Diving head-first into Rustland

A primer to avoid getting lost

Rust Meetup Linz #32

AUGUST 23, 2023

unknowntrojan

19 | he/him |DE

GitHub
/unknowntrojan

Discord
unknowntrojan

GitHub
/unknowntrojan

About Me

- Employed @ AVOLENS GmbH developing a CSPM with Rust.
- Have been using C++ for a long time.
- Background in Reverse Engineering and Gamehacking.
- Upon a friend’s recommendation, I tried Rust roughly 2 years ago.

What is the goal of the talk?

My goal is to help C++ programmers that are looking to try Rust aren’t overwhelmed with unfamiliar constructs and syntax.

I want to make the talk that I would have loved to see when I was trying Rust for the first time.

What will this talk go over?

We will touch on...

- The Style of Rust code
- Expressions and FP
- Memory Management (Ownership & Borrowing)
- The Package management (Cargo)
- Pattern Matching
- Enums/Tagged Unions
- Error handling
- Composition as an alternative to inheritance
- Generics

https://doc.rust-lang.org/book
Coming into Rust, I was not expecting the documentation to be *this good*.

https://cheats.rs/
is a great resource for quick, concise information whenever you need it.

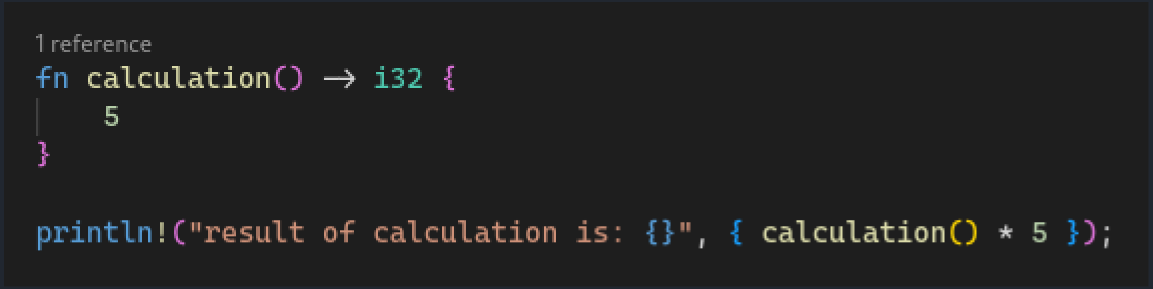
Rust’s style

- Compared to a lot of other languages, Rust’s style is pretty consistent.
- Style is enforced via rustfmt.
- There is comparably little freedom in Rust’s code style.
- This means reading someone else’s Code roughly feels the same as reading your own.

Expressions

- Rust makes heavy use of expressions. A lot of constructs that are statements in other languages are expressions in Rust. For example **blocks, if-expressions, and match-expressions.**
- In C, one might use a ternary operator to choose a value based on a condition. In Rust, you can simply inline an **if** or **match** expression.
- Switch cases are often used in C to do conditional branching with many possible outcomes. In Rust, **match-expressions** can be used to pattern match on a value and conditionally evaluate the respective expression.
- A block of code is also an expression, allowing you to assign a value to the value a block of code evaluates to.

A block of code as an expression



```
1 reference
fn calculation() -> i32 {
    5
}

println!("result of calculation is: {}", { calculation() * 5 });
```

FP

- Rust borrows a lot of concepts from functional programming languages such as **OCaml**.
- It makes heavy use of iterators, allowing easy and efficient processing of collections of data.
- Generally, you will be using a lot of **.map()**, **.iter()**, **.find()** and **.fold()/.reduce()**.
- Those functions call a closure for each element of a collection, and act a certain way accordingly.
- Sadly I do not have the time to cover this topic fully, but it is a very important aspect of rust.

Memory Management

- Rust uses a model called “Ownership and Borrowing”.
- This is one of the biggest talking points when it comes to Rust, as this model prevents common bugs such as UAFs, Double Frees, and Data Races.
- Unlike other popular languages like C++ and Java, Memory Management is not achieved through “manual” management or garbage collection.
- It uses the Rust compiler to enforce memory safety at compile time.
- When starting out, this can feel like a constant fight against the compiler, but after a while this fades.

Ownership and Borrowing

- Every variable has an owner.
- Once the owner is out of scope, the variable is “dropped”. The **Drop** trait allows the variable to clean up after itself.
- Variables are move-by-default.
- References to a variable can only be given out in line with Rust’s borrowing rules.

Borrowing rules

- There can be only one mutable reference to a variable at a time, and no immutable references while the mutable reference is active.
- If the variable is not mutably borrowed, it can be immutably borrowed as often as desired.
- This is achieved using lifetimes.

Lifetimes

- Every variable has a lifetime determined at compile time.
- A struct can hold a reference to another variable only if it is certain that the variable will live as long or longer than the struct will.

Package management

- Rust’s package manager is called Cargo.
- It manages downloading, compiling and linking dependencies automatically.
- Rust has a package repository called crates.io
- Adding a dependency from a git repository, crates.io or a path is very easy.
- It is consistently used on almost every single Rust project, so it’s easy to integrate.

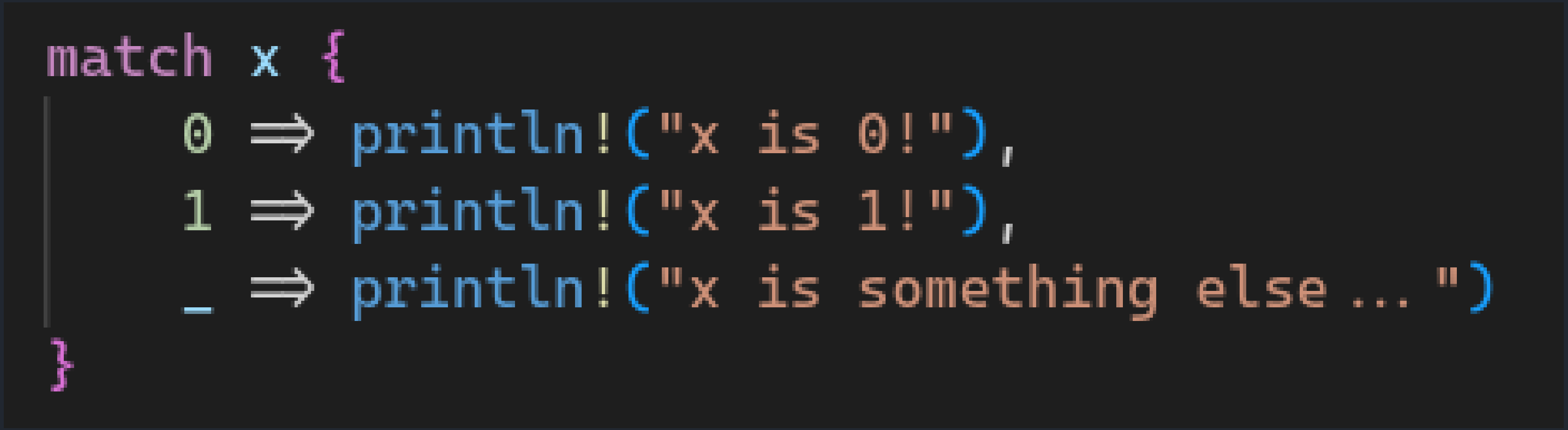
Crates

- A rust project is called a “crate”.
- A crate can be a library to be used by other rust code, or a binary crate with an entry point.
- Crates can expose feature flags the end user can select to enable or disable certain features of the crate.
- A crate is versioned using SemVer.
- When adding a crate as a dependency, a SemVer constraint is specified to tell Cargo which versions are fine to use.

Pattern Matching

- Pattern matching allows you to match patterns on values, for example matching a literal, matching an enum variant, and more.
- In C, one might use a switch statement to selectively execute code based on a variable’s contents.
- In Rust, this syntax is improved upon with the **match-expression** and allows for writing more readable code.

A match expression

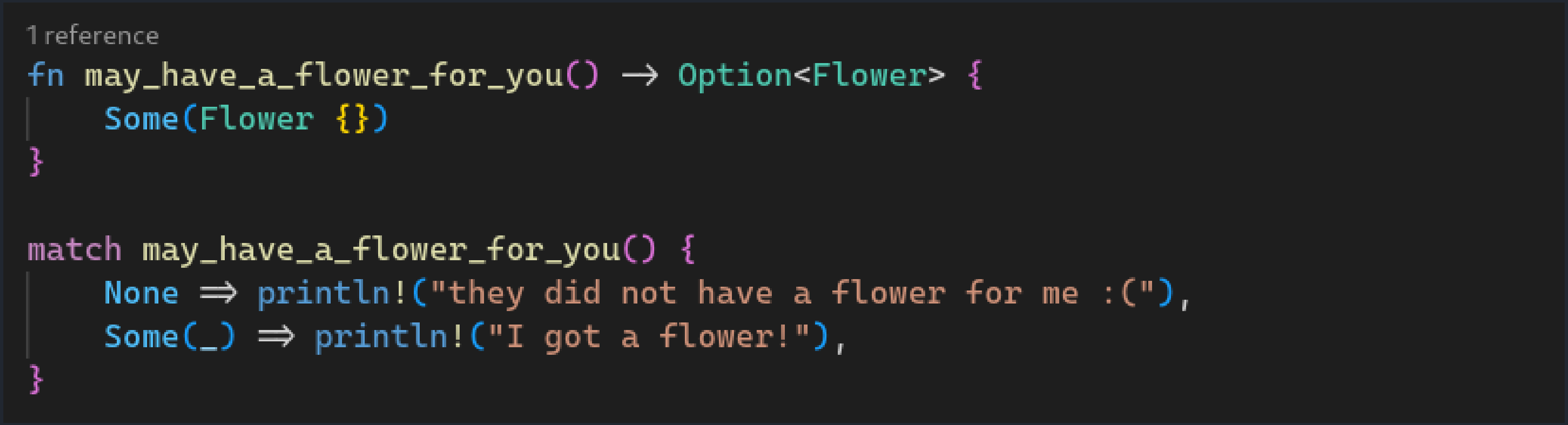


```
match x {
    0 => println!("x is 0!"),
    1 => println!("x is 1!"),
    _ => println!("x is something else...")
}
```

Enums

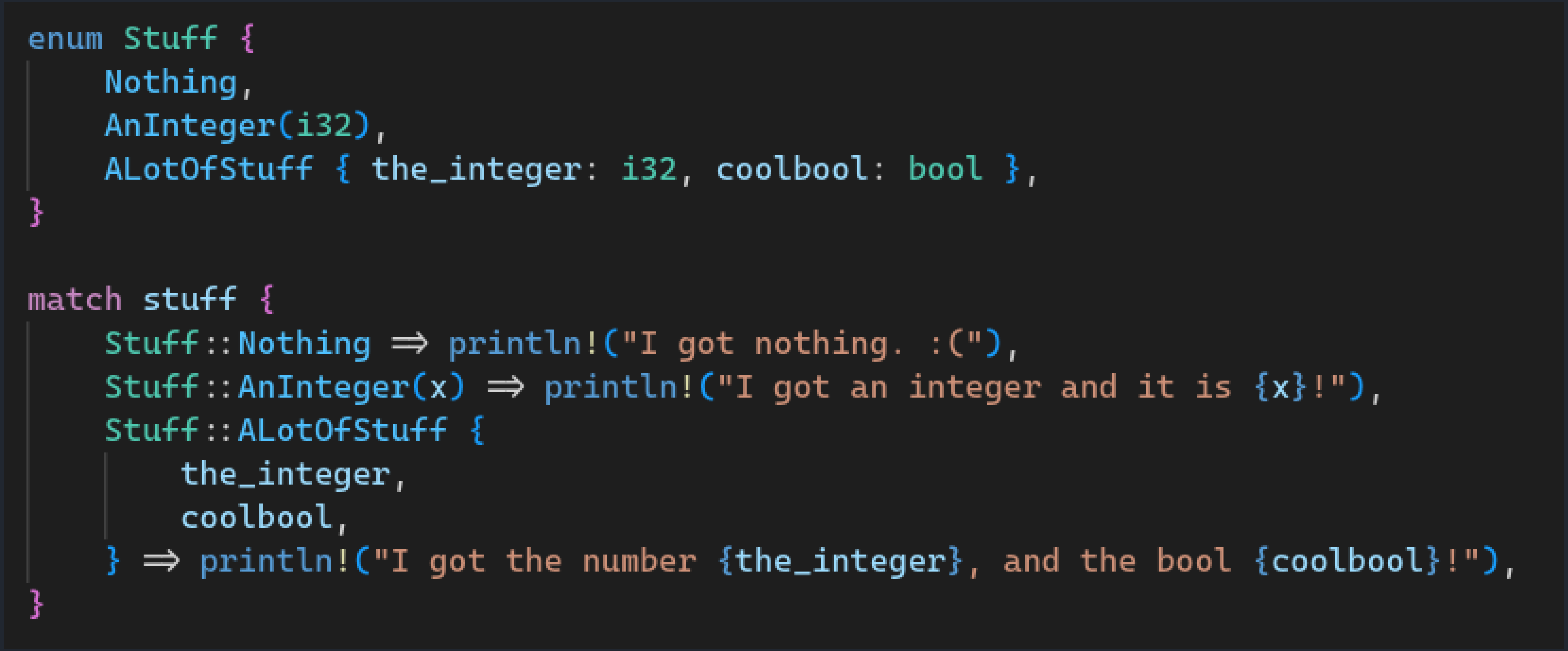
- Rust enum variants can not only be numeric, but also contain an arbitrary type the user can use after matching for it.
- For example, Rust has two very common enums: **Option<T>** and **Result<T, E>**.
- **Option<T>** can be one of two variants: **None** and **Some(T)**. No need for **null**! If a function can conditionally return a value, **Option<T>** is used to let the user easily match on it and decide what to do.
- **Result<T, E>** can be one of two variants: **Ok(T)** and **Err(E)**. It is the enum used for error handling in Rust. If a function can fail to provide a value, **Result<T, E>** is used to provide either the desired result or a type describing the error.

Option<T>



```
1 reference
fn may_have_a_flower_for_you() -> Option<Flower> {
    Some(Flower {})
}

match may_have_a_flower_for_you() {
    None => println!("they did not have a flower for me :("),
    Some(_) => println!("I got a flower!"),
}
```

Matching on an enum

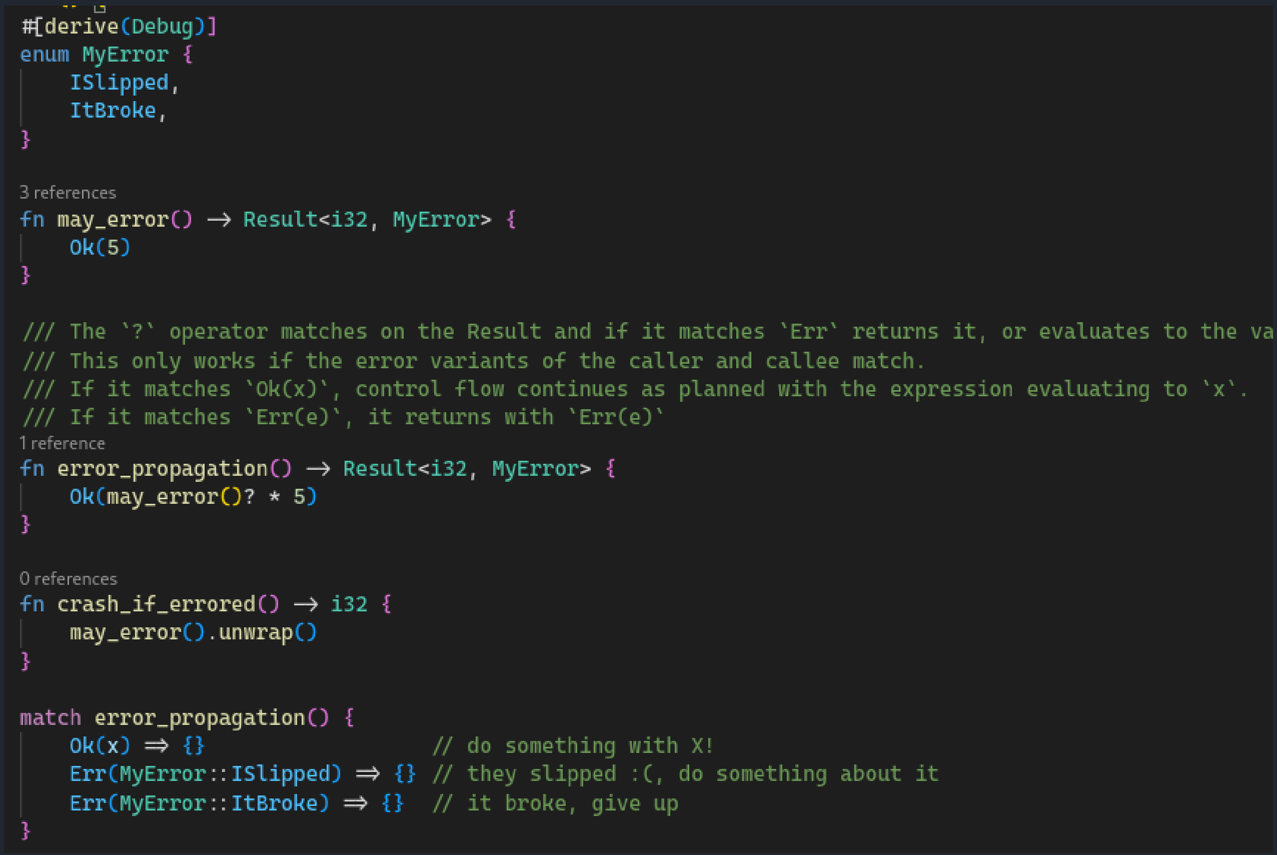


```rust
enum Stuff {
    Nothing,
    AnInteger(i32),
    ALotOfStuff { the_integer: i32, coolbool: bool },
}

match stuff {
    Stuff::Nothing => println!("I got nothing. :("),
    Stuff::AnInteger(x) => println!("I got an integer and it is {x}!"),
    Stuff::ALotOfStuff {
        the_integer,
        coolbool,
    } => println!("I got the number {the_integer}, and the bool {coolbool}!"),
}
```

Error Handling

- Other languages usually use **Exceptions** to signal an error state.
- Rust uses **Result<T, E>** for error handling.
- Usually **E** is an enum or struct containing error information.
- The end user of the **Result<T, E>** can match on it and best decide what to do.
- There are crates aiming to make error handling even easier such as **thiserror**, which allows you to specify error messages on variants of an error enum that can be formatted with the values the enum contains, or **anyhow**, which aims to provide a generic error type.

Error Handling Example Code



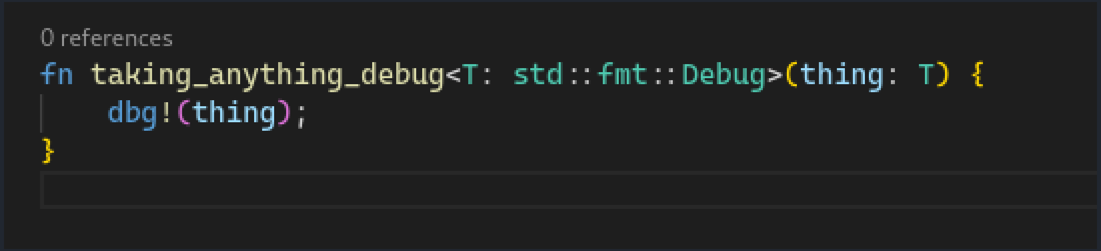
```
#[derive(Debug)]
enum MyError {
    ISlipped,
    ItBroke,
}

3 references
fn may_error() -> Result<i32, MyError> {
    Ok(5)
}

/// The `?` operator matches on the Result and if it matches `Err` returns it, or evaluates to the va
/// This only works if the error variants of the caller and callee match.
/// If it matches `Ok(x)`, control flow continues as planned with the expression evaluating to `x`.
/// If it matches `Err(e)`, it returns with `Err(e)`
1 reference
fn error_propagation() -> Result<i32, MyError> {
    Ok(may_error()? * 5)
}

0 references
fn crash_if_errored() -> i32 {
    may_error().unwrap()
}

match error_propagation() {
    Ok(x) => {}                     // do something with X!
    Err(MyError::ISlipped) => {} // they slipped :(, do something about it
    Err(MyError::ItBroke) => {}  // it broke, give up
}
```

Composition over Inheritance

- Other popular languages use Inheritance for OOP. Rust uses an approach called **composition**.
- A **trait** is a definition of an object's behavior. It could be compared to a C++ interface class.
- **traits** can be implemented for a type with an **impl** block.
- An object can have as many **traits** implemented for it as required.
- If you would like to take any object implementing a certain **trait** or **traits**, you can use a **Box<dyn TraitOne + TraitTwo>**. This is a fat pointer with vtable information, and can be used to dynamically work with objects matching this **trait bound**.

Generics

- **Generics** are a way to have functions, structs and more take any type matching certain **bounds**.
- Code is generated at compile-time accordingly, generating multiple versions of the construct for each real type used.

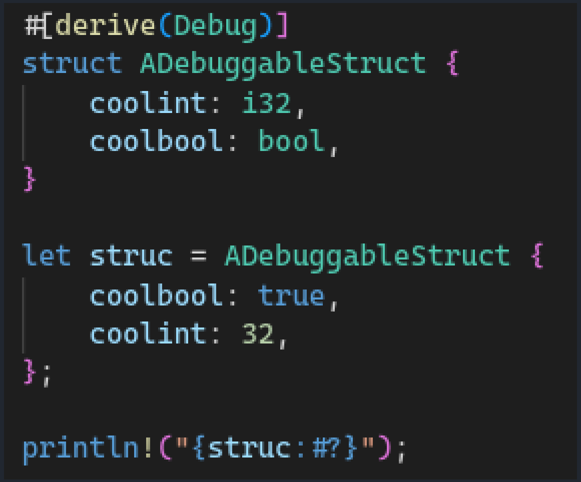


```
0 references
fn taking_anything_debug<T: std::fmt::Debug>(thing: T) {
    dbg!(thing);
}
```

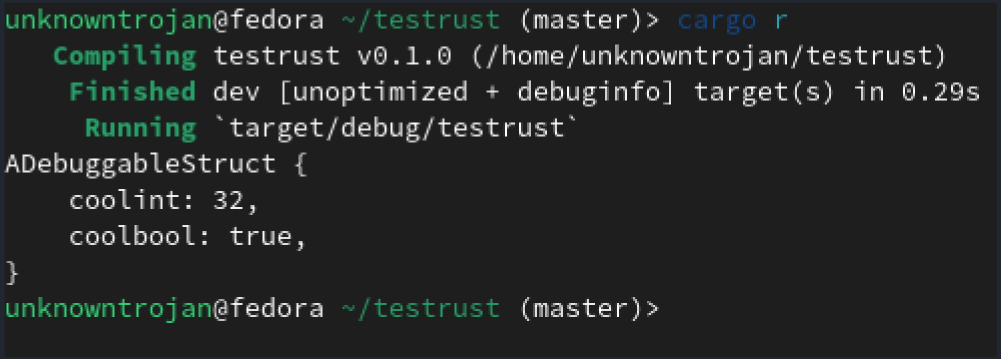
Miscellaneous Things

- As any body of code is an **expression**, the **tail expression** of the block is what it evaluates to. By extension, functions do not need a **return** statement, but can simply place their return value as the **tail expression**.

Debug Derives



```rust
#[derive(Debug)]
struct ADebuggableStruct {
    coolint: i32,
    coolbool: bool,
}

let struc = ADebuggableStruct {
    coolbool: true,
    coolint: 32,
};

println!("{struc:#?}");
```



```
unknowntrojan@fedora ~/testrust (master)> cargo r
   Compiling testrust v0.1.0 (/home/unknowntrojan/testrust)
    Finished dev [unoptimized + debuginfo] target(s) in 0.29s
     Running `target/debug/testrust`
ADebuggableStruct {
    coolint: 32,
    coolbool: true,
}
unknowntrojan@fedora ~/testrust (master)>
```

Thanks for attending!

Feel free to ask any Questions

Rust Meetup Linz #32

unknowntrojan

19 | he/him |DE

AUGUST 23, 2023

GitHub
/unknowntrojan

Discord
unknowntrojan

GitHub
/unknowntrojan