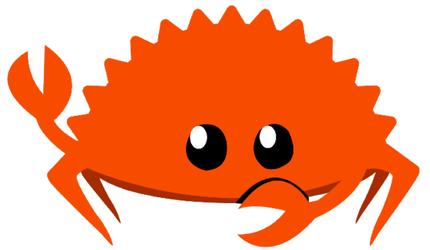
The Rust Programming Language

Tech Spotlight 2019-05-01

Rich Infante

"**Rust** is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety. "



Features?

zero-cost abstractions

guaranteed memory safety

move semantics

pattern matching

type inference

trait-based generics

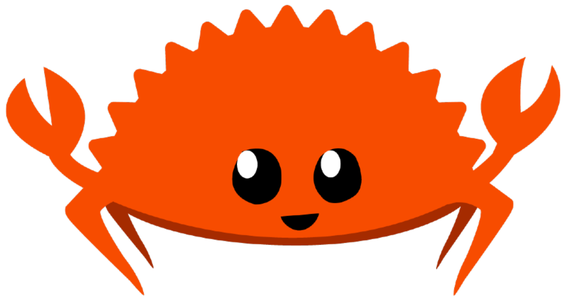
minimal runtime

efficient C bindings

threads without data races

Hello, World!

```
fn main() {
    println!("Hello, World!");
}
```

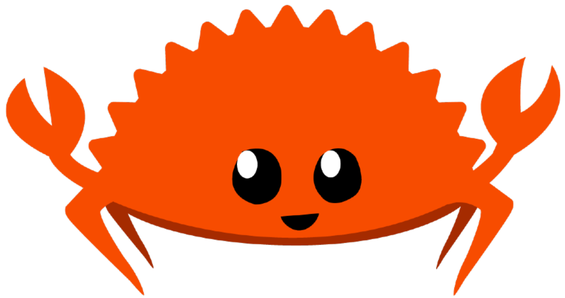


A "Quick" Overview...


Let's jump in!
gifbin.com


Let's jump in!
gifbin.com

Variables



Variables

```
// Number
let luggage_combination = 12345;
```

Variables

```
// Number
let luggage_combination = 12345;

// Boolean
let mut is_airshield_open = false;
```

Variables

```
// Number
let luggage_combination = 12345;

// Boolean
let mut is_airshield_open = false;

// Text
let planet_name = "Druidia";
```

Logic Statements

```
// Simple Statements (check for number ranges, values)
x > 100
x == 200
x <= 300
y == true

// Compound Statements (combine them)
x == 100 && y == 200
x == 200 || (y == 200 && z == 200)
```

Structs

```
struct Planet {
  /// Name of the planet
  name: String,

  /// Store the airshield combiation
  airshield_combination: u64,

  /// Is airshield open?
  is_airshield_open: bool
}
```

Structs

```
fn main() {
  // Create a planet.
  let druidia = Planet {
    name: "Druidia".to_string(),
    airshield_combination: 12345,
    is_airshield_open: false
  };
}
```

Statements

```
// Run code depending on the value of some
statement
let y = true;
if y {
  println!("Y is true");
} else {
  println!("Y is not true!!");
}
```

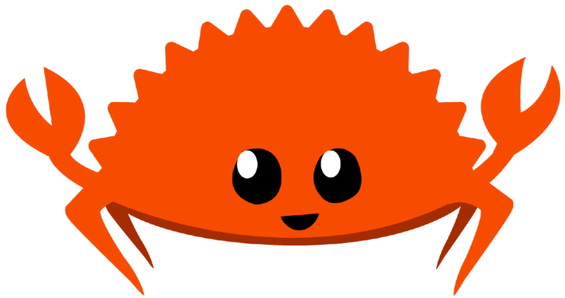
Statements

```
// Declare a variable
let mut x = 120;

// Repeatedly do something
while x > 100 {
  println!("X = {}", x);
  x -= 1;
}

// Another way...
loop {
  if x <= 100 {
    break;
  }

  println!("X = {}", x);
  x -= 1;
}
```

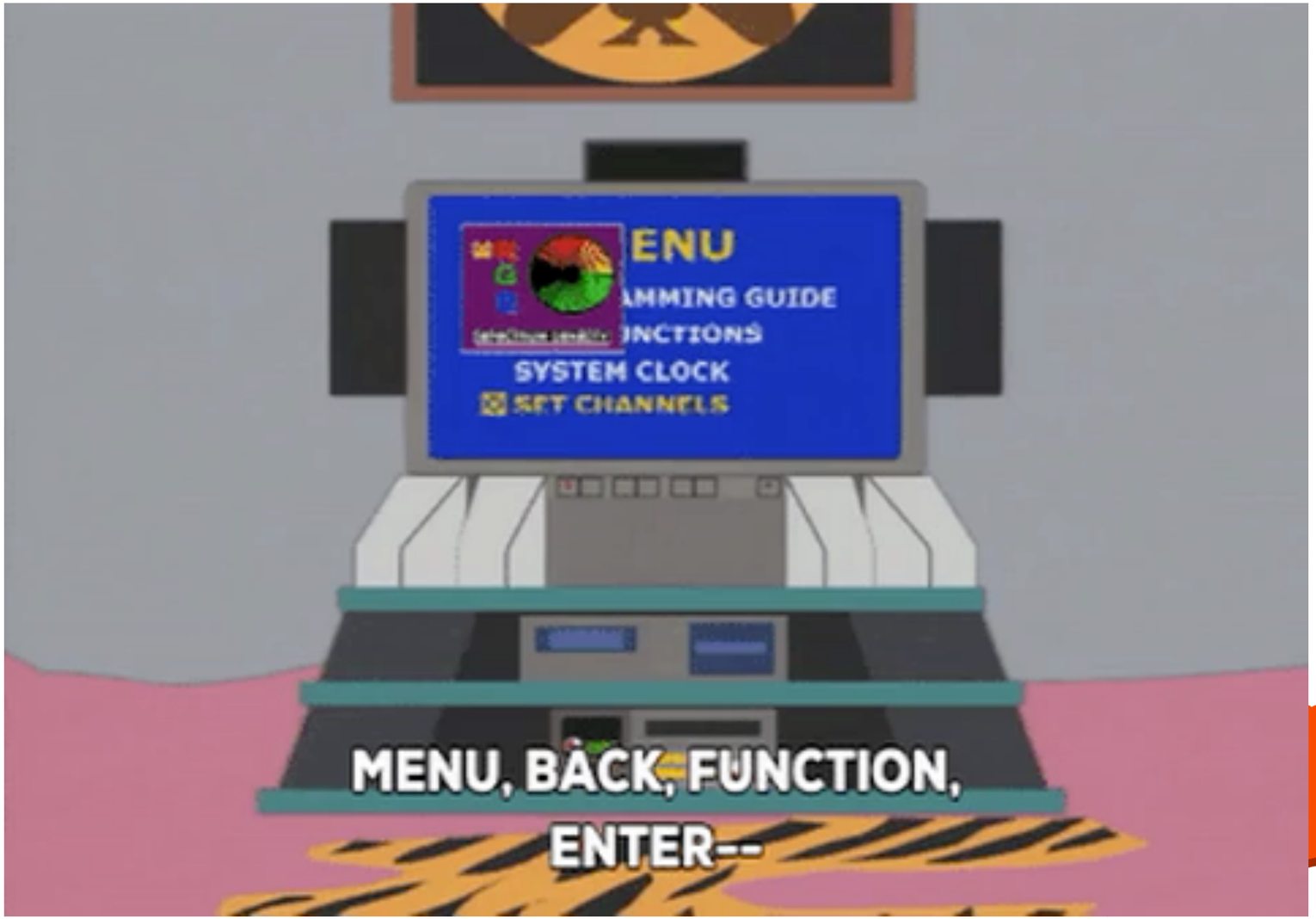
Statements

```
// Store a restaurant rating.
let restaurant_rating = 5;

// Pattern matching statement.
let result = match restaurant_rating {
  1 | 2 => "Bad",
  3 => "Ok",
  4 => "Good",
  5 => "Excellent",

  // Number isn't a star rating - unknown.
  _ => "Unknown"
};

// Print to screen
println!("Rating is {} stars ({})", restaurant_rating, result);
```

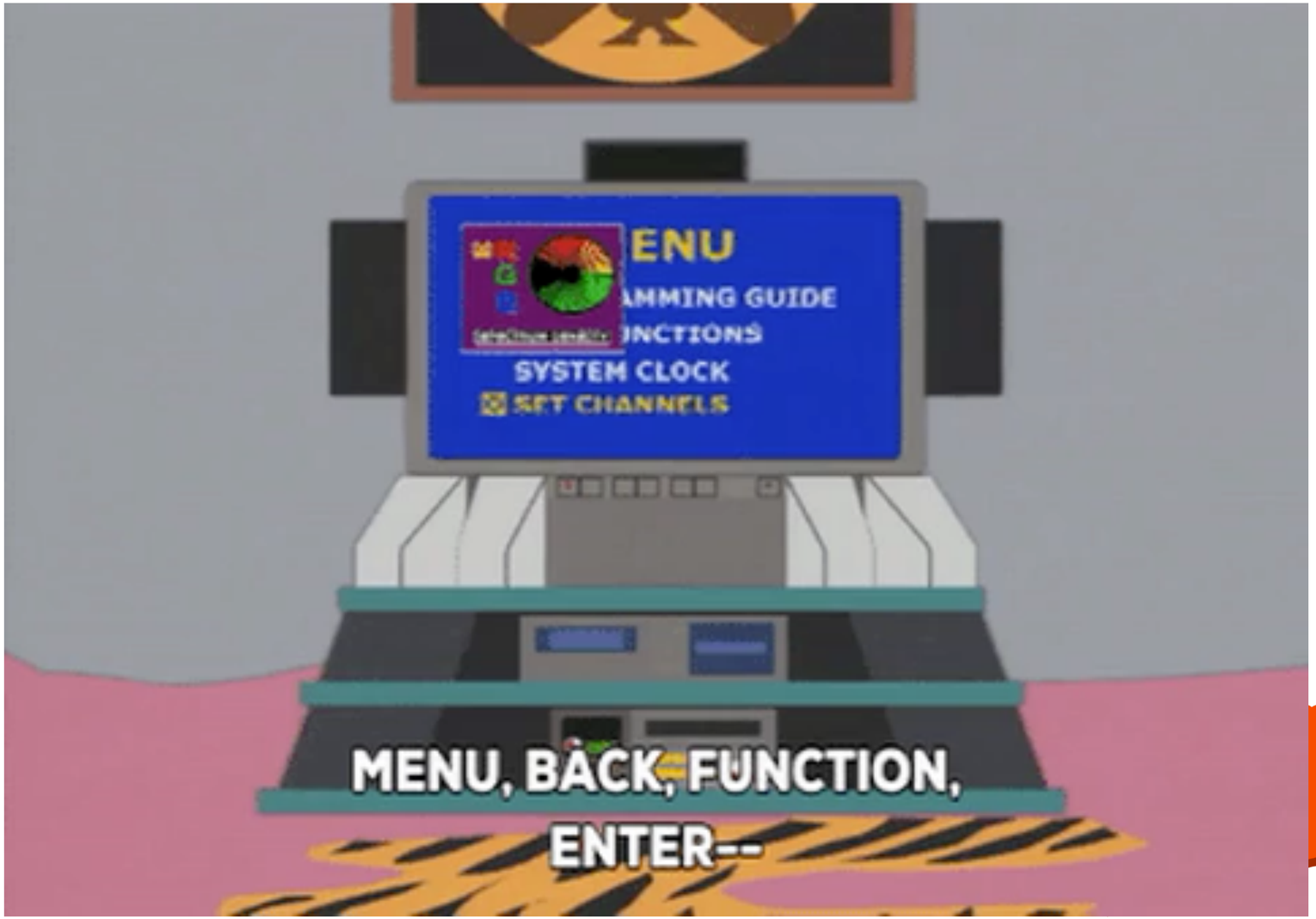
Functions & Methods



Functions & Methods



Functions & Methods



Functions

```
/// Check the combination for an airshield.
fn check_combination(passcode: u64) -> bool {
  if passcode == 12345 {
    return true
  } else {
    return false
  }
}
```

Methods

```
impl Planet {
  /// Check the combination for an airshield.
  fn check_combination(&self, passcode: u64) -> bool {
    if passcode == self.airshield_combination {
      return true
    } else {
      return false
    }
  }
}
```

Methods

```
fn main() {
  // Create a planet.
  let druidia = Planet {
    name: "Druidia".to_string(),
    airshield_combination: 12345,
    is_airshield_open: false
  };

  if druidia.check_combination(12345) {
    println!("Combination correct!");
  }
}
```

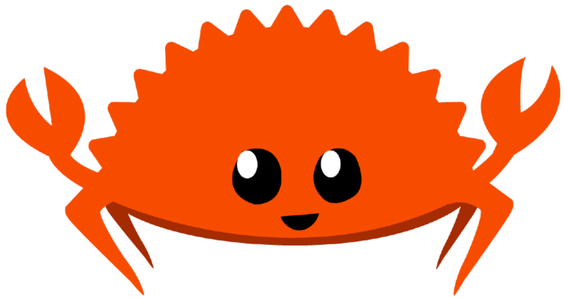
Block Rules

```
/// Check the combination for an airshield.
fn check_combination(passcode: u64) -> bool {
  if passcode == 12345 {
    true
  } else {
    false
  }
}
```

Block Rules

```
/// Check the combination for an airshield.
fn check_combination(passcode: u64) -> bool {
  if passcode == 12345 {
    true; [!] Expected to return bool
  } else {
    false; [!] Expected to return bool
  }
}
```

Error Handling



Rust Error Handling

```rust
/// Rust's Standard Error Type
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Rust Error Handling

```rust
// A function that returns an error type.
fn failable(param: u64) -> std::result::Result<u64, ()> {
  if param < 100 {
    // If the number's less than 100, add 100 to it.
    Ok(param + 100)
  } else {
    // Otherwise, We fail as an error.
    Err(())
  }
}
```

Rust Error Handling

```
fn main() {
  // Check result value
  if let Ok(res) = failable(40) {
    println!("got result: {}", res);
  } else {
    println!("got error!");
  }
}
```

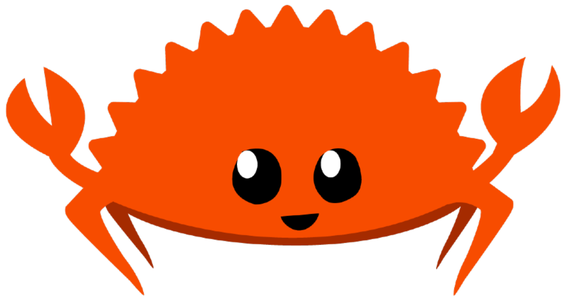
Rust Error Handling

```
fn main() {
  // Perform a match on the result
  match failable(50) {
    Ok(res) => println!("result: {}", res),
    Err(_) => println!("Got an error.")
  }
}
```

Optionals

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

fn something() -> Option<u64> {
  return Some(1337)
}

fn something_else() -> Option<u64> {
  return None
}
```

Traits



Traits

- Zero-Cost Abstraction
- Statically and Dynamically dispatched
  - Statically dispatched traits are compiled into separate copies depending on types used
  - Dynamic traits are dispatched at runtime
- Use as interfaces, markers, allow overloading of operators, functions.

```
use std::fmt;

// a re-implementation of std::fmt:Display trait.
trait Display {
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error>;
}
```

```rust
// We have some struct
struct Point {
    x: i32,
    y: i32,
}

// Implement the trait
impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // write! writes to anything implementing std::io::Write
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

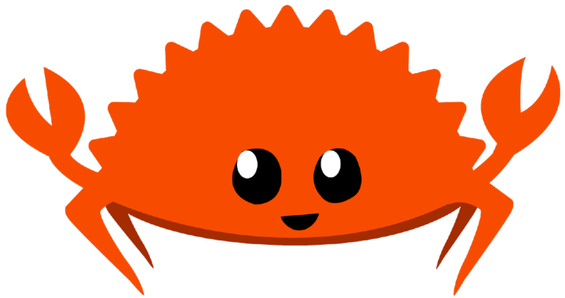
```
fn main() {
  let origin = Point { x: 0, y: 0 };
  println!("The origin is: {}", origin);
}
```

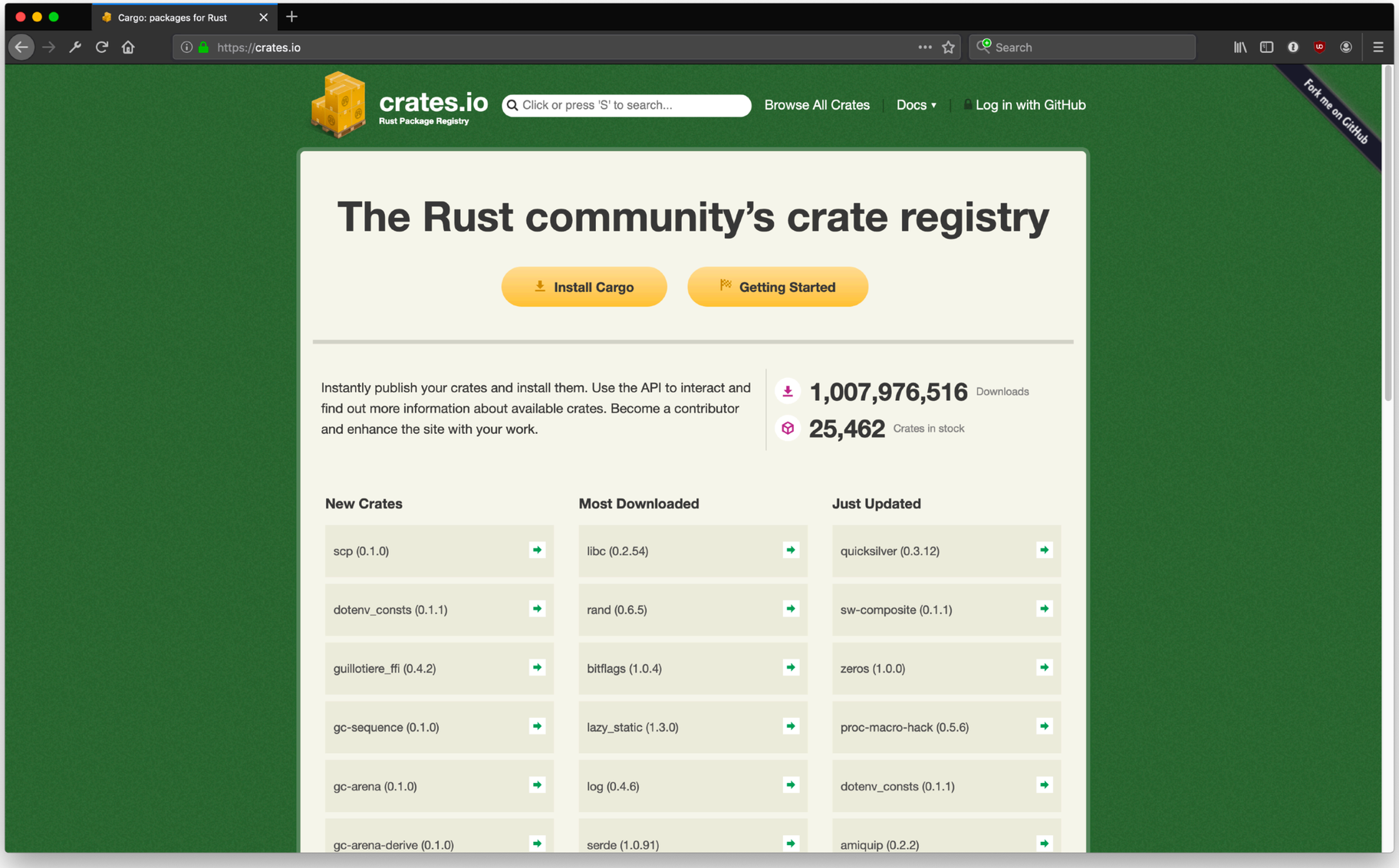
Some Traits in standard library:

```
std::convert::{From, Into}
std::fmt::Display
std::string::ToString;
std::borrow::Borrow
std::iter::Iterator
```

https://stevedonovan.github.io/rustifications/2018/09/08/common-rust-traits.html

Crates & Modules




Cargo: packages for Rust
https://crates.io
Search
crates.io
Rust Package Registry
Click or press 'S' to search...
Browse All Crates
Docs
Log in with GitHub
Fork me on GitHub
The Rust community's crate registry
Install Cargo
Getting Started
Instantly publish your crates and install them. Use the API to interact and find out more information about available crates. Become a contributor and enhance the site with your work.
1,007,976,516 Downloads
25,462 Crates in stock
New Crates
scp (0.1.0)
dotenv_consts (0.1.1)
guillotiere_ffi (0.4.2)
gc-sequence (0.1.0)
gc-arena (0.1.0)
gc-arena-derive (0.1.0)
Most Downloaded
libc (0.2.54)
rand (0.6.5)
bitflags (1.0.4)
lazy_static (1.3.0)
log (0.4.6)
serde (1.0.91)
Just Updated
quicksilver (0.3.12)
sw-composite (0.1.1)
zeros (1.0.0)
proc-macro-hack (0.5.6)
dotenv_consts (0.1.1)
amiquip (0.2.2)

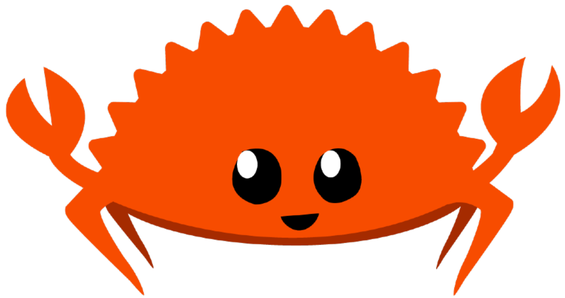
Cargo.toml

```
[package]

name = "satellite-tracking"

version = "0.1.0"

authors = ["Rich Infante <rich@richinfante.com>"]

edition = "2018"


[dependencies]

satellite = { path = "../satellite-rs" }

# { git = "https://github.com/richinfante/satellite-rs.git", branch = "master" }

chrono = "0.4"

reqwest = "0.9.12"

space-plot = { path = "../space-plot" }

rand = "0.6.5"
```

Using Modules

```
use std::fmt::Formatter;

use chrono::prelude::*
```

Ownership




Here's where it get a bit complicated...


Here's where it get a bit complicated...

Ownership in Rust

- You may have seen the & and * operators in C for referencing / dereferencing variables.
- They are used in a similar manner in rust, but with some more restrictions.

Ownership in Rust

- In Rust, you can either have an "Owned" or "Borrowed" reference to an object.
- Syntax is similar to C-pointers
- However, it actively prevents you from doing bad things.
- Usually, code that would break thread safety or other expectations won't even **compile**.

```rust
struct Food {
    name: String
}

// Eat food - takes ownership of the variable.
fn eat_food(meal: Food) {
    println!("ate food: {}", meal.name);
}

fn main() {
    let pizza = Food { name: "Pizza".into() };
    eat_food(pizza);
}
```

```
struct Food {
    name: String
}

// Eat food - takes ownership of the variable.
fn eat_food(meal: Food) {
    println!("ate food: {}", meal.name);
}

fn main() {
    let pizza = Food { name: "Pizza".into() };
    eat_food(pizza);
    eat_food(pizza);
}
```

```
struct Food {
    name: String
}

// Eat food - takes ownership of the variable.
fn eat_food(meal: Food) {
    println!("ate food: {}", meal.name);
}

fn main() {
    let pizza = Food { name: "Pizza".into() };
    eat_food(pizza);
    eat_food(pizza);
}
```

WILL NOT COMPILE!

```
struct Food {
    name: String
}

// Eat food - takes ownership of the variable.
fn eat_food(meal: Food) {
    println!("ate food: {}", meal.name);
}

fn main() {
    let pizza = Food { name: "Pizza".into() };
    eat_food(pizza); <-- Moved Here
    eat_food(pizza); [!!] Use of moved value
}
```

WILL NOT COMPILE!

```
struct Food {
    name: String
}

// Eat food - takes ownership of the variable.
fn eat_food(meal: Food) {
    println!("ate food: {}", meal.name);
}

fn main() {
    let pizza = Food { name: "Pizza".into() };
    eat_food(pizza); <-- Moved Here
    eat_food(pizza); [!!] Use of moved value
}
```

WILL NOT COMPILE!

```rust
struct Food {
    name: String
}

// Eat food - takes ownership of the variable.
fn eat_food(meal: Food) {
    println!("ate food: {}", meal.name);
}

// Inspect food - borrows the variable.
fn inspect_food(meal: &Food) {
    println!("inspected food: {}", meal.name);
}

fn main() {
    let pizza = Food { name: "Pizza".into() };
    inspect_food(&pizza);
    eat_food(pizza);
}
```

```rust
struct Food {
    name: String
}

// Eat food - takes ownership of the variable.
fn eat_food(meal: Food) {
    println!("ate food: {}", meal.name);
}


// Inspect food - borrows the variable.
fn inspect_food(meal: &Food) {
    println!("inspected food: {}", meal.name);
    eat_food(*pizza);
}

fn main() {
    let pizza = Food { name: "Pizza".into() };
    inspect_food(&pizza);
    eat_food(pizza);
}
```

```rust
struct Food {
    name: String
}

// Eat food - takes ownership of the variable.
fn eat_food(meal: Food) {
    println!("ate food: {}", meal.name);
}


// Inspect food - borrows the variable.
fn inspect_food(meal: &Food) {
    println!("inspected food: {}", meal.name);
    eat_food(*pizza); [!!] Cannot move out of borrowed context.
}

fn main() {
    let pizza = Food { name: "Pizza".into() };
    inspect_food(&pizza);
    eat_food(pizza);
}
```

WILL NOT COMPILE!

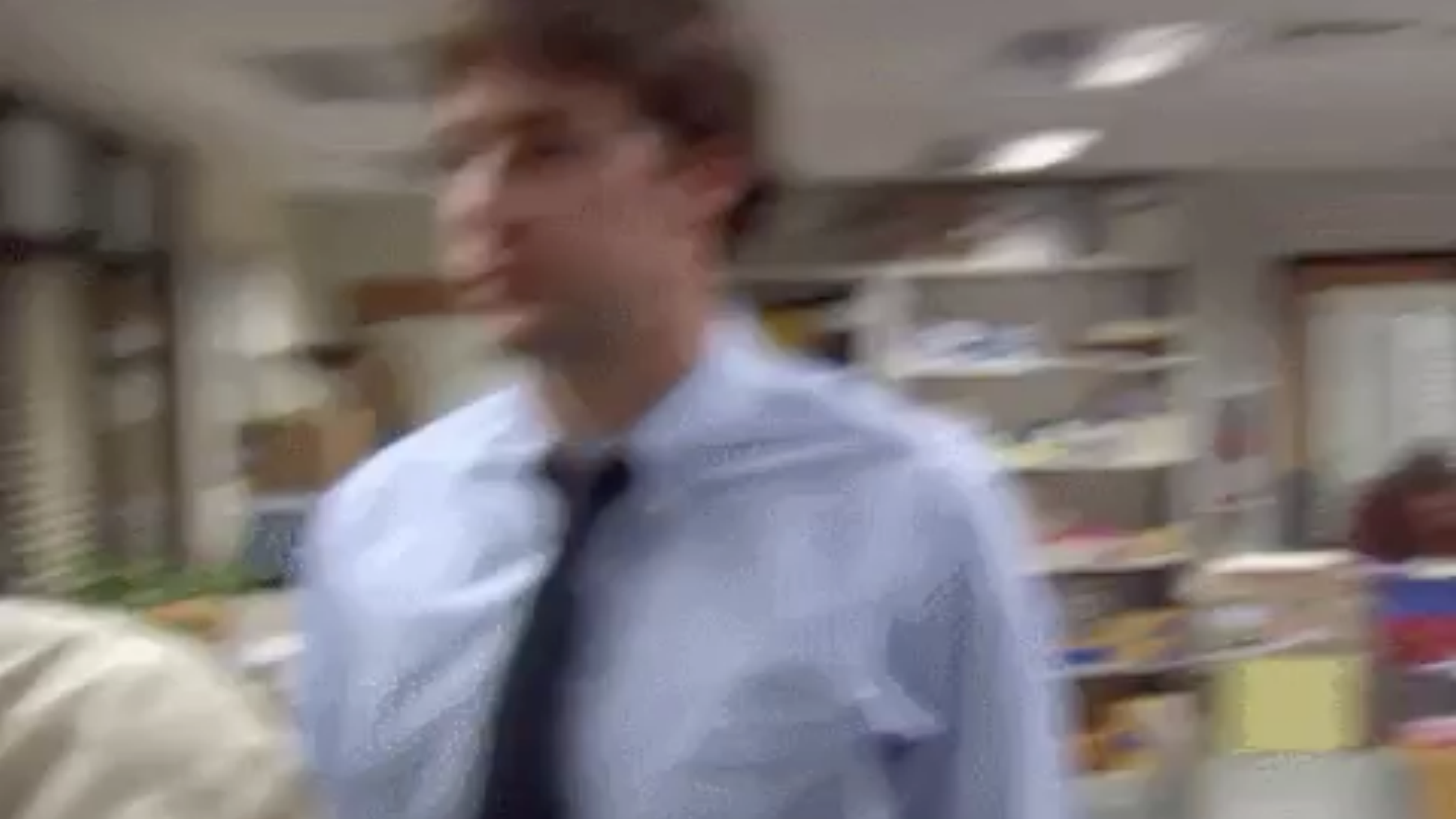
```rust
struct Food {
    name: String
}

// Eat food - takes ownership of the variable.
fn eat_food(meal: Food) {
    println!("ate food: {}", meal.name);
}


// Inspect food - borrows the variable.
fn inspect_food(meal: &Food) {
    println!("inspected food: {}", meal.name);
    eat_food(*pizza); [!!] Cannot move out of borrowed context.
}

fn main() {
    let pizza = Food { name: "Pizza".into() };
    inspect_food(&pizza);
    eat_food(pizza);
}
```

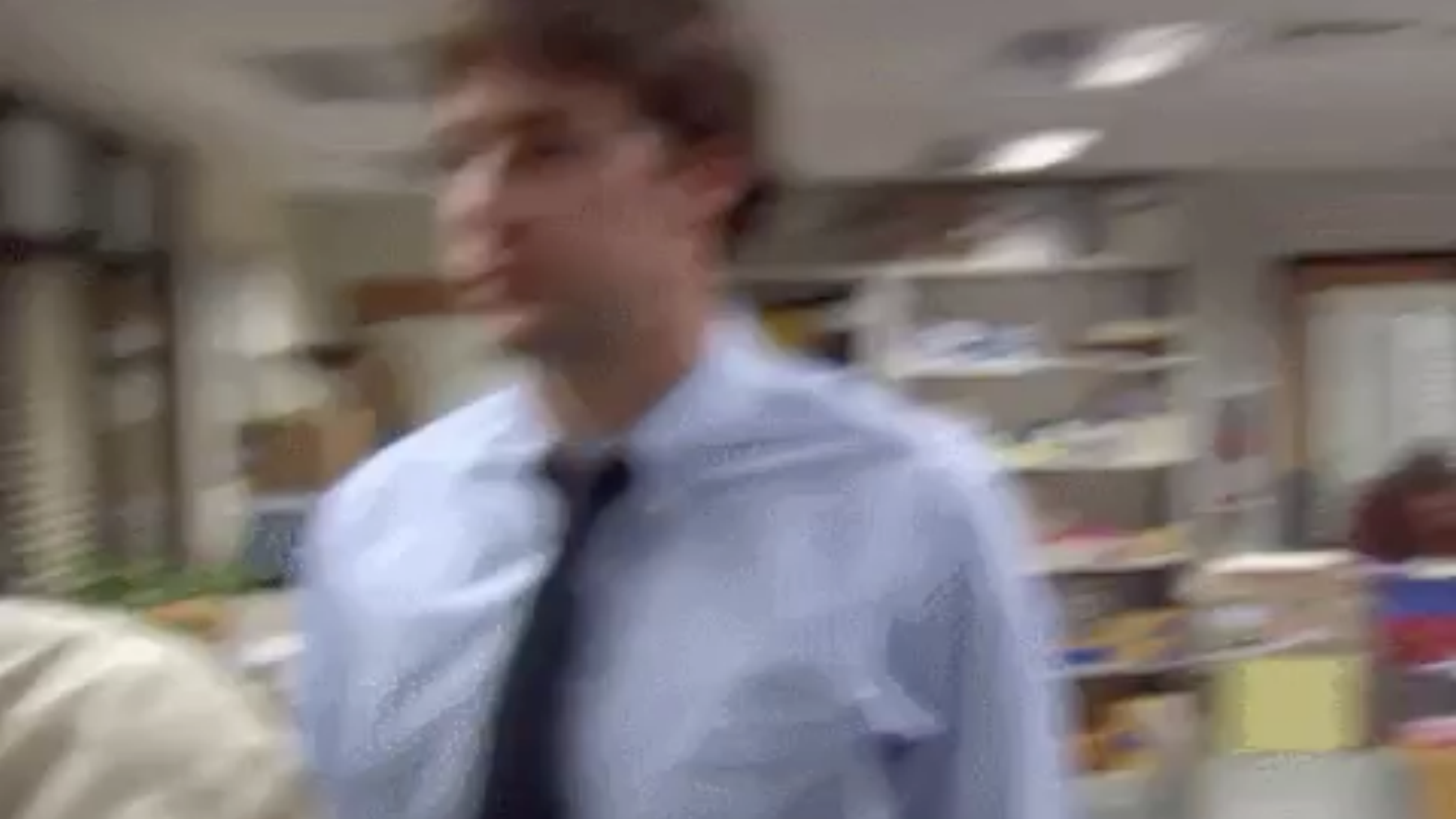
WILL NOT COMPILE!

```rust
struct Food {
    name: String
}

// Eat food - takes ownership of the variable.
fn eat_food(meal: Food) {
    println!("ate food: {}", meal.name);
}


// Inspect food - borrows the variable.
fn inspect_food(meal: &Food) {
    println!("inspected food: {}", meal.name);
    eat_food(*pizza); [!!] Cannot move out of borrowed context.
}

fn main() {
    let pizza = Food { name: "Pizza".into() };
    inspect_food(&pizza);
    eat_food(pizza);
}
```

WILL NOT COMPILE!

Got all that?





Further Reading

- https://doc.rust-lang.org/book/title-page.html - The Book
- https://www.rust-lang.org/ - Language Website
- https://crates.io/ - Crate registry
- https://rustup.rs/ - Installer

Slides Available Online:

http://richinfante.com/talks/