
    Error(String),
}
```

Pattern Matching

Match expressions branch on a given pattern. The pattern must be complete

- Missing pattern options are compiler errors
- A “catch-all” arm value of “_” is allowed that can cover all other values

```
enum Temperature {
    Kelvin(f32),
    Celsius(f32),
    Fahrenheit(f32),
    Rankine(f32),
}

fn to_absolute(temp: Temperature) -> f32 {
    match temp {
        Temperature::Kelvin(t) => t,
        Temperature::Celsius(t) => t + 273.15_f32,
        Temperature::Fahrenheit(t) => (t + 459.67_f32) * (5_f32 / 9_f32),
        Temperature::Rankine(t) => t * (5_f32 / 9_f32),
    }
}
```

Ownership

Run-time garbage collection is often used to track application memory and avoid these errors

- However, GC is very slow, requires a significant runtime, and vastly complicates parallel software development

Rust uses a completely different approach that makes most memory safety bugs compiler errors:

- Each value in has an owner
- There can *only be one owner* at a time
- When the owner goes out of scope, the value is dropped

```
let a = String::from("Hello, World!");
let b = a;
println!("{}", a);
```

References & The Borrow Checker

If we want to refer to a variable without getting ownership of it, we can borrow it:

```
let a = String::from("Hello, World!");
let b = &a;
println!("{}", a);
```

- We can create any number of immutable references to a variable
- We can only ever have a single mutable reference at a time

The Rust compiler makes sure that ownership rules are obeyed. This system is called the “borrow checker”

Borrow checker failures are compiler errors

Exception Handling & Optional Values

Without the concept of NULL, how do we represent a failed or empty result?

- The standard library provides two enums to help here: `Option<T>` and `Result<T, E>`

The `Option<T>` represents either some value (of generic type T), or None:

```
enum Option<T> {
    None,
    Some(T),
}
```

`Result<T, E>` enum represents either a success value (of generic type T) or a failure (of generic type E):

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

Rust is Compiled

By default, Rust binaries are statically linked. This means that there are usually very few dependencies.

When creating a rust crate, you specify the versions of all dependencies you need.

- The compiler compiles all dependencies in the crate you're building
- This means that compilation is (relatively) slow
- Compilation requires relatively large amounts of space
- The exact versions of all dependencies in your binary are defined and don't change

This negates the need for tools like conda

Very simple to generate the complete tree of dependencies with version data ("`cargo tree`")

Testing & Documenting Rust is Easy

Documentation generated using “///”

Examples containing code will be run as tests

Tests can live with the code being tested

Command	Action
`cargo build`	Compile the current crate
`cargo run`	Run the current crate
`cargo clippy`	Run the “Clippy” linter
`cargo fmt`	Reformat all code into a standard style
`cargo test`	Run all tests
`cargo doc`	Compile the crate documentation

```rust
/// Temperature scales
///
/// Allows different temperature scales to be used without confusion
/// See [Wikipedia](https://en.wikipedia.org/wiki/Temperature).
///
pub enum Temperature {
    Kelvin(f32),
    Celsius(f32),
    Fahrenheit(f32),
    Rankine(f32),
}

/// Converts temperatures to an absolute value in Kelvin
/// # Examples
///
/// ```
/// # use example_rust::*;
/// assert_eq!(to_absolute(Temperature::Celsius(0_f32)), 273.15_f32);
/// assert_eq!(to_absolute(Temperature::Celsius(100_f32)), 373.15_f32);
/// ```
pub fn to_absolute(temp: Temperature) -> f32 {
    match temp {
        Temperature::Kelvin(t) => t,
        Temperature::Celsius(t) => t + 273.15_f32,
        Temperature::Fahrenheit(t) => (t + 459.67_f32) * (5_f32 / 9_f32),
        Temperature::Rankine(t) => t * (5_f32 / 9_f32),
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_zero() {
        assert_eq!(to_absolute(Temperature::Kelvin(0_f32)), 0_f32);
        assert_eq!(to_absolute(Temperature::Celsius(-273.15_f32)), 0_f32);
        assert_eq!(to_absolute(Temperature::Fahrenheit(-459.67_f32)), 0_f32);
        assert_eq!(to_absolute(Temperature::Rankine(0_f32)), 0_f32);
    }
}
```

UNIVERSITY of York

So, Why Rust?

Rust has a few features that prevent memory bugs and help ensure reliable & reproducible code

- Values are statically typed (this is good)
- Strict ownership model prevents memory errors and data races
- Asynchronous & parallel code is easy & safe to write
- Simple testing and documentation
- Strict error handling
- Unsafe code (for example FFI) is clearly marked with the `unsafe` keyword
- Zero-cost high-level abstractions (*e.g.* iterators) feel like Python
- Code is statically compiled, removing most(?) dependency issues
- Code is semantically versioned
- Cross compilation is easy
- Installation is usually trivial

Useful Links

The Rust homepage	https://www.rust-lang.org/
The Rust Book	https://doc.rust-lang.org/book/
The Cargo Book	https://doc.rust-lang.org/cargo/
The Rustlings Course	https://github.com/rust-lang/rustlings/
Rust by Example	https://doc.rust-lang.org/stable/rust-by-example/
The Rust Standard Library	https://doc.rust-lang.org/std/
The Rust community crate registry	https://crates.io/

Live Demonstration

Let's pretend we need to build a program to find all palindromic sequences of length *n* in a nucleotide sequence

- The palindrome length *n* is fixed
- A palindrome is defined as a string of nucleotides directly followed by its reverse complement
- The query sequence is loaded from a FASTA file
- We need to loop through all windows of length *n* across the entire sequence
- We test each window and return a list of the starting index and the sequence to the user

We need to be able to run in sensible time across large input sequences

Both speed and memory efficiency are important

Demo code is at `https://github.com/uoy-research/tf-202301-rust-workshop/`