Wrapper Types in Rust: Choosing Your Guarantees

Posted by Manish Goregaokar on May 27, 2015 in programming, rust

This post is now a part of the official rust book

In my previous post I talked a bit about why the RWlock pattern is important for accessing data, which is why Rust enforces this pattern either at compile time or runtime depending on the abstractions used.

It occurred to me that there are many such abstractions in Rust, each with their unique guarantees. The programmer once again has the choice between runtime and compile time enforcement. It occurred to me that this plethora of “wrapper types”[1] could be daunting to newcomers; in this post I intend to give a thorough explanation of what some prominent ones do and when they should be used.

I’m assuming the reader knows about ownership and borrowing in Rust. Nevertheless, I will attempt to keep the majority of this post accessible to those not yet familiar with these concepts. Aside from the two links into the book above, these two blog posts cover the topic in depth.

§ Basic pointer types

§ `Box<T>`

`Box<T>` is an “owned pointer” or a “box”. While it can hand out borrowed references to the data, it is the only owner of the data. In particular, when something like the following occurs:

```
let x = Box::new(1);
let y = x;
// x no longer accessible here
```

Here, the box was *moved* into `y`. As `x` no longer owns it, the compiler will no longer allow the programmer to use `x` after this. A box can similarly be moved *out* of a function by returning, and when a box (one which hasn’t been moved) goes out of scope, destructors are run, deallocating the inner data.

This abstraction is a low cost abstraction for dynamic allocation. If you want to allocate some memory on the heap and safely pass a pointer to that memory around, this is ideal. Note that you will only be allowed to share borrowed references to this by the regular borrowing rules, checked at compile time.

§ Interlude: `Copy`

Move/ownership semantics are not special to `Box<T>`; it is a feature of all types which are not `Copy`.

A `Copy` type is one where all the data it logically encompasses (usually, owns) is part of its stack representation[2]. Most types containing pointers to other data are not `Copy`, since there is additional data elsewhere, and simply copying the stack representation may accidentally share ownership of that data in an unsafe manner.

Types like `Vec<T>` and `String` which also have data on the heap are also not `Copy`. Types like the integer/boolean types are `Copy`

`&T` and raw pointers *are* `Copy`. Even though they do point to further data, they do not "own" that data. Whereas `Box<T>` can be thought of as "some data which happens to be dynamically allocated", `&T` is thought of as "a borrowing reference to some data". Even though both are pointers, only the first is considered to be "data". Hence, a copy of the first should involve a copy of the data (which is not part of its stack representation), but a copy of the second only needs a copy of the reference. `&mut T` is not `Copy` because mutable aliases cannot be shared, and `&mut T` "owns" the data it points to somewhat since it can mutate.

Practically speaking, a type can be `Copy` if a copy of its stack representation doesn't violate memory safety.

§ `&T` and `&mut T`

These are immutable and mutable references respectively. They follow the "read-write lock" pattern described in my previous post, such that one may either have only one mutable reference to some data, or any number of immutable ones, but not both. This guarantee is enforced at compile time, and has no visible cost at runtime. In most cases such pointers suffice for sharing cheap references between sections of code.

These pointers cannot be copied in such a way that they outlive the lifetime associated with them.

§ `*const T` and `*mut T`

These are C-like raw pointers with no lifetime or ownership attached to them. They just point to some location in memory with no other restrictions. The only guarantee that these provide is that they cannot be dereferenced except in code marked `unsafe`.

These are useful when building safe, low cost abstractions like `Vec<T>`, but should be avoided in safe code.

§ `Rc<T>`

This is the first wrapper we will cover that has a runtime cost.

`Rc<T>` is a reference counted pointer. In other words, this lets us have multiple "owning" pointers to the same data, and the data will be freed (destructors will be run) when all pointers are out of scope.

Internally, it contains a shared "reference count", which is incremented each time the `Rc` is cloned, and decremented each time one of the `Rc`s goes out of scope. The main responsibility of `Rc<T>` is to ensure that destructors are called for shared data.

The internal data here is immutable, and if a cycle of references is created, the data will be leaked. If we want data that doesn't leak when there are cycles, we need a *garbage collector*. I do not know of any existing GCs in Rust, but I am working on one with Nika Layzell and there's another cycle collecting one being written by Nick Fitzgerald.

§ Guarantees

The main guarantee provided here is that the data will not be destroyed until all references to it are out of scope.

This should be used when you wish to dynamically allocate and share some data (read-only) between various portions of your program, where it is not certain which portion will finish using the pointer last. It's a viable alternative to `&T` when `&T` is either impossible to statically check for correctness, or creates extremely unergonomic code where the programmer does not wish to spend the development cost of working with.

This pointer is *not* thread safe, and Rust will not let it be sent or shared with other threads. This lets one avoid the cost of atomics in situations where they are unnecessary.

There is a sister smart pointer to this one, `Weak<T>`. This is a non-owning, but also non-borrowed, smart pointer. It is also similar to `&T`, but it is not restricted in lifetime — a `Weak<T>` can be held on to forever. However, it is possible that an attempt to access the inner data may fail and return `None`, since this can outlive the owned `Rc`s. This is useful for when one wants cyclic data structures and other things.

§ Cost

As far as memory goes, `Rc<T>` is a single allocation, though it will allocate two extra words as compared to a regular `Box<T>` (for "strong" and "weak" refcounts).

`Rc<T>` has the computational cost of incrementing/decrementing the refcount whenever it is cloned or goes out of scope respectively. Note that a clone will not do a deep copy, rather it will simply increment the inner reference count and return a copy of the `Rc<T>`

§ Cell types

"Cells" provide interior mutability. In other words, they contain data which can be manipulated even if the type cannot be obtained in a mutable form (for example, when it is behind an `&`-ptr or `Rc<T>`).

The documentation for the `cell` module has a pretty good explanation for these.

These types are *generally* found in struct fields, but they may be found elsewhere too.

§ `Cell<T>`

`Cell<T>` is a type that provides zero-cost interior mutability, but only for `Copy` types. Since the compiler knows that all the data owned by the contained value is on the stack, there's no worry of leaking any data behind references (or worse!) by simply replacing the data.

It is still possible to violate your own invariants using this wrapper, so be careful when using it. If a field is wrapped in `Cell`, it's a nice indicator that the chunk of data is mutable and may not stay the same between the time you first read it and when you intend to use it.

```
let x = Cell::new(1);
let y = &x;
let z = &x;
x.set(2);
y.set(3);
z.set(4);
println!("{}", x.get());
```

Note that here we were able to mutate the same value from various immutable references.

This has the same runtime cost as the following:

```
let mut x = 1;
let y = &mut x;
let z = &mut x;
x = 2;
*y = 3;
*z = 4;
println!("{}", x;
```

but it has the added benefit of actually compiling successfully.

§ Guarantees

This relaxes the "no aliasing with mutability" restriction in places where it's unnecessary. However, this also relaxes the guarantees that the restriction provides; so if one's invariants depend on data stored within `Cell`, one should be careful.

This is useful for mutating primitives and other `Copy` types when there is no easy way of doing it in line with the static rules of `&` and `&mut`.

Gábor Lehel summed up the guarantees provided by `Cell` in a rather succinct manner:

> The basic guarantee we need to ensure is that interior references can't be invalidated (left dangling) by mutation of the outer structure. (Think about references to the interiors of types like `Option`, `Box`, `Vec`, etc.) `&`, `&mut`, and Cell each make a different tradeoff here. `&` allows shared interior references but forbids mutation; `&mut` allows mutation xor interior references but not sharing; `Cell` allows shared mutability but not interior references.

Ultimately, while shared mutability can cause many logical errors (as outlined in my previous post), it can only cause memory safety errors when coupled with "interior references". This is for types who have an "interior" whose type/size can itself be changed. One example of this is a Rust enum; where by changing the variant you can change what type is contained. If you have an alias to the inner type whilst the variant is changed, pointers within that alias may be invalidated. Similarly, if you change the length of a vector while you have an alias to one of its elements, that alias may be invalidated.

Since `Cell` doesn't allow references to the insides of a type (you can only copy out and copy back in), enums and structs alike are safe to be aliased mutably within this.

This comment by Eddy also touches on the guarantees of `Cell` and the alternatives

§ Cost

There is no runtime cost to using `Cell<T>`, however if one is using it to wrap larger (`Copy`) structs, it might be worthwhile to instead wrap individual fields in `Cell<T>` since each write is a full copy of the struct.

§ `RefCell<T>`

`RefCell<T>` also provides interior mutability, but isn't restricted to `Copy` types.

Instead, it has a runtime cost. `RefCell<T>` enforces the RWLock pattern at runtime (it's like a single-threaded mutex), unlike `&T`/`&mut T` which do so at compile time. This is done by the `borrow()` and `borrow_mut()` functions, which modify an internal reference count and return smart pointers which can be dereferenced immutably and mutably respectively. The refcount is restored when the smart pointers go out of scope. With this system, we can dynamically ensure that there are never any other borrows active when a mutable borrow is active. If the programmer attempts to make such a borrow, the thread will panic.

```
let x = RefCell::new(vec![1,2,3,4]);
{
    println!("{:?}", *x.borrow())
}

{
    let my_ref = x.borrow_mut();
    my_ref.push(1);
}
```

Similar to `Cell`, this is mainly useful for situations where it's hard or impossible to satisfy the borrow checker. Generally one knows that such mutations won't happen in a nested form, but it's good to check.

For large, complicated programs, it becomes useful to put some things in `RefCell`s to make things simpler. For example, a lot of the maps in the `ctxt` struct in the rust compiler internals are inside this wrapper. These are only modified once (during creation, which is not right after initialization) or a couple of times in well-separated places. However, since this struct is pervasively used everywhere, juggling mutable and immutable pointers would be hard (perhaps impossible) and probably form a soup of `&`-ptrs which would be hard to extend. On the other hand, the `RefCell` provides a cheap (not zero-cost) way of safely accessing these. In the future, if someone adds some code that attempts to modify the cell when it's already borrowed, it will cause a (usually deterministic) panic which can be traced back to the offending borrow.

Similarly, in Servo's DOM we have a lot of mutation, most of which is local to a DOM type, but some of which crisscrosses the DOM and modifies various things. Using `RefCell` and `Cell` to guard all mutation lets us avoid worrying about mutability everywhere, and it simultaneously highlights the places where mutation is *actually* happening.

Note that `RefCell` should be avoided if a mostly simple solution is possible with `&` pointers.

§ Guarantees

`RefCell` relaxes the *static* restrictions preventing aliased mutation, and replaces them with *dynamic* ones. As such the guarantees have not changed.

§ Cost

`RefCell` does not allocate, but it contains an additional "borrow state" indicator (one word in size) along with the data.

At runtime each borrow causes a modification/check of the refcount.

§ Synchronous types

Many of the types above cannot be used in a threadsafe manner. Particularly, `Rc<T>` and `RefCell<T>`, which both use non-atomic ref counts, cannot be used this way. This makes them cheaper to use, but one needs thread safe versions of these too. They exist, in the form of `Arc<T>` and `Mutex<T>`/`RWLock<T>`

Note that the non-threadsafe types *cannot* be sent between threads, and this is checked at compile time. I'll touch on how this is done in a later blog post.

There are many useful wrappers for concurrent programming in the sync module, but I'm only going to cover the major ones.

§ `Arc<T>`

`Arc<T>` is just a version of `Rc<T>` that uses an atomic reference count (hence, "Arc"). This can be sent freely between threads.

C++'s `shared_ptr` is similar to `Arc`, however in C++s case the inner data is always mutable. For semantics similar to that from C++, we should use `Arc<Mutex<T>>`, `Arc<RwLock<T>>`, or `Arc<UnsafeCell<T>>`[3] (`UnsafeCell<T>` is a cell type that can be used to hold any data and has no runtime cost, but accessing it requires `unsafe` blocks). The last one should only be used if one is certain that the usage won't cause any memory unsafety. Remember that writing to a struct is not an atomic operation, and many functions like `vec.push()` can reallocate internally and cause unsafe behavior (so even monotonicity[4] may not be enough to justify `UnsafeCell`)

§ Guarantees

Like `Rc`, this provides the (thread safe) guarantee that the destructor for the internal data will be run when the last `Arc` goes out of scope (barring any cycles).

§ Cost

This has the added cost of using atomics for changing the refcount (which will happen whenever it is cloned or goes out of scope). When sharing data from an `Arc` in a single thread, it is preferable to share `&` pointers whenever possible.

§ `Mutex<T>` and `RwLock<T>`

`Mutex<T>` and `RwLock<T>` provide mutual-exclusion via RAII guards. For both of these, the mutex is opaque until one calls `lock()` on it, at which point the thread will block until a lock can be acquired, and then a guard will be returned. This guard can be used to access the inner data (mutably), and the lock will be released when the guard goes out of scope.

```
{
    let guard = mutex.lock();
    // guard dereferences mutably to the inner type
    *guard += 1;
} // lock released when destructor runs
```

`RwLock` has the added benefit of being efficient for multiple reads. It is always safe to have multiple readers to shared data as long as there are no writers; and `RwLock` lets readers acquire a "read lock". Such locks can be acquired concurrently and are kept track of via a reference count. Writers must obtain a "write lock" which can only be obtained when all readers have gone out of scope.

§ Guarantees

Both of these provide safe shared mutability across threads, however they are prone to deadlocks. Some level of additional protocol safety can be obtained via the type system. An example of this is rust-sessions, an experimental library which uses session types for protocol safety.

§ Costs

These use internal atomic-like types to maintain the locks, and these are similar pretty costly (they can block all memory reads across processors till they're done). Waiting on these locks can also be slow when there's a lot of concurrent access happening.

§ Composition

A common gripe when reading Rust code is with stuff like `Rc<RefCell<Vec<T>>>` and more complicated compositions of such types.

Usually, it's a case of composing together the guarantees that one needs, without paying for stuff that is unnecessary.

For example, `Rc<RefCell<T>>` is one such composition. `Rc` itself can't be dereferenced mutably; because `Rc` provides sharing and shared mutability isn't good, so we put `RefCell` inside to get dynamically verified shared mutability. Now we have shared mutable data, but it's shared in a way that there can only be one mutator (and no readers) or multiple readers.

Now, we can take this a step further, and have `Rc<RefCell<Vec<T>>>` or `Rc<Vec<RefCell<T>>>`. These are both shareable, mutable vectors, but they're not the same.

With the former, the `RefCell` is wrapping the `Vec`, so the `Vec` in its entirety is mutable. At the same time, there can only be one mutable borrow of the whole `Vec` at a given time. This means that your code cannot simultaneously work on different elements of the vector from different `Rc` handles. However, we are able to push and pop from the `Vec` at will. This is similar to an `&mut Vec<T>` with the borrow checking done at runtime.

With the latter, the borrowing is of individual elements, but the overall vector is immutable. Thus, we can independently borrow separate elements, but we cannot push or pop from the vector. This is similar to an `&mut [T]`[5], but, again, the borrow checking is at runtime.

In concurrent programs, we have a similar situation with `Arc<Mutex<T>>`, which provides shared mutability and ownership.

When reading code that uses these, go in step by step and look at the guarantees/costs provided.

When choosing a composed type, we must do the reverse; figure out which guarantees we want, and at which point of the composition we need them. For example, if there is a choice between `Vec<RefCell<T>>` and `RefCell<Vec<T>>`, we should figure out the tradeoffs as done above and pick one.

Discuss: HN, Reddit

1. I'm not sure if this is the technical term for them, but I'll be calling them that throughout this post. ↩
2. By "stack representation" I mean the data on the stack when a value of this type is held on the stack. For example, a `Vec<T>` has a stack representation of a pointer and two integers (length, capacity). While there is more data behind the indirection of the pointer, it is not part of the stack-held portion of the `Vec`. Looking at this a different way, a type is `Copy` if a `memcopy` of the data copies all the data owned by it. ↩
3. `Arc<UnsafeCell<T>>` actually won't compile since `UnsafeCell<T>` isn't `Send` or `Sync`, but we can wrap it in a type and implement `Send`/`Sync` for it manually to get `Arc<Wrapper<T>>` where `Wrapper` is `struct Wrapper<T>(UnsafeCell<T>)`. ↩
4. By this I mean a piece of data that has a monotonic consistency requirement; i.e. a counter or a monotonically growing stack ↩
5. `&[T]` and `&mut [T]` are *slices*; they consist of a pointer and a length and can refer to a portion of a vector or array. `&mut [T]` can have its elements mutated, however its length cannot be touched. ↩

Posted by Manish Goregaokar programming, rust

Tweet

« The problem with single-threaded shared mutability Github streak: End-game and post-mortem »

About Me

I'm a self-taught programmer with interests in programming languages, human languages, Rust, physics, and online communities to name a few.

I'm heavily involved in the Rust programming language, leading the Devtools and Clippy teams. I also work at Google on ICU4X.

Recent Posts

- So Zero It's ... Negative? (Zero-Copy #3)
- Zero-Copy All the Things! (Zero-Copy #2)
- Not a Yoking Matter (Zero-Copy #1)
- Colophon: Waiter, There Are Pions in My Blog Post!
- A Tour of Safe Tracing GC Designs in Rust

Categories

- c++ (2)
- cryptography (5)
- css (1)
- elections (1)
- html (1)
- js (1)
- meta (2)
- physics (2)
- poetry (2)
- politics (1)
- programming (46)
- rust (30)
- systems (1)
- tidbits (5)
- unicode (3)
- web (2)
- writing (2)

Copyright © 2023 - Manish Goregaokar - Licensed under CC BY SA 4.0 - Powered by Octopress