In this workshop we'll look at rust's ownership system, understand why it has the restrictions it has and how to live with them.
This workshop focuses mostly on a practical side of things, on how to actually use rust primitives. For a detailed examination of internal implementation I would recommend Crust of Rust videos.

Classic diagram of program memory with different regions.

- We have static data, things here are soldered in binary itself, they exist from the start of the program till the end.
- We have a stack, each time we call a function in our program we get a new stack frame which includes all incoming parameters, local variables and internal service data. Stack frame for each function has a constant size which should be known at compile-time.
- And for dynamically sized data we have a heap;

In reality things can be more complicated, for example we can have TLS, or memory allocated via mmap syscall, in multithreaded programs there will be multiple stacks.

Static
Rust provides 2 distinct but very similar keywords: const and static.
Const is not a real variable. Meaning it is not a specific named memory location. It is an alias for a value. Each usage or reference of this name is substituted by the compiler to that value.
To have a "real" static variable with a single memory location we have a static keyword.
Rust is not particularly happy with mutable global statics, so every operation is unsafe. Forces us not to use global mutable things.
Consts do not have a single memory location, just fancy way to alias values with names. Static is for real static variables. Immutable static are perfectly fine, mutable are unsafe.

Stack
The important bit here is that move operation is not free. C/C++ people understand that really well, they have this concept of passing parameters by value and by reference. If we pass by value each time we fully copy the object to the next stack frame.
With reference we copy not an entire object, but only an indirect reference (which is a pointer under-the-hood) to it.

Rust memory model allows to have multiple immutable references to an object, or a single mutable. In terms of ownership this means that only one entity can own and mutate the object. Either shared ownership with immutable access or exclusive with mutable.

Heap

What if we want to have objects whose size is determined only in runtime, or if we want to have objects whose lifetime is not tied to a scope where they are created? Then we need to use a heap. In C/C++ languages we had to manually ask to return memory to the heap, this changed a little bit in C++. Rust from the start enforces smart-pointer RAII approach to heap memory allocations. We have 2 types of objects to get heap memory: Box and Rc (short for reference counter).

Box

Uses RAII pattern which means it ties lifetime of heap memory to lifetime of a Box object. While Box object exists memory is allocated, whenever Box object is dropped, memory it references is deallocated. clone() method of Box object allocates new memory, copies the referenced content there and returns a new, completely independent Box.
Box is very similar to immutable reference. Only one Box per one memory allocation, thus exclusive ownership and ability to mutate and do whatever we want with an object inside. It has implemented Deref/DerefMut traits which allow to access referenced object fields or methods inside the box directly via dot operator.

Reference Counter

And also Atomic Reference Counter. Here we get the same smart-raii pointer to a heap allocated object. But this time whenever we clone it we get a copy of the object pointing to the same memory location. Inside of it there is a reference counter which tracks how many object pointers exist there. When the last smart pointer is dropped, the counter goes to zero and we deallocate memory. Almost the same as shared_ptr in C++.
It represents shared ownership. And as with shared ownership all owners have only immutable access. Same as with immutable references.

Difference between Rc and Arc.

Rc uses a plain usize counter, thus it is quick but not thread-safe. Arc uses atomic as a reference counter, hence it is slightly slower but thread-safe (meaning it can be shared and used between multiple threads).

To sum up. Things can be allocated in different sections: static data, stack, heap. We know that with exclusive access (by owning the object, by mutable reference or by Box) we can mutate object. We can share object (by immutable references or by Rc/Arc), in this case we have shared ownership and can only access object immutably.

Multithreaded applications
What if we want to share an object and still mutate it?
Imagine we have a shared object between 2 threads. If we allowed mutating access to it, then we would be in a complete mess. 2 threads writing data to the same memory location. Terrible. That’s one of the reasons why we distinguish between exclusive “full” access and shared immutable access.
Rust provides primitives to synchronize access and allow mutations. In multithreaded applications it is done by using Mutexes and Atomics primitives.

Mutexes
Mutex wraps the value inside. We can lock Mutex to get MutexGuard object, which uses the RAII pattern to ensure free of mutex lock after it goes out of scope. Other threads which try to access data need to lock as well. And if mutex is already taken, then they are blocked on their lock() call. With this additional limitations it becomes more safe to mutate object (still possibility of deadlocks). So after lock we get an exclusive access to object, can mutate it, until we exit the scope and release the mutex. Here we rely on OS level primitives.

Atomics
Atomics use a different approach for synchronization between threads. Instead of OS syscalls we rely on behaviour of specific instructions in our CPUs instruction sets. Most CPUs can guarantee that access to memory of a specific data type can be performed “atomically”. Meaning no interference between different read and write to the same variable.

Unfortunately atomics exist only for primitive types. But we can use primitive types such as pointers to atomically mutate complex objects.

Difference in synchronization approach in mutexes and atomics: in mutexes we get the lock to ensure exclusive access and while we have a lock we mutate things inside of it. In atomics we get the copy of an object, mutate it as we like and then try to apply the changes.

All atomic operations require explicitly state memory ordering. Good explanation is provided both by rust docs and by C++ docs (because rust inherit memory barriers model from C++)
We have:
Relaxed - means no synchronization between threads at all. The only thing that is guaranteed is that atomic operations themselves are “atomic”, no partial reads or writes to atomic variables. Useful for simple flags, or for metrics.
Acquire for load and Release for store. In this case we establish “happened before” semantics to have some order between atomic operations and surrounding code.
SeqCon - the most strict, guarantees that one can put instructions in an order as if they were executed by a single threaded machine.

Single-threaded application.
So imagine we have 2 tasks trying to mutate the same shared object. Again immutable references and Rc smart pointers forbid mutation. How to workaround this?
We could have used the same primitives from a multithreaded environment like mutexes or atomics. It would have worked. But it would be extremely inefficient to make blocking syscalls on a single thread (also remember the possibility of deadlocks), or suffer from atomic operations overhead.

Rust introduces the notion of interior mutability. What this effectively means is that it provides “synchronization” wrappers: Cell and RefCell. Basically they operate on the same logic as their multithreaded counterparts. Cell allows you to get a copy of what is in Cell or put something there. After you get the copy - mutate it as you like.
RefCell operates very similarly to mutex. You can try to “lock” the value inside the RefCell and get a RefMut or Ref RAII object by calling borrow_mut or borrow. If no one else has a mutable reference to insides of RefCell we are fine, we get the exclusive access and mutate things inside. If someone already has exclusive access then we panic which makes sense: in a multithreaded world we can block and wait until mutex lock is released, but here we have only a single thread, no one to wait, and someone is already mutating stuff, so the only thing we can do is panic instead of blocking (or just return an error if we use try_borrow() family).