2025, 2nd Sept – Dublin, Republic of Ireland

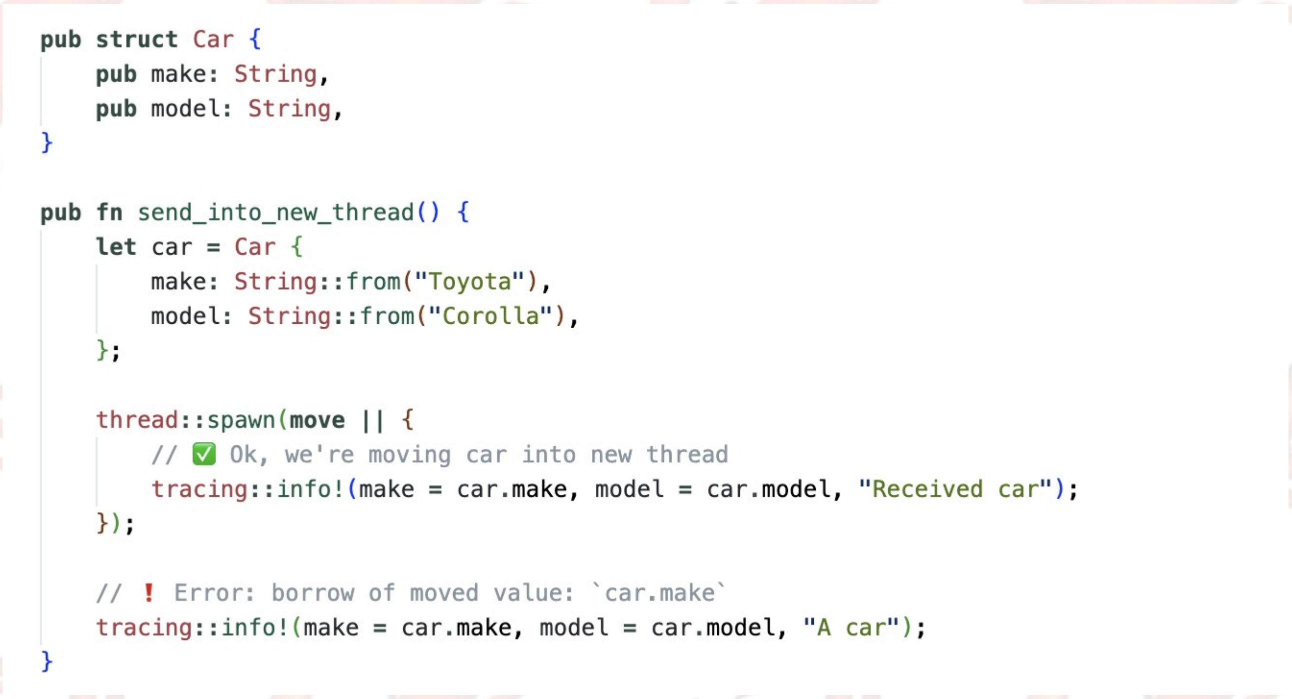
Rust vs. Threads

The intricate rules of the meticulous borrow checker in Rust

author: Sonia Sadhbh Kolasinska

Lifetimes and thread-safety

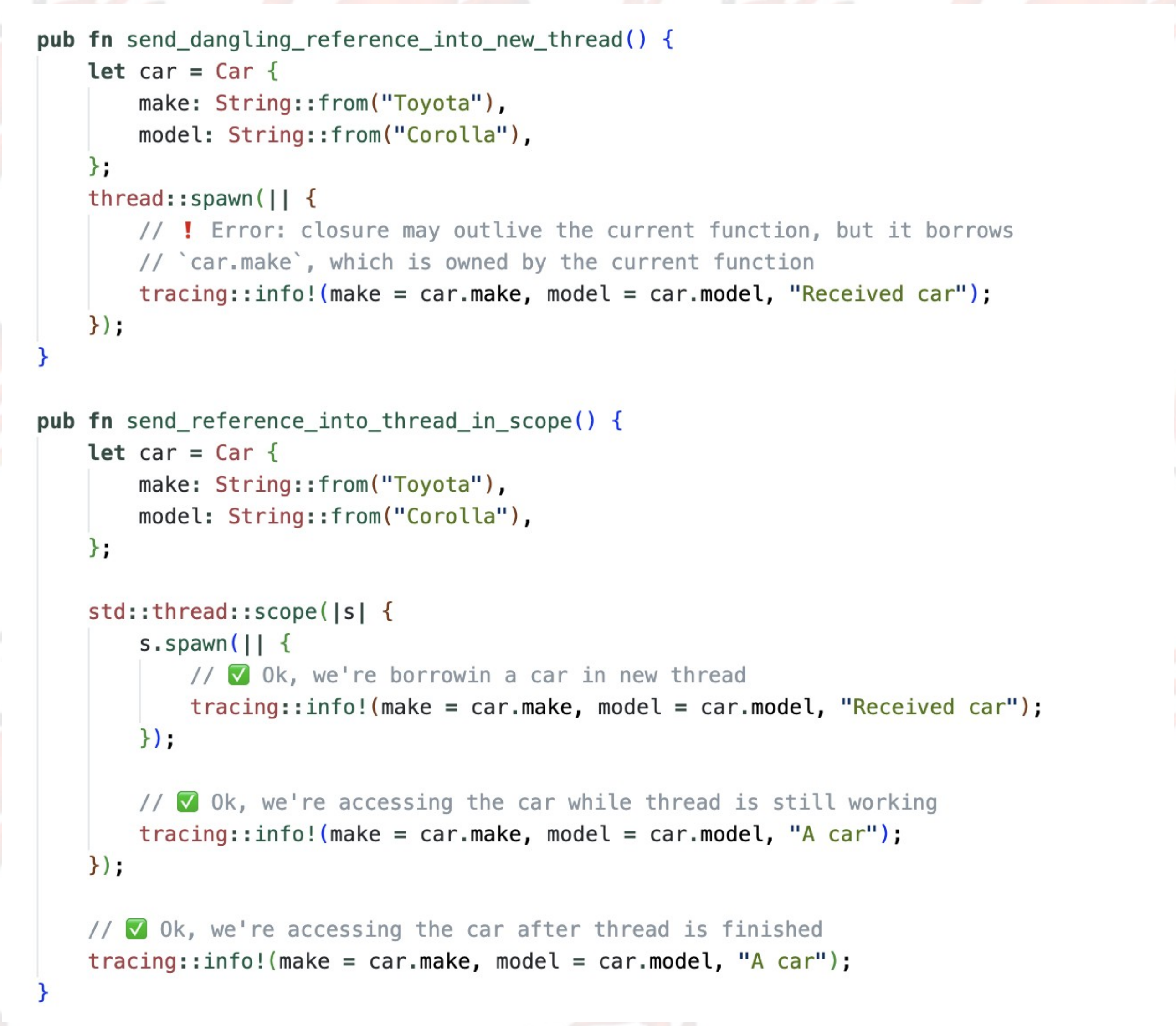
Rust introduces Send and Sync traits as a mechanism to protect against race-conditions. To fully understand the concept in Rust we always need to discuss borrowing rules. So in Rust everything is strictly guarded by borrow checker *(whenever you take a reference to the data it's called "borrowing")*. Say you have a String. String is a type that is definitely Send, because once you move it from one context *(stack frame or heap allocation)* to another nothing in the first context can access it anymore. However if you have a reference to that String *(if you "borrowed" a reference to it)* in that first context, then String remains Send, but borrow checker will complain if you try to use that reference after moving that String into context of that thread. So we guarantee that Send type can indeed be "sent" safely to another thread, because any reference to it will invalidate after the move, and race condition will not be possible.



```rust
pub struct Car {
    pub make: String,
    pub model: String,
}

pub fn send_into_new_thread() {
    let car = Car {
        make: String::from("Toyota"),
        model: String::from("Corolla"),
    };

    thread::spawn(move || {
        // ✅ Ok, we're moving car into new thread
        tracing::info!(make = car.make, model = car.model, "Received car");
    });

    // ❗ Error: borrow of moved value: `car.make`
    tracing::info!(make = car.make, model = car.model, "A car");
}
```

A reference can be sent to another thread

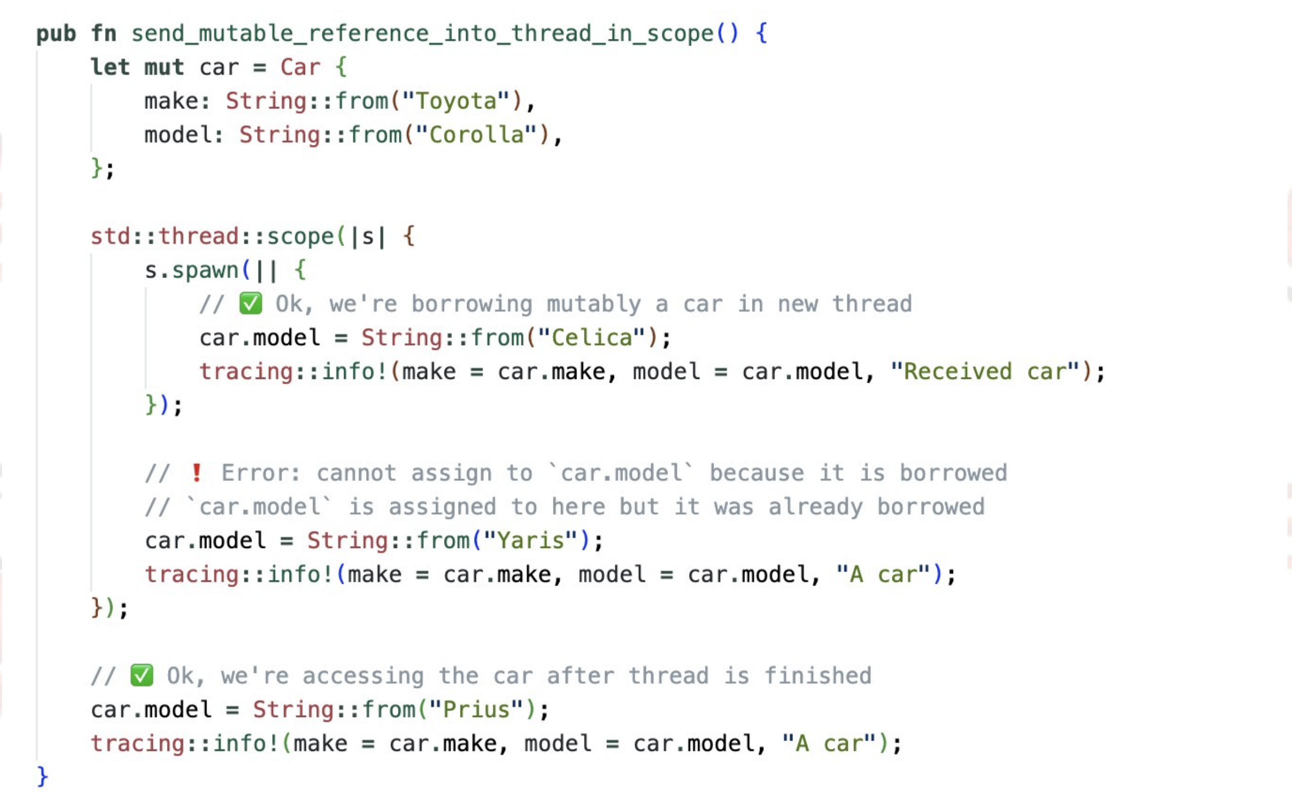
A reference can also be sent to another thread given that the lifetime of the object it points to is longer than the thread. This is possible, because reference is const by default in Rust, so having two threads accessing same object at same time via reference is a case of safe race, as object remains unchanged and two threads can read from it.



```rust
pub fn send_dangling_reference_into_new_thread() {
    let car = Car {
        make: String::from("Toyota"),
        model: String::from("Corolla"),
    };
    thread::spawn(|| {
        // ❗ Error: closure may outlive the current function, but it borrows
        // `car.make`, which is owned by the current function
        tracing::info!(make = car.make, model = car.model, "Received car");
    });
}

pub fn send_reference_into_thread_in_scope() {
    let car = Car {
        make: String::from("Toyota"),
        model: String::from("Corolla"),
    };

    std::thread::scope(|s| {
        s.spawn(|| {
            // ✅ Ok, we're borrowin a car in new thread
            tracing::info!(make = car.make, model = car.model, "Received car");
        });

        // ✅ Ok, we're accessing the car while thread is still working
        tracing::info!(make = car.make, model = car.model, "A car");
    });

    // ✅ Ok, we're accessing the car after thread is finished
    tracing::info!(make = car.make, model = car.model, "A car");
}
```

When type cannot be sent to another thread?

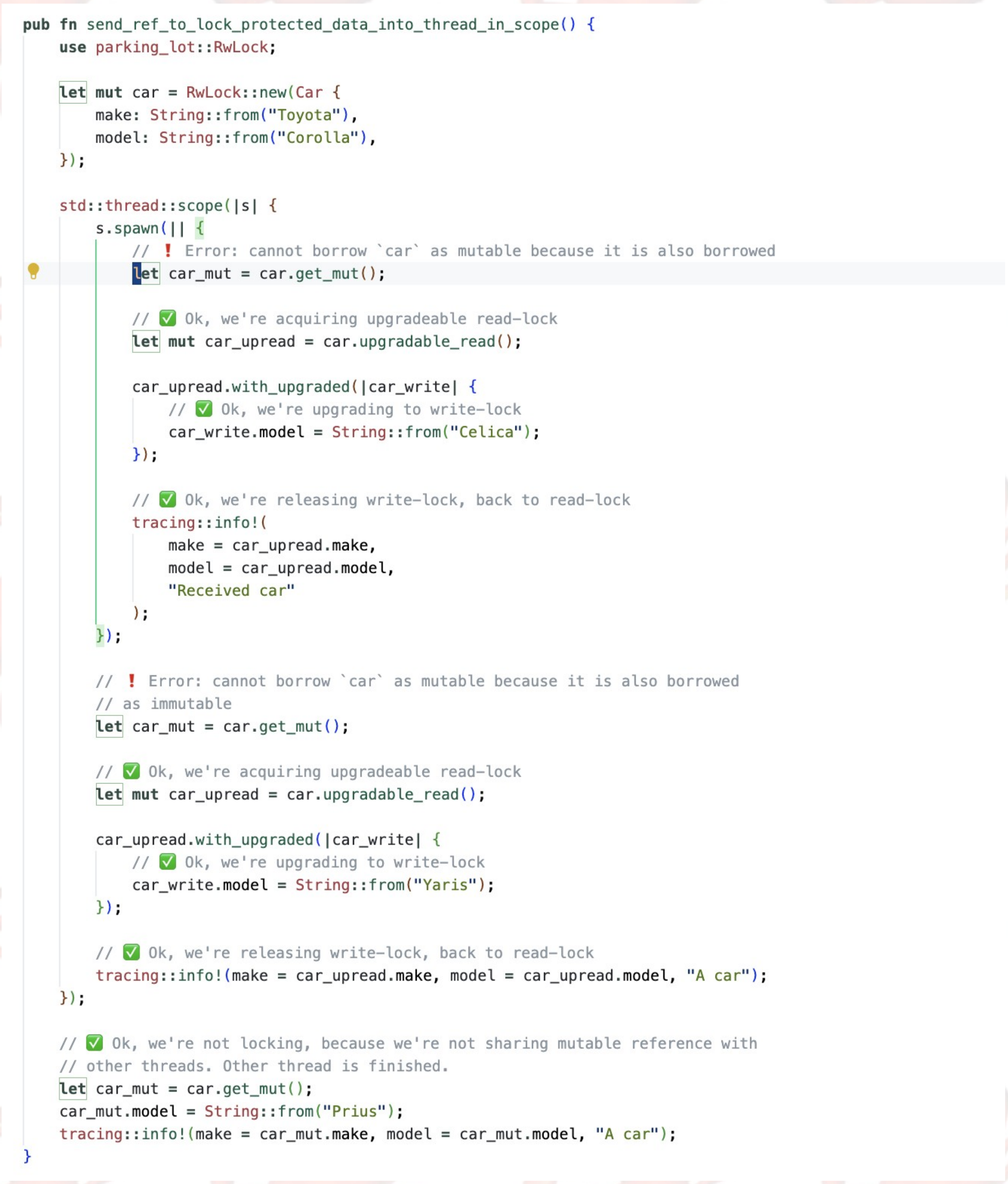
So when is the type not-Send and not-Sync? One and only case of such type is mutable reference or struct containing mutable reference. This is where Rust's Send+Sync traits shine. The race condition is only possible if at least one of the two threads accessing a shared data attempts to modify that data. In Rust you cannot just go ahead and modify data. Rust is all const until you un-const it, i.e. until you need mutable reference. When you have "borrowed" a mutable reference to your data, and you send that reference to the other thread, that data cannot be accessed anymore from the first context for the whole duration of the thread, and also the data referred by it must live longer that that thread. If that the case how can we actually share things between threads?



```rust
pub fn send_mutable_reference_into_thread_in_scope() {
    let mut car = Car {
        make: String::from("Toyota"),
        model: String::from("Corolla"),
    };

    std::thread::scope(|s| {
        s.spawn(|| {
            // ✅ Ok, we're borrowing mutably a car in new thread
            car.model = String::from("Celica");
            tracing::info!(make = car.make, model = car.model, "Received car");
        });

        // ❗ Error: cannot assign to `car.model` because it is borrowed
        // `car.model` is assigned to here but it was already borrowed
        car.model = String::from("Yaris");
        tracing::info!(make = car.make, model = car.model, "A car");
    });

    // ✅ Ok, we're accessing the car after thread is finished
    car.model = String::from("Prius");
    tracing::info!(make = car.make, model = car.model, "A car");
}
```

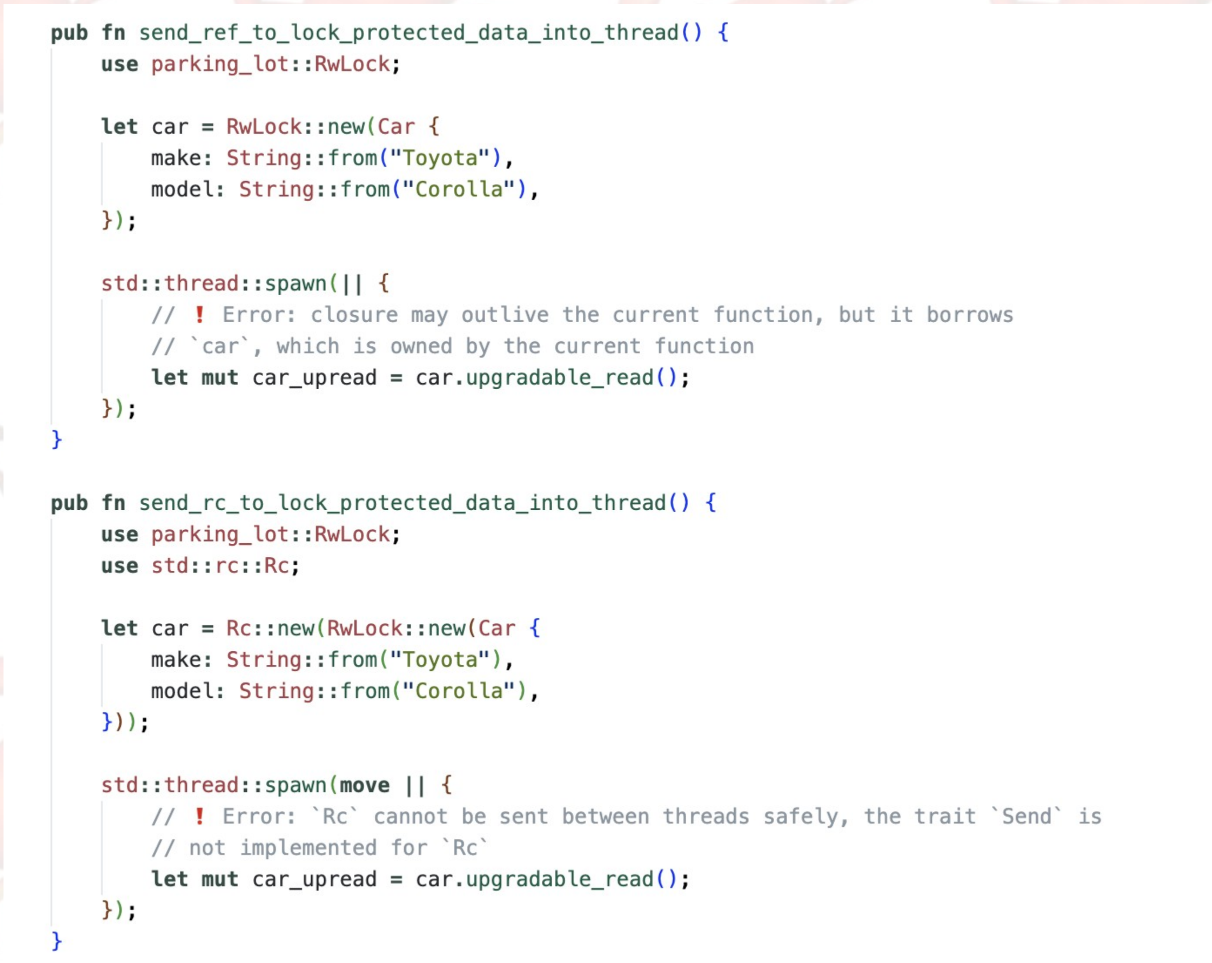
Sharing data across threads

Sharing in Rust is only possible via reference counted pointers. There is two of them: Rc and Arc. Rc cannot be sent to another thread, because other thread might be cloning it, and it could already have been cloned on the first thread, and for cloning reference count needs to be Send+Sync, and in case of Rc it's not. Arc on the other hand can be sent to another thread, because cloning atomically increments reference count. But there is one gotcha...



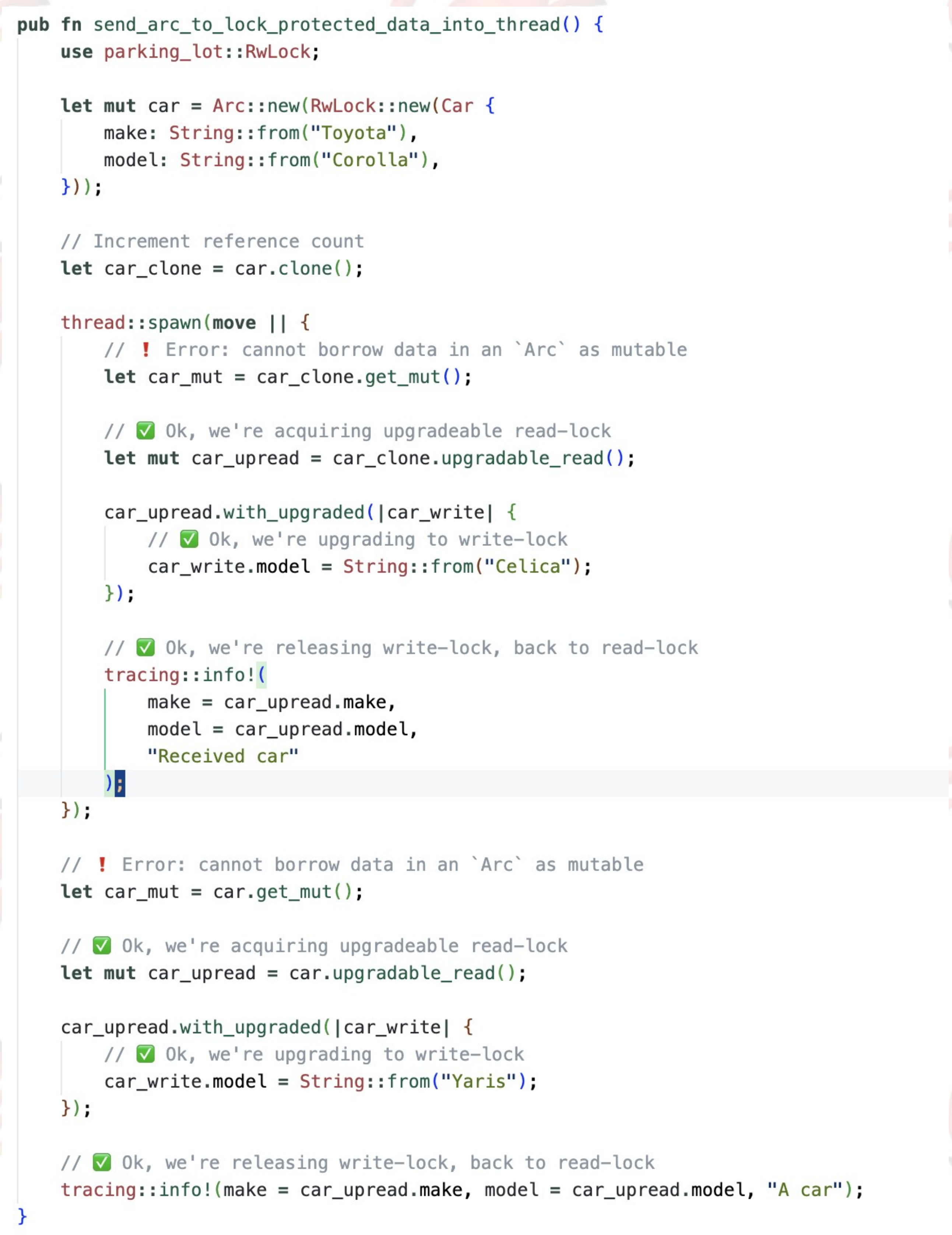
```rust
pub fn send_ref_to_lock_protected_data_into_thread_in_scope() {
    use parking_lot::RwLock;

    let mut car = RwLock::new(Car {
        make: String::from("Toyota"),
        model: String::from("Corolla"),
    });

    std::thread::scope(|s| {
        s.spawn(|| {
            // ❗ Error: cannot borrow `car` as mutable because it is also borrowed
            let car_mut = car.get_mut();

            // ✅ Ok, we're acquiring upgradeable read-lock
            let mut car_upread = car.upgradable_read();

            car_upread.with_upgraded(|car_write| {
                // ✅ Ok, we're upgrading to write-lock
                car_write.model = String::from("Celica");
            });

            // ✅ Ok, we're releasing write-lock, back to read-lock
            tracing::info!(
                make = car_upread.make,
                model = car_upread.model,
                "Received car"
            );
        });

        // ❗ Error: cannot borrow `car` as mutable because it is also borrowed
        // as immutable
        let car_mut = car.get_mut();

        // ✅ Ok, we're acquiring upgradeable read-lock
        let mut car_upread = car.upgradable_read();

        car_upread.with_upgraded(|car_write| {
            // ✅ Ok, we're upgrading to write-lock
            car_write.model = String::from("Yaris");
        });

        // ✅ Ok, we're releasing write-lock, back to read-lock
        tracing::info!(make = car_upread.make, model = car_upread.model, "A car");
    });

    // ✅ Ok, we're not locking, because we're not sharing mutable reference with
    // other threads. Other thread is finished.
    let car_mut = car.get_mut();
    car_mut.model = String::from("Prius");
    tracing::info!(make = car_mut.make, model = car_mut.model, "A car");
}
```

Arc only provides you with const reference to pointed data. Why is that? This is because multiple clones of same Arc point to same data, and having a method in Arc to provide mutable reference would immediately create race condition. So to actually allow modification of the data in the Arc you need to put it into RwLock (or Mutex).



```rust
pub fn send_ref_to_lock_protected_data_into_thread() {
    use parking_lot::RwLock;

    let car = RwLock::new(Car {
        make: String::from("Toyota"),
        model: String::from("Corolla"),
    });

    std::thread::spawn(|| {
        // ! Error: closure may outlive the current function, but it borrows
        // `car`, which is owned by the current function
        let mut car_upread = car.upgradable_read();
    });
}

pub fn send_rc_to_lock_protected_data_into_thread() {
    use parking_lot::RwLock;
    use std::rc::Rc;

    let car = Rc::new(RwLock::new(Car {
        make: String::from("Toyota"),
        model: String::from("Corolla"),
    }));

    std::thread::spawn(move || {
        // ! Error: `Rc` cannot be sent between threads safely, the trait `Send` is
        // not implemented for `Rc`
        let mut car_upread = car.upgradable_read();
    });
}
```

So now you've got clones of Arc pointing to same RwLock guarding access to your data, and thread to access that data needs to acquire the lock first, which then gives to the user a LockGuard. Another interesting fact is that you're holding the lock for as long as lifetime of that LockGuard. For example you cannot have a method that would acquire write lock, and return mutable reference to your data, because you'd need to return it together with the LockGuard, because the lifetime of that mutable reference is tied to the lifetime of that LockGuard. Right... all sounds complicated... how do we deal with all this? Just use Arc<RwLock<YourData>> if you need to share YourData across threads.



```rust
pub fn send_arc_to_lock_protected_data_into_thread() {
    use parking_lot::RwLock;

    let mut car = Arc::new(RwLock::new(Car {
        make: String::from("Toyota"),
        model: String::from("Corolla"),
    }));

    // Increment reference count
    let car_clone = car.clone();

    thread::spawn(move || {
        // ❗ Error: cannot borrow data in an `Arc` as mutable
        let car_mut = car_clone.get_mut();

        // ✅ Ok, we're acquiring upgradeable read-lock
        let mut car_upread = car_clone.upgradable_read();

        car_upread.with_upgraded(|car_write| {
            // ✅ Ok, we're upgrading to write-lock
            car_write.model = String::from("Celica");
        });

        // ✅ Ok, we're releasing write-lock, back to read-lock
        tracing::info!(
            make = car_upread.make,
            model = car_upread.model,
            "Received car"
        );
    });

    // ❗ Error: cannot borrow data in an `Arc` as mutable
    let car_mut = car.get_mut();

    // ✅ Ok, we're acquiring upgradeable read-lock
    let mut car_upread = car.upgradable_read();

    car_upread.with_upgraded(|car_write| {
        // ✅ Ok, we're upgrading to write-lock
        car_write.model = String::from("Yaris");
    });

    // ✅ Ok, we're releasing write-lock, back to read-lock
    tracing::info!(make = car_upread.make, model = car_upread.model, "A car");
}
```