Slide 1:

Hi, I'm Sherry, and this video was created by David Atkinson, Rami AlQunaibit, and me, about Rust's memory model.

Slide 2:

Have you ever had memory errors? Segmentation faults? Memory leaks? Dereferenced a null pointer?

These are big problems, and while languages have made a lot of progress on this front since Algol, there's still room for improvement. We want to talk about a language, Rust, that thinks it has something to offer here.

Slide 3:

Rust is a systems programming language sponsored by Mozilla Research, which describes it as a "safe, concurrent, practical language," supporting functional and imperative-procedural paradigms. Rust is syntactically similar to C++, but its designers intend it to provide better memory safety while still maintaining performance.

Rust won first place for "most loved programming language" in the Stack Overflow Developer Survey three years in a row 2016, 2017 and 2018.

As you can see in the code sample, Rust has C-like syntax, along with some features that will be familiar to Scala programmers, such as type annotations, anonymous and higher-order functions, and compile time macros.

Rust is a big language, with many features, but today, we're going to focus specifically on Rust's approach to memory management, which is arguably its biggest draw.

Slide 4:

Before we dive in to Rust's memory model, let's take a look at Javascripty's. Javascripty is a simple language, but it does have a pretty rudimentary system of memory addressing, dereferencing, and mutation. As we'll see, this basic framework is similar to Rust's approach. However, Rust is much more restrictive.

Javascripty also distinguishes between parameters that are called by value, and those that are called by reference. Again, Rust's approach is similar, but with added safety rules.

One significant difference is that, with a cast, we can assign a Null value to Javascripty objects. In the main Rust language though, this is impossible. Instead, Rust encourages an approach that's similar to Scala's, in which the possibility of not having a value is expressed directly

through the type system.

Rust also allows a second language hiding inside of it that does not enforce these memory safety guarantees: unsafe Rust. This works just like regular Rust, but gives you extra superpowers.

Unsafe Rust exists because, by nature, static analysis is conservative. When the compiler is trying to determine if code upholds the guarantees or not, it's better for it to reject some programs that are valid than accept some programs that are invalid. That inevitably means there are some times when your code might be okay, but Rust thinks it's not! In these cases, you can use unsafe code to tell the compiler, "trust me, I know what I'm doing." The downside is that you're on your own; if you get unsafe code wrong, problems due to memory unsafety, like null pointer dereferencing, can occur. This allows:

1- Dereference a raw pointer
2- Call an unsafe function or method
3- Access or modify a mutable static variable
4- Implement an unsafe trait

Slide 5:

Rust's memory model is driven by different requirements than Javascripty's. Rust would like to have both: the safety and ease of garbarge collected languages like Java and Scala, AND the control and predictability of the manual memory management provided by languages like C and C++. The way it does so is the *ownership model*.

With the ownership model, each memory value has a single owner, usually the variable it was initially assigned to. For example, in the figure on the right, on line 2, "v" is the owner of a heap--allocated vector.

In Rust, when one variable is assigned to another, the ownership of the relevant memory value is "moved." Moving is not the same as copying, in that after a move, Accessing the first variable now results in an error! You can see this in the figure on the left. First, we initialize "a" with a string value. But, when we assign "a" to "b" on line 2, "a" becomes uninitialized and 'b' is now completely responsible for the string initially in 'a'. Importantly, since this is the default behavior, moves are cheap--internally, nothing on the heap actually changes, just the internal pointers.

This may seem inconvenient for cases where you simply want to set one int equal to another. Luckily, Rust does have copy traits that can be used on primitive variables that allows you to do so.

In the next few slides we'll talk about some other ways Rust makes it easier to actually write useful programs.

Slide 6:

In cases where you want to use a variable from several places, moving it to each of those places isn't a very good method. Instead, Rust also has a notion of references, like C, C++, Javascripty, and so on. Unlike those languages, Rust has two types of references: shared, and mutable. If you're familiar with the classic Readers/Writers concurrency model, this will be familiar. Rust let's you have multiple shared references, but only if there are no mutable references to the object. You can use a mutable reference to modify an object, but not a shared reference.

So, in the screenshots, the left code segment will produce an error. We're trying to modify the string we've passed in to "concat_bar". But we're using a shared reference. Changing everything to be mutable will fix things.

Slide 7:

In a nutshell, lifetimes are the way the compiler ensures that every reference is a good one. In Rust, every variable has a lifetime. To use a reference to a variable, that reference has to be "alive." One of the technical advances offered by Rust is compile time verification of references. In the case of lifetimes, Rust is actually removing the necessity of a garbage collector entirely by pushing additional rules onto the programmer. If your program compiles - it's almost certainly memory safe. These rules force us to think about the memory safety of the way we are programming upfront rather than in bug fixes months down the track. Yes, it's potentially more work, but it will undoubtedly save us headaches in the future!

This is a little abstract, so here's example to illustrate how it all works in practice.

Slide 8:

This code will actually not compile, as I'll explain in a minute.

The first thing to note is that scope is the key to understanding lifetimes. Like Scala, Rust is lexically scoped. So from this code segment, we can determine the values that each variable has, and when it has them.

So what's happening here? First, we declare a variable "r". Then, in another block, we declare a variable "n", before setting "r" equal to a shared reference to "n". Finally, we check if the dereferenced "r" equals the original value of "n", 45.

The problem here is that "r" is in scope from lines 2-7, and "n" is in scope from lines 4-5. So the lifetime of the reference "&n" both shouldn't exceed the bounds of lines 4-5 AND needs to extend to line 7, where it's used.

Since there's no way to satisfy those requirements, Rust gives us a helpful error message:

Slide 9:

“‘n’ does not live long enough”

It’s worth noting here that this is happening at compile time. In Javascripty for example, the equivalent program would create a totally valid parse tree. To realize that our reference actually points to nothing useful, we’d have to wait until runtime. This makes sense for Javascripty, since its interpreted, and because the kind of static analyses that Rust uses to determine lifetimes is way too complicated for an undergraduate class. But in Rust, it’s nice to know that a whole class of errors just isn’t possible, as long as your code compiles.

“It often feels almost like Test Driven Development in that I can be certain the code is at least technically correct if it will compile. When it doesn't compile, I get a usually-helpful error message that helps me figure out what went wrong. This is nice compared to writing Ruby or JavaScript because I often don't know that something will break until runtime. I would much rather know about these kinds of small mistakes before they become a problem!” -- Software Engineer Lee Baillie.

Slide 10:

We’ve only talked about the very basics here. Rust has a ton of other features just related to memory management, that are all worth checking out.

We’ve mentioned some of these earlier in the video.

Not all assignments in Rust actually transfer ownership, for example. If you’re interested in that, the Copy trait is worth reading about.

Also, some parts of Rust are sort of garbage collected … ish. For example, the Arc type stands for “automatic reference counting”. It lets you create the same kind of program structure as you might get in C, with big webs of references, all managed by Rust.

Finally, we’ve been talking about safe Rust code. If you’re an advanced user, Rust lets you do all the unsafe things that you can do in C (and Javascripty). Searching for “the Rustonomicon” will tell you how.