Learn Rust With Entirely Too Many Linked Lists

Got any issues or want to check out all the final code at once? Everything's on Github!

NOTE: The current edition of this book is written against Rust 2018, which was first released with rustc 1.31 (Dec 8, 2018). If your rust toolchain is new enough, the Cargo.toml file that `cargo new` creates should contain the line `edition = "2018"` (or if you're reading this in the far future, perhaps some even larger number!). Using an older toolchain is possible, but unlocks a secret **hardmode**, where you get extra compiler errors that go completely unmentioned in the text of this book. Wow, sounds like fun!

I fairly frequently get asked how to implement a linked list in Rust. The answer honestly depends on what your requirements are, and it's obviously not super easy to answer the question on the spot. As such I've decided to write this book to comprehensively answer the question once and for all.

In this series I will teach you basic and advanced Rust programming entirely by having you implement 6 linked lists. In doing so, you should learn:

- The following pointer types: `&`, `&mut`, `Box`, `Rc`, `Arc`, `*const`, `*mut`, `NonNull` (?)
- Ownership, borrowing, inherited mutability, interior mutability, Copy
- All The Keywords: struct, enum, fn, pub, impl, use, ...
- Pattern matching, generics, destructors
- Testing, installing new toolchains, using `miri`
- Unsafe Rust: raw pointers, aliasing, stacked borrows, UnsafeCell, variance

Yes, linked lists are so truly awful that you deal with all of these concepts in making them real.

Everything's in the sidebar (may be collapsed on mobile), but for quick reference, here's what we're going to be making:

1. A Bad Singly-Linked Stack
2. An Ok Singly-Linked Stack
3. A Persistent Singly-Linked Stack
4. A Bad But Safe Doubly-Linked Deque
5. An Unsafe Singly-Linked Queue
6. TODO: An Ok Unsafe Doubly-Linked Deque
7. Bonus: A Bunch of Silly Lists

Just so we're all the same page, I'll be writing out all the commands that I feed into my terminal. I'll also be using Rust's standard package manager, Cargo, to develop the project. Cargo isn't necessary to write a Rust program, but it's *so much* better than using rustc directly. If you just want to futz around you can also run some simple programs in the browser via play.rust-lang.org.

In later sections, we'll be using "rustup" to install extra Rust tooling. I strongly recommend installing all of your Rust toolchains using rustup.

Let's get started and make our project:

```
> cargo new --lib lists
> cd lists
```

We'll put each list in a separate file so that we don't lose any of our work.

It should be noted that the *authentic* Rust learning experience involves writing code, having the compiler scream at you, and trying to figure out what the heck that means. I will be carefully ensuring that this occurs as frequently as possible. Learning to read and understand Rust's generally excellent compiler errors and documentation is *incredibly* important to being a productive Rust programmer.

Although actually that's a lie. In writing this I encountered *way* more compiler errors than I show. In particular, in the later chapters I won't be showing a lot of the random "I typed (copy-pasted) bad" errors that you expect to encounter in every language. This is a *guided tour* of having the compiler scream at us.

We're going to be going pretty slow, and I'm honestly not going to be very serious pretty much the entire time. I think programming should be fun, dang it! If you're the type of person who wants maximally information-dense, serious, and formal content, this book is not for you. Nothing I will ever make is for you. You are wrong.

An Obligatory Public Service Announcement

Just so we're totally 100% clear: I hate linked lists. With a passion. Linked lists are terrible data structures. Now of course there's several great use cases for a linked list:

- You want to do *a lot* of splitting or merging of big lists. *A lot*.
- You're doing some awesome lock-free concurrent thing.
- You're writing a kernel/embedded thing and want to use an intrusive list.
- You're using a pure functional language and the limited semantics and absence of mutation makes linked lists easier to work with.
- ... and more!

But all of these cases are *super rare* for anyone writing a Rust program. 99% of the time you should just use a Vec (array stack), and 99% of the other 1% of the time you should be using a VecDeque (array deque). These are blatantly superior data structures for most workloads due to less frequent allocation, lower memory overhead, true random access, and cache locality.

Linked lists are as *niche* and *vague* of a data structure as a trie. Few would balk at me claiming a trie is a niche structure that your average programmer could happily never learn in an entire productive career -- and yet linked lists have some bizarre celebrity status. We teach every undergrad how to write a linked list. It's the only niche collection I couldn't kill from std::collections. It's *the* list in C++!

We should all as a community say *no* to linked lists as a "standard" data structure. It's a fine data structure with several great use cases, but those use cases are *exceptional*, not common.

Several people apparently read the first paragraph of this PSA and then stop reading. Like, literally they'll try to rebut my argument by listing one of the things in my list of *great use cases*. The thing right after the first paragraph!

Just so I can link directly to a detailed argument, here are several attempts at counter-arguments I have seen, and my response to them. Feel free to skip to the first chapter if you just want to learn some Rust!

Performance doesn't always matter

Yes! Maybe your application is I/O-bound or the code in question is in some cold case that just doesn't matter. But this isn't even an argument for using a linked list. This is an argument for using *whatever at all*. Why settle for a linked list? Use a linked hash map!

If performance doesn't matter, then it's *surely* fine to apply the natural default of an array.

They have O(1) split-append-insert-remove if you have a pointer there

Yep! Although as Bjarne Stroustrup notes *this doesn't actually matter* if the time it takes to get that pointer completely dwarfs the time it would take to just copy over all the elements in an array (which is really quite fast).

Unless you have a workload that is heavily dominated by splitting and merging costs, the penalty *every other* operation takes due to caching effects and code complexity will eliminate any theoretical gains.

But yes, if you're profiling your application to spend a lot of time in splitting and merging, you may have gains in a linked list.

I can't afford amortization

You've already entered a pretty niche space -- most can afford amortization. Still, arrays are amortized *in the worst case*. Just because you're using an array, doesn't mean you have amortized costs. If you can predict how many elements you're going to store (or even have an upper-bound), you can pre-reserve all the space you need. In my experience it's *very* common to be able to predict how many elements you'll need. In Rust in particular, all iterators provide a `size_hint` for exactly this case.

Then `push` and `pop` will be truly O(1) operations. And they're going to be *considerably* faster than `push` and `pop` on linked list. You do a pointer offset, write the bytes, and increment an integer. No need to go to any kind of allocator.

How's that for low latency?

But yes, if you can't predict your load, there are worst-case latency savings to be had!

Linked lists waste less space

Well, this is complicated. A "standard" array resizing strategy is to grow or shrink so that at most half the array is empty. This is indeed a lot of wasted space. Especially in Rust, we don't automatically shrink collections (it's a waste if you're just going to fill it back up again), so the wastage can approach infinity!

But this is a worst-case scenario. In the best-case, an array stack only has three pointers of overhead for the entire array. Basically no overhead.

Linked lists on the other hand unconditionally waste space per element. A singly-linked list wastes one pointer while a doubly-linked list wastes two. Unlike an array, the relative wasteage is proportional to the size of the element. If you have *huge* elements this approaches 0 waste. If you have tiny elements (say, bytes), then this can be as much as 16x memory overhead (8x on 32-bit)!

Actually, it's more like 23x (11x on 32-bit) because padding will be added to the byte to align the whole node's size to a pointer.

This is also assuming the best-case for your allocator: that allocating and deallocating nodes is being done densely and you're not losing memory to fragmentation.

But yes, if you have huge elements, can't predict your load, and have a decent allocator, there are memory savings to be had!

I use linked lists all the time in <functional language>

Great! Linked lists are super elegant to use in functional languages because you can manipulate them without any mutation, can describe them recursively, and also work with infinite lists due to the magic of laziness.

Specifically, linked lists are nice because they represent an iteration without the need for any mutable state. The next step is just visiting the next sublist.

Rust mostly does this kind of thing with iterators. They can be infinite and you can map, filter, reverse, and concatenate them just like a functional list, and it will all be done just as lazily!

Rust also lets you easily talk about sub-arrays with *slices*. Your usual head/tail split in a functional language is just `slice.split_at_mut(1)`. For a long time, Rust had an experimental system for pattern matching on slices which was super cool, but the feature was simplified when it was stabilized. Still, basic slice patterns are neat! And of course, slices can be turned into iterators!

But yes, if you're limited to immutable semantics, linked lists can be very nice.

Note that I'm not saying that functional programming is necessarily weak or bad. However it *is* fundamentally semantically limited: you're largely only allowed to talk about how things *are*, and not how they should be *done*. This is actually a *feature*, because it enables the compiler to do tons of exotic transformations and potentially figure out the *best* way to do things without you having to worry about it. However this comes at the cost of being *able* to worry about it. There are usually escape hatches, but at some limit you're just writing procedural code again.

Even in functional languages, you should endeavour to use the appropriate data structure for the job when you actually need a data structure. Yes, singly-linked lists are your primary tool for control flow, but they're a really poor way to actually store a bunch of data and query it.

Linked lists are great for building concurrent data structures!

Yes! Although writing a concurrent data structure is really a whole different beast, and isn't something that should be taken lightly. Certainly not something many people will even *consider* doing. Once one's been written, you're also not really choosing to use a linked list. You're choosing to use an MPSC queue or whatever. The implementation strategy is pretty far removed in this case!

But yes, linked lists are the defacto heroes of the dark world of lock-free concurrency.

Mumble mumble kernel embedded something something intrusive.

It's niche. You're talking about a situation where you're not even using your language's *runtime*. Is that not a red flag that you're doing something strange?

It's also wildly unsafe.

But sure. Build your awesome zero-allocation lists on the stack.

Iterators don't get invalidated by unrelated insertions/removals

That's a delicate dance you're playing. Especially if you don't have a garbage collector. I might argue that your control flow and ownership patterns are probably a bit too tangled, depending on the details.

But yes, you can do some really cool crazy stuff with cursors.

They're simple and great for teaching!

Well, yeah. You're reading a book dedicated to that premise. Well, singly-linked lists are pretty simple. Doubly-linked lists can get kinda gnarly, as we'll see.

Take a Breath

Ok. That's out of the way. Let's write a bajillion linked lists.

On to the first chapter!

A Bad Singly-Linked Stack

This one's gonna be *by far* the longest, as we need to introduce basically all of Rust, and are gonna build up some things "the hard way" to better understand the language.

We'll put our first list in `src/first.rs`. We need to tell Rust that `first.rs` is something that our lib uses. All that requires is that we put this at the top of `src/lib.rs` (which Cargo made for us):

```
// in lib.rs
pub mod first;
```

Basic Data Layout

Alright, so what's a linked list? Well basically, it's a bunch of pieces of data on the heap (hush, kernel people!) that point to each other in sequence. Linked lists are something procedural programmers shouldn't touch with a 10-foot pole, and what functional programmers use for everything. It seems

fair, then, that we should ask functional programmers for the definition of a linked list. They will probably give you something like the following definition:

```
List a = Empty | Elem a (List a)
```

Which reads approximately as "A List is either Empty or an Element followed by a List". This is a recursive definition expressed as a *sum type*, which is a fancy name for "a type that can have different values which may be different types". Rust calls sum types `enum`s! If you're coming from a C-like language, this is exactly the enum you know and love, but in overdrive. So let's transcribe this functional definition into Rust!

For now we'll avoid generics to keep things simple. We'll only support storing signed 32-bit integers:

```
// in first.rs

// pub says we want people outside this module to be able to use List
pub enum List {
    Empty,
    Elem(i32, List),
}
```

phew, I'm swamped. Let's just go ahead and compile that:

```
> cargo build

error[E0072]: recursive type `first::List` has infinite size
 --> src/first.rs:4:1
  |
4 | pub enum List {
  | ^^^^^^^^^^^^^ recursive type has infinite size
5 |     Empty,
6 |     Elem(i32, List),
  |               ---- recursive without indirection
  |
  = help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to make
`first::List` representable
```

Well. I don't know about you, but I certainly feel betrayed by the functional programming community.

If we actually check out the error message (after we get over the whole betrayal thing), we can see that rustc is actually telling us exactly how to solve this problem:

---

insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to make `first::List` representable

---

Alright, `box`. What's that? Let's google `rust box`...

---

std::boxed::Box - Rust

Lesse here...

`pub struct Box<T>(_);`

A pointer type for heap allocation. See the module-level documentation for more.

---

clicks link

`Box<T>`, casually referred to as a 'box', provides the simplest form of heap allocation in Rust. Boxes provide ownership for this allocation, and drop their contents when they go out of scope.

Examples

Creating a box:

```
let x = Box::new(5);
```

Creating a recursive data structure:

```
#[derive(Debug)]
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}
```

```
fn main() {
    let list: List<i32> = List::Cons(1, Box::new(List::Cons(2, Box::new(List::Nil))));
    println!("{:?}", list);
}
```

This will print `Cons(1, Box(Cons(2, Box(Nil))))`.

Recursive structures must be boxed, because if the definition of Cons looked like this:

```
Cons(T, List<T>),
```

It wouldn't work. This is because the size of a List depends on how many elements are in the list, and so we don't know how much memory to allocate for a Cons. By introducing a Box, which has a defined size, we know how big Cons needs to be.

Wow, uh. That is perhaps the most relevant and helpful documentation I have ever seen. Literally the first thing in the documentation is *exactly what we're trying to write, why it didn't work, and how to fix it*.

Dang, docs rule.

Ok, let's do that:

```
pub enum List {
    Empty,
    Elem(i32, Box<List>),
}
```

```
> cargo build

   Finished dev [unoptimized + debuginfo] target(s) in 0.22s
```

Hey it built!

...but this is actually a really foolish definition of a List, for a few reasons.

Consider a list with two elements:

```
[] = Stack
() = Heap

[Elem A, ptr] -> (Elem B, ptr) -> (Empty, *junk*)
```

There are two key issues:

- We're allocating a node that just says "I'm not actually a Node"
- One of our nodes isn't heap-allocated at all.

On the surface, these two seem to cancel each-other out. We heap-allocate an extra node, but one of our nodes doesn't need to be heap-allocated at all. However, consider the following potential layout for our list:

```
[ptr] -> (Elem A, ptr) -> (Elem B, *null*)
```

In this layout we now unconditionally heap allocate our nodes. The key difference is the absence of the *junk* from our first layout. What is this junk? To understand that, we'll need to look at how an enum is laid out in memory.

In general, if we have an enum like:

```
enum Foo {
    D1(T1),
    D2(T2),
    ...
    Dn(Tn),
}
```

A Foo will need to store some integer to indicate which *variant* of the enum it represents (`D1` , `D2` , .. `Dn`). This is the *tag* of the enum. It will also need enough space to store the *largest* of `T1` , `T2` , .. `Tn` (plus some extra space to satisfy alignment requirements).

The big takeaway here is that even though `Empty` is a single bit of information, it necessarily consumes enough space for a pointer and an element, because it has to be ready to become an `Elem` at any time. Therefore the first layout heap allocates an extra element that's just full of junk, consuming a bit more space than the second layout.

One of our nodes not being allocated at all is also, perhaps surprisingly, *worse* than always allocating it. This is because it gives us a *non-uniform* node layout. This doesn't have much of an appreciable effect on pushing and popping nodes, but it does have an effect on splitting and merging lists.

Consider splitting a list in both layouts:

```
layout 1:

[Elem A, ptr] -> (Elem B, ptr) -> (Elem C, ptr) -> (Empty *junk*)

split off C:

[Elem A, ptr] -> (Elem B, ptr) -> (Empty *junk*)
[Elem C, ptr] -> (Empty *junk*)
```

```
layout 2:

[ptr] -> (Elem A, ptr) -> (Elem B, ptr) -> (Elem C, *null*)

split off C:

[ptr] -> (Elem A, ptr) -> (Elem B, *null*)
[ptr] -> (Elem C, *null*)
```

Layout 2's split involves just copying B's pointer to the stack and nulling the old value out. Layout 1 ultimately does the same thing, but also has to copy C from the heap to the stack. Merging is the same process in reverse.

One of the few nice things about a linked list is that you can construct the element in the node itself, and then freely shuffle it around lists without ever moving it. You just fiddle with pointers and stuff gets "moved". Layout 1 trashes this property.

Alright, I'm reasonably convinced Layout 1 is bad. How do we rewrite our List? Well, we could do something like:

```
pub enum List {
    Empty,
    ElemThenEmpty(i32),
    ElemThenNotEmpty(i32, Box<List>),
}
```

Hopefully this seems like an even worse idea to you. Most notably, this really complicates our logic, because there is now a completely invalid state: `ElemThenNotEmpty(0, Box(Empty))`. It also *still* suffers from non-uniformly allocating our elements.

However it does have *one* interesting property: it totally avoids allocating the Empty case, reducing the total number of heap allocations by 1. Unfortunately, in doing so it manages to waste *even more space*! This is because the previous layout took advantage of the *null pointer optimization*.

We previously saw that every enum has to store a *tag* to specify which variant of the enum its bits represent. However, if we have a special kind of enum:

```
enum Foo {
    A,
    B(ContainsANonNullPtr),
}
```

the null pointer optimization kicks in, which *eliminates the space needed for the tag*. If the variant is A, the whole enum is set to all `0`'s. Otherwise, the variant is B. This works because B can never be all `0`'s, since it contains a non-zero pointer. Slick!

Can you think of other enums and types that could do this kind of optimization? There's actually a lot! This is why Rust leaves enum layout totally unspecified. There are a few more complicated enum layout optimizations that Rust will do for us, but the null pointer one is definitely the most important! It means `&`, `&mut`, `Box`, `Rc`, `Arc`, `Vec`, and several other important types in Rust have no overhead when put in an `Option`! (We'll get to most of these in due time.)

So how do we avoid the extra junk, uniformly allocate, *and* get that sweet null-pointer optimization? We need to better separate out the idea of having an element from allocating another list. To do this, we have to think a little more C-like: structs!

While enums let us declare a type that can contain *one* of several values, structs let us declare a type that contains *many* values at once. Let's break our List into two types: A List, and a Node.

As before, a List is either Empty or has an element followed by another List. By representing the "has an element followed by another List" case by an entirely separate type, we can hoist the Box to be in a more optimal position:

```
struct Node {
    elem: i32,
    next: List,
}

pub enum List {
    Empty,
    More(Box<Node>),
}
```

Let's check our priorities:

- Tail of a list never allocates extra junk: check!
- `enum` is in delicious null-pointer-optimized form: check!
- All elements are uniformly allocated: check!

Alright! We actually just constructed exactly the layout that we used to demonstrate that our first layout (as suggested by the official Rust documentation) was problematic.

```
> cargo build

warning: private type `first::Node` in public interface (error E0446)
 --> src/first.rs:8:10
  |
8 |     More(Box<Node>),
  |          ^^^^^^^^^
  |
  = note: #[warn(private_in_public)] on by default
  = warning: this was previously accepted by the compiler but
    is being phased out; it will become a hard error in a future release!
```

:(

Rust is mad at us again. We marked the `List` as public (because we want people to be able to use it), but not the `Node`. The problem is that the internals of an `enum` are totally public, and we're not allowed to publicly talk about private types. We could make all of `Node` totally public, but generally in Rust we favour keeping implementation details private. Let's make `List` a struct, so that we can hide the implementation details:

```
pub struct List {
    head: Link,
}

enum Link {
    Empty,
    More(Box<Node>),
}

struct Node {
    elem: i32,
    next: Link,
}
```

Because `List` is a struct with a single field, its size is the same as that field. Yay zero-cost abstractions!

```
> cargo build

warning: field is never used: `head`
 --> src/first.rs:2:5
  |
2 |     head: Link,
  |     ^^^^^^^^^^
  |
  = note: #[warn(dead_code)] on by default

warning: variant is never constructed: `Empty`
 --> src/first.rs:6:5
  |
6 |     Empty,
  |     ^^^^^

warning: variant is never constructed: `More`
 --> src/first.rs:7:5
  |
7 |     More(Box<Node>),
  |     ^^^^^^^^^^^^^^^

warning: field is never used: `elem`
  --> src/first.rs:11:5
   |
11 |     elem: i32,
   |     ^^^^^^^^^

warning: field is never used: `next`
  --> src/first.rs:12:5
   |
12 |     next: Link,
   |     ^^^^^^^^^^
```

Alright, that compiled! Rust is pretty mad, because as far as it can tell, everything we've written is totally useless: we never use `head`, and no one who uses our library can either since it's private. Transitively, that means Link and Node are useless too. So let's solve that! Let's implement some code for our List!

New

To associate actual code with a type, we use `impl` blocks:

```
impl List {
    // TODO, make code happen
}
```

Now we just need to figure out how to actually write code. In Rust we declare a function like so:

```
fn foo(arg1: Type1, arg2: Type2) -> ReturnType {
    // body
}
```

The first thing we want is a way to *construct* a list. Since we hide the implementation details, we need to provide that as a function. The usual way to do that in Rust is to provide a static method, which is just a normal function inside an `impl`:

```
impl List {
    pub fn new() -> Self {
        List { head: Link::Empty }
    }
}
```

A few notes on this:

- Self is an alias for "that type I wrote at the top next to `impl`". Great for not repeating yourself!
- We create an instance of a struct in much the same way we declare it, except instead of providing the types of its fields, we initialize them with values.
- We refer to variants of an enum using `::`, which is the namespacing operator.
- The last expression of a function is implicitly returned. This makes simple functions a little neater. You can still use `return` to return early like other C-like languages.

Ownership 101

Now that we can construct a list, it'd be nice to be able to *do* something with it. We do that with "normal" (non-static) methods. Methods are a special case of function in Rust because of the `self` argument, which doesn't have a declared type:

```
fn foo(self, arg2: Type2) -> ReturnType {
    // body
}
```

There are 3 primary forms that self can take: `self`, `&mut self`, and `&self`. These 3 forms represent the three primary forms of ownership in Rust:

- `self` - Value
- `&mut self` - mutable reference
- `&self` - shared reference

A value represents *true* ownership. You can do whatever you want with a value: move it, destroy it, mutate it, or loan it out via a reference. When you pass something by value, it's *moved* to the new location. The new location now owns the value, and the old location can no longer access it. For this reason most methods don't want `self` -- it would be pretty lame if trying to work with a list made it go away!

A mutable reference represents temporary *exclusive access* to a value that you don't own. You're allowed to do absolutely anything you want to a value you have a mutable reference to as long you leave it in a valid state when you're done (it would be rude to the owner otherwise!). This means you can actually completely overwrite the value. A really useful special case of this is *swapping* a value out for another, which we'll be using a lot. The only thing you can't do with an `&mut` is move the value out with no replacement. `&mut self` is great for methods that want to mutate `self`.

A shared reference represents temporary *shared access* to a value that you don't own. Because you have shared access, you're generally not allowed to mutate anything. Think of `&` as putting the value out on display in a museum. `&` is great for methods that only want to observe `self`.

Later we'll see that the rule about mutation can be bypassed in certain cases. This is why shared references aren't called *immutable* references. Really, mutable references could be called *unique* references, but we've found that relating ownership to mutability gives the right intuition 99% of the time.

Push

So let's write pushing a value onto a list. `push` *mutates* the list, so we'll want to take `&mut self`. We also need to take an i32 to push:

```
impl List {
    pub fn push(&mut self, elem: i32) {
        // TODO
    }
}
```

First things first, we need to make a node to store our element in:

```
    pub fn push(&mut self, elem: i32) {
        let new_node = Node {
            elem: elem,
            next: ?????
        };
    }
```

What goes `next`? Well, the entire old list! Can we... just do that?

```
impl List {
    pub fn push(&mut self, elem: i32) {
        let new_node = Node {
            elem: elem,
            next: self.head,
        };
    }
}
```

```
> cargo build
error[E0507]: cannot move out of borrowed content
  --> src/first.rs:19:19
   |
19 |             next: self.head,
   |                   ^^^^^^^^^ cannot move out of borrowed content
```

Nooooope. Rust is telling us the right thing, but it's certainly not obvious what exactly it means, or what to do about it:

---

cannot move out of borrowed content

We're trying to move the `self.head` field out to `next`, but Rust doesn't want us doing that. This would leave `self` only partially initialized when we end the borrow and "give it back" to its rightful owner. As we said before, that's the *one* thing you can't do with an `&mut`: It would be super rude, and Rust is very polite (it would also be incredibly dangerous, but surely *that* isn't why it cares).

What if we put something back? Namely, the node that we're creating:

```
pub fn push(&mut self, elem: i32) {
    let new_node = Box::new(Node {
        elem: elem,
        next: self.head,
    });

    self.head = Link::More(new_node);
}
```

```
> cargo build
error[E0507]: cannot move out of borrowed content
  --> src/first.rs:19:19
   |
19 |             next: self.head,
   |                   ^^^^^^^^^ cannot move out of borrowed content
```

No dice. In principle, this is something Rust could actually accept, but it won't (for various reasons -- the most serious being exception safety). We need some way to get the head without Rust noticing that it's gone. For advice, we turn to infamous Rust Hacker Indiana Jones:

Ah yes, Indy suggests the `mem::replace` maneuver. This incredibly useful function lets us steal a value out of a borrow by *replacing* it with another value. Let's just pull in `std::mem` at the top of the file, so that `mem` is in local scope:

```
use std::mem;
```

and use it appropriately:

```
pub fn push(&mut self, elem: i32) {
    let new_node = Box::new(Node {
        elem: elem,
        next: mem::replace(&mut self.head, Link::Empty),
    });

    self.head = Link::More(new_node);
}
```

Here we `replace` self.head temporarily with Link::Empty before replacing it with the new head of the list. I'm not gonna lie: this is a pretty unfortunate thing to have to do. Sadly, we must (for now).

But hey, that's `push` all done! Probably. We should probably test it, honestly. Right now the easiest way to do that is probably to write `pop`, and make sure that it produces the right results.

Pop

Like `push`, `pop` wants to mutate the list. Unlike `push`, we actually want to return something. But `pop` also has to deal with a tricky corner case: what if the list is empty? To represent this case, we use the trusty `Option` type:

```
pub fn pop(&mut self) -> Option<i32> {
    // TODO
}
```

`Option<T>` is an enum that represents a value that may exist. It can either be `Some(T)` or `None`. We could make our own enum for this like we did for Link, but we want our users to be able to understand what the heck our return type is, and Option is so ubiquitous that *everyone* knows it. In fact, it's so fundamental that it's implicitly imported into scope in every file, as well as its variants `Some` and `None` (so we don't have to say `Option::None`).

The pointy bits on `Option<T>` indicate that Option is actually *generic* over T. That means that you can make an Option for *any* type!

So uh, we have this `Link` thing, how do we figure out if it's Empty or has More? Pattern matching with `match`!

```
pub fn pop(&mut self) -> Option<i32> {
    match self.head {
        Link::Empty => {
            // TODO
        }
        Link::More(node) => {
            // TODO
        }
    };
}
```

```
> cargo build

error[E0308]: mismatched types
  --> src/first.rs:27:30
   |
27 |     pub fn pop(&mut self) -> Option<i32> {
   |            ---              ^^^^^^^^^^^ expected enum `std::option::Option`,
found ()
   |            |
   |            this function's body doesn't return
   |
   = note: expected type `std::option::Option<i32>`
              found type `()`
```

Whoops, `pop` has to return a value, and we're not doing that yet. We *could* return `None`, but in this case it's probably a better idea to return `unimplemented!()`, to indicate that we aren't done implementing the function. `unimplemented!()` is a macro (`!` indicates a macro) that panics the program when we get to it (~crashes it in a controlled manner).

```
pub fn pop(&mut self) -> Option<i32> {
    match self.head {
        Link::Empty => {
            // TODO
        }
        Link::More(node) => {
            // TODO
        }
    };
    unimplemented!()
}
```

Unconditional panics are an example of a diverging function. Diverging functions never return to the caller, so they may be used in places where a value of any type is expected. Here, `unimplemented!()` is being used in place of a value of type `Option<T>`.

Note also that we don't need to write `return` in our program. The last expression (basically line) in a function is implicitly its return value. This lets us express really simple things a bit more concisely. You can always explicitly return early with `return` like any other C-like language.

```
> cargo build

error[E0507]: cannot move out of borrowed content
  --> src/first.rs:28:15
   |
28 |         match self.head {
   |               ^^^^^^^^^
   |               |
   |               cannot move out of borrowed content
   |               help: consider borrowing here: `&self.head`
...
32 |             Link::More(node) => {
   |                        ---- data moved here
   |
note: move occurs because `node` has type `std::boxed::Box<first::Node>`, which does
not implement the `Copy` trait
  --> src/first.rs:32:24
   |
32 |             Link::More(node) => {
   |                        ^^^^
```

Come on Rust, get off our back! As always, Rust is hella mad at us. Thankfully, this time it's also giving us the full scoop! By default, a pattern match will try to move its contents into the new branch, but we can't do this because we don't own self by-value here.

```
help: consider borrowing here: `&self.head`
```

Rust says we should add a reference to our `match` to fix that. 🤷 Let's try it:

```
pub fn pop(&mut self) -> Option<i32> {
    match &self.head {
        Link::Empty => {
            // TODO
        }
        Link::More(node) => {
            // TODO
        }
    };
    unimplemented!()
}
```

```
> cargo build

warning: unused variable: `node`
  --> src/first.rs:32:24
   |
32 |             Link::More(node) => {
   |                        ^^^^ help: consider prefixing with an underscore: `_node`
   |
   = note: #[warn(unused_variables)] on by default

warning: field is never used: `elem`
  --> src/first.rs:13:5
   |
13 |     elem: i32,
   |     ^^^^^^^^^
   |
   = note: #[warn(dead_code)] on by default

warning: field is never used: `next`
  --> src/first.rs:14:5
   |
14 |     next: Link,
   |     ^^^^^^^^^^
```

Hooray, compiling again! Now let's figure out that logic. We want to make an Option, so let's make a variable for that. In the Empty case we need to return None. In the More case we need to return `Some(i32)`, and change the head of the list. So, let's try to do basically that?

```
pub fn pop(&mut self) -> Option<i32> {
    let result;
    match &self.head {
        Link::Empty => {
            result = None;
        }
        Link::More(node) => {
            result = Some(node.elem);
            self.head = node.next;
        }
    };
    result
}
```

```
> cargo build
   Compiling lists v0.1.0 (/Users/ABeingessner/dev/temp/lists)
error[E0507]: cannot move out of borrowed content
  --> src/first.rs:35:29
   |
35 |                 self.head = node.next;
   |                             ^^^^^^^^^ cannot move out of borrowed content
```

head

desk

We're trying to move out of `node` when all we have is a shared reference to it.

We should probably step back and think about what we're trying to do. We want to:

- Check if the list is empty.
- If it's empty, just return None
- If it's *not* empty
    - remove the head of the list

- remove its `elem`
- replace the list's head with its `next`
- return `Some(elem)`

The key insight is we want to *remove* things, which means we want to get the head of the list *by value*. We certainly can't do that through the shared reference we get through `&self.head`. We also "only" have a mutable reference to `self`, so the only way we can move stuff is to *replace it*. Looks like we're doing the Empty dance again!

Let's try that:

```
pub fn pop(&mut self) -> Option<i32> {
    let result;
    match mem::replace(&mut self.head, Link::Empty) {
        Link::Empty => {
            result = None;
        }
        Link::More(node) => {
            result = Some(node.elem);
            self.head = node.next;
        }
    };
    result
}
```

```
cargo build

   Finished dev [unoptimized + debuginfo] target(s) in 0.22s
```

O M G

It compiled without *any* warnings!!!!!

Actually I'm going to apply my own personal lint here: we made this `result` value to return, but actually we didn't need to do that at all! Just as a function evaluates to its last expression, every block also evaluates to its last expression. Normally we supress this behaviour with semi-colons, which instead makes the block evaluate to the empty tuple, `()`. This is actually the value that functions which don't declare a return value -- like `push` -- return.

So instead, we can write `pop` as:

```
pub fn pop(&mut self) -> Option<i32> {
    match mem::replace(&mut self.head, Link::Empty) {
        Link::Empty => None,
        Link::More(node) => {
            self.head = node.next;
            Some(node.elem)
        }
    }
}
```

Which is a bit more concise and idiomatic. Note that the Link::Empty branch completely lost its braces, because we only have one expression to evaluate. Just a nice shorthand for simple cases.

```
cargo build

   Finished dev [unoptimized + debuginfo] target(s) in 0.22s
```

Nice, still works!

Testing

Alright, so we've got `push` and `pop` written, now we can actually test out our stack! Rust and cargo support testing as a first-class feature, so this will be super easy. All we have to do is write a function, and annotate it with `#[test]`.

Generally, we try to keep our tests next to the code that it's testing in the Rust community. However we usually make a new namespace for the tests, to avoid conflicting with the "real" code. Just as we used `mod` to specify that `first.rs` should be included in `lib.rs`, we can use `mod` to basically create a whole new file *inline*:

```
// in first.rs

mod test {
    #[test]
    fn basics() {
        // TODO
    }
}
```

And we invoke it with `cargo test`.

```
> cargo test
   Compiling lists v0.1.0 (/Users/ABeingessner/dev/temp/lists)
    Finished dev [unoptimized + debuginfo] target(s) in 1.00s
     Running /Users/ABeingessner/dev/lists/target/debug/deps/lists-86544f1d97438f1f

running 1 test
test first::test::basics ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
; 0 filtered out
```

Yay our do-nothing test passed! Let's make it not-do-nothing. We'll do that with the `assert_eq!` macro. This isn't some special testing magic. All it does is compare the two things you give it, and panic the program if they don't match. Yep, you indicate failure to the test harness by freaking out!

```
mod test {
    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop(), None);

        // Populate list
        list.push(1);
        list.push(2);
        list.push(3);

        // Check normal removal
        assert_eq!(list.pop(), Some(3));
        assert_eq!(list.pop(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push(4);
        list.push(5);

        // Check normal removal
        assert_eq!(list.pop(), Some(5));
        assert_eq!(list.pop(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop(), Some(1));
        assert_eq!(list.pop(), None);
    }
}
```

```
> cargo test

error[E0433]: failed to resolve: use of undeclared type or module `List`
  --> src/first.rs:43:24
   |
43 |         let mut list = List::new();
   |                        ^^^^ use of undeclared type or module `List`
```

Oops! Because we made a new module, we need to pull in List explicitly to use it.

```
mod test {
    use super::List;
    // everything else the same
}
```

```
> cargo test

warning: unused import: `super::List`
  --> src/first.rs:45:9
   |
45 |     use super::List;
   |         ^^^^^^^^^^^
   |
   = note: #[warn(unused_imports)] on by default

    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running /Users/ABeingessner/dev/lists/target/debug/deps/lists-86544f1d97438f1f

running 1 test
test first::test::basics ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
; 0 filtered out
```

Yay!

What's up with that warning though...? We clearly use List in our test!

...but only when testing! To appease the compiler (and to be friendly to our consumers), we should indicate that the whole `test` module should only be compiled if we're running tests.

```
#[cfg(test)]
mod test {
    use super::List;
    // everything else the same
}
```

And that's everything for testing!

Drop

We can make a stack, push on to, pop off it, and we've even tested that it all works right!

Do we need to worry about cleaning up our list? Technically, no, not at all! Like C++, Rust uses destructors to automatically clean up resources when they're done with. A type has a destructor if it implements a *trait* called Drop. Traits are Rust's fancy term for interfaces. The Drop trait has the following interface:

```
pub trait Drop {
    fn drop(&mut self);
}
```

Basically, "when you go out of scope, I'll give you a second to clean up your affairs".

You don't actually need to implement Drop if you contain types that implement Drop, and all you'd want to do is call *their* destructors. In the case of List, all it would want to do is drop its head, which in turn would *maybe* try to drop a `Box<Node>`. All that's handled for us automatically... with one hitch.

The automatic handling is going to be bad.

Let's consider a simple list:

```
list -> A -> B -> C
```

When `list` gets dropped, it will try to drop A, which will try to drop B, which will try to drop C. Some of you might rightly be getting nervous. This is recursive code, and recursive code can blow the stack!

Some of you might be thinking "this is clearly tail recursive, and any decent language would ensure that such code wouldn't blow the stack". This is, in fact, incorrect! To see why, let's try to write what the compiler has to do, by manually implementing Drop for our List as the compiler would:

```
impl Drop for List {
    fn drop(&mut self) {
        // NOTE: you can't actually explicitly call `drop` in real Rust code;
        // we're pretending to be the compiler!
        self.head.drop(); // tail recursive - good!
    }
}

impl Drop for Link {
    fn drop(&mut self) {
        match *self {
            Link::Empty => {} // Done!
            Link::More(ref mut boxed_node) => {
                boxed_node.drop(); // tail recursive - good!
            }
        }
    }
}

impl Drop for Box<Node> {
    fn drop(&mut self) {
        self.ptr.drop(); // uh oh, not tail recursive!
        deallocate(self.ptr);
    }
}

impl Drop for Node {
    fn drop(&mut self) {
        self.next.drop();
    }
}
```

We *can't* drop the contents of the Box *after* deallocating, so there's no way to drop in a tail-recursive manner! Instead we're going to have to manually write an iterative drop for `List` that hoists nodes out of their boxes.

```
impl Drop for List {
    fn drop(&mut self) {
        let mut cur_link = mem::replace(&mut self.head, Link::Empty);
        // `while let` == "do this thing until this pattern doesn't match"
        while let Link::More(mut boxed_node) = cur_link {
            cur_link = mem::replace(&mut boxed_node.next, Link::Empty);
            // boxed_node goes out of scope and gets dropped here;
            // but its Node's `next` field has been set to Link::Empty
            // so no unbounded recursion occurs.
        }
    }
}
```

```
> cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 1 test
test first::test::basics ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

Great!

Bonus Section For Premature Optimization!

Our implementation of drop is actually *very* similar to `while let Some(_) = self.pop() { }`, which is certainly simpler. How is it different, and what performance issues could result from it once we start generalizing our list to store things other than integers?

▶ Click to expand for answer

The Final Code

Alright, 6000 words later, here's all the code we managed to actually write:

```
use std::mem;

pub struct List {
    head: Link,
}

enum Link {
    Empty,
    More(Box<Node>),
}

struct Node {
    elem: i32,
    next: Link,
}

impl List {
    pub fn new() -> Self {
        List { head: Link::Empty }
    }

    pub fn push(&mut self, elem: i32) {
        let new_node = Box::new(Node {
            elem: elem,
            next: mem::replace(&mut self.head, Link::Empty),
        });

        self.head = Link::More(new_node);
    }

    pub fn pop(&mut self) -> Option<i32> {
        match mem::replace(&mut self.head, Link::Empty) {
            Link::Empty => None,
            Link::More(node) => {
                self.head = node.next;
                Some(node.elem)
            }
        }
    }
}

impl Drop for List {
    fn drop(&mut self) {
        let mut cur_link = mem::replace(&mut self.head, Link::Empty);

        while let Link::More(mut boxed_node) = cur_link {
            cur_link = mem::replace(&mut boxed_node.next, Link::Empty);
        }
    }
}

#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop(), None);

        // Populate list
        list.push(1);
        list.push(2);
        list.push(3);
```

```
        // Check normal removal
        assert_eq!(list.pop(), Some(3));
        assert_eq!(list.pop(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push(4);
        list.push(5);

        // Check normal removal
        assert_eq!(list.pop(), Some(5));
        assert_eq!(list.pop(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop(), Some(1));
        assert_eq!(list.pop(), None);
    }
}
```

Geez. 80 lines, and half of it was tests! Well, I did say this first one was going to take a while!

An Ok Singly-Linked Stack

In the previous chapter we wrote up a minimum viable singly-linked stack. However there's a few design decisions that make it kind of sucky. Let's make it less sucky. In doing so, we will:

- Deinvent the wheel
- Make our list able to handle any element type
- Add peeking
- Make our list iterable

And in the process we'll learn about

- Advanced Option use
- Generics
- Lifetimes
- Iterators

Let's add a new file called `second.rs`:

```
// in lib.rs

pub mod first;
pub mod second;
```

And copy everything from `first.rs` into it.

Using Option

Particularly observant readers may have noticed that we actually reinvented a really bad version of Option:

```
enum Link {
    Empty,
    More(Box<Node>),
}
```

Link is just `Option<Box<Node>>`. Now, it's nice not to have to write `Option<Box<Node>>` everywhere, and unlike `pop`, we're not exposing this to the outside world, so maybe it's fine. However Option has some *really nice* methods that we've been manually implementing ourselves. Let's *not* do that, and replace everything with Options. First, we'll do it naively by just renaming everything to use Some and None:

```
use std::mem;

pub struct List {
    head: Link,
}

// yay type aliases!
type Link = Option<Box<Node>>;

struct Node {
    elem: i32,
    next: Link,
}

impl List {
    pub fn new() -> Self {
        List { head: None }
    }

    pub fn push(&mut self, elem: i32) {
        let new_node = Box::new(Node {
            elem: elem,
            next: mem::replace(&mut self.head, None),
        });

        self.head = Some(new_node);
    }

    pub fn pop(&mut self) -> Option<i32> {
        match mem::replace(&mut self.head, None) {
            None => None,
            Some(node) => {
                self.head = node.next;
                Some(node.elem)
            }
        }
    }
}

impl Drop for List {
    fn drop(&mut self) {
        let mut cur_link = mem::replace(&mut self.head, None);
        while let Some(mut boxed_node) = cur_link {
            cur_link = mem::replace(&mut boxed_node.next, None);
        }
    }
}
```

This is marginally better, but the big wins will come from Option's methods.

First, `mem::replace(&mut option, None)` is such an incredibly common idiom that Option actually just went ahead and made it a method: `take`.

```
pub struct List {
    head: Link,
}

type Link = Option<Box<Node>>;

struct Node {
    elem: i32,
    next: Link,
}

impl List {
    pub fn new() -> Self {
        List { head: None }
    }

    pub fn push(&mut self, elem: i32) {
        let new_node = Box::new(Node {
            elem: elem,
            next: self.head.take(),
        });

        self.head = Some(new_node);
    }

    pub fn pop(&mut self) -> Option<i32> {
        match self.head.take() {
            None => None,
            Some(node) => {
                self.head = node.next;
                Some(node.elem)
            }
        }
    }
}

impl Drop for List {
    fn drop(&mut self) {
        let mut cur_link = self.head.take();
        while let Some(mut boxed_node) = cur_link {
            cur_link = boxed_node.next.take();
        }
    }
}
```

Second, `match option { None => None, Some(x) => Some(y) }` is such an incredibly common idiom that it was called `map`. `map` takes a function to execute on the `x` in the `Some(x)` to produce the `y` in `Some(y)`. We could write a proper `fn` and pass it to `map`, but we'd much rather write what to do *inline*.

The way to do this is with a *closure*. Closures are anonymous functions with an extra super-power: they can refer to local variables *outside* the closure! This makes them super useful for doing all sorts of conditional logic. The only place we do a `match` is in `pop`, so let's just rewrite that:

```
pub fn pop(&mut self) -> Option<i32> {
    self.head.take().map(|node| {
        self.head = node.next;
        node.elem
    })
}
```

Ah, much better. Let's make sure we didn't break anything:

```
> cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 2 tests
test first::test::basics ... ok
test second::test::basics ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

Great! Let's move on to actually improving the code's *behaviour*.

Making it all Generic

We've already touched a bit on generics with Option and Box. However so far we've managed to avoid declaring any new type that is actually generic over arbitrary elements.

It turns out that's actually really easy. Let's make all of our types generic right now:

```
pub struct List<T> {
    head: Link<T>,
}

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

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

You just make everything a little more pointy, and suddenly your code is generic. Of course, we can't *just* do this, or else the compiler's going to be Super Mad.

```
> cargo test

error[E0107]: wrong number of type arguments: expected 1, found 0
  --> src/second.rs:14:6
   |
14 | impl List {
   |      ^^^^ expected 1 type argument

error[E0107]: wrong number of type arguments: expected 1, found 0
  --> src/second.rs:36:15
   |
36 | impl Drop for List {
   |               ^^^^ expected 1 type argument
```

The problem is pretty clear: we're talking about this `List` thing but that's not real anymore. Like Option and Box, we now always have to talk about `List<Something>`.

But what's the Something we use in all these impls? Just like List, we want our implementations to work with *all* the T's. So, just like List, let's make our `impl`s pointy:

```
impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None }
    }

    pub fn push(&mut self, elem: T) {
        let new_node = Box::new(Node {
            elem: elem,
            next: self.head.take(),
        });

        self.head = Some(new_node);
    }

    pub fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            self.head = node.next;
            node.elem
        })
    }
}

impl<T> Drop for List<T> {
    fn drop(&mut self) {
        let mut cur_link = self.head.take();
        while let Some(mut boxed_node) = cur_link {
            cur_link = boxed_node.next.take();
        }
    }
}
```

...and that's it!

```
> cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 2 tests
test first::test::basics ... ok
test second::test::basics ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

All of our code is now completely generic over arbitrary values of T. Dang, Rust is *easy*. I'd like to make a particular shout-out to `new` which didn't even change:

```
pub fn new() -> Self {
    List { head: None }
}
```

Bask in the Glory that is Self, guardian of refactoring and copy-pasta coding. Also of interest, we don't write `List<T>` when we construct an instance of list. That part's inferred for us based on the fact that we're returning it from a function that expects a `List<T>`.

Alright, let's move on to totally new *behaviour*!

Peek

One thing we didn't even bother to implement last time was peeking. Let's go ahead and do that. All we need to do is return a reference to the element in the head of the list, if it exists. Sounds easy,

let's try:

```
pub fn peek(&self) -> Option<&T> {
    self.head.map(|node| {
        &node.elem
    })
}
```

```
> cargo build

error[E0515]: cannot return reference to local data `node.elem`
  --> src/second.rs:37:13
   |
37 |             &node.elem
   |             ^^^^^^^^^^ returns a reference to data owned by the current function

error[E0507]: cannot move out of borrowed content
  --> src/second.rs:36:9
   |
36 |         self.head.map(|node| {
   |         ^^^^^^^^^ cannot move out of borrowed content
```

Sigh. What now, Rust?

Map takes `self` by value, which would move the Option out of the thing it's in. Previously this was fine because we had just `take`n it out, but now we actually want to leave it where it was. The *correct* way to handle this is with the `as_ref` method on Option, which has the following definition:

```
impl<T> Option<T> {
    pub fn as_ref(&self) -> Option<&T>;
}
```

It demotes the `Option<T>` to an Option to a reference to its internals. We could do this ourselves with an explicit match but *ugh no*. It does mean that we need to do an extra dereference to cut through the extra indirection, but thankfully the `.` operator handles that for us.

```
pub fn peek(&self) -> Option<&T> {
    self.head.as_ref().map(|node| {
        &node.elem
    })
}
```

```
cargo build

    Finished dev [unoptimized + debuginfo] target(s) in 0.32s
```

Nailed it.

We can also make a *mutable* version of this method using `as_mut`:

```
pub fn peek_mut(&mut self) -> Option<&mut T> {
    self.head.as_mut().map(|node| {
        &mut node.elem
    })
}
```

```
> cargo build
```

EZ

Don't forget to test it:

```
#[test]
fn peek() {
    let mut list = List::new();
    assert_eq!(list.peek(), None);
    assert_eq!(list.peek_mut(), None);
    list.push(1); list.push(2); list.push(3);

    assert_eq!(list.peek(), Some(&3));
    assert_eq!(list.peek_mut(), Some(&mut 3));
}
```

```
cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 3 tests
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::peek ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured
```

That's nice, but we didn't really test to see if we could mutate that `peek_mut` return value, did we? If a reference is mutable but nobody mutates it, have we really tested the mutability? Let's try using `map` on this `Option<&mut T>` to put a profound value in:

```
#[test]
fn peek() {
    let mut list = List::new();
    assert_eq!(list.peek(), None);
    assert_eq!(list.peek_mut(), None);
    list.push(1); list.push(2); list.push(3);

    assert_eq!(list.peek(), Some(&3));
    assert_eq!(list.peek_mut(), Some(&mut 3));
    list.peek_mut().map(|&mut value| {
        value = 42
    });

    assert_eq!(list.peek(), Some(&42));
    assert_eq!(list.pop(), Some(42));
}
```

```
> cargo test

error[E0384]: cannot assign twice to immutable variable `value`
   --> src/second.rs:100:13
    |
99  |         list.peek_mut().map(|&mut value| {
    |                                  -----
    |                                  |
    |                                  first assignment to `value`
    |                                  help: make this binding mutable: `mut value`
100 |             value = 42
    |             ^^^^^^^^^^ cannot assign twice to immutable variable          ^~~~~
```

The compiler is complaining that `value` is immutable, but we pretty clearly wrote `&mut value`; what gives? It turns out that writing the argument of the closure that way doesn't specify that `value` is a

mutable reference. Instead, it creates a pattern that will be matched against the argument to the closure; `|&mut value|` means "the argument is a mutable reference, but just copy the value it points to into `value`, please." If we just use `|value|`, the type of `value` will be `&mut i32` and we can actually mutate the head:

```
    #[test]
    fn peek() {
        let mut list = List::new();
        assert_eq!(list.peek(), None);
        assert_eq!(list.peek_mut(), None);
        list.push(1); list.push(2); list.push(3);

        assert_eq!(list.peek(), Some(&3));
        assert_eq!(list.peek_mut(), Some(&mut 3));

        list.peek_mut().map(|value| {
            *value = 42
        });

        assert_eq!(list.peek(), Some(&42));
        assert_eq!(list.pop(), Some(42));
    }
```

```
cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 3 tests
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::peek ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured
```

Much better!

IntoIter

Collections are iterated in Rust using the *Iterator* trait. It's a bit more complicated than `Drop`:

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

The new kid on the block here is `type Item`. This is declaring that every implementation of Iterator has an *associated type* called Item. In this case, this is the type that it can spit out when you call `next`.

The reason Iterator yields `Option<Self::Item>` is because the interface coalesces the `has_next` and `get_next` concepts. When you have the next value, you yield `Some(value)`, and when you don't you yield `None`. This makes the API generally more ergonomic and safe to use and implement, while avoiding redundant checks and logic between `has_next` and `get_next`. Nice!

Sadly, Rust has nothing like a `yield` statement (yet), so we're going to have to implement the logic ourselves. Also, there's actually 3 different kinds of iterator each collection should endeavour to implement:

- IntoIter - `T`
- IterMut - `&mut T`
- Iter - `&T`

We actually already have all the tools to implement IntoIter using List's interface: just call `pop` over and over. As such, we'll just implement IntoIter as a newtype wrapper around List:

```
// Tuple structs are an alternative form of struct,
// useful for trivial wrappers around other types.
pub struct IntoIter<T>(List<T>);

impl<T> List<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        // access fields of a tuple struct numerically
        self.0.pop()
    }
}
```

And let's write a test:

```
#[test]
fn into_iter() {
    let mut list = List::new();
    list.push(1); list.push(2); list.push(3);

    let mut iter = list.into_iter();
    assert_eq!(iter.next(), Some(3));
    assert_eq!(iter.next(), Some(2));
    assert_eq!(iter.next(), Some(1));
    assert_eq!(iter.next(), None);
}
```

```
> cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 4 tests
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::into_iter ... ok
test second::test::peek ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured
```

Nice!

Iter

Alright, let's try to implement Iter. This time we won't be able to rely on List giving us all the features we want. We'll need to roll our own. The basic logic we want is to hold a pointer to the current node we want to yield next. Because that node may not exist (the list is empty or we're otherwise done

iterating), we want that reference to be an Option. When we yield an element, we want to proceed to the current node's `next` node.

Alright, let's try that:

```
pub struct Iter<T> {
    next: Option<&Node<T>>,
}

impl<T> List<T> {
    pub fn iter(&self) -> Iter<T> {
        Iter { next: self.head.map(|node| &node) }
    }
}

impl<T> Iterator for Iter<T> {
    type Item = &T;

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

```
> cargo build

error[E0106]: missing lifetime specifier
  --> src/second.rs:72:18
   |
72 |     next: Option<&Node<T>>,
   |                  ^ expected lifetime parameter

error[E0106]: missing lifetime specifier
  --> src/second.rs:82:17
   |
82 |     type Item = &T;
   |                 ^ expected lifetime parameter
```

Oh god. Lifetimes. I've heard of these things. I hear they're a nightmare.

Let's try something new: see that `error[E0106]` thing? That's a compiler error code. We can ask rustc to explain those with, well, `--explain`:

```
> rustc --explain E0106
This error indicates that a lifetime is missing from a type. If it is an error
inside a function signature, the problem may be with failing to adhere to the
lifetime elision rules (see below).

Here are some simple examples of where you'll run into this error:

struct Foo { x: &bool }        // error
struct Foo<'a> { x: &'a bool } // correct

enum Bar { A(u8), B(&bool), }        // error
enum Bar<'a> { A(u8), B(&'a bool), } // correct

type MyStr = &str;         // error
type MyStr<'a> = &'a str; //correct
...
```

That uh... that didn't really clarify much (these docs assume we understand Rust better than we currently do). But it looks like we should add those `'a` things to our struct? Let's try that.

```
pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}
```

```
> cargo build

error[E0106]: missing lifetime specifier
  --> src/second.rs:83:22
   |
83 | impl<T> Iterator for Iter<T> {
   |                      ^^^^^^^ expected lifetime parameter

error[E0106]: missing lifetime specifier
  --> src/second.rs:84:17
   |
84 |     type Item = &T;
   |                 ^ expected lifetime parameter

error: aborting due to 2 previous errors
```

Alright I'm starting to see a pattern here... let's just add these little guys to everything we can:

```
pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

impl<'a, T> List<T> {
    pub fn iter(&'a self) -> Iter<'a, T> {
        Iter { next: self.head.map(|node| &'a node) }
    }
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;
    fn next(&'a mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.map(|node| &'a node);
            &'a node.elem
        })
    }
}
```

```
> cargo build

error: expected `:`, found `node`
  --> src/second.rs:77:47
   |
77 |         Iter { next: self.head.map(|node| &'a node) }
   |         ---- while parsing this struct        ^^^^ expected `:`

error: expected `:`, found `node`
  --> src/second.rs:85:50
   |
85 |             self.next = node.next.map(|node| &'a node);
   |                                                  ^^^^ expected `:`

error[E0063]: missing field `next` in initializer of `second::Iter<'_, _>`
  --> src/second.rs:77:9
   |
77 |         Iter { next: self.head.map(|node| &'a node) }
   |         ^^^^ missing `next`
```

Oh god. We broke Rust.

Maybe we should actually figure out what the heck this `'a` lifetime stuff even means.

Lifetimes can scare off a lot of people because they're a change to something we've known and loved since the dawn of programming. We've actually managed to dodge lifetimes so far, even though they've been tangled throughout our programs this whole time.

Lifetimes are unnecessary in garbage collected languages because the garbage collector ensures that everything magically lives as long as it needs to. Most data in Rust is *manually* managed, so that data needs another solution. C and C++ give us a clear example what happens if you just let people take pointers to random data on the stack: pervasive unmanageable unsafety. This can be roughly separated into two classes of error:

- Holding a pointer to something that went out of scope
- Holding a pointer to something that got mutated away

Lifetimes solve both of these problems, and 99% of the time, they do this in a totally transparent way.

So what's a lifetime?

Quite simply, a lifetime is the name of a region (~block/scope) of code somewhere in a program. That's it. When a reference is tagged with a lifetime, we're saying that it has to be valid for that *entire* region. Different things place requirements on how long a reference must and can be valid for. The entire lifetime system is in turn just a constraint-solving system that tries to minimize the region of every reference. If it successfully finds a set of lifetimes that satisfies all the constraints, your program compiles! Otherwise you get an error back saying that something didn't live long enough.

Within a function body you generally can't talk about lifetimes, and wouldn't want to *anyway*. The compiler has full information and can infer all the constraints to find the minimum lifetimes. However at the type and API-level, the compiler *doesn't* have all the information. It requires you to tell it about the relationship between different lifetimes so it can figure out what you're doing.

In principle, those lifetimes *could* also be left out, but then checking all the borrows would be a huge whole-program analysis that would produce mind-bogglingly non-local errors. Rust's system means all borrow checking can be done in each function body independently, and all your errors should be fairly local (or your types have incorrect signatures).

But we've written references in function signatures before, and it was fine! That's because there are certain cases that are so common that Rust will automatically pick the lifetimes for you. This is *lifetime elision*.

In particular:

```
// Only one reference in input, so the output must be derived from that input
fn foo(&A) -> &B; // sugar for:
fn foo<'a>(&'a A) -> &'a B;

// Many inputs, assume they're all independent
fn foo(&A, &B, &C); // sugar for:
fn foo<'a, 'b, 'c>(&'a A, &'b B, &'c C);

// Methods, assume all output lifetimes are derived from `self`
fn foo(&self, &B, &C) -> &D; // sugar for:
fn foo<'a, 'b, 'c>(&'a self, &'b B, &'c C) -> &'a D;
```

So what does `fn foo<'a>(&'a A) -> &'a B` *mean*? In practical terms, all it means is that the input must live at least as long as the output. So if you keep the output around for a long time, this will expand the region that the input must be valid for. Once you stop using the output, the compiler will know it's ok for the input to become invalid too.

With this system set up, Rust can ensure nothing is used after free, and nothing is mutated while outstanding references exist. It just makes sure the constraints all work out!

Alright. So. Iter.

Let's roll back to the no lifetimes state:

```
pub struct Iter<T> {
    next: Option<&Node<T>>,
}

impl<T> List<T> {
    pub fn iter(&self) -> Iter<T> {
        Iter { next: self.head.map(|node| &node) }
    }
}

impl<T> Iterator for Iter<T> {
    type Item = &T;
    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.map(|node| &node);
            &node.elem
        })
    }
}
```

We need to add lifetimes only in function and type signatures:

```
// Iter is generic over *some* lifetime, it doesn't care
pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

// No lifetime here, List doesn't have any associated lifetimes
impl<T> List<T> {
    // We declare a fresh lifetime here for the *exact* borrow that
    // creates the iter. Now &self needs to be valid as long as the
    // Iter is around.
    pub fn iter<'a>(&'a self) -> Iter<'a, T> {
        Iter { next: self.head.map(|node| &node) }
    }
}

// We *do* have a lifetime here, because Iter has one that we need to define
impl<'a, T> Iterator for Iter<'a, T> {
    // Need it here too, this is a type declaration
    type Item = &'a T;

    // None of this needs to change, handled by the above.
    // Self continues to be incredibly hype and amazing
    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.map(|node| &node);
            &node.elem
        })
    }
}
```

Alright, I think we got it this time y'all.

```
cargo build

error[E0308]: mismatched types
  --> src/second.rs:77:22
   |
77 |         Iter { next: self.head.map(|node| &node) }
   |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^ expected struct `second::Node`,
found struct `std::boxed::Box`
   |
   = note: expected type `std::option::Option<&second::Node<T>>`
              found type `std::option::Option<&std::boxed::Box<second::Node<T>>>`

error[E0308]: mismatched types
  --> src/second.rs:85:25
   |
85 |             self.next = node.next.map(|node| &node);
   |                         ^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected struct
`second::Node`, found struct `std::boxed::Box`
   |
   = note: expected type `std::option::Option<&'a second::Node<T>>`
              found type `std::option::Option<&std::boxed::Box<second::Node<T>>>`
```

(ノ °□°)ノ ⌒┻━┻

OK. SO. We fixed our lifetime errors but now we're getting some new type errors.

We want to be storing `&Node`'s, but we're getting `&Box<Node>`s. Ok, that's easy enough, we just need to dereference the Box before we take our reference:

```
impl<T> List<T> {
    pub fn iter<'a>(&'a self) -> Iter<'a, T> {
        Iter { next: self.head.map(|node| &*node) }
    }
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.map(|node| &*node);
            &node.elem
        })
    }
}
```

```
cargo build
   Compiling lists v0.1.0 (/Users/ABeingessner/dev/temp/lists)
error[E0515]: cannot return reference to local data `*node`
  --> src/second.rs:77:43
   |
77 |         Iter { next: self.head.map(|node| &*node) }
   |                                           ^^^^^^ returns a reference to data owned
by the current function

error[E0507]: cannot move out of borrowed content
  --> src/second.rs:77:22
   |
77 |         Iter { next: self.head.map(|node| &*node) }
   |                      ^^^^^^^^^ cannot move out of borrowed content

error[E0515]: cannot return reference to local data `*node`
  --> src/second.rs:85:46
   |
85 |             self.next = node.next.map(|node| &*node);
   |                                              ^^^^^^ returns a reference to data
owned by the current function

error[E0507]: cannot move out of borrowed content
  --> src/second.rs:85:25
   |
85 |             self.next = node.next.map(|node| &*node);
   |                         ^^^^^^^^^ cannot move out of borrowed content
```

(/ಥ益ಥ)/┻━┻

We forgot `as_ref`, so we're moving the box into `map`, which means it would be dropped, which means our references would be dangling:

```rust
pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

impl<T> List<T> {
    pub fn iter<'a>(&'a self) -> Iter<'a, T> {
        Iter { next: self.head.as_ref().map(|node| &*node) }
    }
}

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

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

```
cargo build
   Compiling lists v0.1.0 (/Users/ABeingessner/dev/temp/lists)
error[E0308]: mismatched types
  --> src/second.rs:77:22
   |
77 |         Iter { next: self.head.as_ref().map(|node| &*node) }
   |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected struct
`second::Node`, found struct `std::boxed::Box`
   |
   = note: expected type `std::option::Option<&second::Node<T>>`
              found type `std::option::Option<&std::boxed::Box<second::Node<T>>>`

error[E0308]: mismatched types
  --> src/second.rs:85:25
   |
85 |             self.next = node.next.as_ref().map(|node| &*node);
   |                         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected struct
`second::Node`, found struct `std::boxed::Box`
   |
   = note: expected type `std::option::Option<&'a second::Node<T>>`
              found type `std::option::Option<&std::boxed::Box<second::Node<T>>>`
```

😭

`as_ref` added another layer of indirection we need to remove:

```
pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

impl<T> List<T> {
    pub fn iter<'a>(&'a self) -> Iter<'a, T> {
        Iter { next: self.head.as_deref() }
    }
}

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

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

```
cargo build
```

The as_deref and as_deref_mut functions are stable as of Rust 1.40. Before that you would need to do `map(|node| &**node)` and `map(|node| &mut**node)`. You may be thinking "wow that `&**` thing is really janky", and you're not wrong, but like a fine wine Rust gets better over time and we no longer need to do such. Normally Rust is very good at doing this kind of conversion implicitly, through a process called *deref coercion*, where basically it can insert *'s throughout your code to make it type-check. It can do this because we have the borrow checker to ensure we never mess up pointers!

But in this case the closure in conjunction with the fact that we have an `Option<&T>` instead of `&T` is a bit too complicated for it to work out, so we need to help it by being explicit. Thankfully this is pretty rare, in my experience.

Just for completeness' sake, we *could* give it a *different* hint with the *turbofish*:

```
    self.next = node.next.as_ref().map::<&Node<T>, _>(|node| &node);
```

See, map is a generic function:

```
pub fn map<U, F>(self, f: F) -> Option<U>
```

The turbofish, `::<>`, lets us tell the compiler what we think the types of those generics should be. In this case `::<&Node<T>, _>` says "it should return a `&Node<T>`, and I don't know/care about that other type".

This in turn lets the compiler know that `&node` should have deref coercion applied to it, so we don't need to manually apply all those *'s!

But in this case I don't think it's really an improvement, this was just a thinly veiled excuse to show off deref coercion and the sometimes-useful turbofish. 😈

Let's write a test to be sure we didn't no-op it or anything:

```
#[test]
fn iter() {
    let mut list = List::new();
    list.push(1); list.push(2); list.push(3);

    let mut iter = list.iter();
    assert_eq!(iter.next(), Some(&3));
    assert_eq!(iter.next(), Some(&2));
    assert_eq!(iter.next(), Some(&1));
}
```

```
> cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 5 tests
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::peek ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured
```

Heck yeah.

Finally, it should be noted that we *can* actually apply lifetime elision here:

```
impl<T> List<T> {
    pub fn iter<'a>(&'a self) -> Iter<'a, T> {
        Iter { next: self.head.as_deref() }
    }
}
```

is equivalent to:

```
impl<T> List<T> {
    pub fn iter(&self) -> Iter<T> {
        Iter { next: self.head.as_deref() }
    }
}
```

Yay fewer lifetimes!

Or, if you're not comfortable "hiding" that a struct contains a lifetime, you can use the Rust 2018 "explicitly elided lifetime" syntax, `'_`:

```
impl<T> List<T> {
    pub fn iter(&self) -> Iter<'_, T> {
        Iter { next: self.head.as_deref() }
    }
}
```

IterMut

I'm gonna be honest, IterMut is WILD. Which in itself seems like a wild thing to say; surely it's identical to Iter!

Semantically, yes, but the nature of shared and mutable references means that Iter is "trivial" while IterMut is Legit Wizard Magic.

The key insight comes from our implementation of Iterator for Iter:

```
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    fn next(&mut self) -> Option<Self::Item> { /* stuff */ }
}
```

Which can be desugared to:

```
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    fn next<'b>(&'b mut self) -> Option<&'a T> { /* stuff */ }
}
```

The signature of `next` establishes *no* constraint between the lifetime of the input and the output! Why do we care? It means we can call `next` over and over unconditionally!

```
let mut list = List::new();
list.push(1); list.push(2); list.push(3);

let mut iter = list.iter();
let x = iter.next().unwrap();
let y = iter.next().unwrap();
let z = iter.next().unwrap();
```

Cool!

This is *definitely fine* for shared references because the whole point is that you can have tons of them at once. However mutable references *can't* coexist. The whole point is that they're exclusive.

The end result is that it's notably harder to write IterMut using safe code (and we haven't gotten into what that even means yet...). Surprisingly, IterMut can actually be implemented for many structures completely safely!

We'll start by just taking the Iter code and changing everything to be mutable:

```
pub struct IterMut<'a, T> {
    next: Option<&'a mut Node<T>>,
}

impl<T> List<T> {
    pub fn iter_mut(&self) -> IterMut<'_, T> {
        IterMut { next: self.head.as_deref_mut() }
    }
}

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

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

```
> cargo build
error[E0596]: cannot borrow `self.head` as mutable, as it is behind a `&` reference
  --> src/second.rs:95:25
   |
94 |     pub fn iter_mut(&self) -> IterMut<'_, T> {
   |                     ----- help: consider changing this to be a mutable reference:
`&mut self`
95 |         IterMut { next: self.head.as_deref_mut() }
   |                         ^^^^^^^^^ `self` is a `&` reference, so the data it refers
to cannot be borrowed as mutable

error[E0507]: cannot move out of borrowed content
   --> src/second.rs:103:9
    |
103 |         self.next.map(|node| {
    |         ^^^^^^^^^ cannot move out of borrowed content
```

Ok looks like we've got two different errors here. The first one looks really clear though, it even tells us how to fix it! You can't upgrade a shared reference to a mutable one, so `iter_mut` needs to take `&mut self`. Just a silly copy-paste error.

```
pub fn iter_mut(&mut self) -> IterMut<'_, T> {
    IterMut { next: self.head.as_deref_mut() }
}
```

What about the other one?

Oops! I actually accidentally made an error when writing the `iter` impl in the previous section, and we were just getting lucky that it worked!

We have just had our first run in with the magic of Copy. When we introduced ownership we said that when you move stuff, you can't use it anymore. For some types, this makes perfect sense. Our good friend Box manages an allocation on the heap for us, and we certainly don't want two pieces of code to think that they need to free its memory.

However for other types this is *garbage*. Integers have no ownership semantics; they're just meaningless numbers! This is why integers are marked as Copy. Copy types are known to be perfectly copyable by a bitwise copy. As such, they have a super power: when moved, the old value *is* still usable. As a consequence, you can even move a Copy type out of a reference without replacement!

All numeric primitives in Rust (i32, u64, bool, f32, char, etc...) are Copy. You can also declare any user-defined type to be Copy as well, as long as all its components are Copy.

Critically to why this code was working, shared references are also Copy! Because `&` is copy, `Option<&>` is *also* Copy. So when we did `self.next.map` it was fine because the Option was just copied. Now we can't do that, because `&mut` isn't Copy (if you copied an &mut, you'd have two &mut's to the same location in memory, which is forbidden). Instead, we should properly `take` the Option to get it.

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

```
> cargo build
```

Uh... wow. Holy shit! IterMut Just Works!

Let's test this:

```
#[test]
fn iter_mut() {
    let mut list = List::new();
    list.push(1); list.push(2); list.push(3);

    let mut iter = list.iter_mut();
    assert_eq!(iter.next(), Some(&mut 3));
    assert_eq!(iter.next(), Some(&mut 2));
    assert_eq!(iter.next(), Some(&mut 1));
}
```

```
> cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 6 tests
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::iter_mut ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::peek ... ok

test result: ok. 7 passed; 0 failed; 0 ignored; 0 measured
```

Yep. It works.

Holy shit.

What.

Ok I mean it actually *is* supposed to work, but there's usually something stupid that gets in the way! Let's be clear here:

We have just implemented a piece of code that takes a singly-linked list, and returns a mutable reference to every single element in the list at most once. And it's statically verified to do that. And it's totally safe. And we didn't have to do anything wild.

That's kind of a big deal, if you ask me. There are a couple reasons why this works:

- We `take` the `Option<&mut>` so we have exclusive access to the mutable reference. No need to worry about someone looking at it again.
- Rust understands that it's ok to shard a mutable reference into the subfields of the pointed-to struct, because there's no way to "go back up", and they're definitely disjoint.

It turns out that you can apply this basic logic to get a safe IterMut for an array or a tree as well! You can even make the iterator DoubleEnded, so that you can consume the iterator from the front *and* the back at once! Woah!

Final Code

Alright, that's it for the second list; here's the final code!

```rust
pub struct List<T> {
    head: Link<T>,
}

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

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

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None }
    }

    pub fn push(&mut self, elem: T) {
        let new_node = Box::new(Node {
            elem: elem,
            next: self.head.take(),
        });

        self.head = Some(new_node);
    }

    pub fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            self.head = node.next;
            node.elem
        })
    }

    pub fn peek(&self) -> Option<&T> {
        self.head.as_ref().map(|node| {
            &node.elem
        })
    }

    pub fn peek_mut(&mut self) -> Option<&mut T> {
        self.head.as_mut().map(|node| {
            &mut node.elem
        })
    }

    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }

    pub fn iter(&self) -> Iter<'_, T> {
        Iter { next: self.head.as_deref() }
    }

    pub fn iter_mut(&mut self) -> IterMut<'_, T> {
        IterMut { next: self.head.as_deref_mut() }
    }
}

impl<T> Drop for List<T> {
    fn drop(&mut self) {
        let mut cur_link = self.head.take();
        while let Some(mut boxed_node) = cur_link {
            cur_link = boxed_node.next.take();
        }
    }
}
```

```rust
pub struct IntoIter<T>(List<T>);

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        // access fields of a tuple struct numerically
        self.0.pop()
    }
}

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.next.as_deref();
            &node.elem
        })
    }
}

pub struct IterMut<'a, T> {
    next: Option<&'a mut Node<T>>,
}

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

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

#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop(), None);

        // Populate list
        list.push(1);
        list.push(2);
        list.push(3);

        // Check normal removal
        assert_eq!(list.pop(), Some(3));
        assert_eq!(list.pop(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push(4);
        list.push(5);

        // Check normal removal
        assert_eq!(list.pop(), Some(5));
        assert_eq!(list.pop(), Some(4));

        // Check exhaustion
```

```
        assert_eq!(list.pop(), Some(1));
        assert_eq!(list.pop(), None);
    }

    #[test]
    fn peek() {
        let mut list = List::new();
        assert_eq!(list.peek(), None);
        assert_eq!(list.peek_mut(), None);
        list.push(1); list.push(2); list.push(3);

        assert_eq!(list.peek(), Some(&3));
        assert_eq!(list.peek_mut(), Some(&mut 3));

        list.peek_mut().map(|value| {
            *value = 42
        });

        assert_eq!(list.peek(), Some(&42));
        assert_eq!(list.pop(), Some(42));
    }

    #[test]
    fn into_iter() {
        let mut list = List::new();
        list.push(1); list.push(2); list.push(3);

        let mut iter = list.into_iter();
        assert_eq!(iter.next(), Some(3));
        assert_eq!(iter.next(), Some(2));
        assert_eq!(iter.next(), Some(1));
        assert_eq!(iter.next(), None);
    }

    #[test]
    fn iter() {
        let mut list = List::new();
        list.push(1); list.push(2); list.push(3);

        let mut iter = list.iter();
        assert_eq!(iter.next(), Some(&3));
        assert_eq!(iter.next(), Some(&2));
        assert_eq!(iter.next(), Some(&1));
    }

    #[test]
    fn iter_mut() {
        let mut list = List::new();
        list.push(1); list.push(2); list.push(3);

        let mut iter = list.iter_mut();
        assert_eq!(iter.next(), Some(&mut 3));
        assert_eq!(iter.next(), Some(&mut 2));
        assert_eq!(iter.next(), Some(&mut 1));
    }
}
```

Getting beefier!

A Persistent Singly-Linked Stack

Alright, we've mastered the art of mutable singly-linked stacks.

Let's move from *single* ownership to *shared* ownership by writing a *persistent* immutable singly-linked list. This will be exactly the list that functional programmers have come to know and love. You can get the head *or* the tail and put someone's head on someone else's tail... and... that's basically it. Immutability is a hell of a drug.

In the process we'll largely just become familiar with Rc and Arc, but this will set us up for the next list which will *change the game*.

Let's add a new file called `third.rs`:

```
// in lib.rs

pub mod first;
pub mod second;
pub mod third;
```

No copy-pasta this time. This is a clean room operation.

Layout

Alright, back to the drawing board on layout.

The most important thing about a persistent list is that you can manipulate the tails of lists basically for free:

For instance, this isn't an uncommon workload to see with a persistent list:

```
list1 = A -> B -> C -> D
list2 = tail(list1) = B -> C -> D
list3 = push(list2, X) = X -> B -> C -> D
```

But at the end we want the memory to look like this:

```
list1 -> A ---+
              |
              v
list2 ------> B -> C -> D
              ^
              |
list3 -> X ---+
```

This just can't work with Boxes, because ownership of `B` is *shared*. Who should free it? If I drop list2, does it free B? With boxes we certainly would expect so!

Functional languages — and indeed almost every other language — get away with this by using *garbage collection*. With the magic of garbage collection, B will be freed only after everyone stops looking at it. Hooray!

Rust doesn't have anything like the garbage collectors these languages have. They have *tracing* GC, which will dig through all the memory that's sitting around at runtime and figure out what's garbage automatically. Instead, all Rust has today is *reference counting*. Reference counting can be thought of as a very simple GC. For many workloads, it has significantly less throughput than a tracing collector, and it completely falls over if you manage to build cycles. But hey, it's all we've got! Thankfully, for our usecase we'll never run into cycles (feel free to try to prove this to yourself — I sure won't).

So how do we do reference-counted garbage collection? `Rc`! Rc is just like Box, but we can duplicate it, and its memory will *only* be freed when *all* the Rc's derived from it are dropped. Unfortunately,

this flexibility comes at a serious cost: we can only take a shared reference to its internals. This means we can't ever really get data out of one of our lists, nor can we mutate them.

So what's our layout gonna look like? Well, previously we had:

```
pub struct List<T> {
    head: Link<T>,
}

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

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

Can we just change Box to Rc?

```
// in third.rs

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

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

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

```
cargo build

error[E0412]: cannot find type `Rc` in this scope
 --> src/third.rs:5:23
  |
5 | type Link<T> = Option<Rc<Node<T>>>;
  |                       ^^ not found in this scope
help: possible candidate is found in another module, you can import it into scope
  |
1 | use std::rc::Rc;
  |
```

Oh dang, sick burn. Unlike everything we used for our mutable lists, Rc is so lame that it's not even implicitly imported into every single Rust program. *What a loser*.

```
use std::rc::Rc;
```

```
cargo build

warning: field is never used: `head`
 --> src/third.rs:4:5
  |
4 |     head: Link<T>,
  |     ^^^^^^^^^^^^^
  |
  = note: #[warn(dead_code)] on by default

warning: field is never used: `elem`
  --> src/third.rs:10:5
   |
10 |     elem: T,
   |     ^^^^^^^

warning: field is never used: `next`
  --> src/third.rs:11:5
   |
11 |     next: Link<T>,
   |     ^^^^^^^^^^^^^
```

Seems legit. Rust continues to be *completely* trivial to write. I bet we can just find-and-replace Box with Rc and call it a day!

...

No. No we can't.

Basics

We already know a lot of the basics of Rust now, so we can do a lot of the simple stuff again.

For the constructor, we can again just copy-paste:

```
impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None }
    }
}
```

`push` and `pop` don't really make sense anymore. Instead we can provide `prepend` and `tail`, which provide approximately the same thing.

Let's start with prepending. It takes a list and an element, and returns a List. Like the mutable list case, we want to make a new node, that has the old list as its `next` value. The only novel thing is how to *get* that next value, because we're not allowed to mutate anything.

The answer to our prayers is the Clone trait. Clone is implemented by almost every type, and provides a generic way to get "another one like this one" that is logically disjoint, given only a shared reference. It's like a copy constructor in C++, but it's never implicitly invoked.

Rc in particular uses Clone as the way to increment the reference count. So rather than moving a Box to be in the sublist, we just clone the head of the old list. We don't even need to match on the head, because Option exposes a Clone implementation that does exactly the thing we want.

Alright, let's give it a shot:

```
pub fn prepend(&self, elem: T) -> List<T> {
    List { head: Some(Rc::new(Node {
        elem: elem,
        next: self.head.clone(),
    }))}
}
```

```
> cargo build

warning: field is never used: `elem`
  --> src/third.rs:10:5
   |
10 |     elem: T,
   |     ^^^^^^^
   |
   = note: #[warn(dead_code)] on by default

warning: field is never used: `next`
  --> src/third.rs:11:5
   |
11 |     next: Link<T>,
   |     ^^^^^^^^^^^^^
```

Wow, Rust is really hard-nosed about actually using fields. It can tell no consumer can ever actually observe the use of these fields! Still, we seem good so far.

`tail` is the logical inverse of this operation. It takes a list and returns the whole list with the first element removed. All that is is cloning the *second* element in the list (if it exists). Let's try this:

```
pub fn tail(&self) -> List<T> {
    List { head: self.head.as_ref().map(|node| node.next.clone()) }
}
```

```
cargo build

error[E0308]: mismatched types
  --> src/third.rs:27:22
   |
27 |         List { head: self.head.as_ref().map(|node| node.next.clone()) }
   |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected
struct `std::rc::Rc`, found enum `std::option::Option`
   |
   = note: expected type `std::option::Option<std::rc::Rc<_>>`
              found type `std::option::Option<std::option::Option<std::rc::Rc<_>>>`
```

Hrm, we messed up. `map` expects us to return a Y, but here we're returning an `Option<Y>`. Thankfully, this is another common Option pattern, and we can just use `and_then` to let us return an Option.

```
pub fn tail(&self) -> List<T> {
    List { head: self.head.as_ref().and_then(|node| node.next.clone()) }
}
```

```
> cargo build
```

Great.

Now that we have `tail`, we should probably provide `head`, which returns a reference to the first element. That's just `peek` from the mutable list:

```
pub fn head(&self) -> Option<&T> {
    self.head.as_ref().map(|node| &node.elem)
}
```

```
> cargo build
```

Nice.

That's enough functionality that we can test it:

```
#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn basics() {
        let list = List::new();
        assert_eq!(list.head(), None);

        let list = list.prepend(1).prepend(2).prepend(3);
        assert_eq!(list.head(), Some(&3));

        let list = list.tail();
        assert_eq!(list.head(), Some(&2));

        let list = list.tail();
        assert_eq!(list.head(), Some(&1));

        let list = list.tail();
        assert_eq!(list.head(), None);

        // Make sure empty tail works
        let list = list.tail();
        assert_eq!(list.head(), None);

    }
}
```

```
> cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 5 tests
test first::test::basics ... ok
test second::test::into_iter ... ok
test second::test::basics ... ok
test second::test::iter ... ok
test third::test::basics ... ok

test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured
```

Perfect!

Iter is also identical to how it was for our mutable list:

```rust
pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

impl<T> List<T> {
    pub fn iter(&self) -> Iter<'_, T> {
        Iter { next: self.head.as_deref() }
    }
}

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

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

```rust
#[test]
fn iter() {
    let list = List::new().prepend(1).prepend(2).prepend(3);

    let mut iter = list.iter();
    assert_eq!(iter.next(), Some(&3));
    assert_eq!(iter.next(), Some(&2));
    assert_eq!(iter.next(), Some(&1));
}
```

```
cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 7 tests
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::iter ... ok
test second::test::into_iter ... ok
test second::test::peek ... ok
test third::test::basics ... ok
test third::test::iter ... ok

test result: ok. 6 passed; 0 failed; 0 ignored; 0 measured
```

Who ever said dynamic typing was easier?

(chumps did)

Note that we can't implement IntoIter or IterMut for this type. We only have shared access to elements.

Drop

Like the mutable lists, we have a recursive destructor problem. Admittedly, this isn't as bad of a problem for the immutable list: if we ever hit another node that's the head of another list *somewhere*, we won't recursively drop it. However it's still a thing we should care about, and how to deal with isn't as clear. Here's how we solved it before:

```
impl<T> Drop for List<T> {
    fn drop(&mut self) {
        let mut cur_link = self.head.take();
        while let Some(mut boxed_node) = cur_link {
            cur_link = boxed_node.next.take();
        }
    }
}
```

The problem is the body of the loop:

```
cur_link = boxed_node.next.take();
```

This is mutating the Node inside the Box, but we can't do that with Rc; it only gives us shared access, because any number of other Rc's could be pointing at it.

But if we know that we're the last list that knows about this node, it *would* actually be fine to move the Node out of the Rc. Then we could also know when to stop: whenever we *can't* hoist out the Node.

And look at that, Rc has a method that does exactly this: `try_unwrap`:

```
impl<T> Drop for List<T> {
    fn drop(&mut self) {
        let mut head = self.head.take();
        while let Some(node) = head {
            if let Ok(mut node) = Rc::try_unwrap(node) {
                head = node.next.take();
            } else {
                break;
            }
        }
    }
}
```

```
cargo test
   Compiling lists v0.1.0 (/Users/ABeingessner/dev/too-many-lists/lists)
    Finished dev [unoptimized + debuginfo] target(s) in 1.10s
     Running /Users/ABeingessner/dev/too-many-lists/lists/target/debug/deps/lists-
86544f1d97438f1f

running 8 tests
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::basics ... ok
test third::test::iter ... ok

test result: ok. 8 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Great! Nice.

Arc

One reason to use an immutable linked list is to share data across threads. After all, shared mutable state is the root of all evil, and one way to solve that is to kill the *mutable* part forever.

Except our list isn't thread-safe at all. In order to be thread-safe, we need to fiddle with reference counts *atomically*. Otherwise, two threads could try to increment the reference count, *and only one would happen*. Then the list could get freed too soon!

In order to get thread safety, we have to use *Arc*. Arc is completely identical to Rc except for the fact that reference counts are modified atomically. This has a bit of overhead if you don't need it, so Rust exposes both. All we need to do to make our list is replace every reference to Rc with `std::sync::Arc`. That's it. We're thread safe. Done!

But this raises an interesting question: how do we *know* if a type is thread-safe or not? Can we accidentally mess up?

No! You can't mess up thread-safety in Rust!

The reason this is the case is because Rust models thread-safety in a first-class way with two traits: `Send` and `Sync`.

A type is *Send* if it's safe to *move* to another thread. A type is *Sync* if it's safe to *share* between multiple threads. That is, if `T` is Sync, `&T` is Send. Safe in this case means it's impossible to cause *data races*, (not to be mistaken with the more general issue of *race conditions*).

These are marker traits, which is a fancy way of saying they're traits that provide absolutely no interface. You either *are* Send, or you aren't. It's just a property *other* APIs can require. If you aren't appropriately Send, then it's statically impossible to be sent to a different thread! Sweet!

Send and Sync are also automatically derived traits based on whether you are totally composed of Send and Sync types. It's similar to how you can only implement Copy if you're only made of Copy types, but then we just go ahead and implement it automatically if you are.

Almost every type is Send and Sync. Most types are Send because they totally own their data. Most types are Sync because the only way to share data across threads is to put them behind a shared reference, which makes them immutable!

However there are special types that violate these properties: those that have *interior mutability*. So far we've only really interacted with *inherited mutability* (AKA external mutability): the mutability of a value is inherited from the mutability of its container. That is, you can't just randomly mutate some field of a non-mutable value because you feel like it.

Interior mutability types violate this: they let you mutate through a shared reference. There are two major classes of interior mutability: cells, which only work in a single-threaded context; and locks, which work in a multi-threaded context. For obvious reasons, cells are cheaper when you can use them. There's also atomics, which are primitives that act like a lock.

So what does all of this have to do with Rc and Arc? Well, they both use interior mutability for their *reference count*. Worse, this reference count is shared between every instance! Rc just uses a cell, which means it's not thread safe. Arc uses an atomic, which means it *is* thread safe. Of course, you can't magically make a type thread safe by putting it in Arc. Arc can only derive thread-safety like any other type.

I really really really don't want to get into the finer details of atomic memory models or non-derived Send implementations. Needless to say, as you get deeper into Rust's thread-safety story, stuff gets more complicated. As a high-level consumer, it all *just works* and you don't really need to think about it.

Final Code

That's all I really have to say on the immutable stack. We're getting pretty good at implementing lists now!

```rust
use std::rc::Rc;

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

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

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

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None }
    }

    pub fn prepend(&self, elem: T) -> List<T> {
        List { head: Some(Rc::new(Node {
            elem: elem,
            next: self.head.clone(),
        }))}
    }

    pub fn tail(&self) -> List<T> {
        List { head: self.head.as_ref().and_then(|node| node.next.clone()) }
    }

    pub fn head(&self) -> Option<&T> {
        self.head.as_ref().map(|node| &node.elem)
    }

    pub fn iter(&self) -> Iter<'_, T> {
        Iter { next: self.head.as_deref() }
    }
}

impl<T> Drop for List<T> {
    fn drop(&mut self) {
        let mut head = self.head.take();
        while let Some(node) = head {
            if let Ok(mut node) = Rc::try_unwrap(node) {
                head = node.next.take();
            } else {
                break;
            }
        }
    }
}

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

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

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

```
#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn basics() {
        let list = List::new();
        assert_eq!(list.head(), None);

        let list = list.prepend(1).prepend(2).prepend(3);
        assert_eq!(list.head(), Some(&3));

        let list = list.tail();
        assert_eq!(list.head(), Some(&2));

        let list = list.tail();
        assert_eq!(list.head(), Some(&1));

        let list = list.tail();
        assert_eq!(list.head(), None);

        // Make sure empty tail works
        let list = list.tail();
        assert_eq!(list.head(), None);
    }

    #[test]
    fn iter() {
        let list = List::new().prepend(1).prepend(2).prepend(3);

        let mut iter = list.iter();
        assert_eq!(iter.next(), Some(&3));
        assert_eq!(iter.next(), Some(&2));
        assert_eq!(iter.next(), Some(&1));
    }
}
```

A Bad but Safe Doubly-Linked Deque

Now that we've seen Rc and heard about interior mutability, this gives an interesting thought... maybe we *can* mutate through an Rc. And if *that's* the case, maybe we can implement a *doubly-linked* list totally safely!

In the process we'll become familiar with *interior mutability*, and probably learn the hard way that safe doesn't mean *correct*. Doubly-linked lists are hard, and I always make a mistake somewhere.

Let's add a new file called `fourth.rs`:

```
// in lib.rs

pub mod first;
pub mod second;
pub mod third;
pub mod fourth;
```

This will be another clean-room operation, though as usual we'll probably find some logic that applies verbatim again.

Disclaimer: this chapter is basically a demonstration that this is a very bad idea.

Layout

The key to our design is the `RefCell` type. The heart of RefCell is a pair of methods:

```
fn borrow(&self) -> Ref<'_, T>;
fn borrow_mut(&self) -> RefMut<'_, T>;
```

The rules for `borrow` and `borrow_mut` are exactly those of `&` and `&mut`: you can call `borrow` as many times as you want, but `borrow_mut` requires exclusivity.

Rather than enforcing this statically, RefCell enforces them at runtime. If you break the rules, RefCell will just panic and crash the program. Why does it return these Ref and RefMut things? Well, they basically behave like `Rc`s but for borrowing. They also keep the RefCell borrowed until they go out of scope. We'll get to that later.

Now with Rc and RefCell we can become... an incredibly verbose pervasively mutable garbage collected language that can't collect cycles! Y-yaaaaay...

Alright, we want to be *doubly-linked*. This means each node has a pointer to the previous and next node. Also, the list itself has a pointer to the first and last node. This gives us fast insertion and removal on *both* ends of the list.

So we probably want something like:

```
use std::rc::Rc;
use std::cell::RefCell;

pub struct List<T> {
    head: Link<T>,
    tail: Link<T>,
}

type Link<T> = Option<Rc<RefCell<Node<T>>>>;

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

```
> cargo build

warning: field is never used: `head`
 --> src/fourth.rs:5:5
  |
5 |     head: Link<T>,
  |     ^^^^^^^^^^^^^
  |
  = note: #[warn(dead_code)] on by default

warning: field is never used: `tail`
 --> src/fourth.rs:6:5
  |
6 |     tail: Link<T>,
  |     ^^^^^^^^^^^^^

warning: field is never used: `elem`
  --> src/fourth.rs:12:5
   |
12 |     elem: T,
   |     ^^^^^^^

warning: field is never used: `next`
  --> src/fourth.rs:13:5
   |
13 |     next: Link<T>,
   |     ^^^^^^^^^^^^^

warning: field is never used: `prev`
  --> src/fourth.rs:14:5
   |
14 |     prev: Link<T>,
   |     ^^^^^^^^^^^^^
```

Hey, it built! Lots of dead code warnings, but it built! Let's try to use it.

Building Up

Alright, we'll start with building the list. That's pretty straight-forward with this new system. `new` is still trivial, just None out all the fields. Also because it's getting a bit unwieldy, let's break out a Node constructor too:

```
impl<T> Node<T> {
    fn new(elem: T) -> Rc<RefCell<Self>> {
        Rc::new(RefCell::new(Node {
            elem: elem,
            prev: None,
            next: None,
        }))
    }
}

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None, tail: None }
    }
}
```

```
> cargo build

**A BUNCH OF DEAD CODE WARNINGS BUT IT BUILT**
```

Yay!

Now let's try to write pushing onto the front of the list. Because doubly-linked lists are significantly more complicated, we're going to need to do a fair bit more work. Where singly-linked list operations could be reduced to an easy one-liner, doubly-linked list ops are fairly complicated.

In particular we now need to specially handle some boundary cases around empty lists. Most operations will only touch the `head` or `tail` pointer. However when transitioning to or from the empty list, we need to edit *both* at once.

An easy way for us to validate if our methods make sense is if we maintain the following invariant: each node should have exactly two pointers to it. Each node in the middle of the list is pointed at by its predecessor and successor, while the nodes on the ends are pointed to by the list itself.

Let's take a crack at it:

```
pub fn push_front(&mut self, elem: T) {
    // new node needs +2 links, everything else should be +0
    let new_head = Node::new(elem);
    match self.head.take() {
        Some(old_head) => {
            // non-empty list, need to connect the old_head
            old_head.prev = Some(new_head.clone()); // +1 new_head
            new_head.next = Some(old_head);         // +1 old_head
            self.head = Some(new_head);             // +1 new_head, -1 old_head
            // total: +2 new_head, +0 old_head -- OK!
        }
        None => {
            // empty list, need to set the tail
            self.tail = Some(new_head.clone());     // +1 new_head
            self.head = Some(new_head);             // +1 new_head
            // total: +2 new_head -- OK!
        }
    }
}
```

```
cargo build

error[E0609]: no field `prev` on type
`std::rc::Rc<std::cell::RefCell<fourth::Node<T>>>`
  --> src/fourth.rs:39:26
   |
39 |                 old_head.prev = Some(new_head.clone()); // +1 new_head
   |                          ^^^^ unknown field

error[E0609]: no field `next` on type
`std::rc::Rc<std::cell::RefCell<fourth::Node<T>>>`
  --> src/fourth.rs:40:26
   |
40 |                 new_head.next = Some(old_head);         // +1 old_head
   |                          ^^^^ unknown field
```

Alright. Compiler error. Good start. Good start.

Why can't we access the `prev` and `next` fields on our nodes? It worked before when we just had an `Rc<Node>`. Seems like the `RefCell` is getting in the way.

We should probably check the docs.

Google's "rust refcell"

clicks first link

A mutable memory location with dynamically checked borrow rules

See the module-level documentation for more.

---

clicks link

Shareable mutable containers.

Values of the `Cell<T>` and `RefCell<T>` types may be mutated through shared references (i.e. the common `&T` type), whereas most Rust types can only be mutated through unique (`&mut T`) references. We say that `Cell<T>` and `RefCell<T>` provide 'interior mutability', in contrast with typical Rust types that exhibit 'inherited mutability'.

Cell types come in two flavors: `Cell<T>` and `RefCell<T>`. `Cell<T>` provides `get` and `set` methods that change the interior value with a single method call. `Cell<T>` though is only compatible with types that implement `Copy`. For other types, one must use the `RefCell<T>` type, acquiring a write lock before mutating.

`RefCell<T>` uses Rust's lifetimes to implement 'dynamic borrowing', a process whereby one can claim temporary, exclusive, mutable access to the inner value. Borrows for `RefCell<T>`s are tracked 'at runtime', unlike Rust's native reference types which are entirely tracked statically, at compile time. Because `RefCell<T>` borrows are dynamic it is possible to attempt to borrow a value that is already mutably borrowed; when this happens it results in thread panic.

When to choose interior mutability

The more common inherited mutability, where one must have unique access to mutate a value, is one of the key language elements that enables Rust to reason strongly about pointer aliasing, statically preventing crash bugs. Because of that, inherited mutability is preferred, and interior mutability is something of a last resort. Since cell types enable mutation where it would otherwise be disallowed though, there are occasions when interior mutability might be appropriate, or even *must* be used, e.g.

- Introducing inherited mutability roots to shared types.
- Implementation details of logically-immutable methods.
- Mutating implementations of `Clone`.

Introducing inherited mutability roots to shared types

Shared smart pointer types, including `Rc<T>` and `Arc<T>`, provide containers that can be cloned and shared between multiple parties. Because the contained values may be multiply-aliased, they can only be borrowed as shared references, not mutable references. Without cells it would be impossible to mutate data inside of shared boxes at all!

It's very common then to put a `RefCell<T>` inside shared pointer types to reintroduce mutability:

```
use std::collections::HashMap;
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let shared_map: Rc<RefCell<_>> = Rc::new(RefCell::new(HashMap::new()));
    shared_map.borrow_mut().insert("africa", 92388);
    shared_map.borrow_mut().insert("kyoto", 11837);
    shared_map.borrow_mut().insert("piccadilly", 11826);
    shared_map.borrow_mut().insert("marbles", 38);
}
```

Note that this example uses `Rc<T>` and not `Arc<T>`. `RefCell<T>`s are for single-threaded scenarios. Consider using `Mutex<T>` if you need shared mutability in a multi-threaded situation.

Hey, Rust's docs continue to be incredibly awesome.

The meaty bit we care about is this line:

```
shared_map.borrow_mut().insert("africa", 92388);
```

In particular, the `borrow_mut` thing. Seems we need to explicitly borrow a RefCell. The `.` operator's not going to do it for us. Weird. Let's try:

```
pub fn push_front(&mut self, elem: T) {
    let new_head = Node::new(elem);
    match self.head.take() {
        Some(old_head) => {
            old_head.borrow_mut().prev = Some(new_head.clone());
            new_head.borrow_mut().next = Some(old_head);
            self.head = Some(new_head);
        }
        None => {
            self.tail = Some(new_head.clone());
            self.head = Some(new_head);
        }
    }
}
```

```
> cargo build

warning: field is never used: `elem`
  --> src/fourth.rs:12:5
   |
12 |     elem: T,
   |     ^^^^^^^
   |
   = note: #[warn(dead_code)] on by default
```

Hey, it built! Docs win again.

Breaking Down

`pop_front` should be the same basic logic as `push_front`, but backwards. Let's try:

```
pub fn pop_front(&mut self) -> Option<T> {
    // need to take the old head, ensuring it's -2
    self.head.take().map(|old_head| {                           // -1 old
        match old_head.borrow_mut().next.take() {
            Some(new_head) => {                                 // -1 new
                // not emptying list
                new_head.borrow_mut().prev.take();              // -1 old
                self.head = Some(new_head);                     // +1 new
                // total: -2 old, +0 new
            }
            None => {
                // emptying list
                self.tail.take();                               // -1 old
                // total: -2 old, (no new)
            }
        }
        old_head.elem
    })
}
```

```
> cargo build

error[E0609]: no field `elem` on type
`std::rc::Rc<std::cell::RefCell<fourth::Node<T>>>`
  --> src/fourth.rs:64:22
   |
64 |             old_head.elem
   |                      ^^^^ unknown field
```

ACK. *RefCells*. Gotta `borrow_mut` again I guess...

```
pub fn pop_front(&mut self) -> Option<T> {
    self.head.take().map(|old_head| {
        match old_head.borrow_mut().next.take() {
            Some(new_head) => {
                new_head.borrow_mut().prev.take();
                self.head = Some(new_head);
            }
            None => {
                self.tail.take();
            }
        }
        old_head.borrow_mut().elem
    })
}
```

```
cargo build

error[E0507]: cannot move out of borrowed content
  --> src/fourth.rs:64:13
   |
64 |             old_head.borrow_mut().elem
   |             ^^^^^^^^^^^^^^^^^^^^^^^^^^ cannot move out of borrowed content
```

sigh

---

cannot move out of borrowed content

Hrm... It seems that Box was *really* spoiling us. `borrow_mut` only gets us an `&mut Node<T>`, but we can't move out of that!

We need something that takes a `RefCell<T>` and gives us a `T`. Let's check the docs for something like that:

`fn into_inner(self) -> T`

Consumes the RefCell, returning the wrapped value.

That looks promising!

```
old_head.into_inner().elem
```

```
> cargo build

error[E0507]: cannot move out of an `Rc`
  --> src/fourth.rs:64:13
   |
64 |             old_head.into_inner().elem
   |             ^^^^^^^^ cannot move out of an `Rc`
```

Ah dang. `into_inner` wants to move out the RefCell, but we can't, because it's in an `Rc`. As we saw in the previous chapter, `Rc<T>` only lets us get shared references into its internals. That makes sense, because that's *the whole point* of reference counted pointers: they're shared!

This was a problem for us when we wanted to implement Drop for our reference counted list, and the solution is the same: `Rc::try_unwrap`, which moves out the contents of an Rc if its refcount is 1.

```
Rc::try_unwrap(old_head).unwrap().into_inner().elem
```

`Rc::try_unwrap` returns a `Result<T, Rc<T>>`. Results are basically a generalized `Option`, where the `None` case has data associated with it. In this case, the `Rc` you tried to unwrap. Since we don't care about the case where it fails (if we wrote our program correctly, it *has* to succeed), we just call `unwrap` on it.

Anyway, let's see what compiler error we get next (let's face it, there's going to be one).

```
> cargo build

error[E0599]: no method named `unwrap` found for type
`std::result::Result<std::cell::RefCell<fourth::Node<T>>,
std::rc::Rc<std::cell::RefCell<fourth::Node<T>>>>` in the current scope
  --> src/fourth.rs:64:38
   |
64 |             Rc::try_unwrap(old_head).unwrap().into_inner().elem
   |                                      ^^^^^^
   |
   = note: the method `unwrap` exists but the following trait bounds were not
satisfied:
           `std::rc::Rc<std::cell::RefCell<fourth::Node<T>>> : std::fmt::Debug`
```

UGH. `unwrap` on Result requires that you can debug-print the error case. `RefCell<T>` only implements `Debug` if `T` does. `Node` doesn't implement Debug.

Rather than doing that, let's just work around it by converting the Result to an Option with `ok`:

```
Rc::try_unwrap(old_head).ok().unwrap().into_inner().elem
```

PLEASE.

```
cargo build
```

YES.

phew

We did it.

We implemented `push` and `pop`.

Let's test by stealing the old `stack` basic test (because that's all that we've implemented so far):

```
#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop_front(), None);

        // Populate list
        list.push_front(1);
        list.push_front(2);
        list.push_front(3);

        // Check normal removal
        assert_eq!(list.pop_front(), Some(3));
        assert_eq!(list.pop_front(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push_front(4);
        list.push_front(5);

        // Check normal removal
        assert_eq!(list.pop_front(), Some(5));
        assert_eq!(list.pop_front(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop_front(), Some(1));
        assert_eq!(list.pop_front(), None);
    }
}
```

```
cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 9 tests
test first::test::basics ... ok
test fourth::test::basics ... ok
test second::test::iter_mut ... ok
test second::test::basics ... ok
test fifth::test::iter_mut ... ok
test third::test::basics ... ok
test second::test::iter ... ok
test third::test::iter ... ok
test second::test::into_iter ... ok

test result: ok. 9 passed; 0 failed; 0 ignored; 0 measured
```

Nailed it.

Now that we can properly remove things from the list, we can implement Drop. Drop is a little more conceptually interesting this time around. Where previously we bothered to implement Drop for our stacks just to avoid unbounded recursion, now we need to implement Drop to get *anything* to happen at all.

`Rc` can't deal with cycles. If there's a cycle, everything will keep everything else alive. A doubly-linked list, as it turns out, is just a big chain of tiny cycles! So when we drop our list, the two end nodes will have their refcounts decremented down to 1... and then nothing else will happen. Well, if our list contains exactly one node we're good to go. But ideally a list should work right if it contains multiple elements. Maybe that's just me.

As we saw, removing elements was a bit painful. So the easiest thing for us to do is just `pop` until we get None:

```
impl<T> Drop for List<T> {
    fn drop(&mut self) {
        while self.pop_front().is_some() {}
    }
}
```

```
cargo build
```

(We actually could have done this with our mutable stacks, but shortcuts are for people who understand things!)

We could look at implementing the `_back` versions of `push` and `pop`, but they're just copy-paste jobs which we'll defer to later in the chapter. For now let's look at more interesting things!

Peeking

Alright, we made it through `push` and `pop`. I'm not gonna lie, it got a bit emotional there. Compile-time correctness is a hell of a drug.

Let's cool off by doing something simple: let's just implement `peek_front`. That was always really easy before. Gotta still be easy, right?

Right?

In fact, I think I can just copy-paste it!

```
pub fn peek_front(&self) -> Option<&T> {
    self.head.as_ref().map(|node| {
        &node.elem
    })
}
```

Wait. Not this time.

```
pub fn peek_front(&self) -> Option<&T> {
    self.head.as_ref().map(|node| {
        // BORROW!!!!
        &node.borrow().elem
    })
}
```

HAH.

```
cargo build

error[E0515]: cannot return value referencing temporary value
  --> src/fourth.rs:66:13
   |
66 |             &node.borrow().elem
   |             ^   ----------^^^^^
   |             |   |
   |             |   temporary value created here
   |             |
   |             returns a value referencing data owned by the current function
```

Ok I'm just burning my computer.

This is exactly the same logic as our singly-linked stack. Why are things different. WHY.

The answer is really the whole moral of this chapter: RefCells make everything sadness. Up until now, RefCells have just been a nuisance. Now they're going to become a nightmare.

So what's going on? To understand that, we need to go back to the definition of `borrow`:

```
fn borrow<'a>(&'a self) -> Ref<'a, T>
fn borrow_mut<'a>(&'a self) -> RefMut<'a, T>
```

In the layout section we said:

> Rather than enforcing this statically, RefCell enforces them at runtime. If you break the rules, RefCell will just panic and crash the program. Why does it return these Ref and RefMut things? Well, they basically behave like `Rc`s but for borrowing. Also they keep the RefCell borrowed until they go out of scope. **We'll get to that later.**

It's later.

`Ref` and `RefMut` implement `Deref` and `DerefMut` respectively. So for most intents and purposes they behave *exactly* like `&T` and `&mut T`. However, because of how those traits work, the reference that's returned is connected to the lifetime of the Ref, and not the actual RefCell. This means that the Ref has to be sitting around as long as we keep the reference around.

This is in fact necessary for correctness. When a Ref gets dropped, it tells the RefCell that it's not borrowed anymore. So if we *did* manage to hold onto our reference longer than the Ref existed, we could get a RefMut while a reference was kicking around and totally break Rust's type system in half.

So where does that leave us? We only want to return a reference, but we need to keep this Ref thing around. But as soon as we return the reference from `peek`, the function is over and the `Ref` goes out of scope.

As far as I know, we're actually totally dead in the water here. You can't totally encapsulate the use of RefCells like that.

But... what if we just give up on totally hiding our implementation details? What if we returns Refs?

```
pub fn peek_front(&self) -> Option<Ref<T>> {
    self.head.as_ref().map(|node| {
        node.borrow()
    })
}
```

```
> cargo build

error[E0412]: cannot find type `Ref` in this scope
  --> src/fourth.rs:63:40
   |
63 |     pub fn peek_front(&self) -> Option<Ref<T>> {
   |                                        ^^^ not found in this scope
help: possible candidates are found in other modules, you can import them into scope
   |
1  | use core::cell::Ref;
   |
1  | use std::cell::Ref;
   |
```

Blurp. Gotta import some stuff.

```
use std::cell::{Ref, RefCell};
```

```
> cargo build

error[E0308]: mismatched types
  --> src/fourth.rs:64:9
   |
64 | /         self.head.as_ref().map(|node| {
65 | |             node.borrow()
66 | |         })
   | |__________^ expected type parameter, found struct `fourth::Node`
   |
   = note: expected type `std::option::Option<std::cell::Ref<'_, T>>`
              found type `std::option::Option<std::cell::Ref<'_, fourth::Node<T>>>`
```

Hmm... that's right. We have a `Ref<Node<T>>`, but we want a `Ref<T>`. We could abandon all hope of encapsulation and just return that. We could also make things even more complicated and wrap `Ref<Node<T>>` in a new type to only expose access to an `&T`.

Both of those options are *kinda* lame.

Instead, we're going to go deeper down. Let's have some *fun*. Our source of fun is *this beast*:

```
map<U, F>(orig: Ref<'b, T>, f: F) -> Ref<'b, U>
    where F: FnOnce(&T) -> &U,
          U: ?Sized
```

Make a new Ref for a component of the borrowed data.

Yes: just like you can map over an Option, you can map over a Ref.

I'm sure someone somewhere is really excited because *monads* or whatever but I don't care about any of that. Also I don't think it's a proper monad since there's no None-like case, but I digress.

It's cool and that's all that matters to me. *I need this*.

```
pub fn peek_front(&self) -> Option<Ref<T>> {
    self.head.as_ref().map(|node| {
        Ref::map(node.borrow(), |node| &node.elem)
    })
}
```

```
> cargo build
```

Awww yissss

Let's make sure this is working by munging up the test from our stack. We need to do some munging to deal with the fact that Refs don't implement comparisons.

```
#[test]
fn peek() {
    let mut list = List::new();
    assert!(list.peek_front().is_none());
    list.push_front(1); list.push_front(2); list.push_front(3);

    assert_eq!(&*list.peek_front().unwrap(), &3);
}
```

```
> cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 10 tests
test first::test::basics ... ok
test fourth::test::basics ... ok
test second::test::basics ... ok
test fourth::test::peek ... ok
test second::test::iter_mut ... ok
test second::test::into_iter ... ok
test third::test::basics ... ok
test second::test::peek ... ok
test second::test::iter ... ok
test third::test::iter ... ok

test result: ok. 10 passed; 0 failed; 0 ignored; 0 measured
```

Great!

Symmetric Junk

Alright let's get all that combinatoric symmetry over with.

All we have to do is some basic text replacement:

```
tail <-> head
next <-> prev
front -> back
```

Oh, also we need to add `_mut` variants for peeking.

```
use std::cell::{Ref, RefCell, RefMut};

//..

pub fn push_back(&mut self, elem: T) {
    let new_tail = Node::new(elem);
    match self.tail.take() {
        Some(old_tail) => {
            old_tail.borrow_mut().next = Some(new_tail.clone());
            new_tail.borrow_mut().prev = Some(old_tail);
            self.tail = Some(new_tail);
        }
        None => {
            self.head = Some(new_tail.clone());
            self.tail = Some(new_tail);
        }
    }
}

pub fn pop_back(&mut self) -> Option<T> {
    self.tail.take().map(|old_tail| {
        match old_tail.borrow_mut().prev.take() {
            Some(new_tail) => {
                new_tail.borrow_mut().next.take();
                self.tail = Some(new_tail);
            }
            None => {
                self.head.take();
            }
        }
        Rc::try_unwrap(old_tail).ok().unwrap().into_inner().elem
    })
}

pub fn peek_back(&self) -> Option<Ref<T>> {
    self.tail.as_ref().map(|node| {
        Ref::map(node.borrow(), |node| &node.elem)
    })
}

pub fn peek_back_mut(&mut self) -> Option<RefMut<T>> {
    self.tail.as_ref().map(|node| {
        RefMut::map(node.borrow_mut(), |node| &mut node.elem)
    })
}

pub fn peek_front_mut(&mut self) -> Option<RefMut<T>> {
    self.head.as_ref().map(|node| {
        RefMut::map(node.borrow_mut(), |node| &mut node.elem)
    })
}
```

And massively flesh out our tests:

```
#[test]
fn basics() {
    let mut list = List::new();

    // Check empty list behaves right
    assert_eq!(list.pop_front(), None);

    // Populate list
    list.push_front(1);
    list.push_front(2);
    list.push_front(3);

    // Check normal removal
    assert_eq!(list.pop_front(), Some(3));
    assert_eq!(list.pop_front(), Some(2));

    // Push some more just to make sure nothing's corrupted
    list.push_front(4);
    list.push_front(5);

    // Check normal removal
    assert_eq!(list.pop_front(), Some(5));
    assert_eq!(list.pop_front(), Some(4));

    // Check exhaustion
    assert_eq!(list.pop_front(), Some(1));
    assert_eq!(list.pop_front(), None);

    // ---- back -----

    // Check empty list behaves right
    assert_eq!(list.pop_back(), None);

    // Populate list
    list.push_back(1);
    list.push_back(2);
    list.push_back(3);

    // Check normal removal
    assert_eq!(list.pop_back(), Some(3));
    assert_eq!(list.pop_back(), Some(2));

    // Push some more just to make sure nothing's corrupted
    list.push_back(4);
    list.push_back(5);

    // Check normal removal
    assert_eq!(list.pop_back(), Some(5));
    assert_eq!(list.pop_back(), Some(4));

    // Check exhaustion
    assert_eq!(list.pop_back(), Some(1));
    assert_eq!(list.pop_back(), None);
}

#[test]
fn peek() {
    let mut list = List::new();
    assert!(list.peek_front().is_none());
    assert!(list.peek_back().is_none());
    assert!(list.peek_front_mut().is_none());
    assert!(list.peek_back_mut().is_none());

    list.push_front(1); list.push_front(2); list.push_front(3);

    assert_eq!(&*list.peek_front().unwrap(), &3);
    assert_eq!(&mut *list.peek_front_mut().unwrap(), &mut 3);
```

```
    assert_eq!(&*list.peek_back().unwrap(), &1);
    assert_eq!(&mut *list.peek_back_mut().unwrap(), &mut 1);
}
```

Are there some cases we're not testing? Probably. The combinatoric space has really blown up here. Our code is at very least not *obviously wrong*.

```
> cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 10 tests
test first::test::basics ... ok
test fourth::test::basics ... ok
test second::test::basics ... ok
test fourth::test::peek ... ok
test second::test::iter ... ok
test third::test::iter ... ok
test second::test::into_iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::basics ... ok

test result: ok. 10 passed; 0 failed; 0 ignored; 0 measured
```

Nice. Copy-pasting is the best kind of programming.

Iteration

Let's take a crack at iterating this bad-boy.

IntoIter

IntoIter, as always, is going to be the easiest. Just wrap the stack and call `pop`:

```
pub struct IntoIter<T>(List<T>);

impl<T> List<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        self.0.pop_front()
    }
}
```

But we have an interesting new development. Where previously there was only ever one "natural" iteration order for our lists, a Deque is inherently bi-directional. What's so special about front-to-back? What if someone wants to iterate in the other direction?

Rust actually has an answer to this: `DoubleEndedIterator`. DoubleEndedIterator *inherits* from Iterator (meaning all DoubleEndedIterator are Iterators) and requires one new method: `next_back`.

It has the exact same signature as `next`, but it's supposed to yield elements from the other end. The semantics of DoubleEndedIterator are super convenient for us: the iterator becomes a deque. You can consume elements from the front and back until the two ends converge, at which point the iterator is empty.

Much like Iterator and `next`, it turns out that `next_back` isn't really something consumers of the DoubleEndedIterator really care about. Rather, the best part of this interface is that it exposes the `rev` method, which wraps up the iterator to make a new one that yields the elements in reverse order. The semantics of this are fairly straight-forward: calls to `next` on the reversed iterator are just calls to `next_back`.

Anyway, because we're already a deque providing this API is pretty easy:

```
impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> {
        self.0.pop_back()
    }
}
```

And let's test it out:

```
#[test]
fn into_iter() {
    let mut list = List::new();
    list.push_front(1); list.push_front(2); list.push_front(3);

    let mut iter = list.into_iter();
    assert_eq!(iter.next(), Some(3));
    assert_eq!(iter.next_back(), Some(1));
    assert_eq!(iter.next(), Some(2));
    assert_eq!(iter.next_back(), None);
    assert_eq!(iter.next(), None);
}
```

```
cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 11 tests
test fourth::test::basics ... ok
test fourth::test::peek ... ok
test fourth::test::into_iter ... ok
test first::test::basics ... ok
test second::test::basics ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test third::test::iter ... ok
test third::test::basics ... ok
test second::test::into_iter ... ok
test second::test::peek ... ok

test result: ok. 11 passed; 0 failed; 0 ignored; 0 measured
```

Nice.

Iter

Iter will be a bit less forgiving. We'll have to deal with those awful `Ref` things again! Because of Refs, we can't store `&Node`s like we did before. Instead, let's try to store `Ref<Node>`s:

```
pub struct Iter<'a, T>(Option<Ref<'a, Node<T>>>);

impl<T> List<T> {
    pub fn iter(&self) -> Iter<T> {
        Iter(self.head.as_ref().map(|head| head.borrow()))
    }
}
```

```
> cargo build
```

So far so good. Implementing `next` is going to be a bit hairy, but I think it's the same basic logic as the old stack IterMut but with extra RefCell madness:

```
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = Ref<'a, T>;
    fn next(&mut self) -> Option<Self::Item> {
        self.0.take().map(|node_ref| {
            self.0 = node_ref.next.as_ref().map(|head| head.borrow());
            Ref::map(node_ref, |node| &node.elem)
        })
    }
}
```

```
cargo build

error[E0521]: borrowed data escapes outside of closure
   --> src/fourth.rs:155:13
    |
153 |     fn next(&mut self) -> Option<Self::Item> {
    |             --------- `self` is declared here, outside of the closure body
154 |         self.0.take().map(|node_ref| {
155 |             self.0 = node_ref.next.as_ref().map(|head| head.borrow());
    |             ^^^^^^   -------- borrow is only valid in the closure body
    |             |
    |             reference to `node_ref` escapes the closure body here

error[E0505]: cannot move out of `node_ref` because it is borrowed
   --> src/fourth.rs:156:22
    |
153 |     fn next(&mut self) -> Option<Self::Item> {
    |             --------- lifetime `'1` appears in the type of `self`
154 |         self.0.take().map(|node_ref| {
155 |             self.0 = node_ref.next.as_ref().map(|head| head.borrow());
    |             ------   -------- borrow of `node_ref` occurs here
    |             |
    |             assignment requires that `node_ref` is borrowed for `'1`
156 |             Ref::map(node_ref, |node| &node.elem)
    |                      ^^^^^^^^ move out of `node_ref` occurs here
```

Shoot.

`node_ref` doesn't live long enough. Unlike normal references, Rust doesn't let us just split Refs up like that. The Ref we get out of `head.borrow()` is only allowed to live as long as `node_ref`, but we end up trashing that in our `Ref::map` call.

The function we want exists, and it's called *[map_split][]*:

```
pub fn map_split<U, V, F>(orig: Ref<'b, T>, f: F) -> (Ref<'b, U>, Ref<'b, V>) where
    F: FnOnce(&T) -> (&U, &V),
    U: ?Sized,
    V: ?Sized,
```

Woof. Let's give it a try...

```
fn next(&mut self) -> Option<Self::Item> {
    self.0.take().map(|node_ref| {
        let (next, elem) = Ref::map_split(node_ref, |node| {
            (&node.next, &node.elem)
        });

        self.0 = next.as_ref().map(|head| head.borrow());

        elem
    })
}
```

```
cargo build
   Compiling lists v0.1.0 (/Users/ABeingessner/dev/temp/lists)
error[E0521]: borrowed data escapes outside of closure
   --> src/fourth.rs:159:13
    |
153 |     fn next(&mut self) -> Option<Self::Item> {
    |             ---------  `self` is declared here, outside of the closure body
...
159 |             self.0 = next.as_ref().map(|head| head.borrow());
    |             ^^^^^^   ---- borrow is only valid in the closure body
    |             |
    |             reference to `next` escapes the closure body here
```

Ergh. We need to `Ref::Map` again to get our lifetimes right. But `Ref::Map` returns a `Ref` and we need an `Option<Ref>`, but we need to go through the Ref to map over our Option...

stares into distance for a long time

??????

```
fn next(&mut self) -> Option<Self::Item> {
    self.0.take().map(|node_ref| {
        let (next, elem) = Ref::map_split(node_ref, |node| {
            (&node.next, &node.elem)
        });

        self.0 = if next.is_some() {
            Some(Ref::map(next, |next| &**next.as_ref().unwrap()))
        } else {
            None
        };

        elem
    })
}
```

```
error[E0308]: mismatched types
   --> src/fourth.rs:162:22
    |
162 |                 Some(Ref::map(next, |next| &**next.as_ref().unwrap()))
    |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected
struct `fourth::Node`, found struct `std::cell::RefCell`
    |
    = note: expected type `std::cell::Ref<'_, fourth::Node<_>>`
               found type `std::cell::Ref<'_, std::cell::RefCell<fourth::Node<_>>>`
```

Oh. Right. There's multiple RefCells. The deeper we walk into the list, the more nested we become under each RefCell. We would need to maintain, like, a stack of Refs to represent all the outstanding loans we're holding, because if we stop looking at an element we need to decrement the borrow-count on every RefCell that comes before it.....

I don't think there's anything we can do here. It's a dead end. Let's try getting out of the RefCells.

What about our `Rc`s. Who said we even needed to store references? Why can't we just Clone the whole Rc to get a nice owning handle into the middle of the list?

```
pub struct Iter<T>(Option<Rc<Node<T>>>);

impl<T> List<T> {
    pub fn iter(&self) -> Iter<T> {
        Iter(self.head.as_ref().map(|head| head.clone()))
    }
}

impl<T> Iterator for Iter<T> {
    type Item =
```

Uh... Wait what do we return now? `&T`? `Ref<T>`?

No, none of those work... our Iter doesn't have a lifetime anymore! Both `&T` and `Ref<T>` require us to declare some lifetime up front before we get into `next`. But anything we manage to get out of our Rc would be borrowing the Iterator... brain... hurt... aaaaaahhhhhh

Maybe we can... map... the Rc... to get an `Rc<T>`? Is that a thing? Rc's docs don't seem to have anything like that. Actually someone made a crate that lets you do that.

But wait, even if we do *that* then we've got an even bigger problem: the dreaded spectre of iterator invalidation. Previously we've been totally immune to iterator invalidation, because the Iter borrowed the list, leaving it totally immutable. However if our Iter was yielding Rcs, they wouldn't borrow the list at all! That means people can start calling `push` and `pop` on the list while they hold pointers into it!

Oh lord, what will that do?!

Well, pushing is actually fine. We've got a view into some sub-range of the list, and the list will just grow beyond our sights. No biggie.

However `pop` is another story. If they're popping elements outside of our range, it should *still* be fine. We can't see those nodes so nothing will happen. However if they try to pop off the node we're pointing at... everything will blow up! In particular when they go to `unwrap` the result of the `try_unwrap`, it will actually fail, and the whole program will panic.

That's actually pretty cool. We can get tons of interior owning pointers into the list and mutate it at the same time *and it will just work* until they try to remove the nodes that we're pointing at. And even then we don't get dangling pointers or anything, the program will deterministically panic!

But having to deal with iterator invalidation on top of mapping Rcs just seems... bad. `Rc<RefCell>` has really truly finally failed us. Interestingly, we've experienced an inversion of the persistent stack case. Where the persistent stack struggled to ever reclaim ownership of the data but could get references all day every day, our list had no problem gaining ownership, but really struggled to loan our references.

Although to be fair, most of our struggles revolved around wanting to hide the implementation details and have a decent API. We *could* do everything fine if we wanted to just pass around Nodes all over the place.

Heck, we could make multiple concurrent IterMuts that were runtime checked to not be mutable accessing the same element!

Really, this design is more appropriate for an internal data structure that never makes it out to consumers of the API. Interior mutability is great for writing safe *applications*. Not so much safe *libraries*.

Anyway, that's me giving up on Iter and IterMut. We could do them, but *ugh*.

Final Code

Alright, so that's implementing a 100% safe doubly-linked list in Rust. It was a nightmare to implement, leaks implementation details, and doesn't support several fundamental operations.

But, it exists.

Oh, I guess it's also riddled with tons of "unnecessary" runtime checks for correctness between `Rc` and `RefCell`. I put unnecessary in quotes because they're actually necessary to guarantee the whole *actually being safe* thing. We encountered a few places where those checks actually *were* necessary. Doubly-linked lists have a horribly tangled aliasing and ownership story!

Still, it's a thing we can do. Especially if we don't care about exposing internal data structures to our consumers.

From here on out, we're going to be focusing on other side of this coin: getting back all the control by making our implementation *unsafe*.

```
use std::rc::Rc;
use std::cell::{Ref, RefMut, RefCell};

pub struct List<T> {
    head: Link<T>,
    tail: Link<T>,
}

type Link<T> = Option<Rc<RefCell<Node<T>>>>;

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


impl<T> Node<T> {
    fn new(elem: T) -> Rc<RefCell<Self>> {
        Rc::new(RefCell::new(Node {
            elem: elem,
            prev: None,
            next: None,
        }))
    }
}

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None, tail: None }
    }

    pub fn push_front(&mut self, elem: T) {
        let new_head = Node::new(elem);
        match self.head.take() {
            Some(old_head) => {
                old_head.borrow_mut().prev = Some(new_head.clone());
                new_head.borrow_mut().next = Some(old_head);
                self.head = Some(new_head);
            }
            None => {
                self.tail = Some(new_head.clone());
                self.head = Some(new_head);
            }
        }
    }

    pub fn push_back(&mut self, elem: T) {
        let new_tail = Node::new(elem);
        match self.tail.take() {
            Some(old_tail) => {
                old_tail.borrow_mut().next = Some(new_tail.clone());
                new_tail.borrow_mut().prev = Some(old_tail);
                self.tail = Some(new_tail);
            }
            None => {
                self.head = Some(new_tail.clone());
                self.tail = Some(new_tail);
            }
        }
    }

    pub fn pop_back(&mut self) -> Option<T> {
        self.tail.take().map(|old_tail| {
            match old_tail.borrow_mut().prev.take() {
                Some(new_tail) => {
```

```
                    new_tail.borrow_mut().next.take();
                    self.tail = Some(new_tail);
                }
                None => {
                    self.head.take();
                }
            }
            Rc::try_unwrap(old_tail).ok().unwrap().into_inner().elem
        })
    }

    pub fn pop_front(&mut self) -> Option<T> {
        self.head.take().map(|old_head| {
            match old_head.borrow_mut().next.take() {
                Some(new_head) => {
                    new_head.borrow_mut().prev.take();
                    self.head = Some(new_head);
                }
                None => {
                    self.tail.take();
                }
            }
            Rc::try_unwrap(old_head).ok().unwrap().into_inner().elem
        })
    }

    pub fn peek_front(&self) -> Option<Ref<T>> {
        self.head.as_ref().map(|node| {
            Ref::map(node.borrow(), |node| &node.elem)
        })
    }

    pub fn peek_back(&self) -> Option<Ref<T>> {
        self.tail.as_ref().map(|node| {
            Ref::map(node.borrow(), |node| &node.elem)
        })
    }

    pub fn peek_back_mut(&mut self) -> Option<RefMut<T>> {
        self.tail.as_ref().map(|node| {
            RefMut::map(node.borrow_mut(), |node| &mut node.elem)
        })
    }

    pub fn peek_front_mut(&mut self) -> Option<RefMut<T>> {
        self.head.as_ref().map(|node| {
            RefMut::map(node.borrow_mut(), |node| &mut node.elem)
        })
    }

    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }
}

impl<T> Drop for List<T> {
    fn drop(&mut self) {
        while self.pop_front().is_some() {}
    }
}

pub struct IntoIter<T>(List<T>);

impl<T> Iterator for IntoIter<T> {
    type Item = T;

    fn next(&mut self) -> Option<T> {
        self.0.pop_front()
```

```
    }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> {
        self.0.pop_back()
    }
}

#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop_front(), None);

        // Populate list
        list.push_front(1);
        list.push_front(2);
        list.push_front(3);

        // Check normal removal
        assert_eq!(list.pop_front(), Some(3));
        assert_eq!(list.pop_front(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push_front(4);
        list.push_front(5);

        // Check normal removal
        assert_eq!(list.pop_front(), Some(5));
        assert_eq!(list.pop_front(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop_front(), Some(1));
        assert_eq!(list.pop_front(), None);

        // ---- back -----

        // Check empty list behaves right
        assert_eq!(list.pop_back(), None);

        // Populate list
        list.push_back(1);
        list.push_back(2);
        list.push_back(3);

        // Check normal removal
        assert_eq!(list.pop_back(), Some(3));
        assert_eq!(list.pop_back(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push_back(4);
        list.push_back(5);

        // Check normal removal
        assert_eq!(list.pop_back(), Some(5));
        assert_eq!(list.pop_back(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop_back(), Some(1));
        assert_eq!(list.pop_back(), None);
    }
```

```
    #[test]
    fn peek() {
        let mut list = List::new();
        assert!(list.peek_front().is_none());
        assert!(list.peek_back().is_none());
        assert!(list.peek_front_mut().is_none());
        assert!(list.peek_back_mut().is_none());

        list.push_front(1); list.push_front(2); list.push_front(3);

        assert_eq!(&*list.peek_front().unwrap(), &3);
        assert_eq!(&mut *list.peek_front_mut().unwrap(), &mut 3);
        assert_eq!(&*list.peek_back().unwrap(), &1);
        assert_eq!(&mut *list.peek_back_mut().unwrap(), &mut 1);
    }

    #[test]
    fn into_iter() {
        let mut list = List::new();
        list.push_front(1); list.push_front(2); list.push_front(3);

        let mut iter = list.into_iter();
        assert_eq!(iter.next(), Some(3));
        assert_eq!(iter.next_back(), Some(1));
        assert_eq!(iter.next(), Some(2));
        assert_eq!(iter.next_back(), None);
        assert_eq!(iter.next(), None);
    }
}
```

An Ok Unsafe Singly-Linked Queue

Ok that reference-counted interior mutability stuff got a little out of control. Surely Rust doesn't really expect you to do that sort of thing in general? Well, yes and no. Rc and Refcell can be great for handling simple cases, but they can get unwieldy. Especially if you want to hide that it's happening. There's gotta be a better way!

In this chapter we're going to roll back to singly-linked lists and implement a singly-linked queue to dip our toes into *raw pointers* and *Unsafe Rust*.

> **NARRATOR:** And I will point out the mistakes.

And we won't make *any* mistakes.

Let's add a new file called `fifth.rs`:

```
// in lib.rs

pub mod first;
pub mod second;
pub mod third;
pub mod fourth;
pub mod fifth;
```

Our code is largely going to be derived from second.rs, since a queue is mostly an augmentation of a stack in the world of linked lists. Still, we're going to go from scratch because there's some fundamental issues we want to address with layout and what-not.

Layout

So what's a singly-linked queue like? Well, when we had a singly-linked stack we pushed onto one end of the list, and then popped off the same end. The only difference between a stack and a queue is that a queue pops off the *other* end. So from our stack implementation we have:

```
input list:
[Some(ptr)] -> (A, Some(ptr)) -> (B, None)

stack push X:
[Some(ptr)] -> (X, Some(ptr)) -> (A, Some(ptr)) -> (B, None)

stack pop:
[Some(ptr)] -> (A, Some(ptr)) -> (B, None)
```

To make a queue, we just need to decide which operation to move to the end of the list: push, or pop? Since our list is singly-linked, we can actually move *either* operation to the end with the same amount of effort.

To move `push` to the end, we just walk all the way to the `None` and set it to Some with the new element.

```
input list:
[Some(ptr)] -> (A, Some(ptr)) -> (B, None)

flipped push X:
[Some(ptr)] -> (A, Some(ptr)) -> (B, Some(ptr)) -> (X, None)
```

To move `pop` to the end, we just walk all the way to the node *before* the None, and `take` it:

```
input list:
[Some(ptr)] -> (A, Some(ptr)) -> (B, Some(ptr)) -> (X, None)

flipped pop:
[Some(ptr)] -> (A, Some(ptr)) -> (B, None)
```

We could do this today and call it quits, but that would stink! Both of these operations walk over the *entire* list. Some would argue that such a queue implementation is indeed a queue because it exposes the right interface. However I believe that performance guarantees are part of the interface. I don't care about precise asymptotic bounds, just "fast" vs "slow". Queues guarantee that push and pop are fast, and walking over the whole list is definitely *not* fast.

One key observation is that we're wasting a ton of work doing *the same thing* over and over. Can we "cache" all that work and reuse it? Why, yes! We can store a pointer to the end of the list, and just jump straight to there!

It turns out that only one inversion of `push` and `pop` works with this. To invert `pop` we would have to move the "tail" pointer backwards, but because our list is singly-linked, we can't do that efficiently. If we instead invert `push` we only have to move the "head" pointer forward, which is easy.

Let's try that:

```
use std::mem;

pub struct List<T> {
    head: Link<T>,
    tail: Link<T>, // NEW!
}

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

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

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None, tail: None }
    }

    pub fn push(&mut self, elem: T) {
        let new_tail = Box::new(Node {
            elem: elem,
            // When you push onto the tail, your next is always None
            next: None,
        });

        // swap the old tail to point to the new tail
        let old_tail = mem::replace(&mut self.tail, Some(new_tail));

        match old_tail {
            Some(mut old_tail) => {
                // If the old tail existed, update it to point to the new tail
                old_tail.next = Some(new_tail);
            }
            None => {
                // Otherwise, update the head to point to it
                self.head = Some(new_tail);
            }
        }
    }
}
```

I'm going a bit faster with the impl details now since we should be pretty comfortable with this sort of thing. Not that you should necessarily expect to produce this code on the first try. I'm just skipping over some of the trial-and-error we've had to deal with before. I actually made a ton of mistakes writing this code that I'm not showing, but you can only see me leave off a `mut` or `;` so many times before it stops being instructive. Don't worry, we'll see plenty of *other* error messages!

```
> cargo build

error[E0382]: use of moved value: `new_tail`
  --> src/fifth.rs:38:38
   |
26 |         let new_tail = Box::new(Node {
   |             -------- move occurs because `new_tail` has type
`std::boxed::Box<fifth::Node<T>>`, which does not implement the `Copy` trait
...
33 |         let old_tail = mem::replace(&mut self.tail, Some(new_tail));
   |                                                     -------- value moved here
...
38 |                 old_tail.next = Some(new_tail);
   |                                 ^^^^^^^^ value used here after move
```

Shoot!

```
use of moved value: `new_tail`
```

Box doesn't implement Copy, so we can't just assign it to two locations. More importantly, Box *owns* the thing it points to, and will try to free it when it's dropped. If our `push` implementation compiled, we'd double-free the tail of our list! Actually, as written, our code would free the old_tail on every push. Yikes! 🙀

Alright, well we know how to make a non-owning pointer. That's just a reference!

```
pub struct List<T> {
    head: Link<T>,
    tail: Option<&mut Node<T>>, // NEW!
}

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

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

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None, tail: None }
    }

    pub fn push(&mut self, elem: T) {
        let new_tail = Box::new(Node {
            elem: elem,
            // When you push onto the tail, your next is always None
            next: None,
        });

        // Put the box in the right place, and then grab a reference to its Node
        let new_tail = match self.tail.take() {
            Some(old_tail) => {
                // If the old tail existed, update it to point to the new tail
                old_tail.next = Some(new_tail);
                old_tail.next.as_deref_mut()
            }
            None => {
                // Otherwise, update the head to point to it
                self.head = Some(new_tail);
                self.head.as_deref_mut()
            }
        };

        self.tail = new_tail;
    }
}
```

Nothing too tricky here. Same basic idea as the previous code, except we're using some of that implicit return goodness to extract the tail reference from wherever we stuff the actual Box.

```
> cargo build

error[E0106]: missing lifetime specifier
 --> src/fifth.rs:3:18
  |
3 |     tail: Option<&mut Node<T>>, // NEW!
  |                  ^ expected lifetime parameter
```

Oh right, we need to give lifetimes to references in types. Hmm... what's the lifetime of this reference? Well, this seems like IterMut, right? Let's try what we did for IterMut, and just add a generic `'a`:

```
pub struct List<'a, T> {
    head: Link<T>,
    tail: Option<&'a mut Node<T>>, // NEW!
}

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

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

impl<'a, T> List<'a, T> {
    pub fn new() -> Self {
        List { head: None, tail: None }
    }

    pub fn push(&mut self, elem: T) {
        let new_tail = Box::new(Node {
            elem: elem,
            // When you push onto the tail, your next is always None
            next: None,
        });

        // Put the box in the right place, and then grab a reference to its Node
        let new_tail = match self.tail.take() {
            Some(old_tail) => {
                // If the old tail existed, update it to point to the new tail
                old_tail.next = Some(new_tail);
                old_tail.next.as_deref_mut()
            }
            None => {
                // Otherwise, update the head to point to it
                self.head = Some(new_tail);
                self.head.as_deref_mut()
            }
        };

        self.tail = new_tail;
    }
}
```

```
cargo build

error[E0495]: cannot infer an appropriate lifetime for autoref due to conflicting
requirements
  --> src/fifth.rs:35:27
   |
35 |                 self.head.as_deref_mut()
   |                           ^^^^^^^^^^^^
   |
note: first, the lifetime cannot outlive the anonymous lifetime #1 defined on the
method body at 18:5...
  --> src/fifth.rs:18:5
   |
18 | /     pub fn push(&mut self, elem: T) {
19 | |         let new_tail = Box::new(Node {
20 | |             elem: elem,
21 | |             // When you push onto the tail, your next is always None
...  |
39 | |         self.tail = new_tail;
40 | |     }
   | |_____^
note: ...so that reference does not outlive borrowed content
  --> src/fifth.rs:35:17
   |
35 |                 self.head.as_deref_mut()
   |                 ^^^^^^^^^
note: but, the lifetime must be valid for the lifetime 'a as defined on the impl at
13:6...
  --> src/fifth.rs:13:6
   |
13 | impl<'a, T> List<'a, T> {
   |      ^^
   = note: ...so that the expression is assignable:
           expected std::option::Option<&'a mut fifth::Node<T>>
              found std::option::Option<&mut fifth::Node<T>>
```

Woah, that's a really detailed error message. That's a bit concerning, because it suggests we're doing something really messed up. Here's an interesting part:

> the lifetime must be valid for the lifetime `'a` as defined on the impl

---

We're borrowing from `self`, but the compiler wants us to last as long as `'a`, what if we tell it `self` *does* last that long..?

```
    pub fn push(&'a mut self, elem: T) {
```

```
cargo build

warning: field is never used: `elem`
 --> src/fifth.rs:9:5
  |
9 |     elem: T,
  |     ^^^^^^^
  |
  = note: #[warn(dead_code)] on by default
```

Oh, hey, that worked! Great!

Let's just do `pop` too:

```
pub fn pop(&'a mut self) -> Option<T> {
    // Grab the list's current head
    self.head.take().map(|head| {
        let head = *head;
        self.head = head.next;

        // If we're out of `head`, make sure to set the tail to `None`.
        if self.head.is_none() {
            self.tail = None;
        }

        head.elem
    })
}
```

And write a quick test for that:

```
#[cfg(test)]
mod test {
    use super::List;
    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop(), None);

        // Populate list
        list.push(1);
        list.push(2);
        list.push(3);

        // Check normal removal
        assert_eq!(list.pop(), Some(1));
        assert_eq!(list.pop(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push(4);
        list.push(5);

        // Check normal removal
        assert_eq!(list.pop(), Some(3));
        assert_eq!(list.pop(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop(), Some(5));
        assert_eq!(list.pop(), None);
    }
}
```

```
cargo test

error[E0499]: cannot borrow `list` as mutable more than once at a time
  --> src/fifth.rs:68:9
   |
65 |         assert_eq!(list.pop(), None);
   |                    ---- first mutable borrow occurs here
...
68 |         list.push(1);
   |         ^^^^
   |         |
   |         second mutable borrow occurs here
   |         first borrow later used here

error[E0499]: cannot borrow `list` as mutable more than once at a time
  --> src/fifth.rs:69:9
   |
65 |         assert_eq!(list.pop(), None);
   |                    ---- first mutable borrow occurs here
...
69 |         list.push(2);
   |         ^^^^
   |         |
   |         second mutable borrow occurs here
   |         first borrow later used here

error[E0499]: cannot borrow `list` as mutable more than once at a time
  --> src/fifth.rs:70:9
   |
65 |         assert_eq!(list.pop(), None);
   |                    ---- first mutable borrow occurs here
...
70 |         list.push(3);
   |         ^^^^
   |         |
   |         second mutable borrow occurs here
   |         first borrow later used here


....

** WAY MORE LINES OF ERRORS **

....

error: aborting due to 11 previous errors
```

Oh my goodness.

The compiler's not wrong for vomiting all over us. We just committed a cardinal Rust sin: we stored a reference to ourselves *inside ourselves*. Somehow, we managed to convince Rust that this totally made sense in our `push` and `pop` implementations (I was legitimately shocked we did).

The reason this *sort of* works is that Rust doesn't really have the notion of a pointer into yourself at all. Each part of the code is *technically* correct in isolation (we *can* call push and pop *once*) but then the absurdity of what we created takes affect and everything just *locks up*.

I'm sure there is *some* use for what we've written, but as far as *I'm* concerned it's just syntatically valid *gibberish*. We're saying we contain something with lifetime `'a`, and that `push` and `pop` borrows *self* for that lifetime. That's *weird* but Rust can look at each part of our code *individually* and it doesn't see any rules being broken.

But as soon as we try to actually *use* the list, the compiler quickly goes "yep you've borrowed `self` mutably for `'a`, so you can't use `self` anymore until the end of `'a`" but *also* "because you contain `'a`, it must be valid for the entire list's existence".

It's *nearly* a contradiction but there *is* one solution: as soon as you `push` or `pop`, the list "pins" itself in place and can't be accessed anymore. It has swallowed its own proverbial tail, and ascended to a world of dreams.

> **NARRATOR**: it didn't exist when this book was first written, but Rust actually formalized the notion of a *pin* into something useful! This was probably the most complex addition to the language since *the borrowchecker*. We don't *want* our list to be pinned though!
>
> Pins *are* necessary and useful for async-await/futures/coroutines because the compiler needs to be able to bundle up all the local variables of a function into some kind of struct and store them somewhere until the future/coroutine is ready to be resumed. Since local variables can reference other local variables, and we want that to *work*, these structs can end up containing references to themselves!
>
> So to `await` or `yield` Rust needs a way to be able to properly describe and manipulate pinned values. Thankfully all of this stuff is *largely* just hidden away in automatic compiler machinery and no one actually has to think about `Pin` (or even *Futures*) under normal circumstances. The main exception is that this stuff is very important for the folks building and designing async *runtimes* like tokio.
>
> We will not be implementing an async runtime in this book. I know my friends know all sorts of "cool" (messed up) *tricks* you can do with `Pin`, but from what I can tell, I'd be happier to just not know them. I will continue to tell myself that Pinned types aren't real and they can't hurt me.

Our `pop` implementation hints at why storing a reference to ourselves *inside* ourselves could be really dangerous:

```
// ...
if self.head.is_none() {
    self.tail = None;
}
```

What if we forgot to do this? Then our tail would point to some node *that had been removed from the list*. Such a node would be instantly freed, and we'd have a dangling pointer which Rust was supposed to protect us from!

And indeed Rust is protecting us from that kind of danger. Just in a very... **roundabout** way.

So what can we do? Go back to `Rc<RefCell>>` hell?

Please. No.

No, instead we're going to go off the rails and use *raw pointers*. Our layout is going to look like this:

```
pub struct List<T> {
    head: Link<T>,
    tail: *mut Node<T>, // DANGER DANGER
}

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

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

And that's that. None of this wimpy reference-counted-dynamic-borrow-checking nonsense! Real. Hard. Unchecked. Pointers.

> **NARRATOR:** This implementation was in fact still dangerously wrong, but it wasn't yet time to learn that lesson. The next section will learn that the hard way, as usual.

Let's be C everyone. Let's be C all day.

I'm home. I'm ready.

Hello `unsafe`.

> **NARRATOR:** Wow, just incredible hubris from the author here.

Unsafe Rust

This is a serious, big, complicated, and dangerous topic. It's so serious that I wrote an entire other book on it.

The long and the short of it is that *every* language is actually unsafe as soon as you allow calling into other languages, because you can just have C do arbitrarily bad things. Yes: Java, Python, Ruby, Haskell... everyone is wildly unsafe in the face of Foreign Function Interfaces (FFI).

Rust embraces this truth by splitting itself into two languages: Safe Rust, and Unsafe Rust. So far we've only worked with Safe Rust. It's completely 100% safe... except that it can FFI into Unsafe Rust.

Unsafe Rust is a *superset* of Safe Rust. It's completely the same as Safe Rust in all its semantics and rules, you're just allowed to do a few *extra* things that are wildly unsafe and can cause the dreaded Undefined Behaviour that haunts C.

Again, this is a really huge topic that has a lot of interesting corner cases. I *really* don't want to go really deep into it (well, I do. I did. Read that book). That's ok, because with linked lists we can actually ignore almost all of it.

> **NARRATOR:** This was a lie, but it did seem true in 2015.

The main Unsafe tool we'll be using are *raw pointers*. Raw pointers are basically C's pointers. They have no inherent aliasing rules. They have no lifetimes. They can be null. They can be misaligned. They can be dangling. They can point to uninitialized memory. They can be cast to and from integers.

They can be cast to point to a different type. Mutability? Cast it. Pretty much everything goes, and that means pretty much anything can go wrong.

> **NARRATOR:** no inherent aliasing rules, eh? Ah, the innocence of youth.

This is some bad stuff and honestly you'll live a happier life never having to touch these. Unfortunately, we want to write linked lists, and linked lists are awful. That means we're going to have to use unsafe pointers.

There are two kinds of raw pointer: `*const T` and `*mut T`. These are meant to be `const T*` and `T*` from C, but we really don't care about what C thinks they mean that much. You can only dereference a `*const T` to an `&T`, but much like the mutability of a variable, this is just a lint against incorrect usage. At most it just means you have to cast the `*const` to a `*mut` first. Although if you don't actually have permission to mutate the referent of the pointer, you're gonna have a bad time.

Anyway, we'll get a better feel for this as we write some code. For now, `*mut T == &unchecked mut T`!

Basics

> **NARRATOR:** This section has a looming fundamental error in it, because that's the whole point of the book. However once we start using `unsafe` it's possible to do things wrong and still have everything compile and *seemingly* work. The fundamental mistake will be identified in the next section. Don't actually use the contents of this section in production code!

Alright, back to basics. How do we construct our list?

Before we just did:

```
impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None, tail: None }
    }
}
```

But we're not using Option for the `tail` anymore:

```
> cargo build

error[E0308]: mismatched types
  --> src/fifth.rs:15:34
   |
15 |         List { head: None, tail: None }
   |                                  ^^^^ expected *-ptr, found
   |                                       enum `std::option::Option`
   |
   = note: expected type `*mut fifth::Node<T>`
              found type `std::option::Option<_>`
```

We *could* use an Option, but unlike Box, `*mut` *is* nullable. This means it can't benefit from the null pointer optimization. Instead, we'll be using `null` to represent None.

So how do we get a null pointer? There's a few ways, but I prefer to use `std::ptr::null_mut()`. If you want, you can also use `0 as *mut _`, but that just seems so *messy*.

```
use std::ptr;

// defns...

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: None, tail: ptr::null_mut() }
    }
}
```

```
cargo build

warning: field is never used: `head`
 --> src/fifth.rs:4:5
  |
4 |     head: Link<T>,
  |     ^^^^^^^^^^^^^
  |
  = note: #[warn(dead_code)] on by default

warning: field is never used: `tail`
 --> src/fifth.rs:5:5
  |
5 |     tail: *mut Node<T>,
  |     ^^^^^^^^^^^^^^^^^^

warning: field is never used: `elem`
  --> src/fifth.rs:11:5
   |
11 |     elem: T,
   |     ^^^^^^^

warning: field is never used: `head`
  --> src/fifth.rs:12:5
   |
12 |     head: Link<T>,
   |     ^^^^^^^^^^^^^
```

shush compiler, we will use them soon.

Alright, let's move on to writing `push` again. This time, instead of grabbing an `Option<&mut Node<T>>` after we insert, we're just going to grab a `*mut Node<T>` to the insides of the Box right away. We know we can soundly do this because the contents of a Box has a stable address, even if we move the Box around. Of course, this isn't *safe*, because if we just drop the Box we'll have a pointer to freed memory.

How do we make a raw pointer from a normal pointer? Coercions! If a variable is declared to be a raw pointer, a normal reference will coerce into it:

```
let raw_tail: *mut _ = &mut *new_tail;
```

We have all the info we need. We can translate our code into, approximately, the previous reference version:

```
pub fn push(&mut self, elem: T) {
    let mut new_tail = Box::new(Node {
        elem: elem,
        next: None,
    });

    let raw_tail: *mut _ = &mut *new_tail;

    // .is_null checks for null, equivalent to checking for None
    if !self.tail.is_null() {
        // If the old tail existed, update it to point to the new tail
        self.tail.next = Some(new_tail);
    } else {
        // Otherwise, update the head to point to it
        self.head = Some(new_tail);
    }

    self.tail = raw_tail;
}
```

```
> cargo build

error[E0609]: no field `next` on type `*mut fifth::Node<T>`
  --> src/fifth.rs:31:23
   |
31 |             self.tail.next = Some(new_tail);
   |             ----------^^^^
   |             |
   |             help: `self.tail` is a raw pointer;
   |             try dereferencing it: `(*self.tail).next`
```

Huh? We have a pointer to a Node, why can't we get the `next` field?

Rust is kinda a jerk when you use raw pointers. To access the contents of a raw pointer, it insists that we manually deref them, because it's such an unsafe operation. So let's do that:

```
*self.tail.next = Some(new_tail);
```

```
> cargo build

error[E0609]: no field `next` on type `*mut fifth::Node<T>`
  --> src/fifth.rs:31:23
   |
31 |             *self.tail.next = Some(new_tail);
   |             -----------^^^^
   |             |
   |             help: `self.tail` is a raw pointer;
   |             try dereferencing it: `(*self.tail).next`
```

Uuuugh operator precedence.

```
(*self.tail).next = Some(new_tail);
```

```
> cargo build

error[E0133]: dereference of raw pointer is unsafe and requires
              unsafe function or block

  --> src/fifth.rs:31:13
   |
31 |             (*self.tail).next = Some(new_tail);
   |             ^^^^^^^^^^^^^^^^^ dereference of raw pointer
   |
   = note: raw pointers may be NULL, dangling or unaligned;
     they can violate aliasing rules and cause data races:
     all of these are undefined behavior
```

THIS. SHOULDN'T. BE. THIS. HARD.

Remember how I said Unsafe Rust is like an FFI language for Safe Rust? Well, the compiler wants us to explicitly delimit where we're doing this FFI-ing. We have two options. First, we can mark our *entire* function as unsafe, in which case it becomes an Unsafe Rust function and can only be called in an `unsafe` context. This isn't great, because we want our list to be safe to use. Second, we can write an `unsafe` block inside our function, to delimit the FFI boundary. This declares the overall function to be safe. Let's do that one:

```
pub fn push(&mut self, elem: T) {
    let mut new_tail = Box::new(Node {
        elem: elem,
        next: None,
    });

    let raw_tail: *mut _ = &mut *new_tail;

    if !self.tail.is_null() {
        // Hello Compiler, I Know I Am Doing Something Dangerous And
        // I Promise To Be A Good Programmer Who Never Makes Mistakes.
        unsafe {
            (*self.tail).next = Some(new_tail);
        }
    } else {
        self.head = Some(new_tail);
    }

    self.tail = raw_tail;
}
```

```
> cargo build
warning: field is never used: `elem`
  --> src/fifth.rs:11:5
   |
11 |     elem: T,
   |     ^^^^^^^
   |
   = note: #[warn(dead_code)] on by default
```

Yay!

It's kind've interesting that that's the *only* place we've had to write an unsafe block so far. We do raw pointer stuff all over the place, what's up with that?

It turns out that Rust is a massive rules-lawyer pedant when it comes to `unsafe`. We quite reasonably want to maximize the set of Safe Rust programs, because those are programs we can be much more confident in. To accomplish this, Rust carefully carves out a minimal surface area for

unsafety. Note that all the other places we've worked with raw pointers has been *assigning* them, or just observing whether they're null or not.

If you never actually dereference a raw pointer *those are totally safe things to do*. You're just reading and writing an integer! The only time you can actually get into trouble with a raw pointer is if you actually dereference it. So Rust says *only* that operation is unsafe, and everything else is totally safe.

Super. Pedantic. But technically correct.

> **NARRATOR:** Somewhere on the other side of the world, a hardware engineer feels a shiver down her spine — someone must be insisting pointers are just integers again. She looks down at her proposal for a new hardware pointer authentication scheme and sheds a single tear. The compiler engineer next door feels nothing — they long ago learned to always wear a heavy sweater.

Having only some of the pointer operations be *actually* unsafe raises an interesting problem: although we're supposed to delimit the scope of the unsafety with the `unsafe` block, it actually depends on state that was established outside of the block. Outside of the function, even!

This is what I call unsafe *taint*. As soon as you use `unsafe` in a module, that whole module is tainted with unsafety. Everything has to be correctly written in order to make sure all invariants are upheld for the unsafe code.

This taint is manageable because of *privacy*. Outside of our module, all of our struct fields are totally private, so no one else can mess with our state in arbitrary ways. As long as no combination of the APIs we expose causes bad stuff to happen, as far as an outside observer is concerned, all of our code is safe! And really, this is no different from the FFI case. No one needs to care if some python math library shells out to C as long as it exposes a safe interface.

Anyway, let's move on to `pop`, which is pretty much verbatim the reference version:

```
pub fn pop(&mut self) -> Option<T> {
    self.head.take().map(|head| {
        let head = *head;
        self.head = head.next;

        if self.head.is_none() {
            self.tail = ptr::null_mut();
        }

        head.elem
    })
}
```

Again we see another case where safety is stateful. If we fail to null out the tail pointer in *this* function, we'll see no problems at all. However subsequent calls to `push` will start writing to the dangling tail!

Let's test it out:

```
#[cfg(test)]
mod test {
    use super::List;
    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop(), None);

        // Populate list
        list.push(1);
        list.push(2);
        list.push(3);

        // Check normal removal
        assert_eq!(list.pop(), Some(1));
        assert_eq!(list.pop(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push(4);
        list.push(5);

        // Check normal removal
        assert_eq!(list.pop(), Some(3));
        assert_eq!(list.pop(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop(), Some(5));
        assert_eq!(list.pop(), None);

        // Check the exhaustion case fixed the pointer right
        list.push(6);
        list.push(7);

        // Check normal removal
        assert_eq!(list.pop(), Some(6));
        assert_eq!(list.pop(), Some(7));
        assert_eq!(list.pop(), None);
    }
}
```

This is just the stack test, but with the expected `pop` results flipped around. I also added some extra steps at the end to make sure that tail-pointer corruption case in `pop` doesn't occur.

```
cargo test

running 12 tests
test fifth::test::basics ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::peek ... ok
test second::test::basics ... ok
test fourth::test::into_iter ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::basics ... ok
test third::test::iter ... ok

test result: ok. 12 passed; 0 failed; 0 ignored; 0 measured
```

Gold Star!

NARRATOR: Here it comes...

Miri

nervously laughs This unsafe stuff is so easy, I don't know why everyone says otherwise. Our program works perfectly.

NARRATOR: 🙂

...right?

NARRATOR: 🙂

Well, we're writing `unsafe` code now, so the compiler can't help us catch mistakes as well. It could be that the tests *happened* to work, but were actually doing something non-deterministic. Something Undefined Behavioury.

But what can we do? We've pried open the windows and snuck out of rustc's classroom. No one can help us now.

...Wait, who's that sketchy looking person in the alleyway?

"Hey kid, you wanna interpret some Rust code?"

Wh- no? Why,

"It's wild man, it can validate that the actual dynamic execution of your program conforms to the semantics of Rust's memory model. Blows your mind..."

What?

"It checks if you Do An Undefined Behaviour."

I guess I could try interpretters just *once*.

"You've got rustup installed right?"

Of course I do, it's *the* tool for having an up to date Rust toolchain!

```
> rustup +nightly-2022-01-21 component add miri

info: syncing channel updates for 'nightly-2022-01-21-x86_64-pc-windows-msvc'
info: latest update on 2022-01-21, rust version 1.60.0-nightly (777bb86bc 2022-01-20)
info: downloading component 'cargo'
info: downloading component 'clippy'
info: downloading component 'rust-docs'
info: downloading component 'rust-std'
info: downloading component 'rustc'
info: downloading component 'rustfmt'
info: installing component 'cargo'
info: installing component 'clippy'
info: installing component 'rust-docs'
info: installing component 'rust-std'
info: installing component 'rustc'
info: installing component 'rustfmt'
info: downloading component 'miri'
info: installing component 'miri'
```

What did you just install on my computer!?

"The Good Stuff"

NARRATOR: Some weird stuff going on with toolchain versions:

The tool we're installing, `miri`, works closely with rustc's internals, so it's only available for nightly toolchains.

`+nightly-2022-01-21` tells `rustup` we want to install miri with the rust nightly toolchain for that date. I'm giving a specific date because sometimes miri falls behind and can't be built for a few nightlies. rustup will automatically download whatever toolchain we specify with `+` if we don't have it installed yet.

2022-01-21 is just a nightly I know has miri support, which you can check on this status page. You can just use `+nightly` if you're feeling lucky.

Whenever we invoke miri via `cargo miri` we will also use this `+` syntax to specify the toolchain we installed miri on. If you don't want to have to specify it every time, you can use `rustup override set`.

```
> cargo +nightly-2022-01-21 miri test

I will run `"cargo.exe" "install" "xargo"` to install
a recent enough xargo. Proceed? [Y/n]
```

UH WHAT ON EARTH IS XARGO?

"It's fine, don't worry about it."

```
> y

    Updating crates.io index
  Installing xargo v0.3.24
...
    Finished release [optimized] target(s) in 10.65s
  Installing C:\Users\ninte\.cargo\bin\xargo-check.exe
  Installing C:\Users\ninte\.cargo\bin\xargo.exe
   Installed package `xargo v0.3.24` (executables `xargo-check.exe`, `xargo.exe`)

I will run `"rustup" "component" "add" "rust-src"` to install
the `rust-src` component for the selected toolchain. Proceed? [Y/n]
```

UH???

"Who doesn't love having a copy of Rust's source code?"

```
> y

info: downloading component 'rust-src'
info: installing component 'rust-src'
```

"Aw yeah it's ready, here's the good part."

```
   Compiling lists v0.1.0 (C:\Users\ninte\dev\tmp\lists)
    Finished test [unoptimized + debuginfo] target(s) in 0.25s
     Running unittests (lists-5cc11d9ee5c3e924.exe)

error: Undefined Behavior: trying to reborrow for Unique at alloc84055,
       but parent tag <209678> does not have an appropriate item in
       the borrow stack

   --> \lib\rustlib\src\rust\library\core\src\option.rs:846:18
    |
846 |             Some(x) => Some(f(x)),
    |                  ^ trying to reborrow for Unique at alloc84055,
    |                    but parent tag <209678> does not have an
    |                    appropriate item in the borrow stack
    |
    = help: this indicates a potential bug in the program:
      it performed an invalid operation, but the rules it
      violated are still experimental
    = help: see https://github.com/rust-lang/unsafe-code-
guidelines/blob/master/wip/stacked-borrows.md
      for further information

    = note: inside `std::option::Option::<std::boxed::Box<fifth::Node<i32>>>::map::
<i32, [closure@src\fifth.rs:31:30: 40:10]>` at
\lib\rustlib\src\rust\library\core\src\option.rs:846:18

note: inside `fifth::List::<i32>::pop` at src\fifth.rs:31:9
   --> src\fifth.rs:31:9
    |
31  | /         self.head.take().map(|head| {
32  | |             let head = *head;
33  | |             self.head = head.next;
34  | |
...   |
39  | |             head.elem
40  | |         })
    | |__________^
note: inside `fifth::test::basics` at src\fifth.rs:74:20
   --> src\fifth.rs:74:20
    |
74  |         assert_eq!(list.pop(), Some(1));
    |                    ^^^^^^^^^^
note: inside closure at src\fifth.rs:62:5
   --> src\fifth.rs:62:5
    |
61  |       #[test]
    |       ------- in this procedural macro expansion
62  | /     fn basics() {
63  | |         let mut list = List::new();
64  | |
65  | |         // Check empty list behaves right
...   |
96  | |         assert_eq!(list.pop(), None);
97  | |     }
    | |_____^
 ...
error: aborting due to previous error
```

Woah. That's one heck of an error.

"Yeah, look at that shit. You love to see it."

Thank you?

"Here take the bottle of estradiol too, you're gonna need it later."

Wait why?

"You're about to think about memory models, trust me."

> **NARRATOR:** The mysterious person then proceeded to transform into a fox and scampered through a hole in the wall. The author then stared into the middle distance for several minutes while they tried to process everything that just happened.

The mysterious fox in the alleyway was right about more than just my gender: miri really is The Good Shit.

Ok so what *is* miri?

> An experimental interpreter for Rust's mid-level intermediate representation (MIR). It can run binaries and test suites of cargo projects and detect certain classes of undefined behavior, for example:
>
> - Out-of-bounds memory accesses and use-after-free
> - Invalid use of uninitialized data
> - Violation of intrinsic preconditions (an unreachable_unchecked being reached, calling copy_nonoverlapping with overlapping ranges, ...)
> - Not sufficiently aligned memory accesses and references
> - Violation of some basic type invariants (a bool that is not 0 or 1, for example, or an invalid enum discriminant)
> - Experimental: Violations of the Stacked Borrows rules governing aliasing for reference types
> - Experimental: Data races (but no weak memory effects)
>
> On top of that, Miri will also tell you about memory leaks: when there is memory still allocated at the end of the execution, and that memory is not reachable from a global static, Miri will raise an error.
>
> ...
>
> However, be aware that Miri will not catch all cases of undefined behavior in your program, and cannot run all programs

---

TL;DR: it interprets your program and notices if you break the rules *at runtime* and Do An Undefined Behaviour. This is necessary because Undefined Behaviour is *generally* a thing that happens at runtime. If the issue could be found at compile time, the compiler would just make it an error!

If you're familiar with tools like ubsan and tsan: it's basically that but all together and more extreme.

Miri is now hanging outside the classroom window with a knife. A learning knife.

If we ever want miri to check our work, we can ask them to interpret our test suite with

```
> cargo +nightly-2022-01-21 miri test
```

Now let's take a closer look at what they carved into our desk:

```
error: Undefined Behavior: trying to reborrow for Unique at alloc84055, but parent tag
<209678> does not have an appropriate item in the borrow stack

   --> \lib\rustlib\src\rust\library\core\src\option.rs:846:18
    |
846 |             Some(x) => Some(f(x)),
    |                  ^ trying to reborrow for Unique at alloc84055,
    |                    but parent tag <209678> does not have an
    |                    appropriate item in the borrow stack
    |

    = help: this indicates a potential bug in the program: it
      performed an invalid operation, but the rules it
      violated are still experimental

    = help: see
      https://github.com/rust-lang/unsafe-code-guidelines/blob/master/wip/stacked-
borrows.md
      for further information
```

Well I can see we made an error, but that's a confusing error message. What's the "borrow stack"?

We'll try to figure that out in the next section.

Attempting To Understand Stacked Borrows

In the previous section we tried running our unsafe singly-linked queue under miri, and it said we had broken the rules of *stacked borrows*, and linked us some documentation.

Normally I'd give a guided tour of the docs, but we're not really the target audience of that documentation. It's more designed for compiler developers and academics who are working on the semantics of Rust.

So I'm going to just give you the high level *idea* of "stacked borrows", and then give you a simple strategy for following the rules.

> **NARRATOR:** Stacked borrows are still "experimental" as a semantic model for Rust, so breaking these rules may not actually mean your program is "wrong". But unless you literally work on the compiler, you should just fix your program when miri complains. Better safe than sorry when it comes to Undefined Behaviour.

The Motivation: Pointer Aliasing

Before we get into *what* rules we've broken, it will help to understand *why* the rules exist in the first place. There are a few different motivating problems, but I think the most important one is *pointer aliasing*.

We say two pointers *alias* when the pieces of memory they point to overlap. Just as someone who "goes by an alias" can be referred to by two different names, that overlapping piece of memory can be referred to by two different pointers. This can lead to problems.

The compiler uses information about pointer aliasing to optimize accesses to memory, so if the information it has is *wrong* then the program will be miscompiled and do random garbage.

> **NARRATOR:** Practically speaking, aliasing is more concerned with memory accesses than the pointers themselves, and only really matters when one of the accesses is mutating. Pointers are emphasized because they're a convenient thing to attach rules to.

To understand why pointer aliasing information is important, let's consider *The Parable of the Tiny Angry Man*.

Michiel was looking through their bookshelf one day when they saw a book they didn't remember. They pulled it from the bookcase and looked at the cover.

"Oh yes, my old copy of *War and Peace*, a book I definitely have read. I loved the part with all the Peace."

Suddenly there was a knock at the door. Michiel returned the book to its shelf and opened the door -- it was their sworn nemesis **Hamslaw**. As Hamslaw prepared a devastating remark about Michiel's clearly inferior codegolfing skills, they sensed an opening:

"Hey Hamslaw, have you ever read War and Peace?"

"Pfft, no one's *actually* read War and Peace."

"Well I have, look it's right there in my bookcase, which *obviously* means I've read it."

Hamslaw couldn't believe it. Her face shifted from its usual smug demeanor to an iron mask of rage and determination. Hamslaw pushed Michiel aside and power-walked to the book shelf, cleaving the tome from its resting place with the fury of a thousand Valkyries. She turned the ancient text over in her hands, and the instant she saw the cover she began to shake.

Michiel prepared to boast of their clearly unparalleled brilliance, but was interrupted by the sudden laughter of Hamslaw.

"This isn't War and Peace, this is War and *Feet*!"

Tears were rolling down Hamslaw's face. This was clearly the greatest moment of her life.

"N-no! I just looked at it!"

They grabbed the book from Hamslaw and checked the cover. Indeed, the word "Peace" had been scratched out and replaced with "Feet". Michiel was mortified. This was clearly the worst moment of their life.

They fell to their knees and stared blankly at the bookcase. How could this have happened? They had checked the cover only a moment ago!

And then they saw a bit of motion in the bookcase. It was a tiny man. A tiny many with the angriest scowl Michiel had ever seen. The tiny man flipped Michiel off and mouthed the words "no one will believe you" and disappeared back between the books.

Michiel's plan *had* been perfect, but they had failed to account for the possibility of a tiny angry man with a sharpie and the desire for destruction. They thought they knew what the cover of the book said, and they thought that no one could have possibly changed it. But alas, they were wrong.

Hamslaw was already working on a zine commemorating her incredible victory — Michiel's reputation at the local Internet Cafe would never recover.

No one wants to be like Michiel, but no one wants to live in constant fear of the tiny angry man either. We want to know when the tiny angry man could be playing tricks on us. When he is, we will be very careful and paranoid about checking everything before we use it. But when the tiny angry man is gone, we want to be able to remember things.

That's the (very simplified) crux of pointer aliasing: when can the compiler assume it's safe to "remember" (cache) values instead of loading them over and over? To know that, the compiler needs to know whenever there *could* be little angry men mutating the memory behind your back.

> **NARRATOR:** the compiler also uses this information to cache stores, which just means it can avoid committing things to memory if it thinks no one will notice. In this case the problem is still tiny angry men, but they only need to read the memory for it to be a problem.

Safe Stacked Borrows

Ok so we want the compiler to have good pointer aliasing information, can we do that? Well, seemingly Rust is *designed* for it. Mutable references aren't aliased by definition, and although shared references *can* alias eachother, they can't mutate. Perfect! Ship it!

Except it's more complicated than that. We can "reborrow" mutable pointers like this:

```
let mut data = 10;
let ref1 = &mut data;
let ref2 = &mut *ref1;

*ref2 += 2;
*ref1 += 1;

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

The compiles and runs fine. What's the deal?

Well we can see what's going on by swapping the two uses:

```
let mut data = 10;
let ref1 = &mut data;
let ref2 = &mut *ref1;

// ORDER SWAPPED!
*ref1 += 1;
*ref2 += 2;

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

```
error[E0503]: cannot use `*ref1` because it was mutably borrowed
 --> src/main.rs:6:5
  |
4 |     let ref2 = &mut *ref1;
  |                ---------- borrow of `*ref1` occurs here
5 |
6 |     *ref1 += 1;
  |     ^^^^^^^^^ use of borrowed `*ref1`
7 |     *ref2 += 2;
  |     ---------- borrow later used here

For more information about this error, try `rustc --explain E0503`.
error: could not compile `playground` due to previous error
```

It's suddenly a compiler error!

When we reborrow a mutable pointer, the original pointer can't be used anymore until the borrower is done with it (no more uses).

In the code that works, there's a nice little nesting of the uses. We reborrow the pointer, use the new pointer for a while, and then stop using it before using the older pointer again. In the code that *doesn't* work, that doesn't happen. We just interleave the uses arbitrarily.

This is how we can have reborrows and still have aliasing information: all of our reborrows clearly nest, so we can consider only one of them "live" at any given time.

Hey, you know what's a great way to represent cleanly nested things? A stack. A stack of borrows.

Oh hey it's *Stacked Borrows*!

Whatever's at the top of the borrow stack is "live" and knows it's effectively unaliased. When you reborrow a pointer, the new pointer is pushed onto the stack, becoming *the* live pointer. When you use an older pointer it's brought back to life by popping everything on the borrow stack above it. At this point the pointer "knows" it was reborrowed and that the memory might have been modified, but that it once more has exclusive access -- no need to worry about little angry men.

So it's actually *always* ok to access a reborrowed pointer, because we can always pop everything above it. The real trouble is accessing a pointer that has already been popped off of the borrow stack -- then you've messed up.

Thankfully the design of the borrowchecker ensures that safe Rust programs follow these rules, as we saw in the above example, but the compiler generally views this problem "backwards" from the stacked borrows perspective. Instead of saying using `ref1` invalidates `ref2`, it insists that `ref2` *must* be valid for all its uses, and that `ref1` is the one messing things up by going out of turn.

Hence "cannot use `*ref1` because it was mutably borrowed". It's the same result (especially with non-lexical lifetimes), but framed in a way that's probably more intuitive.

But the borrowchecker can't help us when we start using unsafe pointers!

Unsafe Stacked Borrows

So we want to somehow have a way for unsafe pointers to participate in this stacked borrows system, even though the compiler can't track them properly. And we also want the system to be fairly permissive so that it's not *too* easy to mess it up and cause UB.

That's a hard problem, and I don't know how to solve it, but the folks who worked on Stacked Borrows came up with something plausible, and miri tries to implement it.

The very high-level concept is that when you convert a reference (or any other safe pointer) into an raw pointer it's *basically* like taking a reborrow. So now the raw pointer is allowed to do whatever it wants with that memory, and when the reborrow expires it's just like when that happens with normal reborrows.

But the question is, when does that reborrow expire? Well, probably a good time to expire it is when you start using the original reference again. Otherwise things aren't a nice nested stack.

But wait, you can turn a raw pointer *into* a reference! And you can copy raw pointers! What if you go `&mut -> *mut -> &mut -> *mut` and then access the first `*mut`? How the heck do the stacked borrows work then?

I genuinely don't know! That's why things are complicated. In fact they're *extra* complicated because stacked borrows are *trying* to be more permissive and let more unsafe code work the way you'd expect it to. This is why I run things under miri to try to help me catch mistakes.

In fact, this messiness is why there is an extra-experimental extra-strict mode of miri: `-Zmiri-tag-raw-pointers`.

To enable it, we need to pass it via a MIRIFLAGS environment variable like this:

```
MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri test
```

Or like this on Windows, where you need to just set the variable globally:

```
$env:MIRIFLAGS="-Zmiri-tag-raw-pointers"
cargo +nightly-2022-01-21 miri test
```

We'll generally be trying to conform to this extra-strict mode just to be *extra* confident in our work. It's also in some sense "simpler", so it's actually better for messing around and getting an intuition for stacked borrows.

Managing Stacked Borrows

So when using raw pointers we're going to try to stick to a heuristic that's simple and blunt and will hopefully have a large margin of error:

Once you start using raw pointers, try to ONLY use raw pointers.

This makes it as unlikely as possible to accidentally lose the raw pointer's "permission" to access the memory.

NARRATOR: this is oversimplified in two regards:

1. Safe pointers often assert more properties than just aliasing: the memory is allocated, it's aligned, it's large enough to fit the type of the pointee, the pointee is properly initialized, etc. So it's even more dangerous to wildly throw them around when things are in a dubious state.

2. Even if you stay in raw pointer land, you can't just wildly alias any memory. Pointers are conceptually tied to specific "allocations" (which can be as granular as a local variable on the stack), and you're not supposed to take a pointer from one allocation, offset it, and

then access memory in a different allocation. If this was allowed, there would *always* be the threat of tiny angry men *everywhere*. This is part of the reason "pointers are just integers" is a *problematic* viewpoint.

Now, we still want safe references in our *interface*, because we want to build a nice *safe abstraction* so the user of our list doesn't have to know or worry about.

So what we're going to do is:

1. At the start of a method, use the input references to get our raw pointers
2. Do our best to only use unsafe pointers from this point on
3. Convert our raw pointers back to safe pointers at the end if needed

But the fields of our types are private so we're going to keep those *entirely* as raw pointers.

In fact, part of the big mistake we made was continuing to use Box! Box has a special annotation in it that tells the compiler "hey this is a lot like `&mut`, because it uniquely owns that pointer". Which is true!

But the raw pointer we were keeping to the end of the list was pointing into a Box, so whenever we access the Box normally we're probably invalidating that raw pointer's "reborrow"! ☠️

In the next section we'll return to our true form and hit our heads against a bunch of examples.

Testing Stacked Borrows

TL;DR of the previous section's (simplified) memory model for Rust:

- Rust conceptually handles reborrows by maintaining a "borrow stack"
- Only the one on the top of the stack is "live" (has exclusive access)
- When you access a lower one it becomes "live" and the ones above it get popped
- You're not allowed to use pointers that have been popped from the borrow stack
- The borrowchecker ensures safe code code obeys this
- Miri theoretically checks that raw pointers obey this at runtime

---

That was a lot of theory and ideas -- let's move on to the true heart and soul of this book: writing some bad code and getting our tools to scream at us. We're going to go through a *ton* of examples to try to see if our mental model makes sense, and to try to get an intuitive feel for stacked borrows.

NARRATOR: Catching Undefined Behaviour in practice is a hairy business. After all, you're dealing with situations that the compiler literally *assumes* don't happen.

If you're lucky, things will "seem to work" today, but they'll be a ticking time bomb for a Smarter Compiler or slight change to the code. If you're *really* lucky things will reliably crash so you can just catch the mistake and fix it. But if you're unlucky, then things will be broken in weird and baffling ways.

Miri tries to work around this by getting rustc's most naive and unoptimized view of the program and tracking extra state as it interprets. As far as "sanitizers" go, this is a fairly deterministic and robust approach but it will never be *perfect*. You need your test program to actually have an execution with that UB, and for a large enough program it's very easy to introduce all sorts of non-determinism (HashMaps use RNG by default!).

> We can never take miri approving of our program's execution as an absolute certain statement there's no UB. It's also possible for miri to *think* something's UB when it really isn't. But if we have a mental model of how things work, and miri seems to agree with us, that's a good sign that we're on the right track.

Basic Borrows

In previous sections we saw that the borrowchecker didn't like this code:

```
let mut data = 10;
let ref1 = &mut data;
let ref2 = &mut *ref1;

// ORDER SWAPPED!
*ref1 += 1;
*ref2 += 2;

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

Let's see what happens when we replace `ref2` with `*mut`:

```
unsafe {
    let mut data = 10;
    let ref1 = &mut data;
    let ptr2 = ref1 as *mut _;

    // ORDER SWAPPED!
    *ref1 += 1;
    *ptr2 += 2;

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

```
cargo run
   Compiling miri-sandbox v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 0.71s
     Running `target\debug\miri-sandbox.exe`
13
```

Rustc seems perfectly happy with this: no warnings and the program produced the result we expected! Now let's look at what miri (in strict mode) thinks of it:

```
MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run

    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running cargo-miri.exe target\miri

error: Undefined Behavior: no item granting read access
to tag <untagged> at alloc748 found in borrow stack.

 --> src\main.rs:9:9
  |
9 |         *ptr2 += 2;
  |         ^^^^^^^^^^ no item granting read access to tag <untagged>
  |                    at alloc748 found in borrow stack.
  |
  = help: this indicates a potential bug in the program:
    it performed an invalid operation, but the rules it
    violated are still experimental
```

Nice! Our intuitive model of how things work held up: although the compiler couldn't catch the issue for us, miri did.

Let's try something more complicated, the `&mut -> *mut -> &mut -> *mut` case we alluded to before:

```
unsafe {
    let mut data = 10;
    let ref1 = &mut data;
    let ptr2 = ref1 as *mut _;
    let ref3 = &mut *ptr2;
    let ptr4 = ref3 as *mut _;

    // Access the first raw pointer first
    *ptr2 += 2;

    // Then access things in "borrow stack" order
    *ptr4 += 4;
    *ref3 += 3;
    *ptr2 += 2;
    *ref1 += 1;

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

```
cargo run
22

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run

error: Undefined Behavior: no item granting read access
to tag <1621> at alloc748 found in borrow stack.

  --> src\main.rs:13:5
   |
13 |     *ptr4 += 4;
   |     ^^^^^^^^^^ no item granting read access to tag <1621>
   |                at alloc748 found in borrow stack.
   |
```

Wow yep! In strict mode miri can "tell apart" the two raw pointers and have using the second one invalidate the first one. Let's see if everything works when we remove the first use that messes everything up:

```
unsafe {
    let mut data = 10;
    let ref1 = &mut data;
    let ptr2 = ref1 as *mut _;
    let ref3 = &mut *ptr2;
    let ptr4 = ref3 as *mut _;

    // Access things in "borrow stack" order
    *ptr4 += 4;
    *ref3 += 3;
    *ptr2 += 2;
    *ref1 += 1;

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

```
cargo run
20

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
20
```

NICE.

Yeah I'm pretty sure at this point we can all get PhD's in programming language memory model design and implementation. Who even *needs* compilers, this stuff is *easy*.

NARRATOR: it was not, but I'm proud of you nonetheless.

Testing Arrays

Let's mess with some arrays and pointer offsets (`add` and `sub`). This should work, right?

```
unsafe {
    let mut data = [0; 10];
    let ref1_at_0 = &mut data[0];           // Reference to 0th element
    let ptr2_at_0 = ref1_at_0 as *mut i32;  // Ptr to 0th element
    let ptr3_at_1 = ptr2_at_0.add(1);       // Ptr to 1st element

    *ptr3_at_1 += 3;
    *ptr2_at_0 += 2;
    *ref1_at_0 += 1;

    // Should be [3, 3, 0, ...]
    println!("{:?}", &data[..]);
}
```

```
cargo run
[3, 3, 0, 0, 0, 0, 0, 0, 0, 0]

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run

error: Undefined Behavior: no item granting read access
to tag <1619> at alloc748+0x4 found in borrow stack.
 --> src\main.rs:8:5
  |
8 |     *ptr3_at_1 += 3;
  |     ^^^^^^^^^^^^^^^ no item granting read access to tag <1619>
  |                     at alloc748+0x4 found in borrow stack.
```

Rips up gradschool application

What happened? We're using the borrow stack perfectly fine! Does something weird happen when we go `ptr -> ptr`? What if we just copy the pointer so they all go to the same location:

```
unsafe {
    let mut data = [0; 10];
    let ref1_at_0 = &mut data[0];           // Reference to 0th element
    let ptr2_at_0 = ref1_at_0 as *mut i32;  // Ptr to 0th element
    let ptr3_at_0 = ptr2_at_0;              // Ptr to 0th element

    *ptr3_at_0 += 3;
    *ptr2_at_0 += 2;
    *ref1_at_0 += 1;

    // Should be [6, 0, 0, ...]
    println!("{:?}", &data[..]);
}
```

```
cargo run
[6, 0, 0, 0, 0, 0, 0, 0, 0, 0]

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
[6, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Nope, that works fine. Maybe we're getting lucky, let's just make a real big mess of pointers:

```
unsafe {
    let mut data = [0; 10];
    let ref1_at_0 = &mut data[0];            // Reference to 0th element
    let ptr2_at_0 = ref1_at_0 as *mut i32;   // Ptr to 0th element
    let ptr3_at_0 = ptr2_at_0;               // Ptr to 0th element
    let ptr4_at_0 = ptr2_at_0.add(0);        // Ptr to 0th element
    let ptr5_at_0 = ptr3_at_0.add(1).sub(1); // Ptr to 0th element

    // An absolute jumbled hash of ptr usages
    *ptr3_at_0 += 3;
    *ptr2_at_0 += 2;
    *ptr4_at_0 += 4;
    *ptr5_at_0 += 5;
    *ptr3_at_0 += 3;
    *ptr2_at_0 += 2;
    *ref1_at_0 += 1;

    // Should be [20, 0, 0, ...]
    println!("{:?}", &data[..]);
}
```

```
cargo run
[20, 0, 0, 0, 0, 0, 0, 0, 0, 0]

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
[20, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Nope! Miri is actually *way* more permissive when it comes to raw pointers that are derived from other raw pointers. They all share the same "borrow" (or miri calls it, a *tag*).

Once you start using raw pointers they can freely split into their own tiny angry men and mess with themselves. This is ok because the compiler understands that and won't optimize the reads and writes the same it does with references.

> **NARRATOR:** If the code is simple enough, the compiler can keep track of all the derived pointers and still optimize things where possible, but it's going to be a lot more brittle than the reasoning it can use for references.

So what's the *real* problem?

Even though `data` is one "allocation" (local variable), `ref1_at_0` is only borrowing the first element. Rust allows borrows to be broken up so that they only apply to particular parts of the allocation! Let's try it out:

```
unsafe {
    let mut data = [0; 10];
    let ref1_at_0 = &mut data[0];           // Reference to 0th element
    let ref2_at_1 = &mut data[1];           // Reference to 1th element
    let ptr3_at_0 = ref1_at_0 as *mut i32;  // Ptr to 0th element
    let ptr4_at_1 = ref2_at_1 as *mut i32;   // Ptr to 1th element

    *ptr4_at_1 += 4;
    *ptr3_at_0 += 3;
    *ref2_at_1 += 2;
    *ref1_at_0 += 1;

    // Should be [3, 3, 0, ...]
    println!("{:?}", &data[..]);
}
```

```
error[E0499]: cannot borrow `data[_]` as mutable more than once at a time
 --> src\main.rs:5:21
  |
4 |     let ref1_at_0 = &mut data[0];           // Reference to 0th element
  |                      ------------ first mutable borrow occurs here
5 |     let ref2_at_1 = &mut data[1];           // Reference to 1th element
  |                      ^^^^^^^^^^^^ second mutable borrow occurs here
6 |     let ptr3_at_0 = ref1_at_0 as *mut i32;  // Ptr to 0th element
  |                     --------- first borrow later used here
  |
  = help: consider using `.split_at_mut(position)` or similar method
    to obtain two mutable non-overlapping sub-slices
```

Shoot! Rust doesn't track array indices to prove these borrows are disjoint, but it does give us `split_at_mut` to break a slice into multiple parts in a way that is safe to assume works:

```
unsafe {
    let mut data = [0; 10];

    let slice1 = &mut data[..];
    let (slice2_at_0, slice3_at_1) = slice1.split_at_mut(1);

    let ref4_at_0 = &mut slice2_at_0[0];    // Reference to 0th element
    let ref5_at_1 = &mut slice3_at_1[0];    // Reference to 1th element
    let ptr6_at_0 = ref4_at_0 as *mut i32;  // Ptr to 0th element
    let ptr7_at_1 = ref5_at_1 as *mut i32;  // Ptr to 1th element

    *ptr7_at_1 += 7;
    *ptr6_at_0 += 6;
    *ref5_at_1 += 5;
    *ref4_at_0 += 4;

    // Should be [10, 12, 0, ...]
    println!("{:?}", &data[..]);
}
```

```
cargo run
[10, 12, 0, 0, 0, 0, 0, 0, 0, 0]

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
[10, 12, 0, 0, 0, 0, 0, 0, 0, 0]
```

Hey, that works! Slices properly tell the compiler and miri "hey I'm taking a huge loan on all of the memory in my range", so they know all of the elements can be mutated.

Also note that operations like `split_at_mut` being allowed tells us that borrows can be less of a *stack* and more of a *tree*, because we can break one big borrow into a bunch of disjoint smaller ones, and everything still works.

(I think in the actual stacked borrows model everything's still stacks because the stacks are conceptually tracking permissions for each byte of the program..?)

What if we *directly* turn a slice into a pointer? Will that pointer have access to the full slice?

```
unsafe {
    let mut data = [0; 10];

    let slice1_all = &mut data[..];         // Slice for the entire array
    let ptr2_all = slice1_all.as_mut_ptr(); // Pointer for the entire array

    let ptr3_at_0 = ptr2_all;               // Pointer to 0th elem (the same)
    let ptr4_at_1 = ptr2_all.add(1);        // Pointer to 1th elem
    let ref5_at_0 = &mut *ptr3_at_0;        // Reference to 0th elem
    let ref6_at_1 = &mut *ptr4_at_1;        // Reference to 1th elem

    *ref6_at_1 += 6;
    *ref5_at_0 += 5;
    *ptr4_at_1 += 4;
    *ptr3_at_0 += 3;

    // Just for fun, modify all the elements in a loop
    // (Could use any of the raw pointers for this, they share a borrow!)
    for idx in 0..10 {
        *ptr2_all.add(idx) += idx;
    }

    // Safe version of this same code for fun
    for (idx, elem_ref) in slice1_all.iter_mut().enumerate() {
        *elem_ref += idx;
    }

    // Should be [8, 12, 4, 6, 8, 10, 12, 14, 16, 18]
    println!("{:?}", &data[..]);
}
```

```
cargo run
[8, 12, 4, 6, 8, 10, 12, 14, 16, 18]

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
[8, 12, 4, 6, 8, 10, 12, 14, 16, 18]
```

Nice! Pointers aren't just integers: they have a range of memory associated with them, and with Rust we're allowed to narrow that range!

Testing Shared References

In all of these examples I have been very carefully only using mutable references and doing read-modify-write operations (`+=`) to keep things as simple as possible.

But Rust has shared references that are read-only and can be freely copied, how should those work? Well we've seen that raw pointers can be freely copied and we can handle that by saying they "share" a single borrow. Maybe we think of shared references the same way?

Let's test that out with a function that reads a value (`println!` can be a little magical with auto-ref/deref stuff, so I'm wrapping it in a function to make sure we're testing what we want to be):

```
fn opaque_read(val: &i32) {
    println!("{}", val);
}

unsafe {
    let mut data = 10;
    let mref1 = &mut data;
    let sref2 = &mref1;
    let sref3 = sref2;
    let sref4 = &*sref2;

    // Random hash of shared reference reads
    opaque_read(sref3);
    opaque_read(sref2);
    opaque_read(sref4);
    opaque_read(sref2);
    opaque_read(sref3);

    *mref1 += 1;

    opaque_read(&data);
}
```

```
cargo run

warning: unnecessary `unsafe` block
 --> src\main.rs:6:1
  |
6 | unsafe {
  | ^^^^^^ unnecessary `unsafe` block
  |
  = note: `#[warn(unused_unsafe)]` on by default

warning: `miri-sandbox` (bin "miri-sandbox") generated 1 warning

10
10
10
10
10
11
```

Oh yeah we forgot to do anything with raw pointers, but at least we can see that it's fine for all the shared references to be used interchangeably. Now let's mix in some raw pointers:

```
fn opaque_read(val: &i32) {
    println!("{}", val);
}

unsafe {
    let mut data = 10;
    let mref1 = &mut data;
    let ptr2 = mref1 as *mut i32;
    let sref3 = &mref1;
    let ptr4 = sref3 as *mut i32;

    *ptr4 += 4;
    opaque_read(sref3);
    *ptr2 += 2;
    *mref1 += 1;

    opaque_read(&data);
}
```

```
cargo run

error[E0606]: casting `&&mut i32` as `*mut i32` is invalid
  --> src\main.rs:11:16
   |
11 |     let ptr4 = sref3 as *mut i32;
   |                ^^^^^^^^^^^^^^^^
```

Oh whoops, we were actually messing around with `& &mut` instead of `&`! Rust is very good at papering over that when it doesn't matter. Let's properly reborrow it with `let sref3 = &*mref1`:

```
cargo run

error[E0606]: casting `&i32` as `*mut i32` is invalid
  --> src\main.rs:11:16
   |
11 |     let ptr4 = sref3 as *mut i32;
   |                ^^^^^^^^^^^^^^^^
```

Nope, Rust still doesn't like that! You can only cast a shared reference to a `*const` which can only read. But what if we just... do... this...?

```
    let ptr4 = sref3 as *const i32 as *mut i32;
```

```
cargo run

14
17
```

WHAT. OK SURE FINE? Great cast system there Rust. It's almost like the `*const` is a pretty useless type that only really exists to describe C APIs and to vaguely suggest correct usage (it is, it does). What does miri think?

```
MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run

error: Undefined Behavior: no item granting write access to
tag <1621> at alloc742 found in borrow stack.
  --> src\main.rs:13:5
   |
13 |     *ptr4 += 4;
   |     ^^^^^^^^^^ no item granting write access to tag <1621>
   |                at alloc742 found in borrow stack.
```

Alas, though we can get around the compiler complaining with a double cast, it doesn't actually make this operation *allowed*. When we take the shared reference, we're promising not to modify the value.

This is important because that means when the shared borrow is popped off the borrow stack, the mutable pointers below it *can* assume the memory hasn't changed. There may have been some tiny angry men *reading* the memory (so writes had to be comitted) but they weren't able to modify it and the mutable pointers can assume the last value they wrote is still there!

Once a shared reference is on the borrow-stack, everything that gets pushed on top of it only has read permissions.

We can however do this:

```
fn opaque_read(val: &i32) {
    println!("{}", val);
}

unsafe {
    let mut data = 10;
    let mref1 = &mut data;
    let ptr2 = mref1 as *mut i32;
    let sref3 = &*mref1;
    let ptr4 = sref3 as *const i32 as *mut i32;

    opaque_read(&*ptr4);
    opaque_read(sref3);
    *ptr2 += 2;
    *mref1 += 1;

    opaque_read(&data);
}
```

Note how it was still "fine" to create a mutable raw pointer as long as we only actually read from it!

```
cargo run
10
10
13

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
10
10
13
```

And just to be sure, let's check that a shared reference gets popped like normal:

```
fn opaque_read(val: &i32) {
    println!("{}", val);
}

unsafe {
    let mut data = 10;
    let mref1 = &mut data;
    let ptr2 = mref1 as *mut i32;
    let sref3 = &*mref1;

    *ptr2 += 2;
    opaque_read(sref3); // Read in the wrong order?
    *mref1 += 1;

    opaque_read(&data);
}
```

```
cargo run
12
13

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run

error: Undefined Behavior: trying to reborrow for SharedReadOnly
at alloc742, but parent tag <1620> does not have an appropriate
item in the borrow stack

  --> src\main.rs:13:17
   |
13 |     opaque_read(sref3); // Read in the wrong order?
   |                 ^^^^^ trying to reborrow for SharedReadOnly
   |                       at alloc742, but parent tag <1620>
   |                       does not have an appropriate item
   |                       in the borrow stack
   |
```

Hey, we even got a slightly different error message about SharedReadOnly instead of some specific tag. That makes sense: once there's *any* shared references, basically everything else is just a big SharedReadOnly soup so there's no need to distinguish any of them!

Testing Interior Mutability

Remember that really horrible chapter of the book where we tried to make a linked list with RefCell and Rc and everything was even worse than usual when trying to write this godforsaken linked lists?

We've been insisting shared references can't be used for mutation but that chapter was all about how you could actually mutate through shared references with *interior mutability*. Let's try the nice and simple std::cell::Cell type:

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

unsafe {
    let mut data = Cell::new(10);
    let mref1 = &mut data;
    let ptr2 = mref1 as *mut Cell<i32>;
    let sref3 = &*mref1;

    sref3.set(sref3.get() + 3);
    (*ptr2).set((*ptr2).get() + 2);
    mref1.set(mref1.get() + 1);

    println!("{}", data.get());
}
```

Ah, such a beautiful mess. It will be lovely to see miri spit on it.

```
cargo run
16

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
16
```

Wait, really? *That's* fine? Why? How? What even is a *Cell*?

Smashes the padlock on the stdlib

```
pub struct Cell<T: ?Sized> {
    value: UnsafeCell<T>,
}
```

What the heck is `UnsafeCell`?

Smashes another padlock just to really show the stdlib we mean business

```
#[lang = "unsafe_cell"]
#[repr(transparent)]
#[repr(no_niche)]
pub struct UnsafeCell<T: ?Sized> {
    value: T,
}
```

Oh it's wizard magic. Ok. I guess. `#[lang = "unsafe_cell"]` is literally just saying UnsafeCell is UnsafeCell. Let's stop breaking locks and check the actual documentation of std::cell::UnsafeCell.

> The core primitive for interior mutability in Rust.
>
> If you have a reference `&T`, then normally in Rust the compiler performs optimizations based on the knowledge that `&T` points to immutable data. Mutating that data, for example through an alias or by transmuting an `&T` into an `&mut T`, is considered undefined behavior. `UnsafeCell<T>` opts-out of the immutability guarantee for `&T`: a shared reference `&UnsafeCell<T>` may point to data that is being mutated. This is called “interior mutability”.

Oh it *really is* just wizard magic.

UnsafeCell basically tells the compiler "hey listen, we're gonna get goofy with this memory, don't make any of the usual aliasing assumptions about it". Like putting up a big "CAUTION: TINY ANGRY MEN CROSSING" sign.

Let's see how adding UnsafeCell makes miri happy:

```
use std::cell::UnsafeCell;

fn opaque_read(val: &i32) {
    println!("{}", val);
}

unsafe {
    let mut data = UnsafeCell::new(10);
    let mref1 = data.get_mut();      // Get a mutable ref to the contents
    let ptr2 = mref1 as *mut i32;
    let sref3 = &*ptr2;

    *ptr2 += 2;
    opaque_read(sref3);
    *mref1 += 1;

    println!("{}", *data.get());
}
```

```
cargo run
12
13

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run

error: Undefined Behavior: trying to reborrow for SharedReadOnly
at alloc748, but parent tag <1629> does not have an appropriate
item in the borrow stack

  --> src\main.rs:15:17
   |
15 |     opaque_read(sref3);
   |                 ^^^^^ trying to reborrow for SharedReadOnly
   |                       at alloc748, but parent tag <1629> does
   |                       not have an appropriate item in the
   |                       borrow stack
   |
```

Wait, what? We spoke the magic words! What am I going to do with all this federally approved ritual-enhancing goat blood?

Well, we did, but then we completely discarded the spell by using `get_mut` which peeks inside the UnsafeCell and makes a proper `&mut i32` to it anyway!

Think about it: if the compiler had to assume `&mut i32` *could* be looking inside an `UnsafeCell`, then it would never be able to make any assumptions about aliasing at all! Everything could be full of tiny angry men.

So what we need to do is keep the `UnsafeCell` in our pointer types so that the compiler understands what we're doing.

```
use std::cell::UnsafeCell;

fn opaque_read(val: &i32) {
    println!("{}", val);
}

unsafe {
    let mut data = UnsafeCell::new(10);
    let mref1 = &mut data;              // Mutable ref to the *outside*
    let ptr2 = mref1.get();             // Get a raw pointer to the insides
    let sref3 = &*mref1;                // Get a shared ref to the *outside*

    *ptr2 += 2;                         // Mutate with the raw pointer
    opaque_read(&*sref3.get());         // Read from the shared ref
    *sref3.get() += 3;                  // Write through the shared ref
    *mref1.get() += 1;                  // Mutate with the mutable ref

    println!("{}", *data.get());
}
```

```
cargo run
12
16

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
12
16
```

It works! I won't have to throw out all this blood after all.

Actually, hey wait. We're still being a bit goofy with the order here. We made ptr2 first, and then made sref3 from the mutable pointer. And then we used the raw pointer before the shared pointer. That all seems... wrong.

Actually wait we did that with the Cell example too. HMMM.

We're forced to conclude one of two things:

- Miri is imperfect and this is actually still UB.
- Our simplified model is in fact an oversimplication.

I'd put my money on the second one, but just to be safe let's make a version that's definitely airtight in our simplified model of stacked borrows:

```
use std::cell::UnsafeCell;

fn opaque_read(val: &i32) {
    println!("{}", val);
}

unsafe {
    let mut data = UnsafeCell::new(10);
    let mref1 = &mut data;
    // These two are swapped so the borrows are *definitely* totally stacked
    let sref2 = &*mref1;
    // Derive the ptr from the shared ref to be super safe!
    let ptr3 = sref2.get();

    *ptr3 += 3;
    opaque_read(&*sref2.get());
    *sref2.get() += 2;
    *mref1.get() += 1;

    println!("{}", *data.get());
}
```

```
cargo run
13
16

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
13
16
```

Now, one reason why the first implementation we had *might* actually be correct is because if you *really* think about it `&UnsafeCell<T>` really is no different from `*mut T` as far as aliasing is concerned. You can infinitely copy it and mutate through it!

So in some sense we just created two raw pointers and used them interchangeably like normal. It's *a little* sketchy that both were derived from the mutable reference, so maybe the second one's creation should still pop the first one off the borrow stack, but that's not really necessary since we're not *actually* accessing the contents of the mutable reference, just copying its address.

A line like `let sref2 = &*mref1` is a tricksy thing. *Syntactically* it looks like we're dereferencing it, but dereferencing on it's own isn't actually a *thing*? Consider `&my_tuple.0`: you aren't actually doing anything to `my_tuple` or `.0`, you're just using them to refer to a location in memory and putting `&` in front of it that says "don't load this, just write the address down".

`&*` is the same thing: the `*` is just saying "hey let's talk about the location this pointer points to" and the `&` is just saying "now write that address down". Which is of course the same value the original pointer had. But the type of the pointer has changed, because, uh, types!

That said, if you do `&**` then you are in fact loading a value with the first `*`! `*` is weird!

NARRATOR: No one cares that you know the word "lvalue", *Jonathan*. In Rust we call them *places*, which is totally different and *so* much cooler?

Testing Box

Hey remember why we started this extremely long aside? You don't? Weird.

Well it was because we mixed Box and raw pointers. Box is *kind of* like `&mut`, because it claims unique ownership of the memory it points to. Let's test that claim out:

```
unsafe {
    let mut data = Box::new(10);
    let ptr1 = (&mut *data) as *mut i32;

    *data += 10;
    *ptr1 += 1;

    // Should be 21
    println!("{}", data);
}
```

```
cargo run
21

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run

error: Undefined Behavior: no item granting read access
       to tag <1707> at alloc763 found in borrow stack.

 --> src\main.rs:7:5
  |
7 |     *ptr1 += 1;
  |     ^^^^^^^^^^ no item granting read access to tag <1707>
  |                at alloc763 found in borrow stack.
  |
```

Yep, miri hates that. Let's check that doing things in the right order is ok:

```
unsafe {
    let mut data = Box::new(10);
    let ptr1 = (&mut *data) as *mut i32;

    *ptr1 += 1;
    *data += 10;

    // Should be 21
    println!("{}", data);
}
```

```
cargo run
21

MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri run
21
```

Yep!

Whelp that's all folks, we're finally done talking and thinking about stacked borrows!

...wait how do we solve this problem with Box? Like, sure we can write toy programs like this but we need to store the Box somewhere and hold onto our raw pointers for a potentially long time. Surely stuff will get mixed up and invalidated?

Great question! To answer that we'll finally be returning to our true calling: writing some god damn linked lists.

Wait, I need to write linked lists again? Let's not be hasty folks. Be reasonable. Just hold on I'm sure there's some other interesting issues for me to discu—

Layout and Basics 2: Getting Raw

> TL;DR on the previous three sections: randomly mixing safe pointers like `&`, `&mut`, and `Box` with unsafe pointers like `*mut` and `*const` is a recipe for Undefined Behaviour because the safe pointers introduce extra constraints that we aren't obeying with the raw pointers.

Oh god I need to write linked lists again. Fine. FINE. It's Fine. We're fine.

We're gonna knock a lot of this section out real quick since we already discussed the design in the first try around, and everything we did *was* basically correct except for how we mixed together safe and unsafe pointers.

Layout

So in the new layout we're only going to only use raw pointers and everything will be perfect and we'll never make mistakes again.

Here's our old broken layout:

```
pub struct List<T> {
    head: Link<T>,
    tail: *mut Node<T>, // INNOCENT AND KIND
}

type Link<T> = Option<Box<Node<T>>>; // THE REAL EVIL

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

And here's our new layout:

```
pub struct List<T> {
    head: Link<T>,
    tail: *mut Node<T>,
}

type Link<T> = *mut Node<T>; // MUCH BETTER

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

Remember: Option isn't as nice or useful when we're using raw pointers, so we're not using that anymore. In later sections we'll look at the `NonNull` type, but don't worry about that for now.

Basics

List::new is basically the same.

```
use ptr;

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: ptr::null_mut(), tail: ptr::null_mut() }
    }
}
```

Push is basically the s-

```
pub fn push(&mut self, elem: T) {
    let mut new_tail = Box::new(
```

Wait we're not using Box anymore. How do we allocate memory without Box?

Well, we *could* with `std::alloc::alloc`, but that's like bringing a katana into the kitchen. It'll get the job done but it's kinda overkill and unwieldy.

We want to *have* boxes, but, *not*. One completely wild but *maybe* viable option would be to do something like this:

```
struct Node<T> {
    elem: T,
    real_next: Option<Box<Node<T>>>,
    next: *mut Node<T>,
}
```

With the idea that we create the Boxes and store them in our node, but then we take a raw pointer into them and only use that raw pointer until we're done with the Node and want to destroy it. Then we can `take` the Box out of `real_next` and drop it. I *think* that would conform to our very simplified stacked borrows model?

If you wanna try to make that, have "fun", but that just looks awful right? This isn't the chapter on Rc and RefCell, we're not gonna play this *game* anymore. We're gonna just make simple and clean stuff.

So instead we're going to use the very nice Box::into_raw function:

```
pub fn into_raw(b: Box<T>) -> *mut T
```

Consumes the Box, returning a wrapped raw pointer.

The pointer will be properly aligned and non-null.

After calling this function, the caller is responsible for the memory previously managed by the Box. In particular, the caller should properly destroy T and release the memory, taking into account the memory layout used by Box. The easiest way to do this is to convert the raw pointer back into a Box with the `Box::from_raw` function, allowing the Box destructor to perform the cleanup.

Note: this is an associated function, which means that you have to call it as `Box::into_raw(b)` instead of `b.into_raw()`. This is so that there is no conflict with a method on the inner type.

Examples

Converting the raw pointer back into a Box with Box::from_raw for automatic cleanup:

```
let x = Box::new(String::from("Hello"));
let ptr = Box::into_raw(x);
let x = unsafe { Box::from_raw(ptr) };
```

Nice, that looks *literally* designed for our use case. It also matches the rules we're trying to follow: start with safe stuff, turn into into raw pointers, and then only convert back to safe stuff at the end (when we want to Drop it).

This is basically exactly like doing the weird `real_next` thing but without having to faff around storing the Box when it's the exact same pointer as the raw pointer anyway.

Also now that we're just using raw pointers everywhere, let's not worry about keeping those `unsafe` blocks narrow: it's all unsafe now. (It always was, but it's nice to lie to yourself sometimes.)

```
pub fn push(&mut self, elem: T) {
    unsafe {
        // Immediately convert the Box into a raw pointer
        let new_tail = Box::into_raw(Box::new(Node {
            elem: elem,
            next: ptr::null_mut(),
        }));

        if !self.tail.is_null() {
            (*self.tail).next = new_tail;
        } else {
            self.head = new_tail;
        }

        self.tail = new_tail;
    }
}
```

Hey that code's actually looking a lot cleaner now that we're sticking to raw pointers!

On to pop, which is also pretty similar to how we left it, although we've got to remember to use `Box::from_raw` to clean up the allocation:

```
pub fn pop(&mut self) -> Option<T> {
    unsafe {
        if self.head.is_null() {
            None
        } else {
            // RISE FROM THE GRAVE
            let head = Box::from_raw(self.head);
            self.head = head.next;

            if self.head.is_null() {
                self.tail = ptr::null_mut();
            }

            Some(head.elem)
        }
    }
}
```

Our nice little `take`s and `map`s are dead, gotta just check and set `null` manually now.

And while we're here, let's slap in the destructor. This time we'll implement it as just repeatedly popping, because it's cute and simple:

```
impl<T> Drop for List<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() { }
    }
}
```

Now, for the moment of truth:

```
#[cfg(test)]
mod test {
    use super::List;
    #[test]
    fn basics() {
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop(), None);

        // Populate list
        list.push(1);
        list.push(2);
        list.push(3);

        // Check normal removal
        assert_eq!(list.pop(), Some(1));
        assert_eq!(list.pop(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push(4);
        list.push(5);

        // Check normal removal
        assert_eq!(list.pop(), Some(3));
        assert_eq!(list.pop(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop(), Some(5));
        assert_eq!(list.pop(), None);

        // Check the exhaustion case fixed the pointer right
        list.push(6);
        list.push(7);

        // Check normal removal
        assert_eq!(list.pop(), Some(6));
        assert_eq!(list.pop(), Some(7));
        assert_eq!(list.pop(), None);
    }
}
```

```
cargo test

running 12 tests
test fifth::test::basics ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::peek ... ok
test second::test::basics ... ok
test fourth::test::into_iter ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::basics ... ok
test third::test::iter ... ok

test result: ok. 12 passed; 0 failed; 0 ignored; 0 measured
```

Good, but does miri agree?

```
MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri test

running 12 tests
test fifth::test::basics ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::peek ... ok
test second::test::basics ... ok
test fourth::test::into_iter ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::basics ... ok
test third::test::iter ... ok

test result: ok. 12 passed; 0 failed; 0 ignored; 0 measured
```

EYYYY!!!!!

IT FRIGGIN WORKED!

PROBABLY!

FAILING TO FIND UNDEFINED BEHAVIOUR IS NOT A PROOF THAT IT ISN'T THERE WAITING TO CAUSE PROBLEMS BUT THERE IS A LIMIT TO HOW RIGOROUS I AM WILLING TO BE FOR A JOKE BOOK ABOUT LINKED LISTS SO WE'RE GONNA CALL THIS A 100% MACHINE VERIFIED PROOF AND ANYONE WHO SAYS OTHERWISE CAN SUCK MY COQ!

∴ QED □

Extra Junk

Now that `push` and `pop` are written, everything else is acutally exactly the same as the stack case, weirdly. Only operations that change the length of the list need to touch the tail pointer.

But of course, now that everything's unsafe pointers we need to rewrite the code to use those! And if we're going to be touching all the code, we might as well take the chance to make sure we aren't missing something.

But anyway, let's start copy-pasting code from the stack implementation:

```
// ...

pub struct IntoIter<T>(List<T>);

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

pub struct IterMut<'a, T> {
    next: Option<&'a mut Node<T>>,
}
```

IntoIter looks fine, but `Iter` and `IterMut` are breaking our simple rule of never using safe pointers in our types anymore. Let's be safe and change those to use raw pointers:

```
pub struct IntoIter<T>(List<T>);

pub struct Iter<'a, T> {
    next: *mut Node<T>,
}

pub struct IterMut<'a, T> {
    next: *mut Node<T>,
}

impl<T> List<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }

    pub fn iter(&self) -> Iter<'_, T> {
        Iter { next: self.head }
    }

    pub fn iter_mut(&mut self) -> IterMut<'_, T> {
        IterMut { next: self.head }
    }
}
```

Looks good!

```
error[E0392]: parameter `'a` is never used
  --> src\fifth.rs:17:17
   |
17 | pub struct Iter<'a, T> {
   |                 ^^ unused parameter
   |
   = help: consider removing `'a`, referring to it in a field,
     or using a marker such as `PhantomData`

error[E0392]: parameter `'a` is never used
  --> src\fifth.rs:21:20
   |
21 | pub struct IterMut<'a, T> {
   |                    ^^ unused parameter
   |
   = help: consider removing `'a`, referring to it in a field,
     or using a marker such as `PhantomData`
```

Doesn't look good! What's this PhantomData they're on about?

> Zero-sized type used to mark things that "act like" they own a `T`.
>
> Adding a `PhantomData<T>` field to your type tells the compiler that your type acts as though it stores a value of type `T`, even though it doesn't really. This information is used when computing certain safety properties.
>
> For a more in-depth explanation of how to use `PhantomData<T>`, please see the Nomicon.

Hey don't get hasty there, we're reading the book that *I* wrote. Not that other book that some huge *nerd* probably wrote! I bet if they write a data structure in there it's something lame like an Array Stack and *not* a Linked List.

> Unused lifetime parameters

> Perhaps the most common use case for PhantomData is a struct that has an unused lifetime parameter, typically as part of some unsafe code.

Ah so we're naming a lifetime in our type but not actually using it. We *could* go down the PhantomData path, but I want to save that for the doubly-linked list in the next chapter that will *really* need it.

We're in an interesting situation where we actually don't need PhantomData. *I think*. I'm just going to claim that and trust that it's true, and if miri yells at us at the end I'll concede the point and we'll do the PhantomData thing.

What we're actually going to do is put the references back in these Iterator types and be happy we get to use references in some places still. I think that's sound because there's still a kind of proper nesting when you use an iterator: you create the iterator, use safe references for a while, and then discard the iterator.

Only once the iterator is gone can you access the list and call things like `push` and `pop` which need to mess with the tail pointer and Boxes. Now, during the iteration we *are* going to be dereferencing a bunch of raw pointers, so there is a kind of mixing there, but we should be able to think of those references as reborrows of the unsafe pointers.

I'm not even 100% convinced but I just wanna give it a try and see!

```
pub struct IntoIter<T>(List<T>);

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

pub struct IterMut<'a, T> {
    next: Option<&'a mut Node<T>>,
}

impl<T> List<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }

    pub fn iter(&self) -> Iter<'_, T> {
        unsafe {
            Iter { next: self.head.as_ref() }
        }
    }

    pub fn iter_mut(&mut self) -> IterMut<'_, T> {
        unsafe {
            IterMut { next: self.head.as_mut() }
        }
    }
}
```

If we're going to be storing references, we need to upgrade our raw pointers to options-of-references. We *could* check if the pointer is null, but this is one of the incredibly narrow cases where I *think* it's ok to use the nasty ptr::as_ref and ptr::as_mut methods.

I *usually* recommend avoiding these methods like the plague because they do some surprising and nasty stuff and they're inherently reintroducing references when my whole "easy rule" is to avoid doing that!

Those methods come with a lot of warnings, but the most interesting is this:

> You must enforce Rust's aliasing rules, since the returned lifetime `'a` is arbitrarily chosen and does not necessarily reflect the actual lifetime of the data. In particular, for the duration of this lifetime, the memory the pointer points to must not get accessed (read or written) through any other pointer.

Hey look it's the thing we talked about for 25 pages! I have already asserted we're *definitely* going to be fine to use references here, so aliasing solved! The other evil part is the signature:

```
pub unsafe fn as_mut<'a>(self) -> Option<&'a mut T>
```

Do you see how that lifetime isn't attached to the input at all, because `self` is by-value? Yeah that's what we call an "unbounded lifetime" and it's nasty stuff. It's willing to pretend to be as large as we ask it to be, even `'static`! The way you *deal* with that is by putting it somewhere that *is* bounded, which usually just means "return this from a function as soon as possible so that the function signature limits it".

Boy I'm nervous about this but we're gonna keep pushing through! Let's steal some iterator impls from the stack:

```
impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        self.0.pop()
    }
}

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

    fn next(&mut self) -> Option<Self::Item> {
        unsafe {
            self.next.map(|node| {
                self.next = node.next.as_ref();
                &node.elem
            })
        }
    }
}

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

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

Moment of truth time...

```
cargo test

running 15 tests
test fifth::test::basics ... ok
test fifth::test::into_iter ... ok
test fifth::test::iter ... ok
test fifth::test::iter_mut ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::into_iter ... ok
test fourth::test::peek ... ok
test second::test::basics ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::iter ... ok
test third::test::basics ... ok

test result: ok. 15 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
```

```
MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri test

running 15 tests
test fifth::test::basics ... ok
test fifth::test::into_iter ... ok
test fifth::test::iter ... ok
test fifth::test::iter_mut ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::into_iter ... ok
test fourth::test::peek ... ok
test second::test::basics ... ok
test second::test::into_iter ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::basics ... ok
test third::test::iter ... ok

test result: ok. 15 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

YES!!! Take that **NARRATOR**! Sometimes I don't make mistakes!

NARRATOR: but wasn't the whole point that the mistakes are there to teach the reader.

YEAH WELL SOMETIMES THE LESSON IS THAT I'M RIGHT AND EVERYONE SHOULD LISTEN TO ME WHEN I SAY THINGS ABOUT UNSAFE CODE BECAUSE I HAVE SPENT FAR TOO MUCH TIME THINKING ABOUT THE SOUNDNESS OF ITERATOR IMPLEMENTATIONS?! OK?! OK.

Anyway here's `peek` and `peek_mut`.

```
pub fn peek(&self) -> Option<&T> {
    unsafe {
        self.head.as_ref()
    }
}

pub fn peek_mut(&mut self) -> Option<&mut T> {
    unsafe {
        self.head.as_mut()
    }
}
```

I'm not even gonna test them because I never make mistakes anymore.

NARRATOR: `cargo build`

```
error[E0308]: mismatched types
  --> src\fifth.rs:66:13
   |
25 | impl<T> List<T> {
   |      - this type parameter
...
64 |     pub fn peek(&self) -> Option<&T> {
   |                           ---------- expected `Option<&T>`
   |                                      because of return type
65 |         unsafe {
66 |             self.head.as_ref()
   |             ^^^^^^^^^^^^^^^^^^ expected type parameter `T`,
   |                                found struct `fifth::Node`
   |
   = note: expected enum `Option<&T>`
              found enum `Option<&fifth::Node<T>>`
```

FINE.

```
pub fn peek(&self) -> Option<&T> {
    unsafe {
        self.head.as_ref().map(|node| &node.elem)
    }
}

pub fn peek_mut(&mut self) -> Option<&mut T> {
    unsafe {
        self.head.as_mut().map(|node| &mut node.elem)
    }
}
```

I guess I am going to *continue* to make mistakes, so we're going to be extra careful and add a new test I'm going to call "miri food": something that just messes around and mixes up our APIs a bunch to help miri catch our mistakes.

```rust
#[test]
fn miri_food() {
    let mut list = List::new();

    list.push(1);
    list.push(2);
    list.push(3);

    assert!(list.pop() == Some(1));
    list.push(4);
    assert!(list.pop() == Some(2));
    list.push(5);

    assert!(list.peek() == Some(&3));
    list.push(6);
    list.peek_mut().map(|x| *x *= 10);
    assert!(list.peek() == Some(&30));
    assert!(list.pop() == Some(30));

    for elem in list.iter_mut() {
        *elem *= 100;
    }

    let mut iter = list.iter();
    assert_eq!(iter.next(), Some(&400));
    assert_eq!(iter.next(), Some(&500));
    assert_eq!(iter.next(), Some(&600));
    assert_eq!(iter.next(), None);
    assert_eq!(iter.next(), None);

    assert!(list.pop() == Some(400));
    list.peek_mut().map(|x| *x *= 10);
    assert!(list.peek() == Some(&5000));
    list.push(7);

    // Drop it on the ground and let the dtor exercise itself
}
```

```
cargo test

running 16 tests
test fifth::test::basics ... ok
test fifth::test::into_iter ... ok
test fifth::test::iter ... ok
test fifth::test::iter_mut ... ok
test fifth::test::miri_food ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::into_iter ... ok
test fourth::test::peek ... ok
test second::test::into_iter ... ok
test second::test::basics ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::iter ... ok
test second::test::iter ... ok
test third::test::basics ... ok

test result: ok. 16 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out



MIRIFLAGS="-Zmiri-tag-raw-pointers" cargo +nightly-2022-01-21 miri test

running 16 tests
test fifth::test::basics ... ok
test fifth::test::into_iter ... ok
test fifth::test::iter ... ok
test fifth::test::iter_mut ... ok
test fifth::test::miri_food ... ok
test first::test::basics ... ok
test fourth::test::basics ... ok
test fourth::test::into_iter ... ok
test fourth::test::peek ... ok
test second::test::into_iter ... ok
test second::test::basics ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test third::test::iter ... ok
test second::test::iter ... ok
test third::test::basics ... ok

test result: ok. 16 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Perfect.

Final Code

Alright, so with a teeny-tiny dash of unsafety we managed to get a linear time improvement over the naive safe queue, and we managed to reuse almost all of the logic from the safe stack!

You know, except for that part where miri completely dunked on us and we had to write a short master's thesis on rust's memory model. You know, as you do.

But on the bright side we *didn't* have to write any jank Rc or RefCell stuff.

```
use std::ptr;

pub struct List<T> {
    head: Link<T>,
    tail: *mut Node<T>,
}

type Link<T> = *mut Node<T>;

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

pub struct IntoIter<T>(List<T>);

pub struct Iter<'a, T> {
    next: Option<&'a Node<T>>,
}

pub struct IterMut<'a, T> {
    next: Option<&'a mut Node<T>>,
}

impl<T> List<T> {
    pub fn new() -> Self {
        List { head: ptr::null_mut(), tail: ptr::null_mut() }
    }
    pub fn push(&mut self, elem: T) {
        unsafe {
            let new_tail = Box::into_raw(Box::new(Node {
                elem: elem,
                next: ptr::null_mut(),
            }));

            if !self.tail.is_null() {
                (*self.tail).next = new_tail;
            } else {
                self.head = new_tail;
            }

            self.tail = new_tail;
        }
    }
    pub fn pop(&mut self) -> Option<T> {
        unsafe {
            if self.head.is_null() {
                None
            } else {
                let head = Box::from_raw(self.head);
                self.head = head.next;

                if self.head.is_null() {
                    self.tail = ptr::null_mut();
                }

                Some(head.elem)
            }
        }
    }

    pub fn peek(&self) -> Option<&T> {
        unsafe {
            self.head.as_ref().map(|node| &node.elem)
        }
    }
```

```
    pub fn peek_mut(&mut self) -> Option<&mut T> {
        unsafe {
            self.head.as_mut().map(|node| &mut node.elem)
        }
    }

    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
    }

    pub fn iter(&self) -> Iter<'_, T> {
        unsafe {
            Iter { next: self.head.as_ref() }
        }
    }

    pub fn iter_mut(&mut self) -> IterMut<'_, T> {
        unsafe {
            IterMut { next: self.head.as_mut() }
        }
    }
}

impl<T> Drop for List<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() { }
    }
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        self.0.pop()
    }
}

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

    fn next(&mut self) -> Option<Self::Item> {
        unsafe {
            self.next.map(|node| {
                self.next = node.next.as_ref();
                &node.elem
            })
        }
    }
}

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

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

#[cfg(test)]
mod test {
    use super::List;
    #[test]
    fn basics() {
```

```
        let mut list = List::new();

        // Check empty list behaves right
        assert_eq!(list.pop(), None);

        // Populate list
        list.push(1);
        list.push(2);
        list.push(3);

        // Check normal removal
        assert_eq!(list.pop(), Some(1));
        assert_eq!(list.pop(), Some(2));

        // Push some more just to make sure nothing's corrupted
        list.push(4);
        list.push(5);

        // Check normal removal
        assert_eq!(list.pop(), Some(3));
        assert_eq!(list.pop(), Some(4));

        // Check exhaustion
        assert_eq!(list.pop(), Some(5));
        assert_eq!(list.pop(), None);

        // Check the exhaustion case fixed the pointer right
        list.push(6);
        list.push(7);

        // Check normal removal
        assert_eq!(list.pop(), Some(6));
        assert_eq!(list.pop(), Some(7));
        assert_eq!(list.pop(), None);
    }

    #[test]
    fn into_iter() {
        let mut list = List::new();
        list.push(1); list.push(2); list.push(3);

        let mut iter = list.into_iter();
        assert_eq!(iter.next(), Some(1));
        assert_eq!(iter.next(), Some(2));
        assert_eq!(iter.next(), Some(3));
        assert_eq!(iter.next(), None);
    }

    #[test]
    fn iter() {
        let mut list = List::new();
        list.push(1); list.push(2); list.push(3);

        let mut iter = list.iter();
        assert_eq!(iter.next(), Some(&1));
        assert_eq!(iter.next(), Some(&2));
        assert_eq!(iter.next(), Some(&3));
        assert_eq!(iter.next(), None);
    }

    #[test]
    fn iter_mut() {
        let mut list = List::new();
        list.push(1); list.push(2); list.push(3);

        let mut iter = list.iter_mut();
        assert_eq!(iter.next(), Some(&mut 1));
        assert_eq!(iter.next(), Some(&mut 2));
```

```
        assert_eq!(iter.next(), Some(&mut 3));
        assert_eq!(iter.next(), None);
    }

    #[test]
    fn miri_food() {
        let mut list = List::new();

        list.push(1);
        list.push(2);
        list.push(3);

        assert!(list.pop() == Some(1));
        list.push(4);
        assert!(list.pop() == Some(2));
        list.push(5);

        assert!(list.peek() == Some(&3));
        list.push(6);
        list.peek_mut().map(|x| *x *= 10);
        assert!(list.peek() == Some(&30));
        assert!(list.pop() == Some(30));

        for elem in list.iter_mut() {
            *elem *= 100;
        }

        let mut iter = list.iter();
        assert_eq!(iter.next(), Some(&400));
        assert_eq!(iter.next(), Some(&500));
        assert_eq!(iter.next(), Some(&600));
        assert_eq!(iter.next(), None);
        assert_eq!(iter.next(), None);

        assert!(list.pop() == Some(400));
        list.peek_mut().map(|x| *x *= 10);
        assert!(list.peek() == Some(&5000));
        list.push(7);

        // Drop it on the ground and let the dtor exercise itself
    }
}
```

A Production-Quality Unsafe Doubly-Linked Deque

We finally made it. My greatests nemesis: **std::collections::LinkedList, the Doubly-Linked Deque**.

The one that I tried and failed to destroy.

Our story begins as 2014 was coming to a close and we were rapidly approaching the release of Rust 1.0, Rust's first stable release. I had found myself in the role of caring for `std::collections`, or as we affectionately called it in those times, libcollections.

libcollections had spent years as a dumping ground for everyone's Cute Ideas and Vaguely Useful Things. This was all well and good when Rust was a fledgling experimental language, but if my children were going to escape the nest and be stabilized, they would have to prove their worth.

Until then I had encouraged and nurtured them all, but it was now time for them to face judgement for their failings.

I sunk my claws into the bedrock and carved tombstones for my most foolish children. A grisly monument that I placed in the town square for all to see:

Kill TreeMap, TreeSet, TrieMap, TrieSet, LruCache and EnumSet

Their fates were sealed, for my word was absolute. The other collections were horrified by my brutality, but they were not yet safe from their mother's wrath. I soon returned with two more tombstones:

Deprecate BitSet and BitVec

The Bit twins were more cunning than their fallen comrades, but they lacked the strength to escape me. Most thought my work done, but I soon took one more:

Deprecate VecMap

VecMap had tried to survive through stealth — it was so small and inoffensive! But that wasn't enough for the libcollections I saw in my vision of the future.

I surveyed the land and saw what remained:

- Vec and VecDeque - hearty and simple, the heart of computing.
- HashMap and HashSet - powerful and wise, the brain of computing.
- BTreeMap and BTreeSet - awkward but necessary, the liver of computing.
- BinaryHeap - crafty and dextrous, the ankle of computing.

I nodded in contentment. Simple and effective. My work was don—

No, DList, it can't be! I thought you died in that tragic garbage collection incident! The one which was definitely an accident and not intentional at all!

They had faked their death and taken on a new name, but it was still them: LinkedList, the shadowy and untrustworthy schemer of computing.

I spread word of their misdeeds to all that would hear me, but hearts were unmoved. LinkedList was a silver-tongued devil who had convinced everyone around me that it was some sort of fundamental and natural datastructure of computing. It had even convinced C++ that it was *the* list!

"How could you have a standard library without a *LinkedList*?"

Easily! Trivially!

"It's non-trivial unsafe code, so it makes sense to have it in the standard library!"

So are GPU drivers and video codecs, libcollections is minimalist!

But alas, LinkedList had gathered too many allies and grown too strong while I was distracted with its kin.

I fled to my laboratory and tried to devise some sort of evil clone or enhanced cyborg replicant that could rival and destroy it, but my grant funding ran out because my research was "too murderously evil" or somesuch nonsense.

LinkedList had won. I was defeated and forced into exile.

But you're here now. You've come this far. Surely now you can understand the depths of LinkedList's debauchery! Come, I will you show you everything you need to know to help me destroy it once and

for all — everything you need to know to implement an unsafe production-quality Doubly-Linked Deque.

How production-quality? Well we're going to completely rewrite my ancient Rust 1.0 linked-list crate, the one that is objectively better than the one in std. The one with Cursors on stable Rust, from 2015! Something the 2022 stdlib still doesn't have!

Layout

Let us begin by first studying the structure of our enemy. A Doubly-Linked List is conceptually simple, but that's how it decieves and manipulates you. It's the same kind of linked list we've looked at over and over, but the links go both ways. Double the links, double the evil.

So rather than this (gonna drop the Some/None stuff to keep it cleaner):

```
... -> (A, ptr) -> (B, ptr) -> ...
```

We have this:

```
... <-> (ptr, A, ptr) <-> (ptr, B, ptr) <-> ...
```

This lets you traverse the list from either direction, or seek back and forth with a cursor.

In exchange for this flexibility, every node has to store twice as many pointers, and every operation has to fix up way more pointers. It's a significant enough complication that it's a lot easier to make a mistake, so we're going to be doing a lot of testing.

You might have also noticed that I intentionally haven't drawn the *ends* of the list. This is because this is one of the places where there are genuinely defensible options for our implementation. We *definitely* need our implementation to have two pointers: one to the start of the list, and one to the end of the list.

There are two notable ways to do this in my mind: "traditional" and "dummy node".

The traditional approach is the simple extension of how we did a Stack — just store the head and tail pointers on the stack:

```
[ptr, ptr] <-> (ptr, A, ptr) <-> (ptr, B, ptr)
  ^                                  ^
  +----------------------------------+
```

This is fine, but it has one downside: corner cases. There are now two edges to our list, which means twice as many corner cases. It's easy to forget one and have a serious bug.

The dummy node approach attempts to smooth out these corner cases by adding an extra node to our list which contains no data but links the two ends together into a ring:

```
[ptr] -> (ptr, ?DUMMY?, ptr) <-> (ptr, A, ptr) <-> (ptr, B, ptr)
           ^                                          ^
           +------------------------------------------+
```

By doing this, every node *always* has actual pointers to a previous and next node in the list. Even when you remove the last element from the list, you just end up stitching the dummy node to point at itself:

```
[ptr] -> (ptr, ?DUMMY?, ptr)
          ^             ^
          +-------------+
```

There is a part of me that finds this *very* satisfying and elegant. Unfortunately, it has a couple practical problems:

Problem 1: An extra indirection and allocation, especially for the empty list, which must include the dummy node. Potential solutions include:

- Don't allocate the dummy node until something is inserted: simple and effective, but it adds back some of the corner cases we were trying to avoid by using dummy pointers!
- Use a static copy-on-write empty singleton dummy node, with some really clever scheme that lets the Copy-On-Write checks piggy-back on normal checks: look I'm really tempted, I really do love that shit, but we can't go down that dark path in this book. Read ThinVec's sourcecode if you want to see that kind of perverted stuff.
- Store the dummy node on the stack - not practical in a language without C++-style move-constructors. I'm sure there's something weird thing we could do here with pinning but we're not gonna.

Problem 2: What *value* is stored in the dummy node? Sure if it's an integer it's fine, but what if we're storing a list full of `Box`? It may be impossible for us to initialized this value! Potential solutions include:

- Make every node store `Option<T>`: simple and effective, but also bloated and annoying.
- Make every node store `MaybeUninit<T>`. Horrifying and annoying.
- *Really* careful and clever inheritance-style type punning so the dummy node doesn't include the data field. This is also tempting but it's extremely dangerous and annoying. Read BTreeMap's source if you want to see that kind of perverted stuff.

The problems really outweigh the convenience for a language like Rust, so we're going to stick to the traditional layout. We'll be using the same basic design as we did for the unsafe queue in the previous chapter:

```
pub struct LinkedList<T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
}

type Link<T> = *mut Node<T>;

struct Node<T> {
    front: Link<T>,
    back: Link<T>,
    elem: T,
}
```

(Now that we have reached the doubly-linked-deque, we have finally earned the right to call ourselves LinkedList, for this is the True Linked List.)

This isn't quite a *true* production-quality layout yet. It's *fine* but there's magic tricks we can do to tell Rust what we're doing a bit better. To do that we're going to need to go... deeper.

Variance and PhantomData

It's going to be annoying to punt on this now and fix it later, so we're going to do the Hardcore Layout stuff now.

There are five terrible horsemen of making unsafe Rust collections:

1. Variance
2. Drop Check
3. NonNull Optimizations
4. The isize::MAX Allocation Rule
5. Zero-Sized Types

Mercifully, the last 2 aren't going to be a problem for us.

The third we *could* make into our problem but it's more trouble than it's worth -- if you've opted into a LinkedList you've already given up the battle on memory-effeciency 100-fold already.

The second is something that I used to insist was really important and that std messes around with, but the defaults are safe, the ways to mess with it are unstable, and you need to try *so very hard* to ever notice the limitations of the defaults, so, don't worry about it.

That just leaves us with Variance. To be honest, you can probably punt on this one too, but I still have my pride as a Collections Person, so we're going to Do The Variance Thing.

So, surprise: Rust has subtyping. In particular, `&'big T` is a *subtype* of `&'small T`. Why? Well because if some code needs a reference that lives for some particular region of the program, it's usually perfectly fine to give it a reference that lives for *longer*. Like, intuitively that's just true, right?

Why is this important? Well imagine some code that takes two values with the same type:

```
fn take_two<T>(_val1: T, _val2: T) { }
```

This is some deeply boring code, and so we should expect it to work with T=&u32 fine, right?

```
fn two_refs<'big: 'small, 'small>(
    big: &'big u32,
    small: &'small u32,
) {
    take_two(big, small);
}

fn take_two<T>(_val1: T, _val2: T) { }
```

Yep, that compiles fine!

Now let's have some fun and wrap it in, oh, I don't know, `std::cell::Cell`:

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

fn two_refs<'big: 'small, 'small>(
    // NOTE: these two lines changed
    big: Cell<&'big u32>,
    small: Cell<&'small u32>,
) {
    take_two(big, small);
}

fn take_two<T>(_val1: T, _val2: T) { }
```

```
error[E0623]: lifetime mismatch
 --> src/main.rs:7:19
  |
4 |     big: Cell<&'big u32>,
  |               ---------
5 |     small: Cell<&'small u32>,
  |                 ----------- these two types are declared with different
lifetimes...
6 | ) {
7 |     take_two(big, small);
  |                   ^^^^^ ...but data from `small` flows into `big` here
```

Huh??? We didn't touch the lifetimes, why's the compiler angry now!?

Ah well, the lifetime "subtyping" stuff must be really simple, so it falls over if you wrap the references in anything, see look it breaks with Vec too:

```
fn two_refs<'big: 'small, 'small>(
    big: Vec<&'big u32>,
    small: Vec<&'small u32>,
) {
    take_two(big, small);
}

fn take_two<T>(_val1: T, _val2: T) { }
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 1.07s
     Running `target/debug/playground`
```

See it doesn't compile eith-- wait what??? Vec is magic??????

Well, yes. But also, no. The magic was inside us all along, and that magic is ✨*Variance*✨.

Read the nomicon's chapter on subtyping if you want all the gorey details, but basically subtyping *isn't* always safe. In particular it's not safe when mutable references are involved because you can use things like `mem::swap` and suddenly oops dangling pointers!

Things that are "like mutable references" are *invariant* which means they block subtyping from happening on their generic parameters. So for safety, `&mut T` is invariant over T, and `Cell<T>` is invariant over T because `&Cell<T>` is basically just `&mut T` (because of interior mutability).

Almost everything that isn't invariant is *covariant*, and that just means that subtyping "passes through" it and continues to work normally (there are also contravariant types that make subtyping go backwards but they are really rare and no one likes them so I won't mention them again).

Collections generally contain a mutable pointer to their data, so you might expect them to be invariant too, but in fact, they don't need to be! Because of Rust's ownership system, `Vec<T>` is

semantically equivalent to `T`, and that means it's safe for it to be covariant!

Unfortunately, this definition is invariant:

```
pub struct LinkedList<T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
}

type Link<T> = *mut Node<T>;

struct Node<T> {
    front: Link<T>,
    back: Link<T>,
    elem: T,
}
```

But how is Rust actually deciding the variance of things? Well in the good-old-days before 1.0 we messed around with just letting people specify the variance they wanted and... it was an absolute train-wreck! Subtyping and variance is really hard to wrap your head around, and core developers genuinely disagreed on basic terminology! So we moved to a "variance by example" approach: the compiler just looks at your fields and copies their variances. If there's any kind of disagreement, then invariance always wins, because that's safe.

So what's in our type definitions that Rust is getting mad about? `*mut`!

Raw pointers in Rust really just try to let you do whatever, but they have exactly one safety feature: because most people have no idea that variance and subtyping are a thing in Rust, and being *incorrectly* covariant would be horribly dangerous, `*mut T` is invariant, because there's a good chance it's being used "as" `&mut T`.

This is extremely annoying for Exactly Me as a person who has spent a lot of time writing collections in Rust. This is why when I made std::ptr::NonNull, I added this little piece of magic:

> Unlike `*mut T`, `NonNull<T>` was chosen to be covariant over `T`. This makes it possible to use `NonNull<T>` when building covariant types, but introduces the risk of unsoundness if used in a type that shouldn't actually be covariant.

But hey, it's interface is built around `*mut T`, what's the deal! Is it just magic?! Let's look:

```
pub struct NonNull<T> {
    pointer: *const T,
}


impl<T> NonNull<T> {
    pub unsafe fn new_unchecked(ptr: *mut T) -> Self {
        // SAFETY: the caller must guarantee that `ptr` is non-null.
        unsafe { NonNull { pointer: ptr as *const T } }
    }
}
```

NOPE. NO MAGIC HERE! NonNull just abuses the fact that `*const T` is covariant and stores that instead, casting back and forth between `*mut T` at the API boundary to make it "look like" it's storing a `*mut T`. That's the whole trick! That's how collections in Rust are covariant! And it's

miserable! So I made the Good Pointer Type do it for you! You're welcome! Enjoy your subtyping footgun!

The solution to all your problems is to use NonNull, and then if you want to have nullable pointers again, use `Option<NonNull<T>>`. Are we really going to bother doing that..?

Yep! It sucks, but we're making *production grade linked lists* so we're going to eat all our vegetables and do things the hard way (we could just use bare `*const T` and cast everywhere, but I genuinely want to see how painful this is... for Ergonomics Science).

So here's our final type definitions:

```
use std::ptr::NonNull;

// !!!This changed!!!
pub struct LinkedList<T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
}

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

struct Node<T> {
    front: Link<T>,
    back: Link<T>,
    elem: T,
}
```

...wait nope, one last thing. Any time you do raw pointer stuff, you should add a Ghost to protect your pointers:

```
use std::marker::PhantomData;

pub struct LinkedList<T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    /// We semantically store values of T by-value.
    _boo: PhantomData<T>,
}
```

In this case I don't think we *actually* need PhantomData, but any time you *do* use NonNull (or just raw pointers in general), you should always add it to be safe and make it clear to the compiler and others what you *think* you're doing.

PhantomData is a way for us to give the compiler an extra "example" field that *conceptually* exists in your type but for various reasons (indirection, type erasure, ...) doesn't. In this case we're using NonNull because we're claiming our type behaves "as if" it stored a value T, so we add a PhantomData to make that explicit.

The stdlib actually has other reasons to do this because it has access to the accursed Drop Check overrides, but that feature has been reworked so many times that I don't actually know if the PhantomData thing *is* a thing for it anymore. I'm still going to cargo-cult it for all eternity, because Drop Check Magic is burned into my brain!

(Node literally stores a T, so it doesn't have to do this, yay!)

...ok for real we're done with layout now! On to actual basic functionality!

Basics

Alright, this is the part of the book that sucks, and why it took me 7 years to write this chapter! Time to just burn through a whole lot of really boring stuff we've done 5 times already, but extra verbose and long because we have to do everything twice and with `Option<NonNull<Node<T>>>`!

```
impl<T> LinkedList<T> {
    pub fn new() -> Self {
        Self {
            front: None,
            back: None,
            len: 0,
            _boo: PhantomData,
        }
    }
}
```

PhantomData is a weird type with no fields so you just make one by, saying its type name. *shrug*

```
pub fn push_front(&mut self, elem: T) {
    // SAFETY: it's a linked-list, what do you want?
    unsafe {
        let new = NonNull::new_unchecked(Box::into_raw(Box::new(Node {
            front: None,
            back: None,
            elem,
        })));
        if let Some(old) = self.front {
            // Put the new front before the old one
            (*old).front = Some(new);
            (*new).back = Some(old);
        } else {
            // If there's no front, then we're the empty list and need
            // to set the back too. Also here's some integrity checks
            // for testing, in case we mess up.
            debug_assert!(self.back.is_none());
            debug_assert!(self.front.is_none());
            debug_assert!(self.len == 0);
            self.back = Some(new);
        }
        self.front = Some(new);
        self.len += 1;
    }
}
```

```
error[E0614]: type `NonNull<Node<T>>` cannot be dereferenced
  --> src\lib.rs:39:17
   |
39 |                 (*old).front = Some(new);
   |                 ^^^^^^
```

Ah yes, I truly hate my pointer-y children. We need to explicitly get the raw pointer out of NonNull with `as_ptr`, because DerefMut is defined in terms of `&mut` and we don't want to randomly introduce safe references into our unsafe code!

```
            (*old.as_ptr()).front = Some(new);
            (*new.as_ptr()).back = Some(old);
```

```
   Compiling linked-list v0.0.3
warning: field is never read: `elem`
  --> src\lib.rs:16:5
   |
16 |     elem: T,
   |     ^^^^^^^
   |
   = note: `#[warn(dead_code)]` on by default

warning: `linked-list` (lib) generated 1 warning (1 duplicate)
warning: `linked-list` (lib test) generated 1 warning
    Finished test [unoptimized + debuginfo] target(s) in 0.33s
```

Nice, now for pop (and len):

```
pub fn pop_front(&mut self) -> Option<T> {
    unsafe {
        // Only have to do stuff if there is a front node to pop.
        // Note that we don't need to mess around with `take` anymore
        // because everything is Copy and there are no dtors that will
        // run if we mess up... right? :) Riiiight? :)))
        self.front.map(|node| {
            // Bring the Box back to life so we can move out its value and
            // Drop it (Box continues to magically understand this for us).
            let boxed_node = Box::from_raw(node.as_ptr());
            let result = boxed_node.elem;

            // Make the next node into the new front.
            self.front = boxed_node.back;
            if let Some(new) = self.front {
                // Cleanup its reference to the removed node
                (*new.as_ptr()).front = None;
            } else {
                // If the front is now null, then this list is now empty!
                debug_assert!(self.len == 1);
                self.back = None;
            }

            self.len -= 1;
            result
            // Box gets implicitly freed here, knows there is no T.
        })
    }
}

pub fn len(&self) -> usize {
    self.len
}
```

```
   Compiling linked-list v0.0.3
    Finished dev [unoptimized + debuginfo] target(s) in 0.37s
```

Seems legit to me, time to write a test!

```
#[cfg(test)]
mod test {
    use super::LinkedList;

    #[test]
    fn test_basic_front() {
        let mut list = LinkedList::new();

        // Try to break an empty list
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);

        // Try to break a one item list
        list.push_front(10);
        assert_eq!(list.len(), 1);
        assert_eq!(list.pop_front(), Some(10));
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);

        // Mess around
        list.push_front(10);
        assert_eq!(list.len(), 1);
        list.push_front(20);
        assert_eq!(list.len(), 2);
        list.push_front(30);
        assert_eq!(list.len(), 3);
        assert_eq!(list.pop_front(), Some(30));
        assert_eq!(list.len(), 2);
        list.push_front(40);
        assert_eq!(list.len(), 3);
        assert_eq!(list.pop_front(), Some(40));
        assert_eq!(list.len(), 2);
        assert_eq!(list.pop_front(), Some(20));
        assert_eq!(list.len(), 1);
        assert_eq!(list.pop_front(), Some(10));
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);
    }
}
```

```
   Compiling linked-list v0.0.3
    Finished test [unoptimized + debuginfo] target(s) in 0.40s
     Running unittests src\lib.rs

running 1 test
test test::test_basic_front ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

Hooray, we're perfect!

...Right?

Drop and Panic Safety

So hey, did you notice this comment:

```
// Note that we don't need to mess around with `take` anymore
// because everything is Copy and there are no dtors that will
// run if we mess up... right? :) Riiiight? :)))
```

Is it right?

Sorry did you forget the book you're reading? Of course it's wrong! (Sort Of.)

Let's look at the inner body of pop_front again:

```
// Bring the Box back to life so we can move out its value and
// Drop it (Box continues to magically understand this for us).
let boxed_node = Box::from_raw(node.as_ptr());
let result = boxed_node.elem;

// Make the next node into the new front.
self.front = boxed_node.back;
if let Some(new) = self.front {
    // Cleanup its reference to the removed node
    (*new.as_ptr()).front = None;
} else {
    // If the front is now null, then this list is now empty!
    debug_assert!(self.len == 1);
    self.back = None;
}

self.len -= 1;
result
// Box gets implicitly freed here, knows there is no T.
```

Do you see the bug? Horrifyingly, it's actually this line:

```
debug_assert!(self.len == 1);
```

Really? Our friggin' integrity check for tests is a bug?? Yes!!! Well, if we implement our collection right it *shouldn't* be, but it can turn something benign like "oh we are doing a bad job of keeping len up to date" into *An Exploitable Memory Safety Bug*! Why? Because it can panic! Most of the time you don't have to think or worry about panics, but once you start writing *really* unsafe code and playing fast and loose with "invariants", you need to become hyper-vigilant about panics!

We've gotta talk about *exception safety* (AKA panic safety, AKA unwind safety, ...).

So here's the deal: by default, panics are *unwinding*. Unwinding is just a fancy way to say "make every single function immediately return". You might think "well, if *everyone* returns then the program is about to die, so why care about it?", but you'd be wrong!

We have to care for two reasons: destructors run when a function returns, and the unwind can be *caught*. In both cases, code can keep running after a panic, so we need to be very careful and make sure our unsafe collections are always in *some* kind of coherent state whenever a panic could occur, because each panic is an implicit early return!

Let's think about what state our collection is in when we get to that line:

We have our boxed_node on the stack, and we've extracted the element from it. If we were to return at this point, the Box would be dropped, and the node would be freed. Do you see it now..? self.back is still pointing at that freed node! Once we implement the rest of our collection and start using self.back for things, this could result in a use-after-free! Yikes!

Interestingly, this line has similar problems, but it's much safer:

```
self.len -= 1;
```

By default in debug builds Rust checks for underflows and overflows and will panic when they happen. Yes, every arithmetic operation is a panic-safety hazard! This one is *better* because it happens after we've repaired all of our invariants, so it won't cause memory-safety issues... as long as we don't trust len to be right, but then, if we underflow it's definitely wrong, so we were dead either way! The debug assert is in some sense *worse* because it can escalate a minor issue into a critical one!

I've brought up the term "invariants" a few times, and that's because it's a really useful concept for panic-safety! Basically, to an outside observer of our collection, there are certain property we're always upholding. For a LinkedList, one of those is that any node that is reachable in our list is still allocated and initialized.

Inside the implementation we have a bit more flexibility to break invariants *temporarily* as long as we make sure to repair them *before anyone notices*. This is actually one of the "killer apps" of Rust's ownership and borrowing system for collections: if an operation requires an `&mut Self`, then we are *guaranteed* that we have exclusive access to our collection and that it's fine for us to temporarily break invariants, safe in the knowledge that no one can sneakily mess with it.

Perhaps the greatest expression of this is Vec::drain, which actually lets you completely smash a core invariant of Vec and start moving values out from the *front* or even *middle* of a Vec. The reason this is *sound* is because the Drain iterator that we return holds an `&mut` to the Vec, and so all access is gated behind it! No one can observe the Vec until the Drain iterator goes away, and then it's destructor can "repair" the Vec before anyone can notice, it's perfe--

It's not perfect. Unfortunately, you can't rely on destructors in code you don't control to run, and so even with Drain we need to do a little extra work to make our type always preserved invariants, but in a kind of goofy way: we just set the Vec's len to 0 at the start, so if anyone leaks the Drain, then they will have a *safe* Vec... but they will have also lost a bunch of data. You leak me? I leak you! An eye for an eye! True justice!

For a situation where you *can* actually use destructors for panic-safety, check out the BinaryHeap::sift_up case study.

Anyway, we won't be needing all of this fancy stuff for our LinkedLists, we just need to be a bit more vigilant about where we break our invariants, what we trust/require to be correct, and to avoid introducing unnecessary unwinds in the middle of hairy tasks.

In this case, we have two options to make our code a bit more robust:

- Use operations like Option::take a lot more aggressively, because they are more "transactional" and have a tendency to preserve invariants.
- Kill the debug_asserts and trust ourselves to write better tests with dedicated "integrity check" functions that won't run in user code ever.

In principle I like the first option, but it doesn't actually work great for a doubly-linked list, because everything is doubly-redundantly encoded. Option::take wouldn't fix the problem here, but moving the debug_assert down a line would. But really, why make things harder for ourselves? Let's just remove those debug_asserts, and make sure anything can panic is at the start or end of our methods, where our invariants should be known to hold.

(In this way it's perhaps more accurate to think of them as *preconditions* and *postconditions* but you really should endeavour to treat them as invariants as much as possible!)

Here's our full implementation now:

```
use std::ptr::NonNull;
use std::marker::PhantomData;

pub struct LinkedList<T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<T>,
}

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

struct Node<T> {
    front: Link<T>,
    back: Link<T>,
    elem: T,
}

impl<T> LinkedList<T> {
    pub fn new() -> Self {
        Self {
            front: None,
            back: None,
            len: 0,
            _boo: PhantomData,
        }
    }

    pub fn push_front(&mut self, elem: T) {
        // SAFETY: it's a linked-list, what do you want?
        unsafe {
            let new = NonNull::new_unchecked(Box::into_raw(Box::new(Node {
                front: None,
                back: None,
                elem,
            })));
            if let Some(old) = self.front {
                // Put the new front before the old one
                (*old.as_ptr()).front = Some(new);
                (*new.as_ptr()).back = Some(old);
            } else {
                // If there's no front, then we're the empty list and need
                // to set the back too.
                self.back = Some(new);
            }
            // These things always happen!
            self.front = Some(new);
            self.len += 1;
        }
    }

    pub fn pop_front(&mut self) -> Option<T> {
        unsafe {
            // Only have to do stuff if there is a front node to pop.
            self.front.map(|node| {
                // Bring the Box back to life so we can move out its value and
                // Drop it (Box continues to magically understand this for us).
                let boxed_node = Box::from_raw(node.as_ptr());
                let result = boxed_node.elem;

                // Make the next node into the new front.
                self.front = boxed_node.back;
                if let Some(new) = self.front {
                    // Cleanup its reference to the removed node
                    (*new.as_ptr()).front = None;
                } else {
```

```
                    // If the front is now null, then this list is now empty!
                    self.back = None;
                }

                self.len -= 1;
                result
                // Box gets implicitly freed here, knows there is no T.
            })
        }
    }

    pub fn len(&self) -> usize {
        self.len
    }
}
```

What can panic here? Well, knowing that honestly requires you to be a bit of a Rust expert, but thankfully, I am!

The only places I can see in this code that *possibly* can panic (barring some absolute fuckery where someone recompiles the stdlib with debug_asserts enabled, but this is not something you should ever do) are `Box::new` (for out-of-memory conditions) and the len arithmetic. All of that stuff is at the very end or very start of our methods, so yep, we're being nice and safe!

...were you surprised by `Box::new` being able to panic? Panics will get you like that! Try to preserve those invariants so you don't need to worry about it!

Boring Combinatorics

Ok, back to our regularly scheduled linked lists!

First let's knock out `Drop` which is trivial with pop:

```
impl<T> Drop for LinkedList<T> {
    fn drop(&mut self) {
        // Pop until we have to stop
        while let Some(_) = self.pop_front() { }
    }
}
```

We've got to fill in a bunch of really boring combinatoric implementations like front, front_mut, back, back_mut, iter, iter_mut, into_iter, ...

You could do them with macros or whatever but honestly, that's a worse fate than copy-pasting. We're just going to do a lot of copy-pasting. I have *very carefully* crafted the previous push/pop implementations so that we should be able to *literally* just swap front and back and the code does/says the right thing! Hooray for painful experience! (It's so tempting to talk about "prev and next" for nodes, but I find it's really worth it to just consistently talk about "front" and "back" as much as possible to avoid mistakes.)

Alright, first up, `front`:

```
pub fn front(&self) -> Option<&T> {
    unsafe {
        self.front.map(|node| &(*node.as_ptr()).elem)
    }
}
```

Hey actually, this book is really old and some nice new things have been added like the `?` operator which does an early return on Option::None, does that make our code nicer?

```
pub fn front(&self) -> Option<&T> {
    unsafe {
        Some(&(*self.front?.as_ptr()).elem)
    }
}
```

Maybe? It's kind of a wash for something this simple, and the previous section was all about how early returns are kinda spooky for us, so maybe we should prefer being a bit more explicit here (I'm sticking to the `map` implementation). On to front_mut:

```
pub fn front_mut(&mut self) -> Option<&mut T> {
    unsafe {
        self.front.map(|node| &mut (*node.as_ptr()).elem)
    }
}
```

I'll just dump all the `back` versions at the end.

Next up, iterators. Unlike all of our previous lists we've *finally* unlocked the ability to do DoubleEndedIterator, and if we're going for production quality we're gonna do ExactSizeIterator too.

So in addition to `next` and `size_hint`, we're going to support `next_back` and `len`.

The vigilant among you might notice that IterMut seems a lot more sketchy with double-ended iteration, but it's actually still sound!

... god this is gonna be a lot of boilerplate. Maybe I should really write a macro... no, no, that's still a worse fate.

```
pub struct Iter<'a, T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<&'a T>,
}

impl<T> LinkedList<T> {
    pub fn iter(&self) -> Iter<T> {
        Iter {
            front: self.front,
            back: self.back,
            len: self.len,
            _boo: PhantomData,
        }
    }
}


impl<'a, T> IntoIterator for &'a LinkedList<T> {
    type IntoIter = Iter<'a, T>;
    type Item = &'a T;

    fn into_iter(self) -> Self::IntoIter {
        self.iter()
    }
}

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

    fn next(&mut self) -> Option<Self::Item> {
        // While self.front == self.back is a tempting condition to check here,
        // it won't do the right for yielding the last element! That sort of
        // thing only works for arrays because of "one-past-the-end" pointers.
        if self.len > 0 {
            // We could unwrap front, but this is safer and easier
            self.front.map(|node| unsafe {
                self.len -= 1;
                self.front = (*node.as_ptr()).back;
                &(*node.as_ptr()).elem
            })
        } else {
            None
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        (self.len, Some(self.len))
    }
}

impl<'a, T> DoubleEndedIterator for Iter<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        if self.len > 0 {
            self.back.map(|node| unsafe {
                self.len -= 1;
                self.back = (*node.as_ptr()).front;
                &(*node.as_ptr()).elem
            })
        } else {
            None
        }
    }
}

impl<'a, T> ExactSizeIterator for Iter<'a, T> {
```

```
    fn len(&self) -> usize {
        self.len
    }
}
```

...that's just `.iter()`...

we'll paste IterMut at the end, it's literally the exact same code with `mut` in a lot of places, let's just knock out `into_iter` first. We can mercifully still lean on our tried-and-true solution of just making it wrap our collection and using pop for next:

```
pub struct IntoIter<T> {
    list: LinkedList<T>,
}

impl<T> LinkedList<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter {
            list: self
        }
    }
}


impl<T> IntoIterator for LinkedList<T> {
    type IntoIter = IntoIter<T>;
    type Item = T;

    fn into_iter(self) -> Self::IntoIter {
        self.into_iter()
    }
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;

    fn next(&mut self) -> Option<Self::Item> {
        self.list.pop_front()
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        (self.list.len, Some(self.list.len))
    }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        self.list.pop_back()
    }
}

impl<T> ExactSizeIterator for IntoIter<T> {
    fn len(&self) -> usize {
        self.list.len
    }
}
```

Still a crapload of boiler plate, but at least it's *satisfying* boilerplate.

Alright, here's all of our code with all the combinatorics filled in:

```
use std::ptr::NonNull;
use std::marker::PhantomData;

pub struct LinkedList<T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<T>,
}

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

struct Node<T> {
    front: Link<T>,
    back: Link<T>,
    elem: T,
}

pub struct Iter<'a, T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<&'a T>,
}

pub struct IterMut<'a, T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<&'a mut T>,
}

pub struct IntoIter<T> {
    list: LinkedList<T>,
}

impl<T> LinkedList<T> {
    pub fn new() -> Self {
        Self {
            front: None,
            back: None,
            len: 0,
            _boo: PhantomData,
        }
    }

    pub fn push_front(&mut self, elem: T) {
        // SAFETY: it's a linked-list, what do you want?
        unsafe {
            let new = NonNull::new_unchecked(Box::into_raw(Box::new(Node {
                front: None,
                back: None,
                elem,
            })));
            if let Some(old) = self.front {
                // Put the new front before the old one
                (*old.as_ptr()).front = Some(new);
                (*new.as_ptr()).back = Some(old);
            } else {
                // If there's no front, then we're the empty list and need
                // to set the back too.
                self.back = Some(new);
            }
            // These things always happen!
            self.front = Some(new);
            self.len += 1;
```

```
        }
    }

    pub fn push_back(&mut self, elem: T) {
        // SAFETY: it's a linked-list, what do you want?
        unsafe {
            let new = NonNull::new_unchecked(Box::into_raw(Box::new(Node {
                back: None,
                front: None,
                elem,
            })));
            if let Some(old) = self.back {
                // Put the new back before the old one
                (*old.as_ptr()).back = Some(new);
                (*new.as_ptr()).front = Some(old);
            } else {
                // If there's no back, then we're the empty list and need
                // to set the front too.
                self.front = Some(new);
            }
            // These things always happen!
            self.back = Some(new);
            self.len += 1;
        }
    }

    pub fn pop_front(&mut self) -> Option<T> {
        unsafe {
            // Only have to do stuff if there is a front node to pop.
            self.front.map(|node| {
                // Bring the Box back to life so we can move out its value and
                // Drop it (Box continues to magically understand this for us).
                let boxed_node = Box::from_raw(node.as_ptr());
                let result = boxed_node.elem;

                // Make the next node into the new front.
                self.front = boxed_node.back;
                if let Some(new) = self.front {
                    // Cleanup its reference to the removed node
                    (*new.as_ptr()).front = None;
                } else {
                    // If the front is now null, then this list is now empty!
                    self.back = None;
                }

                self.len -= 1;
                result
                // Box gets implicitly freed here, knows there is no T.
            })
        }
    }

    pub fn pop_back(&mut self) -> Option<T> {
        unsafe {
            // Only have to do stuff if there is a back node to pop.
            self.back.map(|node| {
                // Bring the Box front to life so we can move out its value and
                // Drop it (Box continues to magically understand this for us).
                let boxed_node = Box::from_raw(node.as_ptr());
                let result = boxed_node.elem;

                // Make the next node into the new back.
                self.back = boxed_node.front;
                if let Some(new) = self.back {
                    // Cleanup its reference to the removed node
                    (*new.as_ptr()).back = None;
                } else {
                    // If the back is now null, then this list is now empty!
```

```
                    self.front = None;
                }

                self.len -= 1;
                result
                // Box gets implicitly freed here, knows there is no T.
            })
        }
    }

    pub fn front(&self) -> Option<&T> {
        unsafe {
            self.front.map(|node| &(*node.as_ptr()).elem)
        }
    }

    pub fn front_mut(&mut self) -> Option<&mut T> {
        unsafe {
            self.front.map(|node| &mut (*node.as_ptr()).elem)
        }
    }

    pub fn back(&self) -> Option<&T> {
        unsafe {
            self.back.map(|node| &(*node.as_ptr()).elem)
        }
    }

    pub fn back_mut(&mut self) -> Option<&mut T> {
        unsafe {
            self.back.map(|node| &mut (*node.as_ptr()).elem)
        }
    }

    pub fn len(&self) -> usize {
        self.len
    }

    pub fn iter(&self) -> Iter<T> {
        Iter {
            front: self.front,
            back: self.back,
            len: self.len,
            _boo: PhantomData,
        }
    }

    pub fn iter_mut(&mut self) -> IterMut<T> {
        IterMut {
            front: self.front,
            back: self.back,
            len: self.len,
            _boo: PhantomData,
        }
    }

    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter {
            list: self
        }
    }
}

impl<T> Drop for LinkedList<T> {
    fn drop(&mut self) {
        // Pop until we have to stop
        while let Some(_) = self.pop_front() { }
    }
```

```
}

impl<'a, T> IntoIterator for &'a LinkedList<T> {
    type IntoIter = Iter<'a, T>;
    type Item = &'a T;

    fn into_iter(self) -> Self::IntoIter {
        self.iter()
    }
}

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

    fn next(&mut self) -> Option<Self::Item> {
        // While self.front == self.back is a tempting condition to check here,
        // it won't do the right for yielding the last element! That sort of
        // thing only works for arrays because of "one-past-the-end" pointers.
        if self.len > 0 {
            // We could unwrap front, but this is safer and easier
            self.front.map(|node| unsafe {
                self.len -= 1;
                self.front = (*node.as_ptr()).back;
                &(*node.as_ptr()).elem
            })
        } else {
            None
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        (self.len, Some(self.len))
    }
}

impl<'a, T> DoubleEndedIterator for Iter<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        if self.len > 0 {
            self.back.map(|node| unsafe {
                self.len -= 1;
                self.back = (*node.as_ptr()).front;
                &(*node.as_ptr()).elem
            })
        } else {
            None
        }
    }
}

impl<'a, T> ExactSizeIterator for Iter<'a, T> {
    fn len(&self) -> usize {
        self.len
    }
}

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

    fn into_iter(self) -> Self::IntoIter {
        self.iter_mut()
    }
}

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

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

```
        // While self.front == self.back is a tempting condition to check here,
        // it won't do the right for yielding the last element! That sort of
        // thing only works for arrays because of "one-past-the-end" pointers.
        if self.len > 0 {
            // We could unwrap front, but this is safer and easier
            self.front.map(|node| unsafe {
                self.len -= 1;
                self.front = (*node.as_ptr()).back;
                &mut (*node.as_ptr()).elem
            })
        } else {
            None
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        (self.len, Some(self.len))
    }
}

impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        if self.len > 0 {
            self.back.map(|node| unsafe {
                self.len -= 1;
                self.back = (*node.as_ptr()).front;
                &mut (*node.as_ptr()).elem
            })
        } else {
            None
        }
    }
}

impl<'a, T> ExactSizeIterator for IterMut<'a, T> {
    fn len(&self) -> usize {
        self.len
    }
}

impl<T> IntoIterator for LinkedList<T> {
    type IntoIter = IntoIter<T>;
    type Item = T;

    fn into_iter(self) -> Self::IntoIter {
        self.into_iter()
    }
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;

    fn next(&mut self) -> Option<Self::Item> {
        self.list.pop_front()
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        (self.list.len, Some(self.list.len))
    }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        self.list.pop_back()
    }
}

impl<T> ExactSizeIterator for IntoIter<T> {
```

```
    fn len(&self) -> usize {
        self.list.len
    }
}


#[cfg(test)]
mod test {
    use super::LinkedList;

    #[test]
    fn test_basic_front() {
        let mut list = LinkedList::new();

        // Try to break an empty list
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);

        // Try to break a one item list
        list.push_front(10);
        assert_eq!(list.len(), 1);
        assert_eq!(list.pop_front(), Some(10));
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);

        // Mess around
        list.push_front(10);
        assert_eq!(list.len(), 1);
        list.push_front(20);
        assert_eq!(list.len(), 2);
        list.push_front(30);
        assert_eq!(list.len(), 3);
        assert_eq!(list.pop_front(), Some(30));
        assert_eq!(list.len(), 2);
        list.push_front(40);
        assert_eq!(list.len(), 3);
        assert_eq!(list.pop_front(), Some(40));
        assert_eq!(list.len(), 2);
        assert_eq!(list.pop_front(), Some(20));
        assert_eq!(list.len(), 1);
        assert_eq!(list.pop_front(), Some(10));
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);
    }
}
```

Filling In Random Bits

Hey you said you wanted to be production-quality, didn't you?

Here's some more random gunk to toss in to be a "good" collection:

```
impl<T> LinkedList<T> {
    pub fn is_empty(&self) -> bool {
        self.len == 0
    }

    pub fn clear(&mut self) {
        // Oh look it's drop again
        while let Some(_) = self.pop_front() { }
    }
}
```

And now we've got a bunch of traits to implement that everyone expects:

```
impl<T> Default for LinkedList<T> {
    fn default() -> Self {
        Self::new()
    }
}

impl<T: Clone> Clone for LinkedList<T> {
    fn clone(&self) -> Self {
        let mut new_list = Self::new();
        for item in self {
            new_list.push_back(item.clone());
        }
        new_list
    }
}

impl<T> Extend<T> for LinkedList<T> {
    fn extend<I: IntoIterator<Item = T>>(&mut self, iter: I) {
        for item in iter {
            self.push_back(item);
        }
    }
}

impl<T> FromIterator<T> for LinkedList<T> {
    fn from_iter<I: IntoIterator<Item = T>>(iter: I) -> Self {
        let mut list = Self::new();
        list.extend(iter);
        list
    }
}

impl<T: Debug> Debug for LinkedList<T> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.debug_list().entries(self).finish()
    }
}

impl<T: PartialEq> PartialEq for LinkedList<T> {
    fn eq(&self, other: &Self) -> bool {
        self.len() == other.len() && self.iter().eq(other)
    }

    fn ne(&self, other: &Self) -> bool {
        self.len() != other.len() || self.iter().ne(other)
    }
}

impl<T: Eq> Eq for LinkedList<T> { }

impl<T: PartialOrd> PartialOrd for LinkedList<T> {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        self.iter().partial_cmp(other)
    }
}

impl<T: Ord> Ord for LinkedList<T> {
    fn cmp(&self, other: &Self) -> Ordering {
        self.iter().cmp(other)
    }
}

impl<T: Hash> Hash for LinkedList<T> {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.len().hash(state);
        for item in self {
            item.hash(state);
```

```
        }
    }
}
```

I definitely wrote all of these from scratch, and didn't just copy the std impls. Because they're so interesting, and I definitely remember the subtleties of manually implementing Hash. Yeah, that's something I think about All The Time...

Ok there's actually a few things worth noting here.

First, a nasty namespace clash. For whatever reason std now has macros named Hash and Debug, and so if you don't have the traits imported, you'll get really cryptic errors about macros instead of the proper "missing trait".

The other intersting thing to talk about is Hash itself. Do you see how we hash in `len`? That's actually really important! If collections don't hash in lengths, they can accidentally make themselves vulnerable to prefix collisions. For instance, what distinguishes `["he", "llo"]` from `["hello"]`? If no one is hashing lengths or some other "separator", nothing! Making it too easy for hash collisions to accidentally or maliciously happen can result in serious sadness, so just do it!

Alright, here's our current code:

```
use std::cmp::Ordering;
use std::fmt::{self, Debug};
use std::hash::{Hash, Hasher};
use std::iter::FromIterator;
use std::ptr::NonNull;
use std::marker::PhantomData;

pub struct LinkedList<T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<T>,
}

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

struct Node<T> {
    front: Link<T>,
    back: Link<T>,
    elem: T,
}

pub struct Iter<'a, T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<&'a T>,
}

pub struct IterMut<'a, T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<&'a mut T>,
}

pub struct IntoIter<T> {
    list: LinkedList<T>,
}

impl<T> LinkedList<T> {
    pub fn new() -> Self {
        Self {
            front: None,
            back: None,
            len: 0,
            _boo: PhantomData,
        }
    }

    pub fn push_front(&mut self, elem: T) {
        // SAFETY: it's a linked-list, what do you want?
        unsafe {
            let new = NonNull::new_unchecked(Box::into_raw(Box::new(Node {
                front: None,
                back: None,
                elem,
            })));
            if let Some(old) = self.front {
                // Put the new front before the old one
                (*old.as_ptr()).front = Some(new);
                (*new.as_ptr()).back = Some(old);
            } else {
                // If there's no front, then we're the empty list and need
                // to set the back too.
                self.back = Some(new);
```

```
            }
            // These things always happen!
            self.front = Some(new);
            self.len += 1;
        }
    }

    pub fn push_back(&mut self, elem: T) {
        // SAFETY: it's a linked-list, what do you want?
        unsafe {
            let new = NonNull::new_unchecked(Box::into_raw(Box::new(Node {
                back: None,
                front: None,
                elem,
            })));
            if let Some(old) = self.back {
                // Put the new back before the old one
                (*old.as_ptr()).back = Some(new);
                (*new.as_ptr()).front = Some(old);
            } else {
                // If there's no back, then we're the empty list and need
                // to set the front too.
                self.front = Some(new);
            }
            // These things always happen!
            self.back = Some(new);
            self.len += 1;
        }
    }

    pub fn pop_front(&mut self) -> Option<T> {
        unsafe {
            // Only have to do stuff if there is a front node to pop.
            self.front.map(|node| {
                // Bring the Box back to life so we can move out its value and
                // Drop it (Box continues to magically understand this for us).
                let boxed_node = Box::from_raw(node.as_ptr());
                let result = boxed_node.elem;

                // Make the next node into the new front.
                self.front = boxed_node.back;
                if let Some(new) = self.front {
                    // Cleanup its reference to the removed node
                    (*new.as_ptr()).front = None;
                } else {
                    // If the front is now null, then this list is now empty!
                    self.back = None;
                }

                self.len -= 1;
                result
                // Box gets implicitly freed here, knows there is no T.
            })
        }
    }

    pub fn pop_back(&mut self) -> Option<T> {
        unsafe {
            // Only have to do stuff if there is a back node to pop.
            self.back.map(|node| {
                // Bring the Box front to life so we can move out its value and
                // Drop it (Box continues to magically understand this for us).
                let boxed_node = Box::from_raw(node.as_ptr());
                let result = boxed_node.elem;

                // Make the next node into the new back.
                self.back = boxed_node.front;
                if let Some(new) = self.back {
```

```
                // Cleanup its reference to the removed node
                (*new.as_ptr()).back = None;
            } else {
                // If the back is now null, then this list is now empty!
                self.front = None;
            }

            self.len -= 1;
            result
            // Box gets implicitly freed here, knows there is no T.
        })
    }
}

pub fn front(&self) -> Option<&T> {
    unsafe {
        self.front.map(|node| &(*node.as_ptr()).elem)
    }
}

pub fn front_mut(&mut self) -> Option<&mut T> {
    unsafe {
        self.front.map(|node| &mut (*node.as_ptr()).elem)
    }
}

pub fn back(&self) -> Option<&T> {
    unsafe {
        self.back.map(|node| &(*node.as_ptr()).elem)
    }
}

pub fn back_mut(&mut self) -> Option<&mut T> {
    unsafe {
        self.back.map(|node| &mut (*node.as_ptr()).elem)
    }
}

pub fn len(&self) -> usize {
    self.len
}

pub fn is_empty(&self) -> bool {
    self.len == 0
}

pub fn clear(&mut self) {
    // Oh look it's drop again
    while let Some(_) = self.pop_front() { }
}

pub fn iter(&self) -> Iter<T> {
    Iter {
        front: self.front,
        back: self.back,
        len: self.len,
        _boo: PhantomData,
    }
}

pub fn iter_mut(&mut self) -> IterMut<T> {
    IterMut {
        front: self.front,
        back: self.back,
        len: self.len,
        _boo: PhantomData,
    }
}
```

```
    pub fn into_iter(self) -> IntoIter<T> {
        IntoIter {
            list: self
        }
    }
}

impl<T> Drop for LinkedList<T> {
    fn drop(&mut self) {
        // Pop until we have to stop
        while let Some(_) = self.pop_front() { }
    }
}

impl<T> Default for LinkedList<T> {
    fn default() -> Self {
        Self::new()
    }
}

impl<T: Clone> Clone for LinkedList<T> {
    fn clone(&self) -> Self {
        let mut new_list = Self::new();
        for item in self {
            new_list.push_back(item.clone());
        }
        new_list
    }
}

impl<T> Extend<T> for LinkedList<T> {
    fn extend<I: IntoIterator<Item = T>>(&mut self, iter: I) {
        for item in iter {
            self.push_back(item);
        }
    }
}

impl<T> FromIterator<T> for LinkedList<T> {
    fn from_iter<I: IntoIterator<Item = T>>(iter: I) -> Self {
        let mut list = Self::new();
        list.extend(iter);
        list
    }
}

impl<T: Debug> Debug for LinkedList<T> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.debug_list().entries(self).finish()
    }
}

impl<T: PartialEq> PartialEq for LinkedList<T> {
    fn eq(&self, other: &Self) -> bool {
        self.len() == other.len() && self.iter().eq(other)
    }

    fn ne(&self, other: &Self) -> bool {
        self.len() != other.len() || self.iter().ne(other)
    }
}

impl<T: Eq> Eq for LinkedList<T> { }

impl<T: PartialOrd> PartialOrd for LinkedList<T> {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        self.iter().partial_cmp(other)
```

```
    }
}

impl<T: Ord> Ord for LinkedList<T> {
    fn cmp(&self, other: &Self) -> Ordering {
        self.iter().cmp(other)
    }
}

impl<T: Hash> Hash for LinkedList<T> {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.len().hash(state);
        for item in self {
            item.hash(state);
        }
    }
}

impl<'a, T> IntoIterator for &'a LinkedList<T> {
    type IntoIter = Iter<'a, T>;
    type Item = &'a T;

    fn into_iter(self) -> Self::IntoIter {
        self.iter()
    }
}

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

    fn next(&mut self) -> Option<Self::Item> {
        // While self.front == self.back is a tempting condition to check here,
        // it won't do the right for yielding the last element! That sort of
        // thing only works for arrays because of "one-past-the-end" pointers.
        if self.len > 0 {
            // We could unwrap front, but this is safer and easier
            self.front.map(|node| unsafe {
                self.len -= 1;
                self.front = (*node.as_ptr()).back;
                &(*node.as_ptr()).elem
            })
        } else {
            None
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        (self.len, Some(self.len))
    }
}

impl<'a, T> DoubleEndedIterator for Iter<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        if self.len > 0 {
            self.back.map(|node| unsafe {
                self.len -= 1;
                self.back = (*node.as_ptr()).front;
                &(*node.as_ptr()).elem
            })
        } else {
            None
        }
    }
}

impl<'a, T> ExactSizeIterator for Iter<'a, T> {
    fn len(&self) -> usize {
        self.len
```

```
    }
}

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

    fn into_iter(self) -> Self::IntoIter {
        self.iter_mut()
    }
}

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

    fn next(&mut self) -> Option<Self::Item> {
        // While self.front == self.back is a tempting condition to check here,
        // it won't do the right for yielding the last element! That sort of
        // thing only works for arrays because of "one-past-the-end" pointers.
        if self.len > 0 {
            // We could unwrap front, but this is safer and easier
            self.front.map(|node| unsafe {
                self.len -= 1;
                self.front = (*node.as_ptr()).back;
                &mut (*node.as_ptr()).elem
            })
        } else {
            None
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        (self.len, Some(self.len))
    }
}

impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        if self.len > 0 {
            self.back.map(|node| unsafe {
                self.len -= 1;
                self.back = (*node.as_ptr()).front;
                &mut (*node.as_ptr()).elem
            })
        } else {
            None
        }
    }
}

impl<'a, T> ExactSizeIterator for IterMut<'a, T> {
    fn len(&self) -> usize {
        self.len
    }
}

impl<T> IntoIterator for LinkedList<T> {
    type IntoIter = IntoIter<T>;
    type Item = T;

    fn into_iter(self) -> Self::IntoIter {
        self.into_iter()
    }
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
```

```
    fn next(&mut self) -> Option<Self::Item> {
        self.list.pop_front()
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        (self.list.len, Some(self.list.len))
    }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        self.list.pop_back()
    }
}

impl<T> ExactSizeIterator for IntoIter<T> {
    fn len(&self) -> usize {
        self.list.len
    }
}


#[cfg(test)]
mod test {
    use super::LinkedList;

    #[test]
    fn test_basic_front() {
        let mut list = LinkedList::new();

        // Try to break an empty list
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);

        // Try to break a one item list
        list.push_front(10);
        assert_eq!(list.len(), 1);
        assert_eq!(list.pop_front(), Some(10));
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);

        // Mess around
        list.push_front(10);
        assert_eq!(list.len(), 1);
        list.push_front(20);
        assert_eq!(list.len(), 2);
        list.push_front(30);
        assert_eq!(list.len(), 3);
        assert_eq!(list.pop_front(), Some(30));
        assert_eq!(list.len(), 2);
        list.push_front(40);
        assert_eq!(list.len(), 3);
        assert_eq!(list.pop_front(), Some(40));
        assert_eq!(list.len(), 2);
        assert_eq!(list.pop_front(), Some(20));
        assert_eq!(list.len(), 1);
        assert_eq!(list.pop_front(), Some(10));
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);
    }
}
```

Testing

Alright I put off testing for a while because well, we both know that we're now masters of Rust and we don't make mistakes anymore! Also just, this is a rewrite of an old crate, so I just have all the tests already. They're tests, you've seen tests a lot already. Here they are:

```rust
#[cfg(test)]
mod test {
    use super::LinkedList;

    fn generate_test() -> LinkedList<i32> {
        list_from(&[0, 1, 2, 3, 4, 5, 6])
    }

    fn list_from<T: Clone>(v: &[T]) -> LinkedList<T> {
        v.iter().map(|x| (*x).clone()).collect()
    }

    #[test]
    fn test_basic_front() {
        let mut list = LinkedList::new();

        // Try to break an empty list
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);

        // Try to break a one item list
        list.push_front(10);
        assert_eq!(list.len(), 1);
        assert_eq!(list.pop_front(), Some(10));
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);

        // Mess around
        list.push_front(10);
        assert_eq!(list.len(), 1);
        list.push_front(20);
        assert_eq!(list.len(), 2);
        list.push_front(30);
        assert_eq!(list.len(), 3);
        assert_eq!(list.pop_front(), Some(30));
        assert_eq!(list.len(), 2);
        list.push_front(40);
        assert_eq!(list.len(), 3);
        assert_eq!(list.pop_front(), Some(40));
        assert_eq!(list.len(), 2);
        assert_eq!(list.pop_front(), Some(20));
        assert_eq!(list.len(), 1);
        assert_eq!(list.pop_front(), Some(10));
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);
    }

    #[test]
    fn test_basic() {
        let mut m = LinkedList::new();
        assert_eq!(m.pop_front(), None);
        assert_eq!(m.pop_back(), None);
        assert_eq!(m.pop_front(), None);
        m.push_front(1);
        assert_eq!(m.pop_front(), Some(1));
        m.push_back(2);
        m.push_back(3);
        assert_eq!(m.len(), 2);
        assert_eq!(m.pop_front(), Some(2));
        assert_eq!(m.pop_front(), Some(3));
        assert_eq!(m.len(), 0);
        assert_eq!(m.pop_front(), None);
```

```
        m.push_back(1);
        m.push_back(3);
        m.push_back(5);
        m.push_back(7);
        assert_eq!(m.pop_front(), Some(1));

        let mut n = LinkedList::new();
        n.push_front(2);
        n.push_front(3);
        {
            assert_eq!(n.front().unwrap(), &3);
            let x = n.front_mut().unwrap();
            assert_eq!(*x, 3);
            *x = 0;
        }
        {
            assert_eq!(n.back().unwrap(), &2);
            let y = n.back_mut().unwrap();
            assert_eq!(*y, 2);
            *y = 1;
        }
        assert_eq!(n.pop_front(), Some(0));
        assert_eq!(n.pop_front(), Some(1));
    }

    #[test]
    fn test_iterator() {
        let m = generate_test();
        for (i, elt) in m.iter().enumerate() {
            assert_eq!(i as i32, *elt);
        }
        let mut n = LinkedList::new();
        assert_eq!(n.iter().next(), None);
        n.push_front(4);
        let mut it = n.iter();
        assert_eq!(it.size_hint(), (1, Some(1)));
        assert_eq!(it.next().unwrap(), &4);
        assert_eq!(it.size_hint(), (0, Some(0)));
        assert_eq!(it.next(), None);
    }

    #[test]
    fn test_iterator_double_end() {
        let mut n = LinkedList::new();
        assert_eq!(n.iter().next(), None);
        n.push_front(4);
        n.push_front(5);
        n.push_front(6);
        let mut it = n.iter();
        assert_eq!(it.size_hint(), (3, Some(3)));
        assert_eq!(it.next().unwrap(), &6);
        assert_eq!(it.size_hint(), (2, Some(2)));
        assert_eq!(it.next_back().unwrap(), &4);
        assert_eq!(it.size_hint(), (1, Some(1)));
        assert_eq!(it.next_back().unwrap(), &5);
        assert_eq!(it.next_back(), None);
        assert_eq!(it.next(), None);
    }

    #[test]
    fn test_rev_iter() {
        let m = generate_test();
        for (i, elt) in m.iter().rev().enumerate() {
            assert_eq!(6 - i as i32, *elt);
        }
        let mut n = LinkedList::new();
        assert_eq!(n.iter().rev().next(), None);
        n.push_front(4);
```

```
        let mut it = n.iter().rev();
        assert_eq!(it.size_hint(), (1, Some(1)));
        assert_eq!(it.next().unwrap(), &4);
        assert_eq!(it.size_hint(), (0, Some(0)));
        assert_eq!(it.next(), None);
    }

    #[test]
    fn test_mut_iter() {
        let mut m = generate_test();
        let mut len = m.len();
        for (i, elt) in m.iter_mut().enumerate() {
            assert_eq!(i as i32, *elt);
            len -= 1;
        }
        assert_eq!(len, 0);
        let mut n = LinkedList::new();
        assert!(n.iter_mut().next().is_none());
        n.push_front(4);
        n.push_back(5);
        let mut it = n.iter_mut();
        assert_eq!(it.size_hint(), (2, Some(2)));
        assert!(it.next().is_some());
        assert!(it.next().is_some());
        assert_eq!(it.size_hint(), (0, Some(0)));
        assert!(it.next().is_none());
    }

    #[test]
    fn test_iterator_mut_double_end() {
        let mut n = LinkedList::new();
        assert!(n.iter_mut().next_back().is_none());
        n.push_front(4);
        n.push_front(5);
        n.push_front(6);
        let mut it = n.iter_mut();
        assert_eq!(it.size_hint(), (3, Some(3)));
        assert_eq!(*it.next().unwrap(), 6);
        assert_eq!(it.size_hint(), (2, Some(2)));
        assert_eq!(*it.next_back().unwrap(), 4);
        assert_eq!(it.size_hint(), (1, Some(1)));
        assert_eq!(*it.next_back().unwrap(), 5);
        assert!(it.next_back().is_none());
        assert!(it.next().is_none());
    }

    #[test]
    fn test_eq() {
        let mut n: LinkedList<u8> = list_from(&[]);
        let mut m = list_from(&[]);
        assert!(n == m);
        n.push_front(1);
        assert!(n != m);
        m.push_back(1);
        assert!(n == m);

        let n = list_from(&[2, 3, 4]);
        let m = list_from(&[1, 2, 3]);
        assert!(n != m);
    }

    #[test]
    fn test_ord() {
        let n = list_from(&[]);
        let m = list_from(&[1, 2, 3]);
        assert!(n < m);
        assert!(m > n);
        assert!(n <= n);
```

```
        assert!(n >= n);
    }

    #[test]
    fn test_ord_nan() {
        let nan = 0.0f64 / 0.0;
        let n = list_from(&[nan]);
        let m = list_from(&[nan]);
        assert!(!(n < m));
        assert!(!(n > m));
        assert!(!(n <= m));
        assert!(!(n >= m));

        let n = list_from(&[nan]);
        let one = list_from(&[1.0f64]);
        assert!(!(n < one));
        assert!(!(n > one));
        assert!(!(n <= one));
        assert!(!(n >= one));

        let u = list_from(&[1.0f64, 2.0, nan]);
        let v = list_from(&[1.0f64, 2.0, 3.0]);
        assert!(!(u < v));
        assert!(!(u > v));
        assert!(!(u <= v));
        assert!(!(u >= v));

        let s = list_from(&[1.0f64, 2.0, 4.0, 2.0]);
        let t = list_from(&[1.0f64, 2.0, 3.0, 2.0]);
        assert!(!(s < t));
        assert!(s > one);
        assert!(!(s <= one));
        assert!(s >= one);
    }

    #[test]
    fn test_debug() {
        let list: LinkedList<i32> = (0..10).collect();
        assert_eq!(format!("{:?}", list), "[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]");

        let list: LinkedList<&str> = vec!["just", "one", "test", "more"]
            .iter().copied()
            .collect();
        assert_eq!(format!("{:?}", list), r#"["just", "one", "test", "more"]"#);
    }

    #[test]
    fn test_hashmap() {
        // Check that HashMap works with this as a key

        let list1: LinkedList<i32> = (0..10).collect();
        let list2: LinkedList<i32> = (1..11).collect();
        let mut map = std::collections::HashMap::new();

        assert_eq!(map.insert(list1.clone(), "list1"), None);
        assert_eq!(map.insert(list2.clone(), "list2"), None);

        assert_eq!(map.len(), 2);

        assert_eq!(map.get(&list1), Some(&"list1"));
        assert_eq!(map.get(&list2), Some(&"list2"));

        assert_eq!(map.remove(&list1), Some("list1"));
        assert_eq!(map.remove(&list2), Some("list2"));

        assert!(map.is_empty());
```

```
    }
}
```

And now for the moments of truth:

```
cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
     Running unittests src\lib.rs

running 12 tests
test test::test_basic ... ok
test test::test_basic_front ... ok
test test::test_eq ... ok
test test::test_iterator ... ok
test test::test_iterator_mut_double_end ... ok
test test::test_ord_nan ... ok
test test::test_iterator_double_end ... ok
test test::test_mut_iter ... ok
test test::test_rev_iter ... ok
test test::test_hashmap ... ok
test test::test_ord ... ok
test test::test_debug ... ok

test result: ok. 12 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s
```

```
$env:MIRIFLAGS="-Zmiri-tag-raw-pointers"
cargo miri test
   Compiling linked-list v0.0.3
    Finished test [unoptimized + debuginfo] target(s) in 0.35s
     Running unittests src\lib.rs

running 12 tests
test test::test_basic ... ok
test test::test_basic_front ... ok
test test::test_debug ... ok
test test::test_eq ... ok
test test::test_hashmap ... ok
test test::test_iterator ... ok
test test::test_iterator_double_end ... ok
test test::test_iterator_mut_double_end ... ok
test test::test_mut_iter ... ok
test test::test_ord ... ok
test test::test_ord_nan ... ok
test test::test_rev_iter ... ok

test result: ok. 12 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

We did it, we actually didn't screw up. This isn't a trick! All of our practice and training was finally worth it, we've finally written *good code*!!!

Now with all that gunk out of the way, we can get back to the Interesting Stuff!

Send, Sync, and Compile Tests

Ok actually we do have one more pair of traits to think about, but they're special. We have to deal with Rust's Holy Roman Empire: The Unsafe Opt-In Built-In Traits (OIBITs): Send and Sync, which are in fact opt-out and built-out (1 out of 3 is pretty good!).

Like Copy, these traits have absolutely no code associated with them, and are just markers that your type has a particular property. Send says that your type is safe to send to another thread. Sync says your type is safe to share between threads (&Self: Send).

The same argument for LinkedList being covariant applies here: generally normal collections which don't use fancy interior mutability tricks are safe to make Send and Sync.

But I said they're *opt out*. So actually, are we already? How would we know?

Let's add some new magic to our code: random private garbage that won't compile unless our types have the properties we expect:

```
#[allow(dead_code)]
fn assert_properties() {
    fn is_send<T: Send>() {}
    fn is_sync<T: Sync>() {}

    is_send::<LinkedList<i32>>();
    is_sync::<LinkedList<i32>>();

    is_send::<IntoIter<i32>>();
    is_sync::<IntoIter<i32>>();

    is_send::<Iter<i32>>();
    is_sync::<Iter<i32>>();

    is_send::<IterMut<i32>>();
    is_sync::<IterMut<i32>>();

    is_send::<Cursor<i32>>();
    is_sync::<Cursor<i32>>();

    fn linked_list_covariant<'a, T>(x: LinkedList<&'static T>) -> LinkedList<&'a T> { x
}
    fn iter_covariant<'i, 'a, T>(x: Iter<'i, &'static T>) -> Iter<'i, &'a T> { x }
    fn into_iter_covariant<'a, T>(x: IntoIter<&'static T>) -> IntoIter<&'a T> { x }
}
```

```
cargo build
   Compiling linked-list v0.0.3
error[E0277]: `NonNull<Node<i32>>` cannot be sent between threads safely
   --> src\lib.rs:433:5
    |
433 |     is_send::<LinkedList<i32>>();
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^ `NonNull<Node<i32>>` cannot be sent between
threads safely
    |
    = help: within `LinkedList<i32>`, the trait `Send` is not implemented for
`NonNull<Node<i32>>`
    = note: required because it appears within the type `Option<NonNull<Node<i32>>>`
note: required because it appears within the type `LinkedList<i32>`
   --> src\lib.rs:8:12
    |
8   | pub struct LinkedList<T> {
    |            ^^^^^^^^^^
note: required by a bound in `is_send`
   --> src\lib.rs:430:19
    |
430 |     fn is_send<T: Send>() {}
    |                   ^^^^ required by this bound in `is_send`

<a million more errors>
```

Oh geez, what gives! I had that great Holy Roman Empire joke!

Well, I lied to you when I said raw pointers have only one safety guard: this is the other. `*const` AND `*mut` explicitly opt out of Send and Sync to be safe, so we do *actually* have to opt back in:

```
unsafe impl<T: Send> Send for LinkedList<T> {}
unsafe impl<T: Sync> Sync for LinkedList<T> {}

unsafe impl<'a, T: Send> Send for Iter<'a, T> {}
unsafe impl<'a, T: Sync> Sync for Iter<'a, T> {}

unsafe impl<'a, T: Send> Send for IterMut<'a, T> {}
unsafe impl<'a, T: Sync> Sync for IterMut<'a, T> {}
```

Note that we have to write *unsafe impl* here: these are *unsafe traits*! Unsafe code (like concurrency libraries) gets to rely on us only implementing these traits correctly! Since there's no actual code, the guarantee we're making is just that, yes, we are indeed safe to Send or Share between threads!

Don't just slap these on lightly, but I am a Certified Professional here to say: yep there's are totally fine. Note how we don't need to implement Send and Sync for IntoIter: it just contains LinkedList, so it auto-derives Send and Sync — I told you they were actually opt out! (You opt out with the hillarious syntax of `impl !Send for MyType {}`.)

```
cargo build
   Compiling linked-list v0.0.3
    Finished dev [unoptimized + debuginfo] target(s) in 0.18s
```

Ok nice!

...Wait, actually it would be really dangerous if stuff that *shouldn't* be these things wasn't. In particular, IterMut *definitely* shouldn't be covariant, because it's "like" `&mut T`. But how can we check that?

With Magic! Well, actually, with rustdoc! Ok well we don't have to use rustdoc for this, but it's the funniest way to do it. See, if you write a doccomment and include a code block, then rustdoc will try to compile and run it, so we can use that to make fresh anonymous "programs" that don't affect the main one:

```
    /// ```
    /// use linked_list::IterMut;
    ///
    /// fn iter_mut_covariant<'i, 'a, T>(x: IterMut<'i, &'static T>) -> IterMut<'i, &'a
T> { x }
    /// ```
    fn iter_mut_invariant() {}
```

```
cargo test

...

   Doc-tests linked-list

running 1 test
test src\lib.rs - assert_properties::iter_mut_invariant (line 458) ... FAILED

failures:

---- src\lib.rs - assert_properties::iter_mut_invariant (line 458) stdout ----
error[E0308]: mismatched types
 --> src\lib.rs:461:86
  |
6 | fn iter_mut_covariant<'i, 'a, T>(x: IterMut<'i, &'static T>) -> IterMut<'i, &'a T>
{ x }
  |
^ lifetime mismatch
  |
  = note: expected struct `linked_list::IterMut<'_, &'a T>`
             found struct `linked_list::IterMut<'_, &'static T>`
```

Ok cool, we've proved it's invariant, but uh, now our tests fail. No worries, rustdoc lets you say that's expected by annotating the fence with compile_fail!

(Actually we only proved it's "not covariant" but honestly if you manage to make a type "accidentaly and incorrectly contravariant" then, congrats?)

```
    /// ```compile_fail
    /// use linked_list::IterMut;
    ///
    /// fn iter_mut_covariant<'i, 'a, T>(x: IterMut<'i, &'static T>) -> IterMut<'i, &'a
T> { x }
    /// ```
    fn iter_mut_invariant() {}
```

```
cargo test
   Compiling linked-list v0.0.3
    Finished test [unoptimized + debuginfo] target(s) in 0.49s
     Running unittests src\lib.rs

...

   Doc-tests linked-list

running 1 test
test src\lib.rs - assert_properties::iter_mut_invariant (line 458) - compile fail ...
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.12s
```

Yay! I recommend always making the test without compile_fail so that you can confirm that it fails to compile *for the right reason*. For instance, that test will also fail (and therefore pass) if you forget the `use`, which, is not what we want! While it's conceptually appealing to be able to "require" a specific error from the compiler, this would be an absolute nightmare that would effectively make it a breaking change *for the compiler to produce better errors*. We want the compiler to get better, so, no you don't get to have that.

(Oh wait, we can actually just specify the error code we want next to the compile_fail **but this only works on nightly and is a bad idea to rely on for the reasons state above. It will be silently**

ignored on not-nightly.)

```
    /// ```compile_fail,E0308
    /// use linked_list::IterMut;
    ///
    /// fn iter_mut_covariant<'i, 'a, T>(x: IterMut<'i, &'static T>) -> IterMut<'i, &'a
T> { x }
    /// ```
    fn iter_mut_invariant() {}
```

...also, did you notice the part where we actually made IterMut invariant? It was easy to miss, since I "just" copy-pasted Iter and dumped it at the end. It's the last line here:

```
pub struct IterMut<'a, T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<&'a mut T>,
}
```

Let's try removing that PhantomData:

```
 cargo build
   Compiling linked-list v0.0.3 (C:\Users\ninte\dev\contain\linked-list)
error[E0392]: parameter `'a` is never used
  --> src\lib.rs:30:20
   |
30 | pub struct IterMut<'a, T> {
   |                    ^^ unused parameter
   |
   = help: consider removing `'a`, referring to it in a field, or using a marker such
as `PhantomData`
```

Ha! The compiler has our back and won't just let us *not* use the lifetime. Let's try using the *wrong* example instead:

```
    _boo: PhantomData<&'a T>,
```

```
cargo build
   Compiling linked-list v0.0.3 (C:\Users\ninte\dev\contain\linked-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.17s
```

It builds! Do our tests catch a problem now?

```
cargo test

...

   Doc-tests linked-list

running 1 test
test src\lib.rs - assert_properties::iter_mut_invariant (line 458) - compile fail ...
FAILED

failures:

---- src\lib.rs - assert_properties::iter_mut_invariant (line 458) stdout ----
Test compiled successfully, but it's marked `compile_fail`.

failures:
    src\lib.rs - assert_properties::iter_mut_invariant (line 458)

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.15s
```

Eyyy!!! The system works! I love having tests that actually do their job, so that I don't have to be quite so horrified of looming mistakes!

An Introduction To Cursors

OK!!! We now have a LinkedList that's on par with std's 1.0 implementation! Which of course means that our LinkedList is *still completely useless*. We've taken the enormous performance penalty of implementing a Deque as a linked list, **and we don't have any of the APIs that make it actually useful**.

Here's how we do against the "killer apps" of linked lists:

- 🚫 Getting to do weird intrusive stuff
- 🚫 Getting to do weird lockfree stuff
- 🚫 Getting to store Dynamically Sized Types
- 💥 O(1) push/pop without amortization (if you are willing to believe that malloc is O(1))
- 🚫 O(1) list splitting
- 🚫 O(1) list splicing

Well... 1 out of 6 is... better than nothing! Do you see why I wanted to rip this thing out of std?

We're not going to make our list support "weird" stuff, because that's all adhoc and domain-specific. But the splitting and splicing thing, now that's something we can do!

But here's the problem: actually *reaching* the k^{th} element in a LinkedList takes O(k) time, so how can we *possibly* do arbitrary splits and merges in O(1)? Well, the trick is that you don't have an API like `split_at(index)` -- you make a system where the user can statefully iterate to a position in the list and make O(1) modifications at that point!

Hey, we already have iterators! Can we use them for this? Kind of... but one of their super-powers gets in the way. You may recall that the way that we write out the lifetimes for by-ref iterators means that the references they return *aren't* tied to the iterator. This lets us repeatedly call `next` and hold onto the elements:

```
let mut list = ...;
let iter = list.iter_mut();
let elem1 = list.next();
let elem2 = list.next();

if elem1 == elem2 { ... }
```

If the returned references borrowed the iterator, then this code wouldn't work at all. The compiler would just complain about the second call to `next`! This flexibility is great, but it puts some implicit constraints on us:

- By-Mutable-Ref Iterators can never go backwards and yield an element again, because the user would be able to get two `&mut`'s to the same element, breaking fundamental rules of the language.
- By-Ref Iterators can't have extra methods which could possibly modify the underlying collection in a way that would invalidate any reference that has already been yielded.

Unfortunately, both of these things are *exactly* what we want our LinkedList API to do! So we can't just use iterators, we need something new: *Cursors*.

Cursors are exactly like the little blinking `|` you get when you're editing some text on a computer. It's a position in a sequence (the text) that you can move around (with the arrow keys), and whenever you type the edits happen at that point.

See if I just

press

enter

the whole

text

gets broken in half.

Sorry you're standing behind me and watching me type this right? So that totally makes sense, right? Right.

Now if you've ever had the misfortune of having a keyboard with an "insert" key and actually pressed it, you know that there's actually technically two interpretations of cursors: they can either lie between elements (characters) or *on* elements. I'm pretty sure no one has ever pressed "insert" on purpose in their life, and that it exists purely as a Suffering Button, so it's pretty obvious which one is Better and Right: cursors go between elements!

Pretty rock-solid logic right there, I don't think anyone can disagree with me.

Sorry what? There was an RFC in 2018 to add Cursors to Rust's LinkedList?

> With a Cursor one can seek back and forth through a list and get the current element. With a CursorMut One can seek back and forth and get mutable references to elements, and it can insert and delete elements before and behind the current element (along with performing several list operations such as splitting and splicing).

Current element? This cursor is *on* elements, not between them! I can't believe they didn't accept my totally rock-solid argument! So yeah you can just go use the Cursor in std... wait, it's 2022, and Rust 1.60 still has Cursor marked as unstable?

Hey wait:

> Cursors always rest between two elements in the list, and index in a logically circular way. To accommodate this, there is a "ghost" non-element that yields None between the head and tail of the list.

HEY WAIT. This is the opposite of what the RFC says??? But wait all the docs on the methods still refer to "current" elements... wait hold on, where have I seen this ghost stuff before. Oh wait, didn't I do that in my old linked-list fork where I prototyped?

> Cursors always rest between two elements in the list, and index in a logically circular way. To accomadate this, there is a "ghost" non-element that yields None between the head and tail of the List.

Hold up what the fuck. This isn't a gag, I am actually trying to Read The Docs right now. Did std actually RFC a different design from the one I proposed in 2015, but then copy-paste the docs from my prototype??? Is std meta-shitposting me for writing a book about how much I hate LinkedList????? Like yeah I built that prototype to demonstrate the concept so that people would let me add it to std and make LinkedList not useless but, qu'est-ce que le fuck??????????????

Ok you know what, clearly std is blessing my design as the objectively superior one, so we're going to do my design. Also that's nice because this entire chapter is me actually literally rewriting that library from scratch, so not changing the API sounds Good To Me!

Here's the full top-level docs I wrote:

> A Cursor is like an iterator, except that it can freely seek back-and-forth, and can safely mutate the list during iteration. This is because the lifetime of its yielded references are tied to its own lifetime, instead of just the underlying list. This means cursors cannot yield multiple elements at once.
>
> Cursors always rest between two elements in the list, and index in a logically circular way. To accomadate this, there is a "ghost" non-element that yields None between the head and tail of the List.
>
> When created, cursors start between the ghost and the front of the list. That is, next will yield the front of the list, and prev will yield None. Calling prev again will yield the tail.

Cute, even though we concluded that the whole "sentinel-node" thing was more trouble than it's worth, we're still going to end up with semantics that "pretend" there's a sentinel node so that the cursor can wrap around to the other side of the list.

Skims over my old APIs some more

```
fn splice(&mut self, other: &mut LinkedList<T>)
```

Inserts the entire list's contents right after the cursor.

Oh yeah, this is coming back to me. I wrote this when I was really mad about combinatoric explosion, and was trying to come up with a way for there to only be one copy of each operation. Unfortunately this is... semantically problematic. See, when the user wants to splice one list into another, they might want the cursor to end up *before* the splice or *after it*. The inserted list can be arbitrarily large, so it's a genuine issue for us to only allow for one and expect the user to walk over the entire inserted list!

We're gonna have to rework this design from the ground up after all. What does our Cursor type need? Well it needs to:

- point "between" two elements
- as a nice little feature, keep track of what "index" is next
- update the list itself to modify front/back/len.

How do you point between two elements? Well, you don't. You just point at the "next" element. So, yeah even though we're exposing "cursor goes in-between" semantics, we're really implementing it as "cursor is on", and just pretending everything happens before or after that point.

But there's a reason! The splice use-case wants to let the user choose whether they end up before or after the list, but this is... *horribly* complicated to express with the std API! They have splice_after and splice_before, but neither changes the cursor's position, so really you'd need splice_after_before and splice_after_after...

Wait no I'm being silly. In the std API you can just choose the node you want to end up on, and then use splice_after/before as appropriate.

squints

Wait is the std API actually good.

skims through the code

Ok the std API is actually good.

Alright screw it, we're going to implement the RFC. Or at least the interesting parts of it.

I have my quibbles with some of the terminology std uses, but cursors are always going to be a bit brain-melty: `iter().next_back()` gets you `back()`, so that's good, but then each subsequent `next_back()` is actually bringing you *closer to the front* and indeed, every pointer we follow is a "front" pointer! If I think about this seeming-paradox too much it hurts my brain, so, I can certainly respect going for different terminology to avoid this.

The std API talks about operations before "before" (towards the front) and "after" (towards the back), and instead of `next` and `next_back`, it... calls things `move_next` and `move_prev`. HRM. Ok so they're getting into a bit of the iterator terminology, but at least `next` doesn't evoke front/back, and helps you orient how things behave compared to the iterators.

We can work with this.

Implementing Cursors

Ok so we're only going to bother with std's CursorMut because the immutable version isn't actually interesting. Just like my original design, it has a "ghost" element that contains None to indicate the start/end of the list, and you can "walk over it" to wrap around to the other side of the list. To implement it, we're going to need:

- A pointer to the current node
- A pointer to the list
- The current index

Wait what's the index when we point at the "ghost"?

furrows brow ... checks std ... dislikes std's answer

Ok so quite reasonably `index` on a Cursor returns an `Option<usize>`. The std implementation does a bunch of junk to avoid storing it as an Option but... we're a linked list, it's fine. Also std has the cursor_front/cursor_back stuff which starts the cursor on the front/back elements, which feels intuitive, but then has to do something weird when the list is empty.

You can implement that stuff if you want, but I'm going to cut down on all the repetitive gunk and corner cases and just make a bare `cursor_mut` method that starts at the ghost, and people can use move_next/move_prev to get the one they want (and then you can wrap that up as cursor_front if you really want).

Let's get cracking:

```
pub struct CursorMut<'a, T> {
    cur: Link<T>,
    list: &'a mut LinkedList<T>,
    index: Option<usize>,
}
```

Pretty straight-forward, one field for each item of our bulleted list! Now the `cursor_mut` method:

```
impl<T> LinkedList<T> {
    pub fn cursor_mut(&mut self) -> CursorMut<T> {
        CursorMut {
            list: self,
            cur: None,
            index: None,
        }
    }
}
```

Since we're starting at the ghost, we can just start with everything as None, nice and simple! Next, movement:

```
impl<'a, T> CursorMut<'a, T> {
    pub fn index(&self) -> Option<usize> {
        self.index
    }

    pub fn move_next(&mut self) {
        if let Some(cur) = self.cur {
            unsafe {
                // We're on a real element, go to its next (back)
                self.cur = (*cur.as_ptr()).back;
                if self.cur.is_some() {
                    *self.index.as_mut().unwrap() += 1;
                } else {
                    // We just walked to the ghost, no more index
                    self.index = None;
                }
            }
        } else if !self.list.is_empty() {
            // We're at the ghost, and there is a real front, so move to it!
            self.cur = self.list.front;
            self.index = Some(0)
        } else {
            // We're at the ghost, but that's the only element... do nothing.
        }
    }
}
```

So there's 4 interesting cases:

- The normal case
- The normal case, but we reach the ghost
- The ghost case, where we go to the front of the list
- The ghost case, but the list is empty, so do nothing

move_prev is the exact same logic, but with front/back inverted and the indexing changes inverted:

```
pub fn move_prev(&mut self) {
    if let Some(cur) = self.cur {
        unsafe {
            // We're on a real element, go to its previous (front)
            self.cur = (*cur.as_ptr()).front;
            if self.cur.is_some() {
                *self.index.as_mut().unwrap() -= 1;
            } else {
                // We just walked to the ghost, no more index
                self.index = None;
            }
        }
    } else if !self.list.is_empty() {
        // We're at the ghost, and there is a real back, so move to it!
        self.cur = self.list.back;
        self.index = Some(self.list.len - 1)
    } else {
        // We're at the ghost, but that's the only element... do nothing.
    }
}
```

Next let's add some methods to look at the elements around the cursor: current, peek_next, and peek_prev. **A Very Important Note:** these methods must borrow our cursor by `&mut self`, and the results must be tied to that borrow. We cannot let the user get multiple copies of a mutable reference, and we cannot let them use any of our insert/remove/split/splice APIs while holding onto such a reference!

Thankfully, this is the default assumption rust makes when you use lifetime elision, so, we will just do the right thing by default!

```
pub fn current(&mut self) -> Option<&mut T> {
    unsafe {
        self.cur.map(|node| &mut (*node.as_ptr()).elem)
    }
}

pub fn peek_next(&mut self) -> Option<&mut T> {
    unsafe {
        self.cur
            .and_then(|node| (*node.as_ptr()).back)
            .map(|node| &mut (*node.as_ptr()).elem)
    }
}

pub fn peek_prev(&mut self) -> Option<&mut T> {
    unsafe {
        self.cur
            .and_then(|node| (*node.as_ptr()).front)
            .map(|node| &mut (*node.as_ptr()).elem)
    }
}
```

Head empty, Option methods and (omitted) compiler errors do all thinking now. I was skeptical about the `Option<NonNull>` stuff, but, god damn it really just lets me autopilot this code. I've spent way too much time writing array-based collections where you never get to use Option, wow this is nice! (`(*node.as_ptr())` is still miserable but, that's just Rust's raw pointers for you...)

Next we have a choice: we can either jump right to split and splice, the entire point of these APIs, or we can take a baby-step with single element insert/remove. I have a feeling we're just going to want to implement insert/remove in terms of split and splice so... let's just do those first and see where the cards fall (genuinely have no idea as I type this).

Split

First up, split_before and split_after, which return everything before/after the current element as a LinkedList (stopping at the ghost element, unless you're at the ghost, in which case we just return the whole List and the cursor now points to an empty list):

squints ok this one is actually some non-trivial logic so we're going to have to talk it out one step at a time.

I see 4 potentially interesting cases for split_before:

- The normal case
- The normal case, but prev is the ghost
- The ghost case, where we return the whole list and become empty
- The ghost case, but the list is empty, so do nothing and return the empty list

Let's start with the corner cases. The third case I believe is just

```
mem::replace(self.list, LinkedList::new())
```

Right? We become empty, we return the whole list, and our fields were already None, so nothing to update. Nice. Oh hey, this also Does The Right Thing on the fourth case too!

So now the normal cases... ok I'm going to need some ASCII diagrams for this. In the most general case, we have something like this:

```
list.front -> A <-> B <-> C <-> D <- list.back
                          ^
                         cur
```

And we want to produce this:

```
list.front -> C <-> D <- list.back
              ^
             cur

return.front -> A <-> B <- return.back
```

So we need to break the link between cur and prev, and... god so much needs to change. Ok I just need to break this up into steps so I can convince myself it makes sense. This will be a bit over-verbose but I can at least make sense of it:

```
pub fn split_before(&mut self) -> LinkedList<T> {
    if let Some(cur) = self.cur {
        // We are pointing at a real element, so the list is non-empty.
        unsafe {
            // Current state
            let old_len = self.list.len;
            let old_idx = self.index.unwrap();
            let prev = (*cur.as_ptr()).front;

            // What self will become
            let new_len = old_len - old_idx;
            let new_front = self.cur;
            let new_back = self.list.back;
            let new_idx = Some(0);

            // What the output will become
            let output_len = old_len - new_len;
            let output_front = self.list.front;
            let output_back = prev;

            // Break the links between cur and prev
            if let Some(prev) = prev {
                (*cur.as_ptr()).front = None;
                (*prev.as_ptr()).back = None;
            }

            // Produce the result:
            self.list.len = new_len;
            self.list.front = new_front;
            self.list.back = new_back;
            self.index = new_idx;

            LinkedList {
                front: output_front,
                back: output_back,
                len: output_len,
                _boo: PhantomData,
            }
        }
    } else {
        // We're at the ghost, just replace our list with an empty one.
        // No other state needs to be changed.
        std::mem::replace(self.list, LinkedList::new())
    }
}
```

Note that this if-let is handling the "normal case, but prev is the ghost" situation:

```
if let Some(prev) = prev {
    (*cur.as_ptr()).front = None;
    (*prev.as_ptr()).back = None;
}
```

If *you* want to, you can squash that all together and apply optimizations like:

- fold the two accesses to `(*cur.as_ptr()).front` as just `(*cur.as_ptr()).front.take()`
- note that new_back is a noop, and just remove both

As far as I can tell, everything else just incidentally Does The Right Thing otherwise. We'll see when we write tests! (copy-paste to make split_after)

I am done Making Mistakes and I am just going to try to write the most foolproof code I can. This is how I *actually* write collections: just break things down into trivial steps and cases until it can fit in my

head and seems foolproof. Then write a ton of tests until I'm convinced I didn't manage to mess it up still.

Because most of the collections work I've done is *extremely unsafe* I don't generally get to rely on the compiler catching mistakes, and miri didn't exist back in the day! So I just need to squint at a problem until my head hurts and try my hardest to Never Ever Ever Make A Mistake.

Don't write Unsafe Rust Code! Safe Rust is so much better!!!!

Splice

Just one more boss to fight, splice_before and splice_after, which I expect to be the corner-casiest one of them all. The two functions *take in* a LinkedList and grafts its contents into outrs. Our list could be empty, their list could be empty, we've got ghosts to deal with... *sigh* let's just take it one step at a time with splice_before.

- If their list is empty, we don't need to do anything.
- If our list is empty, then our list just becomes their list.
- If we're pointing at the ghost, then this appends to the back (change list.back)
- If we're pointing at the first element (0), this this appends to the front (change list.front)
- In the general case, we do a whole lot of pointer fuckery.

The general case is this:

```
input.front -> 1 <-> 2 <- input.back

 list.front -> A <-> B <-> C <- list.back
                     ^
                    cur
```

Becoming this:

```
list.front -> A <-> 1 <-> 2 <-> B <-> C <- list.back
```

Ok? Ok. Let's write that out... *TAKES A HUGE BREATH AND PLUNGES IN*:

```
    pub fn splice_before(&mut self, mut input: LinkedList<T>) {
        unsafe {
            if input.is_empty() {
                // Input is empty, do nothing.
            } else if let Some(cur) = self.cur {
                if let Some(0) = self.index {
                    // We're appending to the front, see append to back
                    (*cur.as_ptr()).front = input.back.take();
                    (*input.back.unwrap().as_ptr()).back = Some(cur);
                    self.list.front = input.front.take();

                    // Index moves forward by input length
                    *self.index.as_mut().unwrap() += input.len;
                    self.list.len += input.len;
                    input.len = 0;
                } else {
                    // General Case, no boundaries, just internal fixups
                    let prev = (*cur.as_ptr()).front.unwrap();
                    let in_front = input.front.take().unwrap();
                    let in_back = input.back.take().unwrap();

                    (*prev.as_ptr()).back = Some(in_front);
                    (*in_front.as_ptr()).front = Some(prev);
                    (*cur.as_ptr()).front = Some(in_back);
                    (*in_back.as_ptr()).back = Some(cur);

                    // Index moves forward by input length
                    *self.index.as_mut().unwrap() += input.len;
                    self.list.len += input.len;
                    input.len = 0;
                }
            } else if let Some(back) = self.list.back {
                // We're on the ghost but non-empty, append to the back
                // We can either `take` the input's pointers or `mem::forget`
                // it. Using take is more responsible in case we do custom
                // allocators or something that also needs to be cleaned up!
                (*back.as_ptr()).back = input.front.take();
                (*input.front.unwrap().as_ptr()).front = Some(back);
                self.list.back = input.back.take();
                self.list.len += input.len;
                // Not necessary but Polite To Do
                input.len = 0;
            } else {
                // We're empty, become the input, remain on the ghost
                *self.list = input;
            }
        }
    }
```

Ok this one is genuinely horrendous, and really is feeling that `Option<NonNull>` pain now. But there's a lot of cleanups we can do. For one, we can pull this code out to the very end, because we always want to do it. I don't *love* (although sometimes it's a noop, and setting `input.len` is more a matter of paranoia about future extensions to the code):

```
self.list.len += input.len;
input.len = 0;
```

Use of moved value: `input`

Ah, right, in the "we're empty" case we're moving the list. Let's replace that with a swap:

```
// We're empty, become the input, remain on the ghost
std::mem::swap(self.list, &mut input);
```

In this case the writes will be pointless, but, they still work (we could probably also early-return in this branch to appease the compiler).

This unwrap is just a consequence of me thinking about the cases backwards, and can be fixed by making the if-let ask the right question:

```
if let Some(0) = self.index {

} else {
    let prev = (*cur.as_ptr()).front.unwrap();
}
```

Adjusting the index is duplicated inside the branches, so can also be hoisted out:

```

*self.index.as_mut().unwrap() += input.len;
```

Ok, putting that all together we get this:

```
if input.is_empty() {
    // Input is empty, do nothing.
} else if let Some(cur) = self.cur {
    // Both lists are non-empty
    if let Some(prev) = (*cur.as_ptr()).front {
        // General Case, no boundaries, just internal fixups
        let in_front = input.front.take().unwrap();
        let in_back = input.back.take().unwrap();

        (*prev.as_ptr()).back = Some(in_front);
        (*in_front.as_ptr()).front = Some(prev);
        (*cur.as_ptr()).front = Some(in_back);
        (*in_back.as_ptr()).back = Some(cur);
    } else {
        // We're appending to the front, see append to back below
        (*cur.as_ptr()).front = input.back.take();
        (*input.back.unwrap().as_ptr()).back = Some(cur);
        self.list.front = input.front.take();
    }
    // Index moves forward by input length
    *self.index.as_mut().unwrap() += input.len;
} else if let Some(back) = self.list.back {
    // We're on the ghost but non-empty, append to the back
    // We can either `take` the input's pointers or `mem::forget`
    // it. Using take is more responsible in case we do custom
    // allocators or something that also needs to be cleaned up!
    (*back.as_ptr()).back = input.front.take();
    (*input.front.unwrap().as_ptr()).front = Some(back);
    self.list.back = input.back.take();

} else {
    // We're empty, become the input, remain on the ghost
    std::mem::swap(self.list, &mut input);
}

self.list.len += input.len;
// Not necessary but Polite To Do
input.len = 0;

// Input dropped here
```

Alright this still sucks, but mostly because of -- nope ok just spotted a bug:

```
    (*back.as_ptr()).back = input.front.take();
    (*input.front.unwrap().as_ptr()).front = Some(back);
```

We `take` input.front and then unwrap it on the next line! *sigh* and we do the same thing in the equivalent mirror case. We would have caught this instantly in tests, but, we're trying to be Perfect now, and I'm just kinda doing this live, and this is the exact moment where I saw it. This is what I get for not being my usual tedious self and doing things in phases. More explicit!

```
// We can either `take` the input's pointers or `mem::forget`
// it. Using `take` is more responsible in case we ever do custom
// allocators or something that also needs to be cleaned up!
if input.is_empty() {
    // Input is empty, do nothing.
} else if let Some(cur) = self.cur {
    // Both lists are non-empty
    let in_front = input.front.take().unwrap();
    let in_back = input.back.take().unwrap();

    if let Some(prev) = (*cur.as_ptr()).front {
        // General Case, no boundaries, just internal fixups
        (*prev.as_ptr()).back = Some(in_front);
        (*in_front.as_ptr()).front = Some(prev);
        (*cur.as_ptr()).front = Some(in_back);
        (*in_back.as_ptr()).back = Some(cur);
    } else {
        // No prev, we're appending to the front
        (*cur.as_ptr()).front = Some(in_back);
        (*in_back.as_ptr()).back = Some(cur);
        self.list.front = Some(in_front);
    }
    // Index moves forward by input length
    *self.index.as_mut().unwrap() += input.len;
} else if let Some(back) = self.list.back {
    // We're on the ghost but non-empty, append to the back
    let in_front = input.front.take().unwrap();
    let in_back = input.back.take().unwrap();

    (*back.as_ptr()).back = Some(in_front);
    (*in_front.as_ptr()).front = Some(back);
    self.list.back = Some(in_back);
} else {
    // We're empty, become the input, remain on the ghost
    std::mem::swap(self.list, &mut input);
}

self.list.len += input.len;
// Not necessary but Polite To Do
input.len = 0;

// Input dropped here
```

Alright now this, this I can tolerate. The only complaints I have are that we don't dedupe in_front/in_back (probably we could rejig our conditions but eh whatever). Really this is basically what you would write in C but with `Option<NonNull>` gunk making it tedious. I can live with that. Well no we should just make raw pointers better for this stuff. But, out of scope for this book.

Anyway, I am absolutely exhausted after that, so, `insert` and `remove` and all the other APIs can be left as an excercise to the reader.

Here's the final code for our Cursor with my attempt at copy-pasting the combinatorics. Did I get it right? I'll only find out when I write the next chapter and test this monstrosity!

```
pub struct CursorMut<'a, T> {
    list: &'a mut LinkedList<T>,
    cur: Link<T>,
    index: Option<usize>,
}

impl<T> LinkedList<T> {
    pub fn cursor_mut(&mut self) -> CursorMut<T> {
        CursorMut {
            list: self,
            cur: None,
            index: None,
        }
    }
}

impl<'a, T> CursorMut<'a, T> {
    pub fn index(&self) -> Option<usize> {
        self.index
    }

    pub fn move_next(&mut self) {
        if let Some(cur) = self.cur {
            unsafe {
                // We're on a real element, go to its next (back)
                self.cur = (*cur.as_ptr()).back;
                if self.cur.is_some() {
                    *self.index.as_mut().unwrap() += 1;
                } else {
                    // We just walked to the ghost, no more index
                    self.index = None;
                }
            }
        } else if !self.list.is_empty() {
            // We're at the ghost, and there is a real front, so move to it!
            self.cur = self.list.front;
            self.index = Some(0)
        } else {
            // We're at the ghost, but that's the only element... do nothing.
        }
    }

    pub fn move_prev(&mut self) {
        if let Some(cur) = self.cur {
            unsafe {
                // We're on a real element, go to its previous (front)
                self.cur = (*cur.as_ptr()).front;
                if self.cur.is_some() {
                    *self.index.as_mut().unwrap() -= 1;
                } else {
                    // We just walked to the ghost, no more index
                    self.index = None;
                }
            }
        } else if !self.list.is_empty() {
            // We're at the ghost, and there is a real back, so move to it!
            self.cur = self.list.back;
            self.index = Some(self.list.len - 1)
        } else {
            // We're at the ghost, but that's the only element... do nothing.
        }
    }

    pub fn current(&mut self) -> Option<&mut T> {
        unsafe {
            self.cur.map(|node| &mut (*node.as_ptr()).elem)
        }
```

```
    }

    pub fn peek_next(&mut self) -> Option<&mut T> {
        unsafe {
            self.cur
                .and_then(|node| (*node.as_ptr()).back)
                .map(|node| &mut (*node.as_ptr()).elem)
        }
    }

    pub fn peek_prev(&mut self) -> Option<&mut T> {
        unsafe {
            self.cur
                .and_then(|node| (*node.as_ptr()).front)
                .map(|node| &mut (*node.as_ptr()).elem)
        }
    }

    pub fn split_before(&mut self) -> LinkedList<T> {
        // We have this:
        //
        //     list.front -> A <-> B <-> C <-> D <- list.back
        //                                ^
        //                               cur
        //
        //
        // And we want to produce this:
        //
        //     list.front -> C <-> D <- list.back
        //                    ^
        //                   cur
        //
        //
        //    return.front -> A <-> B <- return.back
        //
        if let Some(cur) = self.cur {
            // We are pointing at a real element, so the list is non-empty.
            unsafe {
                // Current state
                let old_len = self.list.len;
                let old_idx = self.index.unwrap();
                let prev = (*cur.as_ptr()).front;

                // What self will become
                let new_len = old_len - old_idx;
                let new_front = self.cur;
                let new_back = self.list.back;
                let new_idx = Some(0);

                // What the output will become
                let output_len = old_len - new_len;
                let output_front = self.list.front;
                let output_back = prev;

                // Break the links between cur and prev
                if let Some(prev) = prev {
                    (*cur.as_ptr()).front = None;
                    (*prev.as_ptr()).back = None;
                }

                // Produce the result:
                self.list.len = new_len;
                self.list.front = new_front;
                self.list.back = new_back;
                self.index = new_idx;

                LinkedList {
                    front: output_front,
```

```
                    back: output_back,
                    len: output_len,
                    _boo: PhantomData,
                }
            }
        } else {
            // We're at the ghost, just replace our list with an empty one.
            // No other state needs to be changed.
            std::mem::replace(self.list, LinkedList::new())
        }
    }

    pub fn split_after(&mut self) -> LinkedList<T> {
        // We have this:
        //
        //     list.front -> A <-> B <-> C <-> D <- list.back
        //                           ^
        //                          cur
        //
        //
        // And we want to produce this:
        //
        //     list.front -> A <-> B <- list.back
        //                           ^
        //                          cur
        //
        //
        //    return.front -> C <-> D <- return.back
        //
        if let Some(cur) = self.cur {
            // We are pointing at a real element, so the list is non-empty.
            unsafe {
                // Current state
                let old_len = self.list.len;
                let old_idx = self.index.unwrap();
                let next = (*cur.as_ptr()).back;

                // What self will become
                let new_len = old_idx + 1;
                let new_back = self.cur;
                let new_front = self.list.front;
                let new_idx = Some(old_idx);

                // What the output will become
                let output_len = old_len - new_len;
                let output_front = next;
                let output_back = self.list.back;

                // Break the links between cur and next
                if let Some(next) = next {
                    (*cur.as_ptr()).back = None;
                    (*next.as_ptr()).front = None;
                }

                // Produce the result:
                self.list.len = new_len;
                self.list.front = new_front;
                self.list.back = new_back;
                self.index = new_idx;

                LinkedList {
                    front: output_front,
                    back: output_back,
                    len: output_len,
                    _boo: PhantomData,
                }
            }
        } else {
```

```
            // We're at the ghost, just replace our list with an empty one.
            // No other state needs to be changed.
            std::mem::replace(self.list, LinkedList::new())
        }
    }

    pub fn splice_before(&mut self, mut input: LinkedList<T>) {
        // We have this:
        //
        // input.front -> 1 <-> 2 <- input.back
        //
        // list.front -> A <-> B <-> C <- list.back
        //                     ^
        //                    cur
        //
        //
        // Becoming this:
        //
        // list.front -> A <-> 1 <-> 2 <-> B <-> C <- list.back
        //                                 ^
        //                                cur
        //
        unsafe {
            // We can either `take` the input's pointers or `mem::forget`
            // it. Using `take` is more responsible in case we ever do custom
            // allocators or something that also needs to be cleaned up!
            if input.is_empty() {
                // Input is empty, do nothing.
            } else if let Some(cur) = self.cur {
                // Both lists are non-empty
                let in_front = input.front.take().unwrap();
                let in_back = input.back.take().unwrap();

                if let Some(prev) = (*cur.as_ptr()).front {
                    // General Case, no boundaries, just internal fixups
                    (*prev.as_ptr()).back = Some(in_front);
                    (*in_front.as_ptr()).front = Some(prev);
                    (*cur.as_ptr()).front = Some(in_back);
                    (*in_back.as_ptr()).back = Some(cur);
                } else {
                    // No prev, we're appending to the front
                    (*cur.as_ptr()).front = Some(in_back);
                    (*in_back.as_ptr()).back = Some(cur);
                    self.list.front = Some(in_front);
                }
                // Index moves forward by input length
                *self.index.as_mut().unwrap() += input.len;
            } else if let Some(back) = self.list.back {
                // We're on the ghost but non-empty, append to the back
                let in_front = input.front.take().unwrap();
                let in_back = input.back.take().unwrap();

                (*back.as_ptr()).back = Some(in_front);
                (*in_front.as_ptr()).front = Some(back);
                self.list.back = Some(in_back);
            } else {
                // We're empty, become the input, remain on the ghost
                std::mem::swap(self.list, &mut input);
            }

            self.list.len += input.len;
            // Not necessary but Polite To Do
            input.len = 0;

            // Input dropped here
        }
    }
```

```
    pub fn splice_after(&mut self, mut input: LinkedList<T>) {
        // We have this:
        //
        // input.front -> 1 <-> 2 <- input.back
        //
        // list.front -> A <-> B <-> C <- list.back
        //                     ^
        //                    cur
        //
        //
        // Becoming this:
        //
        // list.front -> A <-> B <-> 1 <-> 2 <-> C <- list.back
        //                     ^
        //                    cur
        //
        unsafe {
            // We can either `take` the input's pointers or `mem::forget`
            // it. Using `take` is more responsible in case we ever do custom
            // allocators or something that also needs to be cleaned up!
            if input.is_empty() {
                // Input is empty, do nothing.
            } else if let Some(cur) = self.cur {
                // Both lists are non-empty
                let in_front = input.front.take().unwrap();
                let in_back = input.back.take().unwrap();

                if let Some(next) = (*cur.as_ptr()).back {
                    // General Case, no boundaries, just internal fixups
                    (*next.as_ptr()).front = Some(in_back);
                    (*in_back.as_ptr()).back = Some(next);
                    (*cur.as_ptr()).back = Some(in_front);
                    (*in_front.as_ptr()).front = Some(cur);
                } else {
                    // No next, we're appending to the back
                    (*cur.as_ptr()).back = Some(in_front);
                    (*in_front.as_ptr()).front = Some(cur);
                    self.list.back = Some(in_back);
                }
                // Index doesn't change
            } else if let Some(front) = self.list.front {
                // We're on the ghost but non-empty, append to the front
                let in_front = input.front.take().unwrap();
                let in_back = input.back.take().unwrap();

                (*front.as_ptr()).front = Some(in_back);
                (*in_back.as_ptr()).back = Some(front);
                self.list.front = Some(in_front);
            } else {
                // We're empty, become the input, remain on the ghost
                std::mem::swap(self.list, &mut input);
            }

            self.list.len += input.len;
            // Not necessary but Polite To Do
            input.len = 0;

            // Input dropped here
        }
    }
}
```

Testing Cursors

Time to find out how many horribly embarassing mistakes I made in the previous section!

Oh god we made our API unlike both std and the old impl. Alright well I'm just gonna hastily cobble together something from both of them. Yeah let's "borrow" these tests from std:

```
    #[test]
    fn test_cursor_move_peek() {
        let mut m: LinkedList<u32> = LinkedList::new();
        m.extend([1, 2, 3, 4, 5, 6]);
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        assert_eq!(cursor.current(), Some(&mut 1));
        assert_eq!(cursor.peek_next(), Some(&mut 2));
        assert_eq!(cursor.peek_prev(), None);
        assert_eq!(cursor.index(), Some(0));
        cursor.move_prev();
        assert_eq!(cursor.current(), None);
        assert_eq!(cursor.peek_next(), Some(&mut 1));
        assert_eq!(cursor.peek_prev(), Some(&mut 6));
        assert_eq!(cursor.index(), None);
        cursor.move_next();
        cursor.move_next();
        assert_eq!(cursor.current(), Some(&mut 2));
        assert_eq!(cursor.peek_next(), Some(&mut 3));
        assert_eq!(cursor.peek_prev(), Some(&mut 1));
        assert_eq!(cursor.index(), Some(1));

        let mut cursor = m.cursor_mut();
        cursor.move_prev();
        assert_eq!(cursor.current(), Some(&mut 6));
        assert_eq!(cursor.peek_next(), None);
        assert_eq!(cursor.peek_prev(), Some(&mut 5));
        assert_eq!(cursor.index(), Some(5));
        cursor.move_next();
        assert_eq!(cursor.current(), None);
        assert_eq!(cursor.peek_next(), Some(&mut 1));
        assert_eq!(cursor.peek_prev(), Some(&mut 6));
        assert_eq!(cursor.index(), None);
        cursor.move_prev();
        cursor.move_prev();
        assert_eq!(cursor.current(), Some(&mut 5));
        assert_eq!(cursor.peek_next(), Some(&mut 6));
        assert_eq!(cursor.peek_prev(), Some(&mut 4));
        assert_eq!(cursor.index(), Some(4));
    }

    #[test]
    fn test_cursor_mut_insert() {
        let mut m: LinkedList<u32> = LinkedList::new();
        m.extend([1, 2, 3, 4, 5, 6]);
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        cursor.splice_before(Some(7).into_iter().collect());
        cursor.splice_after(Some(8).into_iter().collect());
        // check_links(&m);
        assert_eq!(m.iter().cloned().collect::<Vec<_>>(), &[7, 1, 8, 2, 3, 4, 5, 6]);
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        cursor.move_prev();
        cursor.splice_before(Some(9).into_iter().collect());
        cursor.splice_after(Some(10).into_iter().collect());
        check_links(&m);
        assert_eq!(m.iter().cloned().collect::<Vec<_>>(), &[10, 7, 1, 8, 2, 3, 4, 5, 6,
9]);

        /* remove_current not impl'd
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        cursor.move_prev();
        assert_eq!(cursor.remove_current(), None);
        cursor.move_next();
        cursor.move_next();
```

```
        assert_eq!(cursor.remove_current(), Some(7));
        cursor.move_prev();
        cursor.move_prev();
        cursor.move_prev();
        assert_eq!(cursor.remove_current(), Some(9));
        cursor.move_next();
        assert_eq!(cursor.remove_current(), Some(10));
        check_links(&m);
        assert_eq!(m.iter().cloned().collect::<Vec<_>>(), &[1, 8, 2, 3, 4, 5, 6]);
        */

        let mut cursor = m.cursor_mut();
        cursor.move_next();
        let mut p: LinkedList<u32> = LinkedList::new();
        p.extend([100, 101, 102, 103]);
        let mut q: LinkedList<u32> = LinkedList::new();
        q.extend([200, 201, 202, 203]);
        cursor.splice_after(p);
        cursor.splice_before(q);
        check_links(&m);
        assert_eq!(
            m.iter().cloned().collect::<Vec<_>>(),
            &[200, 201, 202, 203, 1, 100, 101, 102, 103, 8, 2, 3, 4, 5, 6]
        );
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        cursor.move_prev();
        let tmp = cursor.split_before();
        assert_eq!(m.into_iter().collect::<Vec<_>>(), &[]);
        m = tmp;
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        cursor.move_next();
        cursor.move_next();
        cursor.move_next();
        cursor.move_next();
        cursor.move_next();
        cursor.move_next();
        let tmp = cursor.split_after();
        assert_eq!(tmp.into_iter().collect::<Vec<_>>(), &[102, 103, 8, 2, 3, 4, 5, 6]);
        check_links(&m);
        assert_eq!(m.iter().cloned().collect::<Vec<_>>(), &[200, 201, 202, 203, 1, 100,
101]);
    }

    fn check_links<T>(_list: &LinkedList<T>) {
        // would be good to do this!
    }
```

Moment of truth!

```
cargo test

   Compiling linked-list v0.0.3
    Finished test [unoptimized + debuginfo] target(s) in 1.03s
     Running unittests src\lib.rs

running 14 tests
test test::test_basic_front ... ok
test test::test_basic ... ok
test test::test_debug ... ok
test test::test_iterator_mut_double_end ... ok
test test::test_ord ... ok
test test::test_cursor_move_peek ... FAILED
test test::test_cursor_mut_insert ... FAILED
test test::test_iterator ... ok
test test::test_mut_iter ... ok
test test::test_eq ... ok
test test::test_rev_iter ... ok
test test::test_iterator_double_end ... ok
test test::test_hashmap ... ok
test test::test_ord_nan ... ok

failures:

---- test::test_cursor_move_peek stdout ----
thread 'test::test_cursor_move_peek' panicked at 'assertion failed: `(left == right)`
  left: `None`,
 right: `Some(1)`', src\lib.rs:1079:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

---- test::test_cursor_mut_insert stdout ----
thread 'test::test_cursor_mut_insert' panicked at 'assertion failed: `(left == right)`
  left: `[200, 201, 202, 203, 10, 100, 101, 102, 103, 7, 1, 8, 2, 3, 4, 5, 6, 9]`,
 right: `[200, 201, 202, 203, 1, 100, 101, 102, 103, 8, 2, 3, 4, 5, 6]`',
src\lib.rs:1153:9


failures:
    test::test_cursor_move_peek
    test::test_cursor_mut_insert

test result: FAILED. 12 passed; 2 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

I'll admit, I had some hubris here and was hoping I nailed it. This is why we write tests (but maybe I just did a bad job of porting the tests..?).

What's the first failure?

```
let mut m: LinkedList<u32> = LinkedList::new();
m.extend([1, 2, 3, 4, 5, 6]);
let mut cursor = m.cursor_mut();

cursor.move_next();
assert_eq!(cursor.current(), Some(&mut 1));
assert_eq!(cursor.peek_next(), Some(&mut 2));
assert_eq!(cursor.peek_prev(), None);
assert_eq!(cursor.index(), Some(0));

cursor.move_prev();
assert_eq!(cursor.current(), None);
assert_eq!(cursor.peek_next(), Some(&mut 1)); // DIES HERE
```

Geez I really messed up some basic functionality. Wait,

> Head empty, Option methods and (omitted) compiler errors do all thinking now.

Well I am nothing if not honest.

```
pub fn peek_next(&mut self) -> Option<&mut T> {
    unsafe {
        self.cur
            .and_then(|node| (*node.as_ptr()).back)
            .map(|node| &mut (*node.as_ptr()).elem)
    }
}
```

...Yeah this is just wrong. If `self.cur` is None, we aren't just supposed to give up, we need to check `self.list.front` too, because we're on the ghost! So we just need to add an or_else to the chain:

```
pub fn peek_next(&mut self) -> Option<&mut T> {
    unsafe {
        self.cur
            .and_then(|node| (*node.as_ptr()).back)
            .or_else(|| self.list.front)
            .map(|node| &mut (*node.as_ptr()).elem)
    }
}

pub fn peek_prev(&mut self) -> Option<&mut T> {
    unsafe {
        self.cur
            .and_then(|node| (*node.as_ptr()).front)
            .or_else(|| self.list.back)
            .map(|node| &mut (*node.as_ptr()).elem)
    }
}
```

Did that fix it?

```
---- test::test_cursor_move_peek stdout ----
thread 'test::test_cursor_move_peek' panicked at 'assertion failed: `(left == right)`
  left: `Some(6)`,
 right: `None`', src\lib.rs:1078:9
```

Wait now it's wrong *further back*. Ok I need to stop head-emptying peek because apparently it's a lot harder than I was willing to give it credit for. Just trying to blindly chain these cases is a disaster, let's have a proper if for the cases of ghost vs not:

```
pub fn peek_next(&mut self) -> Option<&mut T> {
    unsafe {
        let next = if let Some(cur) = self.cur {
            // Normal case, try to follow the cur node's back pointer
            (*cur.as_ptr()).back
        } else {
            // Ghost case, try to use the list's front pointer
            self.list.front
        };

        // Yield the element if the next node exists
        next.map(|node| &mut (*node.as_ptr()).elem)
    }
}

pub fn peek_prev(&mut self) -> Option<&mut T> {
    unsafe {
        let prev = if let Some(cur) = self.cur {
            // Normal case, try to follow the cur node's front pointer
            (*cur.as_ptr()).front
        } else {
            // Ghost case, try to use the list's back pointer
            self.list.back
        };

        // Yield the element if the prev node exists
        prev.map(|node| &mut (*node.as_ptr()).elem)
    }
}
```

Feelin' confident about this one!

```
failures:

---- test::test_cursor_mut_insert stdout ----
thread 'test::test_cursor_mut_insert' panicked at 'assertion failed: `(left == right)`
  left: `[200, 201, 202, 203, 10, 100, 101, 102, 103, 7, 1, 8, 2, 3, 4, 5, 6, 9]`,
 right: `[200, 201, 202, 203, 1, 100, 101, 102, 103, 8, 2, 3, 4, 5, 6]`',
src\lib.rs:1168:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    test::test_cursor_mut_insert

test result: FAILED. 13 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Yesss. Ok one more failure to go... oh.

Did you notice the part where I commented out some code for testing remove_current? Yeah I wasn't paying attention to the fact that this test is stateful. Let's just create a new list with the state the remove_current part would have left us in:

```
let mut m: LinkedList<u32> = LinkedList::new();
m.extend([1, 8, 2, 3, 4, 5, 6]);
```

```
 cargo test
   Compiling linked-list v0.0.3
    Finished test [unoptimized + debuginfo] target(s) in 0.70s
     Running unittests src\lib.rs

running 14 tests
test test::test_basic_front ... ok
test test::test_basic ... ok
test test::test_cursor_move_peek ... ok
test test::test_eq ... ok
test test::test_cursor_mut_insert ... ok
test test::test_iterator ... ok
test test::test_iterator_double_end ... ok
test test::test_ord_nan ... ok
test test::test_mut_iter ... ok
test test::test_hashmap ... ok
test test::test_debug ... ok
test test::test_ord ... ok
test test::test_iterator_mut_double_end ... ok
test test::test_rev_iter ... ok

test result: ok. 14 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 0.00s

   Doc-tests linked-list

running 1 test
test src\lib.rs - assert_properties::iter_mut_invariant (line 803) - compile fail ...
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.12s
```

Heyyyy look at thaaat... ok now I'm getting paranoid. Let's properly fill in check_links and test it under miri:

```
fn check_links<T: Eq + std::fmt::Debug>(list: &LinkedList<T>) {
    let from_front: Vec<_> = list.iter().collect();
    let from_back: Vec<_> = list.iter().rev().collect();
    let re_reved: Vec<_> = from_back.into_iter().rev().collect();

    assert_eq!(from_front, re_reved);
}
```

Is this the best way to do this? No. Is it fine? Yes.

```
$env:MIRIFLAGS="-Zmiri-tag-raw-pointers"
cargo miri test
   Compiling linked-list v0.0.3
    Finished test [unoptimized + debuginfo] target(s) in 0.25s
     Running unittests src\lib.rs

running 14 tests
test test::test_basic ... ok
test test::test_basic_front ... ok
test test::test_cursor_move_peek ... ok
test test::test_cursor_mut_insert ... ok
test test::test_debug ... ok
test test::test_eq ... ok
test test::test_hashmap ... ok
test test::test_iterator ... ok
test test::test_iterator_double_end ... ok
test test::test_iterator_mut_double_end ... ok
test test::test_mut_iter ... ok
test test::test_ord ... ok
test test::test_ord_nan ... ok
test test::test_rev_iter ... ok

test result: ok. 14 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests linked-list

running 1 test
test src\lib.rs - assert_properties::iter_mut_invariant (line 803) - compile fail ...
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.10s
```

DONE.

Done.

We did it. We made a god damn production-quality LinkedList, with basically all the same functionality as the one in std. Are we missing little convenience methods here and there? Absolutely. Will I add them into the final published version of the crate? Probably!

But, I am, So Very Tired.

So. We win.

Wait fuck. We're being production quality. Ok one last final boss: clippy.

```
cargo clippy

cargo clippy
    Checking linked-list v0.0.3 (C:\Users\ninte\dev\contain\linked-list)
warning: redundant pattern matching, consider using `is_some()`
   --> src\lib.rs:189:19
    |
189 |         while let Some(_) = self.pop_front() { }
    |         ----------^^^^^^^------------------- help: try this: `while
self.pop_front().is_some()`
    |
    = note: `#[warn(clippy::redundant_pattern_matching)]` on by default
    = note: this will change drop order of the result, as well as all temporaries
    = note: add `#[allow(clippy::redundant_pattern_matching)]` if this is important
    = help: for further information visit https://rust-lang.github.io/rust-
clippy/master/index.html#redundant_pattern_matching

warning: method `into_iter` can be confused for the standard trait method
`std::iter::IntoIterator::into_iter`
   --> src\lib.rs:210:5
    |
210 | /     pub fn into_iter(self) -> IntoIter<T> {
211 | |         IntoIter {
212 | |             list: self
213 | |         }
214 | |     }
    | |_____^
    |
    = note: `#[warn(clippy::should_implement_trait)]` on by default
    = help: consider implementing the trait `std::iter::IntoIterator` or choosing a
less ambiguous method name
    = help: for further information visit https://rust-lang.github.io/rust-
clippy/master/index.html#should_implement_trait

warning: redundant pattern matching, consider using `is_some()`
   --> src\lib.rs:228:19
    |
228 |         while let Some(_) = self.pop_front() { }
    |         ----------^^^^^^^------------------- help: try this: `while
self.pop_front().is_some()`
    |
    = note: this will change drop order of the result, as well as all temporaries
    = note: add `#[allow(clippy::redundant_pattern_matching)]` if this is important
    = help: for further information visit https://rust-lang.github.io/rust-
clippy/master/index.html#redundant_pattern_matching

warning: re-implementing `PartialEq::ne` is unnecessary
   --> src\lib.rs:275:5
    |
275 | /     fn ne(&self, other: &Self) -> bool {
276 | |         self.len() != other.len() || self.iter().ne(other)
277 | |     }
    | |_____^
    |
    = note: `#[warn(clippy::partialeq_ne_impl)]` on by default
    = help: for further information visit https://rust-lang.github.io/rust-
clippy/master/index.html#partialeq_ne_impl

warning: `linked-list` (lib) generated 4 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 0.29s
```

Alright clippy, let's do this.

Complaint 1 (and 3): we use `while let Some(_) =` instead of `while .is_some()`. The loop is empty so this truly doesn't matter but ok fine, clippy, I'll do things your way.

Complaint 2: We have an actual inherent into_iter method. Wait, what *checks std* ok, point to clippy. IntoIterator is in the prelude (and basically a lang item) so, we don't need an inherent version too.

Complaint 4: we copied a weird cargocult from std. *shrug* fine I'll remove it.

```
cargo clippy
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
```

Nice. Just one last thing to do before calling it production quality: fmt.

```
cargo fmt
```

...yeah it added some newlines and removed some trailing whitespace. Nothing interesting.

WE ARE NOW TRULY FINALLY DONE!!!!!!!!!!!!!!!!!!!!!

Final Code

I can't believe I actually just made you sit through me actually reimplementing std::collections::LinkedList from scratch, with all the fiddly little pedantry and mistakes I made along the way.

I did it, the book is done, I can finally rest.

Alright, here's all 1200 lines of our complete rewrite of in all of its glory. This should be the same text as this commit.

I'll put some polish and docs back on and publish 0.1.0 later.

```
use std::cmp::Ordering;
use std::fmt::{self, Debug};
use std::hash::{Hash, Hasher};
use std::iter::FromIterator;
use std::marker::PhantomData;
use std::ptr::NonNull;

pub struct LinkedList<T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<T>,
}

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

struct Node<T> {
    front: Link<T>,
    back: Link<T>,
    elem: T,
}

pub struct Iter<'a, T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<&'a T>,
}

pub struct IterMut<'a, T> {
    front: Link<T>,
    back: Link<T>,
    len: usize,
    _boo: PhantomData<&'a mut T>,
}

pub struct IntoIter<T> {
    list: LinkedList<T>,
}

pub struct CursorMut<'a, T> {
    list: &'a mut LinkedList<T>,
    cur: Link<T>,
    index: Option<usize>,
}

impl<T> LinkedList<T> {
    pub fn new() -> Self {
        Self {
            front: None,
            back: None,
            len: 0,
            _boo: PhantomData,
        }
    }

    pub fn push_front(&mut self, elem: T) {
        // SAFETY: it's a linked-list, what do you want?
        unsafe {
            let new = NonNull::new_unchecked(Box::into_raw(Box::new(Node {
                front: None,
                back: None,
                elem,
            })));
            if let Some(old) = self.front {
                // Put the new front before the old one
```

```
                (*old.as_ptr()).front = Some(new);
                (*new.as_ptr()).back = Some(old);
            } else {
                // If there's no front, then we're the empty list and need
                // to set the back too.
                self.back = Some(new);
            }
            // These things always happen!
            self.front = Some(new);
            self.len += 1;
        }
    }

    pub fn push_back(&mut self, elem: T) {
        // SAFETY: it's a linked-list, what do you want?
        unsafe {
            let new = NonNull::new_unchecked(Box::into_raw(Box::new(Node {
                back: None,
                front: None,
                elem,
            })));
            if let Some(old) = self.back {
                // Put the new back before the old one
                (*old.as_ptr()).back = Some(new);
                (*new.as_ptr()).front = Some(old);
            } else {
                // If there's no back, then we're the empty list and need
                // to set the front too.
                self.front = Some(new);
            }
            // These things always happen!
            self.back = Some(new);
            self.len += 1;
        }
    }

    pub fn pop_front(&mut self) -> Option<T> {
        unsafe {
            // Only have to do stuff if there is a front node to pop.
            self.front.map(|node| {
                // Bring the Box back to life so we can move out its value and
                // Drop it (Box continues to magically understand this for us).
                let boxed_node = Box::from_raw(node.as_ptr());
                let result = boxed_node.elem;

                // Make the next node into the new front.
                self.front = boxed_node.back;
                if let Some(new) = self.front {
                    // Cleanup its reference to the removed node
                    (*new.as_ptr()).front = None;
                } else {
                    // If the front is now null, then this list is now empty!
                    self.back = None;
                }

                self.len -= 1;
                result
                // Box gets implicitly freed here, knows there is no T.
            })
        }
    }

    pub fn pop_back(&mut self) -> Option<T> {
        unsafe {
            // Only have to do stuff if there is a back node to pop.
            self.back.map(|node| {
                // Bring the Box front to life so we can move out its value and
                // Drop it (Box continues to magically understand this for us).
```

```
                let boxed_node = Box::from_raw(node.as_ptr());
                let result = boxed_node.elem;

                // Make the next node into the new back.
                self.back = boxed_node.front;
                if let Some(new) = self.back {
                    // Cleanup its reference to the removed node
                    (*new.as_ptr()).back = None;
                } else {
                    // If the back is now null, then this list is now empty!
                    self.front = None;
                }

                self.len -= 1;
                result
                // Box gets implicitly freed here, knows there is no T.
            })
        }
    }

    pub fn front(&self) -> Option<&T> {
        unsafe { self.front.map(|node| &(*node.as_ptr()).elem) }
    }

    pub fn front_mut(&mut self) -> Option<&mut T> {
        unsafe { self.front.map(|node| &mut (*node.as_ptr()).elem) }
    }

    pub fn back(&self) -> Option<&T> {
        unsafe { self.back.map(|node| &(*node.as_ptr()).elem) }
    }

    pub fn back_mut(&mut self) -> Option<&mut T> {
        unsafe { self.back.map(|node| &mut (*node.as_ptr()).elem) }
    }

    pub fn len(&self) -> usize {
        self.len
    }

    pub fn is_empty(&self) -> bool {
        self.len == 0
    }

    pub fn clear(&mut self) {
        // Oh look it's drop again
        while self.pop_front().is_some() {}
    }

    pub fn iter(&self) -> Iter<T> {
        Iter {
            front: self.front,
            back: self.back,
            len: self.len,
            _boo: PhantomData,
        }
    }

    pub fn iter_mut(&mut self) -> IterMut<T> {
        IterMut {
            front: self.front,
            back: self.back,
            len: self.len,
            _boo: PhantomData,
        }
    }

    pub fn cursor_mut(&mut self) -> CursorMut<T> {
```

```
        CursorMut {
            list: self,
            cur: None,
            index: None,
        }
    }
}

impl<T> Drop for LinkedList<T> {
    fn drop(&mut self) {
        // Pop until we have to stop
        while self.pop_front().is_some() {}
    }
}

impl<T> Default for LinkedList<T> {
    fn default() -> Self {
        Self::new()
    }
}

impl<T: Clone> Clone for LinkedList<T> {
    fn clone(&self) -> Self {
        let mut new_list = Self::new();
        for item in self {
            new_list.push_back(item.clone());
        }
        new_list
    }
}

impl<T> Extend<T> for LinkedList<T> {
    fn extend<I: IntoIterator<Item = T>>(&mut self, iter: I) {
        for item in iter {
            self.push_back(item);
        }
    }
}

impl<T> FromIterator<T> for LinkedList<T> {
    fn from_iter<I: IntoIterator<Item = T>>(iter: I) -> Self {
        let mut list = Self::new();
        list.extend(iter);
        list
    }
}

impl<T: Debug> Debug for LinkedList<T> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.debug_list().entries(self).finish()
    }
}

impl<T: PartialEq> PartialEq for LinkedList<T> {
    fn eq(&self, other: &Self) -> bool {
        self.len() == other.len() && self.iter().eq(other)
    }
}

impl<T: Eq> Eq for LinkedList<T> {}

impl<T: PartialOrd> PartialOrd for LinkedList<T> {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        self.iter().partial_cmp(other)
    }
}

impl<T: Ord> Ord for LinkedList<T> {
```

```
    fn cmp(&self, other: &Self) -> Ordering {
        self.iter().cmp(other)
    }
}

impl<T: Hash> Hash for LinkedList<T> {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.len().hash(state);
        for item in self {
            item.hash(state);
        }
    }
}

impl<'a, T> IntoIterator for &'a LinkedList<T> {
    type IntoIter = Iter<'a, T>;
    type Item = &'a T;

    fn into_iter(self) -> Self::IntoIter {
        self.iter()
    }
}

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

    fn next(&mut self) -> Option<Self::Item> {
        // While self.front == self.back is a tempting condition to check here,
        // it won't do the right for yielding the last element! That sort of
        // thing only works for arrays because of "one-past-the-end" pointers.
        if self.len > 0 {
            // We could unwrap front, but this is safer and easier
            self.front.map(|node| unsafe {
                self.len -= 1;
                self.front = (*node.as_ptr()).back;
                &(*node.as_ptr()).elem
            })
        } else {
            None
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        (self.len, Some(self.len))
    }
}

impl<'a, T> DoubleEndedIterator for Iter<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        if self.len > 0 {
            self.back.map(|node| unsafe {
                self.len -= 1;
                self.back = (*node.as_ptr()).front;
                &(*node.as_ptr()).elem
            })
        } else {
            None
        }
    }
}

impl<'a, T> ExactSizeIterator for Iter<'a, T> {
    fn len(&self) -> usize {
        self.len
    }
}

impl<'a, T> IntoIterator for &'a mut LinkedList<T> {
```

```
    type IntoIter = IterMut<'a, T>;
    type Item = &'a mut T;

    fn into_iter(self) -> Self::IntoIter {
        self.iter_mut()
    }
}

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

    fn next(&mut self) -> Option<Self::Item> {
        // While self.front == self.back is a tempting condition to check here,
        // it won't do the right for yielding the last element! That sort of
        // thing only works for arrays because of "one-past-the-end" pointers.
        if self.len > 0 {
            // We could unwrap front, but this is safer and easier
            self.front.map(|node| unsafe {
                self.len -= 1;
                self.front = (*node.as_ptr()).back;
                &mut (*node.as_ptr()).elem
            })
        } else {
            None
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        (self.len, Some(self.len))
    }
}

impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        if self.len > 0 {
            self.back.map(|node| unsafe {
                self.len -= 1;
                self.back = (*node.as_ptr()).front;
                &mut (*node.as_ptr()).elem
            })
        } else {
            None
        }
    }
}

impl<'a, T> ExactSizeIterator for IterMut<'a, T> {
    fn len(&self) -> usize {
        self.len
    }
}

impl<T> IntoIterator for LinkedList<T> {
    type IntoIter = IntoIter<T>;
    type Item = T;

    fn into_iter(self) -> Self::IntoIter {
        IntoIter { list: self }
    }
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;

    fn next(&mut self) -> Option<Self::Item> {
        self.list.pop_front()
    }
```

```
    fn size_hint(&self) -> (usize, Option<usize>) {
        (self.list.len, Some(self.list.len))
    }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        self.list.pop_back()
    }
}

impl<T> ExactSizeIterator for IntoIter<T> {
    fn len(&self) -> usize {
        self.list.len
    }
}

impl<'a, T> CursorMut<'a, T> {
    pub fn index(&self) -> Option<usize> {
        self.index
    }

    pub fn move_next(&mut self) {
        if let Some(cur) = self.cur {
            unsafe {
                // We're on a real element, go to its next (back)
                self.cur = (*cur.as_ptr()).back;
                if self.cur.is_some() {
                    *self.index.as_mut().unwrap() += 1;
                } else {
                    // We just walked to the ghost, no more index
                    self.index = None;
                }
            }
        } else if !self.list.is_empty() {
            // We're at the ghost, and there is a real front, so move to it!
            self.cur = self.list.front;
            self.index = Some(0)
        } else {
            // We're at the ghost, but that's the only element... do nothing.
        }
    }

    pub fn move_prev(&mut self) {
        if let Some(cur) = self.cur {
            unsafe {
                // We're on a real element, go to its previous (front)
                self.cur = (*cur.as_ptr()).front;
                if self.cur.is_some() {
                    *self.index.as_mut().unwrap() -= 1;
                } else {
                    // We just walked to the ghost, no more index
                    self.index = None;
                }
            }
        } else if !self.list.is_empty() {
            // We're at the ghost, and there is a real back, so move to it!
            self.cur = self.list.back;
            self.index = Some(self.list.len - 1)
        } else {
            // We're at the ghost, but that's the only element... do nothing.
        }
    }

    pub fn current(&mut self) -> Option<&mut T> {
        unsafe { self.cur.map(|node| &mut (*node.as_ptr()).elem) }
    }
```

```
    pub fn peek_next(&mut self) -> Option<&mut T> {
        unsafe {
            let next = if let Some(cur) = self.cur {
                // Normal case, try to follow the cur node's back pointer
                (*cur.as_ptr()).back
            } else {
                // Ghost case, try to use the list's front pointer
                self.list.front
            };

            // Yield the element if the next node exists
            next.map(|node| &mut (*node.as_ptr()).elem)
        }
    }

    pub fn peek_prev(&mut self) -> Option<&mut T> {
        unsafe {
            let prev = if let Some(cur) = self.cur {
                // Normal case, try to follow the cur node's front pointer
                (*cur.as_ptr()).front
            } else {
                // Ghost case, try to use the list's back pointer
                self.list.back
            };

            // Yield the element if the prev node exists
            prev.map(|node| &mut (*node.as_ptr()).elem)
        }
    }

    pub fn split_before(&mut self) -> LinkedList<T> {
        // We have this:
        //
        //     list.front -> A <-> B <-> C <-> D <- list.back
        //                                ^
        //                               cur
        //
        //
        // And we want to produce this:
        //
        //     list.front -> C <-> D <- list.back
        //                    ^
        //                   cur
        //
        //
        //    return.front -> A <-> B <- return.back
        //
        if let Some(cur) = self.cur {
            // We are pointing at a real element, so the list is non-empty.
            unsafe {
                // Current state
                let old_len = self.list.len;
                let old_idx = self.index.unwrap();
                let prev = (*cur.as_ptr()).front;

                // What self will become
                let new_len = old_len - old_idx;
                let new_front = self.cur;
                let new_back = self.list.back;
                let new_idx = Some(0);

                // What the output will become
                let output_len = old_len - new_len;
                let output_front = self.list.front;
                let output_back = prev;

                // Break the links between cur and prev
                if let Some(prev) = prev {
```

```
                (*cur.as_ptr()).front = None;
                (*prev.as_ptr()).back = None;
            }

            // Produce the result:
            self.list.len = new_len;
            self.list.front = new_front;
            self.list.back = new_back;
            self.index = new_idx;

            LinkedList {
                front: output_front,
                back: output_back,
                len: output_len,
                _boo: PhantomData,
            }
        }
    } else {
        // We're at the ghost, just replace our list with an empty one.
        // No other state needs to be changed.
        std::mem::replace(self.list, LinkedList::new())
    }
}

pub fn split_after(&mut self) -> LinkedList<T> {
    // We have this:
    //
    //     list.front -> A <-> B <-> C <-> D <- list.back
    //                           ^
    //                          cur
    //
    //
    // And we want to produce this:
    //
    //     list.front -> A <-> B <- list.back
    //                           ^
    //                          cur
    //
    //
    //    return.front -> C <-> D <- return.back
    //
    if let Some(cur) = self.cur {
        // We are pointing at a real element, so the list is non-empty.
        unsafe {
            // Current state
            let old_len = self.list.len;
            let old_idx = self.index.unwrap();
            let next = (*cur.as_ptr()).back;

            // What self will become
            let new_len = old_idx + 1;
            let new_back = self.cur;
            let new_front = self.list.front;
            let new_idx = Some(old_idx);

            // What the output will become
            let output_len = old_len - new_len;
            let output_front = next;
            let output_back = self.list.back;

            // Break the links between cur and next
            if let Some(next) = next {
                (*cur.as_ptr()).back = None;
                (*next.as_ptr()).front = None;
            }

            // Produce the result:
            self.list.len = new_len;
```

```
                self.list.front = new_front;
                self.list.back = new_back;
                self.index = new_idx;

                LinkedList {
                    front: output_front,
                    back: output_back,
                    len: output_len,
                    _boo: PhantomData,
                }
            }
        } else {
            // We're at the ghost, just replace our list with an empty one.
            // No other state needs to be changed.
            std::mem::replace(self.list, LinkedList::new())
        }
    }

    pub fn splice_before(&mut self, mut input: LinkedList<T>) {
        // We have this:
        //
        // input.front -> 1 <-> 2 <- input.back
        //
        // list.front -> A <-> B <-> C <- list.back
        //                     ^
        //                    cur
        //
        //
        // Becoming this:
        //
        // list.front -> A <-> 1 <-> 2 <-> B <-> C <- list.back
        //                                 ^
        //                                cur
        //
        unsafe {
            // We can either `take` the input's pointers or `mem::forget`
            // it. Using `take` is more responsible in case we ever do custom
            // allocators or something that also needs to be cleaned up!
            if input.is_empty() {
                // Input is empty, do nothing.
            } else if let Some(cur) = self.cur {
                // Both lists are non-empty
                let in_front = input.front.take().unwrap();
                let in_back = input.back.take().unwrap();

                if let Some(prev) = (*cur.as_ptr()).front {
                    // General Case, no boundaries, just internal fixups
                    (*prev.as_ptr()).back = Some(in_front);
                    (*in_front.as_ptr()).front = Some(prev);
                    (*cur.as_ptr()).front = Some(in_back);
                    (*in_back.as_ptr()).back = Some(cur);
                } else {
                    // No prev, we're appending to the front
                    (*cur.as_ptr()).front = Some(in_back);
                    (*in_back.as_ptr()).back = Some(cur);
                    self.list.front = Some(in_front);
                }
                // Index moves forward by input length
                *self.index.as_mut().unwrap() += input.len;
            } else if let Some(back) = self.list.back {
                // We're on the ghost but non-empty, append to the back
                let in_front = input.front.take().unwrap();
                let in_back = input.back.take().unwrap();

                (*back.as_ptr()).back = Some(in_front);
                (*in_front.as_ptr()).front = Some(back);
                self.list.back = Some(in_back);
            } else {
```

```
                // We're empty, become the input, remain on the ghost
                std::mem::swap(self.list, &mut input);
            }

            self.list.len += input.len;
            // Not necessary but Polite To Do
            input.len = 0;

            // Input dropped here
        }
    }

    pub fn splice_after(&mut self, mut input: LinkedList<T>) {
        // We have this:
        //
        // input.front -> 1 <-> 2 <- input.back
        //
        // list.front -> A <-> B <-> C <- list.back
        //                     ^
        //                    cur
        //
        //
        // Becoming this:
        //
        // list.front -> A <-> B <-> 1 <-> 2 <-> C <- list.back
        //                     ^
        //                    cur
        //
        unsafe {
            // We can either `take` the input's pointers or `mem::forget`
            // it. Using `take` is more responsible in case we ever do custom
            // allocators or something that also needs to be cleaned up!
            if input.is_empty() {
                // Input is empty, do nothing.
            } else if let Some(cur) = self.cur {
                // Both lists are non-empty
                let in_front = input.front.take().unwrap();
                let in_back = input.back.take().unwrap();

                if let Some(next) = (*cur.as_ptr()).back {
                    // General Case, no boundaries, just internal fixups
                    (*next.as_ptr()).front = Some(in_back);
                    (*in_back.as_ptr()).back = Some(next);
                    (*cur.as_ptr()).back = Some(in_front);
                    (*in_front.as_ptr()).front = Some(cur);
                } else {
                    // No next, we're appending to the back
                    (*cur.as_ptr()).back = Some(in_front);
                    (*in_front.as_ptr()).front = Some(cur);
                    self.list.back = Some(in_back);
                }
                // Index doesn't change
            } else if let Some(front) = self.list.front {
                // We're on the ghost but non-empty, append to the front
                let in_front = input.front.take().unwrap();
                let in_back = input.back.take().unwrap();

                (*front.as_ptr()).front = Some(in_back);
                (*in_back.as_ptr()).back = Some(front);
                self.list.front = Some(in_front);
            } else {
                // We're empty, become the input, remain on the ghost
                std::mem::swap(self.list, &mut input);
            }

            self.list.len += input.len;
            // Not necessary but Polite To Do
            input.len = 0;
```

```
            // Input dropped here
        }
    }
}

unsafe impl<T: Send> Send for LinkedList<T> {}
unsafe impl<T: Sync> Sync for LinkedList<T> {}

unsafe impl<'a, T: Send> Send for Iter<'a, T> {}
unsafe impl<'a, T: Sync> Sync for Iter<'a, T> {}

unsafe impl<'a, T: Send> Send for IterMut<'a, T> {}
unsafe impl<'a, T: Sync> Sync for IterMut<'a, T> {}

#[allow(dead_code)]
fn assert_properties() {
    fn is_send<T: Send>() {}
    fn is_sync<T: Sync>() {}

    is_send::<LinkedList<i32>>();
    is_sync::<LinkedList<i32>>();

    is_send::<IntoIter<i32>>();
    is_sync::<IntoIter<i32>>();

    is_send::<Iter<i32>>();
    is_sync::<Iter<i32>>();

    is_send::<IterMut<i32>>();
    is_sync::<IterMut<i32>>();

    fn linked_list_covariant<'a, T>(x: LinkedList<&'static T>) -> LinkedList<&'a T> {
        x
    }
    fn iter_covariant<'i, 'a, T>(x: Iter<'i, &'static T>) -> Iter<'i, &'a T> {
        x
    }
    fn into_iter_covariant<'a, T>(x: IntoIter<&'static T>) -> IntoIter<&'a T> {
        x
    }

    /// ```compile_fail,E0308
    /// use linked_list::IterMut;
    ///
    /// fn iter_mut_covariant<'i, 'a, T>(x: IterMut<'i, &'static T>) -> IterMut<'i, &'a
T> { x }
    /// ```
    fn iter_mut_invariant() {}
}

#[cfg(test)]
mod test {
    use super::LinkedList;

    fn generate_test() -> LinkedList<i32> {
        list_from(&[0, 1, 2, 3, 4, 5, 6])
    }

    fn list_from<T: Clone>(v: &[T]) -> LinkedList<T> {
        v.iter().map(|x| (*x).clone()).collect()
    }

    #[test]
    fn test_basic_front() {
        let mut list = LinkedList::new();

        // Try to break an empty list
```

```
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);

        // Try to break a one item list
        list.push_front(10);
        assert_eq!(list.len(), 1);
        assert_eq!(list.pop_front(), Some(10));
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);

        // Mess around
        list.push_front(10);
        assert_eq!(list.len(), 1);
        list.push_front(20);
        assert_eq!(list.len(), 2);
        list.push_front(30);
        assert_eq!(list.len(), 3);
        assert_eq!(list.pop_front(), Some(30));
        assert_eq!(list.len(), 2);
        list.push_front(40);
        assert_eq!(list.len(), 3);
        assert_eq!(list.pop_front(), Some(40));
        assert_eq!(list.len(), 2);
        assert_eq!(list.pop_front(), Some(20));
        assert_eq!(list.len(), 1);
        assert_eq!(list.pop_front(), Some(10));
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);
        assert_eq!(list.pop_front(), None);
        assert_eq!(list.len(), 0);
    }

    #[test]
    fn test_basic() {
        let mut m = LinkedList::new();
        assert_eq!(m.pop_front(), None);
        assert_eq!(m.pop_back(), None);
        assert_eq!(m.pop_front(), None);
        m.push_front(1);
        assert_eq!(m.pop_front(), Some(1));
        m.push_back(2);
        m.push_back(3);
        assert_eq!(m.len(), 2);
        assert_eq!(m.pop_front(), Some(2));
        assert_eq!(m.pop_front(), Some(3));
        assert_eq!(m.len(), 0);
        assert_eq!(m.pop_front(), None);
        m.push_back(1);
        m.push_back(3);
        m.push_back(5);
        m.push_back(7);
        assert_eq!(m.pop_front(), Some(1));

        let mut n = LinkedList::new();
        n.push_front(2);
        n.push_front(3);
        {
            assert_eq!(n.front().unwrap(), &3);
            let x = n.front_mut().unwrap();
            assert_eq!(*x, 3);
            *x = 0;
        }
        {
            assert_eq!(n.back().unwrap(), &2);
            let y = n.back_mut().unwrap();
```

```
            assert_eq!(*y, 2);
            *y = 1;
        }
        assert_eq!(n.pop_front(), Some(0));
        assert_eq!(n.pop_front(), Some(1));
    }

    #[test]
    fn test_iterator() {
        let m = generate_test();
        for (i, elt) in m.iter().enumerate() {
            assert_eq!(i as i32, *elt);
        }
        let mut n = LinkedList::new();
        assert_eq!(n.iter().next(), None);
        n.push_front(4);
        let mut it = n.iter();
        assert_eq!(it.size_hint(), (1, Some(1)));
        assert_eq!(it.next().unwrap(), &4);
        assert_eq!(it.size_hint(), (0, Some(0)));
        assert_eq!(it.next(), None);
    }

    #[test]
    fn test_iterator_double_end() {
        let mut n = LinkedList::new();
        assert_eq!(n.iter().next(), None);
        n.push_front(4);
        n.push_front(5);
        n.push_front(6);
        let mut it = n.iter();
        assert_eq!(it.size_hint(), (3, Some(3)));
        assert_eq!(it.next().unwrap(), &6);
        assert_eq!(it.size_hint(), (2, Some(2)));
        assert_eq!(it.next_back().unwrap(), &4);
        assert_eq!(it.size_hint(), (1, Some(1)));
        assert_eq!(it.next_back().unwrap(), &5);
        assert_eq!(it.next_back(), None);
        assert_eq!(it.next(), None);
    }

    #[test]
    fn test_rev_iter() {
        let m = generate_test();
        for (i, elt) in m.iter().rev().enumerate() {
            assert_eq!(6 - i as i32, *elt);
        }
        let mut n = LinkedList::new();
        assert_eq!(n.iter().rev().next(), None);
        n.push_front(4);
        let mut it = n.iter().rev();
        assert_eq!(it.size_hint(), (1, Some(1)));
        assert_eq!(it.next().unwrap(), &4);
        assert_eq!(it.size_hint(), (0, Some(0)));
        assert_eq!(it.next(), None);
    }

    #[test]
    fn test_mut_iter() {
        let mut m = generate_test();
        let mut len = m.len();
        for (i, elt) in m.iter_mut().enumerate() {
            assert_eq!(i as i32, *elt);
            len -= 1;
        }
        assert_eq!(len, 0);
        let mut n = LinkedList::new();
        assert!(n.iter_mut().next().is_none());
```

```
        n.push_front(4);
        n.push_back(5);
        let mut it = n.iter_mut();
        assert_eq!(it.size_hint(), (2, Some(2)));
        assert!(it.next().is_some());
        assert!(it.next().is_some());
        assert_eq!(it.size_hint(), (0, Some(0)));
        assert!(it.next().is_none());
    }

    #[test]
    fn test_iterator_mut_double_end() {
        let mut n = LinkedList::new();
        assert!(n.iter_mut().next_back().is_none());
        n.push_front(4);
        n.push_front(5);
        n.push_front(6);
        let mut it = n.iter_mut();
        assert_eq!(it.size_hint(), (3, Some(3)));
        assert_eq!(*it.next().unwrap(), 6);
        assert_eq!(it.size_hint(), (2, Some(2)));
        assert_eq!(*it.next_back().unwrap(), 4);
        assert_eq!(it.size_hint(), (1, Some(1)));
        assert_eq!(*it.next_back().unwrap(), 5);
        assert!(it.next_back().is_none());
        assert!(it.next().is_none());
    }

    #[test]
    fn test_eq() {
        let mut n: LinkedList<u8> = list_from(&[]);
        let mut m = list_from(&[]);
        assert!(n == m);
        n.push_front(1);
        assert!(n != m);
        m.push_back(1);
        assert!(n == m);

        let n = list_from(&[2, 3, 4]);
        let m = list_from(&[1, 2, 3]);
        assert!(n != m);
    }

    #[test]
    fn test_ord() {
        let n = list_from(&[]);
        let m = list_from(&[1, 2, 3]);
        assert!(n < m);
        assert!(m > n);
        assert!(n <= n);
        assert!(n >= n);
    }

    #[test]
    fn test_ord_nan() {
        let nan = 0.0f64 / 0.0;
        let n = list_from(&[nan]);
        let m = list_from(&[nan]);
        assert!(!(n < m));
        assert!(!(n > m));
        assert!(!(n <= m));
        assert!(!(n >= m));

        let n = list_from(&[nan]);
        let one = list_from(&[1.0f64]);
        assert!(!(n < one));
        assert!(!(n > one));
        assert!(!(n <= one));
```

```
        assert!(!(n >= one));

        let u = list_from(&[1.0f64, 2.0, nan]);
        let v = list_from(&[1.0f64, 2.0, 3.0]);
        assert!(!(u < v));
        assert!(!(u > v));
        assert!(!(u <= v));
        assert!(!(u >= v));

        let s = list_from(&[1.0f64, 2.0, 4.0, 2.0]);
        let t = list_from(&[1.0f64, 2.0, 3.0, 2.0]);
        assert!(!(s < t));
        assert!(s > one);
        assert!(!(s <= one));
        assert!(s >= one);
    }

    #[test]
    fn test_debug() {
        let list: LinkedList<i32> = (0..10).collect();
        assert_eq!(format!("{:?}", list), "[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]");

        let list: LinkedList<&str> = vec!["just", "one", "test", "more"]
            .iter()
            .copied()
            .collect();
        assert_eq!(format!("{:?}", list), r#"["just", "one", "test", "more"]"#);
    }

    #[test]
    fn test_hashmap() {
        // Check that HashMap works with this as a key

        let list1: LinkedList<i32> = (0..10).collect();
        let list2: LinkedList<i32> = (1..11).collect();
        let mut map = std::collections::HashMap::new();

        assert_eq!(map.insert(list1.clone(), "list1"), None);
        assert_eq!(map.insert(list2.clone(), "list2"), None);

        assert_eq!(map.len(), 2);

        assert_eq!(map.get(&list1), Some(&"list1"));
        assert_eq!(map.get(&list2), Some(&"list2"));

        assert_eq!(map.remove(&list1), Some("list1"));
        assert_eq!(map.remove(&list2), Some("list2"));

        assert!(map.is_empty());
    }

    #[test]
    fn test_cursor_move_peek() {
        let mut m: LinkedList<u32> = LinkedList::new();
        m.extend([1, 2, 3, 4, 5, 6]);
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        assert_eq!(cursor.current(), Some(&mut 1));
        assert_eq!(cursor.peek_next(), Some(&mut 2));
        assert_eq!(cursor.peek_prev(), None);
        assert_eq!(cursor.index(), Some(0));
        cursor.move_prev();
        assert_eq!(cursor.current(), None);
        assert_eq!(cursor.peek_next(), Some(&mut 1));
        assert_eq!(cursor.peek_prev(), Some(&mut 6));
        assert_eq!(cursor.index(), None);
        cursor.move_next();
        cursor.move_next();
```

```
        assert_eq!(cursor.current(), Some(&mut 2));
        assert_eq!(cursor.peek_next(), Some(&mut 3));
        assert_eq!(cursor.peek_prev(), Some(&mut 1));
        assert_eq!(cursor.index(), Some(1));

        let mut cursor = m.cursor_mut();
        cursor.move_prev();
        assert_eq!(cursor.current(), Some(&mut 6));
        assert_eq!(cursor.peek_next(), None);
        assert_eq!(cursor.peek_prev(), Some(&mut 5));
        assert_eq!(cursor.index(), Some(5));
        cursor.move_next();
        assert_eq!(cursor.current(), None);
        assert_eq!(cursor.peek_next(), Some(&mut 1));
        assert_eq!(cursor.peek_prev(), Some(&mut 6));
        assert_eq!(cursor.index(), None);
        cursor.move_prev();
        cursor.move_prev();
        assert_eq!(cursor.current(), Some(&mut 5));
        assert_eq!(cursor.peek_next(), Some(&mut 6));
        assert_eq!(cursor.peek_prev(), Some(&mut 4));
        assert_eq!(cursor.index(), Some(4));
    }

    #[test]
    fn test_cursor_mut_insert() {
        let mut m: LinkedList<u32> = LinkedList::new();
        m.extend([1, 2, 3, 4, 5, 6]);
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        cursor.splice_before(Some(7).into_iter().collect());
        cursor.splice_after(Some(8).into_iter().collect());
        // check_links(&m);
        assert_eq!(
            m.iter().cloned().collect::<Vec<_>>(),
            &[7, 1, 8, 2, 3, 4, 5, 6]
        );
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        cursor.move_prev();
        cursor.splice_before(Some(9).into_iter().collect());
        cursor.splice_after(Some(10).into_iter().collect());
        check_links(&m);
        assert_eq!(
            m.iter().cloned().collect::<Vec<_>>(),
            &[10, 7, 1, 8, 2, 3, 4, 5, 6, 9]
        );

        /* remove_current not impl'd
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        cursor.move_prev();
        assert_eq!(cursor.remove_current(), None);
        cursor.move_next();
        cursor.move_next();
        assert_eq!(cursor.remove_current(), Some(7));
        cursor.move_prev();
        cursor.move_prev();
        cursor.move_prev();
        assert_eq!(cursor.remove_current(), Some(9));
        cursor.move_next();
        assert_eq!(cursor.remove_current(), Some(10));
        check_links(&m);
        assert_eq!(m.iter().cloned().collect::<Vec<_>>(), &[1, 8, 2, 3, 4, 5, 6]);
        */

        let mut m: LinkedList<u32> = LinkedList::new();
        m.extend([1, 8, 2, 3, 4, 5, 6]);
```

```
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        let mut p: LinkedList<u32> = LinkedList::new();
        p.extend([100, 101, 102, 103]);
        let mut q: LinkedList<u32> = LinkedList::new();
        q.extend([200, 201, 202, 203]);
        cursor.splice_after(p);
        cursor.splice_before(q);
        check_links(&m);
        assert_eq!(
            m.iter().cloned().collect::<Vec<_>>(),
            &[200, 201, 202, 203, 1, 100, 101, 102, 103, 8, 2, 3, 4, 5, 6]
        );
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        cursor.move_prev();
        let tmp = cursor.split_before();
        assert_eq!(m.into_iter().collect::<Vec<_>>(), &[]);
        m = tmp;
        let mut cursor = m.cursor_mut();
        cursor.move_next();
        cursor.move_next();
        cursor.move_next();
        cursor.move_next();
        cursor.move_next();
        cursor.move_next();
        cursor.move_next();
        let tmp = cursor.split_after();
        assert_eq!(
            tmp.into_iter().collect::<Vec<_>>(),
            &[102, 103, 8, 2, 3, 4, 5, 6]
        );
        check_links(&m);
        assert_eq!(
            m.iter().cloned().collect::<Vec<_>>(),
            &[200, 201, 202, 203, 1, 100, 101]
        );
    }

    fn check_links<T: Eq + std::fmt::Debug>(list: &LinkedList<T>) {
        let from_front: Vec<_> = list.iter().collect();
        let from_back: Vec<_> = list.iter().rev().collect();
        let re_reved: Vec<_> = from_back.into_iter().rev().collect();

        assert_eq!(from_front, re_reved);
    }
}
```

A Bunch of Silly Lists

Alright. That's it. We made all the lists.

ahahahaha

No

There's always more lists.

This chapter is a living document of the more ridiculous linked lists and how they interact with Rust.

1. The Double Single
2. The Stack Allocated List
3. The Self-Referential Arena List?

4. The GhostCell List?

The Double Singly-Linked List

We struggled with doubly-linked lists because they have tangled ownership semantics: no node strictly owns any other node. However we struggled with this because we brought in our preconceived notions of what a linked list *is*. Namely, we assumed that all the links go in the same direction.

Instead, we can smash our list into two halves: one going to the left, and one going to the right:

```
// lib.rs
// ...
pub mod silly1;     // NEW!
```

```
// silly1.rs
use crate::second::List as Stack;

struct List<T> {
    left: Stack<T>,
    right: Stack<T>,
}
```

Now, rather than having a mere safe stack, we have a general purpose list. We can grow the list leftwards or rightwards by pushing onto either stack. We can also "walk" along the list by popping values off one end and onto the other. To avoid needless allocations, we're going to copy the source of our safe Stack to get access to its private details:

```
pub struct Stack<T> {
    head: Link<T>,
}

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

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

impl<T> Stack<T> {
    pub fn new() -> Self {
        Stack { head: None }
    }

    pub fn push(&mut self, elem: T) {
        let new_node = Box::new(Node {
            elem: elem,
            next: self.head.take(),
        });

        self.head = Some(new_node);
    }

    pub fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            let node = *node;
            self.head = node.next;
            node.elem
        })
    }

    pub fn peek(&self) -> Option<&T> {
        self.head.as_ref().map(|node| {
            &node.elem
        })
    }

    pub fn peek_mut(&mut self) -> Option<&mut T> {
        self.head.as_mut().map(|node| {
            &mut node.elem
        })
    }
}

impl<T> Drop for Stack<T> {
    fn drop(&mut self) {
        let mut cur_link = self.head.take();
        while let Some(mut boxed_node) = cur_link {
            cur_link = boxed_node.next.take();
        }
    }
}
```

And just rework `push` and `pop` a bit:

```
pub fn push(&mut self, elem: T) {
    let new_node = Box::new(Node {
        elem: elem,
        next: None,
    });

    self.push_node(new_node);
}

fn push_node(&mut self, mut node: Box<Node<T>>) {
    node.next = self.head.take();
    self.head = Some(node);
}

pub fn pop(&mut self) -> Option<T> {
    self.pop_node().map(|node| {
        node.elem
    })
}

fn pop_node(&mut self) -> Option<Box<Node<T>>> {
    self.head.take().map(|mut node| {
        self.head = node.next.take();
        node
    })
}
```

Now we can make our List:

```
pub struct List<T> {
    left: Stack<T>,
    right: Stack<T>,
}

impl<T> List<T> {
    fn new() -> Self {
        List { left: Stack::new(), right: Stack::new() }
    }
}
```

And we can do the usual stuff:

```
pub fn push_left(&mut self, elem: T) { self.left.push(elem) }
pub fn push_right(&mut self, elem: T) { self.right.push(elem) }
pub fn pop_left(&mut self) -> Option<T> { self.left.pop() }
pub fn pop_right(&mut self) -> Option<T> { self.right.pop() }
pub fn peek_left(&self) -> Option<&T> { self.left.peek() }
pub fn peek_right(&self) -> Option<&T> { self.right.peek() }
pub fn peek_left_mut(&mut self) -> Option<&mut T> { self.left.peek_mut() }
pub fn peek_right_mut(&mut self) -> Option<&mut T> { self.right.peek_mut() }
```

But most interestingly, we can walk around!

```
pub fn go_left(&mut self) -> bool {
    self.left.pop_node().map(|node| {
        self.right.push_node(node);
    }).is_some()
}

pub fn go_right(&mut self) -> bool {
    self.right.pop_node().map(|node| {
        self.left.push_node(node);
    }).is_some()
}
```

We return booleans here as just a convenience to indicate whether we actually managed to move. Now let's test this baby out:

```
#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn walk_aboot() {
        let mut list = List::new();             // [_]

        list.push_left(0);                      // [0,_]
        list.push_right(1);                     // [0, _, 1]
        assert_eq!(list.peek_left(), Some(&0));
        assert_eq!(list.peek_right(), Some(&1));

        list.push_left(2);                      // [0, 2, _, 1]
        list.push_left(3);                      // [0, 2, 3, _, 1]
        list.push_right(4);                     // [0, 2, 3, _, 4, 1]

        while list.go_left() {}                 // [_, 0, 2, 3, 4, 1]

        assert_eq!(list.pop_left(), None);
        assert_eq!(list.pop_right(), Some(0));  // [_, 2, 3, 4, 1]
        assert_eq!(list.pop_right(), Some(2));  // [_, 3, 4, 1]

        list.push_left(5);                      // [5, _, 3, 4, 1]
        assert_eq!(list.pop_right(), Some(3));  // [5, _, 4, 1]
        assert_eq!(list.pop_left(), Some(5));   // [_, 4, 1]
        assert_eq!(list.pop_right(), Some(4));  // [_, 1]
        assert_eq!(list.pop_right(), Some(1));  // [_]

        assert_eq!(list.pop_right(), None);
        assert_eq!(list.pop_left(), None);

    }
}
```

```
> cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 16 tests
test fifth::test::into_iter ... ok
test fifth::test::basics ... ok
test fifth::test::iter ... ok
test fifth::test::iter_mut ... ok
test fourth::test::into_iter ... ok
test fourth::test::basics ... ok
test fourth::test::peek ... ok
test first::test::basics ... ok
test second::test::into_iter ... ok
test second::test::basics ... ok
test second::test::iter ... ok
test second::test::iter_mut ... ok
test third::test::basics ... ok
test third::test::iter ... ok
test second::test::peek ... ok
test silly1::test::walk_aboot ... ok

test result: ok. 16 passed; 0 failed; 0 ignored; 0 measured
```

This is an extreme example of a *finger* data structure, where we maintain some kind of finger into the structure, and as a consequence can support operations on locations in time proportional to the distance from the finger.

We can make very fast changes to the list around our finger, but if we want to make changes far away from our finger we have to walk all the way over there. We can permanently walk over there by shifting the elements from one stack to the other, or we could just walk along the links with an `&mut` temporarily to do the changes. However the `&mut` can never go back up the list, while our finger can!

The Stack-Allocated Linked List

This book is largely focused on *heap-allocated* linked lists, because those are the most common and practical, but we don't *have* to use heap allocation. Heap allocation is nice because it makes it easy to dynamically allocate memory. Stack allocation is less friendly in this regard — things like C's `alloca` are widely regarded as Very Cursed And Problematic.

So let's allocate memory on the stack the easy way: by calling a function and getting a new stack frame with more space! This is a very silly solution to our problem but also genuinely practical and useful. It's done all the time, potentially without actually even thinking about it as a linked list!

Any time you're doing something recursively, you can just pass a pointer to the current step's state to the next step. If that pointer itself is *part* of the state, then you've created a linked list that's stack-allocated!

Now of course we're in the *silly* part of the book so we're going to do this in a silly way: by making the linked list the star and forcing all the user's code to live in a swamp of callbacks. Everybody loves nested callbacks!

Our List type will just be a Node with a reference to another Node:

```
pub struct List<'a, T> {
    pub data: T,
    pub prev: Option<&'a List<'a, T>>,
}
```

And it will have only one operation, `push`, which will take the old list, the state for the current node, and a callback. The new list will be produced in the callback. We will also let callbacks return any value, which `push` will return when it completes:

```
impl<'a, T> List<'a, T> {
    pub fn push<U>(
        prev: Option<&'a List<'a, T>>,
        data: T,
        callback: impl FnOnce(&List<'a, T>) -> U,
    ) -> U {
        let list = List { data, prev };
        callback(&list)
    }
}
```

That's it! We can use it like this:

```
List::push(None, 3, |list| {
    println!("{}", list.data);
    List::push(Some(list), 5, |list| {
        println!("{}", list.data);
        List::push(Some(list), 13, |list| {
            println!("{}", list.data);
        })
    })
})
```

It's beautiful. 

The user can already traverse this list by using while-let to walk over the `prev` values, but just for fun, let's implement an iterator, which is the usual:

```
impl<'a, T> List<'a, T> {
    pub fn iter(&'a self) -> Iter<'a, T> {
        Iter { next: Some(self) }
    }
}

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

    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.prev;
            &node.data
        })
    }
}
```

Let's test it out:

```
#[cfg(test)]
mod test {
    use super::List;

    #[test]
    fn elegance() {
        List::push(None, 3, |list| {
            assert_eq!(list.iter().copied().sum::<i32>(), 3);
            List::push(Some(list), 5, |list| {
                assert_eq!(list.iter().copied().sum::<i32>(), 5 + 3);
                List::push(Some(list), 13, |list| {
                    assert_eq!(list.iter().copied().sum::<i32>(), 13 + 5 + 3);
                })
            })
        })
    }
}
```

```
> cargo test

running 18 tests
test fifth::test::into_iter ... ok
test fifth::test::iter ... ok
test fifth::test::iter_mut ... ok
test fifth::test::basics ... ok
test fifth::test::miri_food ... ok
test first::test::basics ... ok
test second::test::into_iter ... ok
test fourth::test::peek ... ok
test fourth::test::into_iter ... ok
test second::test::iter_mut ... ok
test fourth::test::basics ... ok
test second::test::basics ... ok
test second::test::iter ... ok
test third::test::basics ... ok
test silly1::test::walk_aboot ... ok
test silly2::test::elegance ... ok
test second::test::peek ... ok
test third::test::iter ... ok

test result: ok. 18 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
```

Now at this point you might be wonder "hey can I mutate the data stored in a node?". Maybe! Let's try to make the list use mutable references instead of shared ones:

```
pub struct List<'a, T> {
    pub data: T,
    pub prev: Option<&'a mut List<'a, T>>,
}

pub struct Iter<'a, T> {
    next: Option<&'a List<'a, T>>,
}

impl<'a, T> List<'a, T> {
    pub fn push<U>(
        prev: Option<&'a mut List<'a, T>>,
        data: T,
        callback: impl FnOnce(&mut List<'a, T>) -> U,
    ) -> U {
        let mut list = List { data, prev };
        callback(&mut list)
    }

    pub fn iter(&'a self) -> Iter<'a, T> {
        Iter { next: Some(self) }
    }
}

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

    fn next(&mut self) -> Option<Self::Item> {
        self.next.map(|node| {
            self.next = node.prev.as_ref().map(|prev| &**prev);
            &node.data
        })
    }
}
```

```
> cargo test

error[E0521]: borrowed data escapes outside of closure
  --> src\silly2.rs:47:32
   |
46 |  List::push(Some(list), 13, |list| {
   |                              ----
   |                              |
   |              `list` declared here, outside of the closure body
   |              `list` is a reference that is only valid in the closure body
47 |      assert_eq!(list.iter().copied().sum::<i32>(), 13 + 5 + 3);
   |                  ^^^^^^^^^^^ `list` escapes the closure body here

error[E0521]: borrowed data escapes outside of closure
  --> src\silly2.rs:45:28
   |
44 |  List::push(Some(list), 5, |list| {
   |                             ----
   |                             |
   |              `list` declared here, outside of the closure body
   |              `list` is a reference that is only valid in the closure body
45 |      assert_eq!(list.iter().copied().sum::<i32>(), 5 + 3);
   |                  ^^^^^^^^^^^ `list` escapes the closure body here


<ad infinitum>
```

Whelp. Seems like it doesn't like our iterator. Maybe we messed that up? Let's simplify the test a bit to check:

```
#[test]
fn elegance() {
    List::push(None, 3, |list| {
        assert_eq!(list.data, 3);
        List::push(Some(list), 5, |list| {
            assert_eq!(list.data, 5);
            List::push(Some(list), 13, |list| {
                assert_eq!(list.data, 13);
            })
        })
    })
}
```

```
> cargo test

error[E0521]: borrowed data escapes outside of closure
  --> src\silly2.rs:46:17
   |
44 |   List::push(Some(list), 5, |list| {
   |                              ----
   |                              |
   |              `list` declared here, outside of the closure body
   |              `list` is a reference that is only valid in the closure body
45 |       assert_eq!(list.data, 5);
46 | /     List::push(Some(list), 13, |list| {
47 | |         assert_eq!(list.data, 13);
48 | |     })
   | |______^ `list` escapes the closure body here

error[E0521]: borrowed data escapes outside of closure
  --> src\silly2.rs:44:13
   |
42 |   List::push(None, 3, |list| {
   |                        ----
   |                        |
   |              `list` declared here, outside of the closure body
   |              `list` is a reference that is only valid in the closure body
43 |       assert_eq!(list.data, 3);
44 | /     List::push(Some(list), 5, |list| {
45 | |         assert_eq!(list.data, 5);
46 | |         List::push(Some(list), 13, |list| {
47 | |             assert_eq!(list.data, 13);
48 | |         })
49 | |     })
   | |_____________^ `list` escapes the closure body here
```

Hmm no that's still some hot garbage.

The problem is that our list is accidentally(😉) relying on *variance*. Variance is a complicated subject but let's look at it in simplified terms here:

Each list contains a reference to a List with *the exact same type as itself*. From the perspective of the inner-most list, that means all lists are using the same lifetime as itself, but this is *objectively* false: each node in the list lives strictly longer than the next one, because they are literally in nested scopes!

So... why did the code compile when we were using shared references? Because in many cases, the compiler knows it's safe to have something that lives "too long"! When we stuff a reference to a list into the next one, the compiler is quietly "shrinking" down the lifetimes to make them fit what the new list expects. This lifetime shrinking is *variance*.

It's the exact same trick in languages with inheritance that let's you pass a Cat where an Animal (a supertype of a Cat) is expected. Intuitively we know it's fine to pass a Cat when an Animal is expected

because a Cat is just an Animal *and more*. It's *fine* to forget the "and more" part for a while, right?

Similarly, a larger lifetime is just a smaller lifetime *and more*. So it's fine to forget the "and more" here too!

But of course you are now wondering: then why doesn't the mutable reference version work!?

Well, variance *isn't* always safe. If our code *did* compile, we could have written a use-after-free like this:

```
List::push(None, 3, |list| {
    List::push(Some(list), 5, |list| {
        List::push(Some(list), 13, |list| {
            // HAHAHA all the lifetimes are the same, so the compiler will
            // let me rewrite my parent to hold a mutable reference to myself!
            // I will create all the use-after-frees!!
            *list.prev.as_mut().unwrap().prev = Some(list);
        })
    })
})
```

The problem with forgetting details is that *somewhere else might remember those details and expect them to remain true*. That is a very big problem once you introduce *mutation*. If you're not careful, the code that doesn't remember the "and more" that we threw away might think it's fine to write things to places that "remember" and *expect* the "and more" to still be there.

Put in terms of inheritance: this code has to be illegal:

```
let mut my_kitty = Cat;                  // Make a Cat (long lifetime)
let animal: &mut Animal = &mut my_kitty; // Forget it's a Cat (shorten lifetime)
*animal = Dog;                           // Write a Dog (short lifetime)
my_kitty.meow();                         // Meowing Dog! (Use After Free)
```

So while you *can* shorten the lifetime of a mutable reference, once you start *nesting* them things become "invariant" and you're not allowed to shorten lifetimes anymore.

Specifically `&mut &'big mut T` cannot be converted to `&mut &'small mut T`, where `'big` is bigger than `'small`. Or more formally, `&'a mut T` is covariant over `'a` but invariant over `T`.

Fun fact: Java actually specifically *lets* you do this kind of thing, but it does runtime checks to prevent meowing dogs.

So what can we do to mutate the data? Use interior mutability! This lets us tell the compiler that we just want to be able to mutate the *data* but won't touch the references.

We can just revert back to the previous version of our code with shared references, and use `Cell` in a new test:

```
#[test]
fn cell() {
    use std::cell::Cell;

    List::push(None, Cell::new(3), |list| {
        List::push(Some(list), Cell::new(5), |list| {
            List::push(Some(list), Cell::new(13), |list| {
                // Multiply every value in the list by 10
                for val in list.iter() {
                    val.set(val.get() * 10)
                }

                let mut vals = list.iter();
                assert_eq!(vals.next().unwrap().get(), 130);
                assert_eq!(vals.next().unwrap().get(), 50);
                assert_eq!(vals.next().unwrap().get(), 30);
                assert_eq!(vals.next(), None);
                assert_eq!(vals.next(), None);
            })
        })
    })
}
```

```
> cargo test

running 19 tests
test fifth::test::into_iter ... ok
test fifth::test::basics ... ok
test fifth::test::iter_mut ... ok
test fifth::test::iter ... ok
test fourth::test::basics ... ok
test fourth::test::into_iter ... ok
test second::test::into_iter ... ok
test first::test::basics ... ok
test fourth::test::peek ... ok
test second::test::basics ... ok
test fifth::test::miri_food ... ok
test silly2::test::cell ... ok
test third::test::iter ... ok
test second::test::iter_mut ... ok
test second::test::peek ... ok
test silly1::test::walk_aboot ... ok
test silly2::test::elegance ... ok
test third::test::basics ... ok
test second::test::iter ... ok

test result: ok. 19 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
```

Easy as recursive pie! ✨