PROGRAMS, TECH

Rust global variables demystified

NOVEMBER 1, 2021 | HRVOJE | LEAVE A COMMENT

Rust has a reputation of a language unfriendly to global variables. While this reputation is not entirely undeserved, most of it stems from guarantees afforded by Rust and not by a desire to stifle the programmer's creativity. In this article we'll show how to use global variables, and how to overcome the limitations.

Contents [hide]

Is it even ok to use global variables?

Global variables are a controversial topic in programming circles, with many educators taking the moral high ground and condemning them as code smell, a shortcut, a crutch, hallmark of throw-away code, and <insert favorite insult>. While there is good reason for the hostility, there is also an abundance of situations where global variables are either appropriate or actually the only way to proceed. For example, pre-compiled regular expressions or state of a logger are examples where you'd probably want to use globals in preference to sending the state all the way from top-level to the bottom-most part of your program. Many low-level system APIs, such as those of signal handlers or hardware-triggered interrupts, have callbacks that don't receive a "state" argument, so communication between the callback and the rest of the system must go through globals. There are other examples, and in this article we'll assume that you have good reasons for using globals that your CS professor would approve of. If that's not the case, don't worry, we won't tell.

Declaring globals

A Rust global variable is declared much like any other variable, except it's declared at top-level and uses `static` instead of `let`:

```
static LOG_LEVEL: u8 = 0;
```

So we use the keyword `static` instead of `let`, and must spell out the type, as the compiler refuses to infer it. That means that you must spell out the type even when it's unambiguous – `static LOG_LEVEL = 0u8` won't compile.

Note that what we've declared is actually a *static*, which is not necessarily *global*. If the same declaration appeared in a function, the variable would be visible only inside the function, but its value would still be shared among all invocations of the function and last until the end of the program. This article talks about global variables, but most of the content equally applies to static variables defined inside functions, the only difference being that of visibility.

The global works much like you'd expect: you can read it from any function in the module, you can import it (if declared `pub`) from another module and then read it from there. You can even borrow it and get a reference with a `'static` lifetime – because the global variable lasts until the end of the program. Neat.

What you can't do is *assign* to the global because it's not declared as `mut`. This is where things get interesting.

Mutable globals – atomics and locks

Ideally we'd like to declare our global as `mut` and have a public API that manipulates it – say, a function that reads it and another that writes it:

```
static mut LOG_LEVEL: u8 = 0;

pub fn get_log_level() -> u8 {
    LOG_LEVEL
}

pub fn set_log_level(level: u8) {
    LOG_LEVEL = level;
}
```

The compiler rejects both the getter and the setter with a similar error message:

```
error[E0133]: use of mutable static is unsafe and requires unsafe function or block
 --> src/lib.rs:8:5
  |
8 |     LOG_LEVEL = level;
  |     ^^^^^^^^^^^^^^^^^ use of mutable static
```

The underlying problem is that a global variable is potentially visible from multiple threads. The above functions don't *synchronize* their access to the global, so there's nothing to prevent `set_log_level()` from being called in one thread while another thread calls `get_log_level()` or `set_log_level()`, either of which would constitute a data race. Rust requires an `unsafe` block to signal that such synchronization has been implemented by the code that surrounds access to the mutable static. Alternatively, the whole function can be `unsafe` to signal that the burden of synchronization is transferred to its callers. Since we don't in fact have such synchronization (and it's unclear how a caller of `set_log_level()` and `get_log_level()` would even ensure it), we won't attempt to "fix" it by adding an `unsafe` to get the code to compile. We want to access globals without advanced reasoning about unsafe and undefined behavior.

Since we're dealing with a potential data race, let's address it with the mechanisms used elsewhere in Rust to avoid data races – locks and atomics. In case of a `u8`, we can simply replace `u8` with `AtomicU8`:

```
use std::sync::atomic::{AtomicU8, Ordering};

static LOG_LEVEL: AtomicU8 = AtomicU8::new(0);

pub fn get_log_level() -> u8 {
    LOG_LEVEL.load(Ordering::Relaxed)
}

pub fn set_log_level(level: u8) {
    LOG_LEVEL.store(level, Ordering::Relaxed);
}
```

The global variable is no longer `mut`, so no unsafe is needed. The code is thread-safe and as performant as an unsafe version would be – on x86 the relaxed atomic load compiles into an ordinary load. If you need stricter ordering guarantees between `LOG_LEVEL` and other data in the program, you can use `Ordering::SeqCst` instead.

But what if we need something that won't neatly fit into an atomic – say, a string? This compiles, but won't allow us to modify the global:

```
static LOG_FILE: String = String::new();

pub fn get_log_file() -> &'static str {
    &LOG_FILE
}
```

Since there is no `AtomicString`, we need to use a proper lock:

```
use std::sync::Mutex;

// XXX - doesn't compile
```

```
static LOG_FILE: Mutex<String> = Mutex::new(String::new());

pub fn get_log_file() -> String {
    LOG_FILE.lock().unwrap().clone()
}

pub fn set_log_file(file: String) {
    *LOG_FILE.lock().unwrap() = file;
}
```

Note that `get_log_file()` must return a fresh copy of the string. Returning a reference would require a lifetime, and there is no lifetime to associate with the global variable other than `'static`, and `'static` is incorrect (and wouldn't compile) because `set_log_file()` can modify it at any time.

The above doesn't compile for a different reason:

```
error[E0015]: calls in statics are limited to constant functions, tuple structs and
tuple variants
 --> src/lib.rs:3:34
  |
3 | static LOG_FILE: Mutex<String> = Mutex::new(String::new());
  |                                  ^^^^^^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0015`.
```

What's going on here? Why did `String::new()` compile, but `Mutex::new(String::new())` didn't?

The difference is that the globals we declared so far were just pieces of data whose initial values were available at compilation time. The compiler didn't need to generate any initialization code for `static LOG_LEVEL: u8 = 0` – it only reserved a byte in the executable's data segment and ensured that it contained 0 *at compile time*. `String::new()` also works because it is a `const fn`, function specifically marked as runnable at compile time. It can be marked like that because an empty string doesn't allocate, so the string returned by `String::new()` can be represented in the executable by a triple of `(0 [length], 0 [capacity], NonNull::dangling() [constant representing unallocated pointer])`. Nothing to do at run time. On the other hand, `static LOG_FILE: String = String::from("foo")` wouldn't compile because `String::from()` requires a run-time allocation and is therefore not a `const fn`.

`std::sync::Mutex::new()` is not `const fn` because it requires an allocation in order to keep the system mutex at a fixed address. And even if we used an allocation-free mutex (such as `parking_lot::Mutex` which supports a `const fn` constructor on nightly Rust), we'd face the same issue if we wanted to start off with a non-empty string, a data structure coming from a library we don't control, or information only available at run-time, such as fresh randomness or the current time. In general, we don't want to be constrained to `const fn` functions when initializing global variables.

As a side note, C++ supports initializing global variables with arbitrary code by simply allowing the compiler to generate code that runs before `main()` (or during `dlopen()` in the case of dynamic libraries). This approach is convenient for simple values, but when used in real programs it led to issues with initialization order, aptly named static initialization order fiasco. To avoid that problem, as well as issues with libraries that require explicit initialization, Rust doesn't allow pre-main initialization, opting instead for the approach C++ calls the *construct on first use* idiom.

We will review three ways to initialize a global variable with arbitrary data, two of them based on external (but extremely well reviewed and widely used) crates, and one based on the standard library.

Once cell

The `once_cell` crate provides a `OnceCell` type that can be used to define global variables. Here is how one would use `OnceCell` for `LOG_FILE`:

```
use once_cell::sync::OnceCell;
use std::sync::Mutex;

static LOG_FILE: OnceCell<Mutex<String>> = OnceCell::new();

fn ensure_log_file() -> &'static Mutex<String> {
    LOG_FILE.get_or_init(|| Mutex::new(String::new()))
}

pub fn get_log_file() -> String {
    ensure_log_file().lock().unwrap().clone()
}

pub fn set_log_file(file: String) {
    *ensure_log_file().lock().unwrap() = file;
}
```

Looking at the implementation of `get_log_file()` and `set_log_file()`, it is immediately apparent that they implement the "construct on first use" idiom – both functions call a method that ensures that the inner value is initialized (and that this is only done once), and retrieve a reference to the globally-stored value. This value can then be manipulated in the usual way through interior mutability.

`OnceCell<T>` is conceptually similar to a `RefCell<Option<T>>`. Like an `Option`, it has two states, one empty and another with a useful value. Like `RefCell<Option<T>>`, it uses interior mutability to allow setting the inner value using just a shared reference. But unlike `Option`, once set to a non-empty value, the stored value can never be set again. This allows a non-empty `OnceCell` to give out shared references to the inner data, which `RefCell<Option>` wouldn't be allowed to do (it could at best return a `Ref<T>`) because the contents may change at any time. `OnceCell` is also thread-safe, so it would actually compare to `Mutex<Option<T>>`, except it uses an atomic to efficiently check whether the cell has been set.

`once_cell` also provides a `Lazy` type that makes the above even simpler, removing the need for a separate `ensure_log_file()` function:

```
use std::sync::Mutex;
use once_cell::sync::Lazy;

static LOG_FILE: Lazy<Mutex<String>> = Lazy::new(|| Mutex::new(String::new()));

pub fn get_log_file() -> String {
    LOG_FILE.lock().unwrap().clone()
}

pub fn set_log_file(file: String) {
    *LOG_FILE.lock().unwrap() = file;
}
```

Our globals now work pretty much the way we wanted them. `Lazy<T>` performs a sleight of hand that even allows us to directly call methods like `lock()` directly on `LOG_FILE`. It achieves this by implementing `Deref`, the trait normally used to treat lightweight containers (typically smart pointers) like the values they contain. `Deref` is the mechanism that allows you to call methods of `Foo` on `Box<Foo>`, or methods of `&str` on `String`, and so on. `Lazy<T>` wraps a `OnceCell<T>` and implements a `Deref<T>` that returns `self.once_cell.get_or_init(|| self.init.take().unwrap())`, where `init` is the closure passed to `Lazy::new()`.

The `Lazy` version still uses the construct on first use idiom, it's just hidden behind the magic of `Deref`. In some cases this can yield surprising results because the actual type of `LOG_FILE` is *not* `Mutex<String>`, it's `Lazy<Mutex<String>>`, so if you use it a context that expects exactly `&Mutex<String>`, it will fail to compile. It's not a big deal because you can always obtain the actual `&Mutex<String>` with `&*LOG_FILE` (equivalent to `LOG_FILE.deref()`), but it is something to be aware of.

The `OnceCell` and `Lazy` types are in the process of getting stabilized, so we can expect them to become part of the standard library in the near future.

Lazy static

Another popular library for creating global variables is the `lazy_static` crate, which defines a macro that hides even the lazy initialization, allowing you to write code that looks almost like an ordinary declaration:

```
use lazy_static::lazy_static;
use std::sync::Mutex;

lazy_static! {
    static ref LOG_FILE: Mutex<String> = Mutex::new(String::new());
}
```

```
// get_log_file() and set_log_file() defined as with once_cell::Lazy

pub fn get_log_file() -> String {
    LOG_FILE.lock().unwrap().clone()
}

pub fn set_log_file(file: String) {
    *LOG_FILE.lock().unwrap() = file;
}
```

An invocation of `lazy_static!` is just syntax sugar for defining a `Lazy` value. Under the hood everything works exactly the same as in the example that used `once_cell::Lazy` (except `lazy_static` defines its own `lazy_static::Lazy`). Like with `once_cell::Lazy`, the actual type of `LOG_FILE` is not `Mutex<String>`, but a different type which uses `Deref` to give out `&'static Mutex<String>` on method calls. Some details differ, e.g. `lazy_static` constructs a dummy type also named `LOG_FILE` and implements `Deref` on that, while hiding the actual `Lazy<T>` value in a static variable defined in a function – but the end result is exactly the same.

If you're curious, you can run `cargo expand` on code generated by `lazy_static! {...}` to learn exactly what it does.

Standard library – Once+unsafe

Until `OnceCell` stabilizes, the standard library doesn't offer a way to implement the global variables initialized with non-const functions without unsafe code. In most cases this should be avoided because you can use use `once_cell` or `lazy_static`. But if you must only depend on the standard library, if you want tighter control, or if you just want to learn how it's done, here is an example that uses `std::sync::Once` to implement a mutable global:

```
use std::mem::MaybeUninit;
use std::sync::{Mutex, Once};

fn ensure_log_file() -> &'static Mutex<String> {
    static mut LOG_FILE: MaybeUninit<Mutex<String>> = MaybeUninit::uninit();
    static LOG_FILE_ONCE: Once = Once::new();

    // Safety: initializing the variable is only done once, and reading is
    // possible only after initialization.
    unsafe {
        LOG_FILE_ONCE.call_once(|| {
            LOG_FILE.write(Mutex::new(String::new()));
        });
        // We've initialized it at this point, so it's safe to return the reference.
        LOG_FILE.assume_init_ref()
    }
}
```

```
// get_log_file() and set_log_file() defined as with once_cell::OnceCell

pub fn get_log_file() -> String {
    ensure_log_file().lock().unwrap().clone()
}

pub fn set_log_file(file: String) {
    *ensure_log_file().lock().unwrap() = file;
}
```

`Once` ensures that `MaybeUninit` is initialized only once, which `OnceCell` guaranteed in previous versions of the code. `Once` is also efficient, using atomics to optimize the fast path when the variable has already been initialized. The definition of the variable, now static rather than global, is placed inside the function to prevent code outside `ensure_log_file()` from accessing it directly. All accesses inside `ensure_log_file()` are synchronized through `call_once()`, writer by running inside it, and readers by waiting for it to complete, which makes the access data-race-free.

Once the initialization of `LOG_FILE` is complete, `ensure_log_file()` can proceed to return the reference to the inside of the `MaybeUninit`, using `assume_init_ref()`.

Which option to choose

In most cases involving real-world types, you'll want to use `once_cell` or `lazy_static`, depending on which syntax you prefer. They have the exact same run-time characteristics, and you won't go wrong with choosing either. Of course, when `once_cell` stabilizes in the stdlib, that will become the obvious choice.

There are two exceptions:

1. When your globals are const-initialized and you don't need to modify them, you can just declare them as `static` or `const`. The difference between the two is that `static` guarantees that they are stored in only one place and `const` doesn't (it inlines them where they're used).

2. When you need to modify your globals, but their type is supported by `std::sync::atomic`: `bool`, `u8-u64`, `i8-i64`, `usize`, or `isize`. In that case you can declare the variable as `static` with the appropriate atomic type, and use the atomic API to read and modify it.

A case not covered by this article are thread-local variables, which can also be global. Those are provided by the `thread_local` macro from the standard library, and allow the use of non-`Sync` types in globals.