The Rustonomicon

⚠Warning: This book is incomplete. Documenting everything and rewriting outdated parts take a while. See the issue tracker to check what's missing/outdated, and if there are any mistakes or ideas that haven't been reported, feel free to open a new issue there.

The Dark Arts of Unsafe Rust

THE KNOWLEDGE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF UNLEASHING INDESCRIBABLE HORRORS THAT SHATTER YOUR PSYCHE AND SET YOUR MIND ADRIFT IN THE UNKNOWABLY INFINITE COSMOS.

The Rustonomicon digs into all the awful details that you need to understand when writing Unsafe Rust programs.

Should you wish a long and happy career of writing Rust programs, you should turn back now and forget you ever saw this book. It is not necessary. However if you intend to write unsafe code — or just want to dig into the guts of the language — this book contains lots of useful information.

Unlike *The Rust Programming Language*, we will be assuming considerable prior knowledge. In particular, you should be comfortable with basic systems programming and Rust. If you don't feel comfortable with these topics, you should consider reading The Book first. That said, we won't assume you have read it, and we will take care to occasionally give a refresher on the basics where appropriate. You can skip straight to this book if you want; just know that we won't be explaining

everything from the ground up.

This book exists primarily as a high-level companion to The Reference. Where The Reference exists to detail the syntax and semantics of every part of the language, The Rustonomicon exists to describe how to use those pieces together, and the issues that you will have in doing so.

The Reference will tell you the syntax and semantics of references, destructors, and unwinding, but it won't tell you how combining them can lead to exception-safety issues, or how to deal with those issues.

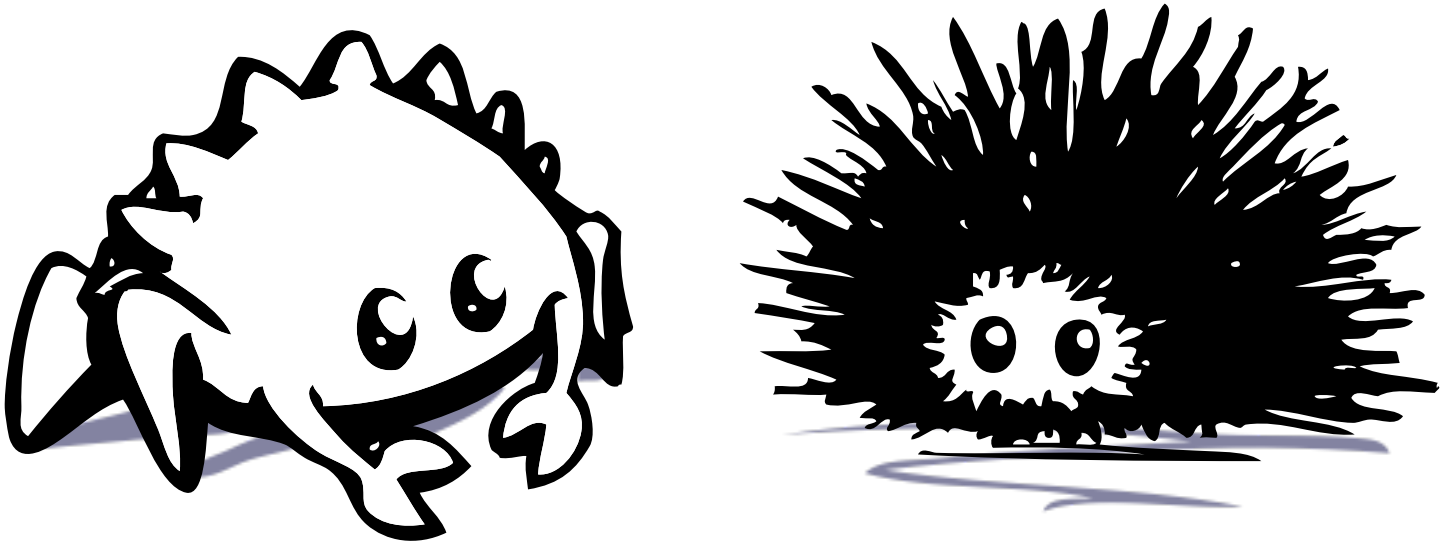
It should be noted that we haven't synced The Rustnomicon and The Reference well, so they may have a duplicate content. In general, if the two documents disagree, The Reference should be assumed to be correct (it isn't yet considered normative, it's just better maintained).

Topics that are within the scope of this book include: the meaning of (un)safety, unsafe primitives provided by the language and standard library, techniques for creating safe abstractions with those unsafe primitives, subtyping and variance, exception-safety (panic/unwind-safety), working with uninitialized memory, type punning, concurrency, interoperating with other languages (FFI), optimization tricks, how constructs lower to compiler/OS/hardware primitives, how to **not** make the memory model people angry, how you're **going** to make the memory model people angry, and more.

The Rustonomicon is not a place to exhaustively describe the semantics and guarantees of every single API in the standard library, nor is it a place to exhaustively describe every feature of Rust.

Unless otherwise noted, Rust code in this book uses the Rust 2018 edition.

Meet Safe and Unsafe



It would be great to not have to worry about low-level implementation details. Who could possibly care how much space the empty tuple occupies? Sadly, it sometimes matters and we need to worry about it. The most common reason developers start to care about implementation details is performance, but more importantly, these details can become a matter of correctness when interfacing directly with hardware, operating systems, or other languages.

When implementation details start to matter in a safe programming language, programmers usually have three options:

- fiddle with the code to encourage the compiler/runtime to perform an optimization
- adopt a more unidiomatic or cumbersome design to get the desired implementation
- rewrite the implementation in a language that lets you deal with those details

For that last option, the language programmers tend to use is *C*. This is often necessary to interface

with systems that only declare a C interface.

Unfortunately, C is incredibly unsafe to use (sometimes for good reason), and this unsafety is magnified when trying to interoperate with another language. Care must be taken to ensure C and the other language agree on what's happening, and that they don't step on each other's toes.

So what does this have to do with Rust?

Well, unlike C, Rust is a safe programming language.

But, like C, Rust is an unsafe programming language.

More accurately, Rust *contains* both a safe and unsafe programming language.

Rust can be thought of as a combination of two programming languages: *Safe Rust* and *Unsafe Rust*. Conveniently, these names mean exactly what they say: Safe Rust is Safe. Unsafe Rust is, well, not. In fact, Unsafe Rust lets us do some *really* unsafe things. Things the Rust authors will implore you not to do, but we'll do anyway.

Safe Rust is the *true* Rust programming language. If all you do is write Safe Rust, you will never have to worry about type-safety or memory-safety. You will never endure a dangling pointer, a use-after-free, or any other kind of Undefined Behavior (a.k.a. UB).

The standard library also gives you enough utilities out of the box that you'll be able to write high-performance applications and libraries in pure idiomatic Safe Rust.

But maybe you want to talk to another language. Maybe you're writing a low-level abstraction not exposed by the standard library. Maybe you're *writing* the standard library (which is written entirely in Rust). Maybe you need to do something the type-system doesn't understand and just *frob some dang bits*. Maybe you need Unsafe Rust.

Unsafe Rust is exactly like Safe Rust with all the same rules and semantics. It just lets you do some *extra* things that are Definitely Not Safe (which we will define in the next section).

The value of this separation is that we gain the benefits of using an unsafe language like C — low level control over implementation details — without most of the problems that come with trying to integrate it with a completely different safe language.

There are still some problems — most notably, we must become aware of properties that the type system assumes and audit them in any code that interacts with Unsafe Rust. That's the purpose of this book: to teach you about these assumptions and how to manage them.

How Safe and Unsafe Interact

What's the relationship between Safe Rust and Unsafe Rust? How do they interact?

The separation between Safe Rust and Unsafe Rust is controlled with the `unsafe` keyword, which acts as an interface from one to the other. This is why we can say Safe Rust is a safe language: all the unsafe parts are kept exclusively behind the `unsafe` boundary. If you wish, you can even toss `#![forbid(unsafe_code)]` into your code base to statically guarantee that you're only writing Safe Rust.

The `unsafe` keyword has two uses: to declare the existence of contracts the compiler can't check, and to declare that a programmer has checked that these contracts have been upheld.

You can use `unsafe` to indicate the existence of unchecked contracts on *functions* and *trait declarations*. On functions, `unsafe` means that users of the function must check that function's documentation to ensure they are using it in a way that maintains the contracts the function requires. On trait declarations, `unsafe` means that implementors of the trait must check the trait documentation to ensure their implementation maintains the contracts the trait requires.

You can use `unsafe` on a block to declare that all unsafe actions performed within are verified to uphold the contracts of those operations. For instance, the index passed to `slice::get_unchecked` is in-bounds.

You can use `unsafe` on a trait implementation to declare that the implementation upholds the trait's contract. For instance, that a type implementing `Send` is really safe to move to another thread.

The standard library has a number of unsafe functions, including:

- `slice::get_unchecked` , which performs unchecked indexing, allowing memory safety to be freely violated.

- `mem::transmute` reinterprets some value as having a given type, bypassing type safety in arbitrary ways (see conversions for details).
- Every raw pointer to a sized type has an `offset` method that invokes Undefined Behavior if the passed offset is not "in bounds".
- All FFI (Foreign Function Interface) functions are `unsafe` to call because the other language can do arbitrary operations that the Rust compiler can't check.

As of Rust 1.29.2 the standard library defines the following unsafe traits (there are others, but they are not stabilized yet and some of them may never be):

- `Send` is a marker trait (a trait with no API) that promises implementors are safe to send (move) to another thread.
- `Sync` is a marker trait that promises threads can safely share implementors through a shared reference.
- `GlobalAlloc` allows customizing the memory allocator of the whole program.

Much of the Rust standard library also uses Unsafe Rust internally. These implementations have generally been rigorously manually checked, so the Safe Rust interfaces built on top of these implementations can be assumed to be safe.

The need for all of this separation boils down a single fundamental property of Safe Rust, the *soundness property*:

No matter what, Safe Rust can't cause Undefined Behavior.

The design of the safe/unsafe split means that there is an asymmetric trust relationship between Safe and Unsafe Rust. Safe Rust inherently has to trust that any Unsafe Rust it touches has been written correctly. On the other hand, Unsafe Rust cannot trust Safe Rust without care.

As an example, Rust has the `PartialOrd` and `Ord` traits to differentiate between types which can "just" be compared, and those that provide a "total" ordering (which basically means that comparison behaves reasonably).

`BTreeMap` doesn't really make sense for partially-ordered types, and so it requires that its keys implement `Ord`. However, `BTreeMap` has Unsafe Rust code inside of its implementation. Because it would be unacceptable for a sloppy `Ord` implementation (which is Safe to write) to cause Undefined Behavior, the Unsafe code in BTreeMap must be written to be robust against `Ord` implementations which aren't actually total — even though that's the whole point of requiring `Ord`.

The Unsafe Rust code just can't trust the Safe Rust code to be written correctly. That said, `BTreeMap` will still behave completely erratically if you feed in values that don't have a total ordering. It just won't ever cause Undefined Behavior.

One may wonder, if `BTreeMap` cannot trust `Ord` because it's Safe, why can it trust *any* Safe code? For instance `BTreeMap` relies on integers and slices to be implemented correctly. Those are safe too, right?

The difference is one of scope. When `BTreeMap` relies on integers and slices, it's relying on one very specific implementation. This is a measured risk that can be weighed against the benefit. In this case there's basically zero risk; if integers and slices are broken, *everyone* is broken. Also, they're maintained by the same people who maintain `BTreeMap`, so it's easy to keep tabs on them.

On the other hand, `BTreeMap`'s key type is generic. Trusting its `Ord` implementation means trusting every `Ord` implementation in the past, present, and future. Here the risk is high: someone somewhere is going to make a mistake and mess up their `Ord` implementation, or even just straight up lie about providing a total ordering because "it seems to work". When that happens, `BTreeMap` needs to be prepared.

The same logic applies to trusting a closure that's passed to you to behave correctly.

This problem of unbounded generic trust is the problem that `unsafe` traits exist to resolve. The `BTreeMap` type could theoretically require that keys implement a new trait called `UnsafeOrd`, rather than `Ord`, that might look like this:

```
use std::cmp::Ordering;

unsafe trait UnsafeOrd {
    fn cmp(&self, other: &Self) -> Ordering;
}
```

Then, a type would use `unsafe` to implement `UnsafeOrd`, indicating that they've ensured their implementation maintains whatever contracts the trait expects. In this situation, the Unsafe Rust in the internals of `BTreeMap` would be justified in trusting that the key type's `UnsafeOrd` implementation is correct. If it isn't, it's the fault of the unsafe trait implementation, which is consistent with Rust's safety guarantees.

The decision of whether to mark a trait `unsafe` is an API design choice. A safe trait is easier to implement, but any unsafe code that relies on it must defend against incorrect behavior. Marking a trait `unsafe` shifts this responsibility to the implementor. Rust has traditionally avoided marking traits `unsafe` because it makes Unsafe Rust pervasive, which isn't desirable.

`Send` and `Sync` are marked unsafe because thread safety is a *fundamental property* that unsafe code can't possibly hope to defend against in the way it could defend against a buggy `Ord` implementation. Similarly, `GlobalAllocator` is keeping accounts of all the memory in the program and other things like `Box` or `Vec` build on top of it. If it does something weird (giving the same chunk of memory to another request when it is still in use), there's no chance to detect that and do anything about it.

The decision of whether to mark your own traits `unsafe` depends on the same sort of consideration. If `unsafe` code can't reasonably expect to defend against a broken implementation of the trait, then marking the trait `unsafe` is a reasonable choice.

As an aside, while `Send` and `Sync` are `unsafe` traits, they are *also* automatically implemented for types when such derivations are provably safe to do. `Send` is automatically derived for all types composed only of values whose types also implement `Send`. `Sync` is automatically derived for all

types composed only of values whose types also implement `Sync`. This minimizes the pervasive unsafety of making these two traits `unsafe`. And not many people are going to *implement* memory allocators (or use them directly, for that matter).

This is the balance between Safe and Unsafe Rust. The separation is designed to make using Safe Rust as ergonomic as possible, but requires extra effort and care when writing Unsafe Rust. The rest of this book is largely a discussion of the sort of care that must be taken, and what contracts Unsafe Rust must uphold.

What Unsafe Rust Can Do

The only things that are different in Unsafe Rust are that you can:

- Dereference raw pointers
- Call `unsafe` functions (including C functions, compiler intrinsics, and the raw allocator)
- Implement `unsafe` traits
- Mutate statics
- Access fields of `union`s

That's it. The reason these operations are relegated to Unsafe is that misusing any of these things will cause the ever dreaded Undefined Behavior. Invoking Undefined Behavior gives the compiler full rights to do arbitrarily bad things to your program. You definitely *should not* invoke Undefined Behavior.

Unlike C, Undefined Behavior is pretty limited in scope in Rust. All the core language cares about is preventing the following things:

- Dereferencing (using the `*` operator on) dangling or unaligned pointers (see below)
- Breaking the pointer aliasing rules
- Calling a function with the wrong call ABI or unwinding from a function with the wrong unwind ABI.
- Causing a data race
- Executing code compiled with target features that the current thread of execution does not support
- Producing invalid values (either alone or as a field of a compound type such as `enum` / `struct` /array/tuple):
    - a `bool` that isn't 0 or 1
    - an `enum` with an invalid discriminant
    - a null `fn` pointer

- a `char` outside the ranges [0x0, 0xD7FF] and [0xE000, 0x10FFFF]
- a `!` (all values are invalid for this type)
- an integer (`i*` / `u*`), floating point value (`f*`), or raw pointer read from uninitialized memory, or uninitialized memory in a `str` .
- a reference/ `Box` that is dangling, unaligned, or points to an invalid value.
- a wide reference, `Box` , or raw pointer that has invalid metadata:
  - `dyn Trait` metadata is invalid if it is not a pointer to a vtable for `Trait` that matches the actual dynamic trait the pointer or reference points to
  - slice metadata is invalid if the length is not a valid `usize` (i.e., it must not be read from uninitialized memory)
- a type with custom invalid values that is one of those values, such as a `NonNull` that is null. (Requesting custom invalid values is an unstable feature, but some stable libstd types, like `NonNull` , make use of it.)

"Producing" a value happens any time a value is assigned, passed to a function/primitive operation or returned from a function/primitive operation.

A reference/pointer is "dangling" if it is null or not all of the bytes it points to are part of the same allocation (so in particular they all have to be part of *some* allocation). The span of bytes it points to is determined by the pointer value and the size of the pointee type. As a consequence, if the span is empty, "dangling" is the same as "null". Note that slices and strings point to their entire range, so it's important that the length metadata is never too large (in particular, allocations and therefore slices and strings cannot be bigger than `isize::MAX` bytes). If for some reason this is too cumbersome, consider using raw pointers.

That's it. That's all the causes of Undefined Behavior baked into Rust. Of course, unsafe functions and traits are free to declare arbitrary other constraints that a program must maintain to avoid Undefined Behavior. For instance, the allocator APIs declare that deallocating unallocated memory is Undefined Behavior.

However, violations of these constraints generally will just transitively lead to one of the above

problems. Some additional constraints may also derive from compiler intrinsics that make special assumptions about how code can be optimized. For instance, Vec and Box make use of intrinsics that require their pointers to be non-null at all times.

Rust is otherwise quite permissive with respect to other dubious operations. Rust considers it "safe" to:

- Deadlock
- Have a race condition
- Leak memory
- Fail to call destructors
- Overflow integers
- Abort the program
- Delete the production database

However any program that actually manages to do such a thing is *probably* incorrect. Rust provides lots of tools to make these things rare, but these problems are considered impractical to categorically prevent.

Working with Unsafe

Rust generally only gives us the tools to talk about Unsafe Rust in a scoped and binary manner. Unfortunately, reality is significantly more complicated than that. For instance, consider the following toy function:

```
fn index(idx: usize, arr: &[u8]) -> Option<u8> {
    if idx < arr.len() {
        unsafe {
            Some(*arr.get_unchecked(idx))
        }
    } else {
        None
    }
}
```

This function is safe and correct. We check that the index is in bounds, and if it is, index into the array in an unchecked manner. We say that such a correct unsafely implemented function is *sound*, meaning that safe code cannot cause Undefined Behavior through it (which, remember, is the single fundamental property of Safe Rust).

But even in such a trivial function, the scope of the unsafe block is questionable. Consider changing the `<` to a `<=`:

```
fn index(idx: usize, arr: &[u8]) -> Option<u8> {
    if idx <= arr.len() {
        unsafe {
            Some(*arr.get_unchecked(idx))
        }
    } else {
        None
    }
}
```

This program is now *unsound*, Safe Rust can cause Undefined Behavior, and yet *we only modified safe code*. This is the fundamental problem of safety: it's non-local. The soundness of our unsafe operations necessarily depends on the state established by otherwise "safe" operations.

Safety is modular in the sense that opting into unsafety doesn't require you to consider arbitrary other kinds of badness. For instance, doing an unchecked index into a slice doesn't mean you suddenly need to worry about the slice being null or containing uninitialized memory. Nothing fundamentally changes. However safety *isn't* modular in the sense that programs are inherently stateful and your unsafe operations may depend on arbitrary other state.

This non-locality gets much worse when we incorporate actual persistent state. Consider a simple implementation of `Vec`:

```
use std::ptr;

// Note: This definition is naive. See the chapter on implementing Vec.
pub struct Vec<T> {
    ptr: *mut T,
    len: usize,
    cap: usize,
}

// Note this implementation does not correctly handle zero-sized types.
// See the chapter on implementing Vec.
impl<T> Vec<T> {
    pub fn push(&mut self, elem: T) {
        if self.len == self.cap {
            // not important for this example
            self.reallocate();
        }
        unsafe {
            ptr::write(self.ptr.add(self.len), elem);
            self.len += 1;
        }
    }
}
```

This code is simple enough to reasonably audit and informally verify. Now consider adding the following method:

```
fn make_room(&mut self) {
    // grow the capacity
    self.cap += 1;
}
```

This code is 100% Safe Rust but it is also completely unsound. Changing the capacity violates the invariants of Vec (that `cap` reflects the allocated space in the Vec). This is not something the rest of Vec can guard against. It *has* to trust the capacity field because there's no way to verify it.

Because it relies on invariants of a struct field, this `unsafe` code does more than pollute a whole function: it pollutes a whole *module*. Generally, the only bullet-proof way to limit the scope of unsafe code is at the module boundary with privacy.

However this works *perfectly*. The existence of `make_room` is *not* a problem for the soundness of Vec because we didn't mark it as public. Only the module that defines this function can call it. Also, `make_room` directly accesses the private fields of Vec, so it can only be written in the same module as Vec.

It is therefore possible for us to write a completely safe abstraction that relies on complex invariants. This is *critical* to the relationship between Safe Rust and Unsafe Rust.

We have already seen that Unsafe code must trust *some* Safe code, but shouldn't trust *generic* Safe code. Privacy is important to unsafe code for similar reasons: it prevents us from having to trust all the safe code in the universe from messing with our trusted state.

Safety lives!

Data Representation in Rust

Low-level programming cares a lot about data layout. It's a big deal. It also pervasively influences the rest of the language, so we're going to start by digging into how data is represented in Rust.

This chapter is ideally in agreement with, and rendered redundant by, the Type Layout section of the Reference. When this book was first written, the reference was in complete disrepair, and the Rustonomicon was attempting to serve as a partial replacement for the reference. This is no longer the case, so this whole chapter can ideally be deleted.

We'll keep this chapter around for a bit longer, but ideally you should be contributing any new facts or improvements to the Reference instead.

repr(Rust)

First and foremost, all types have an alignment specified in bytes. The alignment of a type specifies what addresses are valid to store the value at. A value with alignment `n` must only be stored at an address that is a multiple of `n`. So alignment 2 means you must be stored at an even address, and 1 means that you can be stored anywhere. Alignment is at least 1, and always a power of 2.

Primitives are usually aligned to their size, although this is platform-specific behavior. For example, on x86 `u64` and `f64` are often aligned to 4 bytes (32 bits).

A type's size must always be a multiple of its alignment (Zero being a valid size for any alignment). This ensures that an array of that type may always be indexed by offsetting by a multiple of its size. Note that the size and alignment of a type may not be known statically in the case of dynamically sized types.

Rust gives you the following ways to lay out composite data:

- structs (named product types)
- tuples (anonymous product types)
- arrays (homogeneous product types)
- enums (named sum types -- tagged unions)
- unions (untagged unions)

An enum is said to be *field-less* if none of its variants have associated data.

By default, composite structures have an alignment equal to the maximum of their fields' alignments. Rust will consequently insert padding where necessary to ensure that all fields are properly aligned and that the overall type's size is a multiple of its alignment. For instance:

```
struct A {
    a: u8,
    b: u32,
    c: u16,
}
```

will be 32-bit aligned on a target that aligns these primitives to their respective sizes. The whole struct will therefore have a size that is a multiple of 32-bits. It may become:

```
struct A {
    a: u8,
    _pad1: [u8; 3], // to align `b`
    b: u32,
    c: u16,
    _pad2: [u8; 2], // to make overall size multiple of 4
}
```

or maybe:

```
struct A {
    b: u32,
    c: u16,
    a: u8,
    _pad: u8,
}
```

There is *no indirection* for these types; all data is stored within the struct, as you would expect in C. However with the exception of arrays (which are densely packed and in-order), the layout of data is not specified by default. Given the two following struct definitions:

```
struct A {
    a: i32,
    b: u64,
}

struct B {
    a: i32,
    b: u64,
}
```

Rust *does* guarantee that two instances of A have their data laid out in exactly the same way. However Rust *does not* currently guarantee that an instance of A has the same field ordering or padding as an instance of B.

With A and B as written, this point would seem to be pedantic, but several other features of Rust make it desirable for the language to play with data layout in complex ways.

For instance, consider this struct:

```
struct Foo<T, U> {
    count: u16,
    data1: T,
    data2: U,
}
```

Now consider the monomorphizations of `Foo<u32, u16>` and `Foo<u16, u32>`. If Rust lays out the fields in the order specified, we expect it to pad the values in the struct to satisfy their alignment requirements. So if Rust didn't reorder fields, we would expect it to produce the following:

```
struct Foo<u16, u32> {
    count: u16,
    data1: u16,
    data2: u32,
}

struct Foo<u32, u16> {
    count: u16,
    _pad1: u16,
    data1: u32,
    data2: u16,
    _pad2: u16,
}
```

The latter case quite simply wastes space. An optimal use of space requires different monomorphizations to have *different field orderings*.

Enums make this consideration even more complicated. Naively, an enum such as:

```
enum Foo {
    A(u32),
    B(u64),
    C(u8),
}
```

might be laid out as:

```
struct FooRepr {
    data: u64, // this is either a u64, u32, or u8 based on `tag`
    tag: u8,   // 0 = A, 1 = B, 2 = C
}
```

And indeed this is approximately how it would be laid out (modulo the size and position of `tag`).

However there are several cases where such a representation is inefficient. The classic case of this is Rust's "null pointer optimization": an enum consisting of a single outer unit variant (e.g. `None`) and a (potentially nested) non- nullable pointer variant (e.g. `Some(&T)`) makes the tag unnecessary. A null pointer can safely be interpreted as the unit (`None`) variant. The net result is that, for example, `size_of::<Option<&T>>() == size_of::<&T>()`.

There are many types in Rust that are, or contain, non-nullable pointers such as `Box<T>`, `Vec<T>`, `String`, `&T`, and `&mut T`. Similarly, one can imagine nested enums pooling their tags into a single discriminant, as they are by definition known to have a limited range of valid values. In principle enums could use fairly elaborate algorithms to store bits throughout nested types with forbidden values. As such it is *especially* desirable that we leave enum layout unspecified today.

Exotically Sized Types

Most of the time, we expect types to have a statically known and positive size. This isn't always the case in Rust.

Dynamically Sized Types (DSTs)

Rust supports Dynamically Sized Types (DSTs): types without a statically known size or alignment. On the surface, this is a bit nonsensical: Rust *must* know the size and alignment of something in order to correctly work with it! In this regard, DSTs are not normal types. Because they lack a statically known size, these types can only exist behind a pointer. Any pointer to a DST consequently becomes a *wide* pointer consisting of the pointer and the information that "completes" them (more on this below).

There are two major DSTs exposed by the language:

- trait objects: `dyn MyTrait`
- slices: `[T]`, `str`, and others

A trait object represents some type that implements the traits it specifies. The exact original type is *erased* in favor of runtime reflection with a vtable containing all the information necessary to use the type. The information that completes a trait object pointer is the vtable pointer. The runtime size of the pointee can be dynamically requested from the vtable.

A slice is simply a view into some contiguous storage -- typically an array or `Vec`. The information that completes a slice pointer is just the number of elements it points to. The runtime size of the pointee is just the statically known size of an element multiplied by the number of elements.

Structs can actually store a single DST directly as their last field, but this makes them a DST as well:

```
// Can't be stored on the stack directly
struct MySuperSlice {
    info: u32,
    data: [u8],
}
```

Although such a type is largely useless without a way to construct it. Currently the only properly supported way to create a custom DST is by making your type generic and performing an *unsizing coercion*:

```
struct MySuperSliceable<T: ?Sized> {
    info: u32,
    data: T,
}

fn main() {
    let sized: MySuperSliceable<[u8; 8]> = MySuperSliceable {
        info: 17,
        data: [0; 8],
    };

    let dynamic: &MySuperSliceable<[u8]> = &sized;

    // prints: "17 [0, 0, 0, 0, 0, 0, 0, 0]"
    println!("{} {:?}", dynamic.info, &dynamic.data);
}
```

(Yes, custom DSTs are a largely half-baked feature for now.)

Zero Sized Types (ZSTs)

Rust also allows types to be specified that occupy no space:

```
struct Nothing; // No fields = no size

// All fields have no size = no size
struct LotsOfNothing {
    foo: Nothing,
    qux: (),      // empty tuple has no size
    baz: [u8; 0], // empty array has no size
}
```

On their own, Zero Sized Types (ZSTs) are, for obvious reasons, pretty useless. However as with many curious layout choices in Rust, their potential is realized in a generic context: Rust largely understands that any operation that produces or stores a ZST can be reduced to a no-op. First off, storing it doesn't even make sense -- it doesn't occupy any space. Also there's only one value of that type, so anything that loads it can just produce it from the aether -- which is also a no-op since it doesn't occupy any space.

One of the most extreme examples of this is Sets and Maps. Given a `Map<Key, Value>`, it is common to implement a `Set<Key>` as just a thin wrapper around `Map<Key, UselessJunk>`. In many languages, this would necessitate allocating space for UselessJunk and doing work to store and load UselessJunk only to discard it. Proving this unnecessary would be a difficult analysis for the compiler.

However in Rust, we can just say that `Set<Key> = Map<Key, ()>`. Now Rust statically knows that every load and store is useless, and no allocation has any size. The result is that the monomorphized code is basically a custom implementation of a HashSet with none of the overhead that HashMap would have to support values.

Safe code need not worry about ZSTs, but *unsafe* code must be careful about the consequence of types with no size. In particular, pointer offsets are no-ops, and allocators typically require a non-zero size.

Note that references to ZSTs (including empty slices), just like all other references, must be non-null and suitably aligned. Dereferencing a null or unaligned pointer to a ZST is undefined behavior, just

like for any other type.

Empty Types

Rust also enables types to be declared that *cannot even be instantiated*. These types can only be talked about at the type level, and never at the value level. Empty types can be declared by specifying an enum with no variants:

```
enum Void {} // No variants = EMPTY
```

Empty types are even more marginal than ZSTs. The primary motivating example for an empty type is type-level unreachability. For instance, suppose an API needs to return a Result in general, but a specific case actually is infallible. It's actually possible to communicate this at the type level by returning a `Result<T, Void>`. Consumers of the API can confidently unwrap such a Result knowing that it's *statically impossible* for this value to be an `Err`, as this would require providing a value of type `Void`.

In principle, Rust can do some interesting analyses and optimizations based on this fact. For instance, `Result<T, Void>` is represented as just `T`, because the `Err` case doesn't actually exist (strictly speaking, this is only an optimization that is not guaranteed, so for example transmuting one into the other is still Undefined Behavior).

The following *could* also compile:

```
enum Void {}

let res: Result<u32, Void> = Ok(0);

// Err doesn't exist anymore, so Ok is actually irrefutable.
let Ok(num) = res;
```

But this trick doesn't work yet.

One final subtle detail about empty types is that raw pointers to them are actually valid to construct, but dereferencing them is Undefined Behavior because that wouldn't make sense.

We recommend against modelling C's `void*` type with `*const Void`. A lot of people started doing that but quickly ran into trouble because Rust doesn't really have any safety guards against trying to instantiate empty types with unsafe code, and if you do it, it's Undefined Behaviour. This was especially problematic because developers had a habit of converting raw pointers to references and `&Void` is *also* Undefined Behaviour to construct.

`*const ()` (or equivalent) works reasonably well for `void*`, and can be made into a reference without any safety problems. It still doesn't prevent you from trying to read or write values, but at least it compiles to a no-op instead of Undefined Behavior.

Extern Types

There is an accepted RFC to add proper types with an unknown size, called *extern types*, which would let Rust developers model things like C's `void*` and other "declared but never defined" types more accurately. However as of Rust 2018, the feature is stuck in limbo over how `size_of_val::<MyExternType>()` should behave.

Alternative representations

Rust allows you to specify alternative data layout strategies from the default. There's also the unsafe code guidelines (note that it's **NOT** normative).

repr(C)

This is the most important `repr`. It has fairly simple intent: do what C does. The order, size, and alignment of fields is exactly what you would expect from C or C++. Any type you expect to pass through an FFI boundary should have `repr(C)`, as C is the lingua-franca of the programming world. This is also necessary to soundly do more elaborate tricks with data layout such as reinterpreting values as a different type.

We strongly recommend using rust-bindgen and/or cbindgen to manage your FFI boundaries for you. The Rust team works closely with those projects to ensure that they work robustly and are compatible with current and future guarantees about type layouts and `repr`s.

The interaction of `repr(C)` with Rust's more exotic data layout features must be kept in mind. Due to its dual purpose as "for FFI" and "for layout control", `repr(C)` can be applied to types that will be nonsensical or problematic if passed through the FFI boundary.

- ZSTs are still zero-sized, even though this is not a standard behavior in C, and is explicitly contrary to the behavior of an empty type in C++, which says they should still consume a byte of space.
- DST pointers (wide pointers) and tuples are not a concept in C, and as such are never FFI-safe.
- Enums with fields also aren't a concept in C or C++, but a valid bridging of the types is defined.

- If `T` is an FFI-safe non-nullable pointer type, `Option<T>` is guaranteed to have the same layout and ABI as `T` and is therefore also FFI-safe. As of this writing, this covers `&`, `&mut`, and function pointers, all of which can never be null.
- Tuple structs are like structs with regards to `repr(C)`, as the only difference from a struct is that the fields aren't named.
- `repr(C)` is equivalent to one of `repr(u*)` (see the next section) for fieldless enums. The chosen size is the default enum size for the target platform's C application binary interface (ABI). Note that enum representation in C is implementation defined, so this is really a "best guess". In particular, this may be incorrect when the C code of interest is compiled with certain flags.
- Fieldless enums with `repr(C)` or `repr(u*)` still may not be set to an integer value without a corresponding variant, even though this is permitted behavior in C or C++. It is undefined behavior to (unsafely) construct an instance of an enum that does not match one of its variants. (This allows exhaustive matches to continue to be written and compiled as normal.)

repr(transparent)

This can only be used on structs with a single non-zero-sized field (there may be additional zero-sized fields). The effect is that the layout and ABI of the whole struct is guaranteed to be the same as that one field.

The goal is to make it possible to transmute between the single field and the struct. An example of that is `UnsafeCell`, which can be transmuted into the type it wraps (`UnsafeCell` also uses the unstable no_niche, so its ABI is not actually guaranteed to be the same when nested in other types).

Also, passing the struct through FFI where the inner field type is expected on the other side is guaranteed to work. In particular, this is necessary for `struct Foo(f32)` to always have the same

ABI as `f32`.

This repr is only considered part of the public ABI of a type if either the single field is `pub`, or if its layout is documented in prose. Otherwise, the layout should not be relied upon by other crates.

More details are in the RFC.

repr(u*), repr(i*)

These specify the size to make a fieldless enum. If the discriminant overflows the integer it has to fit in, it will produce a compile-time error. You can manually ask Rust to allow this by setting the overflowing element to explicitly be 0. However Rust will not allow you to create an enum where two variants have the same discriminant.

The term "fieldless enum" only means that the enum doesn't have data in any of its variants. A fieldless enum without a `repr(u*)` or `repr(C)` is still a Rust native type, and does not have a stable ABI representation. Adding a `repr` causes it to be treated exactly like the specified integer size for ABI purposes.

If the enum has fields, the effect is similar to the effect of `repr(C)` in that there is a defined layout of the type. This makes it possible to pass the enum to C code, or access the type's raw representation and directly manipulate its tag and fields. See the RFC for details.

These `repr`s have no effect on a struct.

Adding an explicit `repr(u*)`, `repr(i*)`, or `repr(C)` to an enum with fields suppresses the null-pointer optimization, like:

```
enum MyOption<T> {
    Some(T),
    None,
}

#[repr(u8)]
enum MyReprOption<T> {
    Some(T),
    None,
}

assert_eq!(8, size_of::<MyOption<&u16>>());
assert_eq!(16, size_of::<MyReprOption<&u16>>());
```

This optimization still applies to fieldless enums with an explicit `repr(u*)`, `repr(i*)`, or `repr(C)`.

repr(packed)

`repr(packed)` forces Rust to strip any padding, and only align the type to a byte. This may improve the memory footprint, but will likely have other negative side-effects.

In particular, most architectures *strongly* prefer values to be aligned. This may mean the unaligned loads are penalized (x86), or even fault (some ARM chips). For simple cases like directly loading or storing a packed field, the compiler might be able to paper over alignment issues with shifts and masks. However if you take a reference to a packed field, it's unlikely that the compiler will be able to emit code to avoid an unaligned load.

As this can cause undefined behavior, the lint has been implemented and it will become a hard error.

`repr(packed)` is not to be used lightly. Unless you have extreme requirements, this should not be

used.

This repr is a modifier on `repr(C)` and `repr(Rust)`.

repr(align(n))

`repr(align(n))` (where `n` is a power of two) forces the type to have an alignment of *at least* n.

This enables several tricks, like making sure neighboring elements of an array never share the same cache line with each other (which may speed up certain kinds of concurrent code).

This is a modifier on `repr(C)` and `repr(Rust)`. It is incompatible with `repr(packed)`.

Ownership and Lifetimes

Ownership is the breakout feature of Rust. It allows Rust to be completely memory-safe and efficient, while avoiding garbage collection. Before getting into the ownership system in detail, we will consider the motivation of this design.

We will assume that you accept that garbage collection (GC) is not always an optimal solution, and that it is desirable to manually manage memory in some contexts. If you do not accept this, might I interest you in a different language?

Regardless of your feelings on GC, it is pretty clearly a *massive* boon to making code safe. You never have to worry about things going away *too soon* (although whether you still wanted to be pointing at that thing is a different issue...). This is a pervasive problem that C and C++ programs need to deal with. Consider this simple mistake that all of us who have used a non-GC'd language have made at one point:

```
fn as_str(data: &u32) -> &str {
    // compute the string
    let s = format!("{}", data);

    // OH NO! We returned a reference to something that
    // exists only in this function!
    // Dangling pointer! Use after free! Alas!
    // (this does not compile in Rust)
    &s
}
```

This is exactly what Rust's ownership system was built to solve. Rust knows the scope in which the `&s` lives, and as such can prevent it from escaping. However this is a simple case that even a C compiler could plausibly catch. Things get more complicated as code gets bigger and pointers get fed through various functions. Eventually, a C compiler will fall down and won't be able to perform sufficient escape analysis to prove your code unsound. It will consequently be forced to accept your

program on the assumption that it is correct.

This will never happen to Rust. It's up to the programmer to prove to the compiler that everything is sound.

Of course, Rust's story around ownership is much more complicated than just verifying that references don't escape the scope of their referent. That's because ensuring pointers are always valid is much more complicated than this. For instance in this code,

```
let mut data = vec![1, 2, 3];
// get an internal reference
let x = &data[0];

// OH NO! `push` causes the backing storage of `data` to be reallocated.
// Dangling pointer! Use after free! Alas!
// (this does not compile in Rust)
data.push(4);

println!("{}", x);
```

naive scope analysis would be insufficient to prevent this bug, because `data` does in fact live as long as we needed. However it was *changed* while we had a reference into it. This is why Rust requires any references to freeze the referent and its owners.

References

There are two kinds of reference:

- Shared reference: `&`
- Mutable reference: `&mut`

Which obey the following rules:

- A reference cannot outlive its referent
- A mutable reference cannot be aliased

That's it. That's the whole model references follow.

Of course, we should probably define what *aliased* means.

```
error[E0425]: cannot find value `aliased` in this scope
 --> <rust.rs>:2:20
  |
2 |     println!("{}", aliased);
  |                    ^^^^^^^ not found in this scope

error: aborting due to previous error
```

Unfortunately, Rust hasn't actually defined its aliasing model. 🙀

While we wait for the Rust devs to specify the semantics of their language, let's use the next section to discuss what aliasing is in general, and why it matters.

Aliasing

First off, let's get some important caveats out of the way:

- We will be using the broadest possible definition of aliasing for the sake of discussion. Rust's definition will probably be more restricted to factor in mutations and liveness.
- We will be assuming a single-threaded, interrupt-free, execution. We will also be ignoring things like memory-mapped hardware. Rust assumes these things don't happen unless you tell it otherwise. For more details, see the Concurrency Chapter.

With that said, here's our working definition: variables and pointers *alias* if they refer to overlapping regions of memory.

Why Aliasing Matters

So why should we care about aliasing?

Consider this simple function:

```
fn compute(input: &u32, output: &mut u32) {
    if *input > 10 {
        *output = 1;
    }
    if *input > 5 {
        *output *= 2;
    }
    // remember that `output` will be `2` if `input > 10`
}
```

We would *like* to be able to optimize it to the following function:

```
fn compute(input: &u32, output: &mut u32) {
    let cached_input = *input; // keep `*input` in a register
    if cached_input > 10 {
        // If the input is greater than 10, the previous code would set the output to 1
and then double it,
        // resulting in an output of 2 (because `>10` implies `>5`).
        // Here, we avoid the double assignment and just set it directly to 2.
        *output = 2;
    } else if cached_input > 5 {
        *output *= 2;
    }
}
```

In Rust, this optimization should be sound. For almost any other language, it wouldn't be (barring global analysis). This is because the optimization relies on knowing that aliasing doesn't occur, which most languages are fairly liberal with. Specifically, we need to worry about function arguments that make `input` and `output` overlap, such as `compute(&x, &mut x)`.

With that input, we could get this execution:

```
                    //  input ==  output == 0xabad1dea
                    // *input == *output == 20
if *input > 10 {    // true  (*input == 20)
    *output = 1;    // also overwrites *input, because they are the same
}
if *input > 5 {     // false (*input == 1)
    *output *= 2;
}
                    // *input == *output == 1
```

Our optimized function would produce `*output == 2` for this input, so the correctness of our optimization relies on this input being impossible.

In Rust we know this input should be impossible because `&mut` isn't allowed to be aliased. So we can safely reject its possibility and perform this optimization. In most other languages, this input would be entirely possible, and must be considered.

This is why alias analysis is important: it lets the compiler perform useful optimizations! Some examples:

- keeping values in registers by proving no pointers access the value's memory
- eliminating reads by proving some memory hasn't been written to since last we read it
- eliminating writes by proving some memory is never read before the next write to it
- moving or reordering reads and writes by proving they don't depend on each other

These optimizations also tend to prove the soundness of bigger optimizations such as loop vectorization, constant propagation, and dead code elimination.

In the previous example, we used the fact that `&mut u32` can't be aliased to prove that writes to `*output` can't possibly affect `*input`. This let us cache `*input` in a register, eliminating a read.

By caching this read, we knew that the write in the `> 10` branch couldn't affect whether we take the `> 5` branch, allowing us to also eliminate a read-modify-write (doubling `*output`) when `*input > 10`.

The key thing to remember about alias analysis is that writes are the primary hazard for optimizations. That is, the only thing that prevents us from moving a read to any other part of the program is the possibility of us re-ordering it with a write to the same location.

For instance, we have no concern for aliasing in the following modified version of our function, because we've moved the only write to `*output` to the very end of our function. This allows us to freely reorder the reads of `*input` that occur before it:

```
fn compute(input: &u32, output: &mut u32) {
    let mut temp = *output;
    if *input > 10 {
        temp = 1;
    }
    if *input > 5 {
        temp *= 2;
    }
    *output = temp;
}
```

We're still relying on alias analysis to assume that `temp` doesn't alias `input`, but the proof is much simpler: the value of a local variable can't be aliased by things that existed before it was declared. This is an assumption every language freely makes, and so this version of the function could be optimized the way we want in any language.

This is why the definition of "alias" that Rust will use likely involves some notion of liveness and mutation: we don't actually care if aliasing occurs if there aren't any actual writes to memory happening.

Of course, a full aliasing model for Rust must also take into consideration things like function calls (which may mutate things we don't see), raw pointers (which have no aliasing requirements on their own), and UnsafeCell (which lets the referent of an `&` be mutated).

Lifetimes

Rust enforces these rules through *lifetimes*. Lifetimes are named regions of code that a reference must be valid for. Those regions may be fairly complex, as they correspond to paths of execution in the program. There may even be holes in these paths of execution, as it's possible to invalidate a reference as long as it's reinitialized before it's used again. Types which contain references (or pretend to) may also be tagged with lifetimes so that Rust can prevent them from being invalidated as well.

In most of our examples, the lifetimes will coincide with scopes. This is because our examples are simple. The more complex cases where they don't coincide are described below.

Within a function body, Rust generally doesn't let you explicitly name the lifetimes involved. This is because it's generally not really necessary to talk about lifetimes in a local context; Rust has all the information and can work out everything as optimally as possible. Many anonymous scopes and temporaries that you would otherwise have to write are often introduced to make your code Just Work.

However once you cross the function boundary, you need to start talking about lifetimes. Lifetimes are denoted with an apostrophe: `'a`, `'static`. To dip our toes with lifetimes, we're going to pretend that we're actually allowed to label scopes with lifetimes, and desugar the examples from the start of this chapter.

Originally, our examples made use of *aggressive* sugar -- high fructose corn syrup even -- around scopes and lifetimes, because writing everything out explicitly is *extremely noisy*. All Rust code relies on aggressive inference and elision of "obvious" things.

One particularly interesting piece of sugar is that each `let` statement implicitly introduces a scope. For the most part, this doesn't really matter. However it does matter for variables that refer to each other. As a simple example, let's completely desugar this simple piece of Rust code:

```
let x = 0;
let y = &x;
let z = &y;
```

The borrow checker always tries to minimize the extent of a lifetime, so it will likely desugar to the following:

```
// NOTE: `'a: {` and `&'b x` is not valid syntax!
'a: {
    let x: i32 = 0;
    'b: {
        // lifetime used is 'b because that's good enough.
        let y: &'b i32 = &'b x;
        'c: {
            // ditto on 'c
            let z: &'c &'b i32 = &'c y;
        }
    }
}
```

Wow. That's... awful. Let's all take a moment to thank Rust for making this easier.

Actually passing references to outer scopes will cause Rust to infer a larger lifetime:

```
let x = 0;
let z;
let y = &x;
z = y;
```

```
'a: {
    let x: i32 = 0;
    'b: {
        let z: &'b i32;
        'c: {
            // Must use 'b here because the reference to x is
            // being passed to the scope 'b.
            let y: &'b i32 = &'b x;
            z = y;
        }
    }
}
```

Example: references that outlive referents

Alright, let's look at some of those examples from before:

```
fn as_str(data: &u32) -> &str {
    let s = format!("{}", data);
    &s
}
```

desugars to:

```
fn as_str<'a>(data: &'a u32) -> &'a str {
    'b: {
        let s = format!("{}", data);
        return &'a s;
    }
}
```

This signature of `as_str` takes a reference to a u32 with *some* lifetime, and promises that it can

produce a reference to a str that can live *just as long*. Already we can see why this signature might be trouble. That basically implies that we're going to find a str somewhere in the scope the reference to the u32 originated in, or somewhere *even earlier*. That's a bit of a tall order.

We then proceed to compute the string `s`, and return a reference to it. Since the contract of our function says the reference must outlive `'a`, that's the lifetime we infer for the reference. Unfortunately, `s` was defined in the scope `'b`, so the only way this is sound is if `'b` contains `'a` -- which is clearly false since `'a` must contain the function call itself. We have therefore created a reference whose lifetime outlives its referent, which is *literally* the first thing we said that references can't do. The compiler rightfully blows up in our face.

To make this more clear, we can expand the example:

```
fn as_str<'a>(data: &'a u32) -> &'a str {
    'b: {
        let s = format!("{}", data);
        return &'a s
    }
}

fn main() {
    'c: {
        let x: u32 = 0;
        'd: {
            // An anonymous scope is introduced because the borrow does not
            // need to last for the whole scope x is valid for. The return
            // of as_str must find a str somewhere before this function
            // call. Obviously not happening.
            println!("{}", as_str::<'d>(&'d x));
        }
    }
}
```

Shoot!

Of course, the right way to write this function is as follows:

```
fn to_string(data: &u32) -> String {
    format!("{}", data)
}
```

We must produce an owned value inside the function to return it! The only way we could have returned an `&'a str` would have been if it was in a field of the `&'a u32`, which is obviously not the case.

(Actually we could have also just returned a string literal, which as a global can be considered to reside at the bottom of the stack; though this limits our implementation *just a bit*.)

Example: aliasing a mutable reference

How about the other example:

```
let mut data = vec![1, 2, 3];
let x = &data[0];
data.push(4);
println!("{}", x);
```

```
'a: {
    let mut data: Vec<i32> = vec![1, 2, 3];
    'b: {
        // 'b is as big as we need this borrow to be
        // (just need to get to `println!`)
        let x: &'b i32 = Index::index::<'b>(&'b data, 0);
        'c: {
            // Temporary scope because we don't need the
            // &mut to last any longer.
            Vec::push(&'c mut data, 4);
        }
        println!("{}", x);
    }
}
```

The problem here is a bit more subtle and interesting. We want Rust to reject this program for the following reason: We have a live shared reference `x` to a descendant of `data` when we try to take a mutable reference to `data` to `push`. This would create an aliased mutable reference, which would violate the *second* rule of references.

However this is *not at all* how Rust reasons that this program is bad. Rust doesn't understand that `x` is a reference to a subpath of `data`. It doesn't understand `Vec` at all. What it *does* see is that `x` has to live for `'b` in order to be printed. The signature of `Index::index` subsequently demands that the reference we take to `data` has to survive for `'b`. When we try to call `push`, it then sees us try to make an `&'c mut data`. Rust knows that `'c` is contained within `'b`, and rejects our program because the `&'b data` must still be alive!

Here we see that the lifetime system is much more coarse than the reference semantics we're actually interested in preserving. For the most part, *that's totally ok*, because it keeps us from spending all day explaining our program to the compiler. However it does mean that several programs that are totally correct with respect to Rust's *true* semantics are rejected because lifetimes are too dumb.

The area covered by a lifetime

A reference (sometimes called a *borrow*) is *alive* from the place it is created to its last use. The borrowed value needs to outlive only borrows that are alive. This looks simple, but there are a few subtleties.

The following snippet compiles, because after printing `x`, it is no longer needed, so it doesn't matter if it is dangling or aliased (even though the variable `x` *technically* exists to the very end of the scope).

```
let mut data = vec![1, 2, 3];
let x = &data[0];
println!("{}", x);
// This is OK, x is no longer needed
data.push(4);
```

However, if the value has a destructor, the destructor is run at the end of the scope. And running the destructor is considered a use - obviously the last one. So, this will *not* compile.

```
#[derive(Debug)]
struct X<'a>(&'a i32);

impl Drop for X<'_> {
    fn drop(&mut self) {}
}

let mut data = vec![1, 2, 3];
let x = X(&data[0]);
println!("{:?}", x);
data.push(4);
// Here, the destructor is run and therefore this'll fail to compile.
```

One way to convince the compiler that `x` is no longer valid is by using `drop(x)` before `data.push(4)`.

Furthermore, there might be multiple possible last uses of the borrow, for example in each branch of a condition.

```
let mut data = vec![1, 2, 3];
let x = &data[0];

if some_condition() {
    println!("{}", x); // This is the last use of `x` in this branch
    data.push(4);      // So we can push here
} else {
    // There's no use of `x` in here, so effectively the last use is the
    // creation of x at the top of the example.
    data.push(5);
}
```

And a lifetime can have a pause in it. Or you might look at it as two distinct borrows just being tied to the same local variable. This often happens around loops (writing a new value of a variable at the end of the loop and using it for the last time at the top of the next iteration).

```
let mut data = vec![1, 2, 3];
// This mut allows us to change where the reference points to
let mut x = &data[0];

println!("{}", x); // Last use of this borrow
data.push(4);
x = &data[3]; // We start a new borrow here
println!("{}", x);
```

Historically, Rust kept the borrow alive until the end of scope, so these examples might fail to compile with older compilers. Also, there are still some corner cases where Rust fails to properly shorten the live part of the borrow and fails to compile even when it looks like it should. These'll be solved over time.

Limits of Lifetimes

Given the following code:

```
#[derive(Debug)]
struct Foo;

impl Foo {
    fn mutate_and_share(&mut self) -> &Self { &*self }
    fn share(&self) {}
}

fn main() {
    let mut foo = Foo;
    let loan = foo.mutate_and_share();
    foo.share();
    println!("{:?}", loan);
}
```

One might expect it to compile. We call `mutate_and_share`, which mutably borrows `foo` temporarily, but then returns only a shared reference. Therefore we would expect `foo.share()` to succeed as `foo` shouldn't be mutably borrowed.

However when we try to compile it:

```
error[E0502]: cannot borrow `foo` as immutable because it is also borrowed as mutable
  --> src/main.rs:12:5
   |
11 |     let loan = foo.mutate_and_share();
   |                --- mutable borrow occurs here
12 |     foo.share();
   |     ^^^ immutable borrow occurs here
13 |     println!("{:?}", loan);
```

What happened? Well, we got the exact same reasoning as we did for Example 2 in the previous

section. We desugar the program and we get the following:

```
struct Foo;

impl Foo {
    fn mutate_and_share<'a>(&'a mut self) -> &'a Self { &'a *self }
    fn share<'a>(&'a self) {}
}

fn main() {
    'b: {
        let mut foo: Foo = Foo;
        'c: {
            let loan: &'c Foo = Foo::mutate_and_share::<'c>(&'c mut foo);
            'd: {
                Foo::share::<'d>(&'d foo);
            }
            println!("{:?}", loan);
        }
    }
}
```

The lifetime system is forced to extend the `&mut foo` to have lifetime `'c`, due to the lifetime of `loan` and `mutate_and_share`'s signature. Then when we try to call `share`, and it sees we're trying to alias that `&'c mut foo` and blows up in our face!

This program is clearly correct according to the reference semantics we actually care about, but the lifetime system is too coarse-grained to handle that.

Improperly reduced borrows

The following code fails to compile, because Rust sees that a variable, `map`, is borrowed twice, and can not infer that the first borrow stops to be needed before the second one occurs. This is caused

by Rust conservatively falling back to using a whole scope for the first borow. This will eventually get fixed.

```
fn get_default<'m, K, V>(map: &'m mut HashMap<K, V>, key: K) -> &'m mut V
where
    K: Clone + Eq + Hash,
    V: Default,
{
    match map.get_mut(&key) {
        Some(value) => value,
        None => {
            map.insert(key.clone(), V::default());
            map.get_mut(&key).unwrap()
        }
    }
}
```

Because of the lifetime restrictions imposed, `&mut map` 's lifetime overlaps other mutable borrows, resulting in a compile error:

```
error[E0499]: cannot borrow `*map` as mutable more than once at a time
  --> src/main.rs:12:13
   |
4  |   fn get_default<'m, K, V>(map: &'m mut HashMap<K, V>, key: K) -> &'m mut V
   |                  -- lifetime `'m` defined here
...
9  |       match map.get_mut(&key) {
   |       -     --- first mutable borrow occurs here
   |  _____|
   | |
10 | |         Some(value) => value,
11 | |         None => {
12 | |             map.insert(key.clone(), V::default());
   | |             ^^^ second mutable borrow occurs here
13 | |             map.get_mut(&key).unwrap()
14 | |         }
15 | |     }
   | |_____- returning this value requires that `*map` is borrowed for `'m`
```

Lifetime Elision

In order to make common patterns more ergonomic, Rust allows lifetimes to be *elided* in function signatures.

A *lifetime position* is anywhere you can write a lifetime in a type:

```
&'a T
&'a mut T
T<'a>
```

Lifetime positions can appear as either "input" or "output":

- For `fn` definitions, `fn` types, and the traits `Fn`, `FnMut`, and `FnOnce`, input refers to the types of the formal arguments, while output refers to result types. So `fn foo(s: &str) -> (&str, &str)` has elided one lifetime in input position and two lifetimes in output position. Note that the input positions of a `fn` method definition do not include the lifetimes that occur in the method's `impl` header (nor lifetimes that occur in the trait header, for a default method).
- For `impl` headers, all types are input. So `impl Trait<&T> for Struct<&T>` has elided two lifetimes in input position, while `impl Struct<&T>` has elided one.

Elision rules are as follows:

- Each elided lifetime in input position becomes a distinct lifetime parameter.
- If there is exactly one input lifetime position (elided or not), that lifetime is assigned to *all* elided output lifetimes.
- If there are multiple input lifetime positions, but one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to *all* elided output lifetimes.

- Otherwise, it is an error to elide an output lifetime.

Examples:

```
fn print(s: &str);                                    // elided
fn print<'a>(s: &'a str);                             // expanded

fn debug(lvl: usize, s: &str);                        // elided
fn debug<'a>(lvl: usize, s: &'a str);                 // expanded

fn substr(s: &str, until: usize) -> &str;             // elided
fn substr<'a>(s: &'a str, until: usize) -> &'a str;   // expanded

fn get_str() -> &str;                                 // ILLEGAL

fn frob(s: &str, t: &str) -> &str;                    // ILLEGAL

fn get_mut(&mut self) -> &mut T;                      // elided
fn get_mut<'a>(&'a mut self) -> &'a mut T;            // expanded

fn args<T: ToCStr>(&mut self, args: &[T]) -> &mut Command                  // elided
fn args<'a, 'b, T: ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command // expanded

fn new(buf: &mut [u8]) -> BufWriter;                  // elided
fn new(buf: &mut [u8]) -> BufWriter<'_>;              // elided (with
`rust_2018_idioms`)
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a>        // expanded
```

Unbounded Lifetimes

Unsafe code can often end up producing references or lifetimes out of thin air. Such lifetimes come into the world as *unbounded*. The most common source of this is dereferencing a raw pointer, which produces a reference with an unbounded lifetime. Such a lifetime becomes as big as context demands. This is in fact more powerful than simply becoming `'static`, because for instance `&'static &'a T` will fail to typecheck, but the unbound lifetime will perfectly mold into `&'a &'a T` as needed. However for most intents and purposes, such an unbounded lifetime can be regarded as `'static`.

Almost no reference is `'static`, so this is probably wrong. `transmute` and `transmute_copy` are the two other primary offenders. One should endeavor to bound an unbounded lifetime as quickly as possible, especially across function boundaries.

Given a function, any output lifetimes that don't derive from inputs are unbounded. For instance:

```
fn get_str<'a>() -> &'a str;
```

will produce an `&str` with an unbounded lifetime. The easiest way to avoid unbounded lifetimes is to use lifetime elision at the function boundary. If an output lifetime is elided, then it *must* be bounded by an input lifetime. Of course it might be bounded by the *wrong* lifetime, but this will usually just cause a compiler error, rather than allow memory safety to be trivially violated.

Within a function, bounding lifetimes is more error-prone. The safest and easiest way to bound a lifetime is to return it from a function with a bound lifetime. However if this is unacceptable, the reference can be placed in a location with a specific lifetime. Unfortunately it's impossible to name all lifetimes involved in a function.

Higher-Rank Trait Bounds (HRTBs)

Rust's `Fn` traits are a little bit magic. For instance, we can write the following code:

```
struct Closure<F> {
    data: (u8, u16),
    func: F,
}

impl<F> Closure<F>
    where F: Fn(&(u8, u16)) -> &u8,
{
    fn call(&self) -> &u8 {
        (self.func)(&self.data)
    }
}

fn do_it(data: &(u8, u16)) -> &u8 { &data.0 }

fn main() {
    let clo = Closure { data: (0, 1), func: do_it };
    println!("{}", clo.call());
}
```

If we try to naively desugar this code in the same way that we did in the lifetimes section, we run into some trouble:

```
// NOTE: `&'b data.0` and `'x: {` is not valid syntax!
struct Closure<F> {
    data: (u8, u16),
    func: F,
}

impl<F> Closure<F>
    // where F: Fn(&'??? (u8, u16)) -> &'??? u8,
{
    fn call<'a>(&'a self) -> &'a u8 {
        (self.func)(&self.data)
    }
}

fn do_it<'b>(data: &'b (u8, u16)) -> &'b u8 { &'b data.0 }

fn main() {
    'x: {
        let clo = Closure { data: (0, 1), func: do_it };
        println!("{}", clo.call());
    }
}
```

How on earth are we supposed to express the lifetimes on `F`'s trait bound? We need to provide some lifetime there, but the lifetime we care about can't be named until we enter the body of `call`! Also, that isn't some fixed lifetime; `call` works with *any* lifetime `&self` happens to have at that point.

This job requires The Magic of Higher-Rank Trait Bounds (HRTBs). The way we desugar this is as follows:

```
where for<'a> F: Fn(&'a (u8, u16)) -> &'a u8,
```

Alternatively:

```
where F: for<'a> Fn(&'a (u8, u16)) -> &'a u8,
```

(Where `Fn(a, b, c) -> d` is itself just sugar for the unstable *real* `Fn` trait)

`for<'a>` can be read as "for all choices of `'a`", and basically produces an *infinite list* of trait bounds that F must satisfy. Intense. There aren't many places outside of the `Fn` traits where we encounter HRTBs, and even for those we have a nice magic sugar for the common cases.

In summary, we can rewrite the original code more explicitly as:

```
struct Closure<F> {
    data: (u8, u16),
    func: F,
}

impl<F> Closure<F>
    where for<'a> F: Fn(&'a (u8, u16)) -> &'a u8,
{
    fn call(&self) -> &u8 {
        (self.func)(&self.data)
    }
}

fn do_it(data: &(u8, u16)) -> &u8 { &data.0 }

fn main() {
    let clo = Closure { data: (0, 1), func: do_it };
    println!("{}", clo.call());
}
```

Subtyping and Variance

Subtyping is a relationship between types that allows statically typed languages to be a bit more flexible and permissive.

Subtyping in Rust is a bit different from subtyping in other languages. This makes it harder to give simple examples, which is a problem since subtyping, and especially variance, is already hard to understand properly. As in, even compiler writers mess it up all the time.

To keep things simple, this section will consider a small extension to the Rust language that adds a new and simpler subtyping relationship. After establishing concepts and issues under this simpler system, we will then relate it back to how subtyping actually occurs in Rust.

So here's our simple extension, *Objective Rust*, featuring three new types:

```
trait Animal {
    fn snuggle(&self);
    fn eat(&mut self);
}

trait Cat: Animal {
    fn meow(&self);
}

trait Dog: Animal {
    fn bark(&self);
}
```

But unlike normal traits, we can use them as concrete and sized types, just like structs.

Now, say we have a very simple function that takes an Animal, like this:

```
fn love(pet: Animal) {
    pet.snuggle();
}
```

By default, static types must match *exactly* for a program to compile. As such, this code won't compile:

```
let mr_snuggles: Cat = ...;
love(mr_snuggles);          // ERROR: expected Animal, found Cat
```

Mr. Snuggles is a Cat, and Cats aren't *exactly* Animals, so we can't love him! 😿

This is annoying because Cats *are* Animals. They support every operation an Animal supports, so intuitively `love` shouldn't care if we pass it a `Cat`. We should be able to just **forget** the non-animal parts of our `Cat`, as they aren't necessary to love it.

This is exactly the problem that *subtyping* is intended to fix. Because Cats are just Animals **and more**, we say Cat is a *subtype* of Animal (because Cats are a *subset* of all the Animals). Equivalently, we say that Animal is a *supertype* of Cat. With subtypes, we can tweak our overly strict static type system with a simple rule: anywhere a value of type `T` is expected, we will also accept values that are subtypes of `T`.

Or more concretely: anywhere an Animal is expected, a Cat or Dog will also work.

As we will see throughout the rest of this section, subtyping is a lot more complicated and subtle than this, but this simple rule is a very good 99% intuition. And unless you write unsafe code, the compiler will automatically handle all the corner cases for you.

But this is the Rustonomicon. We're writing unsafe code, so we need to understand how this stuff really works, and how we can mess it up.

The core problem is that this rule, naively applied, will lead to *meowing Dogs*. That is, we can convince someone that a Dog is actually a Cat. This completely destroys the fabric of our static type system,

making it worse than useless (and leading to Undefined Behaviour).

Here's a simple example of this happening when we apply subtyping in a completely naive "find and replace" way.

```
fn evil_feeder(pet: &mut Animal) {
    let spike: Dog = ...;

    // `pet` is an Animal, and Dog is a subtype of Animal,
    // so this should be fine, right..?
    *pet = spike;
}

fn main() {
    let mut mr_snuggles: Cat = ...;
    evil_feeder(&mut mr_snuggles);  // Replaces mr_snuggles with a Dog
    mr_snuggles.meow();             // OH NO, MEOWING DOG!
}
```

Clearly, we need a more robust system than "find and replace". That system is *variance*, which is a set of rules governing how subtyping should compose. Most importantly, variance defines situations where subtyping should be disabled.

But before we get into variance, let's take a quick peek at where subtyping actually occurs in Rust: *lifetimes*!

> NOTE: The typed-ness of lifetimes is a fairly arbitrary construct that some disagree with. However it simplifies our analysis to treat lifetimes and types uniformly.

Lifetimes are just regions of code, and regions can be partially ordered with the *contains* (outlives) relationship. Subtyping on lifetimes is in terms of that relationship: if `'big: 'small` ("big contains small" or "big outlives small"), then `'big` is a subtype of `'small`. This is a large source of confusion, because it seems backwards to many: the bigger region is a *subtype* of the smaller region. But it

makes sense if you consider our Animal example: Cat is an Animal *and more*, just as `'big` is `'small` *and more*.

Put another way, if someone wants a reference that lives for `'small`, usually what they actually mean is that they want a reference that lives for *at least* `'small`. They don't actually care if the lifetimes match exactly. So it should be ok for us to **forget** that something lives for `'big` and only remember that it lives for `'small`.

The meowing dog problem for lifetimes will result in us being able to store a short-lived reference in a place that expects a longer-lived one, creating a dangling reference and letting us use-after-free.

It will be useful to note that `'static`, the forever lifetime, is a subtype of every lifetime because by definition it outlives everything. We will be using this relationship in later examples to keep them as simple as possible.

With all that said, we still have no idea how to actually *use* subtyping of lifetimes, because nothing ever has type `'a`. Lifetimes only occur as part of some larger type like `&'a u32` or `IterMut<'a, u32>`. To apply lifetime subtyping, we need to know how to compose subtyping. Once again, we need *variance*.

Variance

Variance is where things get a bit complicated.

Variance is a property that *type constructors* have with respect to their arguments. A type constructor in Rust is any generic type with unbound arguments. For instance `Vec` is a type constructor that takes a type `T` and returns `Vec<T>`. `&` and `&mut` are type constructors that take two inputs: a lifetime, and a type to point to.

NOTE: For convenience we will often refer to `F<T>` as a type constructor just so that we can easily talk about `T`. Hopefully this is clear in context.

A type constructor F's *variance* is how the subtyping of its inputs affects the subtyping of its outputs. There are three kinds of variance in Rust. Given two types `Sub` and `Super`, where `Sub` is a subtype of `Super`:

- `F` is *covariant* if `F<Sub>` is a subtype of `F<Super>` (subtyping "passes through")
- `F` is *contravariant* if `F<Super>` is a subtype of `F<Sub>` (subtyping is "inverted")
- `F` is *invariant* otherwise (no subtyping relationship exists)

If `F` has multiple type parameters, we can talk about the individual variances by saying that, for example, `F<T, U>` is covariant over `T` and invariant over `U`.

It is very useful to keep in mind that covariance is, in practical terms, "the" variance. Almost all consideration of variance is in terms of whether something should be covariant or invariant. Actually witnessing contravariance is quite difficult in Rust, though it does in fact exist.

Here is a table of important variances which the rest of this section will be devoted to trying to explain:

		'a	T	U
*	`&'a T`	covariant	covariant	
*	`&'a mut T`	covariant	invariant	
*	`Box<T>`		covariant	
	`Vec<T>`		covariant	
*	`UnsafeCell<T>`		invariant	
	`Cell<T>`		invariant	

		'a	T	U
*	`fn(T) -> U`		**contra**variant	covariant
	`*const T`		covariant	
	`*mut T`		invariant	

The types with *'s are the ones we will be focusing on, as they are in some sense "fundamental". All the others can be understood by analogy to the others:

- `Vec<T>` and all other owning pointers and collections follow the same logic as `Box<T>`
- `Cell<T>` and all other interior mutability types follow the same logic as `UnsafeCell<T>`
- `*const T` follows the logic of `&T`
- `*mut T` follows the logic of `&mut T` (or `UnsafeCell<T>`)

For more types, see the "Variance" section on the reference.

> NOTE: the *only* source of contravariance in the language is the arguments to a function, which is why it really doesn't come up much in practice. Invoking contravariance involves higher-order programming with function pointers that take references with specific lifetimes (as opposed to the usual "any lifetime", which gets into higher rank lifetimes, which work independently of subtyping).

Ok, that's enough type theory! Let's try to apply the concept of variance to Rust and look at some examples.

First off, let's revisit the meowing dog example:

```
fn evil_feeder(pet: &mut Animal) {
    let spike: Dog = ...;

    // `pet` is an Animal, and Dog is a subtype of Animal,
    // so this should be fine, right..?
    *pet = spike;
}

fn main() {
    let mut mr_snuggles: Cat = ...;
    evil_feeder(&mut mr_snuggles);  // Replaces mr_snuggles with a Dog
    mr_snuggles.meow();             // OH NO, MEOWING DOG!
}
```

If we look at our table of variances, we see that `&mut T` is *invariant* over `T`. As it turns out, this completely fixes the issue! With invariance, the fact that Cat is a subtype of Animal doesn't matter; `&mut Cat` still won't be a subtype of `&mut Animal`. The static type checker will then correctly stop us from passing a Cat into `evil_feeder`.

The soundness of subtyping is based on the idea that it's ok to forget unnecessary details. But with references, there's always someone that remembers those details: the value being referenced. That value expects those details to keep being true, and may behave incorrectly if its expectations are violated.

The problem with making `&mut T` covariant over `T` is that it gives us the power to modify the original value *when we don't remember all of its constraints*. And so, we can make someone have a Dog when they're certain they still have a Cat.

With that established, we can easily see why `&T` being covariant over `T` *is* sound: it doesn't let you modify the value, only look at it. Without any way to mutate, there's no way for us to mess with any details. We can also see why `UnsafeCell` and all the other interior mutability types must be invariant: they make `&T` work like `&mut T`!

Now what about the lifetime on references? Why is it ok for both kinds of references to be covariant

over their lifetimes? Well, here's a two-pronged argument:

First and foremost, subtyping references based on their lifetimes is *the entire point of subtyping in Rust*. The only reason we have subtyping is so we can pass long-lived things where short-lived things are expected. So it better work!

Second, and more seriously, lifetimes are only a part of the reference itself. The type of the referent is shared knowledge, which is why adjusting that type in only one place (the reference) can lead to issues. But if you shrink down a reference's lifetime when you hand it to someone, that lifetime information isn't shared in any way. There are now two independent references with independent lifetimes. There's no way to mess with original reference's lifetime using the other one.

Or rather, the only way to mess with someone's lifetime is to build a meowing dog. But as soon as you try to build a meowing dog, the lifetime should be wrapped up in an invariant type, preventing the lifetime from being shrunk. To understand this better, let's port the meowing dog problem over to real Rust.

In the meowing dog problem we take a subtype (Cat), convert it into a supertype (Animal), and then use that fact to overwrite the subtype with a value that satisfies the constraints of the supertype but not the subtype (Dog).

So with lifetimes, we want to take a long-lived thing, convert it into a short-lived thing, and then use that to write something that doesn't live long enough into the place expecting something long-lived.

Here it is:

```
fn evil_feeder<T>(input: &mut T, val: T) {
    *input = val;
}

fn main() {
    let mut mr_snuggles: &'static str = "meow! :3";  // mr. snuggles forever!!
    {
        let spike = String::from("bark! >:V");
        let spike_str: &str = &spike;                // Only lives for the block
        evil_feeder(&mut mr_snuggles, spike_str);    // EVIL!
    }
    println!("{}", mr_snuggles);                     // Use after free?
}
```

And what do we get when we run this?

```
error[E0597]: `spike` does not live long enough
  --> src/main.rs:9:31
   |
6  |     let mut mr_snuggles: &'static str = "meow! :3";  // mr. snuggles forever!!
   |                          ------------ type annotation requires that `spike` is
borrowed for `'static`
...
9  |         let spike_str: &str = &spike;                // Only lives for the block
   |                               ^^^^^^ borrowed value does not live long enough
10 |         evil_feeder(&mut mr_snuggles, spike_str);    // EVIL!
11 |     }
   |     - `spike` dropped here while still borrowed
```

Good, it doesn't compile! Let's break down what's happening here in detail.

First let's look at the new `evil_feeder` function:

```
fn evil_feeder<T>(input: &mut T, val: T) {
    *input = val;
}
```

All it does is take a mutable reference and a value and overwrite the referent with it. What's important about this function is that it creates a type equality constraint. It clearly says in its signature the referent and the value must be the *exact same* type.

Meanwhile, in the caller we pass in `&mut &'static str` and `&'spike_str str`.

Because `&mut T` is invariant over `T`, the compiler concludes it can't apply any subtyping to the first argument, and so `T` must be exactly `&'static str`.

The other argument is only an `&'a str`, which *is* covariant over `'a`. So the compiler adopts a constraint: `&'spike_str str` must be a subtype of `&'static str` (inclusive), which in turn implies `'spike_str` must be a subtype of `'static` (inclusive). Which is to say, `'spike_str` must contain `'static`. But only one thing contains `'static` -- `'static` itself!

This is why we get an error when we try to assign `&spike` to `spike_str`. The compiler has worked backwards to conclude `spike_str` must live forever, and `&spike` simply can't live that long.

So even though references are covariant over their lifetimes, they "inherit" invariance whenever they're put into a context that could do something bad with that. In this case, we inherited invariance as soon as we put our reference inside an `&mut T`.

As it turns out, the argument for why it's ok for Box (and Vec, Hashmap, etc.) to be covariant is pretty similar to the argument for why it's ok for lifetimes to be covariant: as soon as you try to stuff them in something like a mutable reference, they inherit invariance and you're prevented from doing anything bad.

However Box makes it easier to focus on by-value aspect of references that we partially glossed over.

Unlike a lot of languages which allow values to be freely aliased at all times, Rust has a very strict rule: if you're allowed to mutate or move a value, you are guaranteed to be the only one with access to it.

Consider the following code:

```
let mr_snuggles: Box<Cat> = ..;
let spike: Box<Dog> = ..;

let mut pet: Box<Animal>;
pet = mr_snuggles;
pet = spike;
```

There is no problem at all with the fact that we have forgotten that `mr_snuggles` was a Cat, or that we overwrote him with a Dog, because as soon as we moved mr_snuggles to a variable that only knew he was an Animal, **we destroyed the only thing in the universe that remembered he was a Cat**!

In contrast to the argument about immutable references being soundly covariant because they don't let you change anything, owned values can be covariant because they make you change *everything*. There is no connection between old locations and new locations. Applying by-value subtyping is an irreversible act of knowledge destruction, and without any memory of how things used to be, no one can be tricked into acting on that old information!

Only one thing left to explain: function pointers.

To see why `fn(T) -> U` should be covariant over `U`, consider the following signature:

```
fn get_animal() -> Animal;
```

This function claims to produce an Animal. As such, it is perfectly valid to provide a function with the following signature instead:

```
fn get_animal() -> Cat;
```

After all, Cats are Animals, so always producing a Cat is a perfectly valid way to produce Animals. Or to relate it back to real Rust: if we need a function that is supposed to produce something that lives

for `'short`, it's perfectly fine for it to produce something that lives for `'long`. We don't care, we can just forget that fact.

However, the same logic does not apply to *arguments*. Consider trying to satisfy:

```
fn handle_animal(Animal);
```

with:

```
fn handle_animal(Cat);
```

The first function can accept Dogs, but the second function absolutely can't. Covariance doesn't work here. But if we flip it around, it actually *does* work! If we need a function that can handle Cats, a function that can handle *any* Animal will surely work fine. Or to relate it back to real Rust: if we need a function that can handle anything that lives for at least `'long`, it's perfectly fine for it to be able to handle anything that lives for at least `'short`.

And that's why function types, unlike anything else in the language, are **contra**variant over their arguments.

Now, this is all well and good for the types the standard library provides, but how is variance determined for types that *you* define? A struct, informally speaking, inherits the variance of its fields. If a struct `MyType` has a generic argument `A` that is used in a field `a`, then MyType's variance over `A` is exactly `a`'s variance over `A`.

However if `A` is used in multiple fields:

- If all uses of `A` are covariant, then MyType is covariant over `A`
- If all uses of `A` are contravariant, then MyType is contravariant over `A`
- Otherwise, MyType is invariant over `A`

```
use std::cell::Cell;

struct MyType<'a, 'b, A: 'a, B: 'b, C, D, E, F, G, H, In, Out, Mixed> {
    a: &'a A,     // covariant over 'a and A
    b: &'b mut B, // covariant over 'b and invariant over B

    c: *const C,  // covariant over C
    d: *mut D,    // invariant over D

    e: E,         // covariant over E
    f: Vec<F>,    // covariant over F
    g: Cell<G>,   // invariant over G

    h1: H,        // would also be covariant over H except...
    h2: Cell<H>,  // invariant over H, because invariance wins all conflicts

    i: fn(In) -> Out,       // contravariant over In, covariant over Out

    k1: fn(Mixed) -> usize, // would be contravariant over Mixed except..
    k2: Mixed,              // invariant over Mixed, because invariance wins all
conflicts
}
```

Drop Check

We have seen how lifetimes provide us some fairly simple rules for ensuring that we never read dangling references. However up to this point we have only ever interacted with the *outlives* relationship in an inclusive manner. That is, when we talked about `'a: 'b`, it was ok for `'a` to live *exactly* as long as `'b`. At first glance, this seems to be a meaningless distinction. Nothing ever gets dropped at the same time as another, right? This is why we used the following desugaring of `let` statements:

```
let x;
let y;
```

desugaring to:

```
{
    let x;
    {
        let y;
    }
}
```

There are some more complex situations which are not possible to desugar using scopes, but the order is still defined - variables are dropped in the reverse order of their definition, fields of structs and tuples in order of their definition. There are some more details about order of drop in RFC 1857.

Let's do this:

```
let tuple = (vec![], vec![]);
```

The left vector is dropped first. But does it mean the right one strictly outlives it in the eyes of the borrow checker? The answer to this question is *no*. The borrow checker could track fields of tuples separately, but it would still be unable to decide what outlives what in case of vector elements, which

are dropped manually via pure-library code the borrow checker doesn't understand.

So why do we care? We care because if the type system isn't careful, it could accidentally make dangling pointers. Consider the following simple program:

```
struct Inspector<'a>(&'a u8);

struct World<'a> {
    inspector: Option<Inspector<'a>>,
    days: Box<u8>,
}

fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days));
}
```

This program is totally sound and compiles today. The fact that `days` does not strictly outlive `inspector` doesn't matter. As long as the `inspector` is alive, so is `days`.

However if we add a destructor, the program will no longer compile!

```
struct Inspector<'a>(&'a u8);

impl<'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("I was only {} days from retirement!", self.0);
    }
}

struct World<'a> {
    inspector: Option<Inspector<'a>>,
    days: Box<u8>,
}

fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days));
    // Let's say `days` happens to get dropped first.
    // Then when Inspector is dropped, it will try to read free'd memory!
}
```

```
error[E0597]: `world.days` does not live long enough
  --> src/main.rs:19:38
   |
19 |     world.inspector = Some(Inspector(&world.days));
   |                                      ^^^^^^^^^^ borrowed value does not live long
enough
...
22 | }
   | -
   | |
   | `world.days` dropped here while still borrowed
   | borrow might be used here, when `world` is dropped and runs the destructor for
type `World<'_>`
```

You can try changing the order of fields or use a tuple instead of the struct, it'll still not compile.

Implementing `Drop` lets the `Inspector` execute some arbitrary code during its death. This means it can potentially observe that types that are supposed to live as long as it does actually were destroyed first.

Interestingly, only generic types need to worry about this. If they aren't generic, then the only lifetimes they can harbor are `'static`, which will truly live *forever*. This is why this problem is referred to as *sound generic drop*. Sound generic drop is enforced by the *drop checker*. As of this writing, some of the finer details of how the drop checker (also called dropck) validates types is totally up in the air. However The Big Rule is the subtlety that we have focused on this whole section:

For a generic type to soundly implement drop, its generics arguments must strictly outlive it.

Obeying this rule is (usually) necessary to satisfy the borrow checker; obeying it is sufficient but not necessary to be sound. That is, if your type obeys this rule then it's definitely sound to drop.

The reason that it is not always necessary to satisfy the above rule is that some Drop implementations will not access borrowed data even though their type gives them the capability for such access, or because we know the specific drop order and the borrowed data is still fine even if the borrow checker doesn't know that.

For example, this variant of the above `Inspector` example will never access borrowed data:

```
struct Inspector<'a>(&'a u8, &'static str);

impl<'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("Inspector(_, {}) knows when *not* to inspect.", self.1);
    }
}

struct World<'a> {
    inspector: Option<Inspector<'a>>,
    days: Box<u8>,
}

fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days, "gadget"));
    // Let's say `days` happens to get dropped first.
    // Even when Inspector is dropped, its destructor will not access the
    // borrowed `days`.
}
```

Likewise, this variant will also never access borrowed data:

```
struct Inspector<T>(T, &'static str);

impl<T> Drop for Inspector<T> {
    fn drop(&mut self) {
        println!("Inspector(_, {}) knows when *not* to inspect.", self.1);
    }
}

struct World<T> {
    inspector: Option<Inspector<T>>,
    days: Box<u8>,
}

fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days, "gadget"));
    // Let's say `days` happens to get dropped first.
    // Even when Inspector is dropped, its destructor will not access the
    // borrowed `days`.
}
```

However, *both* of the above variants are rejected by the borrow checker during the analysis of `fn main`, saying that `days` does not live long enough.

The reason is that the borrow checking analysis of `main` does not know about the internals of each `Inspector`'s `Drop` implementation. As far as the borrow checker knows while it is analyzing `main`, the body of an inspector's destructor might access that borrowed data.

Therefore, the drop checker forces all borrowed data in a value to strictly outlive that value.

An Escape Hatch

The precise rules that govern drop checking may be less restrictive in the future.

The current analysis is deliberately conservative and trivial; it forces all borrowed data in a value to outlive that value, which is certainly sound.

Future versions of the language may make the analysis more precise, to reduce the number of cases where sound code is rejected as unsafe. This would help address cases such as the two `Inspector`s above that know not to inspect during destruction.

In the meantime, there is an unstable attribute that one can use to assert (unsafely) that a generic type's destructor is *guaranteed* to not access any expired data, even if its type gives it the capability to do so.

That attribute is called `may_dangle` and was introduced in RFC 1327. To deploy it on the `Inspector` from above, we would write:

```
#![feature(dropck_eyepatch)]

struct Inspector<'a>(&'a u8, &'static str);

unsafe impl<#[may_dangle] 'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("Inspector(_, {}) knows when *not* to inspect.", self.1);
    }
}

struct World<'a> {
    days: Box<u8>,
    inspector: Option<Inspector<'a>>,
}

fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days, "gatget"));
}
```

Use of this attribute requires the `Drop` impl to be marked `unsafe` because the compiler is not checking the implicit assertion that no potentially expired data (e.g. `self.0` above) is accessed.

The attribute can be applied to any number of lifetime and type parameters. In the following example, we assert that we access no data behind a reference of lifetime `'b` and that the only uses of `T` will be moves or drops, but omit the attribute from `'a` and `U`, because we do access data with that lifetime and that type:

```
#![feature(dropck_eyepatch)]
use std::fmt::Display;

struct Inspector<'a, 'b, T, U: Display>(&'a u8, &'b u8, T, U);

unsafe impl<'a, #[may_dangle] 'b, #[may_dangle] T, U: Display> Drop for Inspector<'a,
'b, T, U> {
    fn drop(&mut self) {
        println!("Inspector({}, _, _, {})", self.0, self.3);
    }
}
```

It is sometimes obvious that no such access can occur, like the case above. However, when dealing with a generic type parameter, such access can occur indirectly. Examples of such indirect access are:

- invoking a callback,
- via a trait method call.

(Future changes to the language, such as impl specialization, may add other avenues for such indirect access.)

Here is an example of invoking a callback:

```
struct Inspector<T>(T, &'static str, Box<for <'r> fn(&'r T) -> String>);

impl<T> Drop for Inspector<T> {
    fn drop(&mut self) {
        // The `self.2` call could access a borrow e.g. if `T` is `&'a _`.
        println!("Inspector({}, {}) unwittingly inspects expired data.",
                 (self.2)(&self.0), self.1);
    }
}
```

Here is an example of a trait method call:

```
use std::fmt;

struct Inspector<T: fmt::Display>(T, &'static str);

impl<T: fmt::Display> Drop for Inspector<T> {
    fn drop(&mut self) {
        // There is a hidden call to `<T as Display>::fmt` below, which
        // could access a borrow e.g. if `T` is `&'a _`
        println!("Inspector({}, {}) unwittingly inspects expired data.",
                 self.0, self.1);
    }
}
```

And of course, all of these accesses could be further hidden within some other method invoked by the destructor, rather than being written directly within it.

In all of the above cases where the `&'a u8` is accessed in the destructor, adding the `#[may_dangle]` attribute makes the type vulnerable to misuse that the borrow checker will not catch, inviting havoc. It is better to avoid adding the attribute.

A related side note about drop order

While the drop order of fields inside a struct is defined, relying on it is fragile and subtle. When the order matters, it is better to use the `ManuallyDrop` wrapper.

Is that all about drop checker?

It turns out that when writing unsafe code, we generally don't need to worry at all about doing the right thing for the drop checker. However there is one special case that you need to worry about, which we will look at in the next section.

PhantomData

When working with unsafe code, we can often end up in a situation where types or lifetimes are logically associated with a struct, but not actually part of a field. This most commonly occurs with lifetimes. For instance, the `Iter` for `&'a [T]` is (approximately) defined as follows:

```
struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
}
```

However because `'a` is unused within the struct's body, it's *unbounded*. Because of the troubles this has historically caused, unbounded lifetimes and types are *forbidden* in struct definitions. Therefore we must somehow refer to these types in the body. Correctly doing this is necessary to have correct variance and drop checking.

We do this using `PhantomData`, which is a special marker type. `PhantomData` consumes no space, but simulates a field of the given type for the purpose of static analysis. This was deemed to be less error-prone than explicitly telling the type-system the kind of variance that you want, while also providing other useful things such as the information needed by drop check.

Iter logically contains a bunch of `&'a T`s, so this is exactly what we tell the `PhantomData` to simulate:

```
use std::marker;

struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
    _marker: marker::PhantomData<&'a T>,
}
```

and that's it. The lifetime will be bounded, and your iterator will be covariant over `'a` and `T`. Everything Just Works.

Another important example is Vec, which is (approximately) defined as follows:

```
struct Vec<T> {
    data: *const T, // *const for variance!
    len: usize,
    cap: usize,
}
```

Unlike the previous example, it *appears* that everything is exactly as we want. Every generic argument to Vec shows up in at least one field. Good to go!

Nope.

The drop checker will generously determine that `Vec<T>` does not own any values of type T. This will in turn make it conclude that it doesn't need to worry about Vec dropping any T's in its destructor for determining drop check soundness. This will in turn allow people to create unsoundness using Vec's destructor.

In order to tell the drop checker that we *do* own values of type T, and therefore may drop some T's when *we* drop, we must add an extra `PhantomData` saying exactly that:

```
use std::marker;

struct Vec<T> {
    data: *const T, // *const for variance!
    len: usize,
    cap: usize,
    _marker: marker::PhantomData<T>,
}
```

Raw pointers that own an allocation is such a pervasive pattern that the standard library made a utility for itself called `Unique<T>` which:

- wraps a `*const T` for variance
- includes a `PhantomData<T>`
- auto-derives `Send` / `Sync` as if T was contained
- marks the pointer as `NonZero` for the null-pointer optimization

Table of `PhantomData` patterns

Here's a table of all the wonderful ways `PhantomData` could be used:

Phantom type	`'a`	`T`
`PhantomData<T>`	-	covariant (with drop check)
`PhantomData<&'a T>`	covariant	covariant
`PhantomData<&'a mut T>`	covariant	invariant
`PhantomData<*const T>`	-	covariant
`PhantomData<*mut T>`	-	invariant
`PhantomData<fn(T)>`	-	contravariant
`PhantomData<fn() -> T>`	-	covariant
`PhantomData<fn(T) -> T>`	-	invariant
`PhantomData<Cell<&'a ()>>`	invariant	-

Splitting Borrows

The mutual exclusion property of mutable references can be very limiting when working with a composite structure. The borrow checker understands some basic stuff, but will fall over pretty easily. It does understand structs sufficiently to know that it's possible to borrow disjoint fields of a struct simultaneously. So this works today:

```
struct Foo {
    a: i32,
    b: i32,
    c: i32,
}

let mut x = Foo {a: 0, b: 0, c: 0};
let a = &mut x.a;
let b = &mut x.b;
let c = &x.c;
*b += 1;
let c2 = &x.c;
*a += 10;
println!("{} {} {} {}", a, b, c, c2);
```

However borrowck doesn't understand arrays or slices in any way, so this doesn't work:

```
let mut x = [1, 2, 3];
let a = &mut x[0];
let b = &mut x[1];
println!("{} {}", a, b);
```

```
error[E0499]: cannot borrow `x[..]` as mutable more than once at a time
 --> src/lib.rs:4:18
  |
3 |     let a = &mut x[0];
  |                 ---- first mutable borrow occurs here
4 |     let b = &mut x[1];
  |                 ^^^^ second mutable borrow occurs here
5 |     println!("{} {}", a, b);
6 | }
  | - first borrow ends here

error: aborting due to previous error
```

While it was plausible that borrowck could understand this simple case, it's pretty clearly hopeless for borrowck to understand disjointness in general container types like a tree, especially if distinct keys actually *do* map to the same value.

In order to "teach" borrowck that what we're doing is ok, we need to drop down to unsafe code. For instance, mutable slices expose a `split_at_mut` function that consumes the slice and returns two mutable slices. One for everything to the left of the index, and one for everything to the right. Intuitively we know this is safe because the slices don't overlap, and therefore alias. However the implementation requires some unsafety:

```
pub fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T]) {
    let len = self.len();
    let ptr = self.as_mut_ptr();

    unsafe {
        assert!(mid <= len);

        (from_raw_parts_mut(ptr, mid),
         from_raw_parts_mut(ptr.add(mid), len - mid))
    }
}
```

This is actually a bit subtle. So as to avoid ever making two `&mut` 's to the same value, we explicitly construct brand-new slices through raw pointers.

However more subtle is how iterators that yield mutable references work. The iterator trait is defined as follows:

```
trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}
```

Given this definition, Self::Item has *no* connection to `self` . This means that we can call `next` several times in a row, and hold onto all the results *concurrently*. This is perfectly fine for by-value iterators, which have exactly these semantics. It's also actually fine for shared references, as they admit arbitrarily many references to the same thing (although the iterator needs to be a separate object from the thing being shared).

But mutable references make this a mess. At first glance, they might seem completely incompatible with this API, as it would produce multiple mutable references to the same object!

However it actually *does* work, exactly because iterators are one-shot objects. Everything an IterMut yields will be yielded at most once, so we don't actually ever yield multiple mutable references to the same piece of data.

Perhaps surprisingly, mutable iterators don't require unsafe code to be implemented for many types!

For instance here's a singly linked list:

```
type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}

pub struct LinkedList<T> {
    head: Link<T>,
}

pub struct IterMut<'a, T: 'a>(Option<&'a mut Node<T>>);

impl<T> LinkedList<T> {
    fn iter_mut(&mut self) -> IterMut<T> {
        IterMut(self.head.as_mut().map(|node| &mut **node))
    }
}

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> {
        self.0.take().map(|node| {
            self.0 = node.next.as_mut().map(|node| &mut **node);
            &mut node.elem
        })
    }
}
```

Here's a mutable slice:

```
use std::mem;

pub struct IterMut<'a, T: 'a>(&'a mut[T]);

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> {
        let slice = mem::replace(&mut self.0, &mut []);
        if slice.is_empty() { return None; }

        let (l, r) = slice.split_at_mut(1);
        self.0 = r;
        l.get_mut(0)
    }
}

impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        let slice = mem::replace(&mut self.0, &mut []);
        if slice.is_empty() { return None; }

        let new_len = slice.len() - 1;
        let (l, r) = slice.split_at_mut(new_len);
        self.0 = l;
        r.get_mut(0)
    }
}
```

And here's a binary tree:

```
use std::collections::VecDeque;

type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    left: Link<T>,
    right: Link<T>,
}

pub struct Tree<T> {
    root: Link<T>,
}

struct NodeIterMut<'a, T: 'a> {
    elem: Option<&'a mut T>,
    left: Option<&'a mut Node<T>>,
    right: Option<&'a mut Node<T>>,
}

enum State<'a, T: 'a> {
    Elem(&'a mut T),
    Node(&'a mut Node<T>),
}

pub struct IterMut<'a, T: 'a>(VecDeque<NodeIterMut<'a, T>>);

impl<T> Tree<T> {
    pub fn iter_mut(&mut self) -> IterMut<T> {
        let mut deque = VecDeque::new();
        self.root.as_mut().map(|root| deque.push_front(root.iter_mut()));
        IterMut(deque)
    }
}

impl<T> Node<T> {
    pub fn iter_mut(&mut self) -> NodeIterMut<T> {
        NodeIterMut {
            elem: Some(&mut self.elem),
```

```
            left: self.left.as_mut().map(|node| &mut **node),
            right: self.right.as_mut().map(|node| &mut **node),
        }
    }
}


impl<'a, T> Iterator for NodeIterMut<'a, T> {
    type Item = State<'a, T>;

    fn next(&mut self) -> Option<Self::Item> {
        match self.left.take() {
            Some(node) => Some(State::Node(node)),
            None => match self.elem.take() {
                Some(elem) => Some(State::Elem(elem)),
                None => match self.right.take() {
                    Some(node) => Some(State::Node(node)),
                    None => None,
                }
            }
        }
    }
}

impl<'a, T> DoubleEndedIterator for NodeIterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        match self.right.take() {
            Some(node) => Some(State::Node(node)),
            None => match self.elem.take() {
                Some(elem) => Some(State::Elem(elem)),
                None => match self.left.take() {
                    Some(node) => Some(State::Node(node)),
                    None => None,
                }
            }
        }
    }
}

impl<'a, T> Iterator for IterMut<'a, T> {
```

```
    type Item = &'a mut T;
    fn next(&mut self) -> Option<Self::Item> {
        loop {
            match self.0.front_mut().and_then(|node_it| node_it.next()) {
                Some(State::Elem(elem)) => return Some(elem),
                Some(State::Node(node)) => self.0.push_front(node.iter_mut()),
                None => if let None = self.0.pop_front() { return None },
            }
        }
    }
}

impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        loop {
            match self.0.back_mut().and_then(|node_it| node_it.next_back()) {
                Some(State::Elem(elem)) => return Some(elem),
                Some(State::Node(node)) => self.0.push_back(node.iter_mut()),
                None => if let None = self.0.pop_back() { return None },
            }
        }
    }
}
```

All of these are completely safe and work on stable Rust! This ultimately falls out of the simple struct case we saw before: Rust understands that you can safely split a mutable reference into subfields. We can then encode permanently consuming a reference via Options (or in the case of slices, replacing with an empty slice).

Type Conversions

At the end of the day, everything is just a pile of bits somewhere, and type systems are just there to help us use those bits right. There are two common problems with typing bits: needing to reinterpret those exact bits as a different type, and needing to change the bits to have equivalent meaning for a different type. Because Rust encourages encoding important properties in the type system, these problems are incredibly pervasive. As such, Rust consequently gives you several ways to solve them.

First we'll look at the ways that Safe Rust gives you to reinterpret values. The most trivial way to do this is to just destructure a value into its constituent parts and then build a new type out of them. e.g.

```
struct Foo {
    x: u32,
    y: u16,
}

struct Bar {
    a: u32,
    b: u16,
}

fn reinterpret(foo: Foo) -> Bar {
    let Foo { x, y } = foo;
    Bar { a: x, b: y }
}
```

But this is, at best, annoying. For common conversions, Rust provides more ergonomic alternatives.

Coercions

Types can implicitly be coerced to change in certain contexts. These changes are generally just *weakening* of types, largely focused around pointers and lifetimes. They mostly exist to make Rust "just work" in more cases, and are largely harmless.

For an exhaustive list of all the types of coercions, see the Coercion types section on the reference.

Note that we do not perform coercions when matching traits (except for receivers, see the next page). If there is an `impl` for some type `U` and `T` coerces to `U`, that does not constitute an implementation for `T`. For example, the following will not type check, even though it is OK to coerce `t` to `&T` and there is an `impl` for `&T`:

```
trait Trait {}

fn foo<X: Trait>(t: X) {}

impl<'a> Trait for &'a i32 {}

fn main() {
    let t: &mut i32 = &mut 0;
    foo(t);
}
```

which fails like as follows:

```
error[E0277]: the trait bound `&mut i32: Trait` is not satisfied
 --> src/main.rs:9:9
  |
3 | fn foo<X: Trait>(t: X) {}
  |           ----- required by this bound in `foo`
...
9 |     foo(t);
  |         ^ the trait `Trait` is not implemented for `&mut i32`
  |
  = help: the following implementations were found:
            <&'a i32 as Trait>
  = note: `Trait` is implemented for `&i32`, but not for `&mut i32`
```

The Dot Operator

The dot operator will perform a lot of magic to convert types. It will perform auto-referencing, auto-dereferencing, and coercion until types match. The detailed mechanics of method lookup are defined here, but here is a brief overview that outlines the main steps.

Suppose we have a function `foo` that has a receiver (a `self`, `&self` or `&mut self` parameter). If we call `value.foo()`, the compiler needs to determine what type `Self` is before it can call the correct implementation of the function. For this example, we will say that `value` has type `T`.

We will use fully-qualified syntax to be more clear about exactly which type we are calling a function on.

- First, the compiler checks if it can call `T::foo(value)` directly. This is called a "by value" method call.
- If it can't call this function (for example, if the function has the wrong type or a trait isn't implemented for `Self`), then the compiler tries to add in an automatic reference. This means that the compiler tries `<&T>::foo(value)` and `<&mut T>::foo(value)`. This is called an "autoref" method call.
- If none of these candidates worked, it dereferences `T` and tries again. This uses the `Deref` trait - if `T: Deref<Target = U>` then it tries again with type `U` instead of `T`. If it can't dereference `T`, it can also try *unsizing* `T`. This just means that if `T` has a size parameter known at compile time, it "forgets" it for the purpose of resolving methods. For instance, this unsizing step can convert `[i32; 2]` into `[i32]` by "forgetting" the size of the array.

Here is an example of the method lookup algorithm:

```
let array: Rc<Box<[T; 3]>> = ...;
let first_entry = array[0];
```

How does the compiler actually compute `array[0]` when the array is behind so many indirections?

First, `array[0]` is really just syntax sugar for the `Index` trait - the compiler will convert `array[0]` into `array.index(0)`. Now, the compiler checks to see if `array` implements `Index`, so that it can call the function.

Then, the compiler checks if `Rc<Box<[T; 3]>>` implements `Index`, but it does not, and neither do `&Rc<Box<[T; 3]>>` or `&mut Rc<Box<[T; 3]>>`. Since none of these worked, the compiler dereferences the `Rc<Box<[T; 3]>>` into `Box<[T; 3]>` and tries again. `Box<[T; 3]>`, `&Box<[T; 3]>`, and `&mut Box<[T; 3]>` do not implement `Index`, so it dereferences again. `[T; 3]` and its autorefs also do not implement `Index`. It can't dereference `[T; 3]`, so the compiler unsizes it, giving `[T]`. Finally, `[T]` implements `Index`, so it can now call the actual `index` function.

Consider the following more complicated example of the dot operator at work:

```
fn do_stuff<T: Clone>(value: &T) {
    let cloned = value.clone();
}
```

What type is `cloned`? First, the compiler checks if it can call by value. The type of `value` is `&T`, and so the `clone` function has signature `fn clone(&T) -> T`. It knows that `T: Clone`, so the compiler finds that `cloned: T`.

What would happen if the `T: Clone` restriction was removed? It would not be able to call by value, since there is no implementation of `Clone` for `T`. So the compiler tries to call by autoref. In this case, the function has the signature `fn clone(&&T) -> &T` since `Self = &T`. The compiler sees that `&T: Clone`, and then deduces that `cloned: &T`.

Here is another example where the autoref behavior is used to create some subtle effects:

```
#[derive(Clone)]
struct Container<T>(Arc<T>);

fn clone_containers<T>(foo: &Container<i32>, bar: &Container<T>) {
    let foo_cloned = foo.clone();
    let bar_cloned = bar.clone();
}
```

What types are `foo_cloned` and `bar_cloned`? We know that `Container<i32>: Clone`, so the compiler calls `clone` by value to give `foo_cloned: Container<i32>`. However, `bar_cloned` actually has type `&Container<T>`. Surely this doesn't make sense - we added `#[derive(Clone)]` to `Container`, so it must implement `Clone`! Looking closer, the code generated by the `derive` macro is (roughly):

```
impl<T> Clone for Container<T> where T: Clone {
    fn clone(&self) -> Self {
        Self(Arc::clone(&self.0))
    }
}
```

The derived `Clone` implementation is only defined where `T: Clone`, so there is no implementation for `Container<T>: Clone` for a generic `T`. The compiler then looks to see if `&Container<T>` implements `Clone`, which it does. So it deduces that `clone` is called by autoref, and so `bar_cloned` has type `&Container<T>`.

We can fix this by implementing `Clone` manually without requiring `T: Clone`:

```
impl<T> Clone for Container<T> {
    fn clone(&self) -> Self {
        Self(Arc::clone(&self.0))
    }
}
```

Now, the type checker deduces that `bar_cloned: Container<T>`.

Casts

Casts are a superset of coercions: every coercion can be explicitly invoked via a cast. However some conversions require a cast. While coercions are pervasive and largely harmless, these "true casts" are rare and potentially dangerous. As such, casts must be explicitly invoked using the `as` keyword: `expr as Type`.

You can find an exhaustive list of all the true casts and casting semantics on the reference.

Safety of casting

True casts generally revolve around raw pointers and the primitive numeric types. Even though they're dangerous, these casts are infallible at runtime. If a cast triggers some subtle corner case no indication will be given that this occurred. The cast will simply succeed. That said, casts must be valid at the type level, or else they will be prevented statically. For instance, `7u8 as bool` will not compile.

That said, casts aren't `unsafe` because they generally can't violate memory safety *on their own*. For instance, converting an integer to a raw pointer can very easily lead to terrible things. However the act of creating the pointer itself is safe, because actually using a raw pointer is already marked as `unsafe`.

Some notes about casting

Lengths when casting raw slices

Note that lengths are not adjusted when casting raw slices; `*const [u16] as *const [u8]` creates a slice that only includes half of the original memory.

Transitivity

Casting is not transitive, that is, even if `e as U1 as U2` is a valid expression, `e as U2` is not necessarily so.

Transmutes

Get out of our way type system! We're going to reinterpret these bits or die trying! Even though this book is all about doing things that are unsafe, I really can't emphasize that you should deeply think about finding Another Way than the operations covered in this section. This is really, truly, the most horribly unsafe thing you can do in Rust. The guardrails here are dental floss.

`mem::transmute<T, U>` takes a value of type `T` and reinterprets it to have type `U`. The only restriction is that the `T` and `U` are verified to have the same size. The ways to cause Undefined Behavior with this are mind boggling.

- First and foremost, creating an instance of *any* type with an invalid state is going to cause arbitrary chaos that can't really be predicted. Do not transmute `3` to `bool`. Even if you never *do* anything with the `bool`. Just don't.
- Transmute has an overloaded return type. If you do not specify the return type it may produce a surprising type to satisfy inference.
- Transmuting an `&` to `&mut` is Undefined Behavior. While certain usages may *appear* safe, note that the Rust optimizer is free to assume that a shared reference won't change through its lifetime and thus such transmutation will run afoul of those assumptions. So:
    - Transmuting an `&` to `&mut` is *always* Undefined Behavior.
    - No you can't do it.
    - No you're not special.
- Transmuting to a reference without an explicitly provided lifetime produces an unbounded lifetime.
- When transmuting between different compound types, you have to make sure they are laid out the same way! If layouts differ, the wrong fields are going to get filled with the wrong data, which will make you unhappy and can also be Undefined Behavior (see above).

> So how do you know if the layouts are the same? For `repr(C)` types and `repr(transparent)` types, layout is precisely defined. But for your run-of-the-mill `repr(Rust)`, it is not. Even different instances of the same generic type can have wildly different layout. `Vec<i32>` and `Vec<u32>` *might* have their fields in the same order, or they might not. The details of what exactly is and is not guaranteed for data layout are still being worked out over at the UCG WG.

`mem::transmute_copy<T, U>` somehow manages to be *even more* wildly unsafe than this. It copies `size_of<U>` bytes out of an `&T` and interprets them as a `U`. The size check that `mem::transmute` has is gone (as it may be valid to copy out a prefix), though it is Undefined Behavior for `U` to be larger than `T`.

Also of course you can get all of the functionality of these functions using raw pointer casts or `union`s, but without any of the lints or other basic sanity checks. Raw pointer casts and `union`s do not magically avoid the above rules.

Working With Uninitialized Memory

All runtime-allocated memory in a Rust program begins its life as *uninitialized*. In this state the value of the memory is an indeterminate pile of bits that may or may not even reflect a valid state for the type that is supposed to inhabit that location of memory. Attempting to interpret this memory as a value of *any* type will cause Undefined Behavior. Do Not Do This.

Rust provides mechanisms to work with uninitialized memory in checked (safe) and unchecked (unsafe) ways.

Checked Uninitialized Memory

Like C, all stack variables in Rust are uninitialized until a value is explicitly assigned to them. Unlike C, Rust statically prevents you from ever reading them until you do:

```
fn main() {
    let x: i32;
    println!("{}", x);
}
```

```
  |
3 |     println!("{}", x);
  |                    ^ use of possibly uninitialized `x`
```

This is based off of a basic branch analysis: every branch must assign a value to `x` before it is first used. For short, we also say that "`x` is init" or "`x` is uninit".

Interestingly, Rust doesn't require the variable to be mutable to perform a delayed initialization if every branch assigns exactly once. However the analysis does not take advantage of constant analysis or anything like that. So this compiles:

```
fn main() {
    let x: i32;

    if true {
        x = 1;
    } else {
        x = 2;
    }

    println!("{}", x);
}
```

but this doesn't:

```
fn main() {
    let x: i32;
    if true {
        x = 1;
    }
    println!("{}", x);
}
```

```
  |
6 |     println!("{}", x);
  |                    ^ use of possibly uninitialized `x`
```

while this does:

```
fn main() {
    let x: i32;
    if true {
        x = 1;
        println!("{}", x);
    }
    // Don't care that there are branches where it's not initialized
    // since we don't use the value in those branches
}
```

Of course, while the analysis doesn't consider actual values, it does have a relatively sophisticated understanding of dependencies and control flow. For instance, this works:

```
let x: i32;

loop {
    // Rust doesn't understand that this branch will be taken unconditionally,
    // because it relies on actual values.
    if true {
        // But it does understand that it will only be taken once because
        // we unconditionally break out of it. Therefore `x` doesn't
        // need to be marked as mutable.
        x = 0;
        break;
    }
}
// It also knows that it's impossible to get here without reaching the break.
// And therefore that `x` must be initialized here!
println!("{}", x);
```

If a value is moved out of a variable, that variable becomes logically uninitialized if the type of the value isn't Copy. That is:

```
fn main() {
    let x = 0;
    let y = Box::new(0);
    let z1 = x; // x is still valid because i32 is Copy
    let z2 = y; // y is now logically uninitialized because Box isn't Copy
}
```

However reassigning `y` in this example *would* require `y` to be marked as mutable, as a Safe Rust program could observe that the value of `y` changed:

```
fn main() {
    let mut y = Box::new(0);
    let z = y; // y is now logically uninitialized because Box isn't Copy
    y = Box::new(1); // reinitialize y
}
```

Otherwise it's like `y` is a brand new variable.

Drop Flags

The examples in the previous section introduce an interesting problem for Rust. We have seen that it's possible to conditionally initialize, deinitialize, and reinitialize locations of memory totally safely. For Copy types, this isn't particularly notable since they're just a random pile of bits. However types with destructors are a different story: Rust needs to know whether to call a destructor whenever a variable is assigned to, or a variable goes out of scope. How can it do this with conditional initialization?

Note that this is not a problem that all assignments need worry about. In particular, assigning through a dereference unconditionally drops, and assigning in a `let` unconditionally doesn't drop:

```
let mut x = Box::new(0); // let makes a fresh variable, so never need to drop
let y = &mut x;
*y = Box::new(1); // Deref assumes the referent is initialized, so always drops
```

This is only a problem when overwriting a previously initialized variable or one of its subfields.

It turns out that Rust actually tracks whether a type should be dropped or not *at runtime*. As a variable becomes initialized and uninitialized, a *drop flag* for that variable is toggled. When a variable might need to be dropped, this flag is evaluated to determine if it should be dropped.

Of course, it is often the case that a value's initialization state can be statically known at every point in the program. If this is the case, then the compiler can theoretically generate more efficient code! For instance, straight- line code has such *static drop semantics*:

```
let mut x = Box::new(0); // x was uninit; just overwrite.
let mut y = x;           // y was uninit; just overwrite and make x uninit.
x = Box::new(0);         // x was uninit; just overwrite.
y = x;                   // y was init; Drop y, overwrite it, and make x uninit!
                         // y goes out of scope; y was init; Drop y!
                         // x goes out of scope; x was uninit; do nothing.
```

Similarly, branched code where all branches have the same behavior with respect to initialization has static drop semantics:

```
let mut x = Box::new(0);    // x was uninit; just overwrite.
if condition {
    drop(x)                 // x gets moved out; make x uninit.
} else {
    println!("{}", x);
    drop(x)                 // x gets moved out; make x uninit.
}
x = Box::new(0);            // x was uninit; just overwrite.
                            // x goes out of scope; x was init; Drop x!
```

However code like this *requires* runtime information to correctly Drop:

```
let x;
if condition {
    x = Box::new(0);        // x was uninit; just overwrite.
    println!("{}", x);
}
                            // x goes out of scope; x might be uninit;
                            // check the flag!
```

Of course, in this case it's trivial to retrieve static drop semantics:

```
if condition {
    let x = Box::new(0);
    println!("{}", x);
}
```

The drop flags are tracked on the stack. In old Rust versions, drop flags were stashed in a hidden field of types that implement `Drop`.

Unchecked Uninitialized Memory

One interesting exception to this rule is working with arrays. Safe Rust doesn't permit you to partially initialize an array. When you initialize an array, you can either set every value to the same thing with `let x = [val; N]`, or you can specify each member individually with `let x = [val1, val2, val3]`. Unfortunately this is pretty rigid, especially if you need to initialize your array in a more incremental or dynamic way.

Unsafe Rust gives us a powerful tool to handle this problem: `MaybeUninit`. This type can be used to handle memory that has not been fully initialized yet.

With `MaybeUninit`, we can initialize an array element-for-element as follows:

```
use std::mem::{self, MaybeUninit};

// Size of the array is hard-coded but easy to change (meaning, changing just
// the constant is sufficient). This means we can't use [a, b, c] syntax to
// initialize the array, though, as we would have to keep that in sync
// with `SIZE`!
const SIZE: usize = 10;

let x = {
    // Create an uninitialized array of `MaybeUninit`. The `assume_init` is
    // safe because the type we are claiming to have initialized here is a
    // bunch of `MaybeUninit`s, which do not require initialization.
    let mut x: [MaybeUninit<Box<u32>>; SIZE] = unsafe {
        MaybeUninit::uninit().assume_init()
    };

    // Dropping a `MaybeUninit` does nothing. Thus using raw pointer
    // assignment instead of `ptr::write` does not cause the old
    // uninitialized value to be dropped.
    // Exception safety is not a concern because Box can't panic
    for i in 0..SIZE {
        x[i] = MaybeUninit::new(Box::new(i as u32));
    }

    // Everything is initialized. Transmute the array to the
    // initialized type.
    unsafe { mem::transmute::<_, [Box<u32>; SIZE]>(x) }
};

dbg!(x);
```

This code proceeds in three steps:

1. Create an array of `MaybeUninit<T>`. With current stable Rust, we have to use unsafe code for this: we take some uninitialized piece of memory (`MaybeUninit::uninit()`) and claim we have fully initialized it (`assume_init()`). This seems ridiculous, because we didn't! The reason this is

correct is that the array consists itself entirely of `MaybeUninit`, which do not actually require initialization. For most other types, doing `MaybeUninit::uninit().assume_init()` produces an invalid instance of said type, so you got yourself some Undefined Behavior.

2. Initialize the array. The subtle aspect of this is that usually, when we use `=` to assign to a value that the Rust type checker considers to already be initialized (like `x[i]`), the old value stored on the left-hand side gets dropped. This would be a disaster. However, in this case, the type of the left-hand side is `MaybeUninit<Box<u32>>`, and dropping that does not do anything! See below for some more discussion of this `drop` issue.

3. Finally, we have to change the type of our array to remove the `MaybeUninit`. With current stable Rust, this requires a `transmute`. This transmute is legal because in memory, `MaybeUninit<T>` looks the same as `T`.

   However, note that in general, `Container<MaybeUninit<T>>>` does *not* look the same as `Container<T>`! Imagine if `Container` was `Option`, and `T` was `bool`, then `Option<bool>` exploits that `bool` only has two valid values, but `Option<MaybeUninit<bool>>` cannot do that because the `bool` does not have to be initialized.

   So, it depends on `Container` whether transmuting away the `MaybeUninit` is allowed. For arrays, it is (and eventually the standard library will acknowledge that by providing appropriate methods).

It's worth spending a bit more time on the loop in the middle, and in particular the assignment operator and its interaction with `drop`. If we would have written something like:

```
*x[i].as_mut_ptr() = Box::new(i as u32); // WRONG!
```

we would actually overwrite a `Box<u32>`, leading to `drop` of uninitialized data, which will cause much sadness and pain.

The correct alternative, if for some reason we cannot use `MaybeUninit::new`, is to use the `ptr`

module. In particular, it provides three functions that allow us to assign bytes to a location in memory without dropping the old value: `write`, `copy`, and `copy_nonoverlapping`.

- `ptr::write(ptr, val)` takes a `val` and moves it into the address pointed to by `ptr`.
- `ptr::copy(src, dest, count)` copies the bits that `count` T's would occupy from src to dest. (this is equivalent to C's memmove -- note that the argument order is reversed!)
- `ptr::copy_nonoverlapping(src, dest, count)` does what `copy` does, but a little faster on the assumption that the two ranges of memory don't overlap. (this is equivalent to C's memcpy -- note that the argument order is reversed!)

It should go without saying that these functions, if misused, will cause serious havoc or just straight up Undefined Behavior. The only things that these functions *themselves* require is that the locations you want to read and write are allocated and properly aligned. However, the ways writing arbitrary bits to arbitrary locations of memory can break things are basically uncountable!

It's worth noting that you don't need to worry about `ptr::write`-style shenanigans with types which don't implement `Drop` or contain `Drop` types, because Rust knows not to try to drop them. This is what we relied on in the above example.

However when working with uninitialized memory you need to be ever-vigilant for Rust trying to drop values you make like this before they're fully initialized. Every control path through that variable's scope must initialize the value before it ends, if it has a destructor. *This includes code panicking*. `MaybeUninit` helps a bit here, because it does not implicitly drop its content - but all this really means in case of a panic is that instead of a double-free of the not yet initialized parts, you end up with a memory leak of the already initialized parts.

Note that, to use the `ptr` methods, you need to first obtain a *raw pointer* to the data you want to initialize. It is illegal to construct a *reference* to uninitialized data, which implies that you have to be careful when obtaining said raw pointer:

- For an array of `T`, you can use `base_ptr.add(idx)` where `base_ptr: *mut T` to compute the address of array index `idx`. This relies on how arrays are laid out in memory.

- For a struct, however, in general we do not know how it is laid out, and we also cannot use `&mut base_ptr.field` as that would be creating a reference. So, you must carefully use the `addr_of_mut` macro. This creates a raw pointer to the field without creating an intermediate reference:

```
use std::{ptr, mem::MaybeUninit};

struct Demo {
    field: bool,
}

let mut uninit = MaybeUninit::<Demo>::uninit();
// `&uninit.as_mut().field` would create a reference to an uninitialized `bool`,
// and thus be Undefined Behavior!
let f1_ptr = unsafe { ptr::addr_of_mut!((*uninit.as_mut_ptr()).field) };
unsafe { f1_ptr.write(true); }

let init = unsafe { uninit.assume_init() };
```

One last remark: when reading old Rust code, you might stumble upon the deprecated `mem::uninitialized` function. That function used to be the only way to deal with uninitialized memory on the stack, but it turned out to be impossible to properly integrate with the rest of the language. Always use `MaybeUninit` instead in new code, and port old code over when you get the opportunity.

And that's about it for working with uninitialized memory! Basically nothing anywhere expects to be handed uninitialized memory, so if you're going to pass it around at all, be sure to be *really* careful.

The Perils Of Ownership Based Resource Management (OBRM)

OBRM (AKA RAII: Resource Acquisition Is Initialization) is something you'll interact with a lot in Rust. Especially if you use the standard library.

Roughly speaking the pattern is as follows: to acquire a resource, you create an object that manages it. To release the resource, you simply destroy the object, and it cleans up the resource for you. The most common "resource" this pattern manages is simply *memory*. `Box`, `Rc`, and basically everything in `std::collections` is a convenience to enable correctly managing memory. This is particularly important in Rust because we have no pervasive GC to rely on for memory management. Which is the point, really: Rust is about control. However we are not limited to just memory. Pretty much every other system resource like a thread, file, or socket is exposed through this kind of API.

Constructors

There is exactly one way to create an instance of a user-defined type: name it, and initialize all its fields at once:

```
struct Foo {
    a: u8,
    b: u32,
    c: bool,
}

enum Bar {
    X(u32),
    Y(bool),
}

struct Unit;

let foo = Foo { a: 0, b: 1, c: false };
let bar = Bar::X(0);
let empty = Unit;
```

That's it. Every other way you make an instance of a type is just calling a totally vanilla function that does some stuff and eventually bottoms out to The One True Constructor.

Unlike C++, Rust does not come with a slew of built-in kinds of constructor. There are no Copy, Default, Assignment, Move, or whatever constructors. The reasons for this are varied, but it largely boils down to Rust's philosophy of *being explicit*.

Move constructors are meaningless in Rust because we don't enable types to "care" about their location in memory. Every type must be ready for it to be blindly memcopied to somewhere else in memory. This means pure on-the-stack-but- still-movable intrusive linked lists are simply not happening in Rust (safely).

Assignment and copy constructors similarly don't exist because move semantics are the only semantics in Rust. At most `x = y` just moves the bits of y into the x variable. Rust does provide two facilities for providing C++'s copy- oriented semantics: `Copy` and `Clone`. Clone is our moral equivalent of a copy constructor, but it's never implicitly invoked. You have to explicitly call `clone` on an element you want to be cloned. Copy is a special case of Clone where the implementation is just "copy the bits". Copy types *are* implicitly cloned whenever they're moved, but because of the definition of Copy this just means not treating the old copy as uninitialized -- a no-op.

While Rust provides a `Default` trait for specifying the moral equivalent of a default constructor, it's incredibly rare for this trait to be used. This is because variables aren't implicitly initialized. Default is basically only useful for generic programming. In concrete contexts, a type will provide a static `new` method for any kind of "default" constructor. This has no relation to `new` in other languages and has no special meaning. It's just a naming convention.

TODO: talk about "placement new"?

Destructors

What the language *does* provide is full-blown automatic destructors through the `Drop` trait, which provides the following method:

```
fn drop(&mut self);
```

This method gives the type time to somehow finish what it was doing.

After `drop` is run, Rust will recursively try to drop all of the fields of `self`.

This is a convenience feature so that you don't have to write "destructor boilerplate" to drop children. If a struct has no special logic for being dropped other than dropping its children, then it means `Drop` doesn't need to be implemented at all!

There is no stable way to prevent this behavior in Rust 1.0.

Note that taking `&mut self` means that even if you could suppress recursive Drop, Rust will prevent you from e.g. moving fields out of self. For most types, this is totally fine.

For instance, a custom implementation of `Box` might write `Drop` like this:

```
#![feature(ptr_internals, allocator_api)]

use std::alloc::{Allocator, Global, GlobalAlloc, Layout};
use std::mem;
use std::ptr::{drop_in_place, NonNull, Unique};

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(self.ptr.as_ptr());
            let c: NonNull<T> = self.ptr.into();
            Global.deallocate(c.cast(), Layout::new::<T>())
        }
    }
}
```

and this works fine because when Rust goes to drop the `ptr` field it just sees a Unique that has no actual `Drop` implementation. Similarly nothing can use-after-free the `ptr` because when drop exits, it becomes inaccessible.

However this wouldn't work:

```
#![feature(allocator_api, ptr_internals)]

use std::alloc::{Allocator, Global, GlobalAlloc, Layout};
use std::ptr::{drop_in_place, Unique, NonNull};
use std::mem;

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(self.ptr.as_ptr());
            let c: NonNull<T> = self.ptr.into();
            Global.deallocate(c.cast(), Layout::new::<T>());
        }
    }
}

struct SuperBox<T> { my_box: Box<T> }

impl<T> Drop for SuperBox<T> {
    fn drop(&mut self) {
        unsafe {
            // Hyper-optimized: deallocate the box's contents for it
            // without `drop`ing the contents
            let c: NonNull<T> = self.my_box.ptr.into();
            Global.deallocate(c.cast::<u8>(), Layout::new::<T>());
        }
    }
}
```

After we deallocate the `box` 's ptr in SuperBox's destructor, Rust will happily proceed to tell the box to Drop itself and everything will blow up with use-after-frees and double-frees.

Note that the recursive drop behavior applies to all structs and enums regardless of whether they implement Drop. Therefore something like

```
struct Boxy<T> {
    data1: Box<T>,
    data2: Box<T>,
    info: u32,
}
```

will have its data1 and data2's fields destructors whenever it "would" be dropped, even though it itself doesn't implement Drop. We say that such a type *needs Drop*, even though it is not itself Drop.

Similarly,

```
enum Link {
    Next(Box<Link>),
    None,
}
```

will have its inner Box field dropped if and only if an instance stores the Next variant.

In general this works really nicely because you don't need to worry about adding/removing drops when you refactor your data layout. Still there's certainly many valid usecases for needing to do trickier things with destructors.

The classic safe solution to overriding recursive drop and allowing moving out of Self during `drop` is to use an Option:

```
#![feature(allocator_api, ptr_internals)]

use std::alloc::{Allocator, GlobalAlloc, Global, Layout};
use std::ptr::{drop_in_place, Unique, NonNull};
use std::mem;

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(self.ptr.as_ptr());
            let c: NonNull<T> = self.ptr.into();
            Global.deallocate(c.cast(), Layout::new::<T>());
        }
    }
}

struct SuperBox<T> { my_box: Option<Box<T>> }

impl<T> Drop for SuperBox<T> {
    fn drop(&mut self) {
        unsafe {
            // Hyper-optimized: deallocate the box's contents for it
            // without `drop`ing the contents. Need to set the `box`
            // field as `None` to prevent Rust from trying to Drop it.
            let my_box = self.my_box.take().unwrap();
            let c: NonNull<T> = my_box.ptr.into();
            Global.deallocate(c.cast(), Layout::new::<T>());
            mem::forget(my_box);
        }
    }
}
```

However this has fairly odd semantics: you're saying that a field that *should* always be Some *may* be None, just because that happens in the destructor. Of course this conversely makes a lot of sense: you can call arbitrary methods on self during the destructor, and this should prevent you from ever doing so after deinitializing the field. Not that it will prevent you from producing any other arbitrarily

invalid state in there.

On balance this is an ok choice. Certainly what you should reach for by default. However, in the future we expect there to be a first-class way to announce that a field shouldn't be automatically dropped.

Leaking

Ownership-based resource management is intended to simplify composition. You acquire resources when you create the object, and you release the resources when it gets destroyed. Since destruction is handled for you, it means you can't forget to release the resources, and it happens as soon as possible! Surely this is perfect and all of our problems are solved.

Everything is terrible and we have new and exotic problems to try to solve.

Many people like to believe that Rust eliminates resource leaks. In practice, this is basically true. You would be surprised to see a Safe Rust program leak resources in an uncontrolled way.

However from a theoretical perspective this is absolutely not the case, no matter how you look at it. In the strictest sense, "leaking" is so abstract as to be unpreventable. It's quite trivial to initialize a collection at the start of a program, fill it with tons of objects with destructors, and then enter an infinite event loop that never refers to it. The collection will sit around uselessly, holding on to its precious resources until the program terminates (at which point all those resources would have been reclaimed by the OS anyway).

We may consider a more restricted form of leak: failing to drop a value that is unreachable. Rust also doesn't prevent this. In fact Rust *has a function for doing this*: `mem::forget`. This function consumes the value it is passed *and then doesn't run its destructor*.

In the past `mem::forget` was marked as unsafe as a sort of lint against using it, since failing to call a destructor is generally not a well-behaved thing to do (though useful for some special unsafe code). However this was generally determined to be an untenable stance to take: there are many ways to fail to call a destructor in safe code. The most famous example is creating a cycle of reference-counted pointers using interior mutability.

It is reasonable for safe code to assume that destructor leaks do not happen, as any program that leaks destructors is probably wrong. However *unsafe* code cannot rely on destructors to be run in

order to be safe. For most types this doesn't matter: if you leak the destructor then the type is by definition inaccessible, so it doesn't matter, right? For instance, if you leak a `Box<u8>` then you waste some memory but that's hardly going to violate memory-safety.

However where we must be careful with destructor leaks are *proxy* types. These are types which manage access to a distinct object, but don't actually own it. Proxy objects are quite rare. Proxy objects you'll need to care about are even rarer. However we'll focus on three interesting examples in the standard library:

- `vec::Drain`
- `Rc`
- `thread::scoped::JoinGuard`

Drain

`drain` is a collections API that moves data out of the container without consuming the container. This enables us to reuse the allocation of a `Vec` after claiming ownership over all of its contents. It produces an iterator (Drain) that returns the contents of the Vec by-value.

Now, consider Drain in the middle of iteration: some values have been moved out, and others haven't. This means that part of the Vec is now full of logically uninitialized data! We could backshift all the elements in the Vec every time we remove a value, but this would have pretty catastrophic performance consequences.

Instead, we would like Drain to fix the Vec's backing storage when it is dropped. It should run itself to completion, backshift any elements that weren't removed (drain supports subranges), and then fix Vec's `len`. It's even unwinding-safe! Easy!

Now consider the following:

```
let mut vec = vec![Box::new(0); 4];

{
    // start draining, vec can no longer be accessed
    let mut drainer = vec.drain(..);

    // pull out two elements and immediately drop them
    drainer.next();
    drainer.next();

    // get rid of drainer, but don't call its destructor
    mem::forget(drainer);
}

// Oops, vec[0] was dropped, we're reading a pointer into free'd memory!
println!("{}", vec[0]);
```

This is pretty clearly Not Good. Unfortunately, we're kind of stuck between a rock and a hard place: maintaining consistent state at every step has an enormous cost (and would negate any benefits of the API). Failing to maintain consistent state gives us Undefined Behavior in safe code (making the API unsound).

So what can we do? Well, we can pick a trivially consistent state: set the Vec's len to be 0 when we start the iteration, and fix it up if necessary in the destructor. That way, if everything executes like normal we get the desired behavior with minimal overhead. But if someone has the *audacity* to mem::forget us in the middle of the iteration, all that does is *leak even more* (and possibly leave the Vec in an unexpected but otherwise consistent state). Since we've accepted that mem::forget is safe, this is definitely safe. We call leaks causing more leaks a *leak amplification*.

Rc

Rc is an interesting case because at first glance it doesn't appear to be a proxy value at all. After all, it

manages the data it points to, and dropping all the Rcs for a value will drop that value. Leaking an Rc doesn't seem like it would be particularly dangerous. It will leave the refcount permanently incremented and prevent the data from being freed or dropped, but that seems just like Box, right?

Nope.

Let's consider a simplified implementation of Rc:

```
struct Rc<T> {
    ptr: *mut RcBox<T>,
}

struct RcBox<T> {
    data: T,
    ref_count: usize,
}

impl<T> Rc<T> {
    fn new(data: T) -> Self {
        unsafe {
            // Wouldn't it be nice if heap::allocate worked like this?
            let ptr = heap::allocate::<RcBox<T>>();
            ptr::write(ptr, RcBox {
                data: data,
                ref_count: 1,
            });
            Rc { ptr: ptr }
        }
    }

    fn clone(&self) -> Self {
        unsafe {
            (*self.ptr).ref_count += 1;
        }
        Rc { ptr: self.ptr }
    }
}

impl<T> Drop for Rc<T> {
    fn drop(&mut self) {
        unsafe {
            (*self.ptr).ref_count -= 1;
            if (*self.ptr).ref_count == 0 {
                // drop the data and then free it
                ptr::read(self.ptr);
                heap::deallocate(self.ptr);
            }
```

```
        }
    }
}
```

This code contains an implicit and subtle assumption: `ref_count` can fit in a `usize`, because there can't be more than `usize::MAX` Rcs in memory. However this itself assumes that the `ref_count` accurately reflects the number of Rcs in memory, which we know is false with `mem::forget`. Using `mem::forget` we can overflow the `ref_count`, and then get it down to 0 with outstanding Rcs. Then we can happily use-after-free the inner data. Bad Bad Not Good.

This can be solved by just checking the `ref_count` and doing *something*. The standard library's stance is to just abort, because your program has become horribly degenerate. Also *oh my gosh* it's such a ridiculous corner case.

thread::scoped::JoinGuard

> Note: This API has already been removed from std, for more information you may refer issue #24292.
>
> This section remains here because we think this example is still important, regardless of whether it is part of std or not.

The thread::scoped API intended to allow threads to be spawned that reference data on their parent's stack without any synchronization over that data by ensuring the parent joins the thread before any of the shared data goes out of scope.

```
pub fn scoped<'a, F>(f: F) -> JoinGuard<'a>
    where F: FnOnce() + Send + 'a
```

Here `f` is some closure for the other thread to execute. Saying that `F: Send +'a` is saying that it closes over data that lives for `'a`, and it either owns that data or the data was Sync (implying `&data` is Send).

Because JoinGuard has a lifetime, it keeps all the data it closes over borrowed in the parent thread. This means the JoinGuard can't outlive the data that the other thread is working on. When the JoinGuard *does* get dropped it blocks the parent thread, ensuring the child terminates before any of the closed-over data goes out of scope in the parent.

Usage looked like:

```
let mut data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
{
    let mut guards = vec![];
    for x in &mut data {
        // Move the mutable reference into the closure, and execute
        // it on a different thread. The closure has a lifetime bound
        // by the lifetime of the mutable reference `x` we store in it.
        // The guard that is returned is in turn assigned the lifetime
        // of the closure, so it also mutably borrows `data` as `x` did.
        // This means we cannot access `data` until the guard goes away.
        let guard = thread::scoped(move || {
            *x *= 2;
        });
        // store the thread's guard for later
        guards.push(guard);
    }
    // All guards are dropped here, forcing the threads to join
    // (this thread blocks here until the others terminate).
    // Once the threads join, the borrow expires and the data becomes
    // accessible again in this thread.
}
// data is definitely mutated here.
```

In principle, this totally works! Rust's ownership system perfectly ensures it! ...except it relies on a destructor being called to be safe.

```
let mut data = Box::new(0);
{
    let guard = thread::scoped(|| {
        // This is at best a data race. At worst, it's also a use-after-free.
        *data += 1;
    });
    // Because the guard is forgotten, expiring the loan without blocking this
    // thread.
    mem::forget(guard);
}
// So the Box is dropped here while the scoped thread may or may not be trying
// to access it.
```

Dang. Here the destructor running was pretty fundamental to the API, and it had to be scrapped in favor of a completely different design.

Unwinding

Rust has a *tiered* error-handling scheme:

- If something might reasonably be absent, Option is used.
- If something goes wrong and can reasonably be handled, Result is used.
- If something goes wrong and cannot reasonably be handled, the thread panics.
- If something catastrophic happens, the program aborts.

Option and Result are overwhelmingly preferred in most situations, especially since they can be promoted into a panic or abort at the API user's discretion. Panics cause the thread to halt normal execution and unwind its stack, calling destructors as if every function instantly returned.

As of 1.0, Rust is of two minds when it comes to panics. In the long-long-ago, Rust was much more like Erlang. Like Erlang, Rust had lightweight tasks, and tasks were intended to kill themselves with a panic when they reached an untenable state. Unlike an exception in Java or C++, a panic could not be caught at any time. Panics could only be caught by the owner of the task, at which point they had to be handled or *that* task would itself panic.

Unwinding was important to this story because if a task's destructors weren't called, it would cause memory and other system resources to leak. Since tasks were expected to die during normal execution, this would make Rust very poor for long-running systems!

As the Rust we know today came to be, this style of programming grew out of fashion in the push for less-and-less abstraction. Light-weight tasks were killed in the name of heavy-weight OS threads. Still, on stable Rust as of 1.0 panics can only be caught by the parent thread. This means catching a panic requires spinning up an entire OS thread! This unfortunately stands in conflict to Rust's philosophy of zero-cost abstractions.

There is an API called `catch_unwind` that enables catching a panic without spawning a thread. Still, we would encourage you to only do this sparingly. In particular, Rust's current unwinding

implementation is heavily optimized for the "doesn't unwind" case. If a program doesn't unwind, there should be no runtime cost for the program being *ready* to unwind. As a consequence, actually unwinding will be more expensive than in e.g. Java. Don't build your programs to unwind under normal circumstances. Ideally, you should only panic for programming errors or *extreme* problems.

Rust's unwinding strategy is not specified to be fundamentally compatible with any other language's unwinding. As such, unwinding into Rust from another language, or unwinding into another language from Rust is Undefined Behavior. You must *absolutely* catch any panics at the FFI boundary! What you do at that point is up to you, but *something* must be done. If you fail to do this, at best, your application will crash and burn. At worst, your application *won't* crash and burn, and will proceed with completely clobbered state.

Exception Safety

Although programs should use unwinding sparingly, there's a lot of code that *can* panic. If you unwrap a None, index out of bounds, or divide by 0, your program will panic. On debug builds, every arithmetic operation can panic if it overflows. Unless you are very careful and tightly control what code runs, pretty much everything can unwind, and you need to be ready for it.

Being ready for unwinding is often referred to as *exception safety* in the broader programming world. In Rust, there are two levels of exception safety that one may concern themselves with:

- In unsafe code, we *must* be exception safe to the point of not violating memory safety. We'll call this *minimal* exception safety.
- In safe code, it is *good* to be exception safe to the point of your program doing the right thing. We'll call this *maximal* exception safety.

As is the case in many places in Rust, Unsafe code must be ready to deal with bad Safe code when it comes to unwinding. Code that transiently creates unsound states must be careful that a panic does not cause that state to be used. Generally this means ensuring that only non-panicking code is run while these states exist, or making a guard that cleans up the state in the case of a panic. This does not necessarily mean that the state a panic witnesses is a fully coherent state. We need only guarantee that it's a *safe* state.

Most Unsafe code is leaf-like, and therefore fairly easy to make exception-safe. It controls all the code that runs, and most of that code can't panic. However it is not uncommon for Unsafe code to work with arrays of temporarily uninitialized data while repeatedly invoking caller-provided code. Such code needs to be careful and consider exception safety.

Vec::push_all

`Vec::push_all` is a temporary hack to get extending a Vec by a slice reliably efficient without specialization. Here's a simple implementation:

```
impl<T: Clone> Vec<T> {
    fn push_all(&mut self, to_push: &[T]) {
        self.reserve(to_push.len());
        unsafe {
            // can't overflow because we just reserved this
            self.set_len(self.len() + to_push.len());

            for (i, x) in to_push.iter().enumerate() {
                self.ptr().add(i).write(x.clone());
            }
        }
    }
}
```

We bypass `push` in order to avoid redundant capacity and `len` checks on the Vec that we definitely know has capacity. The logic is totally correct, except there's a subtle problem with our code: it's not exception-safe! `set_len`, `add`, and `write` are all fine; `clone` is the panic bomb we over-looked.

Clone is completely out of our control, and is totally free to panic. If it does, our function will exit early with the length of the Vec set too large. If the Vec is looked at or dropped, uninitialized memory will be read!

The fix in this case is fairly simple. If we want to guarantee that the values we *did* clone are dropped, we can set the `len` every loop iteration. If we just want to guarantee that uninitialized memory can't be observed, we can set the `len` after the loop.

BinaryHeap::sift_up

Bubbling an element up a heap is a bit more complicated than extending a Vec. The pseudocode is

as follows:

```
bubble_up(heap, index):
    while index != 0 && heap[index] < heap[parent(index)]:
        heap.swap(index, parent(index))
        index = parent(index)
```

A literal transcription of this code to Rust is totally fine, but has an annoying performance characteristic: the `self` element is swapped over and over again uselessly. We would rather have the following:

```
bubble_up(heap, index):
    let elem = heap[index]
    while index != 0 && elem < heap[parent(index)]:
        heap[index] = heap[parent(index)]
        index = parent(index)
    heap[index] = elem
```

This code ensures that each element is copied as little as possible (it is in fact necessary that elem be copied twice in general). However it now exposes some exception safety trouble! At all times, there exists two copies of one value. If we panic in this function something will be double-dropped. Unfortunately, we also don't have full control of the code: that comparison is user-defined!

Unlike Vec, the fix isn't as easy here. One option is to break the user-defined code and the unsafe code into two separate phases:

```
bubble_up(heap, index):
    let end_index = index;
    while end_index != 0 && heap[end_index] < heap[parent(end_index)]:
        end_index = parent(end_index)

    let elem = heap[index]
    while index != end_index:
        heap[index] = heap[parent(index)]
        index = parent(index)
    heap[index] = elem
```

If the user-defined code blows up, that's no problem anymore, because we haven't actually touched the state of the heap yet. Once we do start messing with the heap, we're working with only data and functions that we trust, so there's no concern of panics.

Perhaps you're not happy with this design. Surely it's cheating! And we have to do the complex heap traversal *twice*! Alright, let's bite the bullet. Let's intermix untrusted and unsafe code *for reals*.

If Rust had `try` and `finally` like in Java, we could do the following:

```
bubble_up(heap, index):
    let elem = heap[index]
    try:
        while index != 0 && elem < heap[parent(index)]:
            heap[index] = heap[parent(index)]
            index = parent(index)
    finally:
        heap[index] = elem
```

The basic idea is simple: if the comparison panics, we just toss the loose element in the logically uninitialized index and bail out. Anyone who observes the heap will see a potentially *inconsistent* heap, but at least it won't cause any double-drops! If the algorithm terminates normally, then this operation happens to coincide precisely with how we finish up regardless.

Sadly, Rust has no such construct, so we're going to need to roll our own! The way to do this is to

store the algorithm's state in a separate struct with a destructor for the "finally" logic. Whether we panic or not, that destructor will run and clean up after us.

```
struct Hole<'a, T: 'a> {
    data: &'a mut [T],
    /// `elt` is always `Some` from new until drop.
    elt: Option<T>,
    pos: usize,
}

impl<'a, T> Hole<'a, T> {
    fn new(data: &'a mut [T], pos: usize) -> Self {
        unsafe {
            let elt = ptr::read(&data[pos]);
            Hole {
                data: data,
                elt: Some(elt),
                pos: pos,
            }
        }
    }

    fn pos(&self) -> usize { self.pos }

    fn removed(&self) -> &T { self.elt.as_ref().unwrap() }

    unsafe fn get(&self, index: usize) -> &T { &self.data[index] }

    unsafe fn move_to(&mut self, index: usize) {
        let index_ptr: *const _ = &self.data[index];
        let hole_ptr = &mut self.data[self.pos];
        ptr::copy_nonoverlapping(index_ptr, hole_ptr, 1);
        self.pos = index;
    }
}

impl<'a, T> Drop for Hole<'a, T> {
    fn drop(&mut self) {
        // fill the hole again
        unsafe {
            let pos = self.pos;
            ptr::write(&mut self.data[pos], self.elt.take().unwrap());
```

```
        }
    }
}

impl<T: Ord> BinaryHeap<T> {
    fn sift_up(&mut self, pos: usize) {
        unsafe {
            // Take out the value at `pos` and create a hole.
            let mut hole = Hole::new(&mut self.data, pos);

            while hole.pos() != 0 {
                let parent = parent(hole.pos());
                if hole.removed() <= hole.get(parent) { break }
                hole.move_to(parent);
            }
            // Hole will be unconditionally filled here; panic or not!
        }
    }
}
```

Poisoning

Although all unsafe code *must* ensure it has minimal exception safety, not all types ensure *maximal* exception safety. Even if the type does, your code may ascribe additional meaning to it. For instance, an integer is certainly exception-safe, but has no semantics on its own. It's possible that code that panics could fail to correctly update the integer, producing an inconsistent program state.

This is *usually* fine, because anything that witnesses an exception is about to get destroyed. For instance, if you send a Vec to another thread and that thread panics, it doesn't matter if the Vec is in a weird state. It will be dropped and go away forever. However some types are especially good at smuggling values across the panic boundary.

These types may choose to explicitly *poison* themselves if they witness a panic. Poisoning doesn't entail anything in particular. Generally it just means preventing normal usage from proceeding. The most notable example of this is the standard library's Mutex type. A Mutex will poison itself if one of its MutexGuards (the thing it returns when a lock is obtained) is dropped during a panic. Any future attempts to lock the Mutex will return an `Err` or panic.

Mutex poisons not for true safety in the sense that Rust normally cares about. It poisons as a safety-guard against blindly using the data that comes out of a Mutex that has witnessed a panic while locked. The data in such a Mutex was likely in the middle of being modified, and as such may be in an inconsistent or incomplete state. It is important to note that one cannot violate memory safety with such a type if it is correctly written. After all, it must be minimally exception-safe!

However if the Mutex contained, say, a BinaryHeap that does not actually have the heap property, it's unlikely that any code that uses it will do what the author intended. As such, the program should not proceed normally. Still, if you're double-plus-sure that you can do *something* with the value, the Mutex exposes a method to get the lock anyway. It *is* safe, after all. Just maybe nonsense.

Concurrency and Parallelism

Rust as a language doesn't *really* have an opinion on how to do concurrency or parallelism. The standard library exposes OS threads and blocking sys-calls because everyone has those, and they're uniform enough that you can provide an abstraction over them in a relatively uncontroversial way. Message passing, green threads, and async APIs are all diverse enough that any abstraction over them tends to involve trade-offs that we weren't willing to commit to for 1.0.

However the way Rust models concurrency makes it relatively easy to design your own concurrency paradigm as a library and have everyone else's code Just Work with yours. Just require the right lifetimes and Send and Sync where appropriate and you're off to the races. Or rather, off to the... not... having... races.

Data Races and Race Conditions

Safe Rust guarantees an absence of data races, which are defined as:

- two or more threads concurrently accessing a location of memory
- one or more of them is a write
- one or more of them is unsynchronized

A data race has Undefined Behavior, and is therefore impossible to perform in Safe Rust. Data races are *mostly* prevented through Rust's ownership system: it's impossible to alias a mutable reference, so it's impossible to perform a data race. Interior mutability makes this more complicated, which is largely why we have the Send and Sync traits (see below).

However Rust does not prevent general race conditions.

This is pretty fundamentally impossible, and probably honestly undesirable. Your hardware is racy, your OS is racy, the other programs on your computer are racy, and the world this all runs in is racy. Any system that could genuinely claim to prevent *all* race conditions would be pretty awful to use, if not just incorrect.

So it's perfectly "fine" for a Safe Rust program to get deadlocked or do something nonsensical with incorrect synchronization. Obviously such a program isn't very good, but Rust can only hold your hand so far. Still, a race condition can't violate memory safety in a Rust program on its own. Only in conjunction with some other unsafe code can a race condition actually violate memory safety. For instance:

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];
// Arc so that the memory the AtomicUsize is stored in still exists for
// the other thread to increment, even if we completely finish executing
// before it. Rust won't compile the program without it, because of the
// lifetime requirements of thread::spawn!
let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

// `move` captures other_idx by-value, moving it into this thread
thread::spawn(move || {
    // It's ok to mutate idx because this value
    // is an atomic, so it can't cause a Data Race.
    other_idx.fetch_add(10, Ordering::SeqCst);
});

// Index with the value loaded from the atomic. This is safe because we
// read the atomic memory only once, and then pass a copy of that value
// to the Vec's indexing implementation. This indexing will be correctly
// bounds checked, and there's no chance of the value getting changed
// in the middle. However our program may panic if the thread we spawned
// managed to increment before this ran. A race condition because correct
// program execution (panicking is rarely correct) depends on order of
// thread execution.
println!("{}", data[idx.load(Ordering::SeqCst)]);
```

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];

let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

// `move` captures other_idx by-value, moving it into this thread
thread::spawn(move || {
    // It's ok to mutate idx because this value
    // is an atomic, so it can't cause a Data Race.
    other_idx.fetch_add(10, Ordering::SeqCst);
});

if idx.load(Ordering::SeqCst) < data.len() {
    unsafe {
        // Incorrectly loading the idx after we did the bounds check.
        // It could have changed. This is a race condition, *and dangerous*
        // because we decided to do `get_unchecked`, which is `unsafe`.
        println!("{}", data.get_unchecked(idx.load(Ordering::SeqCst)));
    }
}
```

Send and Sync

Not everything obeys inherited mutability, though. Some types allow you to have multiple aliases of a location in memory while mutating it. Unless these types use synchronization to manage this access, they are absolutely not thread-safe. Rust captures this through the `Send` and `Sync` traits.

- A type is Send if it is safe to send it to another thread.
- A type is Sync if it is safe to share between threads (T is Sync if and only if `&T` is Send).

Send and Sync are fundamental to Rust's concurrency story. As such, a substantial amount of special tooling exists to make them work right. First and foremost, they're unsafe traits. This means that they are unsafe to implement, and other unsafe code can assume that they are correctly implemented. Since they're *marker traits* (they have no associated items like methods), correctly implemented simply means that they have the intrinsic properties an implementor should have. Incorrectly implementing Send or Sync can cause Undefined Behavior.

Send and Sync are also automatically derived traits. This means that, unlike every other trait, if a type is composed entirely of Send or Sync types, then it is Send or Sync. Almost all primitives are Send and Sync, and as a consequence pretty much all types you'll ever interact with are Send and Sync.

Major exceptions include:

- raw pointers are neither Send nor Sync (because they have no safety guards).
- `UnsafeCell` isn't Sync (and therefore `Cell` and `RefCell` aren't).
- `Rc` isn't Send or Sync (because the refcount is shared and unsynchronized).

`Rc` and `UnsafeCell` are very fundamentally not thread-safe: they enable unsynchronized shared mutable state. However raw pointers are, strictly speaking, marked as thread-unsafe as more of a *lint*. Doing anything useful with a raw pointer requires dereferencing it, which is already unsafe. In that sense, one could argue that it would be "fine" for them to be marked as thread safe.

However it's important that they aren't thread-safe to prevent types that contain them from being automatically marked as thread-safe. These types have non-trivial untracked ownership, and it's unlikely that their author was necessarily thinking hard about thread safety. In the case of `Rc`, we have a nice example of a type that contains a `*mut` that is definitely not thread-safe.

Types that aren't automatically derived can simply implement them if desired:

```
struct MyBox(*mut u8);

unsafe impl Send for MyBox {}
unsafe impl Sync for MyBox {}
```

In the *incredibly rare* case that a type is inappropriately automatically derived to be Send or Sync, then one can also unimplement Send and Sync:

```
#![feature(negative_impls)]

// I have some magic semantics for some synchronization primitive!
struct SpecialThreadToken(u8);

impl !Send for SpecialThreadToken {}
impl !Sync for SpecialThreadToken {}
```

Note that *in and of itself* it is impossible to incorrectly derive Send and Sync. Only types that are ascribed special meaning by other unsafe code can possibly cause trouble by being incorrectly Send or Sync.

Most uses of raw pointers should be encapsulated behind a sufficient abstraction that Send and Sync can be derived. For instance all of Rust's standard collections are Send and Sync (when they contain Send and Sync types) in spite of their pervasive use of raw pointers to manage allocations and complex ownership. Similarly, most iterators into these collections are Send and Sync because they largely behave like an `&` or `&mut` into the collection.

Example

`Box` is implemented as its own special intrinsic type by the compiler for various reasons, but we can implement something with similar-ish behavior ourselves to see an example of when it is sound to implement Send and Sync. Let's call it a `Carton`.

We start by writing code to take a value allocated on the stack and transfer it to the heap.

```
use std::{
    mem::{align_of, size_of},
    ptr,
};

struct Carton<T>(ptr::NonNull<T>);

impl<T> Carton<T> {
    pub fn new(value: T) -> Self {
        // Allocate enough memory on the heap to store one T.
        assert_ne!(size_of::<T>(), 0, "Zero-sized types are out of the scope of this
example");
        let mut memptr: *mut T = ptr::null_mut();
        unsafe {
            let ret = libc::posix_memalign(
                (&mut memptr).cast(),
                align_of::<T>(),
                size_of::<T>()
            );
            assert_eq!(ret, 0, "Failed to allocate or invalid alignment");
        };

        // NonNull is just a wrapper that enforces that the pointer isn't null.
        let ptr = {
            // Safety: memptr is dereferenceable because we created it from a
            // reference and have exclusive access.
            ptr::NonNull::new(memptr)
                .expect("Guaranteed non-null if posix_memalign returns 0")
        };

        // Move value from the stack to the location we allocated on the heap.
        unsafe {
            // Safety: If non-null, posix_memalign gives us a ptr that is valid
            // for writes and properly aligned.
            ptr.as_ptr().write(value);
        }

        Self(ptr)
```

```
    }
}
```

This isn't very useful, because once our users give us a value they have no way to access it. `Box` implements `Deref` and `DerefMut` so that you can access the inner value. Let's do that.

```
use std::ops::{Deref, DerefMut};

impl<T> Deref for Carton<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        unsafe {
            // Safety: The pointer is aligned, initialized, and dereferenceable
            //   by the logic in [`Self::new`]. We require readers to borrow the
            //   Carton, and the lifetime of the return value is elided to the
            //   lifetime of the input. This means the borrow checker will
            //   enforce that no one can mutate the contents of the Carton until
            //   the reference returned is dropped.
            self.0.as_ref()
        }
    }
}

impl<T> DerefMut for Carton<T> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        unsafe {
            // Safety: The pointer is aligned, initialized, and dereferenceable
            //   by the logic in [`Self::new`]. We require writers to mutably
            //   borrow the Carton, and the lifetime of the return value is
            //   elided to the lifetime of the input. This means the borrow
            //   checker will enforce that no one else can access the contents
            //   of the Carton until the mutable reference returned is dropped.
            self.0.as_mut()
        }
    }
}
```

Finally, let's think about whether our `Carton` is Send and Sync. Something can safely be Send unless it shares mutable state with something else without enforcing exclusive access to it. Each `Carton` has a unique pointer, so we're good.

```
// Safety: No one besides us has the raw pointer, so we can safely transfer the
// Carton to another thread if T can be safely transferred.
unsafe impl<T> Send for Carton<T> where T: Send {}
```

What about Sync? For `Carton` to be Sync we have to enforce that you can't write to something stored in a `&Carton` while that same something could be read or written to from another `&Carton`. Since you need an `&mut Carton` to write to the pointer, and the borrow checker enforces that mutable references must be exclusive, there are no soundness issues making `Carton` sync either.

```
// Safety: Since there exists a public way to go from a `&Carton<T>` to a `&T`
// in an unsynchronized fashion (such as `Deref`), then `Carton<T>` can't be
// `Sync` if `T` isn't.
// Conversely, `Carton` itself does not use any interior mutability whatsoever:
// all the mutations are performed through an exclusive reference (`&mut`). This
// means it suffices that `T` be `Sync` for `Carton<T>` to be `Sync`:
unsafe impl<T> Sync for Carton<T> where T: Sync  {}
```

When we assert our type is Send and Sync we usually need to enforce that every contained type is Send and Sync. When writing custom types that behave like standard library types we can assert that we have the same requirements. For example, the following code asserts that a Carton is Send if the same sort of Box would be Send, which in this case is the same as saying T is Send.

```
unsafe impl<T> Send for Carton<T> where Box<T>: Send {}
```

Right now `Carton<T>` has a memory leak, as it never frees the memory it allocates. Once we fix that we have a new requirement we have to ensure we meet to be Send: we need to know `free` can be

called on a pointer that was yielded by an allocation done on another thread. We can check this is true in the docs for `libc::free`.

```
impl<T> Drop for Carton<T> {
    fn drop(&mut self) {
        unsafe {
            libc::free(self.0.as_ptr().cast());
        }
    }
}
```

A nice example where this does not happen is with a MutexGuard: notice how it is not Send. The implementation of MutexGuard uses libraries that require you to ensure you don't try to free a lock that you acquired in a different thread. If you were able to Send a MutexGuard to another thread the destructor would run in the thread you sent it to, violating the requirement. MutexGuard can still be Sync because all you can send to another thread is an `&MutexGuard` and dropping a reference does nothing.

TODO: better explain what can or can't be Send or Sync. Sufficient to appeal only to data races?

Atomics

Rust pretty blatantly just inherits the memory model for atomics from C++20. This is not due to this model being particularly excellent or easy to understand. Indeed, this model is quite complex and known to have several flaws. Rather, it is a pragmatic concession to the fact that *everyone* is pretty bad at modeling atomics. At very least, we can benefit from existing tooling and research around the C/C++ memory model. (You'll often see this model referred to as "C/C++11" or just "C11". C just copies the C++ memory model; and C++11 was the first version of the model but it has received some bugfixes since then.)

Trying to fully explain the model in this book is fairly hopeless. It's defined in terms of madness-inducing causality graphs that require a full book to properly understand in a practical way. If you want all the nitty-gritty details, you should check out the C++ specification. Still, we'll try to cover the basics and some of the problems Rust developers face.

The C++ memory model is fundamentally about trying to bridge the gap between the semantics we want, the optimizations compilers want, and the inconsistent chaos our hardware wants. *We* would like to just write programs and have them do exactly what we said but, you know, fast. Wouldn't that be great?

Compiler Reordering

Compilers fundamentally want to be able to do all sorts of complicated transformations to reduce data dependencies and eliminate dead code. In particular, they may radically change the actual order of events, or make events never occur! If we write something like:

```
x = 1;
y = 3;
x = 2;
```

The compiler may conclude that it would be best if your program did:

```
x = 2;
y = 3;
```

This has inverted the order of events and completely eliminated one event. From a single-threaded perspective this is completely unobservable: after all the statements have executed we are in exactly the same state. But if our program is multi-threaded, we may have been relying on `x` to actually be assigned to 1 before `y` was assigned. We would like the compiler to be able to make these kinds of optimizations, because they can seriously improve performance. On the other hand, we'd also like to be able to depend on our program *doing the thing we said*.

Hardware Reordering

On the other hand, even if the compiler totally understood what we wanted and respected our wishes, our hardware might instead get us in trouble. Trouble comes from CPUs in the form of memory hierarchies. There is indeed a global shared memory space somewhere in your hardware, but from the perspective of each CPU core it is *so very far away* and *so very slow*. Each CPU would rather work with its local cache of the data and only go through all the anguish of talking to shared memory only when it doesn't actually have that memory in cache.

After all, that's the whole point of the cache, right? If every read from the cache had to run back to shared memory to double check that it hadn't changed, what would the point be? The end result is that the hardware doesn't guarantee that events that occur in some order on *one* thread, occur in the same order on *another* thread. To guarantee this, we must issue special instructions to the CPU telling it to be a bit less smart.

For instance, say we convince the compiler to emit this logic:

```
initial state: x = 0, y = 1

THREAD 1        THREAD2
y = 3;          if x == 1 {
x = 1;              y *= 2;
                }
```

Ideally this program has 2 possible final states:

- `y = 3` : (thread 2 did the check before thread 1 completed)
- `y = 6` : (thread 2 did the check after thread 1 completed)

However there's a third potential state that the hardware enables:

- `y = 2` : (thread 2 saw `x = 1` , but not `y = 3` , and then overwrote `y = 3`)

It's worth noting that different kinds of CPU provide different guarantees. It is common to separate hardware into two categories: strongly-ordered and weakly-ordered. Most notably x86/64 provides strong ordering guarantees, while ARM provides weak ordering guarantees. This has two consequences for concurrent programming:

- Asking for stronger guarantees on strongly-ordered hardware may be cheap or even free because they already provide strong guarantees unconditionally. Weaker guarantees may only yield performance wins on weakly-ordered hardware.
- Asking for guarantees that are too weak on strongly-ordered hardware is more likely to *happen* to work, even though your program is strictly incorrect. If possible, concurrent algorithms should be tested on weakly-ordered hardware.

Data Accesses

The C++ memory model attempts to bridge the gap by allowing us to talk about the *causality* of our program. Generally, this is by establishing a *happens before* relationship between parts of the program and the threads that are running them. This gives the hardware and compiler room to optimize the program more aggressively where a strict happens-before relationship isn't established, but forces them to be more careful where one is established. The way we communicate these relationships are through *data accesses* and *atomic accesses*.

Data accesses are the bread-and-butter of the programming world. They are fundamentally unsynchronized and compilers are free to aggressively optimize them. In particular, data accesses are free to be reordered by the compiler on the assumption that the program is single-threaded. The hardware is also free to propagate the changes made in data accesses to other threads as lazily and inconsistently as it wants. Most critically, data accesses are how data races happen. Data accesses are very friendly to the hardware and compiler, but as we've seen they offer *awful* semantics to try to write synchronized code with. Actually, that's too weak.

It is literally impossible to write correct synchronized code using only data accesses.

Atomic accesses are how we tell the hardware and compiler that our program is multi-threaded. Each atomic access can be marked with an *ordering* that specifies what kind of relationship it establishes with other accesses. In practice, this boils down to telling the compiler and hardware certain things they *can't* do. For the compiler, this largely revolves around re-ordering of instructions. For the hardware, this largely revolves around how writes are propagated to other threads. The set of orderings Rust exposes are:

- Sequentially Consistent (SeqCst)
- Release
- Acquire
- Relaxed

(Note: We explicitly do not expose the C++ *consume* ordering)

TODO: negative reasoning vs positive reasoning? TODO: "can't forget to synchronize"

Sequentially Consistent

Sequentially Consistent is the most powerful of all, implying the restrictions of all other orderings. Intuitively, a sequentially consistent operation cannot be reordered: all accesses on one thread that happen before and after a SeqCst access stay before and after it. A data-race-free program that uses only sequentially consistent atomics and data accesses has the very nice property that there is a single global execution of the program's instructions that all threads agree on. This execution is also particularly nice to reason about: it's just an interleaving of each thread's individual executions. This does not hold if you start using the weaker atomic orderings.

The relative developer-friendliness of sequential consistency doesn't come for free. Even on strongly-ordered platforms sequential consistency involves emitting memory fences.

In practice, sequential consistency is rarely necessary for program correctness. However sequential consistency is definitely the right choice if you're not confident about the other memory orders. Having your program run a bit slower than it needs to is certainly better than it running incorrectly! It's also mechanically trivial to downgrade atomic operations to have a weaker consistency later on. Just change `SeqCst` to `Relaxed` and you're done! Of course, proving that this transformation is *correct* is a whole other matter.

Acquire-Release

Acquire and Release are largely intended to be paired. Their names hint at their use case: they're perfectly suited for acquiring and releasing locks, and ensuring that critical sections don't overlap.

Intuitively, an acquire access ensures that every access after it stays after it. However operations that occur before an acquire are free to be reordered to occur after it. Similarly, a release access ensures that every access before it stays before it. However operations that occur after a release are free to be reordered to occur before it.

When thread A releases a location in memory and then thread B subsequently acquires *the same* location in memory, causality is established. Every write (including non-atomic and relaxed atomic writes) that happened before A's release will be observed by B after its acquisition. However no causality is established with any other threads. Similarly, no causality is established if A and B access *different* locations in memory.

Basic use of release-acquire is therefore simple: you acquire a location of memory to begin the critical section, and then release that location to end it. For instance, a simple spinlock might look like:

```
use std::sync::Arc;
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

fn main() {
    let lock = Arc::new(AtomicBool::new(false)); // value answers "am I locked?"

    // ... distribute lock to threads somehow ...

    // Try to acquire the lock by setting it to true
    while lock.compare_and_swap(false, true, Ordering::Acquire) { }
    // broke out of the loop, so we successfully acquired the lock!

    // ... scary data accesses ...

    // ok we're done, release the lock
    lock.store(false, Ordering::Release);
}
```

On strongly-ordered platforms most accesses have release or acquire semantics, making release and acquire often totally free. This is not the case on weakly-ordered platforms.

Relaxed

Relaxed accesses are the absolute weakest. They can be freely re-ordered and provide no happens-before relationship. Still, relaxed operations are still atomic. That is, they don't count as data accesses and any read-modify-write operations done to them occur atomically. Relaxed operations are appropriate for things that you definitely want to happen, but don't particularly otherwise care about. For instance, incrementing a counter can be safely done by multiple threads using a relaxed `fetch_add` if you're not using the counter to synchronize any other accesses.

There's rarely a benefit in making an operation relaxed on strongly-ordered platforms, since they usually provide release-acquire semantics anyway. However relaxed operations can be cheaper on weakly-ordered platforms.

Example: Implementing Vec

To bring everything together, we're going to write `std::Vec` from scratch. We will limit ourselves to stable Rust. In particular we won't use any intrinsics that could make our code a little bit nicer or efficient because intrinsics are permanently unstable. Although many intrinsics *do* become stabilized elsewhere (`std::ptr` and `std::mem` consist of many intrinsics).

Ultimately this means our implementation may not take advantage of all possible optimizations, though it will be by no means *naive*. We will definitely get into the weeds over nitty-gritty details, even when the problem doesn't *really* merit it.

You wanted advanced. We're gonna go advanced.

Layout

First off, we need to come up with the struct layout. A Vec has three parts: a pointer to the allocation, the size of the allocation, and the number of elements that have been initialized.

Naively, this means we just want this design:

```
pub struct Vec<T> {
    ptr: *mut T,
    cap: usize,
    len: usize,
}
```

And indeed this would compile. Unfortunately, it would be incorrect. First, the compiler will give us too strict variance. So a `&Vec<&'static str>` couldn't be used where an `&Vec<&'a str>` was expected. More importantly, it will give incorrect ownership information to the drop checker, as it will conservatively assume we don't own any values of type `T`. See the chapter on ownership and lifetimes for all the details on variance and drop check.

As we saw in the ownership chapter, the standard library uses `Unique<T>` in place of `*mut T` when it has a raw pointer to an allocation that it owns. Unique is unstable, so we'd like to not use it if possible, though.

As a recap, Unique is a wrapper around a raw pointer that declares that:

- We are covariant over `T`
- We may own a value of type `T` (for drop check)
- We are Send/Sync if `T` is Send/Sync
- Our pointer is never null (so `Option<Vec<T>>` is null-pointer-optimized)

We can implement all of the above requirements in stable Rust. To do this, instead of using `Unique<T>` we will use `NonNull<T>`, another wrapper around a raw pointer, which gives us two of

the above properties, namely it is covariant over `T` and is declared to never be null. By adding a `PhantomData<T>` (for drop check) and implementing Send/Sync if `T` is, we get the same results as using `Unique<T>`:

```
use std::ptr::NonNull;
use std::marker::PhantomData;

pub struct Vec<T> {
    ptr: NonNull<T>,
    cap: usize,
    len: usize,
    _marker: PhantomData<T>,
}

unsafe impl<T: Send> Send for Vec<T> {}
unsafe impl<T: Sync> Sync for Vec<T> {}
```

Allocating Memory

Using `NonNull` throws a wrench in an important feature of Vec (and indeed all of the std collections): creating an empty Vec doesn't actually allocate at all. This is not the same as allocating a zero-sized memory block, which is not allowed by the global allocator (it results in undefined behavior!). So if we can't allocate, but also can't put a null pointer in `ptr`, what do we do in `Vec::new`? Well, we just put some other garbage in there!

This is perfectly fine because we already have `cap == 0` as our sentinel for no allocation. We don't even need to handle it specially in almost any code because we usually need to check if `cap > len` or `len > 0` anyway. The recommended Rust value to put here is `mem::align_of::<T>()`. `NonNull` provides a convenience for this: `NonNull::dangling()`. There are quite a few places where we'll want to use `dangling` because there's no real allocation to talk about but `null` would make the compiler do bad things.

So:

```
use std::mem;

impl<T> Vec<T> {
    pub fn new() -> Self {
        assert!(mem::size_of::<T>() != 0, "We're not ready to handle ZSTs");
        Vec {
            ptr: NonNull::dangling(),
            len: 0,
            cap: 0,
            _marker: PhantomData,
        }
    }
}
```

I slipped in that assert there because zero-sized types will require some special handling throughout our code, and I want to defer the issue for now. Without this assert, some of our early drafts will do

some Very Bad Things.

Next we need to figure out what to actually do when we *do* want space. For that, we use the global allocation functions `alloc`, `realloc`, and `dealloc` which are available in stable Rust in `std::alloc`. These functions are expected to become deprecated in favor of the methods of `std::alloc::Global` after this type is stabilized.

We'll also need a way to handle out-of-memory (OOM) conditions. The standard library provides a function `alloc::handle_alloc_error`, which will abort the program in a platform-specific manner. The reason we abort and don't panic is because unwinding can cause allocations to happen, and that seems like a bad thing to do when your allocator just came back with "hey I don't have any more memory".

Of course, this is a bit silly since most platforms don't actually run out of memory in a conventional way. Your operating system will probably kill the application by another means if you legitimately start using up all the memory. The most likely way we'll trigger OOM is by just asking for ludicrous quantities of memory at once (e.g. half the theoretical address space). As such it's *probably* fine to panic and nothing bad will happen. Still, we're trying to be like the standard library as much as possible, so we'll just kill the whole program.

Okay, now we can write growing. Roughly, we want to have this logic:

```
if cap == 0:
    allocate()
    cap = 1
else:
    reallocate()
    cap *= 2
```

But Rust's only supported allocator API is so low level that we'll need to do a fair bit of extra work. We also need to guard against some special conditions that can occur with really large allocations or empty allocations.

In particular, `ptr::offset` will cause us a lot of trouble, because it has the semantics of LLVM's GEP inbounds instruction. If you're fortunate enough to not have dealt with this instruction, here's the basic story with GEP: alias analysis, alias analysis, alias analysis. It's super important to an optimizing compiler to be able to reason about data dependencies and aliasing.

As a simple example, consider the following fragment of code:

```
*x *= 7;
*y *= 3;
```

If the compiler can prove that `x` and `y` point to different locations in memory, the two operations can in theory be executed in parallel (by e.g. loading them into different registers and working on them independently). However the compiler can't do this in general because if x and y point to the same location in memory, the operations need to be done to the same value, and they can't just be merged afterwards.

When you use GEP inbounds, you are specifically telling LLVM that the offsets you're about to do are within the bounds of a single "allocated" entity. The ultimate payoff being that LLVM can assume that if two pointers are known to point to two disjoint objects, all the offsets of those pointers are *also* known to not alias (because you won't just end up in some random place in memory). LLVM is heavily optimized to work with GEP offsets, and inbounds offsets are the best of all, so it's important that we use them as much as possible.

So that's what GEP's about, how can it cause us trouble?

The first problem is that we index into arrays with unsigned integers, but GEP (and as a consequence `ptr::offset`) takes a signed integer. This means that half of the seemingly valid indices into an array will overflow GEP and actually go in the wrong direction! As such we must limit all allocations to `isize::MAX` elements. This actually means we only need to worry about byte-sized objects, because e.g. `> isize::MAX u16`s will truly exhaust all of the system's memory. However in order to avoid subtle corner cases where someone reinterprets some array of `< isize::MAX` objects as bytes, std limits all allocations to `isize::MAX` bytes.

On all 64-bit targets that Rust currently supports we're artificially limited to significantly less than all 64 bits of the address space (modern x64 platforms only expose 48-bit addressing), so we can rely on just running out of memory first. However on 32-bit targets, particularly those with extensions to use more of the address space (PAE x86 or x32), it's theoretically possible to successfully allocate more than `isize::MAX` bytes of memory.

However since this is a tutorial, we're not going to be particularly optimal here, and just unconditionally check, rather than use clever platform-specific `cfg` s.

The other corner-case we need to worry about is empty allocations. There will be two kinds of empty allocations we need to worry about: `cap = 0` for all T, and `cap > 0` for zero-sized types.

These cases are tricky because they come down to what LLVM means by "allocated". LLVM's notion of an allocation is significantly more abstract than how we usually use it. Because LLVM needs to work with different languages' semantics and custom allocators, it can't really intimately understand allocation. Instead, the main idea behind allocation is "doesn't overlap with other stuff". That is, heap allocations, stack allocations, and globals don't randomly overlap. Yep, it's about alias analysis. As such, Rust can technically play a bit fast and loose with the notion of an allocation as long as it's *consistent*.

Getting back to the empty allocation case, there are a couple of places where we want to offset by 0 as a consequence of generic code. The question is then: is it consistent to do so? For zero-sized types, we have concluded that it is indeed consistent to do a GEP inbounds offset by an arbitrary number of elements. This is a runtime no-op because every element takes up no space, and it's fine to pretend that there's infinite zero-sized types allocated at `0x01` . No allocator will ever allocate that address, because they won't allocate `0x00` and they generally allocate to some minimal alignment higher than a byte. Also generally the whole first page of memory is protected from being allocated anyway (a whole 4k, on many platforms).

However what about for positive-sized types? That one's a bit trickier. In principle, you can argue that offsetting by 0 gives LLVM no information: either there's an element before the address or after it, but it can't know which. However we've chosen to conservatively assume that it may do bad

things. As such we will guard against this case explicitly.

Phew

Ok with all the nonsense out of the way, let's actually allocate some memory:

```
use std::alloc::{self, Layout};

impl<T> Vec<T> {
    fn grow(&mut self) {
        let (new_cap, new_layout) = if self.cap == 0 {
            (1, Layout::array::<T>(1).unwrap())
        } else {
            // This can't overflow since self.cap <= isize::MAX.
            let new_cap = 2 * self.cap;

            // `Layout::array` checks that the number of bytes is <= usize::MAX,
            // but this is redundant since old_layout.size() <= isize::MAX,
            // so the `unwrap` should never fail.
            let new_layout = Layout::array::<T>(new_cap).unwrap();
            (new_cap, new_layout)
        };

        // Ensure that the new allocation doesn't exceed `isize::MAX` bytes.
        assert!(new_layout.size() <= isize::MAX as usize, "Allocation too large");

        let new_ptr = if self.cap == 0 {
            unsafe { alloc::alloc(new_layout) }
        } else {
            let old_layout = Layout::array::<T>(self.cap).unwrap();
            let old_ptr = self.ptr.as_ptr() as *mut u8;
            unsafe { alloc::realloc(old_ptr, old_layout, new_layout.size()) }
        };

        // If allocation fails, `new_ptr` will be null, in which case we abort.
        self.ptr = match NonNull::new(new_ptr as *mut T) {
            Some(p) => p,
            None => alloc::handle_alloc_error(new_layout),
        };
        self.cap = new_cap;
    }
}
```

Push and Pop

Alright. We can initialize. We can allocate. Let's actually implement some functionality! Let's start with `push`. All it needs to do is check if we're full to grow, unconditionally write to the next index, and then increment our length.

To do the write we have to be careful not to evaluate the memory we want to write to. At worst, it's truly uninitialized memory from the allocator. At best it's the bits of some old value we popped off. Either way, we can't just index to the memory and dereference it, because that will evaluate the memory as a valid instance of T. Worse, `foo[idx] = x` will try to call `drop` on the old value of `foo[idx]`!

The correct way to do this is with `ptr::write`, which just blindly overwrites the target address with the bits of the value we provide. No evaluation involved.

For `push`, if the old len (before push was called) is 0, then we want to write to the 0th index. So we should offset by the old len.

```
pub fn push(&mut self, elem: T) {
    if self.len == self.cap { self.grow(); }

    unsafe {
        ptr::write(self.ptr.as_ptr().add(self.len), elem);
    }

    // Can't fail, we'll OOM first.
    self.len += 1;
}
```

Easy! How about `pop`? Although this time the index we want to access is initialized, Rust won't just let us dereference the location of memory to move the value out, because that would leave the memory uninitialized! For this we need `ptr::read`, which just copies out the bits from the target address and interprets it as a value of type T. This will leave the memory at this address logically

uninitialized, even though there is in fact a perfectly good instance of T there.

For `pop`, if the old len is 1, for example, we want to read out of the 0th index. So we should offset by the new len.

```
pub fn pop(&mut self) -> Option<T> {
    if self.len == 0 {
        None
    } else {
        self.len -= 1;
        unsafe {
            Some(ptr::read(self.ptr.as_ptr().add(self.len)))
        }
    }
}
```

Deallocating

Next we should implement Drop so that we don't massively leak tons of resources. The easiest way is to just call `pop` until it yields None, and then deallocate our buffer. Note that calling `pop` is unneeded if `T: !Drop`. In theory we can ask Rust if `T needs_drop` and omit the calls to `pop`. However in practice LLVM is *really* good at removing simple side-effect free code like this, so I wouldn't bother unless you notice it's not being stripped (in this case it is).

We must not call `alloc::dealloc` when `self.cap == 0`, as in this case we haven't actually allocated any memory.

```
impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            while let Some(_) = self.pop() { }
            let layout = Layout::array::<T>(self.cap).unwrap();
            unsafe {
                alloc::dealloc(self.ptr.as_ptr() as *mut u8, layout);
            }
        }
    }
}
```

Deref

Alright! We've got a decent minimal stack implemented. We can push, we can pop, and we can clean up after ourselves. However there's a whole mess of functionality we'd reasonably want. In particular, we have a proper array, but none of the slice functionality. That's actually pretty easy to solve: we can implement `Deref<Target=[T]>`. This will magically make our Vec coerce to, and behave like, a slice in all sorts of conditions.

All we need is `slice::from_raw_parts`. It will correctly handle empty slices for us. Later once we set up zero-sized type support it will also Just Work for those too.

```
use std::ops::Deref;

impl<T> Deref for Vec<T> {
    type Target = [T];
    fn deref(&self) -> &[T] {
        unsafe {
            std::slice::from_raw_parts(self.ptr.as_ptr(), self.len)
        }
    }
}
```

And let's do DerefMut too:

```
use std::ops::DerefMut;

impl<T> DerefMut for Vec<T> {
    fn deref_mut(&mut self) -> &mut [T] {
        unsafe {
            std::slice::from_raw_parts_mut(self.ptr.as_ptr(), self.len)
        }
    }
}
```

Now we have `len`, `first`, `last`, indexing, slicing, sorting, `iter`, `iter_mut`, and all other sorts of bells and whistles provided by slice. Sweet!

Insert and Remove

Something *not* provided by slice is `insert` and `remove`, so let's do those next.

Insert needs to shift all the elements at the target index to the right by one. To do this we need to use `ptr::copy`, which is our version of C's `memmove`. This copies some chunk of memory from one location to another, correctly handling the case where the source and destination overlap (which will definitely happen here).

If we insert at index `i`, we want to shift the `[i .. len]` to `[i+1 .. len+1]` using the old len.

```
pub fn insert(&mut self, index: usize, elem: T) {
    // Note: `<=` because it's valid to insert after everything
    // which would be equivalent to push.
    assert!(index <= self.len, "index out of bounds");
    if self.cap == self.len { self.grow(); }

    unsafe {
        // ptr::copy(src, dest, len): "copy from src to dest len elems"
        ptr::copy(self.ptr.as_ptr().add(index),
                  self.ptr.as_ptr().add(index + 1),
                  self.len - index);
        ptr::write(self.ptr.as_ptr().add(index), elem);
        self.len += 1;
    }
}
```

Remove behaves in the opposite manner. We need to shift all the elements from `[i+1 .. len + 1]` to `[i .. len]` using the *new* len.

```
pub fn remove(&mut self, index: usize) -> T {
    // Note: `<` because it's *not* valid to remove after everything
    assert!(index < self.len, "index out of bounds");
    unsafe {
        self.len -= 1;
        let result = ptr::read(self.ptr.as_ptr().add(index));
        ptr::copy(self.ptr.as_ptr().add(index + 1),
                  self.ptr.as_ptr().add(index),
                  self.len - index);
        result
    }
}
```

IntoIter

Let's move on to writing iterators. `iter` and `iter_mut` have already been written for us thanks to The Magic of Deref. However there's two interesting iterators that Vec provides that slices can't: `into_iter` and `drain`.

IntoIter consumes the Vec by-value, and can consequently yield its elements by-value. In order to enable this, IntoIter needs to take control of Vec's allocation.

IntoIter needs to be DoubleEnded as well, to enable reading from both ends. Reading from the back could just be implemented as calling `pop`, but reading from the front is harder. We could call `remove(0)` but that would be insanely expensive. Instead we're going to just use ptr::read to copy values out of either end of the Vec without mutating the buffer at all.

To do this we're going to use a very common C idiom for array iteration. We'll make two pointers; one that points to the start of the array, and one that points to one-element past the end. When we want an element from one end, we'll read out the value pointed to at that end and move the pointer over by one. When the two pointers are equal, we know we're done.

Note that the order of read and offset are reversed for `next` and `next_back` For `next_back` the pointer is always after the element it wants to read next, while for `next` the pointer is always at the element it wants to read next. To see why this is, consider the case where every element but one has been yielded.

The array looks like this:

```
          S  E
[X, X, X, O, X, X, X]
```

If E pointed directly at the element it wanted to yield next, it would be indistinguishable from the case where there are no more elements to yield.

Although we don't actually care about it during iteration, we also need to hold onto the Vec's allocation information in order to free it once IntoIter is dropped.

So we're going to use the following struct:

```
pub struct IntoIter<T> {
    buf: NonNull<T>,
    cap: usize,
    start: *const T,
    end: *const T,
    _marker: PhantomData<T>,
}
```

And this is what we end up with for initialization:

```
impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;
    fn into_iter(self) -> IntoIter<T> {
        // Can't destructure Vec since it's Drop
        let ptr = self.ptr;
        let cap = self.cap;
        let len = self.len;

        // Make sure not to drop Vec since that would free the buffer
        mem::forget(self);

        unsafe {
            IntoIter {
                buf: ptr,
                cap: cap,
                start: ptr.as_ptr(),
                end: if cap == 0 {
                    // can't offset off this pointer, it's not allocated!
                    ptr.as_ptr()
                } else {
                    ptr.as_ptr().add(len)
                },
                _marker: PhantomData,
            }
        }
    }
}
```

Here's iterating forward:

```
impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                let result = ptr::read(self.start);
                self.start = self.start.offset(1);
                Some(result)
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let len = (self.end as usize - self.start as usize)
                  / mem::size_of::<T>();
        (len, Some(len))
    }
}
```

And here's iterating backwards.

```
impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                self.end = self.end.offset(-1);
                Some(ptr::read(self.end))
            }
        }
    }
}
```

Because IntoIter takes ownership of its allocation, it needs to implement Drop to free it. However it

also wants to implement Drop to drop any elements it contains that weren't yielded.

```
impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            // drop any remaining elements
            for _ in &mut *self {}
            let layout = Layout::array::<T>(self.cap).unwrap();
            unsafe {
                alloc::dealloc(self.buf.as_ptr() as *mut u8, layout);
            }
        }
    }
}
```

RawVec

We've actually reached an interesting situation here: we've duplicated the logic for specifying a buffer and freeing its memory in Vec and IntoIter. Now that we've implemented it and identified *actual* logic duplication, this is a good time to perform some logic compression.

We're going to abstract out the `(ptr, cap)` pair and give them the logic for allocating, growing, and freeing:

```
struct RawVec<T> {
    ptr: NonNull<T>,
    cap: usize,
    _marker: PhantomData<T>,
}

unsafe impl<T: Send> Send for RawVec<T> {}
unsafe impl<T: Sync> Sync for RawVec<T> {}

impl<T> RawVec<T> {
    fn new() -> Self {
        assert!(mem::size_of::<T>() != 0, "TODO: implement ZST support");
        RawVec {
            ptr: NonNull::dangling(),
            cap: 0,
            _marker: PhantomData,
        }
    }

    fn grow(&mut self) {
        let (new_cap, new_layout) = if self.cap == 0 {
            (1, Layout::array::<T>(1).unwrap())
        } else {
            // This can't overflow because we ensure self.cap <= isize::MAX.
            let new_cap = 2 * self.cap;

            // Layout::array checks that the number of bytes is <= usize::MAX,
            // but this is redundant since old_layout.size() <= isize::MAX,
            // so the `unwrap` should never fail.
            let new_layout = Layout::array::<T>(new_cap).unwrap();
            (new_cap, new_layout)
        };

        // Ensure that the new allocation doesn't exceed `isize::MAX` bytes.
        assert!(new_layout.size() <= isize::MAX as usize, "Allocation too large");

        let new_ptr = if self.cap == 0 {
            unsafe { alloc::alloc(new_layout) }
        } else {
```

```
            let old_layout = Layout::array::<T>(self.cap).unwrap();
            let old_ptr = self.ptr.as_ptr() as *mut u8;
            unsafe { alloc::realloc(old_ptr, old_layout, new_layout.size()) }
        };

        // If allocation fails, `new_ptr` will be null, in which case we abort.
        self.ptr = match NonNull::new(new_ptr as *mut T) {
            Some(p) => p,
            None => alloc::handle_alloc_error(new_layout),
        };
        self.cap = new_cap;
    }
}

impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            let layout = Layout::array::<T>(self.cap).unwrap();
            unsafe {
                alloc::dealloc(self.ptr.as_ptr() as *mut u8, layout);
            }
        }
    }
}
```

And change Vec as follows:

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}

impl<T> Vec<T> {
    fn ptr(&self) -> *mut T {
        self.buf.ptr.as_ptr()
    }

    fn cap(&self) -> usize {
        self.buf.cap
    }

    pub fn new() -> Self {
        Vec {
            buf: RawVec::new(),
            len: 0,
        }
    }

    // push/pop/insert/remove largely unchanged:
    // * `self.ptr.as_ptr() -> self.ptr()`
    // * `self.cap -> self.cap()`
    // * `self.grow() -> self.buf.grow()`
}

impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() {}
        // deallocation is handled by RawVec
    }
}
```

And finally we can really simplify IntoIter:

```
pub struct IntoIter<T> {
    _buf: RawVec<T>, // we don't actually care about this. Just need it to live.
    start: *const T,
    end: *const T,
}

// next and next_back literally unchanged since they never referred to the buf

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        // only need to ensure all our elements are read;
        // buffer will clean itself up afterwards.
        for _ in &mut *self {}
    }
}

impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;
    fn into_iter(self) -> IntoIter<T> {
        unsafe {
            // need to use ptr::read to unsafely move the buf out since it's
            // not Copy, and Vec implements Drop (so we can't destructure it).
            let buf = ptr::read(&self.buf);
            let len = self.len;
            mem::forget(self);

            IntoIter {
                start: buf.ptr.as_ptr(),
                end: if buf.cap == 0 {
                    // can't offset off of a pointer unless it's part of an allocation
                    buf.ptr.as_ptr()
                } else {
                    buf.ptr.as_ptr().add(len)
                },
                _buf: buf,
            }
        }
    }
```

```
}
```

Much better.

Drain

Let's move on to Drain. Drain is largely the same as IntoIter, except that instead of consuming the Vec, it borrows the Vec and leaves its allocation untouched. For now we'll only implement the "basic" full-range version.

```
use std::marker::PhantomData;

struct Drain<'a, T: 'a> {
    // Need to bound the lifetime here, so we do it with `&'a mut Vec<T>`
    // because that's semantically what we contain. We're "just" calling
    // `pop()` and `remove(0)`.
    vec: PhantomData<&'a mut Vec<T>>,
    start: *const T,
    end: *const T,
}

impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
```

-- wait, this is seeming familiar. Let's do some more compression. Both IntoIter and Drain have the exact same structure, let's just factor it out.

```
struct RawValIter<T> {
    start: *const T,
    end: *const T,
}

impl<T> RawValIter<T> {
    // unsafe to construct because it has no associated lifetimes.
    // This is necessary to store a RawValIter in the same struct as
    // its actual allocation. OK since it's a private implementation
    // detail.
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if slice.len() == 0 {
                // if `len = 0`, then this is not actually allocated memory.
                // Need to avoid offsetting because that will give wrong
                // information to LLVM via GEP.
                slice.as_ptr()
            } else {
                slice.as_ptr().add(slice.len())
            }
        }
    }
}

// Iterator and DoubleEndedIterator impls identical to IntoIter.
```

And IntoIter becomes the following:

```
pub struct IntoIter<T> {
    _buf: RawVec<T>, // we don't actually care about this. Just need it to live.
    iter: RawValIter<T>,
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        for _ in &mut *self {}
    }
}

impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;
    fn into_iter(self) -> IntoIter<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            let buf = ptr::read(&self.buf);
            mem::forget(self);

            IntoIter {
                iter: iter,
                _buf: buf,
            }
        }
    }
}
```

Note that I've left a few quirks in this design to make upgrading Drain to work with arbitrary subranges a bit easier. In particular we *could* have RawValIter drain itself on drop, but that won't work right for a more complex Drain. We also take a slice to simplify Drain initialization.

Alright, now Drain is really easy:

```
use std::marker::PhantomData;

pub struct Drain<'a, T: 'a> {
    vec: PhantomData<&'a mut Vec<T>>,
    iter: RawValIter<T>,
}

impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

impl<'a, T> DoubleEndedIterator for Drain<'a, T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<'a, T> Drop for Drain<'a, T> {
    fn drop(&mut self) {
        for _ in &mut *self {}
    }
}

impl<T> Vec<T> {
    pub fn drain(&mut self) -> Drain<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            // this is a mem::forget safety thing. If Drain is forgotten, we just
            // leak the whole Vec's contents. Also we need to do this *eventually*
            // anyway, so why not do it now?
            self.len = 0;

            Drain {
                iter: iter,
                vec: PhantomData,
            }
        }
    }
```

```
}
```

For more details on the `mem::forget` problem, see the section on leaks.

Handling Zero-Sized Types

It's time. We're going to fight the specter that is zero-sized types. Safe Rust *never* needs to care about this, but Vec is very intensive on raw pointers and raw allocations, which are exactly the two things that care about zero-sized types. We need to be careful of two things:

- The raw allocator API has undefined behavior if you pass in 0 for an allocation size.
- raw pointer offsets are no-ops for zero-sized types, which will break our C-style pointer iterator.

Thankfully we abstracted out pointer-iterators and allocating handling into `RawValIter` and `RawVec` respectively. How mysteriously convenient.

Allocating Zero-Sized Types

So if the allocator API doesn't support zero-sized allocations, what on earth do we store as our allocation? `NonNull::dangling()` of course! Almost every operation with a ZST is a no-op since ZSTs have exactly one value, and therefore no state needs to be considered to store or load them. This actually extends to `ptr::read` and `ptr::write`: they won't actually look at the pointer at all. As such we never need to change the pointer.

Note however that our previous reliance on running out of memory before overflow is no longer valid with zero-sized types. We must explicitly guard against capacity overflow for zero-sized types.

Due to our current architecture, all this means is writing 3 guards, one in each method of `RawVec`.

```
impl<T> RawVec<T> {
    fn new() -> Self {
        // !0 is usize::MAX. This branch should be stripped at compile time.
        let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

        // `NonNull::dangling()` doubles as "unallocated" and "zero-sized allocation"
        RawVec {
            ptr: NonNull::dangling(),
            cap: cap,
            _marker: PhantomData,
        }
    }

    fn grow(&mut self) {
        // since we set the capacity to usize::MAX when T has size 0,
        // getting to here necessarily means the Vec is overfull.
        assert!(mem::size_of::<T>() != 0, "capacity overflow");

        let (new_cap, new_layout) = if self.cap == 0 {
            (1, Layout::array::<T>(1).unwrap())
        } else {
            // This can't overflow because we ensure self.cap <= isize::MAX.
            let new_cap = 2 * self.cap;

            // `Layout::array` checks that the number of bytes is <= usize::MAX,
            // but this is redundant since old_layout.size() <= isize::MAX,
            // so the `unwrap` should never fail.
            let new_layout = Layout::array::<T>(new_cap).unwrap();
            (new_cap, new_layout)
        };

        // Ensure that the new allocation doesn't exceed `isize::MAX` bytes.
        assert!(new_layout.size() <= isize::MAX as usize, "Allocation too large");

        let new_ptr = if self.cap == 0 {
            unsafe { alloc::alloc(new_layout) }
        } else {
            let old_layout = Layout::array::<T>(self.cap).unwrap();
            let old_ptr = self.ptr.as_ptr() as *mut u8;
```

```
            unsafe { alloc::realloc(old_ptr, old_layout, new_layout.size()) }
        };

        // If allocation fails, `new_ptr` will be null, in which case we abort.
        self.ptr = match NonNull::new(new_ptr as *mut T) {
            Some(p) => p,
            None => alloc::handle_alloc_error(new_layout),
        };
        self.cap = new_cap;
    }
}

impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        let elem_size = mem::size_of::<T>();

        if self.cap != 0 && elem_size != 0 {
            unsafe {
                alloc::dealloc(
                    self.ptr.as_ptr() as *mut u8,
                    Layout::array::<T>(self.cap).unwrap(),
                );
            }
        }
    }
}
```

That's it. We support pushing and popping zero-sized types now. Our iterators (that aren't provided by slice Deref) are still busted, though.

Iterating Zero-Sized Types

Zero-sized offsets are no-ops. This means that our current design will always initialize `start` and `end` as the same value, and our iterators will yield nothing. The current solution to this is to cast the

pointers to integers, increment, and then cast them back:

```
impl<T> RawValIter<T> {
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if mem::size_of::<T>() == 0 {
                ((slice.as_ptr() as usize) + slice.len()) as *const _
            } else if slice.len() == 0 {
                slice.as_ptr()
            } else {
                slice.as_ptr().add(slice.len())
            },
        }
    }
}
```

Now we have a different bug. Instead of our iterators not running at all, our iterators now run *forever*. We need to do the same trick in our iterator impls. Also, our size_hint computation code will divide by 0 for ZSTs. Since we'll basically be treating the two pointers as if they point to bytes, we'll just map size 0 to divide by 1. Here's what `next` will be:

```
fn next(&mut self) -> Option<T> {
    if self.start == self.end {
        None
    } else {
        unsafe {
            let result = ptr::read(self.start);
            self.start = if mem::size_of::<T>() == 0 {
                (self.start as usize + 1) as *const _
            } else {
                self.start.offset(1)
            };
            Some(result)
        }
    }
}
```

Do you see the "bug"? No one else did! The original author only noticed the problem when linking to this page years later. This code is kind of dubious because abusing the iterator pointers to be *counters* makes them unaligned! Our *one job* when using ZSTs is to keep pointers aligned! *forehead slap*

Raw pointers don't need to be aligned at all times, so the basic trick of using pointers as counters is *fine*, but they *should* definitely be aligned when passed to `ptr::read`! This is *possibly* needless pedantry because `ptr::read` is a noop for a ZST, but let's be a *little* more responsible and read from `NonNull::dangling` on the ZST path.

(Alternatively you could call `read_unaligned` on the ZST path. Either is fine, because either way we're making up a value from nothing and it all compiles to doing nothing.)

```
impl<T> Iterator for RawValIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                if mem::size_of::<T>() == 0 {
                    self.start = (self.start as usize + 1) as *const _;
                    Some(ptr::read(NonNull::<T>::dangling().as_ptr()))
                } else {
                    let old_ptr = self.start;
                    self.start = self.start.offset(1);
                    Some(ptr::read(old_ptr))
                }
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let elem_size = mem::size_of::<T>();
        let len = (self.end as usize - self.start as usize)
                  / if elem_size == 0 { 1 } else { elem_size };
        (len, Some(len))
    }
}

impl<T> DoubleEndedIterator for RawValIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                if mem::size_of::<T>() == 0 {
                    self.end = (self.end as usize - 1) as *const _;
                    Some(ptr::read(NonNull::<T>::dangling().as_ptr()))
                } else {
                    self.end = self.end.offset(-1);
                    Some(ptr::read(self.end))
```

```
                    }
                }
            }
        }
    }
```

And that's it. Iteration works!

The Final Code

```
use std::alloc::{self, Layout};
use std::marker::PhantomData;
use std::mem;
use std::ops::{Deref, DerefMut};
use std::ptr::{self, NonNull};

struct RawVec<T> {
    ptr: NonNull<T>,
    cap: usize,
    _marker: PhantomData<T>,
}

unsafe impl<T: Send> Send for RawVec<T> {}
unsafe impl<T: Sync> Sync for RawVec<T> {}

impl<T> RawVec<T> {
    fn new() -> Self {
        // !0 is usize::MAX. This branch should be stripped at compile time.
        let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

        // `NonNull::dangling()` doubles as "unallocated" and "zero-sized allocation"
        RawVec {
            ptr: NonNull::dangling(),
            cap: cap,
            _marker: PhantomData,
        }
    }

    fn grow(&mut self) {
        // since we set the capacity to usize::MAX when T has size 0,
        // getting to here necessarily means the Vec is overfull.
        assert!(mem::size_of::<T>() != 0, "capacity overflow");

        let (new_cap, new_layout) = if self.cap == 0 {
            (1, Layout::array::<T>(1).unwrap())
        } else {
            // This can't overflow because we ensure self.cap <= isize::MAX.
            let new_cap = 2 * self.cap;
```

```
                // `Layout::array` checks that the number of bytes is <= usize::MAX,
                // but this is redundant since old_layout.size() <= isize::MAX,
                // so the `unwrap` should never fail.
                let new_layout = Layout::array::<T>(new_cap).unwrap();
                (new_cap, new_layout)
            };

            // Ensure that the new allocation doesn't exceed `isize::MAX` bytes.
            assert!(
                new_layout.size() <= isize::MAX as usize,
                "Allocation too large"
            );

            let new_ptr = if self.cap == 0 {
                unsafe { alloc::alloc(new_layout) }
            } else {
                let old_layout = Layout::array::<T>(self.cap).unwrap();
                let old_ptr = self.ptr.as_ptr() as *mut u8;
                unsafe { alloc::realloc(old_ptr, old_layout, new_layout.size()) }
            };

            // If allocation fails, `new_ptr` will be null, in which case we abort.
            self.ptr = match NonNull::new(new_ptr as *mut T) {
                Some(p) => p,
                None => alloc::handle_alloc_error(new_layout),
            };
            self.cap = new_cap;
        }
    }

    impl<T> Drop for RawVec<T> {
        fn drop(&mut self) {
            let elem_size = mem::size_of::<T>();

            if self.cap != 0 && elem_size != 0 {
                unsafe {
                    alloc::dealloc(
                        self.ptr.as_ptr() as *mut u8,
                        Layout::array::<T>(self.cap).unwrap(),
                    );
```

```
            }
        }
    }
}

pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}

impl<T> Vec<T> {
    fn ptr(&self) -> *mut T {
        self.buf.ptr.as_ptr()
    }

    fn cap(&self) -> usize {
        self.buf.cap
    }

    pub fn new() -> Self {
        Vec {
            buf: RawVec::new(),
            len: 0,
        }
    }
    pub fn push(&mut self, elem: T) {
        if self.len == self.cap() {
            self.buf.grow();
        }

        unsafe {
            ptr::write(self.ptr().add(self.len), elem);
        }

        // Can't overflow, we'll OOM first.
        self.len += 1;
    }

    pub fn pop(&mut self) -> Option<T> {
        if self.len == 0 {
```

```
            None
        } else {
            self.len -= 1;
            unsafe { Some(ptr::read(self.ptr().add(self.len))) }
        }
    }

    pub fn insert(&mut self, index: usize, elem: T) {
        assert!(index <= self.len, "index out of bounds");
        if self.cap() == self.len {
            self.buf.grow();
        }

        unsafe {
            ptr::copy(
                self.ptr().add(index),
                self.ptr().add(index + 1),
                self.len - index,
            );
            ptr::write(self.ptr().add(index), elem);
            self.len += 1;
        }
    }

    pub fn remove(&mut self, index: usize) -> T {
        assert!(index < self.len, "index out of bounds");
        unsafe {
            self.len -= 1;
            let result = ptr::read(self.ptr().add(index));
            ptr::copy(
                self.ptr().add(index + 1),
                self.ptr().add(index),
                self.len - index,
            );
            result
        }
    }

    pub fn drain(&mut self) -> Drain<T> {
        unsafe {
```

```
            let iter = RawValIter::new(&self);

            // this is a mem::forget safety thing. If Drain is forgotten, we just
            // leak the whole Vec's contents. Also we need to do this *eventually*
            // anyway, so why not do it now?
            self.len = 0;

            Drain {
                iter: iter,
                vec: PhantomData,
            }
        }
    }
}

impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() {}
        // deallocation is handled by RawVec
    }
}

impl<T> Deref for Vec<T> {
    type Target = [T];
    fn deref(&self) -> &[T] {
        unsafe { std::slice::from_raw_parts(self.ptr(), self.len) }
    }
}

impl<T> DerefMut for Vec<T> {
    fn deref_mut(&mut self) -> &mut [T] {
        unsafe { std::slice::from_raw_parts_mut(self.ptr(), self.len) }
    }
}

impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;
    fn into_iter(self) -> IntoIter<T> {
        unsafe {
```

```
            let iter = RawValIter::new(&self);
            let buf = ptr::read(&self.buf);
            mem::forget(self);

            IntoIter {
                iter: iter,
                _buf: buf,
            }
        }
    }
}

struct RawValIter<T> {
    start: *const T,
    end: *const T,
}

impl<T> RawValIter<T> {
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if mem::size_of::<T>() == 0 {
                ((slice.as_ptr() as usize) + slice.len()) as *const _
            } else if slice.len() == 0 {
                slice.as_ptr()
            } else {
                slice.as_ptr().add(slice.len())
            },
        }
    }
}

impl<T> Iterator for RawValIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                if mem::size_of::<T>() == 0 {
```

```
                    self.start = (self.start as usize + 1) as *const _;
                    Some(ptr::read(NonNull::<T>::dangling().as_ptr()))
                } else {
                    let old_ptr = self.start;
                    self.start = self.start.offset(1);
                    Some(ptr::read(old_ptr))
                }
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let elem_size = mem::size_of::<T>();
        let len = (self.end as usize - self.start as usize)
                  / if elem_size == 0 { 1 } else { elem_size };
        (len, Some(len))
    }
}

impl<T> DoubleEndedIterator for RawValIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                if mem::size_of::<T>() == 0 {
                    self.end = (self.end as usize - 1) as *const _;
                    Some(ptr::read(NonNull::<T>::dangling().as_ptr()))
                } else {
                    self.end = self.end.offset(-1);
                    Some(ptr::read(self.end))
                }
            }
        }
    }
}

pub struct IntoIter<T> {
    _buf: RawVec<T>, // we don't actually care about this. Just need it to live.
    iter: RawValIter<T>,
```

```
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        self.iter.next()
    }
    fn size_hint(&self) -> (usize, Option<usize>) {
        self.iter.size_hint()
    }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> {
        self.iter.next_back()
    }
}

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        for _ in &mut *self {}
    }
}

pub struct Drain<'a, T: 'a> {
    vec: PhantomData<&'a mut Vec<T>>,
    iter: RawValIter<T>,
}

impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        self.iter.next()
    }
    fn size_hint(&self) -> (usize, Option<usize>) {
        self.iter.size_hint()
    }
}

impl<'a, T> DoubleEndedIterator for Drain<'a, T> {
```

```
    fn next_back(&mut self) -> Option<T> {
        self.iter.next_back()
    }
}

impl<'a, T> Drop for Drain<'a, T> {
    fn drop(&mut self) {
        // pre-drain the iter
        for _ in &mut *self {}
    }
}
```

Implementing Arc and Mutex

Knowing the theory is all fine and good, but the *best* way to understand something is to use it. To better understand atomics and interior mutability, we'll be implementing versions of the standard library's `Arc` and `Mutex` types.

TODO: Write `Mutex` chapters.

Implementing Arc

In this section, we'll be implementing a simpler version of `std::sync::Arc`. Similarly to the implementation of `Vec` we made earlier, we won't be taking advantage of as many optimizations, intrinsics, or unstable code as the standard library may.

This implementation is loosely based on the standard library's implementation (technically taken from `alloc::sync` in 1.49, as that's where it's actually implemented), but it will not support weak references at the moment as they make the implementation slightly more complex.

Please note that this section is very work-in-progress at the moment.

Layout

Let's start by making the layout for our implementation of `Arc`.

An `Arc<T>` provides thread-safe shared ownership of a value of type `T`, allocated in the heap. Sharing implies immutability in Rust, so we don't need to design anything that manages access to that value, right? Although interior mutability types like Mutex allow Arc's users to create shared mutability, Arc itself doesn't need to concern itself with these issues.

However there *is* one place where Arc needs to concern itself with mutation: destruction. When all the owners of the Arc go away, we need to be able to `drop` its contents and free its allocation. So we need a way for an owner to know if it's the *last* owner, and the simplest way to do that is with a count of the owners -- Reference Counting.

Unfortunately, this reference count is inherently shared mutable state, so Arc *does* need to think about synchronization. We *could* use a Mutex for this, but that's overkill. Instead, we'll use atomics. And since everyone already needs a pointer to the T's allocation, we might as well put the reference count in that same allocation.

Naively, it would look something like this:

```
use std::sync::atomic;

pub struct Arc<T> {
    ptr: *mut ArcInner<T>,
}

pub struct ArcInner<T> {
    rc: atomic::AtomicUsize,
    data: T,
}
```

This would compile, however it would be incorrect. First of all, the compiler will give us too strict variance. For example, an `Arc<&'static str>` couldn't be used where an `Arc<&'a str>` was expected. More importantly, it will give incorrect ownership information to the drop checker, as it will assume we don't own any values of type `T`. As this is a structure providing shared ownership of a value, at some point there will be an instance of this structure that entirely owns its data. See the chapter on ownership and lifetimes for all the details on variance and drop check.

To fix the first problem, we can use `NonNull<T>`. Note that `NonNull<T>` is a wrapper around a raw pointer that declares that:

- We are covariant over `T`
- Our pointer is never null

To fix the second problem, we can include a `PhantomData` marker containing an `ArcInner<T>`. This will tell the drop checker that we have some notion of ownership of a value of `ArcInner<T>` (which itself contains some `T`).

With these changes we get our final structure:

```
use std::marker::PhantomData;
use std::ptr::NonNull;
use std::sync::atomic::AtomicUsize;

pub struct Arc<T> {
    ptr: NonNull<ArcInner<T>>,
    phantom: PhantomData<ArcInner<T>>,
}

pub struct ArcInner<T> {
    rc: AtomicUsize,
    data: T,
}
```

Base Code

Now that we've decided the layout for our implementation of `Arc`, let's create some basic code.

Constructing the Arc

We'll first need a way to construct an `Arc<T>`.

This is pretty simple, as we just need to box the `ArcInner<T>` and get a `NonNull<T>` pointer to it.

```
impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T> {
        // We start the reference count at 1, as that first reference is the
        // current pointer.
        let boxed = Box::new(ArcInner {
            rc: AtomicUsize::new(1),
            data,
        });
        Arc {
            // It is okay to call `.unwrap()` here as we get a pointer from
            // `Box::into_raw` which is guaranteed to not be null.
            ptr: NonNull::new(Box::into_raw(boxed)).unwrap(),
            phantom: PhantomData,
        }
    }
}
```

Send and Sync

Since we're building a concurrency primitive, we'll need to be able to send it across threads. Thus, we can implement the `Send` and `Sync` marker traits. For more information on these, see the section on `Send` and `Sync`.

This is okay because:

- You can only get a mutable reference to the value inside an `Arc` if and only if it is the only `Arc` referencing that data (which only happens in `Drop`)
- We use atomics for the shared mutable reference counting

```
unsafe impl<T: Sync + Send> Send for Arc<T> {}
unsafe impl<T: Sync + Send> Sync for Arc<T> {}
```

We need to have the bound `T: Sync + Send` because if we did not provide those bounds, it would be possible to share values that are thread-unsafe across a thread boundary via an `Arc`, which could possibly cause data races or unsoundness.

For example, if those bounds were not present, `Arc<Rc<u32>>` would be `Sync` or `Send`, meaning that you could clone the `Rc` out of the `Arc` to send it across a thread (without creating an entirely new `Rc`), which would create data races as `Rc` is not thread-safe.

Getting the `ArcInner`

To dereference the `NonNull<T>` pointer into a `&T`, we can call `NonNull::as_ref`. This is unsafe, unlike the typical `as_ref` function, so we must call it like this:

```
unsafe { self.ptr.as_ref() }
```

We'll be using this snippet a few times in this code (usually with an associated `let` binding).

This unsafety is okay because while this `Arc` is alive, we're guaranteed that the inner pointer is valid.

Deref

Alright. Now we can make `Arc`s (and soon will be able to clone and destroy them correctly), but how do we get to the data inside?

What we need now is an implementation of `Deref`.

We'll need to import the trait:

```
use std::ops::Deref;
```

And here's the implementation:

```
impl<T> Deref for Arc<T> {
    type Target = T;

    fn deref(&self) -> &T {
        let inner = unsafe { self.ptr.as_ref() };
        &inner.data
    }
}
```

Pretty simple, eh? This simply dereferences the `NonNull` pointer to the `ArcInner<T>`, then gets a reference to the data inside.

Code

Here's all the code from this section:

```
use std::ops::Deref;

impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T> {
        // We start the reference count at 1, as that first reference is the
        // current pointer.
        let boxed = Box::new(ArcInner {
            rc: AtomicUsize::new(1),
            data,
        });
        Arc {
            // It is okay to call `.unwrap()` here as we get a pointer from
            // `Box::into_raw` which is guaranteed to not be null.
            ptr: NonNull::new(Box::into_raw(boxed)).unwrap(),
            phantom: PhantomData,
        }
    }
}

unsafe impl<T: Sync + Send> Send for Arc<T> {}
unsafe impl<T: Sync + Send> Sync for Arc<T> {}


impl<T> Deref for Arc<T> {
    type Target = T;

    fn deref(&self) -> &T {
        let inner = unsafe { self.ptr.as_ref() };
        &inner.data
    }
}
```

Cloning

Now that we've got some basic code set up, we'll need a way to clone the `Arc`.

Basically, we need to:

1. Increment the atomic reference count
2. Construct a new instance of the `Arc` from the inner pointer

First, we need to get access to the `ArcInner`:

```
let inner = unsafe { self.ptr.as_ref() };
```

We can update the atomic reference count as follows:

```
let old_rc = inner.rc.fetch_add(1, Ordering::???);
```

But what ordering should we use here? We don't really have any code that will need atomic synchronization when cloning, as we do not modify the internal value while cloning. Thus, we can use a Relaxed ordering here, which implies no happens-before relationship but is atomic. When `Drop` ping the Arc, however, we'll need to atomically synchronize when decrementing the reference count. This is described more in the section on the `Drop` implementation for `Arc`. For more information on atomic relationships and Relaxed ordering, see the section on atomics.

Thus, the code becomes this:

```
let old_rc = inner.rc.fetch_add(1, Ordering::Relaxed);
```

We'll need to add another import to use `Ordering`:

```
use std::sync::atomic::Ordering;
```

However, we have one problem with this implementation right now. What if someone decides to `mem::forget` a bunch of Arcs? The code we have written so far (and will write) assumes that the reference count accurately portrays how many Arcs are in memory, but with `mem::forget` this is false. Thus, when more and more Arcs are cloned from this one without them being `Drop` ped and the reference count being decremented, we can overflow! This will cause use-after-free which is **INCREDIBLY BAD!**

To handle this, we need to check that the reference count does not go over some arbitrary value (below `usize::MAX`, as we're storing the reference count as an `AtomicUsize`), and do *something*.

The standard library's implementation decides to just abort the program (as it is an incredibly unlikely case in normal code and if it happens, the program is probably incredibly degenerate) if the reference count reaches `isize::MAX` (about half of `usize::MAX`) on any thread, on the assumption that there are probably not about 2 billion threads (or about **9 quintillion** on some 64-bit machines) incrementing the reference count at once. This is what we'll do.

It's pretty simple to implement this behavior:

```
if old_rc >= isize::MAX as usize {
    std::process::abort();
}
```

Then, we need to return a new instance of the `Arc`:

```
Self {
    ptr: self.ptr,
    phantom: PhantomData
}
```

Now, let's wrap this all up inside the `Clone` implementation:

```
use std::sync::atomic::Ordering;

impl<T> Clone for Arc<T> {
    fn clone(&self) -> Arc<T> {
        let inner = unsafe { self.ptr.as_ref() };
        // Using a relaxed ordering is alright here as we don't need any atomic
        // synchronization here as we're not modifying or accessing the inner
        // data.
        let old_rc = inner.rc.fetch_add(1, Ordering::Relaxed);

        if old_rc >= isize::MAX as usize {
            std::process::abort();
        }

        Self {
            ptr: self.ptr,
            phantom: PhantomData,
        }
    }
}
```

Dropping

We now need a way to decrease the reference count and drop the data once it is low enough, otherwise the data will live forever on the heap.

To do this, we can implement `Drop`.

Basically, we need to:

1. Decrement the reference count
2. If there is only one reference remaining to the data, then:
3. Atomically fence the data to prevent reordering of the use and deletion of the data
4. Drop the inner data

First, we'll need to get access to the `ArcInner`:

```
let inner = unsafe { self.ptr.as_ref() };
```

Now, we need to decrement the reference count. To streamline our code, we can also return if the returned value from `fetch_sub` (the value of the reference count before decrementing it) is not equal to `1` (which happens when we are not the last reference to the data).

```
if inner.rc.fetch_sub(1, Ordering::Release) != 1 {
    return;
}
```

We then need to create an atomic fence to prevent reordering of the use of the data and deletion of the data. As described in the standard library's implementation of `Arc`:

> This fence is needed to prevent reordering of use of the data and deletion of the data. Because it is marked `Release`, the decreasing of the reference count synchronizes with this `Acquire`

fence. This means that use of the data happens before decreasing the reference count, which happens before this fence, which happens before the deletion of the data.

As explained in the Boost documentation,

> It is important to enforce any possible access to the object in one thread (through an existing reference) to *happen before* deleting the object in a different thread. This is achieved by a "release" operation after dropping a reference (any access to the object through this reference must obviously happened before), and an "acquire" operation before deleting the object.

In particular, while the contents of an Arc are usually immutable, it's possible to have interior writes to something like a Mutex. Since a Mutex is not acquired when it is deleted, we can't rely on its synchronization logic to make writes in thread A visible to a destructor running in thread B.

Also note that the Acquire fence here could probably be replaced with an Acquire load, which could improve performance in highly-contended situations. See 2.

To do this, we do the following:

```
use std::sync::atomic;
atomic::fence(Ordering::Acquire);
```

Finally, we can drop the data itself. We use `Box::from_raw` to drop the boxed `ArcInner<T>` and its data. This takes a `*mut T` and not a `NonNull<T>`, so we must convert using `NonNull::as_ptr`.

```
unsafe { Box::from_raw(self.ptr.as_ptr()); }
```

This is safe as we know we have the last pointer to the `ArcInner` and that its pointer is valid.

Now, let's wrap this all up inside the `Drop` implementation:

```
impl<T> Drop for Arc<T> {
    fn drop(&mut self) {
        let inner = unsafe { self.ptr.as_ref() };
        if inner.rc.fetch_sub(1, Ordering::Release) != 1 {
            return;
        }
        // This fence is needed to prevent reordering of the use and deletion
        // of the data.
        atomic::fence(Ordering::Acquire);
        // This is safe as we know we have the last pointer to the `ArcInner`
        // and that its pointer is valid.
        unsafe { Box::from_raw(self.ptr.as_ptr()); }
    }
}
```

Final Code

Here's the final code, with some added comments and re-ordered imports:

```
use std::marker::PhantomData;
use std::ops::Deref;
use std::ptr::NonNull;
use std::sync::atomic::{self, AtomicUsize, Ordering};

pub struct Arc<T> {
    ptr: NonNull<ArcInner<T>>,
    phantom: PhantomData<ArcInner<T>>,
}

pub struct ArcInner<T> {
    rc: AtomicUsize,
    data: T,
}

impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T> {
        // We start the reference count at 1, as that first reference is the
        // current pointer.
        let boxed = Box::new(ArcInner {
            rc: AtomicUsize::new(1),
            data,
        });
        Arc {
            // It is okay to call `.unwrap()` here as we get a pointer from
            // `Box::into_raw` which is guaranteed to not be null.
            ptr: NonNull::new(Box::into_raw(boxed)).unwrap(),
            phantom: PhantomData,
        }
    }
}

unsafe impl<T: Sync + Send> Send for Arc<T> {}
unsafe impl<T: Sync + Send> Sync for Arc<T> {}

impl<T> Deref for Arc<T> {
    type Target = T;
```

```
    fn deref(&self) -> &T {
        let inner = unsafe { self.ptr.as_ref() };
        &inner.data
    }
}

impl<T> Clone for Arc<T> {
    fn clone(&self) -> Arc<T> {
        let inner = unsafe { self.ptr.as_ref() };
        // Using a relaxed ordering is alright here as we don't need any atomic
        // synchronization here as we're not modifying or accessing the inner
        // data.
        let old_rc = inner.rc.fetch_add(1, Ordering::Relaxed);

        if old_rc >= isize::MAX as usize {
            std::process::abort();
        }

        Self {
            ptr: self.ptr,
            phantom: PhantomData,
        }
    }
}

impl<T> Drop for Arc<T> {
    fn drop(&mut self) {
        let inner = unsafe { self.ptr.as_ref() };
        if inner.rc.fetch_sub(1, Ordering::Release) != 1 {
            return;
        }
        // This fence is needed to prevent reordering of the use and deletion
        // of the data.
        atomic::fence(Ordering::Acquire);
        // This is safe as we know we have the last pointer to the `ArcInner`
        // and that its pointer is valid.
        unsafe { Box::from_raw(self.ptr.as_ptr()); }
    }
}
```

Foreign Function Interface

Introduction

This guide will use the snappy compression/decompression library as an introduction to writing bindings for foreign code. Rust is currently unable to call directly into a C++ library, but snappy includes a C interface (documented in `snappy-c.h`).

A note about libc

Many of these examples use the `libc` crate, which provides various type definitions for C types, among other things. If you're trying these examples yourself, you'll need to add `libc` to your `Cargo.toml`:

```
[dependencies]
libc = "0.2.0"
```

Calling foreign functions

The following is a minimal example of calling a foreign function which will compile if snappy is installed:

```
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x);
}
```

The `extern` block is a list of function signatures in a foreign library, in this case with the platform's C ABI. The `#[link(...)]` attribute is used to instruct the linker to link against the snappy library so the symbols are resolved.

Foreign functions are assumed to be unsafe so calls to them need to be wrapped with `unsafe {}` as a promise to the compiler that everything contained within truly is safe. C libraries often expose interfaces that aren't thread-safe, and almost any function that takes a pointer argument isn't valid for all possible inputs since the pointer could be dangling, and raw pointers fall outside of Rust's safe memory model.

When declaring the argument types to a foreign function, the Rust compiler cannot check if the declaration is correct, so specifying it correctly is part of keeping the binding correct at runtime.

The `extern` block can be extended to cover the entire snappy API:

```
use libc::{c_int, size_t};

#[link(name = "snappy")]
extern {
    fn snappy_compress(input: *const u8,
                       input_length: size_t,
                       compressed: *mut u8,
                       compressed_length: *mut size_t) -> c_int;
    fn snappy_uncompress(compressed: *const u8,
                         compressed_length: size_t,
                         uncompressed: *mut u8,
                         uncompressed_length: *mut size_t) -> c_int;
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
    fn snappy_uncompressed_length(compressed: *const u8,
                                  compressed_length: size_t,
                                  result: *mut size_t) -> c_int;
    fn snappy_validate_compressed_buffer(compressed: *const u8,
                                         compressed_length: size_t) -> c_int;
}
```

Creating a safe interface

The raw C API needs to be wrapped to provide memory safety and make use of higher-level concepts like vectors. A library can choose to expose only the safe, high-level interface and hide the unsafe internal details.

Wrapping the functions which expect buffers involves using the `slice::raw` module to manipulate Rust vectors as pointers to memory. Rust's vectors are guaranteed to be a contiguous block of memory. The length is the number of elements currently contained, and the capacity is the total size in elements of the allocated memory. The length is less than or equal to the capacity.

```
pub fn validate_compressed_buffer(src: &[u8]) -> bool {
    unsafe {
        snappy_validate_compressed_buffer(src.as_ptr(), src.len() as size_t) == 0
    }
}
```

The `validate_compressed_buffer` wrapper above makes use of an `unsafe` block, but it makes the guarantee that calling it is safe for all inputs by leaving off `unsafe` from the function signature.

The `snappy_compress` and `snappy_uncompress` functions are more complex, since a buffer has to be allocated to hold the output too.

The `snappy_max_compressed_length` function can be used to allocate a vector with the maximum required capacity to hold the compressed output. The vector can then be passed to the `snappy_compress` function as an output parameter. An output parameter is also passed to retrieve the true length after compression for setting the length.

```
pub fn compress(src: &[u8]) -> Vec<u8> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen = snappy_max_compressed_length(srclen);
        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        snappy_compress(psrc, srclen, pdst, &mut dstlen);
        dst.set_len(dstlen as usize);
        dst
    }
}
```

Decompression is similar, because snappy stores the uncompressed size as part of the compression format and `snappy_uncompressed_length` will retrieve the exact buffer size required.

```
pub fn uncompress(src: &[u8]) -> Option<Vec<u8>> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen: size_t = 0;
        snappy_uncompressed_length(psrc, srclen, &mut dstlen);

        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        if snappy_uncompress(psrc, srclen, pdst, &mut dstlen) == 0 {
            dst.set_len(dstlen as usize);
            Some(dst)
        } else {
            None // SNAPPY_INVALID_INPUT
        }
    }
}
```

Then, we can add some tests to show how to use them.

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn valid() {
        let d = vec![0xde, 0xad, 0xd0, 0x0d];
        let c: &[u8] = &compress(&d);
        assert!(validate_compressed_buffer(c));
        assert!(uncompress(c) == Some(d));
    }

    #[test]
    fn invalid() {
        let d = vec![0, 0, 0, 0];
        assert!(!validate_compressed_buffer(&d));
        assert!(uncompress(&d).is_none());
    }

    #[test]
    fn empty() {
        let d = vec![];
        assert!(!validate_compressed_buffer(&d));
        assert!(uncompress(&d).is_none());
        let c = compress(&d);
        assert!(validate_compressed_buffer(&c));
        assert!(uncompress(&c) == Some(d));
    }
}
```

Destructors

Foreign libraries often hand off ownership of resources to the calling code. When this occurs, we must use Rust's destructors to provide safety and guarantee the release of these resources (especially in the case of panic).

For more about destructors, see the Drop trait.

Calling Rust code from C

You may wish to compile Rust code in a way so that it can be called from C. This is fairly easy, but requires a few things.

Rust side

First, we assume you have a lib crate named as `rust_from_c`. `lib.rs` should have Rust code as following:

```
#[no_mangle]
pub extern "C" fn hello_from_rust() {
    println!("Hello from Rust!");
}
```

The `extern "C"` makes this function adhere to the C calling convention, as discussed above in "Foreign Calling Conventions". The `no_mangle` attribute turns off Rust's name mangling, so that it has a well defined symbol to link to.

Then, to compile Rust code as a shared library that can be called from C, add the following to your `Cargo.toml`:

```
[lib]
crate-type = ["cdylib"]
```

(NOTE: We could also use the `staticlib` crate type but it needs to tweak some linking flags.)

Run `cargo build` and you're ready to go on the Rust side.

C side

We'll create a C file to call the `hello_from_rust` function and compile it by `gcc`.

C file should look like:

```
extern void hello_from_rust();

int main(void) {
    hello_from_rust();
    return 0;
}
```

We name the file as `call_rust.c` and place it on the crate root. Run the following to compile:

```
gcc call_rust.c -o call_rust -lrust_from_c -L./target/debug
```

`-l` and `-L` tell gcc to find our Rust library.

Finally, we can call Rust code from C with `LD_LIBRARY_PATH` specified:

```
$ LD_LIBRARY_PATH=./target/debug ./call_rust
Hello from Rust!
```

That's it! For more realistic example, check the `cbindgen`.

Callbacks from C code to Rust functions

Some external libraries require the usage of callbacks to report back their current state or intermediate data to the caller. It is possible to pass functions defined in Rust to an external library. The requirement for this is that the callback function is marked as `extern` with the correct calling convention to make it callable from C code.

The callback function can then be sent through a registration call to the C library and afterwards be invoked from there.

A basic example is:

Rust code:

```
extern fn callback(a: i32) {
    println!("I'm called from C with value {0}", a);
}

#[link(name = "extlib")]
extern {
   fn register_callback(cb: extern fn(i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    unsafe {
        register_callback(callback);
        trigger_callback(); // Triggers the callback.
    }
}
```

C code:

```
typedef void (*rust_callback)(int32_t);
rust_callback cb;

int32_t register_callback(rust_callback callback) {
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(7); // Will call callback(7) in Rust.
}
```

In this example Rust's `main()` will call `trigger_callback()` in C, which would, in turn, call back to `callback()` in Rust.

Targeting callbacks to Rust objects

The former example showed how a global function can be called from C code. However it is often desired that the callback is targeted to a special Rust object. This could be the object that represents the wrapper for the respective C object.

This can be achieved by passing a raw pointer to the object down to the C library. The C library can then include the pointer to the Rust object in the notification. This will allow the callback to unsafely access the referenced Rust object.

Rust code:

```
struct RustObject {
    a: i32,
    // Other members...
}

extern "C" fn callback(target: *mut RustObject, a: i32) {
    println!("I'm called from C with value {0}", a);
    unsafe {
        // Update the value in RustObject with the value received from the callback:
        (*target).a = a;
    }
}

#[link(name = "extlib")]
extern {
   fn register_callback(target: *mut RustObject,
                        cb: extern fn(*mut RustObject, i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    // Create the object that will be referenced in the callback:
    let mut rust_object = Box::new(RustObject { a: 5 });

    unsafe {
        register_callback(&mut *rust_object, callback);
        trigger_callback();
    }
}
```

C code:

```
typedef void (*rust_callback)(void*, int32_t);
void* cb_target;
rust_callback cb;

int32_t register_callback(void* callback_target, rust_callback callback) {
    cb_target = callback_target;
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(cb_target, 7); // Will call callback(&rustObject, 7) in Rust.
}
```

Asynchronous callbacks

In the previously given examples the callbacks are invoked as a direct reaction to a function call to the external C library. The control over the current thread is switched from Rust to C to Rust for the execution of the callback, but in the end the callback is executed on the same thread that called the function which triggered the callback.

Things get more complicated when the external library spawns its own threads and invokes callbacks from there. In these cases access to Rust data structures inside the callbacks is especially unsafe and proper synchronization mechanisms must be used. Besides classical synchronization mechanisms like mutexes, one possibility in Rust is to use channels (in `std::sync::mpsc`) to forward data from the C thread that invoked the callback into a Rust thread.

If an asynchronous callback targets a special object in the Rust address space it is also absolutely necessary that no more callbacks are performed by the C library after the respective Rust object gets destroyed. This can be achieved by unregistering the callback in the object's destructor and designing the library in a way that guarantees that no callback will be performed after deregistration.

Linking

The `link` attribute on `extern` blocks provides the basic building block for instructing rustc how it will link to native libraries. There are two accepted forms of the link attribute today:

- `#[link(name = "foo")]`
- `#[link(name = "foo", kind = "bar")]`

In both of these cases, `foo` is the name of the native library that we're linking to, and in the second case `bar` is the type of native library that the compiler is linking to. There are currently three known types of native libraries:

- Dynamic - `#[link(name = "readline")]`
- Static - `#[link(name = "my_build_dependency", kind = "static")]`
- Frameworks - `#[link(name = "CoreFoundation", kind = "framework")]`

Note that frameworks are only available on macOS targets.

The different `kind` values are meant to differentiate how the native library participates in linkage. From a linkage perspective, the Rust compiler creates two flavors of artifacts: partial (rlib/staticlib) and final (dylib/binary). Native dynamic library and framework dependencies are propagated to the final artifact boundary, while static library dependencies are not propagated at all, because the static libraries are integrated directly into the subsequent artifact.

A few examples of how this model can be used are:

- A native build dependency. Sometimes some C/C++ glue is needed when writing some Rust code, but distribution of the C/C++ code in a library format is a burden. In this case, the code will be archived into `libfoo.a` and then the Rust crate would declare a dependency via `#[link(name = "foo", kind = "static")]`.

  Regardless of the flavor of output for the crate, the native static library will be included in the

output, meaning that distribution of the native static library is not necessary.

- A normal dynamic dependency. Common system libraries (like `readline`) are available on a large number of systems, and often a static copy of these libraries cannot be found. When this dependency is included in a Rust crate, partial targets (like rlibs) will not link to the library, but when the rlib is included in a final target (like a binary), the native library will be linked in.

On macOS, frameworks behave with the same semantics as a dynamic library.

Unsafe blocks

Some operations, like dereferencing raw pointers or calling functions that have been marked unsafe are only allowed inside unsafe blocks. Unsafe blocks isolate unsafety and are a promise to the compiler that the unsafety does not leak out of the block.

Unsafe functions, on the other hand, advertise it to the world. An unsafe function is written like this:

```
unsafe fn kaboom(ptr: *const i32) -> i32 { *ptr }
```

This function can only be called from an `unsafe` block or another `unsafe` function.

Accessing foreign globals

Foreign APIs often export a global variable which could do something like track global state. In order to access these variables, you declare them in `extern` blocks with the `static` keyword:

```
#[link(name = "readline")]
extern {
    static rl_readline_version: libc::c_int;
}

fn main() {
    println!("You have readline version {} installed.",
             unsafe { rl_readline_version as i32 });
}
```

Alternatively, you may need to alter global state provided by a foreign interface. To do this, statics can be declared with `mut` so we can mutate them.

```
use std::ffi::CString;
use std::ptr;

#[link(name = "readline")]
extern {
    static mut rl_prompt: *const libc::c_char;
}

fn main() {
    let prompt = CString::new("[my-awesome-shell] $").unwrap();
    unsafe {
        rl_prompt = prompt.as_ptr();

        println!("{:?}", rl_prompt);

        rl_prompt = ptr::null();
    }
}
```

Note that all interaction with a `static mut` is unsafe, both reading and writing. Dealing with global mutable state requires a great deal of care.

Foreign calling conventions

Most foreign code exposes a C ABI, and Rust uses the platform's C calling convention by default when calling foreign functions. Some foreign functions, most notably the Windows API, use other calling conventions. Rust provides a way to tell the compiler which convention to use:

```
#[cfg(all(target_os = "win32", target_arch = "x86"))]
#[link(name = "kernel32")]
#[allow(non_snake_case)]
extern "stdcall" {
    fn SetEnvironmentVariableA(n: *const u8, v: *const u8) -> libc::c_int;
}
```

This applies to the entire `extern` block. The list of supported ABI constraints are:

- `stdcall`
- `aapcs`
- `cdecl`
- `fastcall`
- `vectorcall` This is currently hidden behind the `abi_vectorcall` gate and is subject to change.
- `Rust`
- `rust-intrinsic`
- `system`
- `C`
- `win64`
- `sysv64`

Most of the abis in this list are self-explanatory, but the `system` abi may seem a little odd. This constraint selects whatever the appropriate ABI is for interoperating with the target's libraries. For example, on win32 with a x86 architecture, this means that the abi used would be `stdcall`. On x86_64, however, windows uses the `C` calling convention, so `C` would be used. This means that in

our previous example, we could have used `extern "system" { ... }` to define a block for all windows systems, not only x86 ones.

Interoperability with foreign code

Rust guarantees that the layout of a `struct` is compatible with the platform's representation in C only if the `#[repr(C)]` attribute is applied to it. `#[repr(C, packed)]` can be used to lay out struct members without padding. `#[repr(C)]` can also be applied to an enum.

Rust's owned boxes (`Box<T>`) use non-nullable pointers as handles which point to the contained object. However, they should not be manually created because they are managed by internal allocators. References can safely be assumed to be non-nullable pointers directly to the type. However, breaking the borrow checking or mutability rules is not guaranteed to be safe, so prefer using raw pointers (`*`) if that's needed because the compiler can't make as many assumptions about them.

Vectors and strings share the same basic memory layout, and utilities are available in the `vec` and `str` modules for working with C APIs. However, strings are not terminated with `\0` . If you need a NUL-terminated string for interoperability with C, you should use the `CString` type in the `std::ffi` module.

The `libc` crate on crates.io includes type aliases and function definitions for the C standard library in the `libc` module, and Rust links against `libc` and `libm` by default.

Variadic functions

In C, functions can be 'variadic', meaning they accept a variable number of arguments. This can be

achieved in Rust by specifying `...` within the argument list of a foreign function declaration:

```
extern {
    fn foo(x: i32, ...);
}

fn main() {
    unsafe {
        foo(10, 20, 30, 40, 50);
    }
}
```

Normal Rust functions can *not* be variadic:

```
// This will not compile

fn foo(x: i32, ...) {}
```

The "nullable pointer optimization"

Certain Rust types are defined to never be `null`. This includes references (`&T`, `&mut T`), boxes (`Box<T>`), and function pointers (`extern "abi" fn()`). When interfacing with C, pointers that might be `null` are often used, which would seem to require some messy `transmute`s and/or unsafe code to handle conversions to/from Rust types. However, the language provides a workaround.

As a special case, an `enum` is eligible for the "nullable pointer optimization" if it contains exactly two variants, one of which contains no data and the other contains a field of one of the non-nullable types listed above. This means no extra space is required for a discriminant; rather, the empty variant is represented by putting a `null` value into the non-nullable field. This is called an "optimization", but unlike other optimizations it is guaranteed to apply to eligible types.

The most common type that takes advantage of the nullable pointer optimization is `Option<T>`, where `None` corresponds to `null`. So `Option<extern "C" fn(c_int) -> c_int>` is a correct way to represent a nullable function pointer using the C ABI (corresponding to the C type `int (*)(int)`).

Here is a contrived example. Let's say some C library has a facility for registering a callback, which gets called in certain situations. The callback is passed a function pointer and an integer and it is supposed to run the function with the integer as a parameter. So we have function pointers flying across the FFI boundary in both directions.

```
use libc::c_int;

extern "C" {
    /// Registers the callback.
    fn register(cb: Option<extern "C" fn(Option<extern "C" fn(c_int) -> c_int>, c_int)
-> c_int>);
}

/// This fairly useless function receives a function pointer and an integer
/// from C, and returns the result of calling the function with the integer.
/// In case no function is provided, it squares the integer by default.
extern "C" fn apply(process: Option<extern "C" fn(c_int) -> c_int>, int: c_int) ->
c_int {
    match process {
        Some(f) => f(int),
        None    => int * int
    }
}

fn main() {
    unsafe {
        register(Some(apply));
    }
}
```

And the code on the C side looks like this:

```
void register(int (*f)(int (*)(int), int)) {
    ...
}
```

No `transmute` required!

FFI and panics

It's important to be mindful of `panic!` s when working with FFI. A `panic!` across an FFI boundary is undefined behavior. If you're writing code that may panic, you should run it in a closure with `catch_unwind` :

```
use std::panic::catch_unwind;

#[no_mangle]
pub extern fn oh_no() -> i32 {
    let result = catch_unwind(|| {
        panic!("Oops!");
    });
    match result {
        Ok(_) => 0,
        Err(_) => 1,
    }
}

fn main() {}
```

Please note that `catch_unwind` will only catch unwinding panics, not those who abort the process. See the documentation of `catch_unwind` for more information.

Representing opaque structs

Sometimes, a C library wants to provide a pointer to something, but not let you know the internal details of the thing it wants. A stable and simple way is to use a `void *` argument:

```
void foo(void *arg);
void bar(void *arg);
```

We can represent this in Rust with the `c_void` type:

```
extern "C" {
    pub fn foo(arg: *mut libc::c_void);
    pub fn bar(arg: *mut libc::c_void);
}
```

This is a perfectly valid way of handling the situation. However, we can do a bit better. To solve this, some C libraries will instead create a `struct`, where the details and memory layout of the struct are private. This gives some amount of type safety. These structures are called 'opaque'. Here's an example, in C:

```
struct Foo; /* Foo is a structure, but its contents are not part of the public
interface */
struct Bar;
void foo(struct Foo *arg);
void bar(struct Bar *arg);
```

To do this in Rust, let's create our own opaque types:

```
#[repr(C)]
pub struct Foo {
    _data: [u8; 0],
    _marker:
        core::marker::PhantomData<(*mut u8, core::marker::PhantomPinned)>,
}
#[repr(C)]
pub struct Bar {
    _data: [u8; 0],
    _marker:
        core::marker::PhantomData<(*mut u8, core::marker::PhantomPinned)>,
}

extern "C" {
    pub fn foo(arg: *mut Foo);
    pub fn bar(arg: *mut Bar);
}
```

By including at least one private field and no constructor, we create an opaque type that we can't instantiate outside of this module. (A struct with no field could be instantiated by anyone.) We also want to use this type in FFI, so we have to add `#[repr(C)]`. The marker ensures the compiler does not mark the struct as `Send`, `Sync` and `Unpin` are not applied to the struct. (`*mut u8` is not `Send` or `Sync`, `PhantomPinned` is not `Unpin`)

But because our `Foo` and `Bar` types are different, we'll get type safety between the two of them, so we cannot accidentally pass a pointer to `Foo` to `bar()`.

Notice that it is a really bad idea to use an empty enum as FFI type. The compiler relies on empty enums being uninhabited, so handling values of type `&Empty` is a huge footgun and can lead to buggy program behavior (by triggering undefined behavior).

NOTE: The simplest way would use "extern types". But it's currently (as of June 2021) unstable and has some unresolved questions, see the RFC page and the tracking issue for more details.

Beneath std

This section documents (or will document) features that are provided by the standard library and that `#![no_std]` developers have to deal with (i.e. provide) to build `#![no_std]` binary crates. A (likely incomplete) list of such features is shown below:

- `#[lang = "eh_personality"]`
- `#[lang = "start"]`
- `#[lang = "termination"]`
- `#[panic_implementation]`

#[panic_handler]

`#[panic_handler]` is used to define the behavior of `panic!` in `#![no_std]` applications. The `#[panic_handler]` attribute must be applied to a function with signature `fn(&PanicInfo) -> !` and such function must appear *once* in the dependency graph of a binary / dylib / cdylib crate. The API of `PanicInfo` can be found in the API docs.

Given that `#![no_std]` applications have no *standard* output and that some `#![no_std]` applications, e.g. embedded applications, need different panicking behaviors for development and for release it can be helpful to have panic crates, crate that only contain a `#[panic_handler]` . This way applications can easily swap the panicking behavior by simply linking to a different panic crate.

Below is shown an example where an application has a different panicking behavior depending on whether is compiled using the dev profile (`cargo build`) or using the release profile (`cargo build --release`).

`panic-semihosting` crate -- log panic messages to the host stderr using semihosting:

```
#![no_std]

use core::fmt::{Write, self};
use core::panic::PanicInfo;

struct HStderr {
    // ..
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    let mut host_stderr = HStderr::new();

    // logs "panicked at '$reason', src/main.rs:27:4" to the host stderr
    writeln!(host_stderr, "{}", info).ok();

    loop {}
}
```

`panic-halt` crate -- halt the thread on panic; messages are discarded:

```
#![no_std]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

`app` crate:

```
#![no_std]

// dev profile
#[cfg(debug_assertions)]
extern crate panic_semihosting;

// release profile
#[cfg(not(debug_assertions))]
extern crate panic_halt;

fn main() {
    // ..
}
```