How Rust Programs Programmers

$whoami

Dawid Królak

- General purpose software engineer
- Rust enthusiast
- Passionate about European space exploration and engineering programs.
- Core member of AeroRust community.

https://twitter.com/@Taavicjusz

https://github.com/taavit

When you program in Rust Rust programs you!

Rust Toolbox

Rust Toolbox

- cargo
- fmt
- clippy

It’s not about that

Rust Toolbox

- immutability
  - mut is more than a keyword
- ownership
  - variables as something to carry

It’s about that!

Hello world!

Hello world!

```
fn main() {
    let a = 0;
    println!("{a}");
    a = a + 1;
    println!("{a}");
}
```

Hello world!

```
  Compiling playground v0.0.1 (/playground)
error[E0384]: cannot assign twice to immutable variable `a`
--> src/main.rs:4:5
  |
2 |     let a = 0;
  |         - first assignment to `a`
3 |     println!("{a}");
4 |     a = a + 1;
  |     ^^^^^^^^^ cannot assign twice to immutable variable
  |
help: consider making this binding mutable
  |
2 |     let mut a = 0;
  |         +++

For more information about this error, try `rustc --explain E0384`.
```

Hello world!

```
fn main() {
    let a = String::from("Hello ");
    a += "GDG!";
    println!("{a}");
}
```

Hello world!

```
  Compiling playground v0.0.1 (/playground)
error[E0596]: cannot borrow `a` as mutable, as it is not declared as mutable
--> src/main.rs:3:5
  |
3 |     a += "GDG!";
  |     ^ cannot borrow as mutable
  |
help: consider changing this to be mutable
  |
2 |     let mut a = String::from("Hello ");
  |         +++

For more information about this error, try `rustc --explain E0596`.
```

Hello ownership!

Hello ownership!

```
#include <string>
#include <iostream>

void foobar(std::string a) {
  std::cout<<"Hello "<<a<<"!"<<std::endl;
}

int main() {
  std::string a = "GDG";
  for (int i=0; i<100; i++) {
      foobar(a);
  }
}
```

Hello ownership!

```
fn main() {
    let a = String::from("GDG");
    for _ in 0..100 {
        foobar(a);
    }
}

fn foobar(a: String) {
    println!("Hello {a}!");
}
```

Hello ownership!

```
  Compiling playground v0.0.1 (/playground)
error[E0382]: use of moved value: `a`
--> src/main.rs:4:16
  |
2 |     let a = String::from("GDG");
  |         - move occurs because `a` has type `String`, which does not implement the `Copy` trait
3 |     for _ in 0..100 {
  |     --------------- inside of this loop
4 |         foobar(a);
  |                ^ value moved here, in previous iteration of loop
  |
note: consider changing this parameter type in function `foobar` to borrow instead if owning the value isn't necessary
--> src/main.rs:8:14
  |
8 | fn foobar(a: String) {
  |    ------    ^^^^^^ this parameter takes ownership of the value
  |    |
  |    in this function
help: consider moving the expression out of the loop so it is only moved once
  |
3 ~     let mut value = foobar(a);
4 ~     for _ in 0..100 {
5 ~         value;
  |
help: consider cloning the value if the performance cost is acceptable
  |
4 |         foobar(a.clone());
  |                ++++++++

For more information about this error, try `rustc --explain E0382`.
```

Disclaimer #1: Copy vs Clone

- Copy
  - Applicable whenever **memcpy** would be enough.
  - Primitives types by default (i32, f64, bool)
  - Structs “marked” with “Copy” trait
    - Only if all fields are copyable
    - Copy “attribute” have to be added explicit for non-std structs.
- Clone
  - Structs “marked” with Clone trait
  - Means - I need to something extra:
    - Allocate memory
    - Increment atomic counter

Hello ownership! - Good

```
note: consider changing this parameter type in function `foobar` to borrow instead if
owning the value isn't necessary
--> src/main.rs:8:14
  |
8 | fn foobar(a: String) {
  |    ------    ^^^^^^ this parameter takes ownership of the value
  |    |
  |    in this function
```

Hello ownership! - Good

```
fn main() {
    let a = String::from("GDG");
    for _ in 0..100 {
        foobar(&a);
    }
}

fn foobar(a: &String) {
    println!("Hello {a}!");
}
```

Hello ownership! - Bad

```
help: consider moving the expression out of theloop so it is only moved once
   |
3  ~      let mut value = foobar(a);
4  ~      for _ in 0..100 {
5  ~          value;
```

Hello ownership! - Bad

```
fn main() {
    let a = String::from("GDG");
    let value = foobar(a);
    for _ in 0..100 {
        value;
    }
}

fn foobar(a: String) {
    println!("Hello {a}!");
}
```

Hello ownership! - Ugly

```
help: consider cloning the value if the performance cost is acceptable
  |
4 |         foobar(a.clone());
  |                ++++++++
```

Hello ownership! - Ugly

```
fn main() {
    let a = String::from("GDG");
    for _ in 0..100 {
        foobar(a.clone());
    }
}

fn foobar(a: String) {
    println!("Hello {a}!");
}
```

Hello ownership!

```
#include <iostream>
#include <memory>
#include <string>

void foo(std::unique_ptr<std::string> s) {
    (*s) += "x";
    std::cout << "Inside foo: " << *s << std::endl;
}

int main() {
    auto s = std::make_unique<std::string>("Some string");

    for (int i = 0; i < 1000; ++i) {
        foo(std::move(s));
    }
}
```

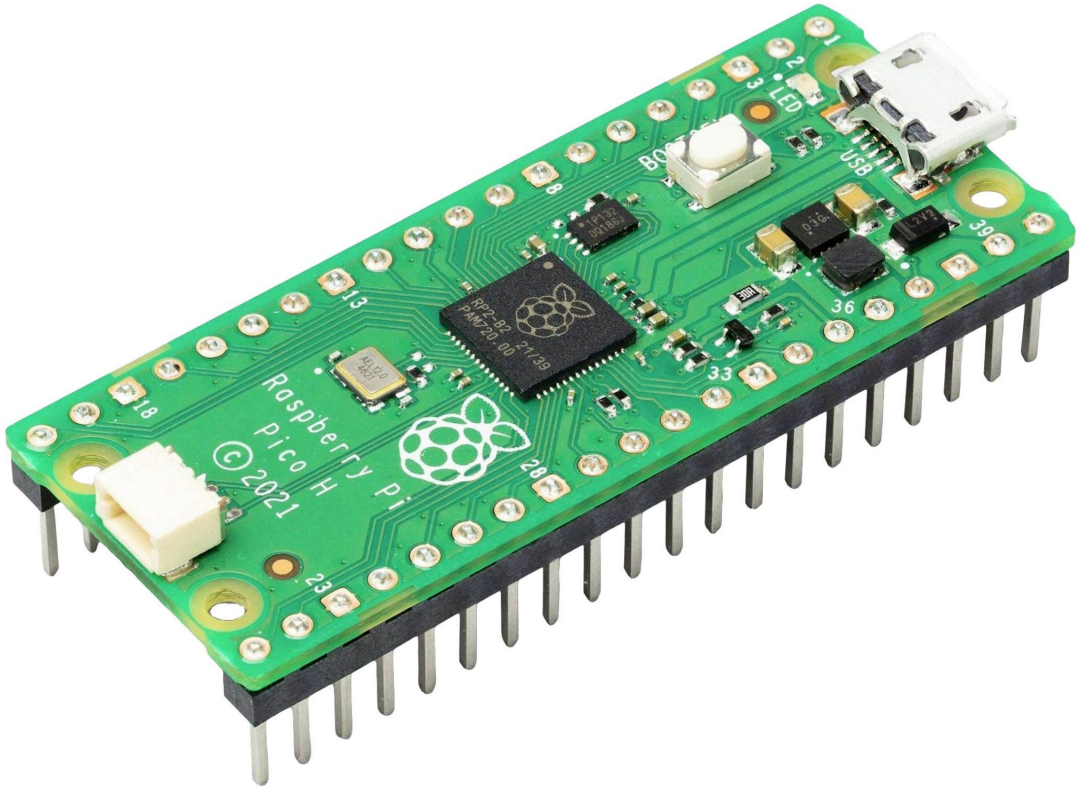
Hello ownership! - Weird

```
fn main() {
    let mut a = String::from("GDG");
    for _ in 0..100 {
        a = foobar(a);
    }
}

fn foobar(a: String) -> String {
    println!("Hello {a}!");
    a
}
```

Ownership in the wild

Embedded



Embedded

```
struct GPIO1;

struct FooDevice {
   pin: GPIO1,
}
struct BarDevice {
   pin: GPIO1,
}

fn main() {
   let p = GPIO1;

   let f = FooDevice { pin: p };
   let b = BarDevice { pin: p };
}
```

Why not borrow mutable?

Disclaimer #2: & and &mut

Reference

Multiple read only references.

XOR

Mutable reference

One mutable reference.

Embedded

```
struct GPIO1;

struct FooDevice {
   pin: GPIO1,
}
struct BarDevice {
   pin: GPIO1,
}

impl FooDevice {
   fn release(self) -> GPIO1 {
       self.pin
   }
}

fn main() {
   let p = GPIO1;

   let f = FooDevice { pin: p };
   let p = f.release(); // f is dropped, pin is returned to scope
   let b = BarDevice { pin: p };
}
```

AppState

```
struct AppState {
   active_clients: HashMap<SessionId, ClientActivity>,
}
impl AppState {
   fn new_state() -> Self {
       Self {
           active_clients: HashMap::new(),
       }
   }
}
fn foo() {
   let state = AppState::new_state();
}
```

AppState

```
struct AppState {
   active_clients: Mutex<HashMap<SessionId, ClientActivity>>,
}

impl AppState {
   fn new_state() -> Self {
       Self {
           active_clients: Mutex::new(HashMap::new()),
       }
   }
}

fn foo() {
   let state = Arc::new(AppState::new_state());
}
```

Wait…
One thing…

Ownership of primitive

```
let mut action_counter: u64 = 0;
```

Ownership of primitive

```
struct ActionCounter(u64);
```

DO NOT REWRITE WORLD TO RUST JUST BECAUSE!

Questions


Ok(())