Rust is a systems programming language with great ergonomics and sweet high-level features, suitable for backend development, web applications, mobile, embedded, graphical user interfaces, and even games. Rust is an imperative language at its core, but borrows a lot of goodness from functional programming. Rust also provides top-notch performance, type safety, and memory safety. Rust supports metaprogramming through attributes and macros. And incredibly, Rust does all of this without a garbage collector.

Graydon Hoare created Rust as a personal project in 2006 while working at Mozilla. Mozilla officially adopted the project in 2009, but the first stable release wasn't until May 2015. Since its release, grassroots enthusiasm has brought it to Amazon, Meta, Alphabet, Microsoft, and tons of others. Linus even welcomed it into the kernel in December 2022! But the adoption of programming languages is slow, yo, so it's only recently that Rust started spreading like wildfire.

Now that wildfire has spread to Xebia Functional. For the next 30 minutes, that wildfire looks like this guy, Todd Smith, the new Rust Solution Architect, laying down some Rust basics, so thanks for joining me today. Without further ado, let's see what Rust brings to table.

There are a lot of things that Rust does the *same* as other languages, but I want to focus on what Rust does *differently*.

It's typical to spend a lot of time *thinking* about data ownership when using a systems programming language, but usually the language doesn't have any useful features for *actually managing* data ownership.

Back in the bad old days of C, if you wanted memory safety, you could bust out third-party tools like Splint and Valgrind and then instrument your code with cheesy annotations and special wrapper calls. But you only got as much support as you paid for with your own effort and diligence; you got nada from the compiler. And you still expected to spend hours, days, maybe weeks, in front of transcripts and debuggers.

They say that possession is nine tenths of the law, but in Rust, it's more like the whole of the law. Rust bakes in all the support that C desperately needs but never had. Ownership is built directly into the type system, so type safety *is* memory safety.

So, what *is* ownership, exactly? Well, it's basically the obligation to destroy something when it goes out of scope. I know that was so 90 seconds ago, but remember how I said that Rust doesn't need a garbage collector? Instead, the Rust compiler has a *borrow checker*, a built-in static analyzer that tracks ownership of values. The borrow checker ensures that Rust always knows when to destroy an object. In the vast majority of cases, the borrow checker can *statically* determine where to insert the destructor call; in the few cases where it can't, the borrow checker can defer ownership tracking until runtime. There's no escaping the borrow checker, and that's a good thing!

Ownership always accompanies introduction of a value. In other words, when code mentions a literal or instantiates a struct or enum, some variable, formal parameter, field, or temporary becomes the owner of the new value. And the owner is responsible for the value's eventual destruction, when the owner goes out of scope.

But the owner can also delegate that responsibility, by assigning the value to another variable or field or by giving ownership to another function or method. Whenever you see an punctuation-free type annotation on a formal parameter of a function or method signature, it means that the formal parameter assumes ownership of the passed value. After giving the value away, the owner becomes defunct — it hasn't technically gone out of scope, but the compiler will signal an error if you mention it again.

For simple values, like booleans and numbers, the compiler makes a copy, and the result is two owned values — one associated with the original binding, one with the new binding.

For more complex values, passing the value transfers ownership to the new binding. After the transfer, you can't use the original binding anymore — no takebacks!

Full disclosure: you *can* opt into copying for your own types, but that's out of scope right now; I need to leave stuff for future talks, capisce?

Anyway, taken together, these rules mean that every value has exactly one owner. And since the owner has the obligation to destroy the value, it means that each value will be destroyed at most one time. If you've spent a lot of time with C or C++, you've probably already drawn the conclusion: by enforcing linearity of ownership, Rust statically prevents the memory error called *double free*.

But you don't have to give away ownership of a value to grant access to it. An owner can *lend* a value out; or, reversing the viewpoint, another binding can *borrow* a value from its owner. Borrowing is different from ownership because it conveys capability to access, or even modify, a value but does not bestow the responsibility for destroying that value.

Naturally, there are some important rules governing borrowed values, otherwise we wouldn't need a borrow checker!

Firstly, there can be only one mutable borrower of some referenced value. So long as a mutable borrow exists, no other borrows can exist at all. And while the mutable borrow exists, even the owner cannot modify the underlying value. If you think of this in physical object terms, it makes perfect sense — how can you scribble in the margins of a book that you've lent out to a friend?

Secondly, so long as no mutable borrower exists, there can be many immutable borrowers of some referenced value. The physical analogy isn't straightforward here, because I usually can't lend the same book to each of my friends. Something to do with conservation of mass, I don't know. But fortunately, there's a good analogy from concurrent programming: read/write locks!

Lastly, references cannot outlive their owners. If they could, then they could become invalid by pointing to freed (and potentially reused) memory. But the borrow checker statically ensures that all borrows occur within the lexical scope of the owner. In other words, borrows have to go out of scope before or simultaneously with the owner. And just like that, Rust prohibits *dangling references* by construction.

But does it? Does it really? What happens if I do something sneaky, like this? Here, I've nested two scopes. Inside the inner scope is the hapless and doomed owner of the value 10, as well as the creatively named inner\_borrow of owner. But in the outer scope is the villainous outer\_borrow, which tries to borrow owner indirectly through the unsuspecting inner\_borrow. Since owner goes out of scope on line 8 and outer\_borrow survives until line 10, outer\_borrow should become a dangling reference after line 8.

Is Rust going to stand for that? Nah, not really. The borrow checker noticed that owner didn't live long enough to accommodate the outer\_borrow, so it forbade the assignment outright, even though it tried using inner\_borrow as a patsy.

But maybe we can get even more creative, by using an intermediate function call to disguise our perfidy. Here, we've introduced two owners, outer\_owner and inner\_owner, and initialized matching borrows, outer\_borrow and inner\_borrow. Nothing suspicious so far. assign\_to\_borrow looks innocent enough, too — it just does an assignment whose effect is visible in the caller. But the actual call of assign\_to\_borrow is super sketch — it mutably borrows inner\_borrow and outer\_borrow so that the callee can make outer\_borrow point to inner\_owner. Muahaha, dangling reference created! Suck it, Rust!

Wait, what's this? Rust won't compile assign\_to\_borrow! Curses, foiled again! But how did it know?

There's more to a borrow than its referent. There's also its *lifetime*. In other words, how long the borrow points to a live owner. In *any language*, a reference must not outlive its referent's owner, because that's how you get dangling references, yo. But in Rust, the compiler enables — nay, *forces* — you to get it right. The borrow checker statically tracks the lifetime of every borrow, and it does this by implicitly or explicitly attaching a lifetime through a built-in property. The property is expressed as a generic type parameter of the enclosing context; in this case, the function assign\_to\_borrow.

Now we can unpack the compiler's error message. The compiler forbade the redirection of b's content to a's content because it assumed that their lifetimes were unrelated. And in a vacuum, what else *could* it assume? Most assumptions would be wrong in most circumstances, so Rust makes the most general possible assumption, thereby forcing us to clarify our intentions.

Okay, let's make one last ditch effort to achieve villainy, because I don't know why. We're going to add two lifetime parameters to assign\_to\_borrow, one for each formal parameter. We'll name the lifetimes after the formal parameters themselves, out of convenience rather than syntactic necessity, and we're going to use a colon to say that 'a outlives 'b. That guarantees locally that we can perform the assignment.

You may already be able to guess why this won't work. And there it is, we're straight back to the original "problem". Now that we've told Rust the relationship between the lifetimes, it throws them right back in our face to defeat our insidious attempt to create a dangling reference. So … yeah. Rust really does prevent dangling references by construction. Pretty sweet, yeah?

But if the story ended here, then it would be an incomplete story. What we've seen is impressive, but it doesn't cover the gamut of memory access patterns. What about heap-resident values? What about shared ownership? What about cycles? More generally, what about situations where it's much harder to decide when a value should be destroyed? Well, Rust provides several smart pointer types that flesh out its memory safety story.

The simplest is Box, which simply designates a value that lives on the heap. By default, Rust allocates all values on the stack, but Box and other smart pointers maintain their referents on the heap. Box is usually the right choice for types whose instances vary in size. Box is generic over two type parameters: T, which represents the type on the heap; and A, the type of the allocator responsible for managing that heap. Usually you only care about T, but A is available for so-called "placement new" situations, à la C++. In the vast majority of cases where you don't care, you don't have to mention A at all, and Rust will sensibly default it to the same type as the global allocator. There's no special magic here — like C++ and TypeScript and unlike Java, Kotlin, Scala, C#, and others, Rust supports default bindings for generic parameters. A Box is the sole owner of its content, so the content lives exactly as long as the Box does. When the Box goes out of scope, it and its contents are both destroyed.

The semantics of single ownership is nice and clean, but what if you have shared ownership? The classical example is a graph structure, where some nodes are held by multiple edges. Ownership of the nodes conceptually belongs to the whole graph, but usually the graph is a network of related objects, not a single object where ownership can be centralized. A natural enough approach is to share ownership of a node among its incoming edges, but how do we achieve this?

Rc to the rescue. Rc stands for *reference counter*. Rc is really just a thin wrapper for a private kind of Box that places both the referent and the reference counter on the heap. This reference counter is incremented whenever the Rc is cloned, and decremented whenever the Rc goes out of scope. When the reference count goes to zero, the referent is destroyed. And because no Rc is outstanding, by definition, there are no dangling references to the defunct referent.

So, what about mutating the referent of an Rc? Well, you can't. The private box inside is the real single owner of the shared data, and each Rc behaves like an immutable borrow of the box. If it didn't, then you could trivially violate the borrow checker's rule that each value can have at most one live mutable reference, which can lead to memory unsafety even in a single-threaded program.

So, does that mean that we can use Rc with multiple threads? Not quite, but we can use its concurrent cousin, Arc, which stands for *atomic reference counter*. Arc leverages special compiler intrinsics to ensure memory coherency in the presence of concurrent access, so it entails a bit more cost than Rc. This is one of several situations where Rust offers you a choice of abstractions, enabling you to right-size your choice based on your actual use case.

Armed with an Arc, you can now share a value between multiple threads and still be confident that it will be destroyed exactly once, as early as possible, without leaving a dangling reference behind. You still can't mutate the shared value, but we're getting closer. There are alternatives to concurrent data access patterns, of course — right, functional programmers? — and Rust enables numerous strategies, but dang it, sometimes it's convenient to mutate shared data rather than, I don't know, pass copies between threads.

Enter Mutex, the canonical mutual exclusion device from imperative programming. You can use the lock and try\_lock methods to obtain a MutexGuard. Once you have a MutexGuard, it acts as an exclusive mutable reference to the protected value, so any code dynamically reachable from the lexical scope of the MutexGuard can access and mutate the protected value. When the MutexGuard goes out of scope, its destruction releases the exclusive hold, giving some other thread a turn to enter its own critical section. This is a textbook example of RAII — *Resource Acquisition Is Initialization*.

Now let's put it all together, literally. Start with our data, which might be of any type, so let's call it T. We need to ensure exclusive access in order to satisfy Rust's rules regarding mutation, so we wrap a Mutex around it to obtain Mutex<T>. And we want shared ownership, so we wrap an Arc around that to obtain an Arc<Mutex<T>>. We can clone the Arc to ratchet up the reference count, and use closures to transfer ownership of the copied smart pointer to another thread. When the last Arc goes out of scope, the Mutex and its protected value are both destroyed.

You may be wondering what keeps me from using an Rc with multiple threads, other than a craftsman's desire not to write bad code that breaks a product at runtime. Let's put the supervillain mustache back on for a moment. Surely, you can use an Rc instead of an Arc to break Rust at runtime, right? Obviously, it would be asking way too much to think that Rust could statically forbid data races, right? So much for Rust's vaunted memory safety guarantees, muahaha!

Huh, looks like the fuzz caught us again. "Rc<i32> cannot be sent between threads safely," because "the trait Send is not implemented for Rc<i32>." Other than, "curses, foiled again," what does this mean? It all comes down to how Rust knows that it's legal to transfer ownership of an Arc to another thread but not an Rc.

Traits in Rust are analogous to interfaces in Java and Kotlin, traits in Scala, and type classes in Haskell. A trait specifies a behavioral contract that any conformant types have to implement. The contract can include default implementations for one or more methods, but cannot directly specify any state. When instantiating a trait for a concrete type, the compiler copies down any default methods that were not overridden, complements them with overrides and explicit implementations of abstract methods, and verifies that all behavior is covered.

This is all well and good, and not too interesting. What's interesting in this regard is Rust's handful of magical marker traits. Markers statically attribute interesting properties to types, and thus inform the compiler's semantic validation and code generation.

Let's return to the concrete problem that sent us down the trait rabbit hole, the Send trait. The Send trait does not specify any methods, but instead specifies that conformant types may be transferred safely across thread boundaries. Rc does not implement Send, so the compiler forbids it from being captured by the closure passed to spawn. But Arc does implement Send, so Rust allows it.

Okay, let's try one more technique to cheat the compiler. Maybe we can embed an Rc within an Arc in order to bypass Rust's borrow checker. But… no. "Rc<i32> cannot be shared between threads safely," because "the trait Sync is not implemented for Rc<i32>". Sync is another marker trait. When a type implements Sync, its instances are permitted to be shared between threads.

Send and Sync are both *auto traits*: they are automatically implemented by the compiler whenever they apply, so they automatically apply to most immutable primitive data — booleans, integers, and floats. For composite types, like structs and enums, Rust looks at the fields and variants of the type to decide whether the type itself is Send or Sync. The rules are a bit complex, so I won't go into them here. But you can wrap types that don't implement Send or Sync in synchronized types like Mutex or RwLock, which *do* implement Send and Sync, thereby allowing transfer or sharing with other threads.

But Rc<T> doesn't implement Sync, so Arc<Rc<T>> *also* doesn't implement Sync. Therefore, it's impossible to send an Arc<Rc<T>> to another thread. Once again, Rust has statically ensured memory safety.

Now that we're talking about concurrent programming, though, a new observation emerges. Memory safety *is* thread safety. Rust ensures memory safety statically, and because it includes deep primitive support for ensuring memory safety across thread boundaries, via Send, Sync, and other mechanisms, it *also* statically protects against data races.

Unless you resort to really weird patterns with unsafe code and foreign function calls, Rust guarantees that any memory safe program is also free of data races. This is a unique selling point for Rust, folks. As of Q4 2023, no other programming language can claim to achieve this effect in quite this way. Rust's deep memory safety enables what the official Rust book aptly likes to call *fearless concurrency*.

I focused this talk around Rust's memory model, because in many ways, this *is* the real novelty. Rust has a lot of cool features — zero-cost abstractions, immutability by default, algebraic data types, pattern matching, higher-order functions, type deduction through unification based on a variant of the Hindley-Milner algorithm, traits with associated type parameters, object-oriented programming through trait objects, interior mutability, metaprogramming through macros, asynchronous I/O — but its deeply reasoned and superbly architected memory model is the heart and soul of Rust. It secures Rust's place in the pantheon of systems programming languages: safe, fast, supreme.

I truly hope that you enjoyed our time together today. I will make a transcript of this presentation, as well as the companion slide deck, generally available to anyone who's interested. At the back of the slide deck, you will find topics that I didn't have time to cover in the talk, like what IDEs you can use and what resources exist to ease newcomers into Rust development. I will open the meeting for questions and answers. If you have access to Slack, you can also reach out on #rust, and I will be happy to talk about the language or its ecosystem, and I'll do my best to answer any questions.