Written by Piotr Sarnacki
on May 28, 2022

Arc and Mutex in Rust

When writing concurrent Rust you will encounter `Arc` and `Mutex` types sooner or later. And although `Mutex` might already sound familiar as it's a concept known in many languages, chances are you haven't heard about `Arc` before Rust. What's more, you can't fully understand these concepts without tying them to the ownership model in Rust. This post is my take on understanding `Arc` and `Mutex` in Rust.

Typically when you share data in a concurrent environment you either share memory or pass data as messages. You might often hear that passing messages (for example by using channels) is a preferred way to handle concurrency, but in Rust I don't think the safety or correctness differences are as big as in other languages due to the ownership model in Rust. Or more specifically: you can't have a data race in safe Rust. That's why when I choose between message passing or memory sharing in Rust, I do it mostly in relation to convinience, not safety.

If you choose to share data by sharing memory you will quickly encounter that you can't do much without `Arc` and `Mutex`. `Arc` is a smart pointer that let's you safely share a value between multiple threads. `Mutex` is a wrapper over another type, which allows safe mutability across threads. In order to fully understand these concepts, though, let's dive into the ownership model.

Ownership in Rust

If you tried to distill the ownership model in Rust, you would probably get the following points:

- a value can have only one owner
- you can have multiple shared immutable references to a value
- you can have only one mutable reference to a value

Let's see how it plays out. Given a `User` struct containing a `String` field named `name` we create a thread and print out a message for the user:

```
use std::thread::spawn;

```

```
#[derive(Debug)]
struct User {
    name: String
}

fn main() {
    let user = User { name: "drogus".to_string() };

    spawn(move || {
        println!("Hello from the first thread {}", user.name);
    }).join().unwrap();
}
```

So far so good, the program compiles and prints the message. Now imagine we need to add a second thread that also has access to the `user` instance:

```
fn main() {
    let user = User { name: "drogus".to_string() };

    let t1 = spawn(move || {
        println!("Hello from the first thread {}", user.name);
    });

    let t2 = spawn(move || {
        println!("Hello from the second thread {}", user.name);
    });

    t1.join().unwrap();
    t2.join().unwrap();
}
```

With this code we get the following error:

```
error[E0382]: use of moved value: `user.name`
  --> src/main.rs:15:20
   |
11 |     let t1 = spawn(move || {
   |                    ------- value moved into closure here
```

```
12 |         println!("Hello from the first thread {}", user.name);
   |                                                    --------- variabl
...
15 |     let t2 = spawn(move || {
   |                    ^^^^^^^ value used here after move
16 |         println!("Hello from the second thread {}", user.name);
   |                                                     --------- use occ
   |
   = note: move occurs because `user.name` has type `String`, which does
```

What does the compiler want? The error reads "use of moved value `user.name`". The compiler is even nice enough to point us to specific places where the problem occurs. We first move the value to the first thread on line 11 and then we try to do the same thing with the second thread on line 15. If you look at the ownership rules, this shouldn't be surprising. A value can have only one owner. With the current version of the code we need to "move" the value to the first thread if we want to use it, and thus we can't move it to the other thread. It already changed ownership. But we don't mutate the data, right? Which means we can have multiple shared references. Let's try that.

```rust
fn main() {
    let user = User { name: "drogus".to_string() };

    let t1 = spawn(|| {
        println!("Hello from the first thread {}", &user.name);
    });

    let t2 = spawn(|| {
        println!("Hello from the second thread {}", &user.name);
    });

    t1.join().unwrap();
    t2.join().unwrap();
}
```

I removed the `move` keyword in the thread closures and I made the threads borrow the `user` value immutably, or in other words get a shared reference, which is represented by the ampersand. With this code we get the following:

```
error[E0373]: closure may outlive the current function, but it borrows `u
```

```
  --> src/main.rs:15:20
   |
15 |     let t2 = spawn(|| {
   |                    ^^ may outlive borrowed value `user.name`
16 |         println!("Hello from the first thread {}", &user.name);
   |                                                    --------- `user.
   |
note: function requires argument type to outlive `'static`
  --> src/main.rs:15:14
   |
15 |       let t2 = spawn(|| {
   |  ______________^
16 | |         println!("Hello from the second thread {}", &user.name);
17 | |     });
   | |______^
help: to force the closure to take ownership of `user.name` (and any othe
   |
15 |     let t2 = spawn(move || {
   |                    ++++
```

Now the error says that the closure can outlive the function. In other words the Rust compiler can't guarantee that the closure in the thread will finish before the `main()` function finishes. Threads are borrowing the user struct, but it's still owned by the main function. In this scenario if the main function finishes, the user struct goes out of scope and the memory is dropped. Thus if it was allowed to share the value with threads in that manner, there could be a scenario when a thread is trying to read freed memory. Which is an undefined behaviour and we certainly don't want that.

The note also says that it may help to move the variable `user` to the thread in order to avoid borrowing, but we're just coming from that scenario, so it's no good. Now, there are two easy solutions to fix this and one of them is to use `Arc`, but let's explore the other solution first: scoped threads.

Scoped threads

Scoped threads is a feature available either from an excellent crossbeam crate or as an experimental nightly feature in Rust. For the purpose of this article I will use crossbeam, but the API is very similar for both versions. After adding `crossbeam = "0.8"` to dependencies in `Cargo.toml` this code will work without problems

```
use crossbeam::scope;

```

```rust
#[derive(Debug)]
struct User {
    name: String,
}

fn main() {
    let user = User {
        name: "drogus".to_string(),
    };

    scope(|s| {
        s.spawn(|_| {
            println!("Hello from the first thread {}", &user.name);
        });

        s.spawn(|_| {
            println!("Hello from the second thread {}", &user.name);
        });
    })
    .unwrap();
}
```

The way scoped threads work is all of the threads created in the scope are guaranteed to be finished before the `scope` closure finishes. Or in other words

- before the scoped closure goes out of scope, the threads are joined and awaited to be finished. Thanks to that the compiler knows that none of the borrows will outlive the owner.

One interesting thing to note here is that as a human reader we would have interpreted both of these programs as valid. In the version that Rust rejects we join both threads before the `main()` function finishes, so it would be actually safe to share the user value with the threads. This is, unfortunately, something that you may encounter when writing in Rust. Writing a compiler that would accept all of the valid programs is not possible, thus we're left with the next best thing: a compiler that will reject all invalid programs at a cost of being overly strict. Scoped threads is a future written specifically to allow us to write this code in a way that compiler can accept.

As useful as the scoped threads feature is, however, you can't always use it, for example when writing async code. Let's get to the `Arc` solution then.

Arc to the rescue

`Arc` is a smart pointer enabling sharing data between threads. Its name is a shortcut for "atomic reference counter". The way `Arc` works is essentially to wrap a value we're trying to share and act as a pointer to it. `Arc` keeps track of all of the copies of the pointer and as soon as the last pointer goes out of scope it can safely free the memory. The solution to our small problem with `Arc` would look something like this:

```rust
 1 use std::thread::spawn;
 2 use std::sync::Arc;
 3 
 4 #[derive(Debug)]
 5 struct User {
 6     name: String
 7 }
 8 
 9 fn main() {
10     let user_original = Arc::new(User { name: "drogus".to_string() });
11 
12     let user = user_original.clone();
13     let t1 = spawn(move || {
14         println!("Hello from the first thread {}", user.name);
15     });
16 
17     let user = user_original.clone();
18     let t2 = spawn(move || {
19         println!("Hello from the first thread {}", user.name);
20     });
21 
22     t1.join().unwrap();
23     t2.join().unwrap();
24 }
```

Let's go through it step by step. First, on line 10, we create a user value, but we also wrap it with an `Arc`. Now the value is stored in memory and `Arc` acts only as a pointer. Whenever we clone the `Arc` we only clone the reference, not the user value itself. On lines 12 and 17 we clone the `Arc` and thus a copy of the pointer is moved to each of the threads. As you can see `Arc` allows us to share the data regardless of the lifetimes. In this example we will have three pointers to the user value. One created when an `Arc` is created, one created by cloning before starting the first thread and moved to the first thread and one created by cloning before starting the second thread and moved the first thread. As long as any of these pointers is alive, Rust will not free the memory. But when both the threads and the main function finish, all of the `Arc` pointers will get out of scope, dropped and as soon as the last one drops, it will also drop the user value.

Send and Sync

Let's go a bit deeper, though. If you look at the Arc documentation, you will see it implements Send and Sync traits, but *only* if the wrapped type also implements both Send and Sync. In order to understand what it means and why it's implemented this way let's start by defining Send and Sync.

The Rustonomicon defines Send and Sync as:

- A type is Send if it is safe to send it to another thread.
- A type is Sync if it is safe to share between threads (T is Sync if and only if &T is Send).

Feel free to read about these traits on Rustonomicon, but I'll also try to share my understanding here. Both Send and Sync are traits acting as markers - they don't have any implemented methods nor they require you to implement anything. What they allow is to notify the compiler about a type's ability to be shared or sent between threads. Let's start with Send, which is a bit more straightforward. What it means is that you can't send type which is !Send (read: not Send) to another thread. For example you can't send it through a channel nor can you move it to a thread. For example this code will not compile:

```rust
#![feature(negative_impls)]

#[derive(Debug)]
struct Foo {}
impl !Send for Foo {}

fn main() {
    let foo = Foo {};
    spawn(move || {
        dbg!(foo);
    });
}
```

Send and Sync are autoderived, meaning that for example if all of the attributes of a type are Send, the type will also be Send. This code uses an experimental feature called negative_impls, which lets us tell the compiler "I explicitly want to mark this type as !Send". Trying to compile this code will result in an error:

```
`Foo` cannot be sent between threads safely
```

The same would happen if you created a channel to send foo to a thread. So now what with Arc? As you might have guessed it will also not help, this will also error out in the same way (and the same would be true for a `!Sync` type as `Arc` needs both traits):

```rust
#![feature(negative_impls)]

#[derive(Debug)]
struct Foo {}
impl !Send for Foo {}

fn main() {
    let foo = Arc::new(Foo {});
    spawn(move || {
        dbg!(foo);
    });
}
```

Now, why is that the case? Isn't `Arc` supposed to be wrapping our type and give it more capabilities? While this is certainly true, `Arc` can't magically make our type threadsafe. I will give you a more in-depth example to show you why at the end of this article, but for now let's continue with learning on how to use these types.

What we learned so far is this: `Arc` enables us to share references to types that are `Send + Sync` between threads without us having to worry about lifetimes (because it's not a regular reference, but rather a smart pointer).

Modifying data with Mutex

Now let's talk about `Mutex`. Mutexes in many languages are treated like semaphores. You create a mutex object and you can guard a certain piece (or pieces) of the code with the mutex in a way that only one thread at a time can access the guarded place. In Rust `Mutex` behaves more like a wrapper. It consumes the underlying value and let's you access it only after locking the mutex. Typically `Mutex` is used with conjunction with `Arc` to make it easier to share it between threads. Let's look at the following example:

```rust
use std::time::Duration;
use std::{thread, thread::sleep};
use std::sync::{Arc, Mutex};

struct User {
    name: String
}

```

```
 9 fn main() {
10     let user_original = Arc::new(Mutex::new(User { name: String::from("
11 
12     let user = user_original.clone();
13     let t1 = thread::spawn(move || {
14         let mut locked_user = user.lock().unwrap();
15         locked_user.name = String::from("piotr");
16         // after locked_user goes out of scope, mutex will be unlocked
17         // but you can also explicitly unlock it with:
18         // drop(locked_user);
19     });
20 
21     let user = user_original.clone();
22     let t2 = thread::spawn(move || {
23         sleep(Duration::from_millis(10));
24 
25         // it will print: Hello piotr
26         println!("Hello {}", user.lock().unwrap().name);
27     });
28 
29     t1.join().unwrap();
30     t2.join().unwrap();
31 }
```

Let's go over it. in the first line of the `main()` function we create an instance of the `User` struct and we wrap it with a `Mutex` and an `Arc`. With an `Arc` we can easily clone the pointer and thus share the mutex between threads. In the 13th line you can see the mutex is locked and since that moment the underlying value can be used exclusively by this thread. Then we modify the value in the next line. The mutex is unlocked once the locked guard goes out of scope or we manually drop it with `drop(locked_user)`.

In the second thread we wait 10ms and print the name, which should be the name updated in the first thread. This time locking is done in one line, so the mutex will be dropped in the same statement.

One more thing that is worth mentioning is the `unwrap()` method we call after `lock()`. `Mutex` from the standard library has a notion of being poisoned. If a thread panics while the mutex is locked we can't be certain if the value inside `Mutex` is still valid and thus the default behaviour is to return an error instead of a guard. So `Mutex` can either return an `Ok()` variant with the wrapped value as an argument or an error. You can read more about it in the docs. In general leaving `unrwap()` methods in the production code is not recommended, but in case of `Mutex` it might be a valid strategy - if a mutex has been poisoned we might decide that the application state is invalid and crash the application.

Another interesting thing about `Mutex` is that as long as a type inside the `Mutex` is `Send`, `Mutex` will also be `Sync`. This is because `Mutex` ensures that only one thread can get access to the underlying value and thus it's safe to share `Mutex` between threads.

Mutex: add Sync to a Send type

As you may remember from the beginning of the article, `Arc` needs an underlying type to be `Send + Sync` in order for `Arc` to be `Send + Sync` too. `Mutex` only requires and underlying type to be `Send` in order for `Mutex` to be `Send + Sync`. In other words `Mutex` will make a `!Sync` type `Sync`, so you can share it between threads and modify it too.

Mutex without `Arc`?

An interesting question that you may ask is if `Mutex` can be used without `Arc`. I encourage you to think about it a little before reading further: what does it mean that `Mutex` is `Send + Sync` for types that are `Send`?

If you get back to the first part of this post you can see what it means for the `Arc` type and in case of `Mutex` it means a very similar thing. If we can use something like scope threads it's entirely possible to use `Mutex` without `Arc`:

```rust
use crossbeam::scope;
use std::{sync::Mutex, thread::sleep, time::Duration};

#[derive(Debug)]
struct User {
    name: String,
}

fn main() {
    let user = Mutex::new(User {
        name: "drogus".to_string(),
    });

    scope(|s| {
        s.spawn(|_| {
            user.lock().unwrap().name = String::from("piotr");
        });

        s.spawn(|_| {
            sleep(Duration::from_millis(10));

            // should print: Hello piotr
            println!("Hello {}", user.lock().unwrap().name);
        });
    })
    .unwrap();
}
```

In this program we achieve the same goal. We are accessing a value behind a mutex in two separate threads, but we share mutexes by reference and not by using an `Arc`. But again, this is not always possible, for example in async code, so `Mutex` is very often used along with `Arc`.

Summary

I'm hoping that in this article I helped you understand what are `Arc` and `Mutex` types in Rust and how to use them. To sum it up I would say you would typically use `Arc` whenever you want to share data between threads and you can't do so using regular references. You would also use `Mutex` if you need to modify data you share between threads. And then you would use `Arc<Mutex<...>>` whenever you want to modify data you share between threads and you can't share a mutex using references.

Bonus: Why `Arc` needs type to be `Sync`

Now let's get back to the question of "why `Arc` needs the underlying type to be both `Send` and `Sync` to mark it as `Send` and `Sync`). Feel free to ignore this last section, though, it's not really needed for you to use `Arc` and `Mutex` in your code. It might help you understand markter traits a bit better.

Lets take `Cell` as an example. `Cell` wraps another type and enables "interior mutability" or in other words it allows us to modify a value inside an immutable struct. `Cell` is `Send`, but it's `!Sync`.

An example of using `Cell` would be:

```rust
use std::cell::Cell;

struct User {
    age: Cell<usize>
}

fn main() {
    let user = User { age: Cell::new(30) };

    user.age.set(36);

    // will print: Age: 36
    println!("Age: {}", user.age.get());
}
```

`Cell` is useful in some situations, but it isn't thread safe or in other words it's `!Sync`. If you somehow shared a value wrapped in a cell between multiple threads you could modify the same place in memory from two threads, for example:

```rust
// this example will not compile, `Cell` is `!Sync` and thus
// `Arc` will be `!Sync` and `!Send`
use std::cell::Cell;

struct User {
    age: Cell<usize>
}

fn main() {
    let user_original = Arc::new(User { age: Cell::new(30) });

    let user = user_original.clone();
    std::thread::spawn(move || {
        user.age.set(2);
    });

    let user = user_original.clone();
    std::thread::spawn(move || {
        user.age.set(3);
    });
}
```

If that worked, it could result in an undefined behaviour. That's why `Arc` will not work with any type that is not `Send` *nor* `Sync`. At the same time `Cell` is `Send`, meaning that you can send it between threads. Why is that? Sending, or in other words moving, will not make a value accessible from more than one thread, it will have to always be only one thread. Once you move it to another, the previous thread doesn't own the value anymore. With that in mind, we can always mutate a `Cell` locally.

Bonus: why `Arc` needs type

At this point you might also wonder why `Arc` will not provide the `Send` trait for a `!Send` type, either. One of the types in Rust which is `!Send` is `Rc`. `Rc` is a cousin of `Arc`, but it's not "atomic", `Rc` expands to just "reference counter". Its role is pretty much the same as `Arc`, but it can only be used in a single thread. Not only it can't be shared between threads, but also it can't be moved between threads. Let's see why.

```rust
// this code won't compile, Rc is !Send and !Sync
use std::rc::Rc;

fn main() {
    let foo = Rc::new(1);

    let foo_clone = foo.clone();
    std::thread::spawn(move || {
        dbg!(foo_clone);
    });

    let foo_clone = foo.clone();
    std::thread::spawn(move || {
        dbg!(foo_clone);
    });
}
```

This example won't compile, because `Rc` is `!Sync + !Send`. Its internal counter is not atomic and thus sharing it between threads could result in an inaccurate count of references. Now imagine that `Arc` would make `!Send` types `Send`:

```rust
use std::rc::Rc;
use std::sync::Arc;

#[derive(Debug)]
struct User {
    name: Rc<String>,
}
unsafe impl Send for User {}
unsafe impl Sync for User {}

fn main() {
    let foo = Arc::new(User {
        name: Rc::new(String::from("drogus")),
    });

    let foo_clone = foo.clone();
    std::thread::spawn(move || {
        let name = foo_clone.name.clone();
    });

    let foo_clone = foo.clone();
    std::thread::spawn(move || {
        let name = foo_clone.name.clone();
    });
}
```

This example will compile, but it's wrong, please don't do it in your actual code! In here I define a `User` struct, which holds an `Rc` inside. Because `Send` and `Sync` are autoderived and `Rc` is `!Send + !Sync`, the `User` struct should also be `!Send + !Sync`, but we can explicitly tell the compiler to mark it differently, in this case `Send + Sync`, using `unsafe impl` syntax.

Now you can clearly see what would go wrong if `Arc` allowed `!Send` types to be moved between threads. In the example `Arc` clones are moved into separate threads and then nothing is stopping us from cloning the `Rc` type. And because `Rc` type is not thread safe, it could result in an inacurate count of references and thus could either free memory too soon or it could not free it at all even though it should.

I know that this article is a long one, so kudos too all of you that got it all the way here, thanks!