A Crash Course in Rust



Let's learn Rust together.

wnat is kust:

- Systems programming language
- Strongly typed
- No garbage collector
- Immutable by default
- Memory safety is checked at compile time
 - Prevents undefined behavior
 - Use after free (dereferencing a null pointer)
 - Data races
- 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 toolchai
```

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!");
}
```

Birthdays



```
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 }
    pub fn celebrate_birthday(&mut self) {
        self.age = self.age + 1;
        println!("Wiggly butt is {} wags old!", 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 }
fn main() {
```

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?

```
impl Dog {
    pub fn new(age: u8) -> Self {
        Self { age, bone: None }
    pub fn celebrate_birthday(&mut self) {
        self.age = self.age + 1;
        println!("Wiggly butt is {} wags old!", self.age);
struct Bone {
    kind: BoneKind,
impl Bone {
    pub fn new(kind: BoneKind) -> Self {
        Self { kind }
enum BoneKind {
```

- How do we represent errors in Rust?
- Scenario 1
 - We take a bone but the dog does not have one.
 What should we get back?
 - We can return a None variant of Option to represent the absence of the bone.
- Scenario 2
 - We give a dog a bone, but it already has one?
 What should we get back?
 - We can return the Err variant of a Result type, to represent an error.
- Result can represent fallible operations

Traits

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

```
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
       writal(f "S" salf datails)
```

Type Aliases

Declare function aliases to abbreviate ubiquitous types.

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

Box?

Smart Pointers in Rust

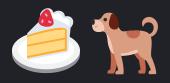
- What is a Box ?
 - Just a smart pointer
 - Used for safe heap allocations

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

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(())
```

- What is ??
 - Sugar for methods returning Result
 - Propagates error to the caller if they fail

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