Lecture 8: Last of us

Barinov Denis

April 23, 2025

barinov.diu@gmail.com

Interior mutability: `Cell` and `RefCell`

Interior mutability

Rust memory safety is based on this rule: Given an object `T`, it is only possible to have one of the following:

- Having several immutable references (`&T`) to the object (also known as aliasing).
- Having one mutable reference (`&mut T`) to the object (also known as mutability).

Interior mutability

But sometimes, we do want to modify the object having multiple aliases. In Rust, this is achieved using a pattern called *interior mutability*.

Important: since Safe Rust is memory safe and does not have undefined behavior, all of these primitives must also guarantee not to break Rust's fundamental guarantees.

Interior mutability

Examples:

- Modifying `Rc` (main use case for this lecture).
- Atomics.
- Mutexes.
- RWLocks.

Basically, any mutation through `&` is interior mutability.

In this lecture, we'll focus on **single-threaded** modification.

Cell

Let's solve the problem by creating a safe abstraction over unsafe modification.

```
pub struct Cell<T: ?Sized> {
    // ...
}

impl<T: Copy> Cell<T> {
    // ...
}
```

The `Cell` type solves this problem by **copying** the underlying value. Because of that, key functionality is **not available** when the type is not `Copy`.

Cell

The most important functions:

```rust
fn new(value: T) -> Cell<T>;
fn set(&self, val: T);

// Only when 'T: Copy'!
fn get(&self) -> T;
```

`set` moves the new value inside the `Cell`. `get` copies the value and gives it to the user.

`Cell`: Why `Copy` and not `Clone`?

Since `Clone` allows to write any logic inside it, we can create situations when we'll cause memory unsafety and undefine behavior!

Let's use `Option` type to create a self referential `Cell` (in the example, we'll assume `Cell` is implemented for `Clone` types).[1]

```rust
struct BadClone<'a> {
    data: i32,
    pointer: &'a Cell<Option<BadClone<'a>>>,
}
```

[1] Why does Cell require Copy instead of Clone?

`Cell`: Why `Copy` and not `Clone`?

```rust
impl<'a> Clone for BadClone<'a> {
    fn clone(&self) -> BadClone<'a> {
        // Grab a reference to our internals
        let data: &i32 = &self.data;
        println!("before: {}", *data);

        // lear out the cell we point to...
        self.pointer.set(None);

        // Print it again (should be no change!)
        println!("after: {}", *data);
        BadClone { data: self.data, pointer: self.pointer }
    }
}
```

`Cell`: Why `Copy` and not `Clone`?

```
let cell = Cell::new(None);
cell.set(Some(BadClone {
    data: 12345678,
    pointer: &cell,
}));
cell.get();
```

Possible output:

```
before: 12345678
after: 0
```

This means the `Cell` with `Clone` is **unsound**.

More generally, this bug is called *reentrancy*.

RefCell

The `RefCell` is one more way to ensure that you correctly modify the variable.

```
type BorrowFlag = isize;

pub struct RefCell<T>
where
    T: ?Sized,
{
    borrow: Cell<BorrowFlag>,
    // ...
}
```

The difference is that this type gives a mutable or immutable link and **counts these links** in runtime instead of copying the underlying type.

RefCell

The most important functions:

```rust
fn new(value: T) -> RefCell<T>;
fn get_mut(&mut self) -> &mut T;
fn borrow(&self) -> Ref<'_, T>;
fn borrow_mut(&self) -> RefMut<'_, T>;
fn try_borrow(&self) -> Result<Ref<'_, T>, BorrowError>;
fn try_borrow_mut(&self) -> Result<RefMut<'_, T>, BorrowMutError>;
```

Structures `Ref` and `RefMut` are pinned to the `RefCell` and will change internal counter when references are dropped.

Default and `try_` variants differs in how they notify about unsuccessful borrow: by panic or using `Result`.

Cell, RefCell and Rc

It's quite common pattern to use Cell and RefCell together with Rc. Rc's value is always immutable to make it safe, and if you want to modify it, you should use Cell or RefCell with runtime checks.

```
pub struct List<T> {
    head: Link<T>,
    tail: Link<T>,
}

type Link<T> = Option<Rc<RefCell<Node<T>>>>;
```

`Cell` and `RefCell`: Summary

- Since it's not possible to check safe invariants on compile time when modifying from multiple places, we moved them to runtime.

`Cell` and `RefCell`: Summary

- Since it's not possible to check safe invariants on compile time when modifying from multiple places, we moved them to runtime.
- In the case of `Cell`, we are just copying the value, and noted impossibility of using `Clone`.

`Cell` and `RefCell`: Summary

- Since it's not possible to check safe invariants on compile time when modifying from multiple places, we moved them to runtime.
- In the case of `Cell`, we are just copying the value, and noted impossibility of using `Clone`.
- In the case of `RefCell`, we are giving a reference, counting them in runtime.

Declarative macros

Declarative macros

Let's check the `json` macro from crate serde!

```rust
let value = json!({
    "code": 200,
    "success": true,
    "payload": {
        "features": [
            "serde",
            "json"
        ]
    }
});
```

Declarative macros

Let's start with the easiest example. Suppose we want to create a vector from the list of arguments. If we do this by hand, the result will be as follows:

```rust
let mut a = Vec::new();
a.push(1);
a.push(1);
a.push(1);
```

Declarative macros

But we already know there's a macro that is doing the same!

```
let a = vec![1; 3];
let a = vec![1, 1, 1];
```

We'll implement it and call `create_vec`.

macro_rules!

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        let mut vec = ::std::vec::Vec::new();
        vec.resize($count, $value);
        vec
    }};
    [$($value:expr),*] => {{
        let mut vec = ::std::vec::Vec::new();
        $(vec.push($value);)*
        vec
    }};
}
```

`macro_rules!` is also a macro, special for compiler. It means "I'm defining a macro", pretty the same as `fn` keyword.

macro_rules!

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        /* some code */
    }};
    [$($value:expr),*] => {{
        /* some code */
    }};
}
```

Our `create_vec` macro works quite like the same as matching an `enum`. We check what arguments are given and match them sequentally with patterns on the left side, called *matchers*. Then, we insert code generated on the right side, called *transcribers*.

`macro_rules!`

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        /* some code */
    }};
    [$($value:expr),*] => {{
        /* some code */
    }};
}
```

Arguments (called *meta-variables*) of macro are specified by $ symbol. After the colon, we put the type of the argument.

`macro_rules!`

Possible types of meta-variables are:

- `expr` - an expression.
- `stmt` - a statement.
- `ty` - a type.
- `ident` - an identifier (or keyword).
- `block` - block in `{}`.
- `tt` - token tree `()`, `[]` or `{}`.
- `literal` - literal.
- And much more!

macro_rules

```
macro_rules! other_create_vec {
    /* variant */
    [$($value:expr),*] => {{
        /* code */
    }};
}
```

To repeat some pattern multiple times, there is "special" syntaxes like `$()*`, `$()+` and `$()?`. They all mean "sequence of patterns inside" and called *repetitions*.

- `*` means "zero or more times".
- `+` means "one or more times".
- `?` means "zero or one time".

macro_rules

```
macro_rules! other_create_vec {
    /* variant */
    [$($value:expr),*] => {{
        /* code */
    }};
}
```

To repeat some pattern multiple times, there is “special” syntaxes like `$()*`, `$()+` and `$()?`. They all mean “sequence of patterns inside” and called *repetitions*.

- `*` means “zero or more times”.
- `+` means “one or more times”.
- `?` means “zero or one time”.

After `$()` and before repetition qualifier, you can write a separator (or don’t choose a separator). Rust allows you not to write it at the end, just like, for instance, in definition of `enum`.

macro_rules!

Just to understand it a bit better, let's write some strange looking macro.

```
macro_rules! example {
    {$($value1:expr),* => $($value2:expr),*} => {};
}

// Note that we've used '{}' on definition but
// in this line 'example' is used with []!
example![1, 2, 3 => 3, 2];
```

macro_rules!

Just to understand it a bit better, let's write some strange looking macro.

```
macro_rules! example {
    {($($value1:expr),* => $($value2:expr),*)} => {};
}

example![(1, 2, 3 => 3, 2)];
```

macro_rules!

Just to understand it a bit better, let's write an example macro.

```
macro_rules! example {
    {$(($($value1:expr),* => $($value2:expr),*)),*} => {{}};
}

example![(1, 2, 3 => 3 + 3, 2), ("hello" => [1, 2, 3], (22, 42))];
```

macro_rules

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        let mut vec = ::std::vec::Vec::new();
        vec.resize($count, $value);
        vec
    }};
    // Note this example!
    [] => { ::std::vec::Vec::new() };
    [$($value:expr),*] => {{
        let mut vec = ::std::vec::Vec::new();
        $(vec.push($value);)*
        vec
    }};
}
```

We use first {} to mark the beginning of macro block with code. The second is needed to show that we'll use multiple statements inside the block

macro_rules!

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        let mut vec = ::std::vec::Vec::new();
        /* code */
    }};
    /* variant */
}
```

We have to use a fully specified type for a `Vec` because our macro is actually placed in the place where it's used.

macro_rules!

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        let mut vec = ::std::vec::Vec::new();
        /* code */
    }};
    /* variant */
}
```

We have to use a fully specified type for a `Vec` because our macro is actually placed in the place where it's used.

- The `::` means "from list of imported crates". If `std` is not imported, we'll receive an error.

macro_rules!

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        let mut vec = ::std::vec::Vec::new();
        /* code */
    }};
    /* variant */
}
```

We have to use a fully specified type for a `Vec` because our macro is actually placed in the place where it's used.

- The `::` means "from list of imported crates". If `std` is not imported, we'll receive an error.
- We can also use `$crate` pattern: for instance, if our crate is called `example`, the name `$crate::module::Example` will change to `example::module::Example`.

macro_rules!

```
macro_rules! new_s {
    [] => { nested::S {} };
}
pub mod example {
    pub mod nested {
        pub struct S;
    }
    pub fn test() -> nested::S {
        new_s!()
    }
}

example::test();
// error: use of undeclared crate or module `nested`
// new_s!();
```

macro_rules!

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        let mut vec = ::std::vec::Vec::new();
        vec.resize($count, $value);
        vec
    }};
    [$($value:expr),*] => {{
        let mut vec = ::std::vec::Vec::new();
        $(vec.push($value);)*
        vec
    }};
}
```

Inside code, we can insert our meta-variables by writing `$META_VAR_NAME`. If we want to expand variadic pattern, we use `$()*`, `$()+` or `$()?`. This forces pattern to expand exactly zero or more times, more than zero times or not more than one time, or you'll get error.

macro_rules!

```
macro_rules! create_vec {
    [$($value:expr),*] => {{
        let mut vec = ::std::vec::Vec::new();
        $(vec.push($value);)*
        vec
    }};
}
```

You can see that this macro will expand to the series of pushes, and it's not effective, since we'll reallocate multiple times. As in many metaprogramming tools, you cannot just get the size of the `$value`, you need to calculate it.

macro_rules!

```
macro_rules! count {
    () => (0usize);
    ($head:tt $($tail:tt)*) => (1usize + count!($($tail)*));
}
```

To solve this, we'll create a macro that returns 0 when there's no arguments and `1 + count($tail)` when there's more arguments.

Note that our macro is recursive!

macro_rules!

```
macro_rules! create_vec {
    [$($value:expr),*] => {{
        let mut vec = ::std::vec::Vec::with_capacity(
            count!($($value)*)
        );
        $(vec.push($value);)*
        vec
    }};
}
```

To solve this, we'll create a macro that returns 0 when there's no arguments and `1 + count($tail)` when there's more arguments.

`macro_rules!`

Question: Do we have any limits on macro recursion?

`macro_rules!`

Question: Do we have any limits on macro recursion?

Yes. If not, user will see strange compiler segmentation faults. To extend the limit, use `recursion_limit` attribute:

macro_rules!

Question: Do we have any limits on macro recursion?

Yes. To extend the limit, use `recursion_limit` attribute:

```
#![recursion_limit = "300"]
macro_rules! count {
    () => (0usize);
    ($head:tt $($tail:tt)*) => (1usize + count!($($tail)*));
}

// This will fail without recursion_limit!
count!(0 1 2 /* ... */ 254 255)
```

The default value of `recursion_limit` attribute is 128. Please note that this attribute applies to all compile-time recursive operations, including dereference and `const` functions.

cargo-expand

Macros are quite difficult to write and debug. One of the tools that can help you with that is `expand`, installable by `cargo install cargo-expand`. If we use `cargo expand`, the following call:

```
count!(0 1 2 3 4);
```

Expands to:

```
1usize + (1usize + (1usize + (1usize + (1usize + 0usize))))
```

macro_rules!

Macro are so powerful that you can write a lot of stuff inside!

```rust
macro_rules! funny {
    [sentence in English with value $value:expr] => {{
        println!("wow, the value is {}!", $value);
    }};
}

funny![sentence in English with value 42];
```

`macro_rules!`

Of course, there's some limitations on syntax. For instance, you cannot use `|` as a separator after `expr` since Rust actually builds AST before macro expansion, and `|` is an operator, therefore compiler won't know where is separator and where is next expression.

The precise rules are not the material of the lecture.

Macros hygiene

Macros are *hygienic*. That means what is written in macro won't affect code in the call site. For instance, `#define` allows us to make the following mistake:

```
#define FIVE(value) (value * 5)
// Fails!
// assert(FIVE(2 + 3) == 25);
```

This happens because `#define` is not hygienic. In Rust:

```
macro_rules! five_times {
    ($x:expr) => (5 * $x);
}
// Works just fine!
assert_eq!(25, five_times!(2 + 3));
```

Macros hygiene

When we say Rust macros are hygienic, we mean that a declarative macro (generally) cannot affect variables that aren't explicitly passed to it.

```
macro_rules! let_foo {
    ($x:expr) => {
        let foo = $x; }
}
let foo = 1;
// expands to let foo = 2;
let_foo!(2);
// ...But instead, the compiler will even complain
// that the 'let foo' in the macro is an unused variable!
assert_eq!(foo, 1);
```

You can, most of the time, think of macro identifiers as existing in their own universe that is separate from that of the code they expand into.

Macros hygiene

This hygienic separation does not apply beyond variable identifiers. Declarative macros do share a namespace for types, modules, and functions with the call site.

Macros hygiene

This hygienic separation does not apply beyond variable identifiers. Declarative macros do share a namespace for types, modules, and functions with the call site.

This means your macro can define new functions that can be called in the invoking scope, add new implementations to a type defined elsewhere (and not passed in), introduce a new module that can then be accessed where the macro was invoked, and so on.

Macros hygiene

You can explicitly choose to share identifiers between a macro and its caller if you specifically want the macro to affect a variable in the caller's scope.

```
macro_rules! please_set {
    ($i:ident, $x:expr) => {
        $i = $x;
    }
}
let mut x = 1;
please_set!(x, x + 1);
assert_eq!(x, 2);
```

Macros visibility

Unlike pretty much everything else in Rust, declarative macros only exist in the source code after they are declared. If you try to use a macro that you define further down in the file, this will not work!

```
// Does not compile!
fn main() {
    count!(0 1 2 3 4 5);
}

macro_rules! count {
    /* macro */
}
```

Macros visibility

Macros are not visible in modules and pub keyword does not affect them. If you want to make your macro visible for users, use #[macro_export] on macro: it's pretty the same as putting macro in the root of the crate and marking it as pub

```
mod a {
    mod b {
        // Now visible to the end user
        #[macro_export]
        macro_rules! count {
            () => (0usize);
            ($x:tt $($xs:tt)*) => (1usize + count!($($xs)*));
        }
    }
}
fn main() {
    count!(0 1 2 3 4 5);
}
```

Parallel Computing

Parallel Computing

It's time to make our programs multithreaded! The best way to start is to create new threads and compute something in parallel.

To do so, we use `std::thread::spawn`.

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

Parallel Computing

Example:

```
const THREAD_NUM: usize = 8;
let result: Vec<usize> = (0..THREAD_NUM)
    .map(|_| thread::spawn(|| simulate()))
    .map(|handle| handle.join().expect("thread panicked!"))
    .collect();
```

Parallel Computing

Example:

```
const THREAD_NUM: usize = 8;
let result: Vec<usize> = (0..THREAD_NUM)
    .map(|_| thread::spawn(|| simulate()))
    .map(|handle| handle.join().expect("thread panicked!"))
    .collect();
```

Question: Why do we need this expect?

Parallel Computing

Example:

```
const THREAD_NUM: usize = 8;
let result: Vec<usize> = (0..THREAD_NUM)
    .map(|_| thread::spawn(|| simulate()))
    .map(|handle| handle.join().expect("thread panicked!"))
    .collect();
```

Question: Why do we need this `expect`?

Thread can panic while executing. This panic should stop in the source thread. When joining, our `JoinHandle` will give us a `Result` with either a final value or an error with a panic value.

’static in thread::spawn

Consider the following code:

```
fn example() {
    let vec = vec![1, 2, 3];
    let handle = thread::spawn(|| {
        for i in vec.iter() {
            println!("{i}");
        }
    });
    handle.join();
}
```

'static in thread::spawn

```
error[E0373]: closure may outlive the current function, but it
              borrows `vec`, which is owned by the current function
 --> src/main.rs:5:31
  |
5 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `vec`
6 |         for i in vec.iter() {
  |                  --- `vec` is borrowed here
  |
note: function requires argument type to outlive `'static`
 ...
help: to force the closure to take ownership of `vec` (and any
      other referenced variables), use the `move` keyword
  |
5 |     let handle = thread::spawn(move || {
  |                                ++++
```

'static in thread::spawn

- Rust knows nothing about a `join`.
- Moreover, even if it will know, it cannot guarrantie that we won't panic until `join`.
- And the most ridiculous: nothings stops us from *leaking* a `JoinHandle`!

'static in thread::spawn

So, we need to make closure 'static to outlive any possible variable in the program. We'll use move here as the compiler suggests.

```
fn example() {
    let vec = vec![1, 2, 3];
    let guard = thread::spawn(move || {
        for i in vec.iter() {
            println!("{i}");
        }
    });
    guard.join();
}
```

The same answer applies to the question about T having 'static lifetime.

’static in thread::spawn

One more program:

```
fn count_foo_bar(data: &str) -> usize {
    let t1 = thread::spawn(|| data.matches("foo").count());
    let t2 = thread::spawn(|| data.matches("bar").count());
    t1.join().unwrap() + t2.join().unwrap()
}
```

'static in thread::spawn

One more program:

```
fn count_foo_bar(data: &str) -> usize {
    let t1 = thread::spawn(|| data.matches("foo").count());
    let t2 = thread::spawn(|| data.matches("bar").count());
    t1.join().unwrap() + t2.join().unwrap()
}
```

```
error[E0621]: explicit lifetime required in the type of
              `data` --> src/lib.rs:4:14
  |
4 | let t1 = thread::spawn(|| data.matches("foo").count());
  |          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
             lifetime `' static` required
```

'static in thread::spawn

```
fn count_foo_bar(data: Rc<str>) -> usize {
    let data_2 = data.clone();
    let t1 = thread::spawn(move || {
        data.matches("foo").count()
    });
    let t2 = thread::spawn(move || {
        data_2.matches("bar").count()
    });
    t1.join().unwrap() + t2.join().unwrap()
}
```

’static in thread::spawn

```
error[E0277]: `Rc<str>` cannot be sent between threads safely
   --> src/main.rs:7:18
    |
7   |           let t1 = thread::spawn(move || {
    |  __________________^^^^^^^^^^^^^_-
    | |                  |
    | |                  `Rc<str>` cannot be sent between
    | |                  threads safely
8   | |             data.matches("foo").count()
9   | |         });
    | |_________- within this `[closure@src/main.rs:7:32: 9:10]`
    |
    = help: within `[closure@src/main.rs:7:32: 9:10]`, the
            trait `Send` is not implemented for `Rc<str>`
```

’static in `thread::spawn`

Question: Can you guess what means `Send`?

`'static` in `thread::spawn`

Question: Can you guess what means `Send`?

If we could send `Rc` between threads, it will be possible to have 2 threads simultaneously modifying the underlying non-atomic counter!

`’static` in `thread::spawn`

Question: Can you guess what means `Send`?

If we could send `Rc` between threads, it will be possible to have 2 threads simultaneously modifying the underlying non-atomic counter!

Or, simply speaking, we’ll run into a **data race**.

`Send` and `Sync` traits

Firstly, let's remember what is a data race.

`Send` and `Sync` traits

Firstly, let's remember what is a data race.

The simple one which doesn't involve memory models is defined as follows:

- Two or more threads concurrently accessing a location of memory.
- One or more of them is a write.
- One or more of them is unsynchronized.

`Send` and `Sync` traits

When Rust was on it's early stages, people believed memory safety and data race safety were totally different things. *But actually, the first implies the second!*

Data races are mostly prevented through Rust's ownership system: it's impossible to alias a mutable reference, so it's impossible to perform a data race.

Interior mutability makes this more complicated, which is largely why we have the `Send` and `Sync` traits.

`Send` and `Sync` traits

`Send` and `Sync` are **unsafe** marker traits with the following meaning:

- 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.

`Send` and `Sync` traits

`Send` and `Sync` are **unsafe** marker traits with the following meaning:

- 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`)

`Send` and `Sync` traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32`
- `Vec<i32>`
- `&str`
- `Rc<T>`
- `Cell<T>`
- `MutexGuard<'static, ()>`
- `*mut T`

`Send` and `Sync` traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>`
- `&str`
- `Rc<T>`
- `Cell<T>`
- `MutexGuard<'static, ()>`
- `*mut T`

`Send` and `Sync` traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>` - `Send` and `Sync`.
- `&str`
- `Rc<T>`
- `Cell<T>`
- `MutexGuard<'static, ()>`
- `*mut T`

Send and Sync traits

We need to complete a little quiz to understand what's happening. What's the types are Sync and Send?

- i32 - Send and Sync.
- Vec<i32> - Send and Sync.
- &str - Send and Sync.
- Rc<T>
- Cell<T>
- MutexGuard<'static, ()>
- *mut T

`Send` and `Sync` traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>` - `Send` and `Sync`.
- `&str` - `Send` and `Sync`.
- `Rc<T>` - not `Send` nor `Sync`.
- `Cell<T>`
- `MutexGuard<'static, ()>`
- `*mut T`

`Send` and `Sync` traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>` - `Send` and `Sync`.
- `&str` - `Send` and `Sync`.
- `Rc<T>` - not `Send` nor `Sync`.
- `Cell<T>` - only `Send`.
- `MutexGuard<'static, ()>`
- `*mut T`

`Send` and `Sync` traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>` - `Send` and `Sync`.
- `&str` - `Send` and `Sync`.
- `Rc<T>` - not `Send` nor `Sync`.
- `Cell<T>` - only `Send`.
- `MutexGuard<'static, ()>` - only `Sync`.
- `*mut T`

`Send` and `Sync` traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>` - `Send` and `Sync`.
- `&str` - `Send` and `Sync`.
- `Rc<T>` - not `Send` nor `Sync`.
- `Cell<T>` - only `Send`.
- `MutexGuard<'static, ()>` - only `Sync`.
- `*mut T` - not `Send` nor `Sync`.

`Send` and `Sync` traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>` - `Send` and `Sync`.
- `&str` - `Send` and `Sync`.
- `Rc<T>` - not `Send` nor `Sync`.
- `Cell<T>` - only `Send`.
- `MutexGuard<'static, ()>` - only `Sync`.
- `*mut T` - not `Send` nor `Sync`.

Most types are `Send` and `Sync`, but there's some exceptions.

`Send` and `Sync` traits

`Send`/`Sync` are also `auto` traits: they are implemented automatically for a type if all of its generics are `Send`/`Sync`.

Their final definition is:

```rust
pub unsafe auto trait Send {}
pub unsafe auto trait Sync {}
```

Send and Sync traits

In case you want to unimplement Send and Sync on the stable compiler, you can use PhantomData.

```
type DisableSend = PhantomData<MutexGuard<'static, ()>>;
type DisableSync = PhantomData<Cell<()>>;

struct Test {
    disable_send: DisableSend,
    disable_sync: DisableSync,
}
```

Send and Sync traits

To solve this problem, we need `Arc` - *atomic* reference counting pointer.

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

fn count_foo_bar(data: Arc<str>) -> usize {
    let data_2 = data.clone();
    let t1 = thread::spawn(move || {
        data.matches("foo").count()
    });
    let t2 = thread::spawn(move || {
        data_2.matches("bar").count()
    });
    t1.join().unwrap() + t2.join().unwrap()
}
```

Poisoning

Imagine our thread to panic while holding a `MutexGuard`. It means some data was partly modified when panic occured! It's not the thing that is actually violates memory safety, but we can break invariants without even noticing!

Poisoning

Imagine our thread to panic while holding a `MutexGuard`. It means some data was partly modified when panic occured! It's not the thing that is actually violates memory safety, but we can break invariants without even noticing!

When locking `Mutex`, you're given `LockResult<MutexGuard<'_, T»` which gives you a lock or `PoisonError`, meaning the mutex is poisoned, i.e thread that was holding a lock panicked!

The same applies to the thread and mpsc-queue.

Questions?