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`
- Ownership, borrowing, inherited mutability, interior mutability, Copy
- All The Keywords: struct, enum, fn, pub, impl, use, ...
- Pattern matching, generics, destructors
- Testing
- Basic Unsafe Rust

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.

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 lists 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.

However it should be noted that Rust can pattern match on arrays and talk about sub-arrays using slices! It's actually even more expressive than a functional list in some regards because you can talk about the last element or even "the array without the first and last two elements" or whatever other crazy thing you want.

It is true that you can't *build* a list using slices. You can only tear them apart.

For laziness we instead have iterators. These 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. No surprise here: slices can also be coerced to an iterator.

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):

```rust ,ignore // in lib.rs pub mod first;
```

~~~
# Basic Data Layout

Alright, so what's a linked list? Well basically, it's a bunch of
on the heap (hush, kernel people!) that point to each other in seq
lists are something procedural programmers shouldn't touch with a
and what functional programmers use for everything. It seems fair,
should ask functional programmers for the definition of a linked l
probably give you something like the following definition:

```haskell
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:

```rust ,ignore // 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:

```text
> 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
~~~

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
    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:

```rust ,ignore pub enum List { Empty, Elem(i32, Box), }

````

```text
> 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 allocate an extra node, but one of our nodes doesn't need to be 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:

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

````

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

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

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

Consider splitting a list in both layouts:

```text
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:

\`\`\`rust ,ignore pub enum List { Empty, ElemThenEmpty(i32), ElemThenNotEmpty(i32, Box), }

~~~
Hopefully this seems like an even worse idea to you. Most notably,
complicates our logic, because there is now a completely invalid s
`ElemThenNotEmpty(0, Box(Empty))`. It also *still* suffers from no
allocating our elements.

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

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

```rust,ignore
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:

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

pub enum List { Empty, More(Box), }

```

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 u
demonstrate that our first layout (as suggested by the official Ru
documentation) was problematic.

```text
> cargo build

warning: private type `first::Node` in public interface (error E04
 --> 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 r
```

:(

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:

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

enum Link { Empty, More(Box), }

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

```

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

```text
> 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:

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

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

```rust ,ignore
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`:

```rust ,ignore 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
  not repeating yourself!
* We create an instance of a struct in much the same way we declar
  instead of providing the types of its fields, we initialize them
* We refer to variants of an enum using `::`, which is the namespa
* The last expression of a function is implicitly returned.
  This makes simple functions a little neater. You can still use `
  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*
with it. We do that with "normal" (non-static) methods. Methods ar
case of function in Rust because of  the `self` argument, which do
a declared type:

```rust ,ignore
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:

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

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

```rust ,ignore
    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?

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

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

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:

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

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

}

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

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:

```rust ,ignore use std::mem;

````

and use it appropriately:

```rust ,ignore
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:

```rust ,ignore pub fn pop(&mut self) -> Option { // TODO }

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

The pointy bits on `Option<T>` indicate that Option is actually *g
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 Emp
More? Pattern matching with `match`!

```rust ,ignore
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::
   |            |
   |            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).

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

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

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

```text
> 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::
  --> 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:

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

````

```text
> cargo build

warning: unused variable: `node`
  --> src/first.rs:32:24
   |
32 |             Link::More(node) => {
   |                        ^^^^ help: consider prefixing with an
   |
   = 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?

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

````
```text
> 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 borr
````

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:

```rust ,ignore pub fn pop(&mut self) -> Option { 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 }

````
```text
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:

```rust ,ignore pub fn pop(&mut self) -> Option { 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::Emp
completely lost its braces, because we only have one expression to
evaluate. Just a nice shorthand for simple cases.

```text
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*:

```rust ,ignore // in first.rs

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

~~~~

And we invoke it with `cargo test`.

```text
> 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

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

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

```rust ,ignore 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);
}
```

}

~~~
```text
> cargo test

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

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

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

````
```text
> 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

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

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

```rust ,ignore

## [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
works right!

Do we need to worry about cleaning up our list? Technically, no, n
Like C++, Rust uses destructors to automatically clean up resource
done with. A type has a destructor if it implements a *trait* call
Traits are Rust's fancy term for interfaces. The Drop trait has th
interface:

```rust ,ignore
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:

```rust ,ignore 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 { 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 t
way to drop in a tail-recursive manner! Instead we're going to hav
write an iterative drop for `List` that hoists nodes out of their


```rust ,ignore
impl Drop for List {
    fn drop(&mut self) {
        let mut cur_link = mem::replace(&mut self.head, Link::Empt
        // `while let` == "do this thing until this pattern doesn'
        while let Link::More(mut boxed_node) = cur_link {
            cur_link = mem::replace(&mut boxed_node.next, Link::Em
            // boxed_node goes out of scope and gets dropped here;
            // but its Node's `next` field has been set to Link::E
            // 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::Empt

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

#[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));
```

```rust

        // 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`:

\`\`\`rust ,ignore // 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 r
a really bad version of Option:

```rust ,ignore
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:

```rust ,ignore use std::mem;

pub struct List { head: Link, }

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

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'

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

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

type Link = Option<Box<Node>>;

struct 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:

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

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

```text
> 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:

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

type Link = Option<Box<Node>>;

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

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

```text
> 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:

```rust ,ignore impl List { 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 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(); } } }

```
...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
Rust is *easy*. I'd like to make a particular shout-out to `new` w
even change:

```rust ,ignore
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:

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

~~~
```text
> 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 t

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:

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

~~~
It demotes the Option<T> to an Option to a reference to its intern
do this ourselves with an explicit match but *ugh no*. It does mea
need to do an extra dereference to cut through the extra indirecti
thankfully the `.` operator handles that for us.



```rust ,ignore
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`:

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

```

```text
lists::cargo build

```

EZ

Don't forget to test it:

```rust ,ignore

# [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));
```

}

```text
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:

```rust ,ignore

## [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));
```

}

~~~
```text
> 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 mu
100 |             value = 42
    |             ^^^^^^^^^^ cannot assign twice to immutable vari
~~~

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:

```rust ,ignore #[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));
}
```

~~~
```text
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`:

```rust ,ignore pub trait Iterator { type Item; fn next(&mut self) -> Option; }

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

~~~~
The reason Iterator yields `Option<Self::Item>` is because the int
coalesces the `has_next` and `get_next` concepts. When you have th
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
redundant checks and logic between `has_next` and `get_next`. Nice

Sadly, Rust has nothing like a `yield` statement (yet), so we're g
implement the logic ourselves. Also, there's actually 3 different
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
just implement IntoIter as a newtype wrapper around List:



```rust ,ignore
// 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:

```rust ,ignore

## [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);
```

}

````

```text
> 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:

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

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

impl Iterator for Iter { type Item = &T;

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

}

```

```text
> 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
inside a function signature, the problem may be with failing to ad
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:

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

impl<'a, T> List { 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.next.map(|node| { self.next = node.next.map(|node| &'a node); &'a node.elem }) } }
```

~~~
```text
> 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);
   |                                                  ^^^^ expecte

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:

```rust ,ignore // 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 ter
means is that the input must live at least as long as the output.
the output around for a long time, this will expand the region tha
be valid for. Once you stop using the output, the compiler will kn
the input to become invalid too.

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

Alright. So. Iter.

Let's roll back to the no lifetimes state:

```rust ,ignore
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:

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

// No lifetime here, List doesn't have any associated lifetimes impl List { // 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.

```text
cargo build

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

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

(╯°□°)╯︵ ┻━┻

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:

```rust ,ignore impl List { 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.next.map(|node| { self.next = node.next.map(|node| &*node); &node.elem }) } }
```

~~~
```text
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 re

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 co

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

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
~~~

(ノ■益■) ノ ┻━┻

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

impl List { 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
    })
}
```

}

~~~
```text
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) }
   |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ ex
   |
   = note: expected type `std::option::Option<&second::Node<T>>`
              found type `std::option::Option<&std::boxed::Box<sec

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

😭

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

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

impl List { 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
    })
}
```

}

````

```text
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*:

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

&node);

See, map is a generic function:

```rust ,ignore
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:

```rust ,ignore

## [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));
```

}
```

~~~~
```text
> 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:

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

~~~~

is equivalent to:

```rust ,ignore
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, `'_`:

```rust ,ignore impl List { 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
thing to say; surely it's identical to Iter!

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

The key insight comes from our implementation of Iterator for Iter

```rust ,ignore
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:

```rust ,ignore 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 li
of the input and the output! Why do we care? It means we can call 
over and over unconditionally!



```rust ,ignore
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:

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

impl List { 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
    })
}
```

}

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

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.

```rust ,ignore 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
```
```

~~~
the previous section, and we were just getting lucky that it worke

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

However for other types this is *garbage*. Integers have no
ownership semantics; they're just meaningless numbers! This is why
marked as Copy. Copy types are known to be perfectly copyable by a
As such, they have a super power: when moved, the old value *is* s
As a consequence, you can even move a Copy type out of a reference
replacement!

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

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



```rust ,ignore
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:

```rust ,ignore

## [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));
```

}

````

```text
> 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!

```
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();
        }
    }
}

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>>,
    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]
    #[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`:

```rust ,ignore // 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 ba
for free:

For instance, this isn't an uncommon workload to see with a persis

```text
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 &emdash; and indeed almost every other language &emdash; 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 &emdash; 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:

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

type Link = Option<Box<Node>>;

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

~~~
Can we just change Box to Rc?

```rust ,ignore
// 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 impor
  |
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*.

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

~~~
```text
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:

```rust ,ignore impl List { pub fn new() -> Self { List { head: None } } }
```

~~~

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

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

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

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

Alright, let's give it a shot:

```rust ,ignore
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:

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

~~~
```text
cargo build

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

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.

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

```
```text
> 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:

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

```
```text
> cargo build
```

Nice.

That's enough functionality that we can test it:

```rust ,ignore

## [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);

}
```

}
```

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

impl List { 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 ,ignore
#[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:

```rust ,ignore 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(); } } }

````

The problem is the body of the loop:

```rust ,ignore
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`:

```rust ,ignore impl Drop for List { 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; } } } }
```

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

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 filt
~~~

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!

```
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.
    }

    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`:

```rust ,ignore // 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 p
some logic that applies verbatim again.

Disclaimer: this chapter is basically a demonstration that this is

# Layout

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

```rust ,ignore
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:

~~~
```rust ,ignore use std::rc::Rc; use std::cell::RefCell;

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

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

struct Node { elem: T, next: Link, prev: Link, }
~~~

~~~
```text
> 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:

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

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

```
```text
> 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:

```rust ,ignore 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! } } }

~~~
```text
cargo build

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

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

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

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

```rust ,ignore
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:

```rust ,ignore pub fn pop_front(&mut self) -> Option { // 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 }) }

```
```text
> cargo build

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

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

```rust ,ignore pub fn pop_front(&mut self) -> Option { 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 }) }

```
```text
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 bor
```

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!

```rust ,ignore old_head.into_inner().elem

````

```text
> 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.

```rust ,ignore Rc::try_unwrap(old_head).unwrap().into_inner().elem
```

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

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

```text
> cargo build

error[E0599]: no method named `unwrap` found for type `std::result
  --> src/fourth.rs:64:38
   |
64 |             Rc::try_unwrap(old_head).unwrap().into_inner().el
   |                                      ^^^^^^
   |
   = note: the method `unwrap` exists but the following trait boun
           `std::rc::Rc<std::cell::RefCell<fourth::Node<T>>> : std
~~~

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`:

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

```
PLEASE.

```text
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):

` ` `rust ,ignore

[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);
}
```

}

~~~~
```text
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:

```rust ,ignore impl Drop for List { fn drop(&mut self) { while
```

self.pop_front().is_some() {} } }

~~~
```text
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!

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

~~~
Wait. Not this time.

```rust ,ignore
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 cur
```

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`:

```rust ,ignore 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
> If you break the rules, RefCell will just panic and crash the pr
> Why does it return these Ref and RefMut things? Well, they basic
> like `Rc`s but for borrowing. Also they keep the RefCell borrowe
> of scope. **We'll get to that later.**

It's later.
```
```

~~~
`Ref` and `RefMut` implement `Deref` and `DerefMut` respectively.
intents and purposes they behave *exactly* like `&T` and `&mut T`.
because of how those traits work, the reference that's returned is
to the lifetime of the Ref, and not the actual RefCell. This means
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
the RefCell that it's not borrowed anymore. So if we *did* manage
reference longer than the Ref existed, we could get a RefMut while
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, b
to keep this Ref thing around. But as soon as we return the refere
`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. Y
totally encapsulate the use of RefCells like that.

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

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

```text
> 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
help: possible candidates are found in other modules, you can impo
   |
1  | use core::cell::Ref;
   |
1  | use std::cell::Ref;
   |
```

Blurp. Gotta import some stuff.

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

```text
```text
> 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::N
   |
   = note: expected type `std::option::Option<std::cell::Ref<'_, T
              found type `std::option::Option<std::cell::Ref<'_, f
```

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*:

```rust ,ignore 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 w
I don't care about any of that. Also I don't think it's a proper m
there's no None-like case, but I digress.

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

```rust ,ignore
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.

```rust ,ignore

## [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:

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

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

```rust ,ignore 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 { 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> { self.tail.as_ref().map(|node| { Ref::map(node.borrow(), |node| &node.elem) }) }

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

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

~~~
And massively flesh out our tests:



```rust ,ignore
#[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`:

```rust ,ignore pub struct IntoIter(List);

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

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

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

Rust actually has an answer to this: `DoubleEndedIterator`. Double
*inherits* from Iterator (meaning all DoubleEndedIterator are Iter
requires one new method: `next_back`. It has the exact same signat
`next`, but it's supposed to yield elements from the other end. Th
of DoubleEndedIterator are super convenient for us: the iterator b
deque. You can consume elements from the front and back until the
converge, at which point the iterator is empty.

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

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

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

And let's test it out:

```rust ,ignore

## [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);
```

}

~~~


```text
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:

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

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

````
```text
> 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:

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

~~~
```text
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 th
154 |         self.0.take().map(|node_ref| {
155 |             self.0 = node_ref.next.as_ref().map(|head| head.
    |             ^^^^^^   -------- borrow is only valid in the cl
    |             |
    |             reference to `node_ref` escapes the closure body

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 `
154 |         self.0.take().map(|node_ref| {
155 |             self.0 = node_ref.next.as_ref().map(|head| head.
    |             ------   -------- borrow of `node_ref` occurs he
    |             |
    |             assignment requires that `node_ref` is borrowed
156 |             Ref::map(node_ref, |node| &node.elem)
    |                      ^^^^^^^^ move out of `node_ref` occurs
~~~

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.

Coincidentally, as of the moment I am writing this, the function we want was actually stabilized 2 days ago. That means it will be a few months before it hits the stable release. So let's continue along with the latest nightly build:

```rust ,ignore 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...

```rust ,ignore
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 th
...
159 |             self.0 = next.as_ref().map(|head| head.borrow())
    |             ^^^^^^   ---- borrow is only valid in the closur
    |             |
    |             reference to `next` escapes the closure body her
```

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

??????

```rust ,ignore fn next(&mut self) -> Option { 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
})
```

}

```

```text
error[E0308]: mismatched types
   --> src/fourth.rs:162:22
    |
162 |                 Some(Ref::map(next, |next| &**next.as_ref().
    |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    |
    = note: expected type `std::cell::Ref<'_, fourth::Node<_>>`
               found type `std::cell::Ref<'_, std::cell::RefCell<f
```

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*.

```rust
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 => {
```

```
                None => {
                    self.head.take();
                }
            }
            Rc::try_unwrap(old_tail).ok().unwrap().into_inner().el
        })
    }

    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().el
        })
    }

    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;
```

```rust
    #[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();
        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 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*.

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

```` ```rust ,ignore // 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 qu
mostly an augmentation of a stack in the world of linked lists. St
going to go from scratch because there's some fundamental issues w
address with layout and what-not.

# Layout

So what's a singly-linked queue like? Well, when we had a singly-l
we pushed onto one end of the list, and then popped off the same e
difference between a stack and a queue is that a queue pops off th
end. So from our stack implementation we have:

```text
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 memoize this work? 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 forwards, which is easy.

Let's try that:

```rust ,ignore use std::mem;

pub struct List { head: Link, tail: Link, // NEW! }

type Link = Option<Box<Node>>;

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

impl List { 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
            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 b
comfortable with this sort of thing. Not that you should necessari
to produce this code on the first try. I'm just skipping over some
trial-and-error we've had to deal with before. I actually made a t
writing this code that I'm not showing. You can only see me leave
`;` so many times before it stops being instructive. Don't worry,
plenty of *other* error messages!

```text
> 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
...
33 |         let old_tail = mem::replace(&mut self.tail, Some(new_
   |                                                          ----
...
38 |                 old_tail.next = Some(new_tail);
   |                                      ^^^^^^^^ value used here
~~~

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!

```rust ,ignore pub struct List { head: Link, tail: Option<&mut Node>, // NEW! }
```

type Link = Option<Box<Node>>;

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

impl List { 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 t
    let new_tail = match self.tail.take() {
        Some(old_tail) => {
            // If the old tail existed, update it to point to the
            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, exc
using some of that implicit return goodness to extract the tail re
wherever we stuff the actual Box.

```text
> 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 references in types lifetimes. 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`:

```rust ,ignore pub struct List<'a, T> { head: Link, tail: Option<&'a mut Node>, // NEW! }

type Link = Option<Box<Node>>;

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

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 t
    let new_tail = match self.tail.take() {
        Some(old_tail) => {
            // If the old tail existed, update it to point to the
            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;
}
```

}

~~~
```text
cargo build

error[E0495]: cannot infer an appropriate lifetime for autoref due
  --> src/fifth.rs:35:27
   |
35 |                 self.head.as_deref_mut()
   |                           ^^^^^^^^^^^^
   |
note: first, the lifetime cannot outlive the anonymous lifetime #1
  --> 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 al
...  |
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 defin
  --> 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..?

```rust ,ignore pub fn push(&'a mut self, elem: T) {

````
```text
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:

```rust ,ignore pub fn pop(&'a mut self) -> Option { // 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:

```rust ,ignore
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
  --> 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
  --> 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
  --> 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). I believe the reason is that Rust can't yet tell that the reference is into ourselves from just `push` and `pop` &emdash; or rather, Rust doesn't really have that notion at all. Reference-into-yourself failing to work is just an emergent behaviour.

As soon as we tried to *use* our list, everything quickly fell apart. When we call `push` or `pop`, we promptly store a reference to ourselves in ourselves and become *trapped*. We are literally borrowing ourselves.

Our `pop` implementation hints at why this could be really dangerous:

```rust ,ignore // ... if self.head.is_none() { self.tail = None; }
```

~~~
What if we forgot to do this? Then our tail would point to some no
had been removed from the list*. Such a node would be instantly fr
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
**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:

```rust ,ignore
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.

Let's be C everyone. Let's be C all day.

I'm home. I'm ready.

Hello `unsafe`.

## 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.

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 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.

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

Alright, back to basics. How do we construct our list?

Before we just did:

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

```
But we're not using Option for the `tail` anymore:

```text
> cargo build

error[E0308]: mismatched types
  --> src/fifth.rs:15:34
   |
15 |         List { head: None, tail: None }
   |                                  ^^^^ expected *-ptr, found e
   |
   = 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*.

```rust ,ignore use std::ptr;

// defns...

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

~~~
```text
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:

```rust ,ignore let raw_tail: *mut _ = &mut *new_tail;

```

We have all the info we need. We can translate our code into, appr
the previous reference version:

```rust ,ignore
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
        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 dereferen
```

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:

```rust ,ignore *self.tail.next = Some(new_tail);

```

```text
> 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 dereferen
```

Uuuugh operator precedence.

```rust ,ignore (*self.tail).next = Some(new_tail);
```

~~~
```text
> cargo build

error[E0133]: dereference of raw pointer is unsafe and requires un
  --> 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 c
~~~

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:

```rust ,ignore 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;

// Put the box in the right place, and then grab a reference to it
if !self.tail.is_null() {
    // If the old tail existed, update it to point to the new tail
    unsafe {
        (*self.tail).next = Some(new_tail);
    }
} else {
    // Otherwise, update the head to point to it
    self.head = Some(new_tail);
}

self.tail = raw_tail;
```

}

```text
> 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.

However this 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 that 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:

```rust ,ignore pub fn pop(&mut self) -> Option { 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
```
```

Again we see another case where safety is stateful. If we fail to
tail pointer in *this* function, we'll see no problems at all. How
subsequent calls to `push` will start writing to the dangling tail

Let's test it out:

```rust ,ignore
#[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 target/debug/lists-5c71138492ad4b4a

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. 8 passed; 0 failed; 0 ignored; 0 measured
```

Gold Star!

## Extra Junk

Now that `push` and `pop` are written, everything else is exactly the same as the stack case. Only operations that change the length of the list need to actually worry about the tail pointer.

So let's just steal all that from our second list (be sure to reverse the expected test output):

```rust ,ignore // ...

pub struct IntoIter(List);

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

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

impl List { // ...
```

```
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 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(); } } }

impl Iterator for IntoIter { type Item = T; fn next(&mut self) -> Option { self.0.pop() } }

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
    })
}
```

}

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 { // ...

```
#[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);
}
```

}

~~~
```text
> cargo test

     Running target/debug/lists-5c71138492ad4b4a

running 15 tests
test fifth::test::into_iter ... ok
test fifth::test::basics ... 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. 11 passed; 0 failed; 0 ignored; 0 measured
~~~

Shout-outs to copy-paste programming.

At first I thought we'd have to mess around with IntoIter, but we still conveniently pop in iteration order!

## 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!

We also notably *didn't* have to write any crazy Rc or RefCell stuff.

```
use std::ptr;
```

```
pub struct List<T> {
    head: Link<T>,
    tail: *mut Node<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, tail: ptr::null_mut() }
    }

    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() {
            unsafe {
                (*self.tail).next = Some(new_tail);
            }
        } else {
            self.head = Some(new_tail);
        }

        self.tail = raw_tail;
    }

    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
        })
    }

    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() }
    }
}

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

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

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

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();
        }
    }
}

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> {
        self.next.map(|node| {
            self.next = node.next.as_deref();
            &node.elem
        })
    }
}

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(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);
    }
}
```

## An Ok Unsafe Doubly-Linked Deque

Nope, still haven't written this one! It's really just not that much more instructive.

Read The Rustonomicon and the source for std::collections::LinkedList if you really want more!

## 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. TODO: BList?
3. TODO: SkipList?
4. TODO: std::channel? -- That's like another whole chapter. Or 3.

## 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:

```rust ,ignore // lib.rs // ... pub mod silly1; // NEW!

~~~~
```rust ,ignore
// silly1.rs
use 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:

```rust ,ignore pub struct Stack { head: Link, }

type Link = Option<Box<Node>>;

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

impl Stack { 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 Drop for Stack { 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:

```rust ,ignore
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:

```rust ,ignore pub struct List { left: Stack, right: Stack, }

impl List { fn new() -> Self { List { left: Stack::new(), right: Stack::new() } } }
```

~~~
And we can do the usual stuff:


```rust ,ignore
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
pub fn peek_right_mut(&mut self) -> Option<&mut T> { self.right.pe
~~~

But most interestingly, we can walk around!

```rust ,ignore 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
managed to move. Now let's test this baby out:

```rust ,ignore
#[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,

        while list.go_left() {}                     // [_, 0, 2, 3, 4,

        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!