Networking in rust

Ryan & Julian

Background

- Networking is a speed/throughput restricted field.
  - Languages used are C and C++, and other systems languages
  - Necessary because the system usually cannot be bogged down by GC, etc.
  - Code has to run on a lot of embedded and low power hardware because for some reason networking is where costs need to be saved.
  - Security is of utmost importance

- 70% of all security bugs are memory safety issues (Microsoft)

The rust language

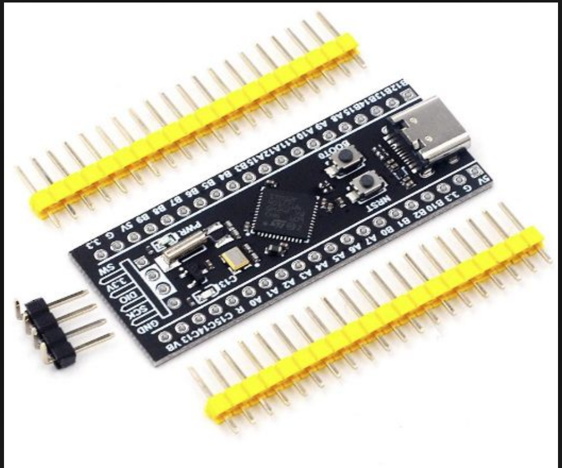
Impossible to introduce memory safety errors without `unsafe` code.

Compiles to native machine code, and matches speed of C most of the time.

Can be compiled to run on low power embedded processors.

Has no garbage compiler, and does not use manual

memory management



THE OWNERSHIP SYSTEM

- All data is “owned”. (Prevents data access from multiple locations)

```rust
struct StringOwner {
    string: String,
}

fn main() {
    let string_data: String = "Hello, World!".into();

    let owner = StringOwner {
        string: string_data,
    };

    // println!("{}", string_data); // WILL NOT COMPILE BECAUSE DATA IS NO LONGER OWNED BY `string_data`
    println!("{}", owner.string); // this works because data is `owned by owner`
}
```

BORROWING DATA

- Data may be “borrowed” many times without having to transfer owners.

```rust
struct StringBorrower<'a> {
    string: &'a String,
}

fn main() {
    let mut string_data: String = "Hello, ".into();
    string_data += "World!"; // strings can be concatenated

    let borrower = StringBorrower {
        string: &string_data,
    };

    println!("{}", string_data); // WILL COMPILE BECAUSE DATA IS STILL OWNED BY `string_data`
    println!("{}", borrower.string); // this still works because data is 'borrowed'

    // THIS WILL NOT COMPILE BECAUSE THE & MEANS IMMUTABLE (READ ONLY) REFERENCE
    // borrower.string += "test";
}
```

BORROWING DATA AS MUTABLE

- Data may be “borrowed” mutably with tight restrictions. (again prevents race conditions)

```rust
struct StringBorrower<'a> {
    string: &'a mut String,
}

fn main() {
    let mut string_data: String = "Hello, ".into();
    string_data += "World!"; // strings can be concatenated

    let borrower = StringBorrower {
        string: &mut string_data,
    };

    // println!("{}", string_data); // NO LONGER COMPILES BECAUSE MUTABLE ACCESS CANNOT BE SHARED
    println!("{}", borrower.string); // this still works because data is 'borrowed'

    // THIS WILL WORK NOW THOUGH SINCE WE HAVE MUTABLE ACCESS
    *borrower.string += "test";
}
```

BOUNDS CHECKING

- Data structures (like arrays and structs) are always bound checked by the compiler. This prevents buffer and data overruns.
- Even better is that bounds checking is mostly done AT COMPILE TIME.

```rust
fn main() {
    let array = [1, 2, 3, 4, 5]; // array type is [i32; 5]

    // WILL NOT COMPILE BECAUSE COMPILER IS DOING BOUNDS CHECKS
    // array[5] = 6;
}
```

POINTERS

- Pointers are explicitly disallowed in rust because they can be used to perform unsafe actions. Direct pointer access is generally considered a bad thing now days.
- Pointers and pointer logic can be used if absolutely necessary

```rust
fn main() {
    let mut string: String = "Hello".into();

    let same_string: &mut String;
    unsafe {
        let string_ptr = string.as_mut_ptr() as *mut String;
        same_string = &mut *string_ptr;
    }

    // It can be done to have multiple mutable access, but it is unsafe
    println!("{}", string);
    println!("{}", same_string);
}
```

EXAMPLE

- We built a TCP Server in rust to show its aptitude.
- Compiles to machine code and can run on the metal.
- In this example we use some items from the standard rust library, but those could be swapped out, and this would even run on low power embedded systems.

Conclusion

- Memory safety is important and prevents many security errors.
- `rust` is surprisingly adept at keeping your code safe, while still being performant.
- The future of programming is definitely provable and memory safe programming.

References

Klabnik, S., & Nichols, C. (2018). The rust programming language. No Starch Press.

Cartas, C. (2019). Rust - the programming language for every industry. Academy of Economic Studies.Economy Informatics, 19(1), 45-51. https://doi.org/10.12948/ei2019.01.05

THANK YOU