

A Crash Course in Rust

Let's learn [Rust](#) together.

What is Rust?

- Strongly typed systems programming language
- No garbage collector
- Memory safety is checked at compile time
 - Prevents undefined behavior
 - Use after free, data races, etc
- Async/Await for high performance apps
 - Core IPC message broker
- Package management

Installation

- [Rustup](#)

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh  
rustup default stable # Install and use the latest stable rust toolchain
```

Tooling

- Visual Studio Code
- `rust-analyzer` extension
 - Language server for rust
 - Provides IDE-like features
 - Intellisense
 - Goto Definition
 - Refactoring support
 - Inlay type hints

Creating a Project

```
cargo new learn-rust  
cd learn-rust  
cargo run -r
```

Data Structures



```
// main.rs
struct Dog;

fn main() {
    let _dog = Dog {};

    println!("Hello, world!");
}
```

Mutability

```
struct Dog {  
    age: u8,  
}  
  
impl Dog {  
    pub fn celebrate_birthday(&mut self) {  
        self.age = self.age + 1;  
        println!("Fluffy is {} years old!", self.age);  
    }  
}  
  
fn main() {  
    let mut dog = Dog { age: 8 };  
    dog.celebrate_birthday();  
}
```

Constructors

```
struct Dog { age: u8 }  
impl Dog {  
    pub fn new(age: u8) -> Self {  
        Self { age }  
    }  
    // ...  
}  
fn main() {  
    let mut dog = Dog::new(8);  
    dog.celebrate_birthday();  
}
```


Enumerations

```
enum BoneKind {  
    Bacon,  
    PeanutButter,  
    Turkey,  
}
```

Option

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

Optional Fields

```
struct Dog {  
    age: u8,  
    pub bone: Option<Bone>,  
}  
  
impl Dog {  
    pub fn new(age: u8) -> Self {  
        Self { age, bone: None }  
    }  
  
    // ...
```

Wait a second...

- What if the dog already has a bone?
- What if the dog doesn't like the flavor?
- What if the dog refuses to take the bone?

Borrowing, Ownership, and Lifetimes

- Data can only have one owner
- `T` is a type
- `&T` is an immutable ref to `T`, referred to as borrowing
 - N number of immutable refs allowed unless a mutable ref exists
- `&mut T` is a mutable ref to `T`
 - Limit of only one at a time
- `&'static mut T` is a mutable ref to `T` with a `'static` lifetime
 - Guaranteed to live for the duration of the program

Results and Errors

How do we represent errors in Rust?

- Scenario 1
 - Take a bone from a dog that is without one
 - `None` represents the absence of the bone
- Scenario 2
 - Give one and then another bone to a dog
 - We can return `Err` to represent an error

Result

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

Traits

- What are traits?
 - Similar to interfaces
 - Only specify behavior and not data
 - Not inheritance
 - Allows for dynamic dispatch
 - i.e `Vec<Box<dyn Animal>>`
- Built-In Rust traits
 - Default
 - Display

Custom Errors

```
struct AnimalError { details: String }  
impl AnimalError {  
    fn new(msg: &str) -> Self {  
        Self { details: msg.to_string() }  
    }  
}  
impl std::error::Error for AnimalError {  
    fn description(&self) -> &str { &self.details }  
}  
impl std::fmt::Display for AnimalError {  
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {  
        write!(f, "{}", self.details)  
    }  
}
```

Type Aliases

Declare function aliases to abbreviate types

```
pub type Result<T, E = Box<dyn std::error::Error>> = std::result::Result<T, E>;
```

Smart Pointers in Rust

- What is a `Box`?
 - Just a smart pointer 🧠➡️😎
 - Used for safe heap allocations

Writing Fallible Methods

```
pub fn receive_bone(&mut self, bone: Bone) -> Result<()> {
    match self.bone.as_ref() {
        Some(bone) => {
            return Err(Box::new(AnimalError::new(&format!(
                "Dog already has a bone! ({:?})",
                bone
            ))))
        }
        None => {
            println!("Fluffy grabbed the {:?} bone!", bone.kind);
            self.bone = Some(bone);
        }
    };
    Ok(())
}
```

More Error Conditions

```
pub fn speak(&self) -> Result<()> {  
    match self.bone.as_ref() {  
        Some(bone) => Err(Box::new(AnimalError::new(&format!(  
            "Dog can't speak because of the {:?} bone!",  
            bone  
        )))),  
        None => Ok(println!("Woof!")),  
    }  
}
```

Happy Birthday, Fluffy! 🍰🐕

```
fn main() -> Result<()> {  
    let mut dog = Dog::new(8);  
    dog.celebrate_birthday();  
    dog.speak()?; // Now we can invoke dog.speak()  
  
    let bone = Bone::new(BoneKind::BaconFlavored);  
    dog.receive_bone(bone)?;  
  
    Ok(())  
}
```

- **?** operator propagates error to the caller

Debug Output

```
#[derive(Debug)]  
struct AnimalError {  
    details: String,  
}
```

Final Program

Available on the [rust playground](#) 

Next Steps

- The [official rust website](#)
- The Rust bookshelf
 - Run `rustup doc`
- [Awesome Rust Learning](#)
 - Large list of learning resources