Jeff Walker “Code Ranger"
Jeff@WalkerCodeRanger.com
@WalkerCodeRangr
WalkerCodeRander.com

- Ownership
    - Move Semantics
        - let p = Point { }
        - let q = p
        - p is NOW blank - it’s value moved to q
    - Copy types
        - basic types do not have move semantics - structs have move semantics
    - Limitations of ownership
        - No return statements in a function - the last statement is the return value. Function declaration is like swift
    - Borrowing reference
        - fn distance_from_origin(p: &Point) -> f64 {}
            - & is borrow
            - in C/C++ land, the & is a pointer to an address and is the same in Rust too
    - Immutability is the default
        - let p = Point {x: 2, y: 4}
        - p.x = 5 // fails
        - To mutate: let mut p = Point { x: 2, y: 4 }
        - fn doube(p: &mut Point) { } <— function takes a mutable reference to a point
    - Rules for immutablility
        - Example:
            - let mut x = 5;
            - let y = &mut x; <— mutable borrow
            - *y += 1;
            - println!(“{}”, x); <— immutable borrow
            - Get error from compiler: cannot borrow x as immutable because it is also borrowed as mutable

        - Rules
            - Any given value, only one owner at a time
            - owner is only one that can access data (unless they lend it out)
            - ownership can be transferred (move semantics)
            - any borrow must last for a scope no greater than that of the owner
            - have one or the other of these two kinds of borrows, but not both at the same time
                - one or more references (&T) to a resource
                - exactly one mutable reference (&mut T)
            - owner limited while borrowed
        - To fix above code
            - let mut x = 5;
            - {
                - let y = &mut x; <— mutable borrow
                - *y += 1;
            - }
            - println!("{}", x); <— immutable borrow
            - // Added braces to add scope and now mutable borrow ends BEFORE println()
    - Why?
        - Safe data structures
        - Iterator invalidation
            - modifying collection while iteratoring - this in Rust is avoided since it's declared static at compile time. Looping and trying to modify you will get a compile error: cannot borrow 'v' as mutable because it is also borrowed as immutable.
            - This is checked at compile time and there's NO run time overhead for this (unlike Java or C# which checks before looping)
        - Use after free
            - let y: &i32;
            - {
                - let x = 5;
                - y = &x;
            - }
            - println!("{}", y);
            - // error: 'x' does not live long enough
        - Resource management

            - Garbage collection
                - In other languages, it doesn’t work well enough for other resources for things like files.
                - Other languages implement garbage collection in several different ways - this is unified in Rust
            - files
            - networking
            - instead of disposable pattern, use Resource Acquisition is Initialization (RAII) pattern
            - Implement ‘Drop’ trait
            - Don’t need something like C# ‘using’ statement
    - Lifetimes
        - string example given
        - generics: fn skip_prefix<‘a, ‘b> (line: &’a str, prefix: &’b str) -> &’a str {
        - The ‘ declares there is a life time defined
        - Lifetimes in structs
            - properties in struct can have a lifetime
            - any reference in a struct and you MUST be explicit about its lifetime
        - Most of the time the compiler figures out the lifetime
            - Lifetime Elision
                - each elided lifetime in a function’s arguments becomes a distinct lifetime parameter
                - if there is exactly one input lifetime, elided or not, that lifetime is assigned to all elided lifetimes in the return values of that function
                - if there are multiple input lifetimes, but one of them is &self or &mut self, the lifetime of self is assigned to all elided output lifetimes
            - Elision examples
    - Summary
        - Use Rust for low level programming (instead of C/C++)
        - Borrow Checker provides safe memory and resource management
        - The rules of the borrow checker make sense in other languages

            - Rust is pretty much the only language that has this memory model
                - Dyon (game scripting language)
                    - Dynamically typed scripting language, designed for game engines and interactive applications
                    - The scripting language uses Rust like borrowing rules
                - Adamant (speaker’s own programming language)

- Questions / Notes
    - Rust is using LLVM so it kicks out an executable.
    - Rust MAY be available for Arduino.
    - C code can call into Rust (C++ can do it too, but it’s way more difficult due to namespace issues).
- Resources
    - rust-lang.org
    - doc.rust-lang.org/book
    - rustbyexample.com
    - www.piston.rs/dyon-tutorial
    - adamant-lang.org