Interior mutability

Interior mutability is the property for which if you have shared references to a wrapper type (eg. &Cell<T>) you can still mutate the value contained in the wrapper (T). It's useful when you need to introduce mutability inside of something immutable or when you need a mutable part of a data structure, but still logically present the structure as immutable. In other words, we can have an immutable value or multiple immutable references to a value, but still mutate its content. Mutation is performed in **controlled** and **safe** ways, depending on the wrapper type.

		Provides	Accessors	Panics	Send	Sync	Safety Notes

`Cell<T>`	Interior mutability for Copy types via **copies**. Setting a value means putting inside the Cell a copy of the value to set. Getting a value means obtaining a copy of the wrapped value. You can never obtain a pointer to the value inside the Cell.	Values (copies)	.get()
			.set()
			to get/set a copy
				Never	(if T is Send)		1) No references to the inner value can be obtained. There's no risk to mutate the value while someone is holding a pointer to the inner value. 2) Cell is not Sync (no &Cell can be shared between threads) because getting/setting the value is not synchronized. 3) Cell is Send if T is Send: if T is Send there's no problem in moving the Cell and using it at *different* times. If T is not Send and Cell was Send nonetheless, T could end up being used in different threads, invalidating the Send safety limit imposed on it.

`RefCell<T>`	Interior mutability for all types via **references** to the inner value. RefCell allows you to borrow the wrapped value either mutably or immutably. *Dynamic run-time borrowing* ensures no more than one mutable borrow or mixed borrows occur at the same time (mixed = both *&* and *&mut* at the same time).
	When the inner value is borrowed, a `Ref` or `RefMut` is returned, which can be used as a reference to the inner value. When this object is dropped, the internal borrowing bookeeping is reverted accordingly. These syntehtic references point to the original RefCell, so the RefCell cannot be moved/dropped until all these refs are dropped.
		References (&/&mut)	.borrow()
			.borrow_mut()
			to get the Ref/RefMut
			.deref()
			.deref_mut()
			on the Ref/RefMut
				Mixed borrows or more than one mutable borrow	(if T is Send)		1) Returned references (Ref/RefMut) are checked via dynamic borrowing. There is no way we can obtain more than one exclusive ref or mixed refs to the inner value *at the same moment*. 2) RefCell is not Sync (no &RefCell can be shared between threads) because updates of the internal borrowing state are not synchronized. 3) RefCell is Send if T is Send: if T is Send there's no problem in moving the RefCell and using it at *different times*. If T is not Send and RefCell was Send nonetheless, T could end up being used in different threads, invalidating the Send safety limit imposed on them.

`Mutex<T>`	A mutual exclusion primitive useful to protect data shared across threads. The Mutex provides interior mutability via **references** in a thread safe way, since the access to the inner value is properly synchronized. We can compare Mutex to RefCell because both provide a similar *dynamic run-time borrowing,* but Mutex blocks the thread waiting for the lock instead of panicking.
	The internal data can be accessed via the *lock* method, which returns a `MutexGuard`. This guard can be treated like a pointer to the inner value. Holding a guard is a proof that the inner data is being accessed only by the (unique) holder of the guard. When the guard is dropped, other *lock()* calls can access the inner value. MutexGuards has a lifetime >= of the original Mutex, so the mutex cannot be moved/dropped until all guards are dropped.
		References (&/&mut)	.lock()
			to get the MutexGuard
			.deref()
			.deref_mut()
			on the MutexGuard
				Never, blocks until the lock is freed	(if T is Send)	(if T is Send)	1) Returned guards are checked via dynamic borrowing. There is no way we can obtain more than one guard *at the same moment*. 2) Mutex is Send + Sync only if the internal T is Send: if T is Send there's no problem in moving the Mutex and using it at different times (because T is itself Send and can be used in different threads at different times safely). If T is not Send, T could end up being used in different threads, invalidating the Send safety limit imposed on them. 3) Sync on T is not influent: the data is accessed from one thread at a time in any case.

Shared Ownership

Shared ownership in Rust allows a value to "simulate" to be owned by multiple variables bindings. First, having a shared ownership of a value could simplify the implementation of several data structures and algorithms (think of a graph structure). Second, shared ownership helps to extend the lifetime of values until needed. As an example, when it's needed to pass a &T to another thread, the T value could be dropped before the other thread ends using the reference &T. To overcome this issue, the value T could be owned in a shared way (in some smart pointer), with each owner sent to a different thread. As a result, the value will continue to leave until all threads drop those pointers. These smart pointers enforce memory safety by only giving out shared references to the value they wrap, and these as usual don't allow direct mutation.

		Provides	Accessors	Panics	Send	Sync	Safety Notes
`Rc<T>`	Smart pointer that provides **single-threaded shared ownership** via reference counting. Rc<T> provides shared ownership of a value of type T, allocated in the heap when included in the smart pointer. Rc can be cloned to produce a new pointer to the very same allocation. When the last Rc pointer to a given value is dropped, the inner value is also dropped. Shared references in Rust disallow mutation by default, and Rc is no exception: you cannot generally obtain a mutable reference to something inside an Rc. If you need mutability, put a Cell or RefCell inside the Rc.	References (& only)	.deref() to get a &ref	Never			1) Only shared/immutable refs can be obtained and so there is no risk of mutation while aliasing the inner value. 2) Not Send: the inner reference count is not synchronized/updated atomicly. If Rc was Send, cloned Rc sent to other threads (pointing to the same data) would mess up the internal bookeeping (during further cloning). 3) Not Sync for the very same reason. Immutable refs to Rc (&Rc) could be used to obtain clones, which could lead to desynchronized mutation of the internal data.
`Arc<T>`	Smart pointer that provides **multi-threaded shared ownership** via reference counting, with atomic updates. Arc<T>is the thread safe equivalent of Rc<T>. Arc can be cloned to produce a new pointer to the same allocation in the heap. When the last Arc pointer to a given allocation is destroyed, the inner value is also dropped. Similarly to Rc, you cannot obtain a mutable reference to something inside an Arc. Arc<T> makes it thread safe to have multiple ownership of the same data, but it doesn't add thread safety to the data itself. Arc<T> is thread safe as long as the inner value is thread safe (see safety notes).	References (& only)	.deref() to get a &ref	Never	(if T is Send + Sync)	(if T is Send + Sync)	1) Only shared/immutable refs can be obtained and so there is no risk of mutation while aliasing the inner value. 2) Send and Sync if T is Send and Sync: if T satisfies these requisites there is no problem in using the T value in different threads at the same time. 2a) If T is not Send and Arc was Send nonetheless, T could end up being used in different threads (since Arc can be cloned and sent to other threads), invalidating the Send safety limit imposed on the T type. In this case Arc clearly cannot be Send nor Sync. 2b) If T is not Sync T cannot be used in different threads at the same moment. If Arc was always Send or Sync, Arc clones could lead to usage of T in different threads. This is not safe since we would violate the Sync limit on T.

Thread Safety

Rust enforces thread safety via marker traits: Send and Sync. These traits are defined as markers because they don't have methods or associated items. Instead they are like "flags" to signal to the compiler some properties about the implementors. Send and Sync are automatically implemented when the compiler determines that it's appropriate OR they can manually be implemented using unsafe. Talking about the automatic implementation of these traits, if a type is composed entirely of Send or Sync types, then it is Send or Sync. Almost all primitive types are Send and Sync, with some important exceptions.

Send	The Send marker trait indicates that ownership of values of types implementing Send can be transferred between threads. The vast majority of Rust's type are Send, with some exceptions. In other words a type is Send if it is safe to send it to another thread, which means more threads can use the value at *different times.* Note that Send types still follow the usual borrowing rules, e.g. if some references to a value exist, the value itself cannot be dropped or moved (e.g. to another thread).	You can think of Send as "Exclusive access is thread-safe".
Sync	The Sync marker trait indicates that it is safe for types implementing Sync to be referenced from multiple threads. In other words, any type T is Sync if &T (an immutable reference) is Send, meaning the reference can be moved to another thread. Sharing shared references (&T) safely means that the references can be used without any problem from multiple threads. The actual meaning of "can be used without any problem" depends on every single case. For example &u32 is safe to be shared between thread boundaries, but that's not true for RefCell. If we have multiple &RefCell across threads, they could lead to desynchronized borrow.	You can think of Sync as "Shared access is thread-safe".