*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.

- are not guaranteed to point to valid memory and are not even guaranteed to be non-null (unlike both Box and &);
- do not have any automatic clean-up, unlike Box, and so require manual resource management;
- are plain-old-data, that is, they don't move ownership, again unlike Box, hence the Rust compiler cannot protect against bugs like use-after-free;
- are considered sendable (if their contents is considered sendable), so the compiler offers no assistance with ensuring their use is thread-safe; for example, one can concurrently access a ***mut i32** from two threads without synchronization.
- lack any form of lifetimes, unlike &, and so the compiler cannot reason about dangling pointers; and
- have no guarantees about aliasing or mutability other than mutation not being allowed directly through a ***const T**.

At runtime, a **raw pointer** * and a **reference** pointing to the same piece of data have an identical representation.

In fact, an **&T** reference will implicitly coerce to an ***const T** raw pointer in safe code and similarly for the **mut** variants (both coercions can be performed explicitly with, respectively, **value as *const T** and **value as *mut T**).

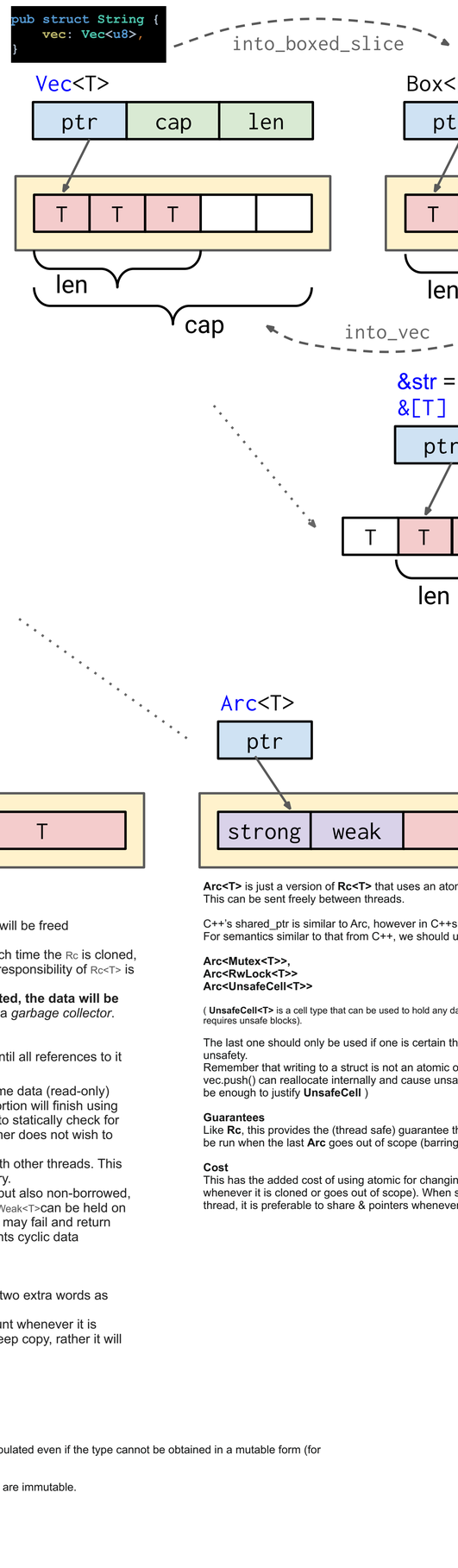
Going the opposite direction, from ***const** to a reference **&**, is not safe. A **&T** is always valid, and so, at a minimum, the raw pointer ***const T** has to point to a valid instance of type **T**. Furthermore, the resulting pointer must satisfy the aliasing and mutability laws of references. The compiler assumes these properties are true for any references, no matter how they are created, and so any conversion from raw pointers is asserting that they hold. The programmer must guarantee this.

The recommended method for the conversion is

```
let i: u32 = 1;
// explicit cast
let p_imm: *const u32 = &i as *const u32;
let mut m: u32 = 2;
// implicit coercion
let p_mut: *mut u32 = &mut m;

unsafe {
    let ref_imm: &u32 = &*p_imm;
    let ref_mut: &mut u32 = &mut *p_mut;
}
```

The **&*x** dereferencing style is preferred to using a **transmute.** The latter is far more powerful than necessary, and the more restricted operation is harder to use incorrectly; for example, it requires that **x** is a pointer (unlike **transmute**).



“**owned pointer**” or a “box”. While it can hand out borrowed references to the data, it is the only owner of the data. 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 unless it has been moved by returning, for example. when the owner is dropped, the referent goes with it.

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.

Move ownership it is a feature of all types which are not **Copy.** Most types containing pointers to other data are not **Copy**

&T and &mut T

These are **immutable** and **mutable** references respectively. They follow the "read-write lock" pattern, 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.

Reference **C**ounted pointer.
This is the first wrapper we will cover that has a runtime cost.
Its 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*.

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 Rcs. 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>

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

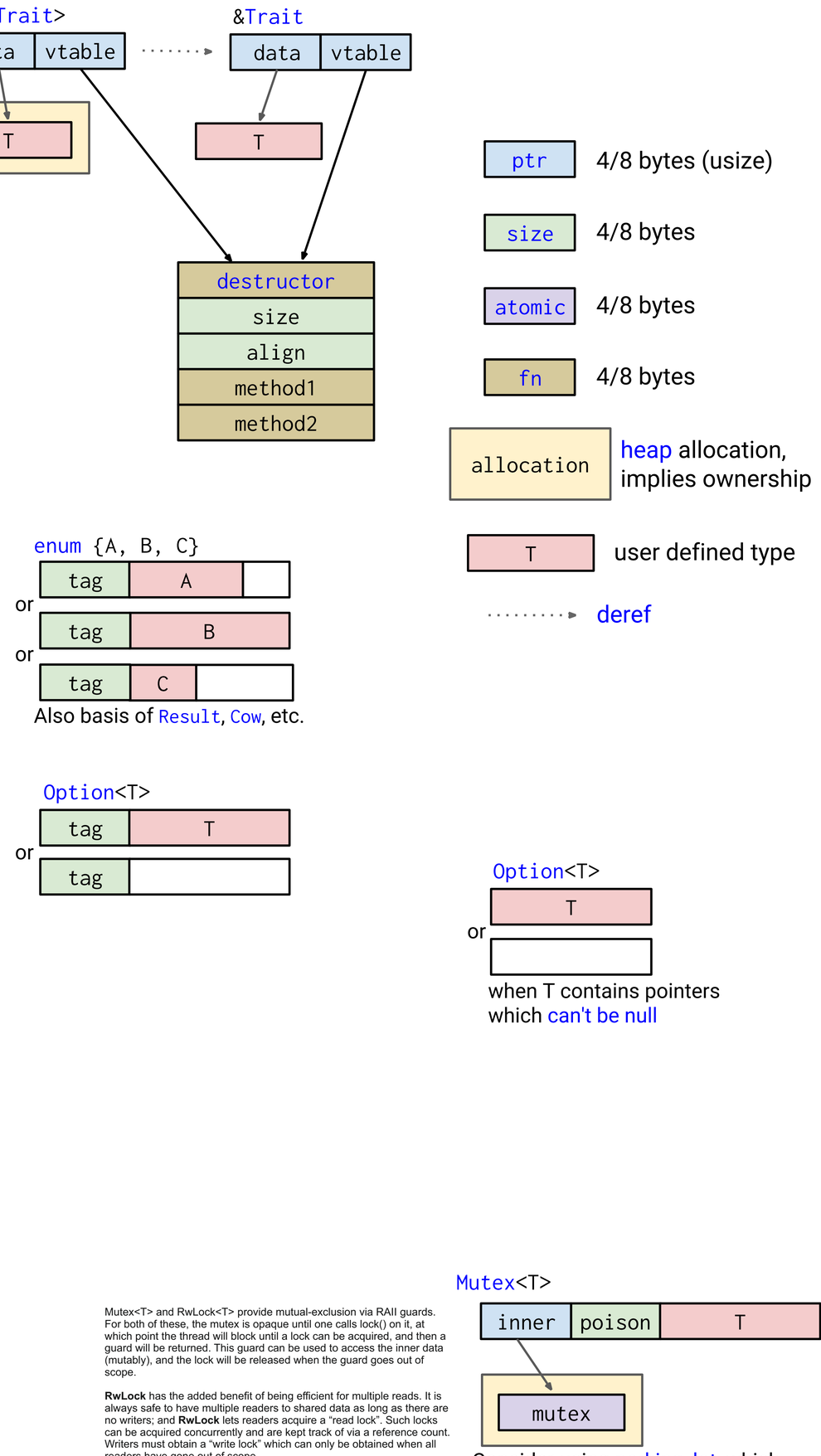
Arc<Mutex<T>>,
Arc<RwLock<T>>
Arc<UnsafeCell<T>>

(**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 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 atomic 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.



Cell types - std::cell::

"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>).

If you have a reference **&SomeStruct,** then normally in Rust **all fields** of **SomeStruct** are immutable.

```
struct SomeStruct {
    regular_field: u8,
    special_field: Cell<u8>,
}
```

The compiler makes optimizations based on the knowledge that **&T** is not mutably aliased or mutated, and that **&mut T** is unique.

UnsafeCell<T> is the only core language feature to work around this restriction. All other types that allow internal mutability, such as **Cell<T>** and **RefCell<T>**, use **UnsafeCell** to wrap their internal data.

provides zero-cost interior mutability, **but only for Copy types**.

std::marker::**Copy** is a marker trait: **trait Copy: Clone { }** , Rust permits a type to implement **Copy** only if a shallow byte-for-byte copy is all it needs. Types that own any other resources, like heap buffers or operating system handles, cannot implement **Copy**.
Any type that implements the **Drop** trait cannot be **Copy** . Rust presumes that if a type needs special clean-up code, it must also require special copying code, and thus can’t be **Copy**

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.

* **get** method retrieves the current interior value
* **take** method replaces the current interior value with **Default::default()** and returns the replaced value. For types that implement **Default.**
* **replace** method replaces the current interior value and returns the replaced value and the into_inner method consumes the Cell<T> and returns the interior value.
* **set** method replaces the interior value, dropping the replaced value.

```
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.

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.
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 , 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.

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.

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**. Uses Rust's lifetimes to implement 'dynamic borrowing', a process whereby one can claim temporary, exclusive, mutable access to the inner value.
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.

```
use std::cell::RefCell;

let x = RefCell::new(vec![1, 2, 3, 4]);
{
    println!("{:?}", *x.borrow())
}
{
    let my_ref  = x.borrow_mut(); // return RefMut
    my_ref.push(1);
}
{
    let my_ref  = x.borrow(); // return Ref
}
```

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 **RefCells** 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.

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 ref count

UnsafeCell API itself is technically very simple: it gives you a raw pointer ***mut T** to its contents. It is up to you as the abstraction designer to use that raw pointer correctly.

The precise Rust aliasing rules are somewhat in flux, but the main points are not contentious:

If you create a safe reference with lifetime **'a** (either a **&T** or **&mut T** reference) that is accessible by safe code (for example, because you returned it), then you must not access the data in any way that contradicts that reference for the remainder of **'a**.
For example, this means that if you take the ***mut T** from an **UnsafeCell<T>** and cast it to an &T, then the data in T must remain immutable (modulo any **UnsafeCell** data found within T, of course) until that reference's lifetime expires. Similarly, if you create a **&mut T** reference that is released to safe code, then you must not access the data within the **UnsafeCell** until that reference expires.

At all times, you must avoid data races. If multiple threads have access to the same **UnsafeCell,** then any writes must have a proper happens-before relation to all other accesses (or use atomics).

To assist with proper design, the following scenarios are explicitly declared legal for single-threaded code:

A **&T** reference can be released to safe code and there it can co-exist with other **&T** references, but not with a **&mut T**

A **&mut T** reference may be released to safe code provided neither other **&mut T** nor **&T** co-exist with it. A **&mut T** must always be unique.

Note that while mutating or mutably aliasing the contents of an **&UnsafeCell<T>** is okay (provided you enforce the invariants some other way), it is still undefined behavior to have multiple **&mut UnsafeCell<T>** aliases.

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.

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]S, 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.