Jan Nils Ferner, Daniel Durante

Rust Standard Library Cookbook

Over 75 recipes to leverage the power of Rust

Packt

Table of Contents

1
Learning the Basics

In this chapter, we will cover the following recipes:

- Concatenating strings
- Using the format! macro
- Providing a default implementation
- Using the constructor pattern
- Using the builder pattern
- Parallelism through simple threads
- Generating random numbers
- Querying with regexes
- Accessing the command line
- Interacting with environment variables
- Reading from stdin
- Accepting a variable number of arguments

Introduction

There are some code snippets and patterns of thought that prove time and again to be the bread and butter of a certain programming language. We will start this book by looking at a handful of such techniques in Rust. They are so quintessential for elegant and flexible code that you will use at least some of them in just about any project you tackle.

The next chapters will then build on this foundation and work hand in hand with Rust's zero costs abstractions, which are as powerful as the ones in higher-level languages. We are also going to look at the intricate inner aspects of the standard library and implement our own similar constructs with the help of fearless concurrency and careful use of `unsafe` blocks, which enable us to work at the same low level that some system languages, such as C, operate at.

Concatenating strings

String manipulation is typically a bit less straightforward in system programming languages than in scripting languages, and Rust is no exception. There are multiple ways to do it, all managing the involved resources differently.

Getting ready

We will assume for the rest of the book that you have an editor open, the newest Rust compiler ready, and a command line available. As of the time of writing, the newest version is `1.24.1`. Because of Rust's strong guarantees about backward compatibility, you can rest assured that all of the recipes shown (with the exception of `Chapter 10`, *Using Experimental Nightly Features*) are always going to work the same way. You can download the newest compiler with its command-line tools at `https://www.rustup.rs`.

How to do it...

1. Create a Rust project to work on during this chapter with `cargo new chapter-one`
2. Navigate to the newly created `chapter-one` folder. For the rest of this chapter, we will assume that your command line is currently in this directory
3. Inside the `src` folder, create a new folder called `bin`
4. Delete the generated `lib.rs` file, as we are not creating a library
5. In the `src/bin` folder, create a file called `concat.rs`
6. Add the following code and run it with `cargo run --bin concat`:

```
fn main() {
 by_moving();
 by_cloning();
 by_mutating();
5  }
6
7  fn by_moving() {
8   let hello = "hello ".to_string();
9   let world = "world!";
10
11   // Moving hello into a new variable
12   let hello_world = hello + world;
13   // Hello CANNOT be used anymore
14   println!("{}", hello_world); // Prints "hello world!"
15  }
16
17  fn by_cloning() {
18   let hello = "hello ".to_string();
19   let world = "world!";
20
21   // Creating a copy of hello and moving it into a new variable
22   let hello_world = hello.clone() + world;
23   // Hello can still be used
24   println!("{}", hello_world); // Prints "hello world!"
25  }
26
27  fn by_mutating() {
28   let mut hello = "hello ".to_string();
29   let world = "world!";
30
31   // hello gets modified in place
32   hello.push_str(world);
33   // hello is both usable and modifiable
34   println!("{}", hello); // Prints "hello world!"
35  }
```

How it works...

In all functions, we start by allocating memory for a string of variable length.
We do this by creating a string slice (`&str`) and applying the `to_string` function on it [8, 18 and 28].
The first way to concatenate strings in Rust, as shown in the `by_moving` function, is by taking said allocated memory and **moving** it, together with an additional string slice, into a new variable [12]. This has a couple of advantages:

- It's very straightforward and clear to look at, as it follows the common programming convention of concatenating with the + operator

- It uses only immutable data. Remember to always try to write code in a style that creates as little stateful behavior as possible, as it results in more robust and reusable code bases
- It reuses the memory allocated by `hello` [8], which makes it very performant

As such, this way of concatenating should be preferred whenever possible.
So, why would we even list other ways to concatenate strings? Well, I'm glad you asked, dear reader. Although elegant, this approach comes with two downsides:

- `hello` is no longer usable after line [12], as it was moved. This means you can no longer read it in any way
- Sometimes you may actually prefer mutable data in order to use state in small, contained environments

The two remaining functions address one concern each.
`by_cloning`[17] looks nearly identical to the first function, but it clones the allocated string [22] into a temporary object, allocating new memory in the process, which it then moves, leaving the original `hello` untouched and still accessible. Of course, this comes at the price of redundant memory allocations at runtime.

`by_mutating`[27] is the stateful way of solving our problem. It performs the involved memory management in-place, which means that the performance should be the same as in `by_moving`. In the end, it leaves `hello` mutable, ready for further changes. You may notice that this function doesn't look as elegant as the others, as it doesn't use the + operator. This is intentional, as Rust tries to push you through its design towards moving data in order to create new variables without mutating existing ones. As mentioned before, you should only do this if you really need mutable data or want to introduce state in a very small and manageable context.

Using the format! macro

There is an additional way to combine strings, which can also be used to combine them with other data types, such as numbers.

How to do it...

1. In the `src/bin` folder, create a file called `format.rs`
2. Add the following code and run it with `cargo run --bin format`

```
fn main() {
  let colour = "red";
  // The '{}' it the formatted string gets replaced by the parameter
  let favourite = format!("My favourite colour is {}", colour);
  println!("{}", favourite);

  // You can add multiple parameters, which will be
  // put in place one after another
  let hello = "hello ";
  let world = "world!";
  let hello_world = format!("{}{}", hello, world);
  println!("{}", hello_world); // Prints "hello world!"

  // format! can concatenate any data types that
  // implement the 'Display' trait, such as numbers
  let favourite_num = format!("My favourite number is {}", 42);
  println!("{}", favourite_num); // Prints "My favourite number is 42"

  // If you want to include certain parameters multiple times
  // into the string, you can use positional parameters
  let duck_duck_goose = format!("{0}, {0}, {0}, {1}!", "duck", "goose");
  println!("{}", duck_duck_goose); // Prints "duck, duck, duck, goose!"

  // You can even name your parameters!
  let introduction = format!(
    "My name is {surname}, {forename} {surname}",
    surname="Bond",
    forename="James"
  );
  println!("{}", introduction) // Prints "My name is Bond, James Bond"
 }
```

How it works...

The `format!` macro combines strings by accepting a format string filled with formatting parameters (example, `{}`, `{0}`, or `{foo}`) and a list of arguments, which are then inserted into the *placeholders*.
We are now going to show this on the example in line [16]:

```
format!("My favourite number is {}", 42);
```

Let's break down the preceding line of code:

- `"My favourite number is {}"` is the format string
- `{}` is the formatting parameter
- `42` is the argument

As demonstrated, `format!` works not only with strings, but also with numbers. In fact, it works with all `structs` that implement the `Display` trait. This means that, by providing such an implementation by yourself, you can easily make your own data structures printable however you want.

By default, `format!` replaces one parameter after another. If you want to override this behavior, you can use positional parameters like `{0}` [21]. With the knowledge that the positions are zero-indexed, the behavior here is pretty straightforward, `{0}` gets replaced by the first argument, `{1}` gets replaced by the second, and so on.

At times, this can become a bit unwieldy when using a lot of parameters. For this purpose, you can use named arguments [26], just like in Python. Keep in mind that all of your unnamed arguments have to be placed before your named ones. For example, the following is invalid:

```
format!("{message} {}", message="Hello there,", "friendo")
```

It should be rewritten as:

```
format!("{message} {}", "friendo", message="Hello there,")
 // Returns "hello there, friendo"
```

There's more...

You can combine positional parameters with normal ones, but it's probably not a good idea, as it can quite easily become confusing to look at. The behavior, in this case, is as follows—imagine that `format!` internally uses a counter to determine which argument is the next to be placed. This counter is increased whenever `format!` encounters a `{}` *without* a position in it. This rule results in the following:

```
format!("{1} {} {0} {}", "a", "b") // Returns "b a a b"
```

There are also a ton of extra formatting options if you want to display your data in different formats. `{:?}` prints the implementation of the `Debug` trait for the respective argument, often resulting in a more verbose output. `{:.*}` lets you specify the decimal precision of floating point numbers via the argument, like so:

```
format!("{:.*}", 2, 1.234567) // Returns "1.23"
```

For a complete list, visit `https://doc.rust-lang.org/std/fmt/`.

All of the information in this recipe applies to `println!` and `print!` as well, as it is essentially the same macro. The only difference is that `println!` doesn't return its processed string but instead, well, prints it!

Providing a default implementation

Often, when dealing with structures that represent configurations, you don't care about certain values and just want to silently assign them a standard value.

How to do it...

1. In the `src/bin` folder, create a file called `default.rs`
2. Add the following code and run it with `cargo run --bin default`:

```
fn main() {
  // There's a default value for nearly every primitive type
  let foo: i32 = Default::default();
  println!("foo: {}", foo); // Prints "foo: 0"


  // A struct that derives from Default can be initialized like this
  let pizza: PizzaConfig = Default::default();
  // Prints "wants_cheese: false
  println!("wants_cheese: {}", pizza.wants_cheese);

  // Prints "number_of_olives: 0"
  println!("number_of_olives: {}", pizza.number_of_olives);

  // Prints "special_message: "
  println!("special message: {}", pizza.special_message);

  let crust_type = match pizza.crust_type {
    CrustType::Thin => "Nice and thin",
    CrustType::Thick => "Extra thick and extra filling",
  };
  // Prints "crust_type: Nice and thin"
  println!("crust_type: {}", crust_type);


  // You can also configure only certain values
  let custom_pizza = PizzaConfig {
    number_of_olives: 12,
    ..Default::default()
  };

  // You can define as many values as you want
  let deluxe_custom_pizza = PizzaConfig {
    number_of_olives: 12,
    wants_cheese: true,
    special_message: "Will you marry me?".to_string(),
    ..Default::default()
  };

}

#[derive(Default)]
struct PizzaConfig {
  wants_cheese: bool,
  number_of_olives: i32,
  special_message: String,
  crust_type: CrustType,
}

// You can implement default easily for your own types
enum CrustType {
  Thin,
  Thick,
54  }
55  impl Default for CrustType {
56    fn default() -> CrustType {
57      CrustType::Thin
58    }
59  }
```

How it works...

Nearly every type in Rust has a `Default` implementation. When you define your own `struct` that only contains elements that already have a `Default`, you have the option to derive from `Default` as well [42]. In the case of enums or complex structs, you can easily write your own implementation of `Default` instead [55], as there's only one method you have to provide. After this, the `struct` returned by `Default::default()` is implicitly inferrable as yours, if you tell the compiler what your type actually is. This is why in line [3] we have to write `foo: i32`, or else Rust wouldn't know what type the default object actually should become.

If you only want to specify some elements and leave the others at the default, you can use the syntax in line [29]. Keep in mind that you can configure and skip as many values as you want, as shown in lines [33 to 37].

Using the constructor pattern

You may have asked yourself how to idiomatically initialize complex structs in Rust, considering it doesn't have constructors. The answer is simple, there is a constructor, it's just a convention rather than a rule. Rust's standard library uses this pattern very often, so we need to understand it if we want to use the std effectively.

Getting ready

In this recipe, we are going to talk about how a **user** interacts with a `struct`. When we say *user* in this context, we don't mean the end user that clicks on the GUI of the app you're writing. We're referring to the programmer that instantiates and manipulates the `struct`.

How to do it...

1. In the `src/bin` folder, create a file called `constructor.rs`
2. Add the following code and run it with `cargo run --bin constructor`:

```
fn main() {
  // We don't need to care about
  // the internal structure of NameLength
  // Instead, we can just call it's constructor
  let name_length = NameLength::new("John");

  // Prints "The name 'John' is '4' characters long"
  name_length.print();
}

 struct NameLength {
   name: String,
   length: usize,
 }

 impl NameLength {
   // The user doesn't need to setup length
   // We do it for him!
   fn new(name: &str) -> Self {
     NameLength {
       length: name.len(),
       name,
     }
   }

   fn print(&self) {
     println!(
       "The name '{}' is '{}' characters long",
         self.name,
         self.length
     );
   }
 }
```

How it works...

If a `struct` provides a method called `new` that returns `Self`, the user of the `struct` will not configure or depend upon the members of the `struct`, as they are considered to be in an internal *hidden* state.

In other words, if you see a `struct` that has a `new` function, always use it to create the structure.
This has the nice effect of enabling you to change as many members of the struct as you want without the user noticing anything, as they are not supposed to look at them anyway.

The other reason to use this pattern is to guide the user to the correct way of instantiating a `struct`. If one has nothing but a big list of members that have to be filled with values, one might feel a bit lost. If one, however, has a method with only a few self-documenting parameters, it feels way more inviting.

There's more...

You might have noticed that for our example we really didn't need a `length` member and could have just calculated a length whenever we print. We use this pattern anyway, to illustrate the point of its usefulness in hiding implementations. Another good use for it is when the members of a `struct` themselves have their own constructors and one needs to *cascade* the constructor calls. This happens, for example, when we have a `Vec` as a member, as we will see later in the book, in the, *Using a vector* section in `Chapter 2`, *Working with Collections*.

Sometimes, your structs might need more than one way to initialize themselves. When this happens, try to still provide a `new()` method as your default way of construction and name the other options according to how they differ from the default. A good example of this is again vector, which not only provides a `Vec::new()` constructor but also a `Vec::with_capacity(10)`, which initializes it with enough space for `10` items. More on that again in the Using a vector section in `Chapter 2`, *Working with Collections*.

When accepting a kind of string (either `&str`, that is, a borrowed string slice, or `String`, that is, an owned string) with plans to store it in your `struct`, like we do in our example, also considering a `Cow`. No, not the big milk animal friends. A `Cow` in Rust is a *Clone On Write* wrapper around a type, which means that it will try to borrow a type for as long as possible and only make an owned clone of the data when absolutely necessary, which happens at the first mutation. The practical effect of this is that, if we rewrote our `NameLength` struct in the following way, it would not care whether the called passed a `&str` or a `String` to it, and would instead try to work in the most efficient way possible:

```
use std::borrow::Cow;
struct NameLength<'a> {
    name: Cow<'a, str>,
    length: usize,
}

impl<'a> NameLength<'a> {
    // The user doesn't need to setup length
    // We do it for him!
    fn new<S>(name: S) -> Self
    where
        S: Into<Cow<'a, str>>,
    {
        let name: Cow<'a, str> = name.into();
        NameLength {
            length: name.len(),
            name,
        }
    }

    fn print(&self) {
        println!(
            "The name '{}' is '{}' characters long",
            self.name, self.length
        );
    }
}
```

If you want to read more about `Cow`, check out this easy-to-understand blog post by Joe Wilm: `https://jwilm.io/blog/from-str-to-cow/`.

The `Into` trait used in the `Cow` code is going to be explained in the Converting types into each other section in `Chapter 5`, *Advanced Data Structures*.

See also

- *Using a vector* recipe in `Chapter 2`, *Working with Collections*
- *Converting types into each other* recipe in `Chapter 5`, *Advanced Data Structures*

Using the builder pattern

Sometimes you need something between the customization of the constructor and the implicitness of the default implementation. Enter the builder pattern, another technique frequently used by the Rust standard library, as it allows a caller to fluidly chain together configurations that they care about and lets them ignore details that they don't care about.

How to do it...

1. In the `src/bin` folder, create a file called `builder.rs`
2. Add all of the following code and run it with `cargo run --bin builder`:

```
fn main() {
  // We can easily create different configurations
  let normal_burger = BurgerBuilder::new().build();
  let cheese_burger = BurgerBuilder::new()
    .cheese(true)
    .salad(false)
    .build();
  let veggie_bigmac = BurgerBuilder::new()
    .vegetarian(true)
    .patty_count(2)
    .build();

  if let Ok(normal_burger) = normal_burger {
    normal_burger.print();
  }
  if let Ok(cheese_burger) = cheese_burger {
    cheese_burger.print();
  }
  if let Ok(veggie_bigmac) = veggie_bigmac {
    veggie_bigmac.print();
  }

  // Our builder can perform a check for
  // invalid configurations
  let invalid_burger = BurgerBuilder::new()
    .vegetarian(true)
    .bacon(true)
    .build();
  if let Err(error) = invalid_burger {
    println!("Failed to print burger: {}", error);
  }

  // If we omit the last step, we can reuse our builder
  let cheese_burger_builder = BurgerBuilder::new().cheese(true);
  for i in 1..10 {
    let cheese_burger = cheese_burger_builder.build();
    if let Ok(cheese_burger) = cheese_burger {
      println!("cheese burger number {} is ready!", i);
      cheese_burger.print();
    }
  }
}
```

This is the configurable object:

```
struct Burger {
   patty_count: i32,
   vegetarian: bool,
   cheese: bool,
   bacon: bool,
   salad: bool,
}
impl Burger {
   // This method is just here for illustrative purposes
   fn print(&self) {
       let pretty_patties = if self.patty_count == 1 {
           "patty"
       } else {
           "patties"
       };
       let pretty_bool = |val| if val { "" } else { "no " };
       let pretty_vegetarian = if self.vegetarian { "vegetarian
"
        }
       else { "" };
       println!(
           "This is a {}burger with {} {}, {}cheese, {}bacon and
           {}salad",
           pretty_vegetarian,
           self.patty_count,
            pretty_patties,
            pretty_bool(self.cheese),
            pretty_bool(self.bacon),
            pretty_bool(self.salad)
        )
    }
}
```

And this is the builder itself. It is used to configure and create a `Burger`:

```
struct BurgerBuilder {
  patty_count: i32,
  vegetarian: bool,
  cheese: bool,
  bacon: bool,
  salad: bool,
}
impl BurgerBuilder {
  // in the constructor, we can specify
  // the standard values
  fn new() -> Self {
    BurgerBuilder {
      patty_count: 1,
      vegetarian: false,
      cheese: false,
      bacon: false,
      salad: true,
    }
  }

  // Now we have to define a method for every
  // configurable value
  fn patty_count(mut self, val: i32) -> Self {
    self.patty_count = val;
    self
  }

  fn vegetarian(mut self, val: bool) -> Self {
    self.vegetarian = val;
    self
  }
  fn cheese(mut self, val: bool) -> Self {
    self.cheese = val;
    self
  }
  fn bacon(mut self, val: bool) -> Self {
    self.bacon = val;
    self
  }
  fn salad(mut self, val: bool) -> Self {
    self.salad = val;
    self
  }

  // The final method actually constructs our object
  fn build(&self) -> Result<Burger, String> {
    let burger = Burger {
      patty_count: self.patty_count,
      vegetarian: self.vegetarian,
      cheese: self.cheese,
      bacon: self.bacon,
      salad: self.salad,
  };
  // Check for invalid configuration
  if burger.vegetarian && burger.bacon {
    Err("Sorry, but we don't server vegetarian bacon
         yet".to_string())
    } else {
      Ok(burger)
    }
  }
}
```

How it works...

Whew, that's a lot of code! Let's start by breaking it up.

In the first part, we illustrate how to use this pattern to effortlessly configure a complex object. We do this by relying on sensible standard values and only specifying what we really care about:

```
let normal_burger = BurgerBuilder::new().build();
let cheese_burger = BurgerBuilder::new()
    .cheese(true)
    .salad(false)
    .build();
let veggie_bigmac = BurgerBuilder::new()
    .vegetarian(true)
    .patty_count(2)
    .build();
```

The code reads pretty nicely, doesn't it?

In our version of the builder pattern, we return the object wrapped in a `Result` in order to tell the world that there are certain invalid configurations and that our builder might not always be able to produce a valid product. Because of this, we have to check the validity of our burger before accessing it[7, 10 and 13].

Our invalid configuration is `vegetarian(true)` and `bacon(true)`. Unfortunately, our restaurant doesn't serve vegetarian bacon yet! When you start the program, you will see that the following line will print an error:

```
        if let Err(error) = invalid_burger {
            println!("Failed to print burger: {}", error);
        }
```

If we omit the final `build` step, we can reuse the builder in order to build as many objects as we want. [25 to 32]

Let's see how we implemented all of this. The first thing after the `main` function is the definition of our `Burger` struct. No surprises here, it's just plain old data. The `print` method is just here to provide us with some nice output during runtime. You can ignore it if you want.

The real logic is in the `BurgerBuilder`[64]. It should have one member for every value you want to configure. As we want to configure every aspect of our burger, we will have the exact same members as `Burger`. In the constructor [74], we can specify some default values. We then create one method for every configuration. In the end, in `build()` [109], we first perform some error checking. If the configuration is OK, we return a `Burger` made out of all of our members [121]. Otherwise, we return an error [119].

There's more...

If you want your object to be constructable without a builder, you could also provide `Burger` with a `Default` implementation. `BurgerBuilder::new()` could then just return `Default::default()`.

In `build()`, if your configuration can inherently not be invalid, you can, of course, return the object directly without wrapping it in a `Result`.

Parallelism through simple threads

Every year, parallelism and concurrency become more important as processors tend to have more and more physical cores. In most languages, writing parallel code is tricky. Very tricky. Not so in Rust, as it has been designed around the principle of *fearless concurrency* since the beginning.

How to do it...

1. In the `src/bin` folder, create a file called `parallelism.rs`
2. Add the following code and run it with `cargo run --bin parallelism`

```
use std::thread;

fn main() {
  // Spawning a thread lets it execute a lambda
  let child = thread::spawn(|| println!("Hello from a new
  thread!"));
  println!("Hello from the main thread!");
  // Joining a child thread with the main thread means
  // that the main thread waits until the child has
  // finished it's work
  child.join().expect("Failed to join the child thread");

  let sum = parallel_sum(&[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
  println!("The sum of the numbers 1 to 10 is {}", sum);
}

// We are going to write a function that
// sums the numbers in a slice in parallel
fn parallel_sum(range: &[i32]) -> i32 {
  // We are going to use exactly 4 threads to sum the numbers
  const NUM_THREADS: usize = 4;

  // If we have less numbers than threads,
  // there's no point in multithreading them
  if range.len() < NUM_THREADS {
    sum_bucket(range)
  } else {
      // We define "bucket" as the amount of numbers
      // we sum in a single thread
      let bucket_size = range.len() / NUM_THREADS;
      let mut count = 0;
      // This vector will keep track of our threads
      let mut threads = Vec::new();
      // We try to sum as much as possible in other threads
      while count + bucket_size < range.len() {
        let bucket = range[count..count +
                           bucket_size].to_vec();
        let thread = thread::Builder::new()
          .name("calculation".to_string())
          .spawn(move || sum_bucket(&bucket))
          .expect("Failed to create the thread");
        threads.push(thread);

        count += bucket_size
    }
    // We are going to sum the rest in the main thread
    let mut sum = sum_bucket(&range[count..]);

    // Time to add the results up
    for thread in threads {
      sum += thread.join().expect("Failed to join thread");
    }
    sum
  }
}

 // This is the function that will be executed in the threads
 fn sum_bucket(range: &[i32]) -> i32 {
   let mut sum = 0;
   for num in range {
     sum += *num;
   }
    sum
 }
```

How it works...

You can create a new thread by calling `thread::spawn`, which will then begin executing the provided lambda. This will return a `JoinHandle`, which you can use to, well, join the thread. Joining a thread means waiting for the thread to finish its work. If you don't join a thread, you have no guarantee of it actually ever finishing. This might be valid though when setting up threads to do tasks that never complete, such as listening for incoming connections.

Keep in mind that you cannot predetermine the order in which your threads will complete any work. In our example, it is impossible to foretell whether *Hello from a new thread!* or *Hello from the main thread!* is going to be printed first, although most of the time it will probably be the main thread, as the operating system needs to put some effort into spawning a new thread. This is the reason why small algorithms can be faster when not executed in parallel. Sometimes, the overhead of letting the OS spawn and manage new threads is just not worth it.

As demonstrated by line [49], joining a thread will return a `Result` that contains the value your lambda returned.

Threads can also be given names. Depending on your OS, in case of a crash, the name of the responsible thread will be displayed. In line [37], we call our new summation threads *calculation*. If one of them were to crash, we would be able to quickly identify the issue. Try it out for yourself, insert a call to `panic!();` at the beginning of `sum_bucket` in order to intentionally crash the program and run it. If your OS supports named threads, you will now be told that your thread *calculation* panicked with an *explicit panic*.

`parallel_sum` is a function that takes a slice of integers and adds them together in parallel on four threads. If you have limited experience in working with parallel algorithms, this function will be hard to grasp at first. I invite you to copy it by hand into your text editor and play around with it in order to get a grasp on it. If you still feel a bit lost, don't worry, we will revisit parallelism again later.

Adapting algorithms to run in parallel normally comes at the risk of data races. A data race is defined as the behavior in a system where the output is dependent on the random timing of external events. In our case, having a data race would mean that multiple threads try to access and modify a resource at the same time. Normally, programmers have to analyze their usage of resources and use external tools in order to catch all of the data races. In contrast, Rust's compiler is smart enough to catch data races at compile time and stops if it finds one. This is the reason why we had to call `.to_vec()` in line [35]:

```
        let bucket = range[count..count + bucket_size].to_vec();
```

We will cover vectors in a later recipe (the *Using a vector* section in `Chapter 2`, *Working with Collections*), so if you're curious about what is happening here, feel free to jump to `Chapter 2`, *Working with Collections* and come back again. The essence of it is that we're copying the data into `bucket`. If we instead passed a reference into `sum_bucket` in our new thread, we would have a problem, the memory referenced by `range` is only guaranteed to live inside of `parallel_sum`, but the threads we spawn are allowed to outlive their parent threads. This would mean that in theory, if we didn't `join` the threads at the right time, `sum_bucket` might get unlucky and get called late enough for `range` to be invalid.

This would then be a data race, as the outcome of our function would depend on the uncontrollable sequence in which our operating system decides to launch the threads.

But don't just take my word for it, try it yourself. Simply replace the aforementioned line with `let bucket = &range[count..count + bucket_size];` and try to compile it.

There's more...

If you're experienced with parallelism, you might have noticed how suboptimal our algorithm here is. This is intentional, as the elegant and efficient way of writing `parallel_sum` would require using techniques we have not discussed yet. We will revisit this algorithm in `Chapter 7`, *Parallelism and Rayon*, and rewrite it in a professional manner. In that chapter, we will also learn how to concurrently modify resources using locks.

See also

- *Access resources in parallel with RwLocks*, recipe in `Chapter 7`, *Parallelism and Rayon*

Generating random numbers

As described in the preface, the Rust core team left some functionality intentionally out of the standard and put it into its own external crate. Generating pseudo-random numbers is one such functionality.

How to do it...

1. Open the `Cargo.toml` file that was generated earlier for you
2. Under `[dependencies]`, add the following line:

```
rand = "0.3"
```

3. If you want, you can go to rand's crates.io page (`https://crates.io/crates/rand`) to check for the newest version and use that one instead
4. In the `bin` folder, create a file called `rand.rs`
5. Add the following code and run it with `cargo run --bin rand`:

```
extern crate rand;

fn main() {
  // random_num1 will be any integer between
  // std::i32::MIN and std::i32::MAX
  let random_num1 = rand::random::<i32>();
  println!("random_num1: {}", random_num1);
  let random_num2: i32 = rand::random();
  println!("random_num2: {}", random_num2);
  // The initialization of random_num1 and random_num2
  // is equivalent.

  // Every primitive data type can be randomized
  let random_char = rand::random::<char>();
  // Altough random_char will probably not be
  // representable on most operating systems
  println!("random_char: {}", random_char);


  use rand::Rng;
  // We can use a reusable generator
  let mut rng = rand::thread_rng();
  // This is equivalent to rand::random()
  if rng.gen() {
    println!("This message has a 50-50 chance of being
              printed");
  }
  // A generator enables us to use ranges
  // random_num3 will be between 0 and 9
  let random_num3 = rng.gen_range(0, 10);
  println!("random_num3: {}", random_num3);

  // random_float will be between 0.0 and 0.999999999999...
  let random_float = rng.gen_range(0.0, 1.0);
  println!("random_float: {}", random_float);

  // Per default, the generator uses a uniform distribution,
  // which should be good enough for nearly all of your
  // use cases. If you require a particular distribution,
  // you specify it when creating the generator:
```

```
40     let mut chacha_rng = rand::ChaChaRng::new_unseeded();
41     let random_chacha_num = chacha_rng.gen::<i32>();
42     println!("random_chacha_num: {}", random_chacha_num);
43   }
```

How it works...

Before you can use `rand`, you have to tell Rust that you're using the `crate` by writing:

```
extern crate rand;
```

After that, `rand` will provide a random generator. We can access it by either calling `rand::random();` [6] or by accessing it directly with `rand::thread_rng();` [22].

If we go the first route, the generator will need to be told what type to generate. You can either explicitly state the type in the method call [6] or annotate the type of the resulting variable [8]. Both are equal and result in the exact same thing. Which one you use is up to you. In this book, we will use the first convention.

As you can see in lines [29 and 33], you need neither if the type is unambiguous in the called context.

The generated value will be between its type's `MIN` and `MAX` constants. In the case of `i32`, this would be `std::i32::MIN` and `std::i32::MAX`, or, in concrete numbers, -2147483648 and 2147483647. You can verify these numbers easily by calling the following:

```
println!("min: {}, max: {}", std::i32::MIN, std::i32::MAX);
```

As you can see, these are very big numbers. For most purposes, you will probably want to define custom limits. You can go the second route discussed earlier and use `rand::Rng` for that[22]. It has a `gen` method, which is actually implicitly called by `rand::random()`, but also a `gen_range()` that accepts a minimum and maximum value. Keep in mind that this range is non-inclusive, which means that the maximum value can never be reached. This is why in line [29], `rng.gen_range(0, 10)` will only generate the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, without the 10.

All of the described ways of generating random values use **uniform distribution**, which means that every number in the range has the same chance of being generated. In some contexts, it makes sense to use other distributions. You can specify a generator's distribution during its creation[40]. As of the time of publication, the rand crate supports the ChaCha and ISAAC distributions.

There's more...

If you want to randomly populate an entire `struct`, you use the `rand_derive` helper crate in order to derive it from **Rand**. You can then generate your own `struct`, just as you would generate any other type.

Querying with regexes

When parsing simple data formats, it is often easier to write regular expressions (or *regex* for short) than use a parser. Rust has pretty decent support for this through its `regex` crate.

Getting ready

In order to really understand this chapter, you should be familiar with regexes. There are countless free online resources for this, like regexone (`https://www.regexone.com/`).

This recipe will not conform to clippy, as we kept the regexes intentionally *too* simple because we want to keep the focus of the recipe on the code, not the regex. Some of the examples shown could have been rewritten to use `.contains()` instead.

How to do it...

1. Open the `Cargo.toml` file that was generated earlier for you
2. Under `[dependencies]`, add the following line:

```
regex = "0.2"
```

3. If you want, you can go to regex's crates.io page (`https://crates.io/crates/regex`) to check for the newest version and use that one instead
4. In the `bin` folder, create a file called `regex.rs`
5. Add the following code and run it with `cargo run --bin regex`:

```
extern crate regex;

fn main() {
```

```
use regex::Regex;
// Beginning a string with 'r' makes it a raw string,
// in which you don't need to escape any symbols
let date_regex =
  Regex::new(r"^\d{2}.\d{2}.\d{4}$").expect("Failed
    to create regex");
let date = "15.10.2017";
// Check for a match
let is_date = date_regex.is_match(date);
println!("Is '{}' a date? {}", date, is_date);

// Let's use capture groups now
let date_regex = Regex::new(r"(\d{2}).(\d{2})
  .(\d{4})").expect("Failed to create regex");
let text_with_dates = "Alan Turing was born on 23.06.1912 and
    died on 07.06.1954. \
  A movie about his life called 'The Imitation Game' came out
    on 14.11.2017";
// Iterate over the matches
for cap in date_regex.captures_iter(text_with_dates) {
  println!("Found date {}", &cap[0]);
  println!("Year: {} Month: {} Day: {}", &cap[3], &cap[2],
    &cap[1]);
}
// Replace the date format
println!("Original text:\t\t{}", text_with_dates);
let text_with_indian_dates =
  date_regex.replace_all(text_with_dates, "$1-$2-$3");
println!("In indian format:\t{}", text_with_indian_dates);

// Replacing groups is easier when we name them
// ?P<somename> gives a capture group a name
let date_regex = Regex::new(r"(?P<day>\d{2}).(?P<month>\d{2})
  .(?P<year>\d{4})")
  .expect("Failed to create regex");
let text_with_american_dates =
  date_regex.replace_all(text_with_dates,
    "$month/$day/$year");
println!("In american format:\t{}",
text_with_american_dates);
let rust_regex = Regex::new(r"(?i)rust").expect("Failed to
  create regex");
println!("Do we match RuSt? {}",
rust_regex.is_match("RuSt"));
use regex::RegexBuilder;
let rust_regex = RegexBuilder::new(r"rust")
  .case_insensitive(true)
  .build()
```

```
39      .expect("Failed to create regex");
40    println!("Do we still match RuSt? {}",
        rust_regex.is_match("RuSt"));
41  }
```

How it works...

You can construct a regex object by calling `Regex::new()` with a valid regex string[7]. Most of the time, you will want to pass a *raw string* in the form of `r"..."`. Raw means that all symbols in the string are taken at literal value without being escaped. This is important because of the backslash (\) character that is used in regex to represent a couple of important concepts, such as digits(`\d`) or whitespace (`\s`). However, Rust already uses the backslash to escape special *non-printable* symbols, such as the newline (`\n`) or the tab (`\t`)[23]. If we wanted to use a backslash in a normal string, we would have to escape it by repeating it (`\\`). Or the regex on line [14] would have to be rewritten as:

```
"(\\d{2}).(\\d{2}).(\\d{4})"
```

Worse yet, if we wanted to match for the backslash itself, we would have to escape it as well because of regex. With normal strings, we would have to quadruple-escape it! (`\\\\`)
We can save ourselves the headache of missing readability and confusion by using raw strings and write our regex normally. In fact, it is considered good style to use raw strings in *every* regex, even when it doesn't have any backslashes [33]. This is a help for your future self if you notice down the line that you actually would like to use a feature that requires a backslash.

We can iterate over the results of our regex [18]. The object we get on every match is a collection of our capture groups. Keep in mind that the zeroeth index is always the entire capture [19]. The first index is then the string from our first capture group, the second index is the string of the second capture group, and so on. [20]. Unfortunately, we do not get a compile-time check on our index, so if we accessed `&cap[4]`, our program would compile but then crash during runtime.

When replacing, we follow the same concept: `$0` is the entire match, `$1` the result of the first capture group, and so on. To make our life easier, we can give the capture groups names by starting them with `?P<somename>`[29] and then use this name when replacing [31].

There are many flags that you can specify, in the form of `(?flag)`, for fine-tuning, such as `i`, which makes the match case insensitive [33], or `x`, which ignores whitespace in the regex string. If you want to read up on them, visit their documentation (`https://doc.rust-lang.org/regex/regex/index.html`). Most of the time though, you can get the same result by using the `RegexBuilder` that is also in the regex crate [36]. Both of the `rust_regex` objects we generate in lines [33] and [36] are equivalent. While the second version is definitely more verbose, it is also way easier to understand at first glance.

There's more...

The regexes work by compiling their strings into the equivalent Rust code on creation. For performance reasons, you are advised to reuse your regexes instead of creating them anew every time you use them. A good way of doing this is by using the `lazy_static` crate, which we will look at later in the book, in the Creating lazy static objects section in `Chapter 5`, *Advanced Data Structures*.

Be careful not to overdo it with regexes. As they say, "When all you have is a hammer, everything looks like a nail." If you parse complicated data, regexes can quickly become an unbelievably complex mess. When you notice that your regex has become too big to understand at first glance, try to rewrite it as a parser.

See also

- *Creating lazy static objects* recipe in `Chapter 5`, *Advanced Data Structures*

Accessing the command line

Sooner or later, you'll want to interact with the user in some way or another. The most basic way to do this is by letting the user pass parameters while calling the application through the command line.

How to do it...

1. In the `bin` folder, create a file called `cli_params.rs`
2. Add the following code and run it with `cargo run --bin cli_params some_option some_other_option`:

```
use std::env;

fn main() {
  // env::args returns an iterator over the parameters
  println!("Got following parameters: ");
  for arg in env::args() {
    println!("- {}", arg);
  }

  // We can access specific parameters using the iterator API
  let mut args = env::args();
  if let Some(arg) = args.nth(0) {
    println!("The path to this program is: {}", arg);
  }
  if let Some(arg) = args.nth(1) {
      println!("The first parameter is: {}", arg);
  }
  if let Some(arg) = args.nth(2) {
      println!("The second parameter is: {}", arg);
  }

  // Or as a vector
  let args: Vec<_> = env::args().collect();
  println!("The path to this program is: {}", args[0]);
  if args.len() > 1 {
      println!("The first parameter is: {}", args[1]);
  }
  if args.len() > 2 {
      println!("The second parameter is: {}", args[2]);
  }
}
```

How it works...

Calling `env::args()` returns an iterator over the provided parameters[6]. By convention, the first command-line parameter on most operating systems is the path to the executable itself [12].

We can access specific parameters in two ways: keep them as an iterator [11] or `collect` them into a collection such as `Vec`[23]. Don't worry, we are going to talk about them in detail in `Chapter 2`, *Working with Collections*. For now, it's enough for you to know that:

- Accessing an iterator forces you to check at compile time whether the element exists, for example, an `if let` binding [12]
- Accessing a vector checks the validity at runtime

This means that we could have executed lines [26] and [29] without checking for their validity first in [25] and [28]. Try it yourself, add the `&args[3];` line at the end of the program and run it.

We check the length anyways because it is considered good style to check whether the expected parameters were provided. With the iterator way of accessing parameters, you don't have to worry about forgetting to check, as it forces you to do it. On the other hand, by using a vector, you can check for the parameters once at the beginning of the program and not worry about them afterward.

There's more...

If you are building a serious command-line utility in the style of *nix tools, you will have to parse a lot of different parameters. Instead of reinventing the wheel, you should take a look at third-party libraries, such as clap (`https://crates.io/crates/clap`).

Interacting with environment variables

According to the Twelve-Factor App (`https://12factor.net/`), you should store your configuration in the environment (`https://12factor.net/config`). This means that you should pass values that could change between deployments, such as ports, domains, or database handles, as environment variables. Many programs also use environment variables to communicate with each other.

How to do it...

1. In the `bin` folder, create a file called `env_vars.rs`
2. Add the following code and run it with `cargo run --bin env_vars`:

```
use std::env;

fn main() {
  // We can iterate over all the env vars for the current
  process
  println!("Listing all env vars:");
  for (key, val) in env::vars() {
    println!("{}: {}", key, val);
  }

  let key = "PORT";
  println!("Setting env var {}", key);
  // Setting an env var for the current process
  env::set_var(key, "8080");

  print_env_var(key);

  // Removing an env var for the current process
  println!("Removing env var {}", key);
  env::remove_var(key);

  print_env_var(key);
}

fn print_env_var(key: &str) {
  // Accessing an env var
  match env::var(key) {
    Ok(val) => println!("{}: {}", key, val),
    Err(e) => println!("Couldn't print env var {}: {}", key,
e),
29    }
30  }
```

How it works...

With `env::vars()`, we can access an iterator over all the `env var` that were set for the current process at the time of execution [6]. This list is pretty huge though, as you'll see when running the code, and for the most part, irrelevant for us.

It's more practical to access a single `env var` with `env::var()` [26], which returns an `Err` if the requested var is either not present or doesn't contain valid Unicode. We can see this in action in line [21], where we try to print a variable that we just deleted.

Because your `env::var` returns a `Result`, you can easily set up default values for them by using `unwrap_or_default`. One real-life example of this, involving the address of a running instance of the popular Redis (`https://redis.io/`) key-value storage, looks like this:

```
redis_addr = env::var("REDIS_ADDR")
    .unwrap_or_default("localhost:6379".to_string());
```

Keep in mind that creating an `env var` with `env::set_var()` [13] and deleting it with `env::remove_var()` [19] both only change the `env var` for our current process. This means that the created `env var` are not going to be readable by other programs. It also means that if we accidentally remove an important `env var`, the rest of the operating system is not going to care, as it can still access it.

There's more...

At the beginning of this recipe, I wrote about storing your configuration in the environment. The industry standard way to do this is by creating a file called `.env` that contains said config in the form of key-value-pairs, and loading it into the process at some point during the build. One easy way to do this in Rust is by using the dotenv (`https://crates.io/crates/dotenv`) third-party crate.

Reading from stdin

If you want to create an interactive application, it's easy to prototype your functionality with the command line. For CLI programs, this will be all the interaction you need.

How to do it...

1. In the `src/bin` folder, create a file called `stdin.rs`
2. Add the following code and run it with `cargo run --bin stdin`:

```
use std::io;
use std::io::prelude::*;

fn main() {
  print_single_line("Please enter your forename: ");
  let forename = read_line_iter();

  print_single_line("Please enter your surname: ");
  let surname = read_line_buffer();

  print_single_line("Please enter your age: ");
  let age = read_number();

  println!(
    "Hello, {} year old human named {} {}!",
    age, forename, surname
  );
}

fn print_single_line(text: &str) {
  // We can print lines without adding a newline
  print!("{}", text);
  // However, we need to flush stdout afterwards
  // in order to guarantee that the data actually displays
  io::stdout().flush().expect("Failed to flush stdout");
}

fn read_line_iter() -> String {
  let stdin = io::stdin();
  // Read one line of input iterator-style
  let input = stdin.lock().lines().next();
  input
    .expect("No lines in buffer")
34      .expect("Failed to read line")
35      .trim()
36      .to_string()
37  }
38
39  fn read_line_buffer() -> String {
40    // Read one line of input buffer-style
41    let mut input = String::new();
42    io::stdin()
43      .read_line(&mut input)
44      .expect("Failed to read line");
45    input.trim().to_string()
46  }
47
48  fn read_number() -> i32 {
49    let stdin = io::stdin();
50    loop {
51      // Iterate over all lines that will be inputted
52      for line in stdin.lock().lines() {
53        let input = line.expect("Failed to read line");
54        // Try to convert a string into a number
55        match input.trim().parse::<i32>() {
56          Ok(num) => return num,
57            Err(e) => println!("Failed to read number: {}", e),
58        }
59      }
60    }
61  }
```

How it works...

In order to read from the standard console input, `stdin`, we first need to obtain a handle to it. We do this by calling `io::stdin()` [29]. Imagine the returned object as a reference to a global `stdin` object. This global buffer is managed by a `Mutex`, which means that only one thread can access it at a time (more on that later in the book, in the Parallelly accessing resources with Mutexes section in `Chapter 7`, *Parallelism and Rayon*). We get this access by locking (using `lock()`) the buffer, which returns a new handle [31]. After we have done this, we can call the `lines` method on it, which returns an iterator over the lines the user will write [31 and 52]. More on iterators in the *Accessing collections as Iterators* section in `Chapter 2`, *Working with Collections*.

Finally, we can iterate over as many submitted lines as we want until some kind of break condition is reached, otherwise the iteration would go on forever. In our example, we break the number-checking loop as soon as a valid number has been entered [56].

If we're not particularly picky about our input and just want the next line, we have two options:

- We can continue using the infinite iterator provided by `lines()`, but simply call next on it in order to just take the first one. This comes with an additional error check as, generally speaking, we cannot guarantee that there is a next element.
- We can use `read_line` in order to populate an existing buffer [43]. This doesn't require that we `lock` the handler first, as it is done implicitly.

Although they both result in the same end effect, you should choose the first option. It is more idiomatic as it uses iterators instead of a mutable state, which makes it more maintainable and readable.

On a side note, we are using `print!` instead of `println!` in some places in this recipe for aesthetic reasons [22]. If you prefer the look of newlines before user input, you can refrain from using them.

There's more...

This recipe is written with the assumption that you want to use stdin for live interaction over the `cli`. If you plan on instead piping some data into it (for example, `cat foo.txt | stdin.rs` on *nix), you can stop treating the iterator returned by `lines()` as infinite and retrieve the individual lines, not unlike how you retrieved the individual parameters in the last recipe.

There are various calls to `trim()` in our recipe [35, 45 and 55]. This method removes leading and trailing whitespace in order to enhance the user-friendliness of our program. We are going to look at it in detail in the *Using a string* section in `Chapter 2`, *Working with Collections*.

See also

- *Interacting with environment variables* recipe in `Chapter 1`, *Learning the Basics*
- *Using a string* and *Accessing collections as iterators recipe* in `Chapter 2`, *Working with Collections*
- *Parallelly accessing resources with Mutexes* recipe in `Chapter 7`, *Parallelism and Rayon*

Accepting a variable number of arguments

Most of the time, when you want to operate on a dataset, you will design a function that takes a collection. In some cases, however, it is nice to have functions that just accept an unbound amount of parameters, like JavaScript's *rest parameters*. This concept is called **variadic functions** and is not supported by Rust. However, we can implement it ourselves by defining a recursive macro.

Getting started

The code in this recipe might be small, but it will look like gibberish if you're not familiar with macros. If you have not yet learned about macros or need a refresh, I recommend that you take a quick look at the relevant chapter in the official Rust book (`https://doc.rust-lang.org/stable/book/first-edition/macros.html`).

How to do it...

1. In the `src/bin` folder, create a file called `variadic.rs`
2. Add the following code and run it with `cargo run --bin variadic`:

```
macro_rules! multiply {
  // Edge case
  ( $last:expr ) => { $last };

  ( $head:expr, $($tail:expr), +) => {
    // Recursive call
    $head * multiply!($($tail),+)
  };
}

fn main() {
  // You can call multiply! with
  // as many parameters as you want
  let val = multiply!(2, 4, 8);
  println!("2*4*8 = {}", val)
}
```

How it works...

Let's start with our intention: we want to create a macro called multiply that accepts an undefined amount of parameters and multiplies them all together. In macros, this is done via recursion. We begin every recursive definition with the **edge case**, that is, the parameters where the recursion should stop. Most of the time, this is where a function call stops making sense. In our case, this is the single parameter. Think about it, what should `multiply!(3)` return? It doesn't make sense to multiply it with anything, since we have no other parameter to multiply it with. Our best reaction is to simply return the parameter unmodified.

Our other condition is a match against more than one parameter, a `$head` and a comma-separated list of parameters inside of a `$tail`. Here, we just define the return value as the `$head` multiplied with the multiplication of the `$tail`. This will call `multiply!` with the `$tail` and without the `$head`, which means that on every call we process one parameter less until we finally reach our edge case, one single parameter.

There's more...

Keep in mind that you should use this technique sparingly. Most of the time, it is clearer to just accept and operate on a slice instead. However, it makes sense to use this in combination with other macros and higher kinds of concepts where the analogy of *a graspable list of things* breaks down. Finding a good example for this is difficult since they tend to be extremely specific. You can find one of them at the end of the book though.

See also

- *Composing functions* recipe in `Chapter 10`, *Using Experimental Nightly Features*

2 Working with Collections

In this chapter, we will cover the following recipes:

- Using a vector
- Using a string
- Accessing collections as iterators
- Using a `VecDeque`
- Using a `HashMap`
- Using a `HashSet`
- Creating an own iterator
- Using a slab

Introduction

Rust provides a very broad set of collections to use. We will look at most of them, see how they're used, discuss how they're implemented, and when to use and choose them. A big part of this chapter focuses on iterators. Much of Rust's flexibility comes from them, as all collections (and more!) can be used as iterators. Learning how to use them is crucial.

Throughout this chapter, we are going to use the *big O notation* to show how effective certain algorithms are. In case you don't know it yet, it is a way of telling how much longer an algorithm takes when working with more elements. Let's look at it briefly.

$O(1)$ means that an algorithm is going to take the same time, no matter how much data is stored in a collection. It doesn't tell us how fast exactly it is, just that it's not going to slow down with size. This is the realistic ideal for a function. A practical example for this is accessing the first number in an infinite list of numbers: no matter how many numbers there are, you're always going to be able to instantly pick out the first one.

$O(n)$ means that an algorithm is going to slow down by the same degree for every element. This is not good, but still okay. An example for this is printing all data in a `for` loop.

$O(n^2)$ is really bad. It tells us that an algorithm is going to be slower and slower with every element. An example of it would be accessing data in a `for` loop nested in another `for` loop over the same data.

Using a vector

The most basic collection is the vector, or `Vec` for short. It is essentially a variable-length array with a very low overhead. As such, it is the collection that you will use most of the time.

How to do it...

1. In the command line, jump one folder up with `cd ..` so you're not in `chapter-one` anymore. In the next chapters, we are going to assume that you always started with this step.
2. Create a Rust project to work on during this chapter with `cargo new chapter-two`.
3. Navigate into the newly-created `chapter-two` folder. For the rest of this chapter, we will assume that your command line is currently in this directory.
4. Inside the folder `src`, create a new folder called `bin`.
5. Delete the generated `lib.rs` file, as we are not creating a library.
6. In the folder `src/bin`, create a file called `vector.rs`.
7. Add the following code blocks to the file and run them with `cargo run --bin vector`:

```
fn main() {
  // Create a vector with some elements
  let fruits = vec!["apple", "tomato", "pear"];
```

```
// A vector cannot be directly printed
// But we can debug-print it
println!("fruits: {:?}", fruits);

// Create an empty vector and fill it
let mut fruits = Vec::new();
 fruits.push("apple");
fruits.push("tomato");
 fruits.push("pear");
 println!("fruits: {:?}", fruits);

 // Remove the last element
 let last = fruits.pop();
 if let Some(last) = last {
     println!("Removed {} from {:?}", last, fruits);
 }

 // Insert an element into the middle of the vector
 fruits.insert(1, "grape");
 println!("fruits after insertion: {:?}", fruits);

 // Swap two elements
 fruits.swap(0, 1);
 println!("fruits after swap: {:?}", fruits);
```

8. This is how you can access single elements in a vector:

```
 // Access the first and last elements
 let first = fruits.first();
 if let Some(first) = first {
   println!("First fruit: {}", first);
 }
 let last = fruits.last();
 if let Some(last) = last {
   println!("Last fruit: {}", last);
 }

 // Access arbitrary elements
 let second = fruits.get(1);
 if let Some(second) = second {
   println!("Second fruit: {}", second);
 }
 // Access arbitrary elements without bonds checking
 let second = fruits[1];
 println!("Second fruit: {}", second);
```

9. The next few methods apply to the whole vector:

```
// Initialize the vector with a value
// Here, we fill our vector with five zeroes
let bunch_of_zeroes = vec![0; 5];
println!("bunch_of_zeroes: {:?}", bunch_of_zeroes);

// Remove some item and shift all that come after
// into place
let mut nums = vec![1, 2, 3, 4];
let second_num = nums.remove(1);
println!("Removed {} from {:?}", second_num, nums);

// Filter the vector in place
let mut names = vec!["Aaron", "Felicia", "Alex", "Daniel"];
// Only keep names starting with 'A'
names.retain(|name| name.starts_with('A'));
println!("Names starting with A: {:?}", names);

// Check if the vector contains an element
println!("Does 'names' contain \"Alex\"? {}",
  names.contains(&"Alex"));



// Remove consecutive(!) duplicates
let mut nums = vec![1, 2, 2, 3, 4, 4, 4, 5];
nums.dedup();
println!("Deduped, pre-sorted nums: {:?}", nums);

// Be careful if your data is not sorted!
let mut nums = vec![2, 1, 4, 2, 3, 5, 1, 2];
nums.dedup();
// Doens't print what you might expect
println!("Deduped, unsorted nums: {:?}", nums);

// Sort a vector
nums.sort();
println!("Manually sorted nums: {:?}", nums);
nums.dedup();
println!("Deduped, sorted nums: {:?}", nums);

// Reverse a vector
nums.reverse();
println!("nums after being reversed: {:?}", nums);

// Create a consuming iterator over a range
let mut alphabet = vec!['a', 'b', 'c'];
print!("The first two letters of the alphabet are: ");
for letter in alphabet.drain(..2) {
  print!("{} ", letter);
}
println!();
// The drained elements are no longer in the vector
println!("alphabet after being drained: {:?}", alphabet);


// Check if a vector is empty
let mut fridge = vec!["Beer", "Leftovers", "Mayonaise"];
println!("Is the fridge empty {}", fridge.is_empty());
// Remove all elements
fridge.clear();
println!("Is the fridge now empty? {}", fridge.is_empty());
```

10. We can split a vector into two and combine them again:

```
// Split a vector into two pieces
let mut colors = vec!["red", "green", "blue", "yellow"];
println!("colors before splitting: {:?}", colors);
let half = colors.len() / 2;
let mut second_half = colors.split_off(half);
println!("colors after splitting: {:?}", colors);
println!("second_half: {:?}", second_half);

// Put two vectors together
colors.append(&mut second_half);
println!("colors after appending: {:?}", colors);
// This empties the second vector
println!("second_half after appending: {:?}", second_half);
```

11. You might remember the `splice` method from JavaScript:

```
let mut stuff = vec!["1", "2", "3", "4", "5", "6"];
println!("Original stuff: {:?}", stuff);
let stuff_to_insert = vec!["a", "b", "c"];
let removed_stuff: Vec<_> = stuff.splice(1..4,
  stuff_to_insert).collect();
println!("Spliced stuff: {:?}", stuff);
println!("Removed stuff: {:?}", removed_stuff);
```

12. If you are working with very big datasets, you can optimize the performance of your vector:

```
136   // Initialize the vector with a certain capacity
137   let mut large_vec: Vec<i32> = Vec::with_capacity(1_000_000);
138   println!("large_vec after creation:");
139   println!("len:\t\t{}", large_vec.len());
140   println!("capacity:\t{}", large_vec.capacity());
141
142   // Shrink the vector as close as possible to its length
143   large_vec.shrink_to_fit();
144   println!("large_vec after shrinking:");
145   println!("len:\t\t{}", large_vec.len());
146   println!("capacity:\t{}", large_vec.capacity());
147
148   // Remove some item, replacing it with the last
149   let mut nums = vec![1, 2, 3, 4];
150   let second_num = nums.swap_remove(1);
151   // This changes the order, but works in O(1)
152   println!("Removed {} from {:?}", second_num, nums);
153 }
```

How it works...

This recipe is going to be a bit longer than the others, because:

- The vector is the most important collection
- Many of its core principles, like preallocation, apply to other collections as well
- It includes methods used on slices, which are also usable by many other collections

Let's start at the beginning.

A vector can be created [9] by using the constructor pattern we mentioned earlier (`Chapter 1`, *Learning the Basics, Using the Constructor Pattern*), and filled by calling `push` on it for every element we want to store [10]. Because this is such a common pattern, Rust provides you with a convenient macro called `vec!`[3]. While its end effect is the same, the macro is implemented with some nice performance optimizations.

Because of the convenience `vec!` provides, other Rustacians have implemented similar macros for the other collections, which you can find here: `https://crates.io/crates/maplit`.

If you want to initialize a vector by repeating an element over and over, you can use the special calling syntax described in line [52] to do so.

The opposite of `push` is `pop`: it removes the last element of the vector, and returns it if the vector wasn't empty before. Because of the memory layout of `Vec`, which we are going to look at in the next section, this operation is done in $O(1)$ complexity. If you don't know what that means, let me rephrase that: it's pretty fast. This is why vectors can be nicely used as **First In, Last Out (FILO)** stacks.

If you need to modify the contents of a vector, `insert`[22], `remove` [58], and `swap` [26] should be self-explanatory. Sometimes, though, you want to access a specific element in the vector. You can use `get` to borrow an element at an index [40], and `get_mut` to mutate it. Both return an `Option` that only contains `Some` element if the index was valid. Most times, though, this fine grade of error checking is unnecessary for vector access, as an out-of-bounds index is usually not recoverable, and will just be handled by unwrapping the `Option`. For this reason, Rust lets you call the `Index` operator, `[]`, on a `Vec` [45]. This will automatically deduce its mutability and perform an unwrap for you.

There are a bunch of methods that help us work with the entire vector at once. `retain` is a very useful one that is also implemented by most other collections [64]. It accepts a so-called **predicate**, which is a fancy word for a function that returns `true` or `false`. It applies that predicate to every element, and only keeps the ones where it returned `true`.

`dedup` removes all *consecutive* duplicates [74]. This means that for the vector `[1, 2, 2, 3, 2, 3]`, `dedup` would result in `[1, 2, 3, 2, 3]`, as only the duplicate 2s were consecutive. Always remember this when using it, as it can cause bugs that are hard to find. If you want to remove *all* duplicates, you need to make them consecutive by sorting the vector first. If your elements are comparable, this is as simple as calling `.sort()` [84].

Using `drain` creates a consuming iterator over your vector, accessing all elements and removing them in the process, leaving your vector empty [96]. This is useful when you have to *work through* your data and reuse your empty vector again afterwards to collect more work.

If you've never seen `splice` in another language, you're probably going to feel a bit confused at first about what it does. Let's take a look at it, shall we?

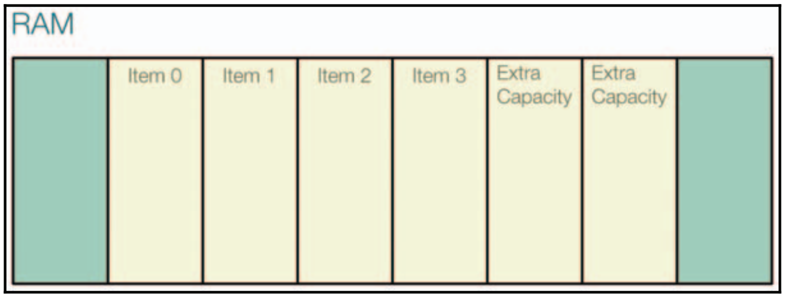
`splice` does three things:

- It takes a range. This range will be *removed* from the vector.
- It takes an iterator. This iterator will be *inserted* into the space left open by the removal from the last step.
- It *returns* the removed elements as an iterator.

How to handle the returned iterator is going to be the topic of the recipe in the *Access collections as iterators* section.

There's more...

The vector should always be your go-to collection. Internally, it is implemented as a continuous chunk of memory stored on the heap:



The important keyword here is *continuous*, which means that the memory is very cache-friendly. In other words, the vector is pretty fast! The vector even allocates a bit of extra memory in case you want to extend it. Be careful, though, when inserting a lot of data at the beginning of the vector: the entire stack will have to be moved.

At the end, you can see a bit of *extra capacity*. This is because `Vec` and many other collections preallocate a bit of extra memory each time you have to move the block, because it has grown too large. This is done in order to prevent as many reallocations as possible. You can check the exact amount of total space of a vector by calling `capacity`[140] on it. You can influence the preallocation by initializing your vector with `with_capacity`[137]. Use it when you have a rough idea about how many elements you plan on storing. This can be a big difference in capacity when working with big amounts of data.

The extra capacity doesn't go away when shortening the vector. If you had a vector with a length of 10,000 and a capacity of 100,000 and called `clear` on it, you would still have a capacity of 100,000 preallocated. When working on systems with memory limitations, like microcontrollers, this can become a problem. The solution is calling `shrink_to_fit` periodically on such vectors [143]. This will bring the capacity as close as possible to the length, but it is allowed to still leave a little bit of preallocated space ready.

Another way to optimize really big vectors is to call `swap_remove` [150]. Normally, when removing an element from a vector, all elements after it will be shifted to the left in order to preserve continuous memory. This is a lot of work when removing the first element in a big vector. If you don't care about the exact order of your vector, you can call `swap_remove` instead of `remove`. It works by swapping the element that is to be removed with the last element, and adjusting the length. This is great, because you don't create a *hole* that needs to be filled by shifting, and because swapping memory is a really fast operation in today's processors.

Using a string

Rust provides an unusually large functionality for its string. Knowing it can save you quite some headache when dealing with raw user input.

How to do it...

1. In the folder `src/bin`, create a file called `string.rs`.
2. Add the following code, and run it with `cargo run --bin string`:

```
fn main() {
  // As a String is a kind of vector,
  // you can construct them the same way
  let mut s = String::new();
  s.push('H');
  s.push('i');
  println!("s: {}", s);

  // The String however can also be constructed
  // from a string slice (&str)
  // The next two ways of doing to are equivalent
  let s = "Hello".to_string();
  println!("s: {}", s);
  let s = String::from("Hello");
```

```
println!("s: {}", s);

// A String in Rust will always be valid UTF-8
let s = "汉语 한글 Þjóðhildur 😉 🍺".to_string();
println!("s: {}", s);

// Append strings to each other
let mut s = "Hello ".to_string();
s.push_str("World");

// Iterate over the character
// A "character" is defined here as a
// Unicode Scalar Value
for ch in "Tubular".chars() {
    print!("{}.", ch);
}
println!();
// Be careful though, a "character" might not
// always be what you expect
for ch in "y̆".chars() {
    // This does NOT print y̆
    print!("{} ", ch);
}
println!();
```

Use the following code to split a string in various ways:

```
// Split a string slice into two halves
let (first, second) = "HelloThere".split_at(5);
println!("first: {}, second: {}", first, second);

// Split on individual lines
let haiku = "\
             she watches\n\
             satisfied after love\n\
             he lies\n\
             looking up at nothing\n\
             ";
for line in haiku.lines() {
    println!("\t{}.", line);
}

// Split on substrings
for s in "Never;Give;Up".split(';') {
    println!("{}", s);
}
```

```
// When the splitted string is at the beginning or end,
// it will result in the empty string
let s: Vec<_> = "::Hi::There::".split("::").collect();
println!("{:?}", s);

// If you can eliminate the empty strings at the end
// by using split_termitor
let s: Vec<_> = "Mr. T.".split_terminator('.').collect();
println!("{:?}", s);

// char has a few method's that you can use to split on
for s in "I'm2fast4you".split(char::is_numeric) {
    println!("{}", s);
}

// Split only a certain amount of times
for s in "It's not your fault, it's mine".splitn(3,
  char::is_whitespace) {
    println!("{}", s);
}

// Get only the substrings that match a pattern
// This is the opposite of splitting
for c in "The Dark Knight rises".matches(char::is_uppercase)
{
    println!("{}", c);
}

// Check if a string starts with something
let saying = "The early bird gets the worm";
let starts_with_the = saying.starts_with("The");
println!(
  "Does \"{}\" start with \"The\"?: {}",
   saying,
   starts_with_the
);
let starts_with_bird = saying.starts_with("bird");
println!(
  "Does \"{}\" start with \"bird\"?: {}",
   saying,
   starts_with_bird
);

// Check if a string ends with something
let ends_with_worm = saying.ends_with("worm");
println!("Does \"{}\" end with \"worm\"?: {}", saying,
  ends_with_worm);
```

```

    // Check if the string contains something somewhere
    let contains_bird = saying.contains("bird");
    println!("Does \"{}\" contain \"bird\"?: {}", saying,
      contains_bird);
```

Remove whitespace:

```
    // Splitting on whitespace might not result in what you expect
    let a_lot_of_whitespace = "    I   love spaaace     ";
    let s: Vec<_> = a_lot_of_whitespace.split(' ').collect();
    println!("{:?}", s);
    // Use split_whitespace instead
    let s: Vec<_> =
      a_lot_of_whitespace.split_whitespace().collect();
    println!("{:?}", s);

    // Remove leading and trailing whitespace
    let username = "   P3ngu1n\n".trim();
    println!("{}", username);
    // Remove only leading whitespace
    let username = "   P3ngu1n\n".trim_left();
    println!("{}", username);
    // Remove only trailing whitespace
    let username = "   P3ngu1n\n".trim_right();
    println!("{}", username);


    // Parse a string into another data type
    // This requires type annotation
    let num = "12".parse::<i32>();
    if let Ok(num) = num {
      println!("{} * {} = {}", num, num, num * num);
    }
```

Modify the string:

```
    // Replace all occurences of a pattern
    let s = "My dad is the best dad";
    let new_s = s.replace("dad", "mom");
    println!("new_s: {}", new_s);

    // Replace all characters with their lowercase
    let lowercase = s.to_lowercase();
    println!("lowercase: {}", lowercase);

    // Replace all characters with their uppercase
```

```
143   let uppercase = s.to_uppercase();
144   println!("uppercase: {}", uppercase);
145
146   // These also work with other languages
147   let greek = "ὈΔΥΣΣΕΎΣ";
148   println!("lowercase greek: {}", greek.to_lowercase());
149
150   // Repeat a string
151   let hello = "Hello! ";
152   println!("Three times hello: {}", hello.repeat(3));
153 }
```

How it works...

Essentially, being a kind of vector, a string can be created the same way by combining `new` and `push`; however, because this is really inconvenient, a `string`, which is an owned chunk of memory, can be created from a string slice (`&str`), which is either a borrowed string or a literal. Both of the ways to do it, that are shown in this recipe, are equivalent:

```
let s = "Hello".to_string();
println!("s: {}", s);
let s = String::from("Hello");
println!("s: {}", s);
```

Out of pure personal preference, we will use the first variant.

Before Rust `1.9`, `to_owned()` was the fastest way to create a string. Now, `to_string()` is equally performant and should be preferred, because it offers more clarity over what is done. We mention this because many old tutorials and guides have not been updated since then, and still use `to_owned()`.

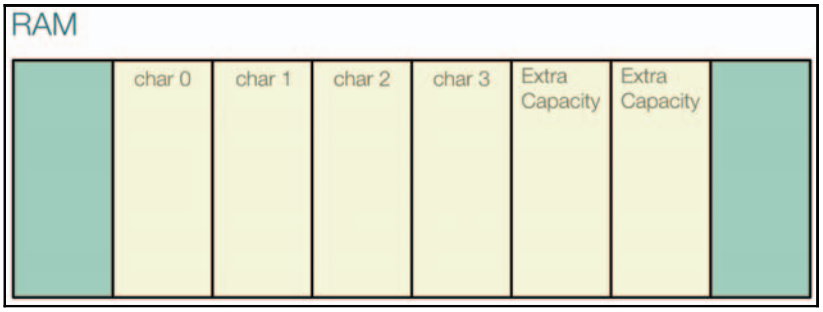
All strings in Rust are valid Unicode in UTF-8 encoding. This can lead to some surprises, as a *character*, as we know it, is an inherently Latin invention. For instance, look at languages that have a modifier for a letter—is ä an own character, or is it merely a variation of `a`? What about languages that allow many combinations in extreme? What would that keyboard even look like? For this reason, Unicode lets you compose your *characters* from different *Unicode scalar values*. With `.chars()`, you can create an iterator that goes through these scalars [28]. If you work with non-Latin characters, you might get surprised by this when accessing composing characters —y̌ is not one, but two scalars, `y` and ˇ [36]. You can get around this by using the `Unicode-segmentation` crate, which supports iteration over graphemes: `https://crates.io/crates/unicode-segmentation`.

When splitting a string on a pattern that is at the beginning, is at the end, or occurs multiple times after each other, each instance gets split into an empty string `""`[107]. This is especially nasty when splitting on spaces (`' '`). In this case, you should use `split_whitespace` instead [110]. Otherwise, `split_terminator` will remove the empty strings from the end of the string [68].

By the way, when we talk about a *pattern* in this recipe, we mean one of three things:
- A character
- A string
- A predicate that takes one `char`

There's more...

The implementation of `String` should not be much of a surprise—it's just a kind of vector:



Accessing collections as iterators

Welcome to one of the most flexible parts of the Rust standard library. Iterators are, as the name suggests, a way of applying actions of items in a collection. If you come from C#, you will already be familiar with iterators because of Linq. Rust's iterators are kind of similar, but come with a more functional approach to things.

Because they are an extremely fundamental part of the standard library, we are going to dedicate this recipe entirely to a showcase of all the different things you can do with them in isolation. For real-world use cases, you can simply continue reading the book, as a big portion of the other recipes features iterators in some way or another.

How to do it...

1. In the folder `src/bin`, create a file called `iterator.rs`.
2. Add the following code, and run it with `cargo run --bin iterator`:

```
fn main() {
  let names = vec!["Joe", "Miranda", "Alice"];
  // Iterators can be accessed in many ways.
  // Nearly all collections implement .iter() for this purpose
  let mut iter = names.iter();
  // A string itself is not iterable, but its characters are
  let mut alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".chars();
  // Ranges are also (limited) iterators
  let nums = 0..10;
  // You can even create infinite iterators!
  let all_nums = 0..;

  // As the name says, you can iterate over iterators
  // This will consume the iterator
  for num in nums {
      print!("{} ", num);
  }
  // nums is no longer usable
  println!();

  // Get the index of the current item
  for (index, letter) in "abc".chars().enumerate() {
    println!("#{}. letter in the alphabet: {}", index + 1,
      letter);
  }
```

3. Access individual items:

```
  // going through an iterator, step by step
  if let Some(name) = iter.next() {
      println!("First name: {}", name);
  }
  if let Some(name) = iter.next() {
      println!("Second name: {}", name);
  }
  if let Some(name) = iter.next() {
      println!("Third name: {}", name);
  }
  if iter.next().is_none() {
      println!("No names left");
  }
```

```

    // Arbitrary access to an item in the iterator
    let letter = alphabet.nth(3);
    if let Some(letter) = letter {
      println!("the fourth letter in the alphabet is: {}",
        letter);
    }
    // This works by consuming all items up to a point
    let current_first = alphabet.nth(0);
    if let Some(current_first) = current_first {
      // This will NOT print 'A'
      println!(
        "The first item in the iterator is currently: {}",
           current_first
      );
    }
    let current_first = alphabet.nth(0);
    if let Some(current_first) = current_first {
      println!(
        "The first item in the iterator is currently: {}",
           current_first
      );
    }

    // Accessing the last item; This will
    // consume the entire iterator
    let last_letter = alphabet.last();
    if let Some(last_letter) = last_letter {
      println!("The last letter of the alphabet is: {}",
        last_letter);
    }
```

4. Collect the iterator into a collection:

```
    // Collect iterators into collections
    // This requires an anotation of which collection we want
    // The following two are equivalent:
    let nums: Vec<_> = (1..10).collect();
    println!("nums: {:?}", nums);
    let nums = (1..10).collect::<Vec<_>>();
    println!("nums: {:?}", nums)
```

5. Change which items are being iterated over:

```
// Taking only the first n items
// This is often used to make an infinite iterator finite
let nums: Vec<_> = all_nums.take(5).collect();
println!("The first five numbers are: {:?}", nums);

// Skip the first few items
let nums: Vec<_> = (0..11).skip(2).collect();
println!("The last 8 letters in a range from zero to 10:
  {:?}", nums);

// take and skip accept predicates in the form of
// take_while and skip_while
let nums: Vec<_> = (0..).take_while(|x| x * x <
50).collect();
println!(
  "All positive numbers that are less than 50 when squared:
    {:?}", nums
);

// This is useful to filter an already sorted vector
let names = ["Alfred", "Andy", "Jose", "Luke"];
let names: Vec<_> = names.iter().skip_while(|x|
  x.starts_with('A')).collect();
println!("Names that don't start with 'A': {:?}", names);

// Filtering iterators
let countries = [
  "U.S.A.",
  "Germany",
  "France",
  "Italy",
  "India",
  "Pakistan",
  "Burma",
];
let countries_with_i: Vec<_> = countries
  .iter()
  .filter(|country| country.contains('i'))
  .collect();
println!(
  "Countries containing the letter 'i': {:?}",
  countries_with_i
);
```

6. Check if an iterator contains an element:

```
// Find the first element that satisfies a condition
if let Some(country) = countries.iter().find(|country|
  country.starts_with('I')) {
    println!("First country starting with the letter 'I':
      {}", country);
  }

// Don't get the searched item but rather its index
if let Some(pos) = countries
  .iter()
  .position(|country| country.starts_with('I'))
  {
    println!("It's index is: {}", pos);
  }

// Check if at least one item satisfies a condition
let are_any = countries.iter().any(|country| country.len() ==
  5);
println!(
  "Is there at least one country that has exactly five
   letters? {}",
   are_any
);

// Check if ALL items satisfy a condition
let are_all = countries.iter().all(|country| country.len() ==
  5);
println!("Do all countries have exactly five letters? {}",
  are_all);
```

7. Useful operations for numeric items:

```
let sum: i32 = (1..11).sum();
let product: i32 = (1..11).product();
println!(
  "When operating on the first ten positive numbers\n\
   their sum is {} and\n\
   their product is {}.",
   sum, product
);

let max = (1..11).max();
let min = (1..11).min();
if let Some(max) = max {
  println!("They have a highest number, and it is {}", max);
}
```

```
if let Some(min) = min {
  println!("They have a smallest number, and it is {}", min);
}
```

8. Combine iterators:

```
// Combine an iterator with itself, making it infinite
// When it reaches its end, it starts again
let some_numbers: Vec<_> = (1..4).cycle().take(10).collect();
// Reader exercise: Try to guess what this will print
println!("some_numbers: {:?}", some_numbers);

// Combine two iterators by putting them after another
let some_numbers: Vec<_> = (1..4).chain(10..14).collect();
println!("some_numbers: {:?}", some_numbers);

// Zip two iterators together by grouping their first items
// together, their second items together, etc.
let swiss_post_codes = [8957, 5000, 5034];
let swiss_towns = ["Spreitenbach", "Aarau", "Suhr"];
let zipped: Vec<_> =
  swiss_post_codes.iter().zip(swiss_towns.iter()).collect();
println!("zipped: {:?}", zipped);

// Because zip is lazy, you can use two infine ranges
let zipped: Vec<_> = (b'A'..)
  .zip(1..)
  .take(10)
  .map(|(ch, num)| (ch as char, num))
  .collect();
println!("zipped: {:?}", zipped);
```

9. Apply functions to all items:

```
// Change the items' types
let numbers_as_strings: Vec<_> = (1..11).map(|x|
  x.to_string()).collect();
println!("numbers_as_strings: {:?}", numbers_as_strings);

// Access all items
println!("First ten squares:");
(1..11).for_each(|x| print!("{} ", x));
println!();

// filter and map items at the same time!
let squares: Vec<_> = (1..50)
  .filter_map(|x| if x % 3 == 0 { Some(x * x) } else { None
})
```

```
200     .collect();
201   println!(
202     "Squares of all numbers under 50 that are divisible by 3:
203      {:?}", squares
204   );
```

10. The real strength of iterators comes from combining them:

```
208   // Retrieve the entire alphabet in lower and uppercase:
209   let alphabet: Vec<_> = (b'A' .. b'z' + 1) // Start as u8
210     .map(|c| c as char) // Convert all to chars
211     .filter(|c| c.is_alphabetic()) // Filter only alphabetic
chars
212     .collect(); // Collect as Vec<char>
213   println!("alphabet: {:?}", alphabet);
214 }
```

How it works...

This recipe is incredibly important. No matter what you do, or which library you use, it's going to use iterators somewhere. All of the operations presented can be used on any collection and all types that implement the `iterator` trait.

In the first section, we looked at different ways to create iterators. I mention that ranges are *limited* because, in order to be iterable, the range-type has to implement `Step`. `char` doesn't, so you wouldn't be able to use `'A'..'D'` as an iterator. For this reason, in line [209], we iterate over the characters as bytes:

```
let alphabet: Vec<_> = (b'A' .. b'z' + 1) // Start as u8
    .map(|c| c as char)            // Convert all to chars
    .filter(|c| c.is_alphabetic()) // Filter only alphabetic chars
    .collect(); // Collect as Vec<char>
```

We have to set the limit of the range to `b'z' + 1`, because ranges are non-inclusive. You might have noticed that this fact makes using ranges confusing sometimes. This is why, on the nightly compiler, you can use inclusive ranges (`Chapter 10`, *Using Experimental Nightly Features*, *Iterating over an inclusive range*).

Let's return to our recipe here, though. While iterating, you have the option to use `enumerate` in order to get an iteration counter [22]. This is the reason that Rust gets away with not supporting traditional, C-style `for` loop syntax. You most probably have seen some variation of the following C code in some language or another:

```
for (int i = 0; i < some_length; i++) {
    ...
}
```

Rust disallows this, because a range-based `for` loop is almost always cleaner to use, which you are going to know if you come from a Python background, as it pioneered this restriction. In fact, most programming languages have shifted their paradigm to promote range-based loops. In the rare cases where you really actually would like to know your iteration count, you can use `enumerate` to emulate this behavior.

When accessing single items with `nth`[41], you have to keep two things in mind:

- It accesses an item by going through all items until it reaches the one you want. In the worst case, this is an $O(n)$access. If you can, use your collection's native access method (most of the time, this would be `.get()`).
- It consumes the iterator up to the specified index. This means that calling `nth` twice with the exact same parameters is going to result in two different returned values [54]. Don't let this catch you by surprise.

Another thing to take note of when using an iterator's various accessors is that they all return an `Option` that will be `None` if the iterator has no more items left.

When collecting an iterator into a collection, the following two forms of annotation are completely equivalent:

```
let nums: Vec<_> = (1..10).collect();
let nums = (1..10).collect::<Vec<_>>();
```

Just use the one that you like most. In this book, we stick to the first form because of personal preference. The second form, by the way, is called the *turbofish*. This is because `::<>` kinda looks like a certain family of fish. Cute, isn't it? Both forms are also able to deduce their exact type automatically, so you don't need to write `Vec<i32>`. The inner type can be omitted with an underscore (_), as shown.

`cycle` [163] takes an iterator and repeats it endlessly. `[1, 2, 3]` would become `[1, 2, 3, 1, 2, 3, 1, ...]`.

`zip` [175] takes two iterators and creates one out of them by putting items that are at the same index into a tuple, and then chaining them. If the iterators have different sizes, it just ignores the extra items of the longer one. For example, `[1, 2, 3]` zipped with `['a', 'b', 'c', 'd']` would become `[(1, 'a'), (2, 'b'), (3, 'c')]`, because `'d'` will be thrown away as it has no partner to zip with. If you zip two infinite ranges, you will have no problem, as `zip` is lazy, which means it will only actually start zipping your iterators when it's really necessary; for example, when using `take` to extract the first few tuples [81].

If you need to mutate all of your items, you can use `map`. It can also be used to change the underlying type of the iterator, as shown in line [182]. `for_each` is really similar, with one big difference: it doesn't return anything. It's basically the same as manually using a `for` loop on an iterator. The intended use case for it is situations in which you have a lot of chained method calls on an iterator, where it can be more elegant to chain `for_each` as well, as a kind of *consumer*.

Iterators are very often chained together to weave complex transformations together. If you find yourself calling many methods on a single iterator, don't worry, as this is exactly what you should be doing. On the other hand, when you see yourself doing a lot of complicated stuff in a `for` loop, you should probably rewrite that code with iterators.

When using `filter_map`[199], you can keep an item by returning it, wrapped in `Some`. If you want to filter it out, return `None`. Before that, you're allowed to change the item in whatever way you want, which is the `map` part of the deal.

There's more...

`iter()` creates an iterator that *borrows* items. If you want to create an iterator that *consumes* items—for example, takes ownership of them by moving them – you can use `into_iter()`.

See also

- *The Iterating over an inclusive range* recipe in `Chapter 10`, *Using Experimental Nightly Features*

Using a VecDeque

When you need to insert or remove elements regularly into or from the beginning of the vector, your performance might take quite a hit, as it will force the vector to reallocate all data that comes after it. This is especially bothersome when implementing a queue. For this reason, Rust provides you with the `VecDeque`.

How to do it...

1. In the folder `src/bin`, create a file called `vecdeque.rs`.
2. Add the following code, and run it with `cargo run --bin vecdeque`:

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

fn main() {
  // A VecDeque is best thought of as a
  // First-In-First-Out (FIFO) queue

  // Usually, you will use it to push_back data
  // and then remove it again with pop_front
  let mut orders = VecDeque::new();
  println!("A guest ordered oysters!");
  orders.push_back("oysters");

  println!("A guest ordered fish and chips!");
  orders.push_back("fish and chips");

  let prepared = orders.pop_front();
  if let Some(prepared) = prepared {
    println!("{} are ready", prepared);
  }

  println!("A guest ordered mozarella sticks!");
  orders.push_back("mozarella sticks");

  let prepared = orders.pop_front();
  if let Some(prepared) = prepared {
    println!("{} are ready", prepared);
  }

  println!("A guest ordered onion rings!");
  orders.push_back("onion rings");

  let prepared = orders.pop_front();
```

```
    if let Some(prepared) = prepared {
      println!("{} are ready", prepared);
    }

    let prepared = orders.pop_front();
    if let Some(prepared) = prepared {
      println!("{} are ready", prepared);
    }

    // You can freely switch your pushing
    // from front to back and vice versa
    let mut sentence = VecDeque::new();
    sentence.push_back("a");
    sentence.push_front("had");
    sentence.push_back("little");
    sentence.push_front("Mary");
    sentence.push_back("Lamb");
    println!("sentence: {:?}", sentence);

    // The same applies to popping data
    sentence.pop_front();
    sentence.push_front("Jimmy");
    sentence.pop_back();
    sentence.push_back("Cat");
    println!("sentence: {:?}", sentence);


    // The rest of the VecDeque's methods are
    // pretty much the same as the vector's
    // However, the VecDeque has additional options
    // when swap removing!
    let mut some_queue = VecDeque::with_capacity(5);
    some_queue.push_back("A");
    some_queue.push_back("B");
    some_queue.push_back("C");
    some_queue.push_back("D");
    some_queue.push_back("E");
    println!("some_queue: {:?}", some_queue);

    // This is the same as Vec's swap_remove
    some_queue.swap_remove_back(2);
    println!("some_quere after swap_remove_back: {:?}",
      some_queue);

    // This is the nearly the same, but swaps the removed
    // element with the first one instead of the last one
    some_queue.swap_remove_front(2);
    println!("some_quere after swap_remove_front: {:?}",
              some_queue);
80    }
```

How it works...

Most of the interface of `VecDeque` is identical to `Vec`. You can even optimize them the same way with `with_capacity` and its `swap_remove` equivalents. The differences come from the fact that `VecDeque` is more oriented around access from both ends. As such, multiple methods from `Vec` that implicitly affect the last element have two equivalents in `VecDeque`: one for the front, and one for the back. These are:

- `push`, which becomes `push_front` [46] and `push_back` [11]
- `pop`, which becomes `pop_front` [16] and `pop_back` [55]
- `swap_remove`, which becomes `remove_front` [78] and `remove_back` [73]

A `VecDeque` has the ability to freely append or remove elements from both ends in a performant way, which makes it an ideal candidate for a **First In, First Out** (**FIFO**) queue [24]. In fact, this is how it's nearly always used.

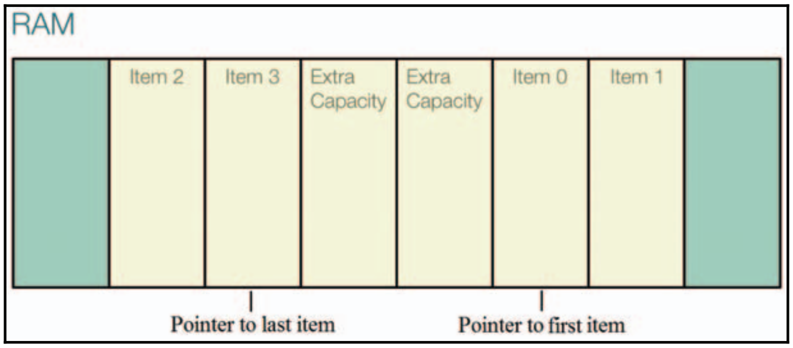
When you see yourself in a situation where you want to respond to any kind of requests in the order they arrive in and remove them again afterwards, a `VecDeque` is an ideal tool for the job.

There's more...

Internally, the `VecDeque` is implemented as a *ring buffer*, also known as a *circular buffer*. It's called like this because it behaves like a circle: the end touches the beginning.

It works by allocating a continuous block of memory, like the `Vec`; however, where the `Vec` always leaves its extra capacity at the end of the block, the `VecDeque` has nothing against leaving spots inside the block empty. It follows that when you remove the first element, the `VecDeque` doesn't move all elements to the left, but simply leaves the first spot empty. If you then push an element into the beginning via `push_front`, it will take the spot freed earlier while leaving the elements after it untouched.

The circular catch in the story is that if you have some capacity in the front of the block but none in the back while using `push_back`, the `VecDeque` will simply use that space to allocate the extra element, leading to the following situation:



This is great, because you will not have to worry about this at all while using it, as its iterating methods hide the implementation by always showing you the *correct* order!

Like the vector, `VecDeque` will resize itself and move all its elements into a new block when its capacity runs out.

Using a HashMap

If you imagine a `Vec` as a collection that assigns an index (0, 1, 2, and so on) to data, the `HashMap` is a collection that assigns any data to any data. It allows you to map arbitrary, hashable data to other arbitrary data. Hashing and mapping, that's where the name comes from!

How to do it...

1. In the folder `src/bin`, create a file called `hashmap.rs`.
2. Add the following code, and run it with `cargo run --bin hashmap`:

```
use std::collections::HashMap;

fn main() {
  // The HashMap can map any hashable type to any other
  // The first type is called the "key"
  // and the second one the "value"
```

```
let mut tv_ratings = HashMap::new();
// Here, we are mapping &str to i32
tv_ratings.insert("The IT Crowd", 8);
tv_ratings.insert("13 Reasons Why", 7);
tv_ratings.insert("House of Cards", 9);
tv_ratings.insert("Stranger Things", 8);
tv_ratings.insert("Breaking Bad", 10);

// Does a key exist?
let contains_tv_show = tv_ratings.contains_key("House of
  Cards");
println!("Did we rate House of Cards? {}", contains_tv_show);
let contains_tv_show = tv_ratings.contains_key("House");
println!("Did we rate House? {}", contains_tv_show);

// Access a value
if let Some(rating) = tv_ratings.get("Breaking Bad") {
  println!("I rate Breaking Bad {} out of 10", rating);
}

// If we insert a value twice, we overwrite it
let old_rating = tv_ratings.insert("13 Reasons Why", 9);
if let Some(old_rating) = old_rating {
  println!("13 Reasons Why's old rating was {} out of 10",
    old_rating);
}
if let Some(rating) = tv_ratings.get("13 Reasons Why") {
  println!("But I changed my mind, it's now {} out of 10",
    rating);
}

// Remove a key and its value
let removed_value = tv_ratings.remove("The IT Crowd");
if let Some(removed_value) = removed_value {
   println!("The removed series had a rating of {}",
     removed_value);
}

// Iterating accesses all keys and values
println!("All ratings:");
for (key, value) in &tv_ratings {
  println!("{}\t: {}", key, value);
}

// We can iterate mutably
println!("All ratings with 100 as a maximum:");
for (key, value) in &mut tv_ratings {
  *value *= 10;
  println!("{}\t: {}", key, value);
}

// Iterating without referencing the HashMap moves its contents
for _ in tv_ratings {}
// tv_ratings is not usable anymore
```

If you don't need to access both keys and values at the same time, you can iterate over either individually:

```
// Like with the other collections, you can preallocate a size
// to gain some performance
let mut age = HashMap::with_capacity(10);
age.insert("Dory", 8);
age.insert("Nemo", 5);
age.insert("Merlin", 10);
age.insert("Bruce", 9);

// Iterate over all keys
println!("All names:");
for name in age.keys() {
  println!("{}", name);
}

// Iterate over all values
println!("All ages:");
for age in age.values() {
  println!("{}", age);
}

// Iterate over all values and mutate them
println!("All ages in 10 years");
for age in age.values_mut() {
  *age += 10;
  println!("{}", age);
}

```

You can use the entry API to assign default values to keys if they're not yet in the `HashMap`:

```
87    {
88      let age_of_coral = age.entry("coral").or_insert(11);
89      println!("age_of_coral: {}", age_of_coral);
90    }
91    let age_of_coral = age.entry("coral").or_insert(15);
92    println!("age_of_coral: {}", age_of_coral);
93 }
```

How it works...

As mentioned earlier, a `HashMap` is a collection to map one type of data to another. You do this by calling `insert`, and passing your key and its value [9]. If the key already had a value, it will be overwritten. This is why `insert` returns an `Option`: if there was a value before, it returns the old value [27], or otherwise `None`. If you want to make sure that you're not overwriting anything, make sure to check the result of `contains_key` [16] before inserting your value.

Both `get` and `remove` won't crash when called with an invalid key. Instead, they return a `Result`. In the case of `remove`, said `Result` contains the removed value.

As with most collections, you have the options to iterate over your data, by borrowing the key-value pairs[43], borrowing the keys while mutating the values [49], or moving them all [55]. Due to its nature, `HashMap` additionally allows you three more options: borrowing all values [74], mutating all values [80], or borrowing all keys [68]. You probably noticed that one combination is missing: you cannot mutate a key. Ever. This is part of the contract you sign when using a `HashMap`. Further down, where we explain how `HashMap` is implemented, you're going to see that, because the key's hash is actually an index, mutating a key is equivalent to deleting an entry and recreating it. This is reflected nicely in the design choice of not letting you modify keys.

Last but not least, the `Entry` API lets you access an abstraction of a value that might or might not be there. Most of the time, it's used while being paired with `or_insert` in order to insert a default value if the key was not found [88]. If you want to insert a default value based on a closure, you can use `or_insert_with`. Another use for the entry object is to match it against its variants: `Occupied`, or `Vacant`. This results in the same thing as calling `get` directly on a key. Note that in our example, we had to scope the entry access like so:

```
{
    let age_of_coral = age.entry("coral").or_insert(11);
```

```
            println!("age_of_coral: {}", age_of_coral);
        }
        let age_of_coral = age.entry("coral").or_insert(15);
        println!("age_of_coral: {}", age_of_coral);
```

This is because `or_insert` returns a mutable reference to a value. If we had omitted the scope, the second call of `entry` would have borrowed our `age` object at the same time as a mutable reference to it existed, which is an error in Rust's borrowing concept in order to guarantee data race-free access to resources.

If you need to fine-tune your `HashMap` for performance, you can call your usual friends —`with_capacity` [60], `shrink_to_fit`, and `reserve` are also available for it, and work the same way as in other collections.

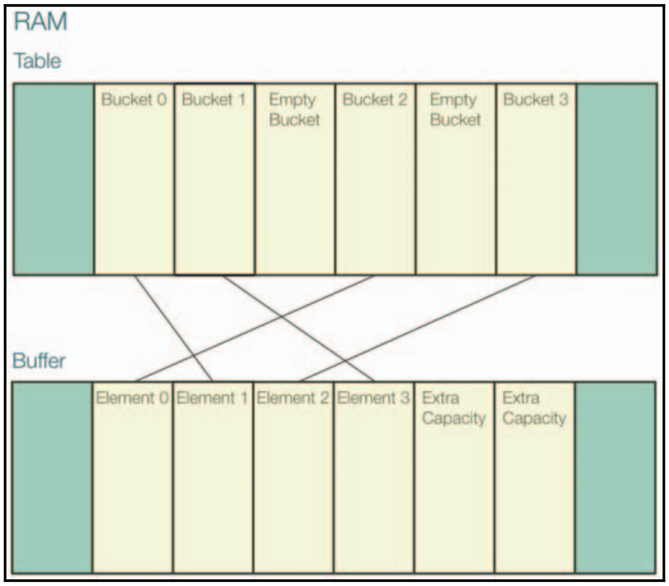
There's more...

Internally, you can imagine the `HashMap` as being implemented as two vectors: a table, and a buffer. Of course, we're simplifying here; there are actually no vectors in the implementation. But this analogy is accurate enough.

If you want to look at the actual implementation, feel free to do so, as Rust is completely open source: `https://github.com/rust-lang/rust/blob/master/src/libstd/collections/hash/table.rs`.

In the background, the buffer stores our values in a sequential fashion. In the front, we have a table storing buckets that don't do much more than point to the element they stand for. When you insert a key-value pair, what happens is:

1. The value gets put in the buffer.
2. The key goes through a hashing function and becomes an index.
3. The table creates a bucket at said index that points to the actual value:



Rust's hashing algorithm doesn't actually generate unique indices, for performance reasons. Instead, Rust uses a clever way to handle hash collisions called **Robin Hood bucket stealing** (`http://codecapsule.com/2013/11/11/robin-hood-hashing/`).

The default hashing algorithm of the standard library has been chosen specifically to protect you from HashDoS attacks (`https://cryptanalysis.eu/blog/2011/12/28/effective-dos-attacks-against-web-application-plattforms-hashdos/`). If you want to squeeze out every bit of performance, you can do that, of your `HashMap` without caring about this particular risk, or you can specify a custom hasher by constructing it with `with_hasher`.

Many people have already implemented various hashers on `crates.io`, so make sure to check them out before rolling with your own solution.

Using a HashSet

The best way to describe a `HashSet` is by describing how it's implemented: `HashMap<K, ()>`. It's just a `HashMap` without any values!

The two best reasons to choose a `HashSet` are:

- You don't want to deal with duplicate values at all, as it doesn't even include them.
- You plan on doing a lot (and I mean a *lot*) of item lookup - that is the question, *Does my collection contain this particular item?*. In a vector, this is done in $O(n)$, while a `HashSet` can do it in $O(1)$.

How to do it...

1. In the folder `src/bin`, create a file called `hashset.rs`.
2. Add the following code, and run it with `cargo run --bin hashset`:

```
use std::collections::HashSet;

fn main() {
  // Most of the interface of HashSet
  // is the same as HashMap, just without
  // the methods that handle values
  let mut books = HashSet::new();
  books.insert("Harry Potter and the Philosopher's Stone");
  books.insert("The Name of the Wind");
  books.insert("A Game of Thrones");

  // A HashSet will ignore duplicate entries
  // but will return if an entry is new or not
  let is_new = books.insert("The Lies of Locke Lamora");
  if is_new {
    println!("We've just added a new book!");
  }

  let is_new = books.insert("A Game of Thrones");
  if !is_new {
    println!("Sorry, we already had that book in store");
  }

  // Check if it contains a key
  if !books.contains("The Doors of Stone") {
    println!("We sadly don't have that book yet");
  }

  // Remove an entry
  let was_removed = books.remove("The Darkness that comes
    before");
    if !was_removed {
      println!("Couldn't remove book; We didn't have it to begin
        with");
    }
    let was_removed = books.remove("Harry Potter and the
      Philosopher's Stone");
    if was_removed {
      println!("Oops, we lost a book");
    }
```

3. Compare different `HashSet`s:

```
    let one_to_five: HashSet<_> = (1..6).collect();
    let five_to_ten: HashSet<_> = (5..11).collect();
    let one_to_ten: HashSet<_> = (1..11).collect();
    let three_to_eight: HashSet<_> = (3..9).collect();

    // Check if two HashSets have no elements in common
    let is_disjoint = one_to_five.is_disjoint(&five_to_ten);
    println!(
      "is {:?} disjoint from {:?}?: {}",
       one_to_five,
       five_to_ten,
       is_disjoint
    );
    let is_disjoint = one_to_five.is_disjoint(&three_to_eight);
    println!(
      "is {:?} disjoint from {:?}?: {}",
       one_to_five,
       three_to_eight,
       is_disjoint
    );

    // Check if a HashSet is fully contained in another
    let is_subset = one_to_five.is_subset(&five_to_ten);
    println!(
      "is {:?} a subset of {:?}?: {}",
       one_to_five,
       five_to_ten,
       is_subset
    );
    let is_subset = one_to_five.is_subset(&one_to_ten);
    println!(
      "is {:?} a subset of {:?}?: {}",
       one_to_five,
       one_to_ten,
       is_subset
    );
```

```

// Check if a HashSet fully contains another
let is_superset = three_to_eight.is_superset(&five_to_ten);
println!(
  "is {:?} a superset of {:?}?: {}",
  three_to_eight,
  five_to_ten,
  is_superset
);
let is_superset = one_to_ten.is_superset(&five_to_ten);
println!(
"is {:?} a superset of {:?}?: {}",
  one_to_ten,
  five_to_ten,
  is_superset
);
```

4. Join two `HashSet`s in various ways:

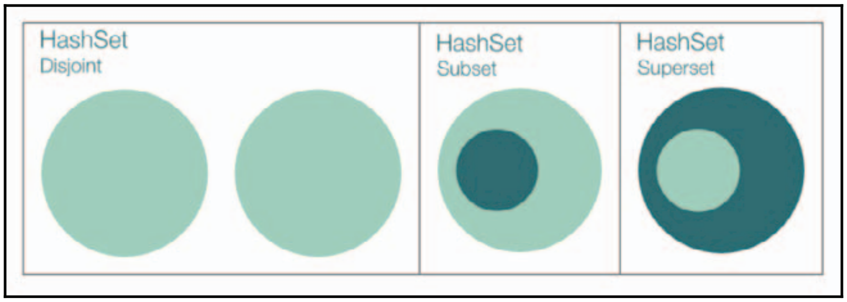
```
// Get the values that are in the first HashSet
// but not in the second
let difference = one_to_five.difference(&three_to_eight);
println!(
  "The difference between {:?} and {:?} is {:?}",
   one_to_five,
   three_to_eight,
   difference
);

// Get the values that are in either HashSets, but not in both
let symmetric_difference =
  one_to_five.symmetric_difference(&three_to_eight);
println!(
  "The symmetric difference between {:?} and {:?} is {:?}",
   one_to_five,
   three_to_eight,
   symmetric_difference
);

// Get the values that are in both HashSets
let intersection = one_to_five.intersection(&three_to_eight);
println!(
  "The intersection difference between {:?} and {:?} is {:?}",
   one_to_five,
   three_to_eight,
   intersection
122   );
123
124   // Get all values in both HashSets
125   let union = one_to_five.union(&three_to_eight);
126   println!(
127     "The union difference between {:?} and {:?} is {:?}",
128      one_to_five,
129      three_to_eight,
130      union
131   );
132 }
```

How it works...

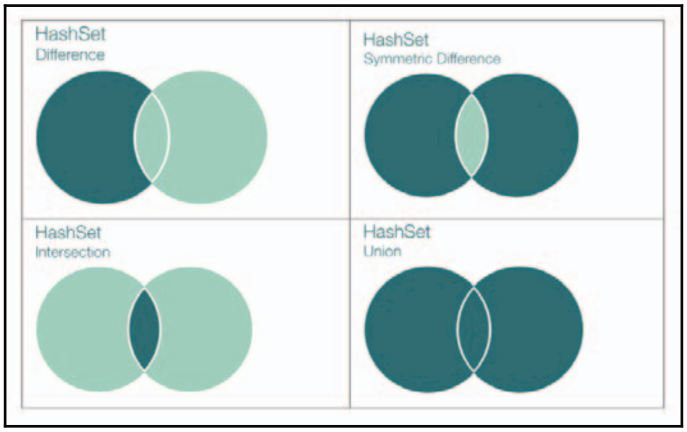
As a `HashSet` is a kind of `HashMap`, most of its interface is pretty similar. The major difference is that the methods that would return a key's value in the `HashMap` instead simply return a `bool` on the `HashSet` in order to tell if a key already existed or not [14].

Additionally, `HashSet` brings a few methods for analyzing two sets [46 to 92] and joining them [96 to 131]. If you've ever heard of set theory or Venn diagrams, or done a bit of SQL, you're going to recognize it all. Otherwise, I advise you to run the example, and study the outputs in combination with the relevant comments.

Some illustrations might help you. For the analytical methods, the dark green part is the object of reference:

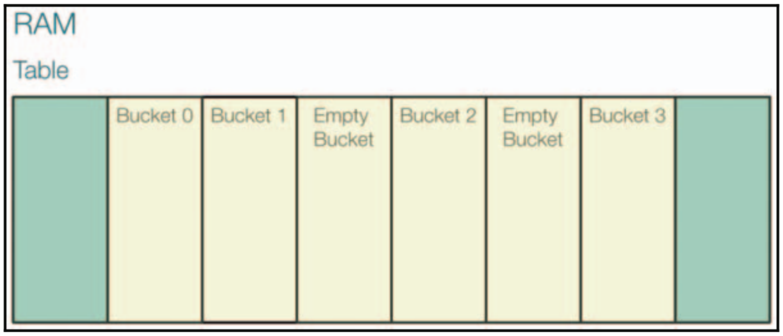


For the selecting methods, the dark green part is the one that is returned:



There's more...

No big surprises in the implementation of the `HashSet`, since it's exactly the same as the `HashMap`, just without any values!



Creating an own iterator

When you create an infinitely applicable algorithm or a collection-like structure, it's really nice to have the dozens of methods that an iterator provides at your disposal. For this, you will have to know how to tell Rust to implement them for you.

How to do it...

1. In the folder `src/bin`, create a file called `own_iterator.rs`.
2. Add the following code, and run it with `cargo run --bin own_iterator`:

```
fn main() {
  let fib: Vec<_> = fibonacci().take(10).collect();
  println!("First 10 numbers of the fibonacci sequence: {:?}",
    fib);

  let mut squared_vec = SquaredVec::new();
  squared_vec.push(1);
  squared_vec.push(2);
  squared_vec.push(3);
  squared_vec.push(4);
  for (index, num) in squared_vec.iter().enumerate() {
    println!("{}^2 is {}", index + 1, num);
  }
}


fn fibonacci() -> Fibonacci {
  Fibonacci { curr: 0, next: 1 }
}
struct Fibonacci {
curr: u32,
next: u32,
}
// A custom iterator has to implement
// only one method: What comes next
impl Iterator for Fibonacci {
  type Item = u32;
  fn next(&mut self) -> Option<u32> {
    let old = self.curr;
    self.curr = self.next;
    self.next += old;
    Some(old)
  }
}


use std::ops::Mul;
struct SquaredVec<T>
where
T: Mul + Copy,
{
    vec: Vec<T::Output>,
  }
  impl<T> SquaredVec<T>
  where
  T: Mul + Copy,
  {
    fn new() -> Self {
    SquaredVec { vec: Vec::new() }
  }
  fn push(&mut self, item: T) {
    self.vec.push(item * item);
  }
}

  // When creating an iterator over a collection-like struct
  // It's best to just allow it to be convertible into
  // a slice of your underlying type.
  // This way you automatically implemented a bunch of methods
  // and are flexible enough to change your implementation later
    on
  use std::ops::Deref;
  impl<T> Deref for SquaredVec<T>
  where
  T: Mul + Copy,
  {
    type Target = [T::Output];
    fn deref(&self) -> &Self::Target {
    &self.vec
  }
}
```

How it works...

In our little example here, we are going to look at two different uses for an iterator:

- `fibonacci()`, which returns an infinite range of the **Fibonacci sequence**
- `SquaredVec`, which implements a (very) small subset of a `Vec` with a twist: it squares all items

The Fibonacci sequence is defined as a series of numbers, starting from 0 and 1, where the next number is the sum of the last two. It starts like this: 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on.
The first two are 0 and 1 per definition. The next one is their sum — *0* + *1* = *1*. After that comes *1* + *1* = 2. Then *2* + *1* = *3*. *3* + *2* = *5*. Repeat ad infinitum.

An algorithm can be turned into an iterator by implementing the `Iterator` trait. This is pretty simple, as it only expects you to provide the type you're iterating over and a single method, `next`, which fetches the next item. If the iterator doesn't have any items left, it should return `None`, otherwise `Some`. Our Fibonacci iterator always returns `Some` item, which makes it an infinite iterator [31].

Our `SquaredVec`, on the other hand, is more of a collection than an algorithm. In lines [37] to [53], we wrap the minimum of the `Vec` interface — we can create a `SquaredVec`, and we can fill it. Our type constraints `Mul + Copy` mean that the item the user wants to store has to be able to be copied and to be multiplied. We need this in order to square it, but it's not relevant for the iterator. `T::Output` is just the type that a multiplication would return, which most of the time is going to be `T` itself.

We could implement the `Iterator` trait again, but there's an easier option that will provide you with even more methods. We can allow our struct to be implicitly convertible into a slice `[T]`, which will not only implement `Iterator` for you, but also a whole bunch of other methods. Because `Vec` implements it already, you can just return it like that [67]. If your underlying collection didn't provide a slice conversion, you could still go the same way as before and implement the `Iterator` trait manually.

There's more...

If you have a lot of complex logic to perform in the iterator and want to separate it a bit from your collection, you can do so by providing your collection with the `IntoIterator` trait instead. This would allow you to return a struct specifically made for your iteration, which itself provides the `Iterator` trait.

Using a slab

Some algorithms require you to hold access tokens to data that may or may not exist. This could be solved in Rust by using `Vec<Option<T>>`, and treating the index of your data as a token. But we can do better! `slab` is an optimized abstraction of exactly this concept.

While it is not meant as a general-purpose collection, `slab` can help you a lot if you use it in the right places.

How to do it...

1. Open the `Cargo.toml` file that has been generated earlier for you.
2. Under `[dependencies]`, add the following line:

    ```
    slab = "0.4.0"
    ```

3. If you want, you can go to slab's crates.io page (`https://crates.io/crates/slab`) to check for the newest version, and use that one instead.
4. In the folder `bin`, create a file called `slab.rs`.
5. Add the following code, and run it with `cargo run --bin slab`:

```rust
extern crate slab;
use slab::{Slab, VacantEntry};

fn main() {
  // A slab is meant to be used as a limited buffer
  // As such, you should initialize it with a pre-
  // defined capacity
  const CAPACITY: usize = 1024;
  let mut slab = Slab::with_capacity(CAPACITY);

  // You cannot simply access a slab's entry by
  // index or by searching it. Instead, every
  // insert gives you a key that you can use to
  // access its entry
  let hello_key = slab.insert("hello");
  let world_key = slab.insert("world");

  println!("hello_key -> '{}'", slab[hello_key],);
  println!("world_key -> '{}'", slab[world_key],);


  // You can pass an "empty spot" around
  // in order to be filled
  let data_key = {
  let entry = slab.vacant_entry();
    fill_some_data(entry)
  };
  println!("data_key -> '{}'", slab[data_key],);

  // When iterating, you get a key-value pair
  for (key, val) in &slab {
```

```
32      println!("{} -> {}", key, val);
33    }
34
35    // If you want to keep your slab at a constant
36    // capacity, you have to manually check its
37    // length before inserting data
38    if slab.len() != slab.capacity() {
39      slab.insert("the slab is not at capacity yet");
40    }
41  }
42
43
44  fn fill_some_data(entry: VacantEntry<&str>) -> usize {
45    let data = "Some data";
46    // insert() consumes the entry
47    // so we need to get the key before
48    let key = entry.key();
49    entry.insert(data);
50    key
51  }
```

How it works...

A slab is very similar to a vector, with one quintessential difference: you don't get to choose your index. Instead, when inserting data [15], you receive the data's index as a kind of *key* that you can use to access it again. It is your responsibility to store this key somewhere; otherwise, the only way to retrieve your data is by iterating over your slab. The flipside is that you don't have to provide any key either. In contrast to a `HashMap`, you don't need any hashable objects at all.

A situation in which this is useful is in a connection pool: if you have multiple clients who want to access individual resources, you can store said resources in a slab and provide the clients with their key as a kind of token.

This example suits the second use case of a slab really well. Suppose you only accept a certain amount of connections at a given time. When accepting a connection, you don't care about the exact index, or the way it is stored. Instead, you care only that it is stored in a retrievable way, and that it doesn't exceed your limit. This fits the bill of the slab quite nicely, which is why most of the time you won't be creating a slab with `Slab::new()`, but with `with_capacity`, set to a constant upper limit [9].

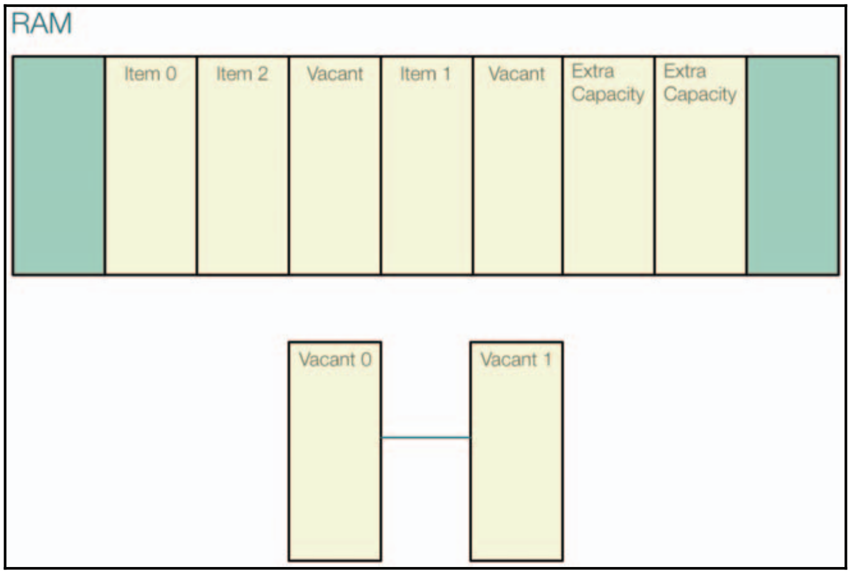
The slab, however, does not impose this limit by itself, as it behaves exactly like the vector in the way it handles capacity: as soon as the length exceeds the capacity, the slab reallocates all objects to a bigger block of memory and ups the capacity. This is why, when dealing with upper bonds, you should insert your data with some kind of variation of line [38]:

```
if slab.len() != slab.capacity() {
    slab.insert("the slab is not at capacity yet");
}
```

Other valid approaches would be to wrap an insertion in a function that returns a `Result`, or an `Option`.

There's more...

A slab is backed by a `Vec<Entry>`. You might remember the `Entry` from our recipe about the `HashMap` earlier. It is the same as an `Option`, with the difference that its variants are not called `Some(...)` and `None`, but `Occupied(...)` and `Vacant`. This means that, in a nutshell, a slab is implemented as a vector with holes in it:



Additionally, in order to guarantee fast occupation of vacant spots, the slab keeps a linked list of all vacant entries.

3 Handling Files and the Filesystem

In this chapter, we will cover the following recipes:

- Working with text files
- Handling bytes
- Working with binary files
- Compressing and decompressing data
- Traversing the filesystem
- Finding files with glob patterns

Introduction

In these times of big data, machine learning, and cloud services, you cannot rely on having all of your data always in memory. Rather, you need to be able to effectively inspect and traverse the filesystem and manipulate its content at your leisure.

Examples of things you will be able to do after reading this chapter include configuring files in subdirectories with different naming variations, saving your data in efficient binary formats, reading protocols generated by other programs, and compressing your data in order to send it over the internet at fast speeds.

Working with text files

In this recipe, we will learn how to read, write, create, truncate, and append text files. Armed with this knowledge, you will be able to apply all other recipes to files instead of in-memory strings.

How to do it...

1. Create a Rust project to work on during this chapter with `cargo new chapter-three`.
2. Navigate into the newly created `chapter-three` folder. For the rest of this chapter, we will assume that your command line is currently in this directory.
3. Inside the `src` folder, create a new folder called `bin`.
4. Delete the generated `lib.rs` file, as we are not creating a library.
5. In the `src/bin` folder, create a file called `text_files.rs`.
6. Add the following code and run it with `cargo run --bin text_files`:

```
use std::fs::{File, OpenOptions};
use std::io::{self, BufReader, BufWriter, Lines, Write};
use std::io::prelude::*;

fn main() {
  // Create a file and fill it with data
  let path = "./foo.txt";
  println!("Writing some data to '{}'", path);
  write_file(path, "Hello World!\n").expect("Failed to write to
   file");
  // Read entire file as a string
  let content = read_file(path).expect("Failed to read file");
  println!("The file '{}' contains:", path);
  println!("{}", content);

  // Overwrite the file
  println!("Writing new data to '{}'", path);
  write_file(path, "New content\n").expect("Failed to write to
   file");
  let content = read_file(path).expect("Failed to read file");
  println!("The file '{}' now contains:", path);
  println!("{}", content);

  // Append data to the file
  println!("Appending data to '{}'", path);
  append_file(path, "Some more content\n").expect("Failed to append to file");
  println!("The file '{}' now contains:", path);
  // Read file line by line as an iterator
  let lines = read_file_iterator(path).expect("Failed to read file");
  for line in lines {
    println!("{}", line.expect("Failed to read line"));
  }

  append_and_read(path, "Last line in the file, goodbye").expect("Failed to read and write file");
  }
```

7. These are the functions called by the `main()` function:

```
fn read_file(path: &str) -> io::Result<String> {
  // open() opens the file in read-only mode
  let file = File::open(path)?;
  // Wrap the file in a BufReader
  // to read in an efficient way
  let mut buf_reader = BufReader::new(file);
  let mut content = String::new();
  buf_reader.read_to_string(&mut content)?;
  Ok(content)
}

fn read_file_iterator(path: &str) ->
  io::Result<Lines<BufReader<File>>> {
  let file = File::open(path)?;
  let buf_reader = BufReader::new(file);
  // lines() returns an iterator over lines
  Ok(buf_reader.lines())
}


fn write_file(path: &str, content: &str) -> io::Result<()> {
  // create() opens a file with the standard options
  // to create, write and truncate a file
  let file = File::create(path)?;
  // Wrap the file in a BufReader
  // to read in an efficient way
  let mut buf_writer = BufWriter::new(file);
  buf_writer.write_all(content.as_bytes())?;
  Ok(())
}

```

```
fn append_file(path: &str, content: &str) -> io::Result<()> {
  // OpenOptions lets you set all options individually
  let file = OpenOptions::new().append(true).open(path)?;
  let mut buf_writer = BufWriter::new(file);
  buf_writer.write_all(content.as_bytes())?;
  Ok(())
}
```

8. Reading and writing on the same handle:

```
fn append_and_read(path: &str, content: &str) -> io::Result<()
{
  let file =
    OpenOptions::new().read(true).append(true).open(path)?;
  // Passing a reference of the file will not move it
  // allowing you to create both a reader and a writer
  let mut buf_reader = BufReader::new(&file);
  let mut buf_writer = BufWriter::new(&file);

  let mut file_content = String::new();
  buf_reader.read_to_string(&mut file_content)?;
  println!("File before appending:\n{}", file_content);

  // Appending will shift your positional pointer
  // so you have to save and restore it
  let pos = buf_reader.seek(SeekFrom::Current(0))?;
  buf_writer.write_all(content.as_bytes())?;
  // Flushing forces the write to happen right now
  buf_writer.flush()?;
  buf_reader.seek(SeekFrom::Start(pos))?;

  buf_reader.read_to_string(&mut file_content)?;
  println!("File after appending:\n{}", file_content);

  Ok(())
}
```

How it works...

Our `main` function is split into three parts:

1. Creating a file.
2. Overwriting a file, which in this context is called *truncating*.
3. Appending to a file.

In the first two parts, we load the entire content of the file into a single `String` and display it [11 and 18]. In the last one, we iterate over the individual lines in the file and print them [28].

`File::open()` opens a file in read-only mode and returns you a handle to it [39]. Because this handle implements the `Read` trait, we could now just directly read it into a string with `read_to_string`. However, in our examples, we wrap it first in a `BufReader`[42]. This is because a dedicated reader can greatly improve the performance of their resource access by collecting read instructions, which is called *buffering*, and executing them in big batches. For the first reading example, `read_file`[37], this doesn't make any difference whatsoever, as we read it all in one go anyway. We still use it because it is a good practice, as it allows us to flexibly change the exact reading mechanisms of our function later on without worrying about performance. If you want to see a function where a `BufReader` actually does something, look a little further down, to `read_file_iterator`[48]. It appears to read the file line by line. This would be a very inefficient operation when dealing with a large file, which is why a `BufReader` actually reads a large chunk of the file in one go and then returns that segment line by line. The result is optimized file reading without us even noticing or caring what is going on in the background, which is pretty convenient.

`File::create()` creates a new file if it doesn't exist, otherwise it truncates the file. In any case, it returns the same kind of `File` handle like `File::open()` did before. Another similarity is the `BufWriter` we wrap around it. Just like with the `BufReader`, we would be able to access our underlying file without it, but use it to optimize future accesses as well.

There are more options than just opening a file in read-only or truncation mode. We can use them by creating our file handle with `OpenOptions`[69], which use the builder pattern we explored in the *Using the builder pattern* section in `Chapter 1`, *Learning the Basics*. In our example, we are interested in the `append` mode, which lets us add content to a file instead of overwriting it on every access.

For a full list of all available options, see the OpenOption documentation: `https://doc.rust-lang.org/std/fs/struct.OpenOptions.html`.

We can read and write on the same file handle. For this, when creating the `ReadBuf` and `WriteBuf`, we pass a reference to the file instead of moving it, as the buffers would otherwise consume the handle, making sharing impossible:

```
let mut buf_reader = BufReader::new(&file);
let mut buf_writer = BufWriter::new(&file);
```

When doing this, be careful when appending and reading the same handle. The internal pointer that stored the current reading position might get shifted when appending. If you want to first read, then append, and then continue reading, you should save the current position before writing and then restore it afterward.

We can access our current position in the file by calling `seek(SeekFrom::Current(0))`[89]. `seek` moves our pointer by a certain amount of bytes and returns its new position. `SeekFrom::Current(0)` means that the distance we want to move is exactly zero bytes away from where we are right now. Because of this, as we don't move at all, `seek` will return our current position.

Then, we append our data using `flush`[92]. We have to call this method, as a `BufWriter` would normally wait for the actual writing until it is dropped, that is, it is no longer in scope. As we want to read before that happens, we use `flush` to force a write.

Finally, we get ready to read again by restoring our position from before, seeking it again:

```
buf_reader.seek(SeekFrom::Start(pos))?;
```

I invite you to run the code, look at the results, and then compare them with the output after commenting this line out.

There's more...

Instead of opening up a new file handle in every function, we could open one single handle at the beginning of the program and pass it around to every function that needs it. This is a trade-off—we get more performance if we don't repeatedly lock and unlock a file. In turn, we disallow other processes to access our file while our program is running.

See also

- *Using the builder pattern* recipe in `Chapter 1`, *Learning the Basics*

Handling bytes

When designing your own protocol or using existing ones, you have to be able to comfortably move around and manipulate binary buffers. Luckily, the extended standard library ecosystem provides the `byteorder` crate to fulfill all your binary needs with various bits and pieces of reading and writing functionality.

Getting ready

In this chapter, we are going to talk about *endianness*. It is a way of describing how the values in a buffer are ordered. There are two ways to order them:

- Put the smallest one first (*Little Endian*)
- Put the biggest one first (*Big Endian*)

Let's try an example. Suppose we wanted to save the hexadecimal value `0x90AB12CD`. We first have to split it into bits of `0x90`, `0xAB`, `0x12`, and `0xCD`. We now can either store them with the biggest value first (Big Endian), `0x90 - 0xAB - 0x12 - 0xCD`, or we could write the smallest number first (Little Endian), `0xCD - 0x12 - 0xAB - 0x90`.

As you can see, it's the exact same set of values, but flipped. If this short explanation left you confused, I advise you to look at this excellent explanation by the University of Maryland's Department of Computer Science: `https://web.archive.org/web/20170808042522/http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/endian.html`.

There is no *better* endianness. Both are used in different domains: microprocessors, such as Intel, use Little Endian, and internet protocols, such as TCP, IPv4, IPv6, and UDP, use Big Endian. This is not a rule but rather a convention maintained to be backward compatible. As such, there are exceptions.

When designing your own protocol, orient yourself on the endianness of similar protocols, choose one and simply stick to it.

How to do it...

Follow these steps:

1. Open the `Cargo.toml` file that was generated earlier for you.
2. Under `[dependencies]`, add the following line:

   ```
   byteorder = "1.1.0"
   ```

3. If you want, you can go to `byteorder`'s crates.io page (`https://crates.io/crates/byteorder`) to check for the newest version and use that one instead.
4. In the `bin` folder, create a file called `bytes.rs`.
5. Add the following code and run it with `cargo run --bin bytes`:

```
extern crate byteorder;
use std::io::{Cursor, Seek, SeekFrom};
use byteorder::{BigEndian, LittleEndian, ReadBytesExt,
WriteBytesExt};

fn main() {
  let binary_nums = vec![2, 3, 12, 8, 5, 0];
  // Wrap a binary collection in a cursor
  // to provide seek functionality
  let mut buff = Cursor::new(binary_nums);
  let first_byte = buff.read_u8().expect("Failed to read
    byte");
  println!("first byte in binary: {:b}", first_byte);

  // Reading advances the internal position,
  // so now we read the second
  let second_byte_as_int = buff.read_i8().expect("Failed to
    read byte as int");
  println!("second byte as int: {}", second_byte_as_int);

  // Overwrite the current position
  println!("Before: {:?}", buff);
  buff.write_u8(123).expect("Failed to overwrite a byte");
  println!("After: {:?}", buff);


  // Set and get the current position
  println!("Old position: {}", buff.position());
  buff.set_position(0);
```

```
    println!("New position: {}", buff.position());

    // This also works using the Seek API
    buff.seek(SeekFrom::End(0)).expect("Failed to seek end");
    println!("Last position: {}", buff.position());

    // Read and write in specific endianness
    buff.set_position(0);
    let as_u32 = buff.read_u32::<LittleEndian>()
      .expect("Failed to read bytes");
    println!(
      "First four bytes as u32 in little endian order:\t{}",
       as_u32
    );

    buff.set_position(0);
    let as_u32 = buff.read_u32::<BigEndian>().expect("Failed to
      read bytes");
    println!("First four bytes as u32 in big endian order:\t{}",
      as_u32);

    println!("Before appending: {:?}", buff);
    buff.seek(SeekFrom::End(0)).expect("Failed to seek end");
    buff.write_f32::<LittleEndian>(-33.4)
      .expect("Failed to write to end");
    println!("After appending: {:?}", buff);

    // Read a sequence of bytes into another buffer
    let mut read_buffer = [0; 5];
    buff.set_position(0);
    buff.read_u16_into::<LittleEndian>(&mut read_buffer)
      .expect("Failed to read all bytes");
    println!(
      "All bytes as u16s in little endian order: {:?}",
       read_buffer
    );
  }
```

How it works...

First things first, we need a binary source. In our example, we simply use a vector. Then, we wrap it into a `Cursor`, as it provides us with a `Seek` implementation and some methods for our convenience.

The `Cursor` has an internal position count that keeps track of which byte we are accessing at the moment. As expected, it starts at zero. With `read_u8` and `read_i8`, we can read the current byte as an unsigned or signed number. This will advance the position by one. Both do the same thing, but return a different type.

Did you notice that we printed the returned byte by using `{:b}` as the formatting parameter [11]?

```
println!("first byte in binary: {:b}", first_byte);
```

By doing so, we tell the underlying `format!` macro to interpret our byte as binary, which is why it will print `10` instead of `2`. If you want to, try replacing `{}` in our other printing calls with `{:b}` and compare the results.

The current position can be read with `position()` [25] and set with `set_position()`. You can also manipulate your position with the more verbose `Seek` API we introduced in the last recipe [30]. When using `SeekFrom::End`, keep in mind that this will *not* count backward from the end. For example, `SeekFrom::End(1)` will point to one byte *after* the end of the buffer and not before. The behavior is defined in this way because, maybe somewhat surprisingly, it is legal to seek past a buffer. This can be useful when writing, as it will simply pad the space between the end of the buffer and the cursor position with zeros.

When dealing with more then one byte, you will need to specify the endianness of the bytes via type annotation. Reading or writing will then advance the position by the number of bytes read or written, which is why, in our example code, we need to frequently reset the position with `set_position(0)`. Note that when you write past the end, you will always simply extend the buffer [48].

If you know that you want to read a very specific amount of bytes, like when parsing a well-defined protocol, you can do so by providing a fixed-size array and filling it by post-fixing your `read` with `_into`, like this:

```
// Read exactly five bytes
let mut read_buffer = [0; 5];
buff.read_u16_into::<LittleEndian>(&mut read_buffer).expect("Failed to
fill buffer");
```

When doing so, the read will return an error if the buffer was not filled completely, in which case its contents are undefined.

There's more...

There are various aliases in the byte order crate to ease your endianness annotation. The `BE` alias, for Big Endian, and the `LE` alias, for Little Endian, are useful if you don't want to type as much. On the other hand, if you keep forgetting which endianness is used where, you can use `NativeEndian`, which sets itself to the default endianness of your operating system, and `NetworkEndian`, for Big Endian.

To use them, you will have to drag them into scope like this:

```
use byteorder::{BE, LE, NativeEndian, NetworkEndian};
```

Working with binary files

We are now going to combine what we learned in the last two chapters in order to parse and write binary files. This will prove essential when you plan on implementing custom, manual processing of file types such as PDFs, torrents, and ZIPs. It will also come in handy when designing custom file types for your own use cases.

How to do it...

1. If you haven't done it already in the last chapter, open the `Cargo.toml` file that was generated earlier for you.
2. Under `[dependencies]`, add the following line:

   ```
   byteorder = "1.1.0"
   ```

3. If you want, you can go to byteorder's crates.io page (`https://crates.io/crates/byteorder`) to check for the newest version and use that one instead.
4. In the `bin` folder, create a file called `binary_files.rs`.
5. Add the following code and run it with `cargo run --bin binary_files`:

```
extern crate byteorder;
use byteorder::{ByteOrder, ReadBytesExt, WriteBytesExt, BE,
LE};
use std::fs::File;
use std::io::{self, BufReader, BufWriter, Read};
use std::io::prelude::*;
```

```


fn main() {
  let path = "./bar.bin";
  write_dummy_protocol(path).expect("Failed write file");
  let payload = read_protocol(path).expect("Failed to read
    file");
  print!("The protocol contained the following payload: ");
  for num in payload {
    print!("0x{:X} ", num);
  }
  println!();
}
```

6. Create a binary file:

```
// Write a simple custom protocol
fn write_dummy_protocol(path: &str) -> io::Result<()> {
  let file = File::create(path)?;
  let mut buf_writer = BufWriter::new(file);

  // Let's say our binary file starts with a magic string
  // to show readers that this is our protocoll
  let magic = b"MyProtocol";
  buf_writer.write_all(magic)?;

  // Now comes another magic value to indicate
  // our endianness
  let endianness = b"LE";
  buf_writer.write_all(endianness)?;

  // Let's fill it with two numbers in u32
  buf_writer.write_u32::<LE>(0xDEAD)?;
  buf_writer.write_u32::<LE>(0xBEEF)?;

  Ok(())
}
```

7. Read and parse the file:

```
fn read_protocol(path: &str) -> io::Result<Vec<u32>> {
  let file = File::open(path)?;
  let mut buf_reader = BufReader::new(file);

  // Our protocol has to begin with a certain string
  // Namely "MyProtocol", which is 10 bytes long
  let mut start = [0u8; 10];
  buf_reader.read_exact(&mut start)?;
  if &start != b"MyProtocol" {
    return Err(io::Error::new(
      io::ErrorKind::Other,
      "Protocol didn't start with the expected magic string",
    ));
  }

  // Now comes the endianness indicator
  let mut endian = [0u8; 2];
  buf_reader.read_exact(&mut endian)?;
  match &endian {
    b"LE" => read_protocoll_payload::<LE, _>(&mut buf_reader),
    b"BE" => read_protocoll_payload::<BE, _>(&mut buf_reader),
    _ => Err(io::Error::new(
      io::ErrorKind::Other,
      "Failed to parse endianness",
    )),
  }
}

// Read as much of the payload as possible
fn read_protocoll_payload<E, R>(reader: &mut R) ->
  io::Result<Vec<u32>>
where
E: ByteOrder,
R: ReadBytesExt,
{
  let mut payload = Vec::new();
  const SIZE_OF_U32: usize = 4;
  loop {
  let mut raw_payload = [0; SIZE_OF_U32];
  // Read the next 4 bytes
  match reader.read(&mut raw_payload)? {
  // Zero means we reached the end
  0 => return Ok(payload),
  // SIZE_OF_U32 means we read a complete number
  SIZE_OF_U32 => {
    let as_u32 = raw_payload.as_ref().read_u32::<E>()?;
    payload.push(as_u32)
  }
  // Anything else means the last element was not
  // a valid u32
  _ => {
    return Err(io::Error::new(
    io::ErrorKind::UnexpectedEof,
      "Payload ended unexpectedly",
    ))
  }
97   }
98  }
99 }
```

How it works...

To demonstrate how to read and write a binary file, we will create a little custom binary protocol. It will start with what is called a *magic number*, that is, a certain hardcoded value. Our magic number will be the binary representation of the `MyProtocol` string. We can put a `b` before the string to tell Rust that we want the text to be represented as a binary slice (`&[u8]`) instead of a string slice(`&str`) [26].

Many protocols and files start with magic numbers to indicate what they are. For example, the internal headers of `.zip` files start with the magic hex numbers `0x50` and `0x4B`. These represent the initials *PH* in ASCII, which is short for the name of its creator Phil Katz. Another example would be PDF; it starts with `0x25`, `0x50`, `0x44`, and `0x46`, which stands for `PDF%`, followed by a version number.

Afterward, we follow it by the binary representation of either `LE` or `BE` to tell the reader the endianness of the rest of the data [31]. Finally, we have the payload, which is just an arbitrary amount of `u32` numbers, encoded in the aforementioned endianness [35 and 36]. By putting `0x` in front of our number, we tell Rust to treat it as a hexadecimal number and convert it into decimal for us. As such, Rust treats `0xDEAD` as the same value as `57005`.

Let's put it all together and write a binary file containing `MyProtocolLE5700548879`. Other files we could have created in accordance with our protocol would be `MyProtocolBE92341739241842518425` or `MyProtocolLE0000`, and so on.

If you read the previous recipes, `write_dummy_protocol` should be easy to understand. We use a combination of good old `write_all` from the standard library to write our binary texts and `write_u32` from `byteorder` to write the values that require an endianness.

The reading of the protocol is split into the `read_protocol` and `read_protocol_payload` functions. The first verifies the validity of the protocol by reading the magic numbers and then calls the latter, which reads the remaining numbers as the payload.

We validate the magic numbers as follows:

1. As we know the exact size of the magic numbers used, prepare buffers of those exact sizes.
2. Fill them with just as many bytes.
3. Compare the bytes with the expected magic number.
4. If they don't match, return an error.

After parsing both magic numbers, we can parse the actual *data* contained in the payload. Remember, we defined it as an arbitrary amount of `32` bit (`= 4` bytes) long, unsigned numbers. To parse them, we are going to repeatedly read up to four bytes into a buffer called `raw_payload`. We are then going to examine the amount of bytes that were actually read. This number can have three forms in our case, as demonstrated nicely by our `match`.

The first value we are interested in is zero, which means that there are no more bytes to read, that is, we have reached the end. In this case, we can return our payload:

```
        // Zero means we reached the end
        0 => return Ok(payload),
```

The second value is `SIZE_OF_U32`, which we have previously defined as four. Receiving this value means that our reader has successfully read four bytes into a four-byte-long buffer. This means that we have successfully read a value! Let's parse it into a `u32` and push it into our `payload` vector:

```
        // SIZE_OF_U32 means we read a complete number
        SIZE_OF_U32 => {
            let as_u32 = raw_payload.as_ref().read_u32::<E>()?;
            payload.push(as_u32)
        }
```

We have to call `as_ref()` on our buffer because a fixed-size array doesn't implement `Read`. Since a slice does implement said trait and a reference to an array is implicitly convertible into a slice, we work on a reference to `raw_payload` instead.

The third and last value we can expect is everything other than zero or four. In this case, the reader was not able to read four bytes, which means that our buffer ended in something other than a `u32` and is malformed. We can react to this by returning an error:

```
// Anything else means the last element was not
// a valid u32
_ => {
    return Err(io::Error::new(
        io::ErrorKind::UnexpectedEof,
        "Payload ended unexpectedly",
        ))
    }
}
```

There's more...

When reading a malformed protocol, we reuse `io::ErrorKind` to show what exactly went wrong. In the recipes of `Chapter 6`, *Handling Errors*, you will learn how to provide your own error to better separate your areas of failure. If you want, you could read them now and then return here to improve our code.

The errors that need to be pushed into an own variant are:

- `InvalidStart`
- `InvalidEndianness`
- `UnexpectedEndOfPayload`

Another improvement to the code would be to put all of our strings, namely `MyProtocol`, `LE`, and `BE`, into their own constants, as in the following line:

```
const PROTOCOL_START: &[u8] = b"MyProtocol";
```

The provided code in this recipe and some others doesn't use many constants, as they proved to be somewhat harder to understand in printed form. In real code bases, however, be sure to always put strings that you find yourself copy-pasting into own constants!

See also

- *Providing user-defined error types* recipes in `Chapter 6`, *Handling Errors*

Compressing and decompressing data

In today's age of bloated websites and daily new web frameworks, many sites feel way more sluggish than they used (and ought) to. One way to mitigate this is by compressing your resources before sending them and then decompressing them when received. This has become the (often ignored) standard on the web. For this purpose, this recipe will teach you how to compress and decompress any kind of data with different algorithms.

How to do it...

Follow these steps:

1. Open the `Cargo.toml` file that was generated earlier for you.
2. Under `[dependencies]`, add the following line:

```
flate2 = "0.2.20"
```

   If you want, you can go to `flate2`'s crates.io page (`https://crates.io/crates/flate2`) to check for the newest version and use that one instead.

2. In the `bin` folder, create a file called `compression.rs`.
3. Add the following code and run it with `cargo run --bin compression`:

```
extern crate flate2;

use std::io::{self, SeekFrom};
use std::io::prelude::*;

use flate2::{Compression, FlateReadExt};
use flate2::write::ZlibEncoder;
use flate2::read::ZlibDecoder;

use std::fs::{File, OpenOptions};
use std::io::{BufReader, BufWriter, Read};

fn main() {
  let bytes = b"I have a dream that one day this nation will rise up, \
  and live out the true meaning of its creed";
  println!("Original: {:?}", bytes.as_ref());
  // Conpress some bytes
  let encoded = encode_bytes(bytes.as_ref()).expect("Failed to encode bytes");
  println!("Encoded: {:?}", encoded);
  // Decompress them again
  let decoded = decode_bytes(&encoded).expect("Failed to decode bytes");
  println!("Decoded: {:?}", decoded);

  // Open file to compress
  let original = File::open("ferris.png").expect("Failed to open file");
  let mut original_reader = BufReader::new(original);

  // Compress it
  let data = encode_file(&mut original_reader).expect("Failed to encode file");

  // Write compressed file to disk
  let encoded = OpenOptions::new()
    .read(true)
    .write(true)
    .create(true)
    .open("ferris_encoded.zlib")
    .expect("Failed to create encoded file");
  let mut encoded_reader = BufReader::new(&encoded);
  let mut encoded_writer = BufWriter::new(&encoded);
  encoded_writer
    .write_all(&data)
    .expect("Failed to write encoded file");


  // Jump back to the beginning of the compressed file
  encoded_reader
    .seek(SeekFrom::Start(0))
    .expect("Failed to reset file");

  // Decompress it
  let data = decode_file(&mut encoded_reader).expect("Failed to
    decode file");

  // Write the decompressed file to disk
  let mut decoded =
    File::create("ferris_decoded.png").expect("Failed to
create
      decoded file");
  decoded
    .write_all(&data)
    .expect("Failed to write decoded file");
}
```

5. These are the functions doing the actual encoding and decoding:

```
fn encode_bytes(bytes: &[u8]) -> io::Result<Vec<u8>> {
  // You can choose your compression algorithm and it's
    efficiency
  let mut encoder = ZlibEncoder::new(Vec::new(),
    Compression::Default);
  encoder.write_all(bytes)?;
  encoder.finish()
}

fn decode_bytes(bytes: &[u8]) -> io::Result<Vec<u8>> {
  let mut encoder = ZlibDecoder::new(bytes);
  let mut buffer = Vec::new();
  encoder.read_to_end(&mut buffer)?;
  Ok(buffer)
}


fn encode_file(file: &mut Read) -> io::Result<Vec<u8>> {
  // Files have a built-in encoder
  let mut encoded = file.zlib_encode(Compression::Best);
  let mut buffer = Vec::new();
  encoded.read_to_end(&mut buffer)?;
  Ok(buffer)
}

fn decode_file(file: &mut Read) -> io::Result<Vec<u8>> {
  let mut buffer = Vec::new();
  // Files have a built-in decoder
  file.zlib_decode().read_to_end(&mut buffer)?;
  Ok(buffer)
}
```

How it works...

A lot of the `main` function work is just repetition for you from the last few chapters. The real deal happens below it. In `encode_bytes`, you can see how to use *encoders*. You can write to it as much as you want and call `finish` when you're done.

`flate2` gives you several compression options. You can choose your compression strength through the passed `Compression` instance:

```
let mut encoder = ZlibEncoder::new(Vec::new(), Compression::Default);
```

`Default` is a compromise between speed and size. Your other options are `Best`, `Fast`, and `None`. Additionally, you can specify the encoding algorithm used. `flate2` supports zlib, which we use in this recipe, gzip, and plain deflate. If you want to use an algorithm other than zlib, simply replace every mention of it with another supported algorithm. For instance, if you wanted to rewrite the preceding code to use gzip instead, it would look like this:

```
use flate2::write::GzEncoder;
let mut encoder = GzEncoder::new(Vec::new(), Compression::Default);
```

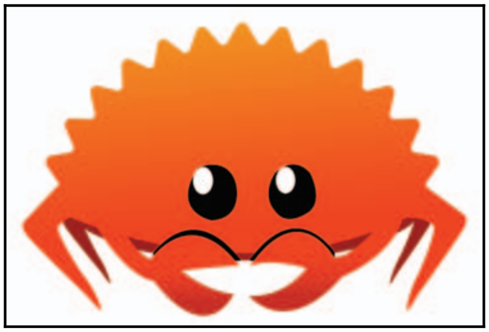
For a full list of how the specific encoders are called, visit `flate2`'s documentation at `https://docs.rs/flate2/`.

Because people would often prefer to compress or decompress whole files instead of byte buffers, there are some convenient methods for that. In fact, they are implemented on every type that implements `Read`, which means that you can also use them on a `BufReader` and many other types. `encode_file` and `decode_file` use them with `zlib` in the form of the following lines:

```
let mut encoded = file.zlib_encode(Compression::Best);
file.zlib_decode().read_to_end(&mut buffer)?;
```

The same applies to the `gzip` and `deflate` algorithms.

In our example, we are compressing and decompressing `ferris.png`, which is an image of Rust's mascot:



You can find it in the GitHub repository at `https://github.com/SirRade/rust-standard-library-cookbook` or you can use any other file you want. If you feel like verifying the compression, you can take a look at the original, compressed, and decompressed files to check how much smaller the compressed one is, and that the original and decompressed ones are identical.

There's more...

The current `encode_something` and `decode_something` functions are designed to be as simple to use as possible. However, they waste some performance by allocating `Vec<u8>` even though we could pipe the data directly into a writer. When writing a library, it would be nice to give the user both possibilities by adding methods in this way:

```
use std::io::Write;
fn encode_file_into(file: &mut Read, target: &mut Write) ->
io::Result<()> {
    // Files have a built-in encoder
    let mut encoded = file.zlib_encode(Compression::Best);
    io::copy(&mut encoded, target)?;
    Ok(())
}
```

The user could call them like this:

```
// Compress it
encode_file_into(&mut original_reader, &mut encoded_writer)
    .expect("Failed to encode file");
```

Traversing the filesystem

Up until now, we always provided our code with the static location of a certain file. Alas, the real world is seldom so predictable, and some digging is going to be necessary when dealing with data scattered throughout different folders.

`walkdir` helps us with this by abstracting away the intricacies and inconsistencies of operating systems' representation of the filesystem by unifying them under one common API, which we are going to learn about in this recipe.

Getting ready

This recipe makes heavy use of iterators to manipulate streams of data. If you are not yet familiar with them or need a quick refresher, you should read the *Access collections as Iterators* section in `Chapter 2`, *Working with Collections*, before continuing.

How to do it...

1. Open the `Cargo.toml` file that was generated earlier for you.
2. Under `[dependencies]`, add the following line:

```
walkdir = "2.0.1"
```

3. If you want, you can go to `walkdir`'s crates.io page (`https://crates.io/crates/walkdir`) to check for the newest version and use that one instead.
4. In the `bin` folder, create a file called `traverse_files.rs`.
5. Add the following code and run it with `cargo run --bin traverse_files`:

```
extern crate walkdir;
use walkdir::{DirEntry, WalkDir};

fn main() {
  println!("All file paths in this directory:");
  for entry in WalkDir::new(".") {
    if let Ok(entry) = entry {
      println!("{}", entry.path().display());
    }
  }
```

```

    println!("All non-hidden file names in this directory:");
    WalkDir::new("../chapter_three")
      .into_iter()
      .filter_entry(|entry| !is_hidden(entry)) // Look only at non-hidden enthries
      .filter_map(Result::ok) // Keep all entries we have access to
      .for_each(|entry| {
        // Convert the name returned by theOS into a Rust string
        // If there are any non-UTF8 symbols in it, replace them with placeholders
        let name = entry.file_name().to_string_lossy();
          println!("{}", name)
      });

      println!("Paths of all subdirectories in this directory:");
      WalkDir::new(".")
        .into_iter()
        .filter_entry(is_dir) // Look only at directories
        .filter_map(Result::ok) // Keep all entries we have access to
        .for_each(|entry| {
          let path = entry.path().display();
          println!("{}", path)
        });

      let are_any_readonly = WalkDir::new("..")
        .into_iter()
        .filter_map(Result::ok) // Keep all entries we have access to
        .filter(|e| has_file_name(e, "vector.rs")) // Get the ones with a certain name
        .filter_map(|e| e.metadata().ok()) // Get metadata if the OS allows it
        .any(|e| e.permissions().readonly()); // Check if at least one entry is readonly
      println!(
        "Are any the files called 'vector.rs' readonly? {}",
         are_any_readonly
      );

     let total_size = WalkDir::new(".")
       .into_iter()
       .filter_map(Result::ok) // Keep all entries we have access
```

```
          to
        .filter_map(|entry| entry.metadata().ok()) // Get
metadata
          if supported
        .filter(|metadata| metadata.is_file()) // Keep all files
        .fold(0, |acc, m| acc + m.len()); // Accumulate sizes

      println!("Size of current directory: {} bytes",
total_size);
    }
```

6. Now, come to the predicates used in this recipe:

```
    fn is_hidden(entry: &DirEntry) -> bool {
      entry
        .file_name()
        .to_str()
        .map(|s| s.starts_with('.'))
        .unwrap_or(false) // Return false if the filename is
         invalid UTF8
    }

    fn is_dir(entry: &DirEntry) -> bool {
      entry.file_type().is_dir()
    }

    fn has_file_name(entry: &DirEntry, name: &str) -> bool {
      // Check if file name contains valid unicode
      match entry.file_name().to_str() {
      Some(entry_name) => entry_name == name,
      None => false,
    }
  }
```

How it works...

`walkdir` consists of three important types:

- `WalkDir`: A builder (see the *Using the builder pattern* section in `Chapter 1`, *Learning the Basics*) for your directory walker
- `IntoIter`: The iterator created by the builder
- `DirEntry`: Represents a single folder or file

If you just want to operate on a list of all entries under a root folder, such as in the first example in line [6], you can implicitly use `WalkDir` directly as an iterator over different instances of `DirEntry`:

```
for entry in WalkDir::new(".") {
    if let Ok(entry) = entry {
        println!("{}", entry.path().display());
    }
}
```

As you can see, the iterator doesn't directly give you a `DirEntry`, but a `Result`. This is because there are some cases where accessing a file or folder might prove difficult. For instance, the OS could prohibit you from reading the contents of a folder, hiding the files in it. Or a symlink, which you could enable by calling `follow_links(true)` on the `WalkDir` instance, could point back to a parent directory, potentially resulting in an endless loop.

Our solution strategy for the errors in this recipe is simple—we just ignore them and carry on with the rest of the entries that didn't report any issues.

When you extract the actual entry, it can tell you a lot about itself. One of those things is its path. Keep in mind, though, that `.path()` [8] doesn't just return the path as a string. Actually, it returns a native Rust `Path` struct that could be used for further analysis. You could, for example, read a file path's extension by calling `.extension()` on it. Or you could get its parent directory by calling `.parent()`. Feel free to explore the possibilities by exploring the `Path` documentation at `https://doc.rust-lang.org/std/path/struct.Path.html`. In our case, we are only going to display it as a simple string by calling `.display()` on it.

When we explicitly convert `WalkDir` into an iterator with `into_iter()`, we can access a special method that no other iterator has: `filter_entry`. It is an optimization over `filter` in that it gets called during the traversal. When its predicate returns `false` on a directory, the walker won't go into the directory at all! This way, you can gain a lot of performance when traversing big filesystems. In the recipe, we use it while looking for non-hidden files [15]. If you need to operate only on files and never on directories, you should use plain old `filter` instead.

We define *hidden files*, by Unix convention, as all directories and files that start with a dot. For this reason, they are sometimes also called *dotfiles.*

In both cases, your filtering requires a predicate. They are usually put in their own function for simplicity and reusability.

Note that `walkdir` doesn't just give us the filename as a normal string. Instead, it returns an `OsStr`. This is a special kind of string that Rust uses when talking directly to the operating system. The type exists because some operating systems allow invalid UTF-8 in their filenames. When looking at such files in Rust, you have two choices—let Rust try to convert them into UTF-8 and replace all invalid characters with the Unicode Replacement Character (�), or instead handle the error yourself. You can go the first route by calling `to_string_lossy` on an `OsStr` [20]. The second route is accessible by calling `to_str` and checking the returned `Option`, like we did in `has_file_name`, where we simply discard invalid names.

In this recipe, you can see a splendid example of when to choose a `for_each` method call (discussed in the *Access collections as Iterators* section in `Chapter 1`, *Learning the Basics, Working with collections*) over a `for` loop—most of our iterator calls are chained together, and so a `for_each` call can naturally be chained into the iterator as well.

There's more...

If you plan on publishing your application on Unix and Unix only, you can access additional permissions on an entry over its `.metadata().unwrap().permissions()` call. Namely, you can see the exact `st_mode` bits by calling `.mode()` and change them by calling `set_mode()` with a new set of bits.

See also

- *Using the builder pattern* recipe in `Chapter 1`, *Learning the Basics*
- *Access collections as iterators* recipe in `Chapter 2`, *Working with Collections*

Finding files with glob patterns

As you may have noticed, using `walkdir` to filter files based on their name can be a bit clunky at times. Luckily, you can greatly simplify this by using the `glob` crate, which brings you its titular patterns, known from Unix, right into Rust.

How to do it...

Follow these steps:

1. Open the `Cargo.toml` file that was generated earlier for you.
2. Under `[dependencies]`, add the following line:

    ```
    glob = "0.2.11"
    ```

3. If you want, you can go to glob's crates.io page (`https://crates.io/crates/glob`) to check for the newest version and use that one instead.
4. In the `bin` folder, create a file called `glob.rs`.
5. Add the following code and run it with `cargo run --bin glob`:

```
extern crate glob;
use glob::{glob, glob_with, MatchOptions};

fn main() {
  println!("All all Rust files in all subdirectories:");
  for entry in glob("**/*.rs").expect("Failed to read glob
  pattern") {
    match entry {
      Ok(path) => println!("{:?}", path.display()),
      Err(e) => println!("Failed to read file: {:?}", e),
    }
  }

  // Set the glob to be case insensitive and ignore hidden
     files
  let options = MatchOptions {
    case_sensitive: false,
    require_literal_leading_dot: true,
    ..Default::default()
  };


  println!(
    "All files that contain the word \"ferris\" case
     insensitive \
     and don't contain an underscore:"
  );
  for entry in glob_with("*Ferris[!_]*",
  &options).expect("Failed to read glob pattern") {
```

```
    if let Ok(path) = entry {
      println!("{:?}", path.display())
    }
  }
}
```

How it works...

This crate is pretty small and simple. With `glob(...)`, you can create an iterator over all matching files by specifying a `glob` pattern. If you aren't familiar with them but remember the regex recipe from earlier (in the *Querying with regexes* section in `Chapter 1`, *Learning the Basics*), think of them as very simplified regexes used primarily for filenames. Its syntax is nicely described on Wikipedia: `https://en.wikipedia.org/wiki/Glob_(programming)`.

As with `WalkDir` before, the `glob` iterator returns a `Result` because the program might not have the permissions to read a filesystem entry. Inside the `Result` sits a `Path`, which we also touched on in the last recipe. If you want to read the contents of the file, refer to the first recipe in this chapter, which deals with file manipulation.

With `glob_with`, you can specify a `MatchOptions` instance to change the way `glob` searches for files. The most useful options you can toggle are:

- `case_sensitive`: This is enabled per default and controls whether lowercase letters (abcd) and uppercase letters (ABCD) should be treated differently or not.
- `require_literal_leading_dot`: This is disabled per default and, when set, prohibits wildcards from matching a leading dot in a filename. This is used when you want to ignore a user's hidden files.

You can view the rest of the options in the documentation of `MatchOption`: `https://doc.rust-lang.org/glob/glob/struct.MatchOptions.html`.

If you have set the options you care about, you can leave the rest at their default by using the `..Default::default()` *update syntax* discussed in the *Providing a default implementation* section in `Chapter 1`, *Learning the Basics*.

See also

- *Querying with Regexes* and *Providing a default implementation* recipes in `Chapter 1`, *Learning the Basics*

4
Serialization

In this chapter, we will cover the following recipes:

- Working with CSV
- Serialization basics with Serde
- Working with TOML
- Working with JSON
- Building JSON dynamically

Introduction

Reinventing the wheel doesn't make sense. A great deal of functionality has already been provided by many programs that are more than happy to interact with your program. Of course, this offer is worthless if you're not able to communicate with them.

In this chapter, we are going to look at the most important formats in the Rust ecosystem in order to enable you to comfortably speak with other services.

Working with CSV

A nice and simple way to store uncomplicated and small datasets is CSV. This format is also of interest if you're working with spreadsheet applications such as Microsoft Excel, as they have excellent support for importing and exporting various flavors of CSV.

Getting started

You probably already know what a CSV is, but a little refresher won't hurt.

The idea of the format is to take a table of values and write all rows down as *records*. Inside a record, every column item is written down and separated by a comma. That's where the format's name comes from—comma-separated values.

Let's do an example. In the following code, we are going to write a CSV comparing various planets in the solar system to our own. A radius, distance from the sun, and gravity of `1` means *exactly as on earth*. Written as a table, our values look like this:

name	radius	distance_from_sun	gravity
Mercury	0.38	0.47	0.38
Venus	0.95	0.73	0.9
Earth	1	1	1
Mars	0.53	1.67	0.38
Jupiter	11.21	5.46	2.53
Saturn	9.45	10.12	1.07
Uranus	4.01	20.11	0.89
Neptune	3.88	30.33	1.14

Take every row, separate the values by commas, put them each on a separate line, and you end up with the CSV file:

```
name,radius,distance_from_sun,gravity
Mercury,0.38,0.47,0.38
Venus,0.95,0.73,0.9
Earth,1,1,1
Mars,0.53,1.67,0.38
Jupiter,11.21,5.46,2.53
Saturn,9.45,10.12,1.07
Uranus,4.01,20.11,0.89
Neptune,3.88,30.33,1.14
```

As you can see, the heading (`planet,radius,distance_from_sun,gravity`) is simply written as the first record.

How to do it...

1. Create a Rust project to work on during this chapter with `cargo new chapter_four`.
2. Navigate into the newly-created `chapter_four` folder. For the rest of this chapter, we will assume that your command line is currently in this directory.
3. Inside the `src` folder, create a new folder called `bin`.
4. Delete the generated `lib.rs` file, as we are not creating a library.
5. Open the `Cargo.toml` file that was generated earlier for you.
6. Under `[dependencies]`, add the following line:

```
csv = "1.0.0-beta.5"
```

7. If you want, you can go to `csv`'s crates.io page (`https://crates.io/crates/csv`) to check for the newest version and use that one instead.
8. In the `src/bin` folder, create a file called `csv.rs`
9. Add the following code and run it with `cargo run --bin csv`:

```
extern crate csv;

use std::io::{BufReader, BufWriter, Read, Seek, SeekFrom,
Write};
use std::fs::OpenOptions;

fn main() {
  let file = OpenOptions::new()
    .read(true)
    .write(true)
    .create(true)
    .open("solar_system_compared_to_earth.csv")
    .expect("failed to create csv file");

  let buf_writer = BufWriter::new(&file);
  write_records(buf_writer).expect("Failed to write csv");

  let mut buf_reader = BufReader::new(&file);
  buf_reader
    .seek(SeekFrom::Start(0))
    .expect("Failed to jump to the beginning of the csv");
  read_records(buf_reader).expect("Failed to read csv");
}

```

```
fn write_records<W>(writer: W) -> csv::Result<()>
where
W: Write,
{
  let mut wtr = csv::Writer::from_writer(writer);

  // The header is just a normal record
  wtr.write_record(&["name", "radius", "distance_from_sun",
  "gravity"])?;

  wtr.write_record(&["Mercury", "0.38", "0.47", "0.38"])?;
  wtr.write_record(&["Venus", "0.95", "0.73", "0.9"])?;
  wtr.write_record(&["Earth", "1", "1", "1"])?;
  wtr.write_record(&["Mars", "0.53", "1.67", "0.38"])?;
  wtr.write_record(&["Jupiter", "11.21", "5.46", "2.53"])?;
  wtr.write_record(&["Saturn", "9.45", "10.12", "1.07"])?;
  wtr.write_record(&["Uranus", "4.01", "20.11", "0.89"])?;
  wtr.write_record(&["Neptune", "3.88", "30.33", "1.14"])?;
  wtr.flush()?;
  Ok(())
}

fn read_records<R>(reader: R) -> csv::Result<()>
where
R: Read,
{
  let mut rdr = csv::Reader::from_reader(reader);
  println!("Comparing planets in the solar system with the
  earth");
  println!("where a value of '1' means 'equal to earth'");
  for result in rdr.records() {
    println!("-------");
    let record = result?;
    if let Some(name) = record.get(0) {
      println!("Name: {}", name);
    }
    if let Some(radius) = record.get(1) {
      println!("Radius: {}", radius);
    }
    if let Some(distance) = record.get(2) {
      println!("Distance from sun: {}", distance);
    }
    if let Some(gravity) = record.get(3) {
      println!("Surface gravity: {}", gravity);
    }
  }
  Ok(())
}
```

How it works...

First of all, we prepare our file[9] and its `OpenOptions` so that we have both `read` and `write` access on the file. You will remember this from `Chapter 3`, *Handling Files and the Filesystem*; *Working with text files*.

Then we write the CSV. We do this by wrapping any kind of `Write` in a `csv::Writer`[30]. You can then use `write_record` on it to write any data type that can be represented as an iterator over `&[u8]`. Most of the time, this will simply be an array of strings.

While reading, we similarly wrap a `Read` in a `csv::Read`. The `records()` method returns an iterator over `Result` of `StringRecord`. This way, you get to decide how to handle a malformed record. In our example, we simply skip it. Lastly, we call `get()` on a record to get a certain field. If there was no entry at the specified index or if it was out of bounds, this would return `None`.

There's more...

If you need to read or write a custom CSV format, like one that uses tabs instead of commas as a delimiter, you can use `WriterBuilder` and `ReaderBuilder` to customize the expected format. Remember this well if you're planning on using Microsoft Excel, as it has the annoying tendency to be regionally inconsistent in its choice of delimiters (`https://stackoverflow.com/questions/10140999/csv-with-comma-or-semicolon`).

When working with CSV and Microsoft Excel, be careful and sanitize your data before handing it to Excel. Even though CSV is defined as plain data with no control identifiers, Excel will interpret and execute macros when importing CSV. For examples of possible attack vectors opened by this, see `http://georgemauer.net/2017/10/07/csv-injection.html`.

This is also useful if a Windows application refuses to accept the `\n` terminator that `csv` uses as per default. In this case, simply specify the following code in the builder to use the Windows-native `\r\n` terminator:

```
.terminator(csv::Terminator::CRLF)
```

The `csv` crate allows you to manipulate your data much more than what is shown in this recipe. You can, for example, insert new fields on the fly into a `StringRecord`. We deliberately don't explore these possibilities in detail as the CSV format is not meant for these kinds of data manipulation. If you need to do more than simple import/export, you should use a more suitable format, such as JSON, which we will explore in this chapter as well.

See also

- *Using the builder pattern* recipe in `Chapter 1`, *Learning the Basics*
- *Working with text files* recipe in `Chapter 3`, *Handling Files and the Filesystem*

Serialization basics with Serde

The *de facto* standard for all things serialization in Rust is the Serde framework. All the other recipes in this chapter are going to use it to some extent. In order to make you familiar with the Serde way of doing things, we are going to rewrite the last recipe using it. Later in the chapter, we will learn in detail how Serde works in order to implement idiomatic deserialization into a homemade format.

How to do it...

1. Open the `Cargo.toml` file that was generated earlier for you.
2. Under `[dependencies]`, add the following lines:

   ```
   serde = "1.0.24"
   serde_derive = "1.0.24"
   ```

3. If you haven't done so already in the last recipe, add the following line as well:

   ```
   csv = "1.0.0-beta.5"
   ```

4. If you want, you can go to the crates.io web pages for Serde (`https://crates.io/crates/serde`), `serde_derive` (`https://crates.io/crates/serde_derive`), and CSV (`https://crates.io/crates/csv`) to check for the newest versions and use those ones instead.
5. In the `bin` folder, create a file called `serde_csv.rs`.

4. Add the following code and run it with `cargo run --bin serde_csv`:

```
extern crate csv;
extern crate serde;
#[macro_use]
extern crate serde_derive;

use std::io::{BufReader, BufWriter, Read, Seek, SeekFrom,
Write};
use std::fs::OpenOptions;

#[derive(Serialize, Deserialize)]
struct Planet {
  name: String,
  radius: f32,
  distance_from_sun: f32,
  gravity: f32,
}

fn main() {
  let file = OpenOptions::new()
    .read(true)
    .write(true)
    .create(true)
    .open("solar_system_compared_to_earth.csv")
    .expect("failed to create csv file");

  let buf_writer = BufWriter::new(&file);
  write_records(buf_writer).expect("Failed to write csv");

  let mut buf_reader = BufReader::new(&file);
  buf_reader
    .seek(SeekFrom::Start(0))
    .expect("Failed to jump to the beginning of the csv");
  read_records(buf_reader).expect("Failed to read csv");
}

fn write_records<W>(writer: W) -> csv::Result<()>
where
W: Write,
{
  let mut wtr = csv::Writer::from_writer(writer);

  // No need to specify a header; Serde creates it for us
  wtr.serialize(Planet {
    name: "Mercury".to_string(),
    radius: 0.38,
     distance_from_sun: 0.47,
     gravity: 0.38,
   })?;
   wtr.serialize(Planet {
     name: "Venus".to_string(),
     radius: 0.95,
     distance_from_sun: 0.73,
     gravity: 0.9,
   })?;
   wtr.serialize(Planet {
     name: "Earth".to_string(),
     radius: 1.0,
     distance_from_sun: 1.0,
     gravity: 1.0,
   })?;
   wtr.serialize(Planet {
     name: "Mars".to_string(),
     radius: 0.53,
     distance_from_sun: 1.67,
     gravity: 0.38,
   })?;
   wtr.serialize(Planet {
     name: "Jupiter".to_string(),
     radius: 11.21,
     distance_from_sun: 5.46,
     gravity: 2.53,
   })?;
   wtr.serialize(Planet {
     name: "Saturn".to_string(),
     radius: 9.45,
     distance_from_sun: 10.12,
     gravity: 1.07,
  })?;
  wtr.serialize(Planet {
    name: "Uranus".to_string(),
    radius: 4.01,
    distance_from_sun: 20.11,
    gravity: 0.89,
  })?;
  wtr.serialize(Planet {
    name: "Neptune".to_string(),
    radius: 3.88,
    distance_from_sun: 30.33,
    gravity: 1.14,
  })?;
  wtr.flush()?;
  Ok(())
}
```

```
93
94  fn read_records<R>(reader: R) -> csv::Result<()>
95  where
96  R: Read,
97  {
98    let mut rdr = csv::Reader::from_reader(reader);
99    println!("Comparing planets in the solar system with the
   earth");
100   println!("where a value of '1' means 'equal to earth'");
101   for result in rdr.deserialize() {
102     println!("-------");
103     let planet: Planet = result?;
104     println!("Name: {}", planet.name);
105     println!("Radius: {}", planet.radius);
106     println!("Distance from sun: {}",
planet.distance_from_sun);
107     println!("Surface gravity: {}", planet.gravity);
108   }
109   Ok(())
110 }
```

How it works...

The code in this recipe reads and writes the exact same CSV as the last recipe. The only difference is how we treat a single record. Serde helps us by enabling us to use plain old Rust structures for this. The only thing we need to do is derive our `Planet` structure from `Serialize` and `Deserialize` [9]. The rest is taken care of automatically by Serde.

Because we now use actual Rust structures to represent a planet, we create a record by calling `serialize` with a structure instead of `write_record` as before [42]. Looks way more readable, doesn't it? If you think that the example became a little bit too verbose, you could hide the actual object creation behind a constructor, as described in `Chapter 1`, *Learning the Basics*; *Using the Constructor Pattern*.

When reading a CSV, we also no longer have to manually access the fields of a `StringRecord`. Instead, `deserialize()` returns an iterator over a `Result` of an already deserialized `Planet` object. Again, look how much more readable this has become.

As you may have already guessed, you should use Serde whenever you can, as it helps you catch possible errors early by providing you with readability and compile-time type safety.

There's more...

Serde gives you the ability to tweak the serialization process somewhat by annotating your fields. For example, you can give a field a standard value if it wasn't able to be parsed by writing `#[serde(default)]` above its declaration. In a struct, it would look like this:

```
#[derive(Serialize, Deserialize)]
struct Foo {
    bar: i32,
    #[serde(default)]
    baz: i32,
}
```

If `baz` hasn't been parsed, its `Default::default` value (See `Chapter 1`, *Learning the Basics*; Providing a default implementation) will be used. Another useful thing you can do with annotations is changed the expected case convention. By default, Serde will expect Rust's `snake_case`, however, you can change this by annotating a `struct` or `enum` with `#[serde(rename_all = "PascalCase")]`. You can use it on a struct like this:

```
#[derive(Serialize, Deserialize)]
#[serde(rename_all = "PascalCase")]
struct Stats {
    number_of_clicks: i32,
    total_time_played: i32,
}
```

This would, instead of parsing `number_of_clicks` and `total_time_played`, expect the `NumberOfClicks` and `TotalTimePlayed` keys to be called. Other possible case conventions than `PascalCase` that Serde supports are lowercase, *camelCase*, and *SCREAMING_SNAKE_CASE*.

There are many different and useful attributes. If you want to, you can familiarize yourself with them at `https://serde.rs/attributes.html`.

You can use Serde to provide idiomatic serialization and deserialization, however, discussing all best practices would cover an entire chapter on its own. If you want to delve into such things, Serde has a nice write-up on how to do it at `https://serde.rs/data-format.html`.

See also

- *Using the constructor pattern* and *Providing a default implementation* recipes in `Chapter 1`, *Learning the Basics*
- *Working with text files* recipe in `Chapter 3`, *Handling Files and the Filesystem*

Working with TOML

Do you like the simplicity of INI files but wish they were formally specified and had a few more features? So did Tom Preston-Werner, founder of services such as GitHub and Gravatar. He created Tom's Obvious, Minimal Language, or TOML for short. This relatively new format is seeing increasing adoption in new projects. In fact, you have used it multiple times by now as well: Cargo's dependencies are specified in every project's `Cargo.toml` file!

Getting started

At its heart, TOML is all about *key-value* pairs. This is the simplest TOML file you can create:

```
message = "Hello World"
```

Here, the key message has the `"Hello World"` value. A value can also be an array:

```
messages: ["Hello", "World", "out", "there"]
```

A group of key-values is called a *table*. The following TOML lets the `smileys` table contain the `happy` key with the `":)"` value and the `sad` key with the `":("` value:

```
[smileys]
happy = ":)"
sad = ":("
```

A particularly small table can be *inlined*, that is, written in one line. The last example is the exact same as the following:

```
smileys = { happy = ":)", sad = ":(" }
```

Tables can be nested by separating their names with a dot:

```
[servers]
  [servers.production]
  ip = "192.168.0.1"
  [servers.beta]
  ip = "192.169.0.2"
  [servers.testing]
  ip = "192.169.0.3"
```

A nice property of TOML is that you can convert any key into a table if you need to specify additional information. For example, Cargo itself expects this when declaring dependency versions. For example, if you wanted to use `rocket_contrib`, a helper crate of the popular `rocket` web framework for Rust, at version 0.3.3, you would write this:

```
[dependencies]
rocket_contrib = 0.3.3
```

However, if you wanted to specify the exact features to be included in `rocket_contrib`, you would need to instead write it as a sub-table of `dependencies`. The following TOML would tell `Cargo` to use its JSON serialization feature:

```
[dependencies]
[dependencies.rocket_contrib]
version = "0.3.3"
default-features = false
features = ["json"]
```

Another nice thing TOML brings to the table is that its whitespace is not significant, that is, you can indent a file however you want. You can even add comments by beginning a line with the following:

```
# some comment
```

If you want to explore the format further, the entirety of the TOML syntax is specified at `https://github.com/toml-lang/toml`.

How to do it...

1. Open the `Cargo.toml` file that was generated earlier for you

2. Under `[dependencies]`, add the following line:

```
toml = "0.4.5"
```

3. If you haven't done so already, add the following lines as well:

```
serde = "1.0.24"
serde_derive = "1.0.24"
```

4. If you want, you can go to the crates.io web pages for TOML (`https://crates.io/crates/toml`), Serde (`https://crates.io/crates/serde`), and `serde_derive` (`https://crates.io/crates/serde_derive`) to check for the newest versions and use those ones instead
5. In the `bin` folder, create a file called `toml.rs`
6. Add the following code and run it with `cargo run --bin toml`:

```
#[macro_use]
extern crate serde_derive;
extern crate toml;

use std::io::{BufReader, BufWriter, Read, Seek, SeekFrom,
Write};
use std::fs::OpenOptions;
```

7. These are the structures we are going to use throughout the recipe:

```
#[derive(Serialize, Deserialize)]
struct Preferences {
  person: Person,
  language: Language,
  privacy: Privacy,
}

#[derive(Serialize, Deserialize)]
struct Person {
  name: String,
  email: String,
}

#[derive(Serialize, Deserialize)]
struct Language {
  display: String,
  autocorrect: Option<Vec<String>>,
}

```

```
#[derive(Serialize, Deserialize)]
struct Privacy {
  share_anonymous_statistics: bool,
  public_name: bool,
  public_email: bool,
}
```

8. Prepare a new file and call the other functions:

```
fn main() {
  let file = OpenOptions::new()
   .read(true)
   .write(true)
   .create(true)
   .open("preferences.toml")
   .expect("failed to create TOML file");

  let buf_writer = BufWriter::new(&file);
  write_toml(buf_writer).expect("Failed to write TOML");

  let mut buf_reader = BufReader::new(&file);
  buf_reader
    .seek(SeekFrom::Start(0))
    .expect("Failed to jump to the beginning of the TOML
    file");
  read_toml(buf_reader).expect("Failed to read TOML");
}
```

9. Save our structures as a TOML file:

```
type SerializeResult<T> = Result<T, toml::ser::Error>;
fn write_toml<W>(mut writer: W) -> SerializeResult<()>
where
W: Write,
{
  let preferences = Preferences {
    person: Person {
      name: "Jan Nils Ferner".to_string(),
      email: "jn_ferner@hotmail.de".to_string(),
    },
    language: Language {
      display: "en-GB".to_string(),
      autocorrect: Some(vec![
        "en-GB".to_string(),
        "en-US".to_string(),
        "de-CH".to_string(),
      ]),
    },
    privacy: Privacy {
      share_anonymous_statistics: false,
      public_name: true,
      public_email: true,
    },
  };

  let toml = toml::to_string(&preferences)?;
  writer
    .write_all(toml.as_bytes())
    .expect("Failed to write file");
  Ok(())
}
```

10. `Read` the TOML file we just created:

```
type DeserializeResult<T> = Result<T, toml::de::Error>;
fn read_toml<R>(mut reader: R) -> DeserializeResult<()>
where
R: Read,
{
  let mut toml = String::new();
  reader
    .read_to_string(&mut toml)
    .expect("Failed to read TOML");
  let preferences: Preferences = toml::from_str(&toml)?;

  println!("Personal data:");
  let person = &preferences.person;
  println!(" Name: {}", person.name);
  println!(" Email: {}", person.email);

  println!("\nLanguage preferences:");
  let language = &preferences.language;
  println!(" Display language: {}", language.display);
  println!(" Autocorrect priority: {:?}",
  language.autocorrect);


  println!("\nPrivacy settings:");
  let privacy = &preferences.privacy;
  println!(
    " Share anonymous usage statistics: {}",
      privacy.share_anonymous_statistics
  );
  println!(" Display name publically: {}",
  privacy.public_name);
  println!(" Display email publically: {}",
            privacy.public_email);
114
115     Ok(())
116 }
```

How it works...

As always with Serde, we first need to declare the structures we plan on using [8 to 32].

While serializing, we can directly call Serde's `to_string` method with a structure as TOML re-exports them [77]. This returns a `String` that we can then write into a file [79]. The same is true of Serde's `from_str`, which, when type-annotated, takes a `&str` and converts it into a structure.

There's more...

You might have noticed that we are not using the try-operator (`?`) while reading or writing in this recipe. This is because the function's expected error types, `se::Error` [77] and `de::Error`[93], are incompatible with `std::io::Error`. In `Chapter 6`, *Handling Errors*; *Providing user-defined Error types*, we will explore how to avoid this by returning our own error type that encompasses the other ones mentioned.

The TOML crate used in this recipe is the same one that Cargo itself uses. If you are interested in how Cargo parses its own `Cargo.toml` file, you can check out `https://github.com/rust-lang/cargo/blob/master/src/cargo/util/toml/mod.rs`.

See also

- *Working with* text *files* recipe in `Chapter 3`, *Handling Files and the Filesystem*
- *Providing user-defined error types* recipe in `Chapter 6`, *Handling Errors*

Working with JSON

Most Web APIs and many native APIs speak JSON nowadays. It should be your format of choice when designing data meant for consumption by other programs, as it is lightweight, simple, easy to use and understand, and has excellent library support across programming languages, most notably JavaScript, of course.

Getting ready

JSON was created at a time when most web communication was done by sending XML over browser plugins such as Java or Flash. This was cumbersome and made the exchanged information quite bloated. Douglas Crockford, the creator of JSLint and author of the famous *JavaScript: The Good Parts*, decided in the early 2000s that it was time for a lightweight format that was easily integrated with JavaScript. He oriented himself on a small subset of JavaScript, namely the way it defined objects, and extended it a little bit to form the JavaScript Object Notation or JSON. Yes, you've read that right; JSON is *not* a subset of JavaScript, as it accepts things that JavaScript doesn't. You can read more about that at `http://timelessrepo.com/json-isnt-a-javascript-subset.`

The sad irony of the story is that today we have gone full-circle: our best practices for web development include a labyrinth of task runners, frameworks, and transpilers, which are all quite nice in theory but end up being a giant bloated mess in the end. But that is a story for another time.

JSON is built upon two structures:

- A group of key-value pairs surrounded by `{` and `}`, which is called an *object* and can be a value itself
- A list of values surrounded by `[` and `]` called an *array*

This might remind you a bit of the TOML syntax we discussed earlier, and you might ask yourself when you should prefer one over the other. The answer is that JSON is a good format when your data is going to be read automatically by a tool, while TOML is excellent when your data is meant to be read and modified manually by a human.

Take note that one thing that JSON *doesn't* allow is comments. This makes sense, as comments are not readable by tools anyway.

An example of a JSON object with values, sub-objects, and arrays could be the following:

```
{
  name: "John",
  age: 23,
  pets: [
    {
      name: "Sparky",
      species: "Dog",
      age: 2
    },
    {
      name: "Speedy",
      species: "Turtle",
      age: 47,
      colour: "Green"
    },
    {
      name: "Meows",
      species: "Cat",
      colour: "Orange"
    }
  ]
}
```

We are going to write and read this exact example in the following code. The definitions of the pets are intentionally inconsistent because many web APIs omit certain keys in some situations.

How to do it...

1. Open the `Cargo.toml` file that has been generated earlier for you
2. Under `[dependencies]`, add the following line:

   ```
   serde_json = "1.0.8"
   ```

3. If you haven't done so already, add the following lines as well:

   ```
   serde = "1.0.24"
   serde_derive = "1.0.24"
   ```

4. If you want, you can go to the crates.io web pages for `serde_json` (`https://crates.io/crates/serde_json`), Serde (`https://crates.io/crates/serde`), and `serde_derive` (`https://crates.io/crates/serde_derive`) to check for the newest versions and use those ones instead
5. In the `bin` folder, create a file called `json.rs`
6. Add the following code and run it with `cargo run --bin json`:

```
extern crate serde;
extern crate serde_json;

#[macro_use]
extern crate serde_derive;

use std::io::{BufReader, BufWriter, Read, Seek, SeekFrom,
Write};
use std::fs::OpenOptions;
```

7. These are the structures we are going to use throughout the recipe:

```
#[derive(Serialize, Deserialize)]
struct PetOwner {
  name: String,
  age: u8,
  pets: Vec<Pet>,
}

#[derive(Serialize, Deserialize)]
struct Pet {
  name: String,
  species: AllowedSpecies,
  // It is usual for many JSON keys to be optional
  age: Option<u8>,
  colour: Option<String>,
}

#[derive(Debug, Serialize, Deserialize)]
enum AllowedSpecies {
  Dog,
  Turtle,
  Cat,
}
```

8. Prepare a new file and call the other functions:

```
fn main() {
  let file = OpenOptions::new()
    .read(true)
    .write(true)
    .create(true)
    .open("pet_owner.json")
    .expect("failed to create JSON file");

  let buf_writer = BufWriter::new(&file);
  write_json(buf_writer).expect("Failed to write JSON");

  let mut buf_reader = BufReader::new(&file);
  buf_reader
    .seek(SeekFrom::Start(0))
    .expect("Failed to jump to the beginning of the JSON
     file");
  read_json(buf_reader).expect("Failed to read JSON");
}
```

9. Save our structures as a JSON file:

```
fn write_json<W>(mut writer: W) -> serde_json::Result<()>
where
W: Write,
{
  let pet_owner = PetOwner {
    name: "John".to_string(),
    age: 23,
    pets: vec![
      Pet {
        name: "Waldo".to_string(),
        species: AllowedSpecies::Dog,
        age: Some(2),
        colour: None,
      },
      Pet {
        name: "Speedy".to_string(),
        species: AllowedSpecies::Turtle,
        age: Some(47),
        colour: Some("Green".to_string()),
      },
      Pet {
        name: "Meows".to_string(),
        species: AllowedSpecies::Cat,
        age: None,
        colour: Some("Orange".to_string()),
        },
      ],
    };

    let json = serde_json::to_string(&pet_owner)?;
    writer
     .write_all(json.as_bytes())
     .expect("Failed to write file");
    Ok(())
  }
```

10. Read the JSON file we just created:

```
  fn read_json<R>(mut reader: R) -> serde_json::Result<()>
  where
  R: Read,
  {
    let mut json = String::new();
    reader
      .read_to_string(&mut json)
      .expect("Failed to read TOML");
    let pet_owner: PetOwner = serde_json::from_str(&json)?;

    println!("Pet owner profile:");
    println!(" Name: {}", pet_owner.name);
    println!(" Age: {}", pet_owner.age);

    println!("\nPets:");
    for pet in pet_owner.pets {
      println!(" Name: {}", pet.name);
      println!(" Species: {:?}", pet.species);
      if let Some(age) = pet.age {
        println!(" Age: {}", age);
      }
      if let Some(colour) = pet.colour {
        println!(" Colour: {}", colour);
      }
      println!();
    }
    Ok(())
  }
```

How it works...

Notice how this recipe looks nearly identical to the last one? Except for the structures, the only significant difference is that we called `serde_json::to_string()` [81] instead of `toml::to_string()`, and `serde_json::from_str()` [96] instead of `toml::from_str()`. This is the beauty of a well-thought-out framework like Serde: the custom serialization and deserialization code are hidden behind trait definitions and we can use the same API without caring about internal implementation details.

Other than that, there is nothing to say that hasn't been said in the previous recipe, which is why we are not going to go over any other formats. All important formats support Serde, so you can use them the exact same way you use the other formats in this chapter. For a full list of all supported formats, see `https://docs.serde.rs/serde/index.html`.

There's more...

JSON has no concept of something resembling an `enum`. As many languages *do* work with them, however, multiple conventions on how to handle the conversion from JSON into `enum` have emerged over the years. Serde allows you to support these conventions with annotations on your enums. For a full list of supported conversions, visit `https://serde.rs/enum-representations.html`.

See also

- *Working with text files* recipe in `Chapter 3`, *Handling Files and the Filesystem*
- *Providing user-defined error types* recipe in `Chapter 6`, *Handling Errors*

Building JSON dynamically

When a JSON API is designed with a poorly-thought-out schema and inconsistent objects, you might end up with giant structures where most members are an `Option`. If you find yourself only sending data to such a service, it might be a bit easier to dynamically build your JSON property by property.

How to do it...

1. Open the `Cargo.toml` file that was generated earlier for you
2. Under `[dependencies]`, if you haven't done so already, add the following line:

   ```
   serde_json = "1.0.8"
   ```

3. If you want, you can go to the `serde_json` crates.io web page (`https://crates.io/crates/serde_json`) to check for the newest version and use that one instead
4. In the `bin` folder, create a file called `dynamic_json.rs`
5. Add the following code and run it with `cargo run --bin dynamic_json`:

```
#[macro_use]
extern crate serde_json;

use std::io::{self, BufRead};
use std::collections::HashMap;

fn main() {
  // A HashMap is the same as a JSON without any schema
  let mut key_value_map = HashMap::new();
  let stdin = io::stdin();
  println!("Enter a key and a value");
  for input in stdin.lock().lines() {
    let input = input.expect("Failed to read line");
    let key_value: Vec<_> =
input.split_whitespace().collect();
    let key = key_value[0].to_string();
    let value = key_value[1].to_string();

    println!("Saving key-value pair: {} -> {}", key, value);
    // The json! macro lets us convert a value into its JSON
       representation
    key_value_map.insert(key, json!(value));
    println!(
      "Enter another pair or stop by pressing '{}'",
       END_OF_TRANSMISSION
    );
  }
  // to_string_pretty returns a JSON with nicely readable
     whitespace
  let json =
  serde_json::to_string_pretty(&key_value_map).expect("Failed
```

```
          to convert HashMap into JSON");
29        println!("Your input has been made into the following
          JSON:");
30        println!("{}", json);
31    }
32
33    #[cfg(target_os = "windows")]
34    const END_OF_TRANSMISSION: &str = "Ctrl Z";
35
36    #[cfg(not(target_os = "windows"))]
37    const END_OF_TRANSMISSION: &str = "Ctrl D";
```

How it works...

In this example, the user can enter any number of key-value pairs until they decide to stop, at which point they receive their input back in the form of JSON. Some example input you could enter could include:

```
name abraham
age 49
fav_colour red
hello world
(press 'Ctrl Z' on Windows or 'Ctrl D' on Unix)
```

TIP Use `#[cfg(target_os = "some_operating_system")]` to handle operating system specific circumstances. In this recipe, we use this to conditionally compile the `END_OF_TRANSMISSION` constant differently on Windows than on Unix. This key combination tells the OS to stop the current input stream.

This program begins with the idea that a JSON object without a clearly defined schema is nothing but a `HashMap<String, String>`[9]. Now, `serde_json` doesn't accept a `String` as a value, as that would not be general enough. Instead, it wants a `serde_json::Value`, which you can easily construct by calling the `json!` macro on pretty much any type [20].

When we are done, we don't call `serde_json::to_string()` as before, but use `serde_json::to_string_pretty()` instead [28], as this results in a less efficient but much more readable JSON. Remember, JSON is not supposed to be primarily read by humans, which is why the default way Serde serialized it is without any whitespace whatsoever. If you're curious about the exact difference, feel free to go ahead and change `to_string_pretty()` to `to_string()` and compare the results.

See also

- *Reading from stdin* recipe in `Chapter 1`, *Learning the Basics*
- *Access collections as Iterators* and *Using a HashMap* recipe in `Chapter 2`, *Working with Collections*

5
Advanced Data Structures

In this chapter, we will cover the following recipes:

- Creating lazy static objects
- Working with bit fields
- Providing custom derives
- Converting types into each other
- Boxing data
- Sharing ownership with smart pointers
- Working with interior mutability

Introduction

So far, we have primarily looked at techniques that were all useful in their own right. Of course, we are continuing that trend, but the recipes shown in this chapter truly shine when combined with other code. You can imagine them as the *glue* holding a nice Rust program together, as they are primarily oriented toward new enabling ways in which different components of your crate can work together.

Creating lazy static objects

Big objects, especially constant ones, should be reused instead of rebuilt. The `lazy_static!` macro helps you with this by extending Rust's normal `static` functionality, which normally requires your objects to be constructable at compile-time, with the ability to create lazy objects that are initialized during runtime.

How to do it...

1. Create a Rust project to work on during this chapter with `cargo new chapter_five`.
2. Navigate into the newly created `chapter_five` folder. For the rest of this chapter, we will assume that your command line is currently in this directory.
3. Inside the `src` folder, create a new folder called `bin`.
4. Delete the generated `lib.rs` file, as we are not creating a library.
5. Open the `Cargo.toml` file that was generated earlier for you.
6. Under `[dependencies]`, add the following lines:

   ```
   lazy_static = "1.0"
   regex = "0.2"
   ```

   If you want, you can go to the crates.io web pages for `lazy_static` (`https://crates.io/crates/lazy_static`) and regex (`https://crates.io/crates/regex`) to check for the newest version and use that one instead.

7. In the `src/bin` folder, create a file called `lazy_static.rs`
8. Add the following code and run it with `cargo run --bin lazy_static`:

   ```
   #[macro_use]
   extern crate lazy_static;
   extern crate regex;

   use regex::Regex;
   use std::collections::HashMap;
   use std::sync::RwLock;

   // Global immutable static
   lazy_static! {
     static ref CURRENCIES: HashMap<&'static str, &'static str> =
       {
   ```

```
    let mut m = HashMap::new();
    m.insert("EUR", "Euro");
    m.insert("USD", "U.S. Dollar");
    m.insert("CHF", "Swiss Francs");
    m
  };
}

// Global mutable static
lazy_static! {
  static ref CLIENTS: RwLock<Vec<String>> =
  RwLock::new(Vec::new());
}

// Local static
fn extract_day(date: &str) -> Option<&str> {
  // lazy static objects are perfect for
  // compiling regexes only once
  lazy_static! {
    static ref RE: Regex =
    Regex::new(r"(\d{2}).(\d{2}).(\d{4})")
      .expect("Failed to create regex");
  }
  RE.captures(date)
    .and_then(|cap| cap.get(1).map(|day| day.as_str()))
}

fn main() {
  // The first access to CURRENCIES initializes it
  let usd = CURRENCIES.get("USD");
  if let Some(usd) = usd {
    println!("USD stands for {}", usd);
  }

  // All accesses will now refer to the same,
  // already constructed object
  if let Some(chf) = CURRENCIES.get("CHF") {
    println!("CHF stands for {}", chf);
  }

  // Mutable the global static
  CLIENTS
    .write()
    .expect("Failed to unlock clients for writing")
    .push("192.168.0.1".to_string());

  // Get an immutable reference to the global static
  let clients = CLIENTS
```

```
59        .read()
60        .expect("Failed to unlock clients for reading");
61      let first_client = clients.get(0).expect("CLIENTS is
        empty");
62      println!("The first client is: {}", first_client);
63
64      let date = "12.01.2018";
65      // The static object is nicely hidden inside
66      // the definition of extract_day()
67      if let Some(day) = extract_day(date) {
68        println!("The date \"{}\" contains the day \"{}\"", date,
          day);
69      }
70    }
```

How it works...

By invoking the `lazy_static!` macro [10, 21 and 29], we define a lazily initialized object in the current scope. *Lazy* here means *created only the first time it is used*.

Contrary to a `let` binding, its scope can also be the global scope [10]. A realistic example for this is creating a collection with a known content that is used by many functions, as the alternative would be to create it once and pass it around endlessly.

If your `lazy_static` consists of a `Vec` with content that is known at compile time, you can instead use a `const` array, as its construction is constant. In terms of code, this means you don't need to use this:

```
lazy_static!{
    static ref FOO: Vec<&'static str> = vec!["a", "b", "c"];
}
```

Instead, you can use the following:

```
const FOO: [&str; 3] = ["a", "b", "c"];
```

Remember when, in `Chapter 1`, *Learning the Basics*; *Querying with Regexes*, we talked about how compiling Regexes is expensive and should be avoided? `lazy_static!` is ideal for this. In fact, the pattern of creating a local `static` regex in a function is so widespread that we included it in this example [29]:

```
fn extract_day(date: &str) -> Option<&str> {
    lazy_static! {
        static ref RE: Regex =
            Regex::new(r"(\d{2}).(\d{2}).(\d{4})")
            .expect("Failed to create regex");
    }
    RE.captures(date)
        .and_then(|cap| cap.get(1).map(|day| day.as_str()))
}
```

Lastly, you can also create a global mutable state with `lazy_static` objects [21]. As mentioned in earlier chapters, the excessive state is the root of many evils in software development and should be treated with care. There are very few cases where having such an object is justifiable, as it is almost always better to pass the object around. There are exceptions, however. Sometimes a program revolves around the manipulation of one specific dataset in memory and all involved actors want to access it. In these cases, it can be very cumbersome to pass an object to literally every function in your code. One possible, but still very rare, an example where this might happen, is when dealing exclusively with a list of active connections [21]:

```
lazy_static! {
    static ref CLIENTS: RwLock<Vec<String>> = RwLock::new(Vec::new());
}
```

Note that, as the borrow checker is disabled for objects with `'static` lifetimes (see the following *There's more...* section), we need to wrap our `static` in a parallel lock, such as `RwLock` or `Mutex`, to guarantee thread safety. You can read more about that in `Chapter 7`, *Parallelism and Rayon*; *Access resources in parallel with RwLocks*.

There's more...

People coming from other languages might wonder what `lazy_static` offers that can't already be done by a normal `static` object. The difference between them is as follows.

In Rust, a `static` variable is a variable that lives for the entire duration of the program, which is why they get their own, special lifetime, `'static`. The catch is that the variable has to be built in a constant way, that is, a way that is known at compile time. In our example, we cannot replace `CURRENCIES` [11] with a normal `static` because `HashMap::new()` returns a newly constructed `HashMap` sitting somewhere in the memory during runtime. As this requires it to live in memory, it's impossible to build a `HashMap` during compile time, so its constructor is not `constant`.

Another catch with `static` variables is that, because they have a global lifetime, the borrow checker cannot make sure that their access is thread-safe. As a consequence, any access on a `static mut` variable will always be `unsafe`.

The convention for `static` variables is to write them in `ALL_CAPS`, just like `const` variables. This is because they are very closely linked. In fact, a `const` is nothing but an inlined `static` that can never be `mut`.

`lazy_static` gets around these restrictions by wrapping your object in a newly created `struct` that can be implicitly dereferenced into your object. This means that you never actually access your object directly. `lazy_static` stresses this by demanding that you write `ref` during the declaration of the `static`, as this makes you mentally treat the variable as a reference rather than an actual object:

```
lazy_static! {
    static ref foo: Foo = Foo::new();
}
```

While dereferencing, the wrapper `struct` works with a `static mut` pointer in your dynamically created object. All it does then is wrap the `unsafe` calls in a safe way.

If you come from a modern C++ background, you can view a normal `static` Rust as a `static constexpr` C++ and a `lazy_static` Rust as a `static` C++ local.

See also

- *Querying with Regexes* recipe in `Chapter 1`, *Learning the Basics*
- *Access resources in parallel with RwLocks* recipe in `Chapter 7`, *Parallelism and Rayon*

Working with bit fields

Programs written in C don't have the possibility to use the Builder Pattern (`Chapter 1`, *Learning the Basics*; *Using the builder pattern*) to provide users with combinable options. Instead, they have to rely on bit fields. As C has historically become the *lingua franca* of system languages, you will have to interact with a lot of C code if you plan on wrapping existing programs in a Rust interface or vice versa. Because of this, you will sooner or later come in contact with bit fields. As Rust's `enum` is way more complex than a C `enum`, you have to instead rely on the `bitflags` crate to provide you with all of the necessary functionality to comfortably handle bit fields.

Getting started

This chapter assumes that you know what a bit field is. Explaining it here doesn't make sense, as it would also incorporate an explanation of binary arithmetic and is not that relevant in your day-to-day Rust experience. For a good introduction to bit fields, check out this forum post and tutorial: `http://forum.codecall.net/topic/56591-bit-fields-flags-tutorial-with-example/` and `https://www.tutorialspoint.com/cprogramming/c_bit_fields.htm`, respectively.

How to do it...

1. Open the `Cargo.toml` file that was generated earlier for you.
2. Under `[dependencies]`, add the following lines:

```
bitflags = "1.0"
```

3. If you want, you can go to bitflags' crates.io page (`https://crates.io/crates/bitflags`) to check for the newest version and use that one instead.
4. In the `bin` folder, create a file called `bit_fields.rs`.
5. Add the following code and run it with `cargo run --bin bit_fields`:

```
#[macro_use]
extern crate bitflags;

bitflags! {
  struct Spices: u32 {
    const SALT    = 0b0000_0001;
    const PEPPER  = 0b0000_0010;
    const CHILI   = 0b0000_0100;
    const SAFFRON = 0b0000_1000;
    const ALL     = Self::SALT.bits
                  | Self::PEPPER.bits
                  | Self::CHILI.bits
                  | Self::SAFFRON.bits;
  }
}

impl Spices {
  // Implementing a "clear" method can be useful
  pub fn clear(&mut self) -> &mut Self {
    self.bits = 0;
    self
  }
}

fn main() {
  let classic = Spices::SALT | Spices::PEPPER;
  let spicy = Spices::PEPPER | Spices::CHILI;
  // Bit fields can nicely be printed
  println!("Classic: {:?}", classic);
  println!("Bits: {:08b}", classic.bits());
  println!("Spicy: {:?}", spicy);
  println!("Bits: {:08b}", spicy.bits());

  println!();

  // Use set operations
  println!("Union: {:?}", classic | spicy);
  println!("Intersection: {:?}", classic & spicy);
  println!("Difference: {:?}", classic - spicy);
  println!("Complement: {:?}", !classic);

  // Interact with flags in a bit field
  let mut custom = classic | spicy;
  println!("Custom spice mix: {:?}", custom);
  custom.insert(Spices::SAFFRON);
  // Note that ALL is now also contained in the bit field
  println!("Custom spice after adding saffron: {:?}", custom);
  custom.toggle(Spices::CHILI);
  println!("Custom spice after toggling chili: {:?}", custom);
  custom.remove(Spices::SALT);
  println!("Custom spice after removing salt: {:?}", custom);

  // This could be user input
```

```
54      let wants_salt = true;
55      custom.set(Spices::SALT, wants_salt);
56      if custom.contains(Spices::SALT) {
57        println!("I hope I didn't put too much salt in it");
58      }
59
60      // Read flags from raw bits
61      let bits = 0b0000_1101;
62      if let Some(from_bits) = Spices::from_bits(bits) {
63        println!("The bits {:08b} represent the flags {:?}", bits,
          from_bits);
64      }
65
66      custom.clear();
67      println!("Custom spice mix after clearing: {:?}", custom);
68    }
```

How it works...

The `bitflags!` macro lets you define all of your flags and their underlying type (in our case, this is `u32`)[4 to 15]. They are written in `ALL_CAPS` because they are constants. We can also define collections of flags this way, as we did with `ALL`[10]. We could have added additional combinations, for example:

```
const SPICY = Self::PEPPER.bits | Self::CHILI.bits;
```

The macro then creates a struct with the specified members for you and implements a bunch of traits for it in order to enable the familiar `|`, `&`, `-`, and `!` notations [37 to 40] and pretty printing. You can still access the raw `bits` used in the background directly over the member of the same name.

Note that, when printing, flag combinations will be listed separately. For instance, look at the output in line [47]. After setting all possible flags in the field to active, it will pretty print itself as the following:

```
Custom spice after adding saffron: SALT | PEPPER | CHILI | SAFFRON |
ALL
```

A useful method to additionally define on a bit field is `clear()`[19]. This hides the underlying `bits` from the user and is nicely readable.

With the aforementioned binary operators, you can perform *set operations* on your bit field [37 to 40]. These are the same operations as the ones you can perform on a `HashSet` and are explained with a nice diagram in `Chapter 2`, *Working with Collections; Using a HashSet*.

Working with single flags in the bit field is also very easy. `insert()` sets a flag in the field to active [45], `remove()` sets it to inactive [50], and `toggle` flips it from active to inactive and vice versa [48]. If you don't yet know whether you're going to `insert` or `remove` a flag, as is the case with unpredictable user input, you can use `set()` to explicitly set the activation of a flag to either `true` or `false` [55].

You can check whether a certain flag is active by calling `contains()` [56]. This also works for another bit field or a flag combination. This means that the following is also valid:

```
if custom.contains(Spices::SALT | Spices::PEPPER) {
    println!("The custom spice contains both salt and pepper");
}
```

Additionally, you can use `intersects()` to check whether *any* flags in two-bit fields match.

Last but not least, you can deserialize raw bytes into your generated bit field struct by calling `from_bits()` on it [62]. This will check whether every bit actually corresponds to a flag and return `None` otherwise. You can skip the error checking and simply ignore invalid bits with `from_bits_truncate()` if you're absolutely 100% sure that the data has to be valid.

See also

- *Using a HashSet* recipe in `Chapter 2`, *Working with Collections*

Providing custom derives

You might have looked at `#[derive(Debug)]` and assumed it's some weird compiler magic. It is not. It is a so-called *procedural macro*, that is, a macro that doesn't simply expand at compile time but instead *runs* at compile time. This way, you can inject code into the actual compilation process. The most useful application for this is creating custom derives, with which you can generate new code based on the analysis of existing code.

Getting started

This recipe will operate with an *Abstract Syntax Tree*, or AST. It is a tree-like representation of a language's elements in relation to each other. In this recipe, we (that is, a cool crate called `syn`) will parse our entire program into a single deep `struct`.

How to do it...

1. Create a new sub-crate for the custom derive with `cargo new chapter-five-derive`.
2. Open the newly generated `chapter-five-derive/Cargo.toml`.
3. Add this directly above the `[dependencies]` section of the file in order to mark the crate as a procedural macro crate:

```
[lib]
proc-macro = true
```

4. Under `[dependencies]`, add the following lines:

```
syn = "0.11.11"
quote = "0.3.15"
```

If you want, you can go to the crates.io web pages for `syn` (`https://crates.io/crates/syn`) and `quote` (`https://crates.io/crates/quote`) to check for the newest version and use that one instead.

5. In the `chapter-five-derive/src/lib.rs` file, delete the generated code and add the following:

```
extern crate proc_macro;
#[macro_use]
extern crate quote;
extern crate syn;

use proc_macro::TokenStream;

// HelloWorld is the name for the derive
// hello_world_name is the name of our optional attribute
#[proc_macro_derive(HelloWorld, attributes(hello_world_name))]
pub fn hello_world(input: TokenStream) -> TokenStream {
  // Construct a string representation of the type definition
  let s = input.to_string();
  // Parse the string representation into an abstract syntax tree
  let ast = syn::parse_derive_input(&s).expect("Failed to parse the source into an AST");

  // Build the implementation
  let gen = impl_hello_world(&ast);

  // Return the generated implementation
  gen.parse()
    .expect("Failed to parse the AST generated from deriving from HelloWorld")
}

fn impl_hello_world(ast: &syn::DeriveInput) -> quote::Tokens {
  let identifier = &ast.ident;
  // Use the name provided by the attribute
  // If there is no attribute, use the identifier
  let hello_world_name = get_name_attribute(ast).unwrap_or_else(|| identifier.as_ref());
  quote! {
    // Insert an implementation for our trait
    impl HelloWorld for #identifier {
      fn hello_world() {
        println!(
          "The struct or enum {} says: \"Hello world from {}!\"",
          stringify!(#identifier),
          #hello_world_name
        );
      } //end of fn hello_world()
    } //end of impl HelloWorld
  } //end of quote
} //end of fn impl_hello_world

fn get_name_attribute(ast: &syn::DeriveInput) -> Option<&str> {
  const ATTR_NAME: &str = "hello_world_name";

  // Go through all attributes and find one with our name
  if let Some(attr) = ast.attrs.iter().find(|a| a.name() == ATTR_NAME) {
    // Check if it's in the form of a name-value pair
    if let syn::MetaItem::NameValue(_, ref value) = attr.value {
      // Check if the value is a string
      if let syn::Lit::Str(ref value_as_str, _) = *value {
            Some(value_as_str)
          } else {
            panic!(
              "Expected a string as the value of {}, found {:?}
               instead",
               ATTR_NAME, value
            );
          }
        } else {
          panic!(
            "Expected an attribute in the form #[{} = \"Some
             value\"]",
             ATTR_NAME
          );
        }
      } else {
        None
      }
}
```

6. In the original `Cargo.toml` file of this chapter, add the following to the `[dependencies]`:

```
custom-derive = { path = "custom-derive" }
```

7. In the `bin` folder, create a file called `custom_derive.rs`.

8. Add the following code and run it with `cargo run --bin custom_derive`:

```
#[macro_use]
extern crate chapter_five_derive;

// trait definitions have to be in "consumer" crate
trait HelloWorld {
  // This method will send a friendly greeting
  fn hello_world();
}

// thanks to the code in the custom_derive crate
// we can derive from HelloWorld in order to provide
// an automatic implementation for the HelloWorld trait
#[derive(HelloWorld)]
struct Switzerland;

#[derive(HelloWorld)]
struct Britain;

```

```
19   #[derive(HelloWorld)]
20   // We can use an optional attribute to change the message
21   #[hello_world_name = "the Land Down Under"]
22   struct Australia;
23
24   fn main() {
25     Switzerland::hello_world();
26     Britain::hello_world();
27     Australia::hello_world();
28   }
```

How it works...

The instructions for this recipe are a bit more complex than the others, as we need to manage two separate crates. If your code doesn't compile, compare your version with the one used in the book at `https://github.com/SirRade/rust-standard-library-cookbook/tree/master/chapter_five`. We need to separate the code into two crates because providing a custom `derive` requires creating a procedural macro, as indicated by all of the instances of `proc_macro` in the code. A procedural macro is Rust code that runs alongside the compiler and interacts directly with it. Because of the special nature and unique restrictions of such code, it needs to be in a separate crate that is annotated with the following:

```
[lib]
proc-macro = true
```

This crate is typically named after the main crate with the `_derive` suffix added. In our example, the main crate is called `chapter_five`, so the crate providing the procedural macro is called `chapter_five_derive`.

In our example, we are going to create a derived version of good, old Hello World: a `struct` or `enum` deriving from `HelloWorld` will implement the `HelloWorld` trait, providing a `hello_world()` function with a friendly greeting containing its own name. Additionally, you can specify a `HelloWorldName` attribute to alter the message.

The code in `custom.rs` should be self-explanatory. We begin by importing our derivation crate [2] where we need to include the `#[macro_use]` attribute in order to actually import the procedural macros. We then define our `HelloWorld` trait [5] and proceed to derive it on a bunch of structures [13, 16 and 19], just like we would with built-in derives such as `Debug` or `Clone`. `Australia` gets a custom message via the `HelloWorldName` attribute. Lastly, in the `main` function, we call the generated `hello_world()` function.

Let's take a look at chapter-five-derive/src/lib.rs now. Procedural macro crates typically begin by importing the proc_macro, quote, and syn crates. Attentive readers will have noticed that we didn't add the proc_macro to our [dependencies] section in the crate's Cargo.toml. We didn't need to because this special support crate is provided by the standard Rust distribution.

The quote crate provides the quote! macro, which lets us translate Rust code into tokens that the compiler can use. The really useful feature of this macro is that it supports code interpolation of a variable by writing a # in front of it. This means that when we write the following, the value inside the struct_name variable is interpreted as Rust code:

```
        impl HelloWorld for #struct_name { ... }
```

If struct_name has the Switzerland value, the following code will be generated:

```
        impl HelloWorld for Switzerland { ... }
```

The syn crate is a Rust parser built upon the nom parser combinator framework (https://github.com/Geal/nom), which you should check out as well if you're thinking about writing a parser. In fact, some of the crates used in Chapter 4, *Serialization*, are written with nom, too. Back on track, syn parses the code annotated by your custom attributes or derives and lets you work with the generated abstract syntax tree.

The convention for a custom derive is to create a function with the name of the derive in snake_case (pub fn hello_world, in our case) that parses the annotated code and then calls a function that generates the new code. The second function typically has the name of the first one, prefixed with impl. In our code, this is fn impl_hello_world.

In a proc_macro crate, only functions tagged with proc_macro_derive are allowed to be published. The consequence of this is, by the way, that we are not able to move our HelloWorld trait into this crate; it wouldn't be allowed to be pub.

The proc_macro_derive annotation requires you to specify which name will be used for the derive (HelloWorld for us) and which attributes it allows. If we didn't want to accept the HelloWorldName attribute, we could simply omit the entire attributes section and annotate our function like this:

```
        #[proc_macro_derive(HelloWorld)]
```

Because `hello_world` hooks itself directly into the compiler, it both accepts and returns a `TokenStream`, which is the compiler-internal representation of Rust code. We start by turning the `TokenStream` back into a `String` in order to be parsed again by `syn`. This is not an expensive action, as the `TokenStream` we receive from the compiler is not the entire program, but only the part annotated by our custom derive. For example, the `String` behind the `TokenStream` of the first `struct` annotated by `HelloWorld` is simply the following:

```
struct Switzerland;
```

We then parse the said string with `syn::parse_derive_input(&s)`, which basically tells `syn` that the code we want to parse is a `struct` or `enum` that is deriving something.

We then generate the code with the following:

```
let gen = impl_hello_world(&ast);
```

Then we convert it back into a `TokenStream` with this:

```
gen.parse()
```

The `TokenStream` is then injected back into the code by the compiler. As you can see, a procedural macro cannot change existing code, but only analyze it and generate additional code.

Here is the process described in `hello_world`:

1. Convert the `TokenStream` into a `String`
2. Parse the `String` with `syn`
3. Generate an implementation of another method
4. Parse the implementation back into a `TokenStream`

It is very typical for a custom derive. You can reuse the code presented in nearly all basic procedural macros.

Let's move on to `impl_hello_world` now. With the help of the `ast` passed, we can analyze the annotated structure. The `ident` member, which stands for *identifier*, tells us the name of the `struct` or `enum`. For instance, in the first struct that derives from `HelloWorld`, this is the `"Switzerland"` string.

We then decide which name to use in the greeting with the help of the `get_name_attribute` little helper function, which we will look at in a moment. It returns the value of the `HelloWorldName` attribute if it has been set. If not, we default to the `identifier`, converted to a string via `as_ref`[29]. How this is done is explained in the next recipe.

Finally, we create some `quote::Tokens` by writing the implementation and surrounding it with `quote!`. Notice again how we interpolate variables into the code by writing `#` in front of it. Additionally, while printing, we surround `#identifier` with `stringify!`, which turns an identifier into a string. We don't need to do this with `#hello_world_identifier` because it already holds a string. To understand why this is needed, let's look at the code that would be generated for the `Switzerland` struct if we *didn't* include `stringify!`:

```
impl HelloWorld for Switzerland {
    fn hello_world() {
        println!(
            "The struct or enum {} says: \"Hello world from {}!\"",
            Switzerland,
            "Switzerland"
        );
    }
}
```

Try it out for yourself, and you will be greeted with an error message stating that something along the lines of "`` `Switzerland` cannot be formatted with the default formatter``". This is because we are not printing the `"Switzerland"` string, but instead trying to print the concept of the `Switzerland` struct itself, which is clearly nonsense. To fix this, we just need to make sure that the interpolated variable is surrounded by quotes (`"`), which is exactly what `stringify!` does.

Let's look at the final piece of the puzzle now: `get_name_attribute`. This function might look a little intimidating at first. Let's go through it step by step:

```
if let Some(attr) = ast.attrs.iter().find(|a| a.name() == ATTR_NAME) {
... }
```

Here we'll go through all available attributes and search for one named `"HelloWorldName"`. If we don't find any, the function call already ends by returning `None`. Otherwise, we continue with the following line:

```
if let syn::MetaItem::NameValue(_, ref value) = attr.value { ... }
```

`syn::MetaItem` is simply how `syn` calls attributes. This line is necessary because there are many ways to write attributes in Rust. For example, a `syn::MetaItem::Word` can be written like `#[foo]`. An example for `syn::MetaItem::List` is `#[foo(Bar, Baz, Quux)]`. `#[derive(...)]` itself also a `syn::MetaItem::List`. We, however, are only interested in `syn::MetaItem::NameValue`, which is an attribute in the form of `#[foo = Bar]`. If the `HelloWorldName` attribute is not in this form, we `panic!` with a message explaining what the problem is. A `panic` in procedural macro results in a compiler error. You can verify this by replacing `#[HelloWorldName = "the Land Down Under"` in `custom.rs` with `#[HelloWorldName]`.

Contrary to normal programs, because of procedural macros `panic!` at compile time, it's okay for them to `panic!` often. When doing so, remember that errors originating from other crates are very nasty to debug, doubly so in any kind of macros, so it's incredibly important to write the error messages as explicitly as possible.

The last check we need to do is on the value of `HelloWorldName`. As we are going to print it, we want to accept only strings:

```
        if let syn::Lit::Str(ref value_as_str, _) = *value { ... }
```

On success, we return the string. Otherwise, we again `panic!` with an error message detailing the problem.

There's more...

If you ran into trouble while running this recipe, you can use David Tolney's `cargo-expand` (`https://github.com/dtolnay/cargo-expand`) to show you how the compiler expanded your `proc_macros`. It's a really useful tool to debug your macros, so be sure to check it out.

The reason behind the two-crate restriction is historical and only temporary. In the beginning, there was only one way to define macros, `macro_rules!`. People with exotic needs, who were ready to put in the effort, were (and still are) able to extend their programs by directly hooking into the Rust compiler itself. Crates written this way are called *compiler plugins*. Of course, this is incredibly unstable because every minor Rust release can break your plugin, but people kept on doing it because it gave them one big advantage, custom derives. The core team reacted to the increased demand for language extensibility by deciding to launch `macros2.0` at some point in the future, bringing an overhaul to the entire macro system and many additional features, such as namespacing macros.

As they saw that most of the plugins were used only for custom derives, they also decided to bridge the time until `macros2.0` with `macros1.1`, which are also called *procedural macros*. Stabilizing the small subset of the compiler that is needed to create custom derives. The only problem was that crates now had parts that ran normally and parts that ran at compile time instead. Mixing them proved difficult to implement and a bit chaotic, so the two-crate system of moving all procedural macro code into a `-derive` crate was created. This is the system used in this recipe, as at the time of writing, `macros2.0` has not been stabilized yet. I encourage you to take a look at the current progress: `https://github.com/rust-lang/rust/issues/39412`.

If by the time you read this book, `macros2.0` has been released, you should update your knowledge about how to write modern custom derives.

Converting types into each other

Having dedicated types is great, but true flexibility can only come when these types can easily be converted from and into each other. Luckily, Rust provides this functionality quite literally through the `From` and `Into` traits, along with some friends.

How to do it...

1. In the `bin` folder, create a file called `conversion.rs`.
2. Add the following code and run it with `cargo run --bin conversion`:

```
use std::ops::MulAssign;
use std::fmt::Display;

// This structure doubles all elements it stores
#[derive(Debug)]
struct DoubleVec<T>(Vec<T>);


// Allowing conversion from a Vec<T>,
// where T is multipliable with an integer
impl<T> From<Vec<T>> for DoubleVec<T>
where
T: MulAssign<i32>,
{
  fn from(mut vec: Vec<T>) -> Self {
    for elem in &mut vec {
      *elem *= 2;
    }
    DoubleVec(vec)
  }
}

// Allowing conversion from a slice of Ts
// where T is again multipliable with an integer
impl<'a, T> From<&'a [T]> for DoubleVec<T>
where
T: MulAssign<i32> + Clone,
{
  fn from(slice: &[T]) -> Self {
    // Vec<T: MulAssign<i32>> automatically
    // implements Into<DoubleVec<T>>
    slice.to_vec().into()
  }
}

// Allowing conversion from a &DoubleVec<T> to a &Vec<T>
impl<T> AsRef<Vec<T>> for DoubleVec<T> {
  fn as_ref(&self) -> &Vec<T> {
    &self.0
  }
}


fn main() {
  // The following three are equivalent
  let hello_world = "Hello World".to_string();
  let hello_world: String = "Hello World!".into();
  let hello_world = String::from("Hello World!");

  // Vec<u8> implements From<&str>
  // so hello_world_bytes has the value b"Hello World!"
  let hello_world_bytes: Vec<u8> = "Hello World!".into();
  let hello_world_bytes = Vec::<u8>::from("Hello World!");

  // We can convert a Vec<T: MulAssign<i32>> into a DoubleVec
  let vec = vec![1, 2, 3];
  let double_vec = DoubleVec::from(vec);
  println!("Creating a DoubleVec from a Vec: {:?}",
  double_vec);

  // Vec<T: MulAssign<i32>> automatically implements
     Into<DoubleVec<T>>
  let vec = vec![1, 2, 3];
  let double_vec: DoubleVec<_> = vec.into();
  println!("Converting a Vec into a DoubleVec: {:?}", double_vec);

  // A reference to DoubleVec can be converted to a reference to Vec
  // Which in turn dereferences to a slice
  print_elements(double_vec.as_ref());

  // The standard library provides From<T> for Option<T>
  // You can design your API in an ergonomic way thanks to this
  easy_public_func(Some(1337), Some(123), None);
  ergonomic_public_func(1337, 123, None);
}


fn print_elements<T>(slice: &[T])
where
T: Display,
{
  for elem in slice {
    print!("{} ", elem);
  }
  println!();
}


// Easily written but cumbersome to use
fn easy_public_func(foo: Option<i32>, bar: Option<i32>, baz: Option<i32>) {
  println!(
    "easy_public_func = foo: {:?}, bar: {:?}, baz: {:?}",
     foo,
     bar,
     baz
  );
}


// This is quite some extra typing, so it's only worth to do for
// public functions with many optional parameters
fn ergonomic_public_func<Foo, Bar, Baz>(foo: Foo, bar: Bar, baz: Baz)
where
Foo: Into<Option<i32>>,
Bar: Into<Option<i32>>,
104  Baz: Into<Option<i32>>,
105  {
106    let foo: Option<i32> = foo.into();
107    let bar: Option<i32> = bar.into();
108    let baz: Option<i32> = baz.into();
109
110    println!(
111     "ergonomic_public_func = foo: {:?}, bar: {:?}, baz: {:?}",
112      foo,
113      bar,
114      baz
115    );
116  }
```

How it works...

The most important trait for conversion is `From`. Implementing it means defining how to obtain a type *from* another.

We present this on the example of `DoubleVec`[6]. Its concept is simple, when you construct it out of a `Vec`, it doubles all its elements. For this purpose, we implement `From<Vec<T>>` [11] with a `where` clause specifying `T: MulAssign<i32>`[13], which means that the trait will be implemented for all types that can be assigned to the result of a multiplication with an integer. Or, in terms of code, all types that allow the following, assuming the `t` variable is of the `T` type:

```
t *= 2;
```

The actual implementation should be self-explanatory, we simply multiply every element in the vector by two and wrap it in our `DoubleVec`[19]. Afterwards, we implement `From` as well for slices of the same type [25].

It is considered good practice to extend all of your trait definitions that work with vectors (`Vec<T>`) to also work with slices (`&[T]`). This way, you gain generality and performance, as you can operate on direct references to arrays (such as `&[1, 2, 3]`) and ranges of other vectors (`vec[1..3]`) without converting them first. This best practice carries over to functions as well, where you should always accept a slice of your type (as shown with `print_elements()`), if possible, for the same reasons.

In this implementation, however, we see something interesting:

```
fn from(slice: &[T]) -> Self {
    slice.to_vec().into()
}
```

`slice.to_vec()` simply converts the slice into a vector. But what's up with `.into()`? Well, it comes from the `Into` trait, which is the opposite of the `From` trait, it converts a type *into* another. But how does a `Vec` know how to turn into a `DoubleVec`?

Let's take a look at the standard library's implementation of `Into` at `https://github.com/rust-lang/rust/blob/master/src/libcore/convert.rs`, where we find the following lines:

```
        // From implies Into
        impl<T, U> Into<U> for T where U: From<T>
        {
            fn into(self) -> U {
                U::from(self)
            }
        }
```

Aha! According to this, every `T` type that implements `From` for `U` automatically lets `U` implement `Into` for `T`. And sure enough because we implemented `From<Vec<T> for DoubleVec<T>`, we automatically also implemented `Into<DoubleVec<T>> for Vec<T>`. Let's look at our code from before again:

```
        fn from(slice: &[T]) -> Self {
            slice.to_vec().into()
        }
```

The slice gets turned into a `Vec`, which implements `Into` for `DoubleVec`, among many others. Because our function signature says that we return `Self`, Rust knows which `Into` implementation to use, as only one of them returns `DoubleVec` as well.

Another useful trait for type conversion is `AsRef`. Its only function, `as_ref`, is nearly identical to `into`, but instead of moving itself into another type, it takes a reference to itself and returns a reference to another type. In a way, it *translates* references. You can expect this operation to be cheap in most cases, as it typically just returns a reference to an internal object. In fact, you have already used this method in the last recipe:

```
let hello_world_name = get_name_attribute(ast).unwrap_or_else(||
identifier.as_ref());
```

identifier internally holds a String of its name. The compiler knows that hello_world_name has to be a &str, as the return type of get_name_attribute(ast) is Option<&str> and we are trying to unwrap it with a default value. Based on this information, as_ref() tries to return a &str, which it can, as the only implementation of AsRef for identifier that returns a &str is the one that returns a reference to the aforementioned String that holds its name.

We are only implementing AsRef for Vec, and not for a slice, because of a reference to a vector (&Vec<T>) with automatically deref-coerce into a slice (&[T]), which means we automatically implement it. You can read more about the concept of deref coercion at https://doc.rust-lang.org/book/second-edition/ch15-02-deref.html#implicit-deref-coercions-with-functions-and-methods.

AsRef also has a brother called AsMut, which is identical but operates on mutable references. We intentionally didn't implement it in this example, as we don't want users messing with the internal state of DoubleVec. In general, you should be very conservative with this trait as well, as excessive access to the internals of anything can quickly become very chaotic.

The main function contains some examples of converting types. A popular example is the conversion from &str to String in lines [46 to 48]. Interestingly, &str can also be converted into a vector of its underlying bytes [52 and 53]. Let's look at how our DoubleVec can be converted in the same way.

The next line showcases how the &Vec<i32> returned by double_vec.as_ref() seamlessly behaves like an &[i32], as print_elements() only accepts slices [67]:

```
print_elements(double_vec.as_ref());
```

The last part of the recipe is about API design. There is a little implementation of From in the standard library that reads:

```
impl<T> From<T> for Option<T> {
    fn from(val: T) -> Option<T> {
        Some(val)
    }
}
```

This means that every type can be converted into an `Option`. You can use this trick, as showcased in the implementation of `ergonomic_public_func` [100], to make functions with multiple parameters of the `Option` type easier to use and look at, as you can see by comparing the following two function calls [71 and 72]:

```
easy_public_func(Some(1337), Some(123), None);
ergonomic_public_func(1337, 123, None);
```

However, because some extra typing is required to achieve this, it's okay if you only do this on functions that are part of your API, that is, available to users of your crate. If you want to read some more tips about clean API design in Rust, check out Rust core developer Pascal Hertleif's excellent blog entry: `https://deterministic.space/elegant-apis-in-rust.html`.

Boxing data

The first smart pointer we are going to look at is the `Box`. This very special type is the analogue to C++'s `unique_ptr`, a pointer to data stored on the heap that deletes said data automatically when it's out of scope. Because of the shift from stack to heap, `Box` can allow you some flexibility by intentionally losing type information.

How to do it...

1. In the `bin` folder, create a file called `boxing.rs`.

2. Add the following code and run it with `cargo run --bin boxing`:

```
use std::fs::File;
use std::io::BufReader;
use std::result::Result;
use std::error::Error;
use std::io::Read;
use std::fmt::Debug;

#[derive(Debug)]
struct Node<T> {
  data: T,
  child_nodes: Option<(BoxedNode<T>, BoxedNode<T>)>,
}
type BoxedNode<T> = Box<Node<T>>;
```

```

impl<T> Node<T> {
  fn new(data: T) -> Self {
    Node {
      data,
      child_nodes: None,
    }
  }

  fn is_leaf(&self) -> bool {
    self.child_nodes.is_none()
  }

  fn add_child_nodes(&mut self, a: Node<T>, b: Node<T>) {
    assert!(
      self.is_leaf(),
      "Tried to add child_nodes to a node that is not a leaf"
    );
    self.child_nodes = Some((Box::new(a), Box::new(b)));
  }
}

// Boxes enable you to use traditional OOP polymorph
trait Animal: Debug {
  fn sound(&self) -> &'static str;
}

#[derive(Debug)]
struct Dog;
impl Animal for Dog {
  fn sound(&self) -> &'static str {
    "Woof!"
  }
}

#[derive(Debug)]
struct Cat;
impl Animal for Cat {
  fn sound(&self) -> &'static str {
    "Meow!"
  }
}

fn main() {
  let mut root = Node::new(12);
  root.add_child_nodes(Node::new(3), Node::new(-24));
  root.child_nodes
    .as_mut()
    .unwrap()
    0
    .add_child_nodes(Node::new(0), Node::new(1803));
  println!("Our binary tree looks like this: {:?}", root);

  // Polymorphism
  let mut zoo: Vec<Box<Animal>> = Vec::new();
  zoo.push(Box::new(Dog {}));
  zoo.push(Box::new(Cat {}));
  for animal in zoo {
    println!("{:?} says {}", animal, animal.sound());
  }

  for word in caps_words_iter("do you feel lucky, punk?") {
    println!("{}", word);
  }

  // Assuming a file called number.txt exists
  let num = read_file_as_number("number.txt").expect("Failed read the file as a number");
  println!("number.txt contains the number {}", num);

  // Dynamically composing functions
  let multiplier = create_multiplier(23);
  let result = multiplier(3);
  println!("23 * 3 = {}", result);
}

// Via trait objects we can return any iterator
fn caps_words_iter<'a>(text: &'a str) -> Box<Iterator<Item = String> + 'a> {
  // Return an iterator over every word converted into ALL_CAPS
  Box::new(text.trim().split(' ').map(|word| word.to_uppercase()))
}

// Same goes for errors
fn read_file_as_number(filename: &str) -> Result<i32, Box<Error>> {
  let file = File::open(filename)?;
  let mut buf_reader = BufReader::new(file);
  let mut content = String::new();
  buf_reader.read_to_string(&mut content)?;
  let number: i32 = content.parse()?;
  Ok(number)
}

```

```
fn create_multiplier(a: i32) -> Box<Fn(i32) -> i32> {
  Box::new(move |b| a * b)
}
```

How it works...

The first thing that we are going to explore are recursive types, that is, a type that contains itself. This cannot be done directly, as the compiler needs to know in advance how much space a type requires. Consider the following `struct`:

```
struct Foo {
    bar: i32
}
```

The compiler will ask itself, `How much space does it take to create a Foo?` and see that it needs just enough space to hold an `i32`. And how much does an `i32` need? Exactly 32 bits. Now, consider the following:

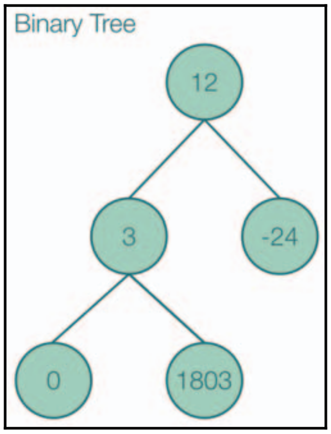
```
struct Foo {
    bar: i32,
    baz: Foo,
}
```

How much space does `Foo` need? Enough to hold an `i32` and a `Foo`. How much is an `i32`? 32 bits. And `Foo`? Enough to hold an `i32` and a `Foo`. And how much does that `Foo` take? Enough for a `Foo`, and so on, until the heat death of the universe. Clearly, we don't want to spend that long on compiling. Let's take a look at the solution to our problem:

```
struct Foo {
    bar: i32,
    baz: Box<Foo>,
}
```

One last time, how big is `Foo`? Enough to hold an `i32` and a `Foo`. How big is an `i32`? 32 bits. How big is a `Box<Foo>`? Just as big as a box of any other type, namely 64 bit. Every `Box` will always have the same size, as they are all the same thing, a pointer to some type in the heap. This way, we resolved our problem, as the compiler now knows the exact size of the type at compile time and is happy. And because it is happy, we are happy.

In our code example, we illustrate one possible use case for a recursive type, a naive binary tree implementation [9]. In case you didn't know, a binary tree consists of a clump of data, which is called a *node*, that can be connected to either zero or two other *child nodes*. A node that is connected to zero nodes is a *leaf*. In our example, we build such a tree that will look like this:



We implement it as a `struct Node` that contains any data and optionally a pair of `BoxedNode`, which is just an alias for `Box<Node>`.

A real binary tree implementation that is optimized for speed will be a little bit more complex than our example. While the concept of recursion fits very nicely into a binary tree, it is rather inefficient to store every node separately somewhere in the heap. Real implementations will instead just appear to be recursive for the user but internally store the nodes in a `Vec<Node>`. This way, the nodes profit from speed gains by simply being allocated in a continuous memory block, as this will optimize caching. Rust's `BTreeMap` and `BTreeSet` follow this concept as well.

A binary tree is an excellent kind of data structure for data traversal. You can read about some of its biggest use cases at the following StackOverflow answer by Danny Pflughoeft: `https://stackoverflow.com/questions/2130416/what-are-the-applications-of-binary-trees#2200588`.

The next thing that a `Box` enables us to do is classic polymorphism how you will recognize it from other languages. For this, we prepare a trait called `Animal` [37] that has a method to produce a `sound()`. Its implementor, `Dog` [42], will produce `"Woof!"`[45] and the `Cat` [50] implementor will produce `"Meow!"` [53]. Our goal is to store both a `Dog` and a `Cat` in a `Vec` of `Animal`. We can do this by creating a so-called *trait object*. It is created by a `Box` of a trait, like in our example in the following line:

```
let mut zoo: Vec<Box<Animal>> = Vec::new();
```

This way, we intentionally erase type information from the actual type in the `Box`. The compiler no longer knows which type is in the `Box`, only that it implements `Animal`, which is all that it needs to know. As you can see by running the code, the Rust runtime will still execute the correct functions and give the `Dog` and the `Cat` different sounds.

With the same mechanism, we can return a trait object of `Iterator`[90]:

```
fn caps_words_iter<'a>(text: &'a str) -> Box<Iterator<Item = String> +
'a> { ... }
```

This way, we can mix and match iterators inside `caps_words_iter()` without caring for the exact return type, so long as it implements `Iterator`, which they all do. Remember that we can't just return `Iterator` directly without any `Box` around it, as we cannot return a trait. A trait object, however, is completely fine.

On we go to `read_file_as_number()`[96]. This method reads a file and returns the content parsed as an `i32`. This file will not be generated for you, so you will have to either download it from our GitHub repo or manually create a file called `number.txt` that contains a number, say `777`. From the signature, you can gather that this time we are boxing the `Error`. This lets us mix the error type returned. Indeed, this method does return two different kinds of errors: `std::io::Error` and `std::num::ParseIntError`.

The last thing that we are going to look at is how to return closures with `create_multiplier()`[105]. As all closures implement either `Fn`, `FnOnce`, and/or `FnMut`, we can return a trait object for them. This way, we can create, compose, and change functions at runtime, just like with functional languages.

There's more...

You might have noticed that returning a `Box<Iterator>` or a `Box<Error>` comes at a small cost in terms of efficiency, as it requires moving an object onto the heap without it having any reason to go there. There are currently two ways around this.

For `Box<Error>`, you should instead create an own `Error` type, combining all the possible errors that can be returned from your function. This is detailed in `Chapter 6`, *Handling Errors*; *Providing user-defined Error types*.

For `Box<Iterator>`, you can analyze the compiler's output in order to find out the exact true type that you're returning. This works for small iterators, but any complex iterator will take a long time to crack. Because this situation is not really desirable, the Rust team has approved the introduction of *abstract types*, which will be introduced in `Chapter 10`, *Using Experimental Nightly Features*; *Returning abstract types* because it has not yet hit stable Rust.

See also

- *Providing user-defined error types* recipe in `Chapter 6`, *Handling Errors*
- *Returning abstract types* recipe in `Chapter 10`, *Using Experimental Nightly Features*

Sharing ownership with smart pointers

Some ownership relationships are not as straightforward as *type A owns B*. Sometimes, an entire group of types owns another type. To handle this, we need another smart pointer that behaves mostly like `Box` but only deletes the underlying resource if no one needs it anymore, it is `Rc`, which stands for Reference Counted.

How to do it...

1. In the `bin` folder, create a file called `shared.rs`

2. Add the following code and run it with `cargo run --bin shared`:

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

// The ball will survive until all kids are done playing with it
struct Kid {
  ball: Rc<Ball>,
}
struct Ball;

fn main() {
  {
    // rc is created and count is at 1
    let foo = Rc::new("foo");
    // foo goes out of scope; count decreases
    // count is zero; the object gets destroyed
  }

  {
    // rc is created and count is at 1
    let bar = Rc::new("bar");
    // rc is cloned; count increases to 2
    let second_bar = Rc::clone(&bar);
    // bar goes out of scode; count decreases to 1
    // bar goes out of scode; count decreases to 0
  }

  {
    // rc is created and count is at 1
    let baz = Rc::new("baz");
    {
      // rc is cloned; count increases to 2
      let second_baz = Rc::clone(&baz);
      // second_baz goes out of scode; count decreases to 1
    }
    // baz goes out of scode; count decreases to 0
  }
  let kid_one = spawn_kid_with_new_ball();
  let kid_two = Kid {
    ball: Rc::clone(&kid_one.ball),
  };
  let kid_three = Kid {
    ball: Rc::clone(&kid_one.ball),
  };
  // ball lives until here
}
```

```

fn spawn_kid_with_new_ball() -> Kid {
  let ball = Rc::new(Ball);
  Kid {
    ball: Rc::clone(&ball),
  }
  // Although the ball goes out of scope here, the object
     behind it

  // will survive as part of the kid
}
```

How it works...

The heart and soul of an `Rc` is its internal counter of how many objects own it at the moment. Every time the `Rc` is cloned, it increases by one, and every time one of its clones go out of scope, it decreases by one. The moment this counter reaches zero, the objects behind the `Rc` are destroyed. The `main` method contains some annotated examples of values the counter reaches during its lifetime to help you understand how the mechanism works.

The effect of the simple rules presented is that a resource behind an `Rc` will only be deleted once it's no longer used, at the slim cost of a very small runtime performance loss because of the ongoing counting. This *delayed deletion* effect is ideal for resources that are shared between objects. Simply wrap them in an `Rc` and they will make sure that everything survives long enough. This is the equivalent of `shared_ptr` in C++.

There's more...

In one edge case, reference counts can cause a *memory leak*, that is, accidentally preventing resources from ever being deleted. This happens when two objects exist that both contain an `Rc` pointing to each other. Because of this circular dependence, none of them will stop using the other and so the two objects will continue existing and pointing at each other long after your code has stopped using them. The solution here is to pick the weaker link in the hierarchy and replace its `Rc` for a `Weak`, which contains a *non-owning reference* instead. Because this situation is fairly rare, however, we are not going to look at it in detail. Instead, simply remember the possibility of a memory leak and come back to read this again when it arises.

Rc is inherently singlethreaded. If you need its functionality in a multithreaded environment (such as the one we are going to explore in Chapter 7, *Parallelism and Rayon*; *Sharing resources in multithreaded closures*), you can use Arc instead. It stands for Atomically Reference Counted and behaves the same way as Rc.

See also

- *Sharing resources in multithreaded closures* recipe in Chapter 7, *Parallelism and Rayon*

Working with interior mutability

Although Rust's borrow checker is one of its biggest selling points, alongside its clever error-handling concept and impressive tooling, it cannot read minds yet. Sometimes you might have to take things into your own hands and borrow objects manually. This is done with the concept of *interior mutability*, which states that certain types can wrap objects while being non-mutable and still operate on them mutably.

How to do it...

1. In the bin folder, create a file called interior_mutability.rs.
2. Add the following code and run it with cargo test --bin interior_mutability:

```
trait EmailSender {
  fn send_mail(&self, msg: &Email) -> Option<String>;
}

#[derive(Debug, Clone)]
struct Email {
  from: String,
  to: String,
  msg: String,
}

#[derive(Debug)]
struct Customer {
    address: String,
    wants_news: bool,
  }

  // Send news to every customer that wants to receive them
  fn publish_news(msg: &str, sender: &EmailSender, customers: &[Customer]) -> Option<i32> {
    let mut count = 0;
    let mut mail = Email {
    from: "Rust Newsletter".to_string(),
    to: "".to_string(),
    msg: msg.to_string(),
  };
  for customer in customers {
    if !customer.wants_news {
      continue;
    }
    mail.to = customer.address.to_string();
    if sender.send_mail(&mail).is_none() {
      return None;
    }
    count += 1;
  }
  Some(count)
}

  fn main() {
  // No code running as we are concentrating on the tests instead
  }


  #[cfg(test)]
  mod tests {
    use super::*;
    use std::cell::RefCell;

    struct MockEmailSender {
      // sent_mails can be modified even if MockEmailSender is immutable
      sent_mails: RefCell<Vec<Email>>,
    }
    impl MockEmailSender {
      fn new() -> Self {
        MockEmailSender {
          sent_mails: RefCell::new(Vec::new()),
        }
      }
      }

      impl EmailSender for MockEmailSender {
      fn send_mail(&self, msg: &Email) -> Option<String> {
        // Borrow sent_mails mutably
        self.sent_mails.borrow_mut().push(msg.clone());
        Some("200 OK".to_string())
      }
    }

    #[test]
    fn sends_zero_to_zero_customers() {
      let sent = publish_news("hello world!",
      &MockEmailSender::new(), &[]);
      assert_eq!(Some(0), sent);
    }

    #[test]
    fn sends_one_to_one_willing() {
      let customer = Customer {
        address: "herbert@herbert.com".to_string(),
        wants_news: true,
      };
      let sent = publish_news("hello world!",
      &MockEmailSender::new(), &[customer]);
      assert_eq!(Some(1), sent);
    }

  #[test]
    fn sends_none_to_unwilling() {
      let customer_one = Customer {
        address: "herbert@herbert.com".to_string(),
        wants_news: false,
      };
      let customer_two = Customer {
        address: "michael@jackson.com".to_string(),
        wants_news: false,
      };
      let sent = publish_news(
        "hello world!",
         &MockEmailSender::new(),
         &[customer_one, customer_two],
      );
      assert_eq!(Some(0), sent);
    }

    #[test]
    fn sends_correct_mail() {
```

```
105     let customer = Customer {
106       address: "herbert@herbert.com".to_string(),
107       wants_news: true,
108     };
109     let sender = MockEmailSender::new();
110     publish_news("hello world!", &sender, &
        [customer]).expect("Failed to send mail");
111
112     // Borrow sent_mails immutable
113     let mails = sender.sent_mails.borrow();
114     assert_eq!(1, mails.len());
115     assert_eq!("Rust Newsletter", mails[0].from);
116     assert_eq!("herbert@herbert.com", mails[0].to);
117     assert_eq!("hello world!", mails[0].msg);
118   }
119 }
```

How it works...

The protagonist of this recipe is `RefCell`, a wrapper around any type that moves the borrow checker's rule enforcement from compile time to runtime. The basics are pretty easy, you borrow the underlying value immutably by calling `.borrow()` and borrow it mutably by calling `.borrow_mut()`. If you don't follow the golden rule of only having multiple readers or one single writer at the same time, the program goes into `panic!`. One application for this is making members of your structs mutable even though your struct itself is immutable. The best use case to show where this is useful is mocking, the art of faking infrastructure for testing purposes.

The idea of our example is as follows, we want to send a newsletter to every customer that is interested. For that, we have the `EmailSender` trait[1], which just specifies a method to send an `Email` and return a response[2]. It's good practice to try to define functionality through traits in order to mock them.

Our `publish_news` function [19] takes a message, an `EmailSender` and a slice of `Customer` (please don't imagine that literally), and sends the message to all customers who want to receive news. If it encounters an error, it returns `None`[32], otherwise, it returns the number of newsletters it sent [36].

You probably don't want to send an email to your customers every time you test your code, which is why we create `MockEmailSender`[49] inside the test configuration, which doesn't actually send anything but instead stores all mail in a `Vec` [64]. In order to do this, it needs to modify its member despite being immutable. That's exactly what `RefCell` is for! Thanks to this, we can efficiently test `publish_news()`, as we can access all messages it would have sent and compare them with what we expect.

There's more...

There are many types that use interior mutability. Among them are `Cell`, which, instead of handing out references, simply copies the value it stores. This is nice when storing primitive types such as `i32` or `bool`, as they all implement copy anyway.

Others are `RwLock` and `Mutex`, which are important for parallelism, as we shall see in their recipes in `Chapter 7`, *Parallelism and Rayon*; *Access resources in parallel with RwLocks*.

See also

- *Access resources in parallel with RwLocks* recipe in `Chapter 7`, *Parallelism and Rayon*

6
Handling Errors

In this chapter, we will cover the following recipes:

- Providing user-defined error types
- Providing logging
- Creating a custom logger
- Implementing the Drop trait
- Understanding RAII

Introduction

Mistakes happen, and that's okay. We are only human after all. The important thing in life and in programming is not which errors we make, but how we deal with them. Rust helps us with the programming aspect of this principle by providing us with an error handling concept that guarantees that we *have* to think about the consequences of failure when dealing with functions that can fail, as they don't return a value directly but wrapped in a `Result` that has to be opened somehow. The only thing left is designing our code in a way that integrates nicely with this concept.

Providing user-defined error types

In the previous chapters, we learned about a few ways to handle functions that have to deal with different kinds of errors. So far, we have:

- Simply panicked when encountering them

- Returned only one kind of error and converted all others into it
- Returned different kinds of errors in a `Box`

Most of these have been used because we didn't reach this recipe yet. Now, we are going to learn about the preferred way of doing things, creating a custom kind of `Error` that contains multiple sub-errors.

How to do it...

1. Create a Rust project to work on during this chapter with `cargo new chapter-six`.
2. Navigate to the newly created `chapter-six` folder. For the rest of this chapter, we will assume that your command line is currently in this directory.
3. Inside the folder `src`, create a new folder called `bin`.
4. Delete the generated `lib.rs` file, as we are not creating a library.
5. In the folder `src/bin`, create a file called `custom_error.rs`.
6. Add the following code and run it with `cargo run --bin custom_error`:

```rust
use std::{error, fmt, io, num, result};
use std::fs::File;
use std::io::{BufReader, Read};

#[derive(Debug)]
// This is going to be our custom Error type
enum AgeReaderError {
  Io(io::Error),
  Parse(num::ParseIntError),
  NegativeAge(),
}

// It is common to alias Result in an Error module
type Result<T> = result::Result<T, AgeReaderError>;

impl error::Error for AgeReaderError {
  fn description(&self) -> &str {
    // Defer to the existing description if possible
    match *self {
    AgeReaderError::Io(ref err) => err.description(),
    AgeReaderError::Parse(ref err) => err.description(),
    // Descriptions should be as short as possible
    AgeReaderError::NegativeAge() => "Age is negative",
    }
  }

  fn cause(&self) -> Option<&error::Error> {
    // Return the underlying error, if any
    match *self {
      AgeReaderError::Io(ref err) => Some(err),
      AgeReaderError::Parse(ref err) => Some(err),
      AgeReaderError::NegativeAge() => None,
    }
  }
}

impl fmt::Display for AgeReaderError {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    // Write a detailed description of the problem
    match *self {
      AgeReaderError::Io(ref err) => write!(f, "IO error: {}",
       err),
      AgeReaderError::Parse(ref err) => write!(f, "Parse
       error: {}", err),
      AgeReaderError::NegativeAge() => write!(f, "Logic error:
       Age cannot be negative"),
    }
  }
}

// Implement From<T> for every sub-error
impl From<io::Error> for AgeReaderError {
  fn from(err: io::Error) -> AgeReaderError {
  AgeReaderError::Io(err)
  }
}

impl From<num::ParseIntError> for AgeReaderError {
  fn from(err: num::ParseIntError) -> AgeReaderError {
  AgeReaderError::Parse(err)
  }
}

fn main() {
  // Assuming a file called age.txt exists
  const FILENAME: &str = "age.txt";
  let result = read_age(FILENAME);
  match result {
    Ok(num) => println!("{} contains the age {}", FILENAME,
     num),
    Err(AgeReaderError::Io(err)) => eprintln!("Failed to open
     the file {}: {}", FILENAME, err),
68      Err(AgeReaderError::Parse(err)) => eprintln!(
69       "Failed to read the contents of {} as a number: {}",
70        FILENAME, err
71         ),
72      Err(AgeReaderError::NegativeAge()) => eprintln!("The age in
         the file is negative"),
73      }
74  }
75
76  // Read an age out of a file
77  fn read_age(filename: &str) -> Result<i32> {
78    let file = File::open(filename)?;
79    let mut buf_reader = BufReader::new(file);
80    let mut content = String::new();
81    buf_reader.read_to_string(&mut content)?;
82    let age: i32 = content.trim().parse()?;
83    if age.is_positive() {
84      Ok(age)
85    } else {
86      Err(AgeReaderError::NegativeAge())
87    }
88  }
```

How it works...

The purpose of our example is to read a file, `age.txt`, and return the number written in it, assuming that it represents some kind of age. We can encounter three errors during this process:

- Failure to read the file (maybe it doesn't exist)
- Failure to read its content as a number (it could contain text as well)
- The number could be negative

These possible error states are the possible variants of our `Error enum`: `AgeReaderError`[7]. It is usual to name the variants after the sub-errors they represent. Because a failure to read the file raises an `io::Error`, we name our corresponding variant `AgeReaderError::Io`[8]. A failure to parse a `&str` as an `i32` raises a `num::ParseIntError`, so we name our encompassing variant `AgeReaderError::Parse`[9].

TIP

These two `std` errors show the naming convention of errors neatly. If you have many different errors that can be returned by a module, export them via their full name, such as `num::ParseIntError`. If your module only returns one kind of `Error`, simply export it as `Error`, such as `io::Error`. We intentionally don't follow this convention in the recipe because the distinct name `AgeReaderError` makes it easier to talk about it. If this recipe was included, one for one, in a crate, we could achieve the conventional effect by exporting it as `pub type Error = AgeReaderError;`.

The next thing we create is an alias for our own `Result`[14]:

```
type Result<T> = result::Result<T, AgeReaderError>;
```

This is an extremely common pattern for your own errors, which makes working with them a charm, as we see in the return type of `read_age`[77]:

```
fn read_age(filename: &str) -> Result<i32> { ... }
```

Looks nice, doesn't it? In order to use our `enum` as an `Error` though, we need to implement it first [16]. The `Error` trait requires two things: a `description`[17], which is a short explanation of what went wrong, and a `cause`[27], which is simply a *redirection* to the underlying error, if any. You can (and should) provide a detailed description of the problem at hand by also implementing `Display` for your `Error`[37]. In all of these implementations, you should refer to the underlying error if possible, as with the following line [20]:

```
AgeReaderError::Io(ref err) => err.description()
```

The last thing you need to provide for a good own `Error` is a `From` implementation for every sub-error. In our case this would be `From<io::Error>`[49] and `From<num::ParseIntError>`[55]. This way, the `try` operator (`?`) will automatically convert the involved errors for us.

After implementing all necessary traits, you can return the custom `Error` from any function and unwrap values in it with the aforementioned operator. In this example, when checking the result of `read_age`, we didn't have to `match` the returned value. In a real `main` function, we would probably just call `.expect("...")` on it, but we matched the individual error variants anyway to show you how nicely you can react to different problems when using known error types [65 to 73]:

```
match result {
    Ok(num) => println!("{} contains the age {}", FILENAME, num),
    Err(AgeReaderError::Io(err)) => eprintln!("Failed to open the file
            {}: {}", FILENAME, err),
            Err(AgeReaderError::Parse(err)) => eprintln!(
                "Failed to read the contents of {} as a number: {}",
                FILENAME, err
            ),
            Err(AgeReaderError::NegativeAge()) => eprintln!("The age in the
        file is negative"),
        }
```

There's more...

A crate's `Error` is usually put inside an own `error` module for organizational reasons and then exported directly for optimal usability. The relevant `lib.rs` entries would look like this:

```
        mod error;
        pub use error::Error;
```

Providing logging

In a big application, things will sooner or later not go as planned. But that's okay, as long as you have provided a system for your users to know what went wrong and, if possible, why. One time-tested tool to accomplish this is detailed logs that let the user specify for themselves how much diagnosis they want to see.

How to do it...

Follow these steps:

1. Open the `Cargo.toml` file that has been generated earlier for you.
2. Under `[dependencies]`, add the following line:

```
log = "0.4.1"
env_logger = "0.5.3"
```

3. If you want, you can go to log's (`https://crates.io/crates/log`) or `env_logger`'s (`https://crates.io/crates/env_log`) crates.io pages to check for the newest version and use that one instead.

4. In the folder `bin`, create a file called `logging.rs`.
5. Add the following code and run it with `RUST_LOG=logging cargo run --bin logging` if you're on a Unix-based system. Otherwise, run `$env:RUST_LOG="logging"; cargo run --bin logging` on Windows:

```
extern crate env_logger;
#[macro_use]
extern crate log;
use log::Level;

fn main() {
  // env_logger's priority levels are:
  // error > warn > info > debug > trace
  env_logger::init();
  // All logging calls log! in the background
  log!(Level::Debug, "env_logger has been initialized");

  // There are convenience macros for every logging level however
  info!("The program has started!");

  // A log's target is its parent module per default
  // ('logging' in our case, as we're in a binary)
  // We can override this target however:
  info!(target: "extra_info", "This is additional info that will only show if you \
  activate info level logging for the extra_info target");

  warn!("Something that requires your attention happened");

  // Only execute code if logging level is active
  if log_enabled!(Level::Debug) {
  let data = expensive_operation();
  debug!("The expensive operation returned: \"{}\"", data);
  }

  error!("Something terrible happened!");
}

fn expensive_operation() -> String {
  trace!("Starting an expensive operation");
  let data = "Imagine this is a very very expensive task".to_string();
  trace!("Finished the expensive operation");
  data
}
```

How it works...

Rust's logging system is based on the `log` crate, which provides a common *facade* for all things logging. This means that it doesn't actually provide any functionality, just the interface. The implementation is left to other crates, `env_logger` in our case. This split into facade and implementation is pretty useful, as anyone can create a cool new way of logging stuff which is automatically compatible with any crate.

The choice of logging implementation used should be up to the consumer of your code. If you write a crate, don't use any implementation but simply log all things via the `log` crate only. Your (or someone else's) executable that uses the crate can then simply initialize their logger of choice[9] in order to actually process the log calls.

The `log` crate provides the `log!` macro[11], which accepts a log `Level`, a message that can be formatted the same way as in `println!`, and an optional `target`. You could log stuff like this, but it's more readable to use the convenience macros for every logging level, `error!`, `warn!`, `info!`, `debug!`, and `trace!`, which all simply call `log!` in the background. A log's `target`[19] is an additional property that helps the logger implementation group the logs thematically. If you omit the `target`, it defaults to the current `module`. So, for example, if you logged something from the `foo` crate, its `target` would default to `foo`. If you logged something in its submodule `foo::bar`, its `target` would default to `bar`. If you then consumed the crate in a `main.rs` and logged something there, its `target` would default to `main`.

Another goodie that `log` provides is the `log_enabled!` macro, which returns whether or not the currently active logger is set to process a certain warning level. This is especially useful in combination with `Debug` logs that provide useful information at the cost of an expensive operation.

`env_logger` is a logger implementation provided by the Rust nursery. It prints its logs on `stderr` and uses pretty colors for different logging levels if supported by your Terminal. It relies on a `RUST_LOG` envvar to filter which logs should be displayed. If you don't define said variable, it will default to `error`, which means that it will print only the log level `error` from all targets. As you can guess, other possible values include `warn`, `info`, `debug`, and `trace`. These will, however, not only filter the specified level, but all levels *above* it as well, where the hierarchy is defined like this:

```
error > warn > info > debug > trace
```

This means that setting your `RUST_LOG` to `warn` will show all `warn` and all `error` logs. Setting it to `debug` will show `error`, `warn`, `info`, and `debug`.

Instead of error levels, you can set `RUST_LOG` to targets, which will show all logs of the selected targets regardless of their log levels. This is what we do in our example, we set `RUST_LOG` to `logging` in order to show all logs with a `target` called `logging`, which is the standard target for all logs in our binary. If you wanted, you could combine a level filter with a target filter like this: `logging=warn`, which would only show `warn` and `error` logs with a `target` of `logging`.

You can combine different filters with a comma. If you want all logs of this example to be displayed, you can set your variable to `logging,extra_info`, which filters for both the targets `logging` and `extra_info`.

Lastly, you can filter your logs by content with a slash (`/`), after which you can write down a regex that has to be matched. If you set `RUST_LOG` to `logging=debug/expensive` for instance, only logs with the logging level of `debug` and upwards with the `target` of `logging` that also contain the word `expensive` will be displayed.

Wow, that's a lot of configuration! I advise you to experiment a bit with the different filtering modes and run the example in order to get a feeling for how the parts fit together. If you need additional information, all possibilities for the value of `RUST_LOG` in the current version of `env_logger` are documented at `https://docs.rs/env_logger/`.

There's more...

If you've never worked with a logger before, you might wonder what the difference between certain log levels is. Of course, you can use them for whatever purpose you want, but the following conventions are usual for loggers in many languages:

Log level	**Usage**	**Example**
`Error`	Some major problem occurred that might terminate the program soon. If the application is a service that should always run, a system administrator should immediately be notified.	The connection to the database has been broken.
`Warn`	An issue that's not severe or has an automatic workaround has happened. Someone should check this out at some point and fix it.	A user's configuration file contains unrecognized options that have been ignored.

`Info`	Some information that might be useful to look at at a later point. This logs normal conditions.	The user has started or stopped a process. A default value has been used because no configuration has been provided.
`Debug`	Information that is helpful to programmers or sysadmins when trying to fix a problem. Contrary to many other languages, a debug log is *not* removed in release builds.	The parameters passed to a major function. The current state of the application at various points.
`Trace`	Very low-level signals of control flow that are only useful to a programmer trying to chase a bug. Allows the reconstruction of a stack trace.	The parameters of a minor helper function. The beginning and end of a function.

Many languages also contain a `Fatal` log level. In Rust, a good old `panic!()` is used for that. If you want to log your panics in some special way as well, you can replace the usual reaction to a panic by simply printing it to `stderr` by calling `std::panic::set_hook()` with whatever functionality you want. An example of what this might look like is as follows:

```
std::panic::set_hook(Box::new(|e| {
    println!("Oh noes, something went wrong D:");
    println!("{:?}", e);
}));
panic!("A thing broke");
```

A good alternative to `env_logger` is the `slog` crate, which provides excellent extensible structured logging at the cost of a steepened learning curve. Plus, its output looks pretty. If that sounds interesting, be sure to check it out at `https://github.com/slog-rs/slog`.

Creating a custom logger

Sometimes you or your users might have very specific logging needs. In this recipe, we are going to learn how to create a custom logger to work with the `log` crate.

How to do it...

1. Open the `Cargo.toml` file that was generated earlier for you.
2. Under `[dependencies]`, if you didn't do so in the last recipe, add the following line:

```
log = "0.4.1"
```

If you want, you can go to log's crates.io page (`https://crates.io/crates/log`) to check for the newest version and use that one instead.

3. In the folder `bin`, create a file called `custom_logger.rs`.
4. Add the following code and run it with `RUST_LOG=custom_logger cargo run --bin custom_logger` if you're on a Unix-based system. Otherwise, run `$env:RUST_LOG="custom_logger"; cargo run --bin custom_logger` on Windows:

```
#[macro_use]
extern crate log;

use log::{Level, Metadata, Record};
use std::fs::{File, OpenOptions};
use std::io::{self, BufWriter, Write};
use std::{error, fmt, result};
use std::sync::RwLock;
use std::time::{SystemTime, UNIX_EPOCH};

// This logger will write logs into a file on disk
struct FileLogger {
  level: Level,
  writer: RwLock<BufWriter<File>>,
}

impl log::Log for FileLogger {
  fn enabled(&self, metadata: &Metadata) -> bool {
    // Check if the logger is enabled for a certain log level
    // Here, you could also add own custom filtering based on
     targets or regex
    metadata.level() <= self.level
  }

  fn log(&self, record: &Record) {
    if self.enabled(record.metadata()) {
      let mut writer = self.writer
        .write()
        .expect("Failed to unlock log file writer in write
         mode");
      let now = SystemTime::now();
      let timestamp = now.duration_since(UNIX_EPOCH).expect(
       "Failed to generate timestamp: This system is
        operating before the unix epoch",
      );
      // Write the log into the buffer
      write!(
        writer,
        "{} {} at {}: {}\n",
        record.level(),
        timestamp.as_secs(),
        record.target(),
        record.args()
      ).expect("Failed to log to file");
    }
    self.flush();
  }

  fn flush(&self) {
    // Write the buffered logs to disk
    self.writer
      .write()
      .expect("Failed to unlock log file writer in write
           mode")
      .flush()
      .expect("Failed to flush log file writer");
  }
}

impl FileLogger {
  // A convenience method to set everything up nicely
  fn init(level: Level, file_name: &str) -> Result<()> {
    let file = OpenOptions::new()
      .create(true)
      .append(true)
      .open(file_name)?;
    let writer = RwLock::new(BufWriter::new(file));
    let logger = FileLogger { level, writer };
    // set the global level filter that log uses to optimize
     ignored logs
    log::set_max_level(level.to_level_filter());
    // set this logger as the one used by the log macros
    log::set_boxed_logger(Box::new(logger))?;
    Ok(())
  }
}
```

This is the custom error used in our logger:

```
// Our custom error for our FileLogger
#[derive(Debug)]
enum FileLoggerError {
  Io(io::Error),
  SetLogger(log::SetLoggerError),
}

type Result<T> = result::Result<T, FileLoggerError>;
impl error::Error for FileLoggerError {
  fn description(&self) -> &str {
    match *self {
      FileLoggerError::Io(ref err) => err.description(),
      FileLoggerError::SetLogger(ref err) => err.description(),
    }
  }

  fn cause(&self) -> Option<&error::Error> {
    match *self {
      FileLoggerError::Io(ref err) => Some(err),
      FileLoggerError::SetLogger(ref err) => Some(err),
    }
  }
}

impl fmt::Display for FileLoggerError {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    match *self {
      FileLoggerError::Io(ref err) => write!(f, "IO error: {}", err),
      FileLoggerError::SetLogger(ref err) => write!(f, "Parse error: {}", err),
    }
  }
}

impl From<io::Error> for FileLoggerError {
  fn from(err: io::Error) -> FileLoggerError {
    FileLoggerError::Io(err)
  }
}

impl From<log::SetLoggerError> for FileLoggerError {
```

```
113   fn from(err: log::SetLoggerError) -> FileLoggerError {
114     FileLoggerError::SetLogger(err)
115   }
116 }
```

Initializing and using the logger:

```
118 fn main() {
119   FileLogger::init(Level::Info, "log.txt").expect("Failed to
       init
      FileLogger");
120   trace!("Beginning the operation");
121   info!("A lightning strikes a body");
122   warn!("It's moving");
123   error!("It's alive!");
124   debug!("Dr. Frankenstein now knows how it feels to be god");
125   trace!("End of the operation");
126 }
```

How it works...

Our `FileLogger` will, as the name suggests, log things to a file. It also accepts a maximum logging level on initialization.

TIP You should not get used to directly logging things on a disk. As stated by *The Twelve-Factor App guidelines* (`https://12factor.net/logs`), logs should be treated as event streams, in the form of a raw dump to `stdout`. The production environment can then route all log streams to their final destination over `systemd` or a dedicated log router such as `Logplex` (`https://github.com/heroku/logplex`) or `Fluentd` (`https://github.com/fluent/fluentd`). These will then decide if the logs should be sent to a file, an analysis system like `Splunk` (`https://www.splunk.com/`), or a data warehouse like `Hive` (`http://hive.apache.org/`).

Every logger needs to implement the `log::Log` trait, which consists of the `enabled`, `log`, and `flush` methods. `enabled` should return if a certain log event is accepted by the logger. Here, you can go wild with whatever filtering logic you want [18]. This method is never called directly by `log`, so its only purpose is to serve you as a helper method inside of the `log` method, which we are going to discuss shortly. `flush` [46] is treated the same way. It should apply whatever changes you have buffered in your logging, but it is never called by `log`.

In fact, if your logger doesn't interact with the filesystem or the network, it will probably simply implement `flush` by doing nothing:

```
fn flush(&self) {}
```

The real bread and butter of the `Log` implementation is the `log` method[24], though, as it is called whenever a logging macro is invoked. The implementation typically starts with the following line, followed by the actual logging and a finishing call to `self.flush()`[25]:

```
if self.enabled(record.metadata()) {
```

Our actual logging then consists of simply writing a combination of the current logging level, Unix timestamp, target, and logging message to the file and flushing it afterward.

Technically speaking, our call to `self.flush()` should be inside the `if` block as well, but that would require additional scope around the mutably borrowed `writer` in order to not borrow it twice. Because this is not relevant to the underlying lesson here, as in, how to create a logger, we placed it outside the block in order to make the example more readable. By the way, the way we borrow `writer` from an `RwLock` is the subject of the *Accessing resources in parallel with RwLocks* in `Chapter 8`, *Parallelism and Rayon*. For now, it's enough to know that an `RwLock` is a `RefCell` that is safe to use in a parallel environment such as a logger.

After implementing `Log` for `FileLogger`, the user can use it as the logger called by `log`. To do that, the user has to do two things:

- Tell `log` which log `Level` is going to be the maximum accepted by our logger via `log::set_max_level()` [66]. This is needed because `log` optimizes `.log()` calls on our logger away during runtime if they use log levels over our maximum level. The function accepts a `LevelFilter` instead of a `Level`, which is why we have to convert our level with `to_level_filter()` first [66]. The reason for this type is explained in the *There's more...* section.
- Specify the logger with `log::set_boxed_logger()`[68]. `log` accepts a box because it treats its logger implementation as a trait object, which we discussed in the *Boxing data* section of `Chapter 5`, *Advanced Data Structures*. If you want a (very) minor performance gain, you can also use `log::set_logger()`, which accepts a `static` which you would have to create via the `lazy_static` crate first. See `Chapter 5`, *Advanced Data Structures*, and the recipe *Creating lazy static objects*, for more on that.

This is conventionally done in a provided `.init()` method on the logger, just like with `env_logger`, which we implement in line [58]:

```
        fn init(level: Level, file_name: &str) -> Result<()> {
            let file = OpenOptions::new()
                .create(true)
                .append(true)
                .open(file_name)?;
            let writer = RwLock::new(BufWriter::new(file));
            let logger = FileLogger { level, writer };
            log::set_max_level(level.to_level_filter());
            log::set_boxed_logger(Box::new(logger))?;
            Ok(())
        }
```

While we're on it, we can also open the file in the same method. Other possibilities include letting the user pass a `File` directly to `init` as a parameter or, for maximum flexibility, making the logger a generic one that accepts any stream implementing `Write`.

We then return a custom error created in the lines that follow [74 to 116].

An example initialization of our logger might look like this [119]:

```
    FileLogger::init(Level::Info, "log.txt").expect("Failed to init
    FileLogger");
```

There's more...

For simplicity's sake, `FileLogger` doesn't discriminate against any targets. A more sophisticated logger, like `env_logger`, can set different logging levels on different targets. For this purpose, `log` provides us with the `LevelFilter` enum, which has an `Off` state that corresponds to *no logging enabled for this target.* If you need to create such a logger, be sure to remember said enum. You can get some inspiration about how to implement target-based filters by looking at the source code of `env_logger` at `https://github.com/sebasmagri/env_logger/blob/master/src/filter/mod.rs`.

In a really user-friendly logger, you'll want to display the timestamp in the user's own local time. For all things related to time measurement, time zones, and dates, check out the `chrono` crate at `https://crates.io/crates/chrono`.

See also

- *Boxing data* recipe in `Chapter 5`, *Advanced Data Structures*
- *Creating lazy static objects* recipe in `Chapter 5`, *Advanced Data Structures*
- *Accessing resources in parallel with RwLocks* recipe in `Chapter 7`, *Parallelism and Rayon*

Implementing the Drop trait

Where traditional object-oriented languages have destructors, Rust has the `Drop` trait, which consists of a single `drop` function that is called whenever a variable's lifetime has ended. By implementing it, you can perform whatever cleanup or advanced logging is necessary. You can also automatically free resources via RAII, as we're going to see in the next recipe.

How to do it...

1. In the folder `bin`, create a file called `drop.rs`.

2. Add the following code and run it with `cargo run --bin drop`:

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

struct CustomSmartPointer<D>
where
D: Debug,
{
data: D,
}

impl<D> CustomSmartPointer<D>
where
D: Debug,
{
  fn new(data: D) -> Self {
  CustomSmartPointer { data }
  }
}

```

```
19  impl<D> Drop for CustomSmartPointer<D>
20  where
21  D: Debug,
22  {
23    // This will automatically be called when a variable is
       dropped
24    // It cannot be called manually
25    fn drop(&mut self) {
26    println!("Dropping CustomSmartPointer with data `{:?}`",
       self.data);
27    }
28  }
29
30  fn main() {
31    let a = CustomSmartPointer::new("A");
32    let b = CustomSmartPointer::new("B");
33    let c = CustomSmartPointer::new("C");
34    let d = CustomSmartPointer::new("D");
35
36    // The next line would cause a compiler error,
37    // as destructors cannot be explicitely called
38    // c.drop();
39
40    // The correct way to drop variables early is the following:
41    std::mem::drop(c);
42  }
```

How it works...

This example, adapted with slight changes from the second edition of the Rust book (`https://doc.rust-lang.org/book/second-edition/`), shows you how to start the implementation of a custom smart pointer. In our case, all it does is print the `Debug` information of the data stored when its dropped [26]. We do this by implementing the `Drop` trait with its single `drop` function [25], which the compiler automatically calls whenever a variable is dropped. All smart pointers are implemented this way.

The moment of a variable drop will nearly always be when it leaves its scope. For this reason, we cannot call the `drop` function directly[38]. The compiler will still call it when it exits its scope, so the cleanup will happen twice, resulting in undefined behavior. If you need to drop a variable early, you can tell the compiler to do so for you by calling `std::mem:drop` on it [41].

Variables that exit their scope are dropped in a **LIFO** way: **Last In, First Out**. That means that the last variable to be declared will be the first one to be dropped. If we allocate the variables `a`, `b`, `c`, and `d` in exactly that order, they will be dropped in the order `d`, `c`, `b`, `a`. In our example, we drop `c` early[41], so our order becomes `c`, `d`, `b`, `a` instead.

There's more...

Do you want to know how a sophisticated low-level function like `std::mem::drop` is implemented:

```
pub fn drop<T>(_x: T) { }
```

That's right, it does nothing! The reason this works is that it takes `T` by value, moving it into the function. The function does nothing and all its owned variables go out of scope. Hurray for Rust's borrow checker!

See also

- *Boxing data* recipe in `Chapter 5`, *Advanced Data Structures*.
- *Sharing ownership with smart pointers* recipe in `Chapter 5`, *Advanced Data Structures*.

Understanding RAII

We can go one step further than simple destructors. We can create structs that can give the user temporary access to some resource or functionality and automatically revoke it again when the user is done. This concept is called **RAII**, which stands for **Resource Acquisition Is Initialization**. Or, in other words, the validity of a resource is tied to the lifetime of a variable.

How to do it...

Follow these steps:

1. Open the `Cargo.toml` file that was generated earlier for you.

2. In the folder `bin`, create a file called `raii.rs`.

3. Add the following code and run it with `cargo run --bin raii`:

```
use std::ops::Deref;

// This represents a low level, close to the metal OS feature that
// needs to be locked and unlocked in some way in order to be accessed
// and is usually unsafe to use directly
struct SomeOsSpecificFunctionalityHandle;

// This is a safe wrapper around the low level struct
struct SomeOsFunctionality<T> {
  // The data variable represents whatever useful information
  // the user might provide to the OS functionality
  data: T,
  // The underlying struct is usually not savely movable,
  // so it's given a constant address in a box
  inner: Box<SomeOsSpecificFunctionalityHandle>,
}

// Access to a locked SomeOsFunctionality is wrapped in a guard
// that automatically unlocks it when dropped
struct SomeOsFunctionalityGuard<'a, T: 'a> {
  lock: &'a SomeOsFunctionality<T>,
}

impl SomeOsSpecificFunctionalityHandle {
  unsafe fn lock(&self) {
    // Here goes the unsafe low level code
  }
  unsafe fn unlock(&self) {
    // Here goes the unsafe low level code
  }
}
```

Now comes the implementations for the `structs`:

```
impl<T> SomeOsFunctionality<T> {
  fn new(data: T) -> Self {
    let handle = SomeOsSpecificFunctionalityHandle;
      SomeOsFunctionality {
        data,
        inner: Box::new(handle),
      }
  }

  fn lock(&self) -> SomeOsFunctionalityGuard<T> {
    // Lock the underlying resource.
    unsafe {
      self.inner.lock();
    }

    // Wrap a reference to our locked selves in a guard
    SomeOsFunctionalityGuard { lock: self }
  }
}

// Automatically unlock the underlying resource on drop
impl<'a, T> Drop for SomeOsFunctionalityGuard<'a, T> {
  fn drop(&mut self) {
    unsafe {
      self.lock.inner.unlock();
    }
  }
}

// Implementing Deref means we can directly
// treat SomeOsFunctionalityGuard as if it was T
impl<'a, T> Deref for SomeOsFunctionalityGuard<'a, T> {
  type Target = T;

  fn deref(&self) -> &T {
  &self.lock.data
  }
}
```

And finally, the actual usage:

```
fn main() {
  let foo = SomeOsFunctionality::new("Hello World");
  {
    // Locking foo returns an unlocked guard
    let bar = foo.lock();
    // Because of the Deref implementation on the guard,
    // we can use it as if it was the underlying data
    println!("The string behind foo is {} characters long",
     bar.len());

    // foo is automatically unlocked when we exit this scope
  }
  // foo could now be unlocked again if needed
}
```

How it works...

Well, that's a load of complicated code.

Let's start by introducing the structures that participate in this example:

- `SomeOsSpecificFunctionalityHandle` [6] stands for an unspecified feature of your operating system that operates on some data and is presumably unsafe to use directly. We assume this feature locks some resource of the operating system that needs to be unlocked again.
- `SomeOsFunctionality` [9] represents a safe wrapper around the feature, plus some data `T` that might be useful for it.
- `SomeOsFunctionalityGuard` [20] is an RAII guard created by using the `lock` function. When it is dropped, it will automatically unlock the underlying resource. Additionally, it can be directly used as if it was the data `T` itself.

These functions might look a bit abstract, as they don't do anything specific, but instead act on *some* unspecified OS feature. This is because most of the really useful candidates are already present in the standard library—see `File`, `RwLock`, `Mutex`, and so on. What's left are particularly domain-specific use cases when writing low-level libraries or dealing with some special, homemade resource that needs automatic unlocking. When you see yourself writing either, you will appreciate the elegance of RAII.

The implementation of the structs introduces some new concepts that might look a bit confusing if encountered for the first time. In the implementation of `SomeOsSpecificFunctionalityHandle`, we can spot some `unsafe` keywords [25 , 28 and 44] :

```
impl SomeOsSpecificFunctionalityHandle {
    unsafe fn lock(&self) {
        // Here goes the unsafe low level code
    }
    unsafe fn unlock(&self) {
        // Here goes the unsafe low level code
    }
}
...
fn lock(&self) -> SomeOsFunctionalityGuard<T> {
    // Lock the underlying resource.
    unsafe {
        self.inner.lock();
    }
```

```
            // Wrap a reference to our locked selves in a guard
            SomeOsFunctionalityGuard { lock: self }
        }
```

Let's start with the unsafe block [44 to 46]:

```
        unsafe {
            self.inner.lock();
        }
```

The `unsafe` keyword tells the compiler to treat the previous block in a special way. It disables the borrow checker and lets you do all kinds of crazy stuff: dereference raw pointers like in C, modify a mutable static variable, and call unsafe functions. In return, the compiler doesn't give you any guarantees about it either. It might, for instance, access invalid memory, resulting in a **SEGFAULT**. If you want to read more about the `unsafe` keyword, check out its section in the second edition of the official Rust book at `https://doc.rust-lang.org/book/second-edition/ch19-01-unsafe-rust.html`.

Generally speaking, writing unsafe code should be avoided. It is, however, okay to do so when:

- You're writing some code that directly interfaces with the OS and you want to create a safe wrapper around the unsafe parts, which is what we are doing here
- You are absolutely 100% completely certain that what you're doing, in a very specific context, is actually not problematic, contrary to the compiler's opinion

If you're wondering why the `unsafe` block is empty, that's again because we are not using any actual OS resources in this recipe. If you wanted to use any, the code handling them would go in those two empty blocks.

The other use for the `unsafe` keyword is the following [25]:

```
        unsafe fn lock(&self) { ... }
```

This marks the function itself as unsafe, meaning that it can only be called inside `unsafe` blocks. Remember, calling `unsafe` code in a function doesn't make the function automatically unsafe because the function could be a safe wrapper around it.

Let's move on now from our hypothetical low-level implementation of `SomeOsSpecificFunctionalityHandle` to our realistic implementation of its safe wrapper, `SomeOsFunctionality`[33]. Its constructor comes with no surprises (see `Chapter 1`, *Learning the Basics* and the *Using the constructor pattern* recipe if you need a refresher on that):

```
        fn new(data: T) -> Self {
            let handle = SomeOsSpecificFunctionalityHandle;
            SomeOsFunctionality {
                data,
                inner: Box::new(handle),
            }
        }
```

We simply prepare the underlying OS functionality and store it with the user-provided data in our `struct`. We `Box` the handle because, as explained in a comment in the code earlier at lines [13 and 14], the low-level struct interfacing with the OS is often not safe to move. Because we don't want to restrict our user from moving our safe wrapper, however, we make the handle movable by putting in on the heap via a `Box`, which gives it a permanent address. What is then moved is simply the smart pointer pointing to the address. For more about that, read `Chapter 5`, *Advanced Data Structures* and the *Boxing data* recipe.

The actual wrapping takes place in the `lock` method:

```
        fn lock(&self) -> SomeOsFunctionalityGuard<T> {
            // Lock the underlying resource.
            unsafe {
                self.inner.lock();
            }

            // Wrap a reference to our locked selves in a guard
            SomeOsFunctionalityGuard { lock: self }
        }
```

When working with an actual OS feature or custom resource, you'll want to guarantee that `self.inner.lock()` is safe to call in this context before doing so, otherwise, the wrapper won't be safe. This is also where you can do interesting things with `self.data`, which you can potentially use in combination with the resource mentioned.

After locking our stuff up, we return a RAII guard with a reference to our structure [49] that will unlock our resource when it is dropped. Looking at the implementation of `SomeOsFunctionalityGuard`, you can see that we don't need to implement any kind of new function for it. We just need to implement two traits. We begin with `Drop`[54], which you have met in the previous recipe. Implementing it means that we can unlock the resource when the guard is dropped by accessing it through our reference to `SomeOsFunctionality`. Again, make sure to arrange the environment in a way that guarantees that `self.lock.inner.unlock()` is actually safe before calling it.

Since we are basically creating a kind of smart pointer to `data`, we can use the `Deref` trait [64]. Implementing `Deref for B` with a `Target` of `A` allows a reference to `B` to be dereferenced into `A`. Or in other, slightly less accurate words, it lets `B` act as if it was `A`. In our case, implementing `Deref for SomeOsFunctionalityGuard` with a `Target` of `T` means that we can use our guard as if it was the underlying data. Because this can cause great confusion to the user if implemented poorly, Rust advises you to only implement it on smart pointers and nothing else.

Implementing `Deref` is of course not mandatory for the RAII pattern, but can prove pretty useful, as we're going to see in a moment.

Let's look at how we can now use all of our fancy functions:

```
fn main() {
    let foo = SomeOsFunctionality::new("Hello World");
    {
        let bar = foo.lock();
        println!("The string behind foo is {} characters long",
        bar.len());
    }
}
```

The user should never have to use `SomeOsSpecificFunctionalityHandle` directly, as it's `unsafe`. Instead, he can construct an instance of `SomeOsFunctionality`, which he can pass around and store however he wants [73]. Whenever he needs to use the cool feature behind it, he can call `lock` in whatever scope he is in right now, and he will receive a guard that will clean up after him after the work is done [81]. Because he implemented `Deref`, he can use the guard directly as if it was the underlying data. In our example, `data` is a `&str`, so we can use the methods of `str` directly on our guard like we do in line [79] by calling `.len()` on it.

After this little scope ends, our guard calls `unlock` on the resource and, because `foo` independently still lives on, we can continue locking it again however much we want.

There's more...

This example is tailored to be in line with the implementations of `RwLock` and `Mutex`. The only thing missing is an extra layer of indirection that has been omitted to not make this recipe even more complex. `SomeOsSpecificFunctionalityHandle` shouldn't contain actual implementations of `lock` and `unlock`, but instead, pass the calls onto a stored implementation that is specific to whatever OS you're using. For example, say you have a struct, `windows::SomeOsSpecificFunctionalityHandle`, for a Windows-based implementation and a struct, `unix::SomeOsSpecificFunctionalityHandle`, for a Unix-based implementation. `SomeOsSpecificFunctionalityHandle` should then, conditionally, depending on the operating system that is being run, pass its `lock` and `unlock` calls onto the correct implementations. These may have many more features. The Windows one could maybe have a `awesome_windows_thing()` function that might be useful to the unlucky Windows developer that needs it. The Unix implementation could have a `confusing_posix_thing()` function that does some very weird things that only Unix hackers would understand. The important thing is that our `SomeOsSpecificFunctionalityHandle` should represent a common interface of the implementations. In our case, that means that every supported OS has the ability to `lock` and `unlock` the resource in question.

See also

- *Using the constructor pattern* recipe in `Chapter 1`, *Learning the Basics*
- *Boxing data* recipe in `Chapter 5`, *Advanced Data Structures*
- *Accessing resources in parallel with RwLocks* recipe in `Chapter 7`, *Parallelism and Rayon*

7
Parallelism and Rayon

In this chapter, we will cover the following recipes:

- Parallelizing iterators
- Running two operations together
- Sending data across threads
- Sharing resources in multithreaded closures
- Accessing resources in parallel with RwLocks
- Atomically accessing primitives
- Putting it all together in a connection handler

Introduction

There used to be a time when your code got faster every year automatically, as processors got better and better. But nowadays, as Herb Sutter famously stated, *The Free Lunch Is Over* (`http://www.gotw.ca/publications/concurrency-ddj.htm`). The age of not better, but more numerous processor cores arrived a long time ago. Not all programming languages are well suited for this radical change towards omnipresent concurrency.

Rust was designed with exactly this problem in mind. Its borrow checker makes sure that most concurrent algorithms work fine. It goes even further: your code won't even compile if it's not parallelizable, even if you don't yet use more than one thread. Because of these unique guarantees, one of Rust's main selling points has been dubbed *fearless concurrency*.

And we are about to find out why.

Parallelizing iterators

Wouldn't it be cool to have a magic button that allowed you to just make any algorithm parallel, without you doing anything? Well, as long as your algorithm uses iterators, `rayon` is exactly that!

How to do it...

1. Create a Rust project to work on during this chapter with `cargo new chapter-seven`.
2. Navigate into the newly-created `chapter-seven` folder. For the rest of this chapter, we will assume that your command line is currently in this directory.
3. Open the `Cargo.toml` file that has been generated for you.
4. Under `[dependencies]`, add the following line:

   ```
   rayon = "1.0.0"
   ```

   If you want, you can go to `rayon`'s crates.io page (`https://crates.io/crates/rayon`) to check for the newest version and use that one instead.

5. Inside the `src` folder, create a new folder called `bin`.
6. Delete the generated `lib.rs` file, as we are not creating a library.
7. In the `src/bin` folder, create a file called `par_iter.rs`.
8. Add the following code and run it with `cargo run --bin par_iter`:

   ```
   extern crate rayon;
   use rayon::prelude::*;

   fn main() {
     let legend = "Did you ever hear the tragedy of Darth Plagueis
     The Wise?";
     let words: Vec<_> = legend.split_whitespace().collect();

    // The following will execute in parallel,
    // so the exact order of execution is not foreseeable
    words.par_iter().for_each(|val| println!("{}", val));

     // par_iter can do everything that a normal iterator does, but
     // in parallel. This way you can easily parallelize any
       algorithm
     let words_with_a: Vec<_> = words
   ```

```
15       .par_iter()
16       .filter(|val| val.find('a').is_some())
17       .collect();
18
19     println!(
20       "The following words contain the letter 'a': {:?}",
21        words_with_a
22     );
23  }
```

How it works...

`rayon` implements the trait `ParallelIterator` for every type that implements its standard library equivalent `Iterator`, which we got to know in `Chapter 2`, *Working with Collections; Access Collections as iterators*. In fact, you can use all the knowledge from said recipe again here. The methods provided by the `ParallelIterator` trait are nearly the same as the ones provided by `Iterator`, so in virtually all cases where you notice an iterator operation taking too long and bottlenecking you, you can simply replace `.iter()` with `.par_iter()` [10]. Similarly, for moving iterators, you can use `.into_par_iter()` instead of `.into_iter()`.

`rayon` handles all the tedious work for you, as it automatically distributes the work evenly between all of your available cores. Just keep in mind that despite this magic, you're still dealing with parallelism here, so you have no guarantees about the order in which the items in your iterator are going to be handled, as evidenced by line [10], which will print in a different order each time you execute the program:

```
words.par_iter().for_each(|val| println!("{}", val));
```

See also

- *Access collections as iterators* recipe in `Chapter 2`, *Working with Collections*

Running two operations together

The parallel iterators from the last recipe are internally built upon a more fundamental function, `rayon::join`, which takes two closures and *potentially* runs them in parallel. This way, even the balance of performance gain versus the overhead of spawning a thread has been done for you.

If you have an algorithm that doesn't use iterators but still consists of some clearly separated parts that could benefit from running concurrently, consider using `rayon::join` for that.

How to do it...

1. Open the `Cargo.toml` file that was generated earlier for you.
2. If you didn't do so in the last recipe, under `[dependencies]`, add the following line:

   ```
   rayon = "1.0.0"
   ```

3. If you want, you can go to `rayon`'s crates.io page (`https://crates.io/crates/rayon`) to check for the newest version and use that one instead.
4. In the folder `bin`, create a file called `join.rs`.
5. Add the following code and run it with `cargo run --bin join`:

```
extern crate rayon;

 #[derive(Debug)]
 struct Rectangle {
   height: u32,
   width: u32,
 }

 impl Rectangle {
    fn area(&self) -> u32 {
       self.height * self.width
    }
    fn perimeter(&self) -> u32 {
       2 * (self.height + self.width)
    }
  }
```

```
17
18  fn main() {
19    let rect = Rectangle {
20       height: 30,
21       width: 20,
22    };
23    // rayon::join makes closures run potentially in parallel and
24    // returns their returned values in a tuple
25    let (area, perimeter) = rayon::join(|| rect.area(), ||
      rect.perimeter());
26    println!("{:?}", rect);
27    println!("area: {}", area);
28    println!("perimeter: {}", perimeter);
29
30    let fib = fibonacci(6);
31    println!("The sixth number in the fibonacci sequence is {}",
      fib);
32  }
33
34  fn fibonacci(n: u32) -> u32 {
35    if n == 0 || n == 1 {
36     n
37    } else {
38      // rayon::join can really shine in recursive functions
39      let (a, b) = rayon::join(|| fibonacci(n - 1), ||
fibonacci(n
          - 2));
40       a + b
41    }
42  }
```

How it works...

`rayon::join` is pretty simple. It takes two closures, potentially runs them in parallel, and returns their returned values in a tuple [25]. Wait a second, did we just say *potentially*? Isn't it always better to run things in parallel?

Nope, at least not always. Sure, if you really care about things running together at all times without blocking, say a GUI and its underlying I/O where you definitely don't want the mouse cursor to freeze when opening a file, you always need to have all processes running in their own thread. But most applications for concurrency don't have this requirement. A big part of what makes concurrency so important is its ability to run code that would normally run sequentially (that is, one line after another) in parallel if required. Notice the choice of words here—*code that would normally run sequentially*. These kinds of algorithms do not inherently need concurrency, but they might get a boost out of it. Now comes the *potential* part—firing up a thread might not be worth it.

To understand why, let's look at the hardware side of things. We are not going to dive too deep into this territory because:

a) the fact that you're reading this book makes me think you're more of a software person and b) the exact mechanisms of CPUs tend to change very rapidly nowadays and we don't want the information provided here to be outdated in a year.

Your CPU divides its work among its *cores*. A core is the basic computation unit of the CPU. If the device you're reading this on is not made out of paper and younger than two decades, it most probably contains multiple cores. These kinds of cores are called *physical* and can work on different things at the same time. A physical core itself also has ways to perform multiple jobs. Some can divide themselves into multiple *logical* cores, splitting work further. For example, an Intel CPU can use *hyper-threading*, which means that if a program only uses the integer addition unit of a physical core, a virtual core might start working on the floating points addition unit for another program until the first one is done.

If you don't care about the available amount of cores and simply start new threads without limit, the operating system will start creating threads that don't actually run concurrently, because it ran out of cores. In this case, it will perform *context switching*, which means that it stores the current state of the thread, pauses it, works on another thread for a split second, and then resumes the thread again. As you can imagine, this costs quite some resources.

This is why if it's not vital to run two things in parallel, you should first check if there are any cores *idle* (that is, available) in the first place. Because `rayon::join` does this check for you; among other things, it will only run the two closures in parallel if it's actually worth it to do so. If you need to do this work yourself, check out the `num_cpus` crate (`https://crates.io/crates/num_cpus`).

By the way, the parallel iterators from the last recipe go even further: If the amount of elements and work in them is so small that it would cost more to initiate a new thread for them than to run it sequentially, they will automatically forego concurrency for you.

There's more...

The underlying mechanism of `rayon` is *work stealing*. This means that when we call the following function, the current thread will immediately start working on `a` and place `b` in a queue:

```
rayon::join(a, b);
```

Meanwhile, whenever a core is idle, `rayon` will let it work on the next task in the queue. The new thread then *steals* the task from the others. In our case, that would be `b`. If `a` happens to finish before `b`, the main thread will look into the queue and try to steal work as well. The queue can contain more than two items if `rayon::join` is called multiple times in a recursive function.

The author of `rayon`, Niko Matsakis, wrote down the following pseudo Rust code to illustrate this principle in his introductory blog post at `http://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust/`:

```
fn join<A,B>(oper_a: A, oper_b: B)
    where A: FnOnce() + Send,
          B: FnOnce() + Send,
{
    // Advertise `oper_b` to other threads as something
    // they might steal:
    let job = push_onto_local_queue(oper_b);
    // Execute `oper_a` ourselves:
    oper_a();
    // Check whether anybody stole `oper_b`:
    if pop_from_local_queue(oper_b) {
        // Not stolen, do it ourselves.
        oper_b();
    } else {
        // Stolen, wait for them to finish. In the
        // meantime, try to steal from others:
        while not_yet_complete(job) {
            steal_from_others();
        }
        result_b = job.result();
    }
}
```

By the way, the recursive Fibonacci implementation provided in this example [34] is easy to look at and illustrates the point of using `rayon::join`, but is also really, really inefficient. To learn why, and how to improve on it, check out the `Chapter 10`, *Using Experimental Nightly Features*; *Benchmarking your code.*

See also

- *Benchmarking your code* recipe in `Chapter 10`, *Using Experimental Nightly Features*

Sharing resources in multithreaded closures

It's time to look at parallelism at a lower level, without any crates to help us. We will now check out how to share a resource across threads so that they all can work with the same object. This recipe will also serve as a refresher on manually creating threads, in case it's been a while since you learned about it.

How to do it...

1. In the folder `bin`, create a file called `sharing_in_closures.rs`.
2. Add the following code and run it with `cargo run --bin sharing_in_closures`:

```
use std::thread;
use std::sync::Arc;

fn main() {
  // An Arc ("Atomically Reference Counted") is used the exact
  // same way as an Rc, but also works in a parallel context
  let some_resource = Arc::new("Hello World".to_string());

  // We use it to give a new thread ownership of a clone of the
     Arc
  let thread_a = {
    // It is very common to give the clone the same name as the
       original
    let some_resource = some_resource.clone();
    // The clone is then moved into the closure:
    thread::spawn(move || {
      println!("Thread A says: {}", some_resource);
     })
  };
  let thread_b = {
      let some_resource = some_resource.clone();
      thread::spawn(move || {
           println!("Thread B says: {}", some_resource);
          })
   };

  // .join() blocks the main thread until the other thread is done
  thread_a.join().expect("Thread A panicked");
  thread_b.join().expect("Thread B panicked");
  }
```

How it works...

A fundamental building block of parallelism in Rust is the `Arc`, which stands for **Atomically Reference Counted**. Functionally, it works the same way as an `Rc`, which we have looked at in `Chapter 5`, *Advanced Data Structures*; *Sharing ownership with smart pointers*. The only difference is that the reference counting is done using *atomic primitives*, which are versions of primitive data types like `usize` that have well-defined parallel interactions. This has two consequences:

- An `Arc` is slightly slower than an `Rc`, as the reference counting involves a bit more work
- An `Arc` can be used safely across threads

The constructor of `Arc` looks the same as `Rc`[7]:

```
let some_resource = Arc::new("Hello World".to_string());
```

This creates an `Arc` over a `String`. A `String` is a `struct` that is not inherently saved to be manipulated across threads. In Rust terms, we say that `String` is not `Sync` (more about that later in the recipe *Atomically access primitives*).

Now let's look at how a thread is initialized. `thread::spawn()` takes a closure and executes it in a new thread. Because this is done in parallel, the main thread doesn't wait until the thread is done; it continues working right after its creation.

The following creates a thread that prints out the content of `some_resource` and gives us a handle to that thread called `thread_a`[10]:

```
let thread_a = {
    let some_resource = some_resource.clone();
    thread::spawn(move || {
        println!("Thread A says: {}", some_resource);
    })
};
```

Afterward (or at the same time), we do the exact same thing in a second thread called `thread_b`.

To understand why we need an `Arc` and can't just pass the resource directly to the closure, let's take a closer look at how closures work.

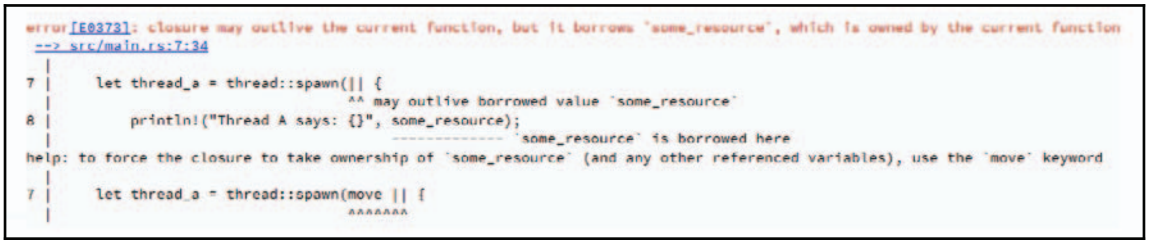
Closures in Rust can only operate on three kinds of variables:

- Arguments passed to them
- `static` variables (variables with the `'static` lifetime; see `Chapter 5`, *Advanced Data Structures*; *Creating lazy static objects*)
- Variables it owns, either by creating them or by moving them into the closure

With this in mind, let's look at the most simplistic approach an inexperienced Rust programmer might take:

```
let thread_a = thread::spawn(|| {
    println!("Thread A says: {}", some_resource);
});
```

If we try to run this, the compiler tells us the following:



```
error[E0373]: closure may outlive the current function, but it borrows `some_resource`, which is owned by the current function
 --> src/main.rs:7:34
  |
7 |     let thread_a = thread::spawn(|| {
  |                                  ^^ may outlive borrowed value `some_resource`
8 |         println!("Thread A says: {}", some_resource);
  |                                       ------------- `some_resource` is borrowed here
help: to force the closure to take ownership of `some_resource` (and any other referenced variables), use the `move` keyword
  |
7 |     let thread_a = thread::spawn(move || {
  |                                  ^^^^^^^
```

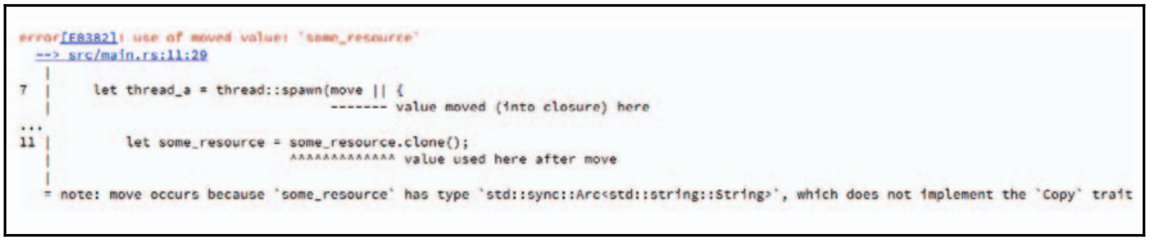
Seems like it doesn't like our usage of `some_resource`. Look at the rules for variable usage in closures again:

- `some_resource` has not been passed as an argument
- It is not `static`
- It was neither created in the closure nor moved into it

But what does *closure may outlive the current function* mean? Well, because closures can be stored in a normal variable, they can be returned from a function. Imagine now if we programmed a function that created a variable called `some_resource`, used it inside a closure, and returned it. Since the function owns `some_resource`, it would be dropped while returning the closure, making any reference to it invalid. We don't want any invalid variables, so the compiler stops us from potentially enabling them. Instead, it suggests moving the ownership of `some_resource` into the closure by using the `move` keyword. Let's try that:

```
let thread_a = thread::spawn(move || {
    println!("Thread A says: {}", some_resource);
});
```

The compiler responds with this:



```
error[E0382]: use of moved value: `some_resource`
  --> src/main.rs:11:29
   |
7  |     let thread_a = thread::spawn(move || {
   |                                  ------- value moved (into closure) here
...
11 |         let some_resource = some_resource.clone();
   |                             ^^^^^^^^^^^^^ value used here after move
   |
   = note: move occurs because `some_resource` has type `std::sync::Arc<std::string::String>`, which does not implement the `Copy` trait
```

Because we moved `some_resource` into the closure inside of `thread_a`, `thread_b` can no longer use it! The solution is to create a clone of the reference to `some_resource` and only move the clone into the closure:

```
let some_resource_clone = some_resource.clone();
let thread_a = thread::spawn(move || {
    println!("Thread A says: {}", some_resource_clone);
});
```

This now runs perfectly fine, but it looks a bit weird, as we are now carrying the mental baggage of the knowledge that the resource we're dealing with is, in fact, a `clone`. This can be solved in a more elegant way by putting the clone into a new scope, where it can have the same name as the original, leaving us with the final version of our code:

```
        let thread_a = {
            let some_resource = some_resource.clone();
            thread::spawn(move || {
                println!("Thread A says: {}", some_resource);
            })
        };
```

Looks way clearer, doesn't it? This way of passing `Rc` and `Arc` variables to a closure is a well-known Rust idiom that we are going to use in all other recipes of the chapter from here on out.

The last thing we are going to do in this recipe is join the two threads by calling `.join()` on them [26 and 27]. Joining a thread means blocking the current thread until the joined thread is done with its work. It's called like that because we *join the two threads of our program back into a single one*. It helps to visually imagine actual sewing threads when thinking about this concept.

We join them before the end of the program, as otherwise, we would have no guarantee that they would actually run all the way through before our program quits. Generally speaking, you should `join` your threads when you need their results and can't wait for them any longer, or they're about to be dropped otherwise.

See also

- *Sharing ownership with smart pointers* and *Creating lazy static objects* recipes in `Chapter 5`, *Advanced Data Structures*

Sending data across threads

So far, we've looked at threads that work independently. Now, let's take a look at intertwined threads that need to share data. This situation is common when setting up servers, as the thread receiving client messages is usually not the same as the one that actually handles and responds to the client input. Rust gives us the concept of *channels* as a solution. A channel is split into a *sender* and a *receiver* which can share data across threads.

How to do it...

1. Open the `Cargo.toml` file that was generated earlier for you.
2. Under `[dependencies]`, add the following line:

   ```
   rand = "0.4.2"
   ```

3. If you want, you can go to rand's crates.io page (`https://crates.io/crates/rand`) to check for the newest version and use that one instead.
4. In the folder `bin`, create a file called `channels.rs`.
5. Add the following code and run it with `cargo run --bin channels`:

```
extern crate rand;

use rand::Rng;
use std::thread;
// mpsc stands for "Multi-producer, single-consumer"
use std::sync::mpsc::channel;

fn main() {
   // channel() creates a connected pair of a sender and a
     receiver.
   // They are usually called tx and rx, which stand for
   // "transmission" and "reception"
   let (tx, rx) = channel();
   for i in 0..10 {
       // Because an mpsc channel is "Multi-producer",
       // the sender can be cloned infinitely
       let tx = tx.clone();
       thread::spawn(move || {
          println!("sending: {}", i);
          // send() pushes arbitrary data to the connected
           receiver
          tx.send(i).expect("Disconnected from receiver");
       });
   }
   for _ in 0..10 {
      // recv() blocks the current thread
      // until a message was received
      let msg = rx.recv().expect("Disconnected from sender");
      println!("received: {}", msg);
   }

```

```
  let (tx, rx) = channel();
  const DISCONNECT: &str = "Goodbye!";
  // The following thread will send random messages
  // until a goodbye message was sent
  thread::spawn(move || {
      let mut rng = rand::thread_rng();
      loop {
        let msg = match rng.gen_range(0, 5) {
            0 => "Hi",
            1 => DISCONNECT,
            2 => "Howdy there, cowboy",
            3 => "How are you?",
            4 => "I'm good, thanks",
            _ => unreachable!(),
        };
        println!("sending: {}", msg);
        tx.send(msg).expect("Disconnected from receiver");
        if msg == DISCONNECT {
          break;
        }
     }
 });

 // An iterator over messages in a receiver is infinite.
 // It will block the current thread until a message is available
 for msg in rx {
      println!("received: {}", msg);
  }
}
```

How it works...

As explained in the comments of the code, calling `std::sync::mpsc::channel()` generates a tuple consisting of a `Sender` and a `Receiver`, which are conventionally called `tx` for *transmission* and `rx` for *reception* [12].

This naming convention doesn't come from Rust, but has been a standard in the telecommunications industry since at least 1960 when the RS-232 (**Recommended Standard 232**) was introduced, detailing how computers and modems should communicate with each other.

These two halves of the same channel can communicate with each other independently of the current thread they're in. The module's name, `mspc`, tells us that this channel is a `Multi-producer, single-consumer` channel, which means that we can `clone` our sender as many times as we want. We can use this fact to our advantage when dealing with closures [16 to 21]:

```
for i in 0..10 {
    let tx = tx.clone();
    thread::spawn(move || {
        println!("sending: {}", i);
        tx.send(i).expect("Disconnected from receiver");
    });
}
```

We do not need to wrap our sender in an `Arc`, because it natively supports arbitrary cloning! Inside of the closure you can see the sender's main functionality. The `send()` method sends data across threads to the receiver. It will return an error if the receiver is not available anymore, as in when it is dropped too early. In this thread here, we will simply send the numbers `0` to `9` concurrently to the receiver. One thing to note is that because a channel's halves are statically typed, they are only going to be able to send one specific data type around. If the first thing you send is an `i32`, your channel will only work with `i32`. If you send a `String`, it will be a `String` channel.

On to the receiver we go [23 to 28]:

```
for _ in 0..10 {
    let msg = rx.recv().expect("Disconnected from sender");
    println!("received: {}", msg);
}
```

The `recv()` method, which stands for *receive*, blocks the current thread until a message has arrived. Similar to its counterpart, it returns an error if the sender is unavailable. Because we know that we only sent 10 messages, we only call it 10 times. There is no need to explicitly `join` the threads we created for the sender, because `recv()` blocked the main thread until no more messages were left, which means that the sender finished sending all they had to send, that is, all the threads already finished their job. This way, we already joined them.

But in real life, you do not have a guarantee about the amount of times a client will send information to you. For a more realistic demonstration, we will now create a thread that sends random messages [37] to the receiver until it finally has enough and quits by sending `"Goodbye!"` [48]. Note how we created a new channel pair, as the old one was set to the type `i32` because integer literals such as `1` or `2` are treated as `i32` by default.

While the sending code looks almost identical to the one before, the receiving end looks a bit different [55 to 57]:

```
        for msg in rx {
            println!("received: {}", msg);
        }
```

As you can see, a receiver can be iterated over. It behaves like an infinite iterator over all messages that will ever come, blocking when waiting for a new one, similar to calling `recv()` in a loop. The difference is that the iteration will automatically stop when the sender is unavailable. Because we terminate the sending thread when it sends `"Goodbye!"` [48], this iteration over the receiver will also stop when receiving it, as the sender will have been dropped at that point. Because this means that we have a guarantee about the sending thread being finished, we do not need to join it.

There's more...

A channel is not `Sync` and, as such, can only be moved across channels but not shared between them. If you need the channel to be `Sync` you can use `std::sync::mpsc::sync_channel`, which blocks when a buffer of unanswered messages is full. An example for when this might be necessary is when a web framework offers to manage your types but only works with `Sync` structs. You can read more on `Sync` in the recipe *Atomically access primitives*.

The `mpsc` channels, as their name suggests, allow many senders but only one receiver. Most of the time, this will be good enough, but if you find yourself needing the exact opposite, as in one sender and multiple receivers, check out Sean McArthur's `spmc` crate at `https://crates.io/crates/spmc`, which provides you with `Single-producer, multi-consumer` channels.

See also

- *Access collections as Iterator* recipe in `Chapter 2`, *Working with Collections*

Accessing resources in parallel with RwLocks

When we shared resources with an `Arc`, we only did so immutably. The moment we want our threads to mutate our resources, we need to use some kind of locking mechanism to secure the golden rule of parallelism: multiple readers or one writer. `RwLock` enforces just that rule across threads and blocks them if they violate the rule.

How to do it...

1. In the folder `bin`, create a file called `rw_lock.rs`.
2. Add the following code and run it with `cargo run --bin rwlock`:

```
use std::sync::{Arc, RwLock};
use std::thread;

fn main() {
// An RwLock works like the RefCell, but blocks the current
// thread if the resource is unavailable
let resource = Arc::new(RwLock::new("Hello
    World!".to_string()));

// The reader_a thread will print the current content of
// our resource fourty times
let reader_a = {
let resource = resource.clone();
thread::spawn(move || {
for _ in 0..40 {
// Lock resource for reading access
let resource = resource
.read()
.expect("Failed to lock resource for reading");
println!("Reader A says: {}", resource);
}
})
};

// The reader_b thread will print the current content of
// our resource fourty times as well. Because RwLock allows
// multiple readers, it will execute at the same time as
     reader_a
let reader_b = {
let resource = resource.clone();
thread::spawn(move || {
for _ in 0..40 {
// Lock resource for reading access
let resource = resource
.read()
.expect("Failed to lock resource for reading");
println!("Reader B says: {}", resource);
}
})
};

// The writer thread will modify the resource ten times.
// Because RwLock enforces Rust's access rules
// (multiple readers xor one writer), this thread will wait until
// thread_a and thread_b are not using the resource and then block
// them both until its done.
let writer = {
let resource = resource.clone();
thread::spawn(move || {
for _ in 0..10 {
// Lock resource for writing access
let mut resource = resource
.write()
.expect("Failed to lock resource for writing");

resource.push('!');
}
})
};

reader_a.join().expect("Reader A panicked");
reader_b.join().expect("Reader B panicked");
writer.join().expect("Writer panicked");
}
```

How it works...

The `RwLock` is the parallel equivalent of the `RefCell` we worked with in `Chapter 5`, *Advanced Data Structures; Working with interior mutability*. The big difference is that, while `RefCell` panics on a violation of Rust's ownership concept, `RwLock` simply blocks the current thread until the violation is over.

The analog of the `borrow()` method of `RefCell` is `read()` [17], which locks the resource for immutable access. The analog of `borrow_mut()` is `write()` [51], which locks the resource for mutable access. Makes sense, doesn't it?

These methods return a `Result`, which tells us whether the thread is *poisoned*. The meaning of poisoning is different for every lock. In an `RwLock`, it means that the thread that locked the resource for `write` access panicked. This way, you can react to panics in other threads and treat them in some way. One example where this can be useful is sending some logs to a server before a crash happens in order to diagnose the problem. In most cases, though, it will be okay if you simply `panic` along, as a `panic` usually stands for a critical failure that cannot be mended.

In our example, we demonstrate the concept by setting up two threads that request `read` access: `reader_a` [11] and `reader_b` [27]. Because an `RwLock` allows multiple readers, they will concurrently print out the value of our resource [19 and 35]. In the meantime, `writer` [45] tries to lock the resource for `write` access. It will have to wait until both `reader_a` and `reader_b` are currently not using the resource. By the same rules, when the `writer` gets their turn and mutates the resource [54], both `reader_a` and `reader_b` have to wait until it's done.

Because all of this happens roughly at the same time, every execution of this example is going to give you slightly different results. I encourage you to run the program multiple times and compare the output.

There's more...

Despite its nice usability, `RwLock` is still no silver bullet for all concurrent problems. There is a concept in concurrent programming called *deadlock*. It arises when two processes wait for the unlocking of resources that the other holds. This will lead to them waiting forever, as no one is ready to take the first step. Kind of like teenagers in love. An example of this would be a `writer_a` requesting access to a file that `writer_b` holds. `writer_b`, in the meantime, needs some kind of user information from `writer_a` before he can give up the file lock. The best way to avoid this problem is to keep it in the back of your mind and remember it when you're about to create processes that depend on each other.

Another lock that is fairly popular in other languages is the `Mutex`, which Rust also provides under `std::sync::Mutex`. When it locks resources, it treats every process like a writer, so no two threads will *ever* be able to work at the same time with a `Mutex`, even if they don't mutate the data. We are going to create a very simple implementation of it ourselves in the next recipe.

See also

- *Working with interior mutability* recipe in `Chapter 5`, *Advanced Data Structures*

Atomically accessing primitives

When reading about all of these parallel structures, you might have wondered how they are implemented. In this recipe, we are going to take a look under the hood and learn about the most basic parallel data types, which are called *atomics*. We are going to do this by implementing our very own `Mutex`.

How to do it...

1. In the folder `bin`, create a file called `atomic.rs`.
2. Add the following code and run it with `cargo run --bin atomic`:

```
use std::sync::Arc;
use std::sync::atomic::{AtomicBool, AtomicUsize, Ordering,
ATOMIC_BOOL_INIT, ATOMIC_USIZE_INIT};
use std::thread;
use std::ops::{Deref, DerefMut};
use std::cell::UnsafeCell;

fn main() {
// Atomics are primitive types suited for
// well defined concurrent behaviour
let some_number = AtomicUsize::new(0);
// They are usually initialized by copying them from
// their global constants, so the following line does the same:
let some_number = ATOMIC_USIZE_INIT;

```

```
// load() gets the current value of the atomic
// Ordering tells the compiler how exactly to handle the interactions
// with other threads. SeqCst ("Sequentially Consistent") can always be used
// as it results in the same thing as if no parallelism was involved
let curr_val = some_number.load(Ordering::SeqCst);
println!("The current value of some_number is {}", curr_val);

// store() sets the variable
some_number.store(123, Ordering::SeqCst);
let curr_val = some_number.load(Ordering::SeqCst);
println!("The current value of some_number is {}", curr_val);

// swap() sets the variable and returns the old value
let old_val = some_number.swap(12_345, Ordering::SeqCst);
let curr_val = some_number.load(Ordering::SeqCst);
println!("The old value of some_number was {}", old_val);
println!("The current value of some_number is {}", curr_val);

// compare_and_swap only swaps the variable if it
// is currently equal to the first argument.
// It will always return the old variable
let comparison = 12_345;
let new_val = 6_789;
let old_val = some_number.compare_and_swap(comparison, new_val, Ordering::SeqCst);
if old_val == comparison {
println!("The value has been updated");
}

// The previous atomic code is equivalent to
// the following sequential code
let mut some_normal_number = 12_345;
let old_val = some_normal_number;
if old_val == comparison {
some_normal_number = new_val;
println!("The value has been updated sequentially");
}

// fetch_add() and fetch_sub() add/subtract a number from the value,
// returning the old value
let old_val_one = some_number.fetch_add(12, Ordering::SeqCst);
let old_val_two = some_number.fetch_sub(24, Ordering::SeqCst);
let curr_val = some_number.load(Ordering::SeqCst);
println!(
"some_number was first {}, then {} and is now {}",
old_val_one, old_val_two, curr_val
);

// fetch_or() performs an "or" ("||") operation on the variable
       and
// an argument and sets the variable to the result. It then
       returns the old value.
// For the other logical operations, fetch_and(), fetch_nand()
       and fetch_xor also exist
let some_bool = ATOMIC_BOOL_INIT;
let old_val = some_bool.fetch_or(true, Ordering::SeqCst);
let curr_val = some_bool.load(Ordering::SeqCst);
println!("({} || true) is {}", old_val, curr_val);

// The following is a demonstration of our own Mutex
       implementation,
// based on an AtomicBool that checks if it's locked or not
let naive_mutex = Arc::new(NaiveMutex::new(1));

// The updater thread will set the value in the mutex to 2
let updater = {
let naive_mutex = naive_mutex.clone();
thread::spawn(move || {
let mut val = naive_mutex.lock();
*val = 2;
})
};

// The updater thread will print the value in the mutex
let printer = {
let naive_mutex = naive_mutex.clone();
thread::spawn(move || {
let val = naive_mutex.lock();
println!("The value in the naive mutex is: {}", *val);
})
};

// The exact order of execution is unpredictable,
// but our mutex guarantees that the two threads will
// never access the data at the same time
updater.join().expect("The updater thread panicked");
printer.join().expect("The printer thread panicked");
}
```

3. Now comes the implementation of our very own homemade mutex:

```
// NaiveMutex is an easy, albeit very suboptimal,
// implementation of a Mutex, similar to std::sync::Mutex
// A mutex is a lock that only allows one thread to access a ressource at all times
pub struct NaiveMutex<T> {
locked: AtomicBool,
// UnsafeCell is the underlying struct of every
// internally mutable container such as ours
data: UnsafeCell<T>,
}

// This is a RAII guard, identical to the one from the last chapter
pub struct NaiveMutexGuard<'a, T: 'a> {
naive_mutex: &'a NaiveMutex<T>,
}

impl<T> NaiveMutex<T> {
pub fn new(data: T) -> Self {
NaiveMutex {
locked: ATOMIC_BOOL_INIT,
data: UnsafeCell::new(data),
}
}

pub fn lock(&self) -> NaiveMutexGuard<T> {
// The following algorithm is called a "spinlock", because it keeps
// the current thread blocked by doing nothing (it keeps it "spinning")
while self.locked.compare_and_swap(false, true, Ordering::SeqCst) {}
NaiveMutexGuard { naive_mutex: self }
}
}

// Every type that is safe to send between threads is automatically
// safe to share between threads if wrapped in our mutex, as it
// guarantees that no threads will access it ressource at the same time
unsafe impl<T: Send> Sync for NaiveMutex<T> {}

// Automatically unlock the mutex on drop
impl<'a, T> Drop for NaiveMutexGuard<'a, T> {
```

```
fn drop(&mut self) {
self.naive_mutex.locked.store(false, Ordering::SeqCst);
}
}

// Automatically dereference to the underlying data
impl<'a, T> Deref for NaiveMutexGuard<'a, T> {
type Target = T;
fn deref(&self) -> &T {
unsafe { &*self.naive_mutex.data.get() }
}
}

impl<'a, T> DerefMut for NaiveMutexGuard<'a, T> {
fn deref_mut(&mut self) -> &mut T {
unsafe { &mut *self.naive_mutex.data.get() }
}
}
```

How it works...

As of the time of writing, there are four `atomic` types in the standard library under the `std::sync::atomic` module: `AtomicBool`, `AtomicIsize`, `AtomicUsize`, and `AtomicPtr`. Each one of them represents a primitive type, namely `bool`, `isize`, `usize`, and `*mut`. We are not going to look at the last, which, being a pointer, you will probably only have to deal with when interfacing with programs written in other languages anyways.

In case you haven't encountered `isize` and `usize` before, they are representations of the smallest amount of bytes needed to address any part of the memory of your machine. On 32-bit targets this is 4 bytes, while 64-bit systems will need 8 bytes. `isize` uses those bytes to represent a *signed* number, as in an integer that can be negative. `usize` instead represents an *unsigned* number, which can only be positive but has a lot more capacity for huge numbers in that direction. They are usually used when dealing with collections capacities. For example, `Vec` returns a `usize` when calling its `.len()` method. Additionally, on the nightly toolchain, there are atomic variants of all other concrete integer types like `u8` or `i32`.

The `atomic` versions of our primitives work the same way as their cousins, with one important distinction: they have well-defined behavior when used in parallel environments. All their methods take a parameter of the type `atomic::Ordering`, which stands for which low-level concurrent strategy to use. In this example, we are only going to use `Ordering::SeqCst`, which stands for s*equentially consistent*. This, in turn, means that the behavior is quite intuitive. If some data is stored or modified using this ordering, another thread can see its content after the write as if the two threads ran one after another. Or, in other words, the behavior is *consistent* with that of a *sequential* series of events. This strategy will always work with all parallel algorithms. All other orderings merely relax the constraints on the data involved in order to get some kind of performance benefit.

With this knowledge in hand, you should be able to understand most things done in `main` up to the usage of `NaiveMutex`[72]. Note how some of the `atomic` methods are just different ways of doing the same as with our normal primitives, with the added twist of specifying an ordering and most of them returning the old value. For instance, `some_number.fetch_add(12, Ordering::SeqCst)`, apart from returning the old value of `some_number`, is essentially nothing but `some_number += 12`.

A real use case for atomics comes up in the second part of the example code, where we implement our very own `Mutex`. A mutex, prominently featured in all modern programming languages, is a kind of lock that does not allow *any* two threads to access a resource at the same time. After reading the last recipe, you know that you can imagine a `Mutex` as a kind of `RwLock` that always locks everything in `write` mode.

Let's jump a few lines forward in our code to [102]:

```
    pub struct NaiveMutex<T> {
        locked: AtomicBool,
        data: UnsafeCell<T>,
    }
```

As you can see, we are going to base our `NaiveMutex` on a simple atomic flag, `locked`, which is going to track whether our mutex is available or not. The other member, `data`, holds the underlying resource we are interested in locking. Its type, `UnsafeCell`, is the underlying type of every struct that implements some kind of interior mutability (see `Chapter 5`, *Advanced Data Structures*; *Working with interior mutability*).

The next struct is going to look familiar to you if you've read `Chapter 6`, *Handling Errors*; *Understanding RAII*, as it's an RAII guard with a reference to its parent [110]:

```
    pub struct NaiveMutexGuard<'a, T: 'a> {
        naive_mutex: &'a NaiveMutex<T>,
    }
```

Let's take a look at how we lock a thread:

```
pub fn lock(&self) -> NaiveMutexGuard<T> {
  while self.locked.compare_and_swap(false, true, Ordering::SeqCst) {}
    NaiveMutexGuard { naive_mutex: self }
}
```

Looks a bit weird at first glance, doesn't it? `compare_and_swap` is one of the more complex `atomic` operations. It works as follows:

1. It compares the value of the atomic with the first parameter
2. If they are the same, it stores the second parameter in the atomic
3. Lastly, it returns the value of the atomic from before the function call

Let's apply that to our call:

- `compare_and_swap` checks if `self.locked` contains `false`
- If so, it sets `self.locked` to `true`
- In any case, it will return the old value

If the returned value is `true`, it means our mutex is currently locked. What should our thread do then? Absolutely nothing: `{ }`. Because we call this in a `while` loop, we will continue doing nothing (this is called *spinning*) until the situation changes. This algorithm is called **spinlock**.

When our mutex is finally available, we set its `locked` flag to `true` and return an RAII guard with a reference to our `NaiveMutex`.

This is not how the real `std::sync::Mutex` is implemented. Because exclusively locking a resource is a very basic concurrent task, operating systems natively support it. The `Mutex` implemented in the Rust standard library is still built by the RAII pattern as well, but uses the OS's mutex handles instead of our custom logic. Fun fact—the Windows implementation uses SRWLocks (`https://msdn.microsoft.com/en-us/library/windows/desktop/aa904937(v=vs.85).aspx`), which are Windows's native version of `RwLock`, as they proved to be faster than a native `Mutex`. So, on Windows at least, the two types really are very similar.

The implementation of `NaiveMutexGuard` provides the counterpart of `lock` during its dropping [138]:

```
    fn drop(&mut self) {
        self.naive_mutex.locked.store(false, Ordering::SeqCst);
    }
```

We simply `store` the value `false` in `self.locked` whenever our guard goes out of scope (see `Chapter 6`, *Handling Errors*; *Implementing the Drop trait)*. The next two trait `NaiveMutexGuard` implements are `Deref` and `DerefMut`, which let us call methods of type `T` directly on `NaiveMutexGuard<T>`. They both share nearly the same implementation [146]:

```
    unsafe { &*self.naive_mutex.data.get() }
```

Remember when we said that you'll have to deal with pointers on rare occasions? Well, this is one of those times.

`UnsafeCell` doesn't guarantee any borrowing safety, hence the name and the `unsafe` block. It relies on you to make sure all calls to it are actually safe. Because of this, it gives you a raw mutable pointer, which you can manipulate in any way you want. What we do here is dereference it with `*`, so `*mut T` becomes only `T`. Then we return a normal reference to that with `&`[146]. The only thing different in the implementation of `deref_mut` is that we instead return a mutable reference with `&mut` [152]. All of our `unsafe` calls are guaranteed to follow Rust's ownership principles, as we only allow one scope to borrow our resource anyway.

The last thing required for our `Mutex` implementation is the following line, which we skipped before:

```
    unsafe impl<T: Send> Sync for NaiveMutex<T> {}
```

The `Sync` trait has a pretty small implementation, right? That's because it is a *marker*. It belongs to a family of traits that don't actually do anything themselves but only exist to tell the compiler something about the types that implement them. Another trait in the `std::marker` module is `Send`, which we also use here.

If a type `T` implements `Send`, it tells the world that it is safe to be moved (*sent*) between threads by passing it around as a value instead of a reference. Nearly all types of Rust implement `Send`.

If `T` is `Sync`, it tells the compiler that it is safe to be shared between threads (it behaves in a *synchronized* way) by passing it around per reference, `&T`. This is harder to accomplish than `Send`, but our `NaiveMutex` guarantees that types in it can be shared around, as we only allow one access to its inner type at a time. This is why we implement the `Sync` trait for every `Send` in our mutex. If it's safe to pass it around, it's automatically also safe to share it within `NaiveMutex`.

Back in `main` you can now find some usage examples of our `Mutex`[75 and 84], similar to the examples in the previous recipe.

There's more...

Because `SeqCst` is good enough for most applications and the complexity involved in all other orderings, we are not going to look at any others. Don't be disappointed, however—Rust uses nearly the same `atomic` layout and functionality as C++, so there are plenty of sources to tell you how complex the issue really is. Anthony Williams, author of the well-known book *C++: Concurrency In Action* (`http://www.cplusplusconcurrencyinaction.com/`), uses an entire 45 pages (!) to simply describe all the atomic orderings and how to use them. An additional 44 pages go into showing examples of all of these orderings. Does an average program benefit from this level of dedication? Let's look at the man's own words, with the background knowledge that `std::memory_order_seq_cst` is how C++ calls `SeqCst`:

> *The basic premise is: do not use anything other than* `std::memory_order_seq_cst` *(the default) unless (a) you really* ***really*** *know what you are doing, and can* ***prove*** *that the relaxed usage is safe in all cases, and (b) your profiler demonstrates that the data structure and operations you are intending to use the relaxed orderings with are a bottleneck.*

Source: `https://stackoverflow.com/a/9564877/5903309`

In short, you should wait to learn about the different kinds of orderings until you have a very good reason to use them. This is, by the way, also the approach of Java, which makes all variables marked as `volatile` behave in a sequentially consistent way.

See also

- *Working with interior mutability* recipe in `Chapter 5`, *Advanced Data Structures*
- *Implementing the Drop trait* and *Understanding RAII* recipe in `Chapter 6`, *Handling Errors*

Putting it all together in a connection handler

We have looked at a lot of different practices in isolation now. The true strength of these building blocks, however, comes from combining them. This recipe is going to show you how to combine some of them into a realistic starting point for the connection handling part of a server.

How to do it...

1. In the folder `bin`, create a file called `connection_handler.rs`.
2. Add the following code and run it with `cargo run --bin connection_handler`:

```
use std::sync::{Arc, RwLock};
use std::net::Ipv6Addr;
use std::collections::HashMap;
use std::{thread, time};
use std::sync::atomic::{AtomicUsize, Ordering,
ATOMIC_USIZE_INIT};

// Client holds whatever state your client might have
struct Client {
ip: Ipv6Addr,
}

// ConnectionHandler manages a list of connections
// in a parallelly safe way
struct ConnectionHandler {
// The clients are identified by a unique key
clients: RwLock<HashMap<usize, Client>>,
next_id: AtomicUsize,
}

impl Client {
fn new(ip: Ipv6Addr) -> Self {
Client { ip }
}
}

impl ConnectionHandler {
fn new() -> Self {
ConnectionHandler {
clients: RwLock::new(HashMap::new()),
next_id: ATOMIC_USIZE_INIT,
}
}

fn client_count(&self) -> usize {
self.clients
.read()
.expect("Failed to lock clients for reading")
.len()
}

fn add_connection(&self, ip: Ipv6Addr) -> usize {
let last = self.next_id.fetch_add(1, Ordering::SeqCst);
self.clients
.write()
.expect("Failed to lock clients for writing")
.insert(last, Client::new(ip));
last
}

fn remove_connection(&self, id: usize) -> Option<()> {
self.clients
.write()
.expect("Failed to lock clients for writing")
.remove(&id)
.and(Some(()))
}
}
```

Using our connection handler by simulating connecting and disconnecting clients:

```
fn main() {
let connections = Arc::new(ConnectionHandler::new());

// the connector thread will add a new connection every now and then
let connector = {
let connections = connections.clone();
let dummy_ip = Ipv6Addr::new(0, 0, 0, 0, 0, 0xffff, 0xc00a, 0x2ff);
let ten_millis = time::Duration::from_millis(10);
thread::spawn(move || {
for _ in 0..20 {
connections.add_connection(dummy_ip);
thread::sleep(ten_millis);
}
})
};

// the disconnector thread will remove the third connection at some point
let disconnector = {
let connections = connections.clone();
let fifty_millis = time::Duration::from_millis(50);
thread::spawn(move || {
thread::sleep(fifty_millis);
connections.remove_connection(2);
})
};

// The main thread will print the active connections in a short interval
let five_millis = time::Duration::from_millis(5);
for _ in 0..40 {
let count = connections.client_count();
println!("Active connections: {}", count);
thread::sleep(five_millis);
}

connector.join().expect("The connector thread panicked");
disconnector
.join()
.expect("The disconnector thread panicked");
}
```

How it works...

This recipe doesn't introduce any new modules or concepts. It's here to provide you with a general idea of how to combine all the things you've learned in this recipe in a somewhat realistic context. Specifically, our context consists of code that manages clients that connect with us in some way.

`Client` [8] holds all information relevant to a connection. As a basic example, it currently contains the client's IP address. Other possibilities would be the client's username, location, device, ping, and so on.

The `ConnectionHandler` [14] itself holds a list, more specifically a `HashMap`, of the active connections, indexed by a unique ID. Analogous to that, it also stores the ID for the next connection.

We are using unique IDs instead of a `Vec<Client>` because clients might be able to connect, multiple times, to whatever service we are providing on the same device. The easiest example for this is multiple tabs open in a browser, all accessing the same website. Generally speaking, it is good practice to always hold your data behind unique keys to save yourself from trouble down the road.

The implementations of the structs should be straightforward. Methods that need to modify the `clients` member lock it with `.write()`, all others with `.read()`.

The code used to get a new ID at `add_connection` adds one to `next_id` and returns its last value, as usual for an `atomic`[42]:

```
        let last = self.next_id.fetch_add(1, Ordering::SeqCst);
```

After adding the connection to the `clients`, we return the newly-acquired ID to the caller, so that they can store the ID however they want and reuse it when it's time to kick the client with `remove_connection` [50], which in turn returns an `Option` telling the caller if the removed ID was in the client list in the first place. We do not return the removed `Client` directly because that would reveal unnecessary implementation details to the user of `ConnectionHandler`.

The code in `main` simulates parallel access to the hypothetical service. A bunch of clients connect to our `ConnectionHandler` and some leave again. `thread::sleep` [70, 80 and 90] blocks the current thread for a specified time and is used here to simulate the effect of various events happening at irregular intervals, represented by the different waiting times for each task.

As with the `RwLock` example, this program will have very different output every time you run it, so try it out multiple times.

There's more...

If you need to react to the messages from your clients in a different thread, you can use `channel`, which we looked at earlier in the chapter. One use case for this would be programming an online video game. You'll want to aggregate all input from your players, react to it by simulating your world, and then broadcast local changes to the players, with each of these tasks happening concurrently in a single thread.

8
Working with Futures

In this chapter, we will cover the following recipes:

- Providing futures with a CPU pool and waiting for them
- Implementing error handling for futures
- Combining futures
- Using Streams
- Using Sinks
- Using the oneshot channel
- Returning futures
- Locking resources with BiLocks

Introduction

Futures provide the building blocks for asynchronous computations with zero-cost abstraction. Asynchronous communication is useful for handling timeouts, computing across thread pools, network responses, and any function that does not immediately return a value.

In a synchronous block, the computer would execute each command sequentially after waiting for each command to return a value. If you were to apply the synchronous model when sending an email, you would send the message, stare at your inbox, and wait until you have received a response from your recipient.

Fortunately, life does not work synchronously. After we send an email, we could switch to another application or get off our chair. We can start performing other tasks such as getting the groceries, cooking dinner, or reading a book. Our attention can focus on, and perform, other tasks simultaneously. Periodically, we will check our inbox for a response from our recipient. The process of periodically checking for the new message illustrates the asynchronous model. Unlike humans, computers can check for a new message in our inbox and perform other tasks at the same time.

Rust's futures work by implementing the polling model, which utilizes a central component (for example, a piece of software, hardware devices, and network hosts) to handle status reports from other components. The central, or master, component sends signals to other components repetitively until the master component receives an update, an interruption signal, or the polling event has timed out.

To get a better understanding on how concurrency works within Rust's model, you can view Alex Crichton's concurrency presentations at `https://github.com/alexcrichton/talks`. Throughout our recipes, we will be using the `futures::executor::block_on` function within our main thread to return values. This is intentionally done for demonstrative purposes only. In a real application, you would use `block_on` within another a separate thread and your functions would return some sort of `futures::Future` implementation such as `futures::future::FutureResult`.

At the time of writing, futures is performing a lot of developmental changes throughout its code base. You can view futures' RFCs (Request For Comments) on their official repository at `https://github.com/rust-lang-nursery/futures-rfcs`

Providing futures with a CPU pool and waiting for them

Futures are usually assigned to a `Task`, which gets assigned to an `Executor`. When a task is *awake,* the executor will place the task into a queue, and will call `poll()` on the task until the process has been completed. Futures offer us a few convenient ways to execute tasks:

- Spawn a future task manually with `futures::executor::block_on()`.

- Using `futures::executor::LocalPool`, which is useful for performing many small tasks on a single thread. In our future returns, we would not be required to implement `Send` since we are only involving the task on a single thread. However, you are required to use `futures::executor::spawn_local()` on the `Executor` if you omit the `Send` trait.
- Using `futures::executor::ThreadPool`, which allows us to offload tasks to other threads.

How to do it...

1. Create a Rust project to work on during this chapter with `cargo new futures`.
2. Navigate into the newly-created `futures` folder. For the rest of this chapter, we will assume that your command line is within this directory.
3. Inside the `src` folder, create a new folder called `bin`.
4. Delete the generated `lib.rs` file, as we are not creating a library.
5. Open the `Cargo.toml` file that has been generated.
6. Under `[dependencies]`, add the following lines:

```
futures = "0.2.0-beta"
futures-util = "0.2.0-beta"
```

7. In the `src/bin` folder, create a file called `pool.rs`.
8. Add the following code and run it with `cargo run —bin pool`:

```
extern crate futures;

use futures::prelude::*;
use futures::task::Context;
use futures::channel::oneshot;
use futures::future::{FutureResult, lazy, ok};
use futures::executor::{block_on, Executor, LocalPool,
ThreadPoolBuilder};

use std::cell::Cell;
use std::rc::Rc;
use std::sync::mpsc;
use std::thread;
use std::time::Duration;
```

Let's add our constants, enums, structures, and trait implementations:

```
#[derive(Clone, Copy, Debug)]
enum Status {
  Loading,
  FetchingData,
  Loaded,
}

#[derive(Clone, Copy, Debug)]
struct Container {
  name: &'static str,
  status: Status,
  ticks: u64,
}

impl Container {
  fn new(name: &'static str) -> Self {
    Container {
      name: name,
      status: Status::Loading,
      ticks: 3,
    }
  }

  // simulate ourselves retreiving a score from a remote database
  fn pull_score(&mut self) -> FutureResult<u32, Never> {
    self.status = Status::Loaded;
    thread::sleep(Duration::from_secs(self.ticks));
    ok(100)
  }
}

impl Future for Container {
  type Item = ();
  type Error = Never;

  fn poll(&mut self, _cx: &mut Context) -> Poll<Self::Item, Self::Error> {
    Ok(Async::Ready(()))
  }
}
const FINISHED: Result<(), Never> = Ok(());

fn new_status(unit: &'static str, status: Status) {
  println!("{}: new status: {:?}", unit, status);
}
```

Let's add our first local threaded function:

```
fn local_until() {
  let mut container = Container::new("acme");

  // setup our green thread pool
  let mut pool = LocalPool::new();
  let mut exec = pool.executor();

  // lazy will only execute the closure once the future has been polled
  // we will simulate the poll by returning using the future::ok method

  // typically, we perform some heavy computational process within this closure
  // such as loading graphic assets, sound, other parts of our framework/library/etc.
  let f = lazy(move |_| -> FutureResult<Container, Never> {
    container.status = Status::FetchingData;
    ok(container)
  });

  println!("container's current status: {:?}", container.status);

  container = pool.run_until(f, &mut exec).unwrap();
  new_status("local_until", container.status);

  // just to demonstrate a simulation of "fetching data over a network"
  println!("Fetching our container's score...");
  let score = block_on(container.pull_score()).unwrap();
  println!("Our container's score is: {:?}", score);

  // see if our status has changed since we fetched our score
  new_status("local_until", container.status);
}
```

And now for our locally-spawned threading examples:

```
fn local_spawns_completed() {
  let (tx, rx) = oneshot::channel();
  let mut container = Container::new("acme");

  let mut pool = LocalPool::new();
  let mut exec = pool.executor();

  // change our container's status and then send it to our oneshot channel
  exec.spawn_local(lazy(move |_| {
      container.status = Status::Loaded;
      tx.send(container).unwrap();
      FINISHED
    }))
    .unwrap();

  container = pool.run_until(rx, &mut exec).unwrap();
  new_status("local_spanws_completed", container.status);
}

fn local_nested() {
  let mut container = Container::new("acme");
  // we will need Rc (reference counts) since we are referencing multiple owners
  // and we are not using Arc (atomic reference counts) since we are only using
  // a local pool which is on the same thread technically
  let cnt = Rc::new(Cell::new(container));
  let cnt_2 = cnt.clone();

  let mut pool = LocalPool::new();
  let mut exec = pool.executor();
  let mut exec_2 = pool.executor();

  let _ = exec.spawn_local(lazy(move |_| {
    exec_2.spawn_local(lazy(move |_| {
        let mut container = cnt_2.get();
        container.status = Status::Loaded;

        cnt_2.set(container);
        FINISHED
      }))
      .unwrap();
    FINISHED
  }));

  let _ = pool.run(&mut exec);

  container = cnt.get();
  new_status("local_nested", container.status);
}
```

And now for our thread pool example:

```
fn thread_pool() {
  let (tx, rx) = mpsc::sync_channel(2);
  let tx_2 = tx.clone();

  // there are various thread builder options which are referenced at
  // https://docs.rs/futures/0.2.0-beta/futures/executor/struct.ThreadPoolBuilder.html
  let mut cpu_pool = ThreadPoolBuilder::new()
    .pool_size(2) // default is the number of cpus
    .create();

  // We need to box this part since we need the Send +'static trait
  // in order to safely send information across threads
  let _ = cpu_pool.spawn(Box::new(lazy(move |_| {
    tx.send(1).unwrap();
    FINISHED
  })));

  let f = lazy(move |_| {
    tx_2.send(1).unwrap();
    FINISHED
  });

  let _ = cpu_pool.run(f);

  let cnt = rx.into_iter().count();
  println!("Count should be 2: {:?}", cnt);
}
```

And lastly, our `main` function:

```
fn main() {
  println!("local_until():");
  local_until();

  println!("\nlocal_spawns_completed():");
  local_spawns_completed();

  println!("\nlocal_nested():");
  local_nested();

  println!("\nthread_pool():");
  thread_pool();
}
```

How it works...

Let's start by introducing the `Future` trait:

- Implementing the `Future` trait requires only three constraints: an `Item` type, an `Error` type, and a `poll()` function. The actual trait looks as follows:

```
pub trait Future {
    type Item;
    type Error;
    fn poll(
        &mut self,
        cx: &mut Context
    ) -> Result<Async<Self::Item>, Self::Error>;
}
```

- The `Poll<Self::Item, Self::Error>` is a type that translates into `Result<Async<T>, E>`, where `T = Item` and `E = Error`. This is what our example is using on line 50.
- `poll()` is called upon whenever a `futures::task::Waker` (can also be known as a *Task*) is executed with one of our executors located at `futures::executor`, or manually woken up by building a `futures::task::Context` and running with a future wrapper such as `futures::future::poll_fn`.

Now, onto our `local_until()` function:

- `LocalPool` offers us the ability to run tasks concurrently using a single thread. This is useful for functions with minimal complexity, such as traditional I/O bound functions. `LocalPools` can have multiple `LocalExecutors` (as we have created one on line 65), which can spawn our task. Since our task is single-threaded, we do not need to `Box` or add the `Send` trait to our future.
- The `futures::future::lazy` function will create a new future, from a `FnOnce` closure, which becomes the same future as the one that the closure returns (any `futures::future::IntoFuture` trait), which in our case that future is `FutureResult<Container, Never>`.
- Executing the `run_until(F: Future)` function from the `LocalPool` will perform all of the future tasks until the `Future` (indicated as `F`) has been marked as completed. This function will return `Result<<F as Future>::Item, <F as Future>::Error>` upon completion. In the example, we are returning `futures::future::ok(Container)`, on line 75, so our `F::Item` will be our `Container`.

For our `local_spawns_completed()` function:

- First, we set up our `futures::channel::oneshot` channel (which is explained later, in the *Using the oneshot channel* section).
- We will use the `oneshot` channel's `futures::channel::oneshot::Receiver` as the future to run until completion within the `run_until()` function. This allows us to demonstrate how polling would work until a signal has been received from another thread or task (in our example, this happens on line 102 with the `tx.send(...)` command).
- The `LocalExecutor`'s `spawn_local()` is a special `spawn` function that gives us the capability of executing future functions without implementing the `Send` trait.

Next, our `local_nested()` function:

- We set up our usual `Container` and then declare a reference counter that will allow us to keep a value (this would be our `Container`) across multiple executors or threads. We do not need to use an atomic reference counter, since we are using `spawn_local()`, which performs the future on a green thread (a thread that is scheduled by a virtual machine or a runtime library).
- The `LocalPool`'s `run(exec: &mut Executor)` function will run any futures spawned within the pool until all of the futures have been completed. This also includes any executors that may `spawn` additional tasks within other tasks, as our example shows.

Onto our `thread_pool()` function:

- An `std::sync::mspc::sync_channel` is created with the intention of blocking the thread for demonstration purposes.
- Next, we created a `ThreadPool` with default settings and called its `spawn(F: Box<Future<Item = (), Error = Never> + 'static + Send>)` function, which will poll the task until completion whenever we decide to execute the pool.
- After setting up our tasks, we execute the `ThreadPool`'s `run(F: Future)` function, which will block the thread in which is invoking `run()` until the `F: Future` has been completed. The function will return a value upon the future's completion even if there are other tasks spawned, and running, within the pool. Using the `mspc::sync_channel` earlier helps mitigate this issue, but will block the thread upon being invoked.

- With the `ThreadPoolBuilder`, you can:
    - Set the number of worker threads
    - Adjust the stack size
    - Set a prefixed name for the pools
    - Run a function (with the signature as `Fn(usize) + Send + Sync + 'static`) after each worker thread has started, right before the worker thread runs any tasks
    - Execute a function (with the signature as `Fn(usize) + Send + Sync + 'static`) before each worker thread shuts down

Handling errors in futures

In a real application, we would not be returning a value instantly from an asynchronous function that directly returns `Async::Ready<T>` or `FutureResult<T, E>`. Network requests time out, buffers become full, services become unavailable due to bugs or outages, and many more issues pop up on a daily basis. As much as we like to build order from chaos, usually chaos wins due to naturally-occurring entropy (programmers may know this as *scope creep*) and decay (software updates, new computer science paradigms, and so on). Luckily for us, the futures library offers us a simple way to implement error handling.

How to do it...

1. Inside the `bin` folder, create a new file called `errors.rs`.
2. Add the following code and run it with `cargo run --bin errors`:

```
extern crate futures;

use futures::prelude::*;
use futures::executor::block_on;
use futures::stream;
use futures::task::Context;
use futures::future::{FutureResult, err};
```

3. After that, let's add our structures and implementations:

```
struct MyFuture {}
impl MyFuture {
  fn new() -> Self {
    MyFuture {}
  }
}

fn map_error_example() -> FutureResult<(), &'static str> {
  err::<(), &'static str>("map_error has occurred")
}

fn err_into_example() -> FutureResult<(), u8> {
  err::<(), u8>(1)
}

fn or_else_example() -> FutureResult<(), &'static str> {
  err::<(), &'static str>("or_else error has occurred")
}

impl Future for MyFuture {
  type Item = ();
  type Error = &'static str;

  fn poll(&mut self, _cx: &mut Context) -> Poll<Self::Item, Self::Error> {
    Err("A generic error goes here")
  }
}

struct FuturePanic {}

impl Future for FuturePanic {
  type Item = ();
  type Error = ();

  fn poll(&mut self, _cx: &mut Context) -> Poll<Self::Item, Self::Error> {
    panic!("It seems like there was a major issue with catch_unwind_example")
  }
}
```

4. After that, let's add our generic error handling functions/examples:

```
fn using_recover() {
  let f = MyFuture::new();

  let f_recover = f.recover::<Never, _>(|err| {
    println!("An error has occurred: {}", err);
    ()
  });

  block_on(f_recover).unwrap();
}

fn map_error() {
  let map_fn = |err| format!("map_error_example: {}", err);

  if let Err(e) = block_on(map_error_example().map_err(map_fn))
 {
    println!("block_on error: {}", e)
  }
}

fn err_into() {
  if let Err(e) = block_on(err_into_example().err_into::()) {
    println!("block_on error code: {:?}", e)
  }
}

fn or_else() {
  if let Err(e) = block_on(or_else_example()
    .or_else(|_| Err("changed or_else's error message"))) {
    println!("block_on error: {}", e)
  }
}
```

5. And now for our `panic` functions:

```
fn catch_unwind() {
  let f = FuturePanic {};

  if let Err(e) = block_on(f.catch_unwind()) {
    let err = e.downcast::<&'static str>().unwrap();
    println!("block_on error: {:?}", err)
  }
}

fn stream_panics() {
  let stream_ok = stream::iter_ok::<_, bool>(vec![Some(1),
  Some(7), None, Some(20)]);
  // We panic on "None" values in order to simulate a stream
  that panics
  let stream_map = stream_ok.map(|o| o.unwrap());

  // We can use catch_unwind() for catching panics
  let stream = stream_map.catch_unwind().then(|r| Ok::<_, ()>
  (r));
  let stream_results: Vec<_> =
   block_on(stream.collect()).unwrap();

  // Here we can use the partition() function to separate the Ok
  and Err values
  let (oks, errs): (Vec<_>, Vec<_>) =
  stream_results.into_iter().partition(Result::is_ok);
  let ok_values: Vec<_> =
  oks.into_iter().map(Result::unwrap).collect();
  let err_values: Vec<_> =
  errs.into_iter().map(Result::unwrap_err).collect();

  println!("Panic's Ok values: {:?}", ok_values);
  println!("Panic's Err values: {:?}", err_values);
}
```

6. And finally, our `main` function:

```
fn main() {
  println!("using_recover():");
  using_recover();

  println!("\nmap_error():");
  map_error();

  println!("\nerr_into():");
  err_into();

  println!("\nor_else():");
  or_else();

  println!("\ncatch_unwind():");
  catch_unwind();

  println!("\nstream_panics():");
  stream_panics();
}
```

How it works...

Let's start with the `using_recover()` function:

- Any errors that have occurred within the future will be transformed into `<Self as Future>::Item`. Any `<Self as Future>::Error` type can be passed through, since we never produce an actual error.

- The `futures::executor::block_on(F: Future)` function will run a future until completion within the invoking thread. Any tasks within futures' `default executor` will also run on the invoking thread, but completion on the tasks may never occur since `F` may finish before the tasks have been completed. If this is the case, then the spawned tasks are dropped. `LocalPool` is often recommended for mitigating this issue, but for our examples `block_on()` will be sufficient.

All of these error handling functions can be found within the `futures::FutureExt` trait.

Now, onto our `map_error()` function:

- The `<Self as Future>::map_err<E, F>(F: FnOnce(Self::Error) -> E)` function will map a future's (`Self`) error into another error while returning a new future. This function is often used in conjunction with combinators, such as select or join, since we can guarantee that the futures will have the same error type to complete the composition.

Next, the `err_into()` function:

- Transforms the `Self::Error` into another `Error` type using the `std::convert::Into` trait
- Like `futures::FutureExt::map_err`, this function is useful for aggregating combinators together

The `or_else()` function:

- If `<Self as Future>` returns an error, `futures::FutureExt::or_else` will execute a closure with the following signature: `FnOnce(Self::Error) -> futures::future::IntoFuture<Item = Self::Item>`
- Useful for chaining failing combinators together
- The closure will not execute if the future has completed successfully, panics, or its future is dropped

Then the `catch_unwind()` function:

- This function is generally not recommended as a way to handle errors, and is only enabled with Rust's `std` option (which is enabled by default)
- Future traits implement the `AssertUnwindSafe` trait as `AssertUnwindSafe<F: Future>` trait

And lastly, the `stream_panics()` function:

- On line 95, this `futures::StreamExt::catch_unwind` function is similar to `futures::FutureExt::catch_unwind`
- If a panic occurs, it will be the last element of the stream for the stream
- This feature is only enabled with Rust's `std` option as well
- The `AssertUnwindSafe` trait is also implemented for streams as `AssertUnwindSafe<S: Stream>`

The combinators for streams are located in the `futures::StreamExt` trait, which has the same functions as `futures::FutureExt` with some additional stream-specific combinators such as `split()` and `skip_while()` that may prove to be useful for your projects.

See also

- `Chapter 6`, *Handling Errors*

Combining futures

Combining, and chaining, our futures allows us to perform multiple operations in sequential order and helps organize our code a bit more. They can be used to transform, splice, filter, and so on `<Self as Future>::Item`s.

How to do it...

1. Inside the `bin` folder, create a new file called `combinators.rs`.

2. Add the following code and run it with `cargo run --bin combinators`:

```rust
extern crate futures;
extern crate futures_util;

use futures::prelude::*;
use futures::channel::{mpsc, oneshot};
use futures::executor::block_on;
use futures::future::{ok, err, join_all, select_all, poll_fn};
use futures::stream::iter_result;
use futures_util::stream::select_all as select_all_stream;

use std::thread;

const FINISHED: Result<Async<()>, Never> = Ok(Async::Ready(()));
```

3. Let's add our `join_all` example function:

```rust
fn join_all_example() {
  let future1 = Ok::<_, ()>(vec![1, 2, 3]);
  let future2 = Ok(vec![10, 20, 30]);
  let future3 = Ok(vec![100, 200, 300]);

  let results = block_on(join_all(vec![future1, future2, future3])).unwrap();
  println!("Results of joining 3 futures: {:?}", results);

  // For parameters with a lifetime
  fn sum_vecs<'a>(vecs: Vec<&'a [i32]>) -> Box<Future, Error = ()> + 'static> {
    Box::new(join_all(vecs.into_iter().map(|x| Ok::<i32, ()>(x.iter().sum()))))
  }

  let sum_results = block_on(sum_vecs(vec![&[1, 3, 5], &[6, 7, 8], &[0]])).unwrap();
  println!("sum_results: {:?}", sum_results);
}

```

Next, we will write out our `shared` function:

```rust
fn shared() {
  let thread_number = 2;
  let (tx, rx) = oneshot::channel::();
  let f = rx.shared();
  let threads = (0..thread_number)
```

```
    .map(|thread_index| {
      let cloned_f = f.clone();
      thread::spawn(move || {
        let value = block_on(cloned_f).unwrap();
        println!("Thread #{}: {:?}", thread_index, *value);
      })
    })
    .collect::<Vec<_>>();
  tx.send(42).unwrap();

  let shared_return = block_on(f).unwrap();
  println!("shared_return: {:?}", shared_return);

  for f in threads {
    f.join().unwrap();
  }
}
```

And now for our `select_all` example:

```
fn select_all_example() {
  let vec = vec![ok(3), err(24), ok(7), ok(9)];

  let (value, _, vec) = block_on(select_all(vec)).unwrap();
  println!("Value of vec: = {}", value);

  let (value, _, vec) =
  block_on(select_all(vec)).err().unwrap();
  println!("Value of vec: = {}", value);

  let (value, _, vec) = block_on(select_all(vec)).unwrap();
  println!("Value of vec: = {}", value);

  let (value, _, _) = block_on(select_all(vec)).unwrap();
  println!("Value of vec: = {}", value);

  let (tx_1, rx_1) = mpsc::unbounded::();
  let (tx_2, rx_2) = mpsc::unbounded::();
  let (tx_3, rx_3) = mpsc::unbounded::();

  let streams = vec![rx_1, rx_2, rx_3];
  let stream = select_all_stream(streams);

  tx_1.unbounded_send(3).unwrap();
  tx_2.unbounded_send(6).unwrap();
  tx_3.unbounded_send(9).unwrap();

  let (value, details) = block_on(stream.next()).unwrap();

  println!("value for select_all on streams: {:?}", value);
  println!("stream details: {:?}", details);
}
```

Now we can add our `flatten`, `fuse`, and `inspect` functions:

```
fn flatten() {
  let f = ok::<_, _>(ok::<u32, Never>(100));
  let f = f.flatten();
  let results = block_on(f).unwrap();
  println!("results: {}", results);
}

fn fuse() {
  let mut f = ok::<u32, Never>(123).fuse();

  block_on(poll_fn(move |mut cx| {
      let first_result = f.poll(&mut cx);
      let second_result = f.poll(&mut cx);
      let third_result = f.poll(&mut cx);

      println!("first result: {:?}", first_result);
      println!("second result: {:?}", second_result);
      println!("third result: {:?}", third_result);

      FINISHED
    }))
    .unwrap();
}

fn inspect() {
  let f = ok::<u32, Never>(111);
  let f = f.inspect(|&val| println!("inspecting: {}", val));
  let results = block_on(f).unwrap();
  println!("results: {}", results);
}
```

Then we can add our `chaining` example:

```
fn chaining() {
  let (tx, rx) = mpsc::channel(3);
  let f = tx.send(1)
    .and_then(|tx| tx.send(2))
    .and_then(|tx| tx.send(3));

  let t = thread::spawn(move || {
    block_on(f.into_future()).unwrap();
  });

  t.join().unwrap();

  let result: Vec<_> = block_on(rx.collect()).unwrap();
  println!("Result from chaining and_then: {:?}", result);

  // Chaining streams together
  let stream1 = iter_result(vec![Ok(10), Err(false)]);
  let stream2 = iter_result(vec![Err(true), Ok(20)]);

  let stream = stream1.chain(stream2)
    .then(|result| Ok::<_, ()>(result));

  let result: Vec<_> = block_on(stream.collect()).unwrap();
  println!("Result from chaining our streams together: {:?}", result);
}
```

And now for our `main` function:

```
fn main() {
  println!("join_all_example():");
  join_all_example();

  println!("\nshared():");
  shared();

  println!("\nselect_all_example():");
  select_all_example();

  println!("\nflatten():");
  flatten();

  println!("\nfuse():");
  fuse();

  println!("\ninspect():");
  inspect();

  println!("\nchaining():");
  chaining();
}
```

How it works...

The `join_all()` function:

- Collects results from several futures and returns a new future with the `futures::future::JoinAll<F: Future>` trait
- The new future will perform commands for all of the aggregated futures within the `futures::future::join_all` call, returning a vector of `Vec<T: Future::Item>` in FIFO ordering
- An error will return itself immediately and cancel the other related futures

And the `shared()` function:

- `futures::FutureExt::shared` will create a handle that can be cloned, which resolves to the returning value of `<T as futures::future::SharedItem>` which can be deferred into `T`.
- Useful for polling a future on more than one thread
- This method is enabled only when Rust's `std` option is enabled (which it is by default)
- The underlying result is `futures::future::Shared<Future::Item>`, which implements `Send` and `Sync` traits
- Using `futures::future::Shared::peek(&self)` will return a value without blocking if any single shared handle has been completed

Next, the `select_all_example()` function:

- `futures::FutureExt::select_all` returns a new future that selects from a list of vectors
- The return value is `futures::future::SelectAll`, which allows us to iterate through the results
- The future's item, index of execution, and a list of futures that still need to be processed will be returned by this function as soon as one of the futures completes its execution

Then the `flatten()` function:

- `futures::FutureExt::flatten` will combine futures together with a returning result of their items being flattened

- The resultant item must implement the `futures::future::IntoFuture` trait

Onto the `fuse()` function:

- There is a small chance of `undefined behavior`, such as panicking or blocking forever, when polling a future that has already returned a `futures::Async::Ready` or `Err` value. The `futures::FutureExt::fuse` function allows us to `poll` the future again without worrying about `undefined behavior`, and will always return `futures::Async::Pending`.
- The future that's being fused will be dropped upon completion in order to reclaim resources.

The `inspect()` function:

- `futures::FutureExt::inspect` allows us to peek at an item of a future which is useful for when we are chaining combinators.

And then the `chaining()` function:

- We first create a channel with three values, and `spawn` a thread to send those three values to the channel's receiver using the `futures::FutureExt::and_then` combinator. We collect the results on line 130 from the channel.
- Then we chain two streams together on line 134 and 135 with the collection occurring on line 140. The result of both streams should be chained together on lines 137 and 138.

See also

- *Using a vector* and *Access collections as iterators* recipes in `Chapter 2`, *Working with Collections*

Using Streams

A stream is a pipeline for events that returns a value asynchronously to the invoker. `Streams` are more useful for items that require the `Iterator` trait, while `Futures` are more apt for `Result` values. When an error occurs throughout a stream, the error will not halt the stream, and polling on the stream will still return other results until the `None` value has been returned.

`Streams` and `Channels` can be a bit confusing for some. `Streams` are used for continuous, buffered data, and `Channels` are more suited for completed messages between endpoints.

How to do it...

1. Inside the `bin` folder, create a new file called `streams.rs`.
2. Add the following code and run it with `cargo run --bin streams`:

```rust
extern crate futures;

use std::thread;

use futures::prelude::*;
use futures::executor::block_on;
use futures::future::poll_fn;
use futures::stream::{iter_ok, iter_result};
use futures::channel::mpsc;
```

3. Now, let's add our constants, implementations, and so on:

```rust
#[derive(Debug)]
struct QuickStream {
  ticks: usize,
}

impl Stream for QuickStream {
  type Item = usize;
  type Error = Never;

  fn poll_next(&mut self, _cx: &mut task::Context) ->
  Poll<Option, Self::Error> {
    match self.ticks {
      ref mut ticks if *ticks > 0 => {
```

```
    *ticks -= 1;
    println!("Ticks left on QuickStream: {}", *ticks);
    Ok(Async::Ready(Some(*ticks)))
   }
   _ => {
    println!("QuickStream is closing!");
    Ok(Async::Ready(None))
   }
  }
 }
}

const FINISHED: Result<Async<()>, Never> = Ok(Async::Ready(()));
```

4. Our `quick_streams` example would be:

```
fn quick_streams() {
 let mut quick_stream = QuickStream { ticks: 10 };

 // Collect the first poll() call
 block_on(poll_fn(|cx| {
   let res = quick_stream.poll_next(cx).unwrap();
   println!("Quick stream's value: {:?}", res);
   FINISHED
  }))
  .unwrap();

 // Collect the second poll() call
 block_on(poll_fn(|cx| {
   let res = quick_stream.poll_next(cx).unwrap();
   println!("Quick stream's next svalue: {:?}", res);
   FINISHED
  }))
  .unwrap();

 // And now we should be starting from 7 when collecting the rest of the stream
 let result: Vec<_> = block_on(quick_stream.collect()).unwrap();
 println!("quick_streams final result: {:?}", result);
}
```

5. There are several ways to iterate through streams; let's add them to our code base:

```
fn iterate_streams() {
 use std::borrow::BorrowMut;
```

```

  let stream_response = vec![Ok(5), Ok(7), Err(false), Ok(3)];
  let stream_response2 = vec![Ok(5), Ok(7), Err(false), Ok(3)];

  // Useful for converting any of the `Iterator` traits into a `Stream` trait.
  let ok_stream = iter_ok::<_, ()>(vec![1, 5, 23, 12]);
  let ok_stream2 = iter_ok::<_, ()>(vec![7, 2, 14, 19]);

  let mut result_stream = iter_result(stream_response);
  let result_stream2 = iter_result(stream_response2);

  let ok_stream_response: Vec<_> = block_on(ok_stream.collect()).unwrap();
  println!("ok_stream_response: {:?}", ok_stream_response);

  let mut count = 1;
  loop {
    match block_on(result_stream.borrow_mut().next()) {
      Ok((res, _)) => {
        match res {
          Some(r) => println!("iter_result_stream result #{}: {}", count, r),
          None => { break }
        }
      },
      Err((err, _)) => println!("iter_result_stream had an error #{}: {:?}", count, err),
    }
    count += 1;
  }

  // Alternative way of iterating through an ok stream
  let ok_res: Vec<_> = block_on(ok_stream2.collect()).unwrap();
  for ok_val in ok_res.into_iter() {
    println!("ok_stream2 value: {}", ok_val);
  }

  let (_, stream) = block_on(result_stream2.next()).unwrap();
  let (_, stream) = block_on(stream.next()).unwrap();
  let (err, _) = block_on(stream.next()).unwrap_err();

  println!("The error for our result_stream2 was: {:?}", err);

  println!("All done.");
}
```

6. And now for our channeling example:

```
fn channel_threads() {
  const MAX: usize = 10;
  let (mut tx, rx) = mpsc::channel(0);

  let t = thread::spawn(move || {
    for i in 0..MAX {
      loop {
        if tx.try_send(i).is_ok() {
          break;
        } else {
          println!("Thread transaction #{} is still pending!", i);
        }
      }
    }
  });

  let result: Vec<_> = block_on(rx.collect()).unwrap();
  for (index, res) in result.into_iter().enumerate() {
    println!("Channel #{} result: {}", index, res);
  }

  t.join().unwrap();
}
```

7. Dealing with errors and channels can be done as follows:

```
fn channel_error() {
  let (mut tx, rx) = mpsc::channel(0);

  tx.try_send("hola").unwrap();

  // This should fail
  match tx.try_send("fail") {
    Ok(_) => println!("This should not have been successful"),
    Err(err) => println!("Send failed! {:?}", err),
  }

  let (result, rx) = block_on(rx.next()).ok().unwrap();
  println!("The result of the channel transaction is: {}",
       result.unwrap());

  // Now we should be able send to the transaction since we poll'ed a result already
  tx.try_send("hasta la vista").unwrap();
  drop(tx);

  let (result, rx) = block_on(rx.next()).ok().unwrap();
  println!("The next result of the channel transaction is: {}",
       result.unwrap());

  // Pulling more should result in None
  let (result, _) = block_on(rx.next()).ok().unwrap();
  println!("The last result of the channel transaction is: {:?}",
       result);
}
```

8. We can even work with buffers and channels together. Let's add our `channel_buffer` function:

```
fn channel_buffer() {
  let (mut tx, mut rx) = mpsc::channel::(0);

  let f = poll_fn(move |cx| {
    if !tx.poll_ready(cx).unwrap().is_ready() {
      panic!("transactions should be ready right away!");
    }

    tx.start_send(20).unwrap();
    if tx.poll_ready(cx).unwrap().is_pending() {
      println!("transaction is pending...");
    }

    // When we're still in "Pending mode" we should not be able
    // to send more messages/values to the receiver
    if tx.start_send(10).unwrap_err().is_full() {
      println!("transaction could not have been sent to the receiver due \
            to being full...");
    }

    let result = rx.poll_next(cx).unwrap();
    println!("the first result is: {:?}", result);
    println!("is transaction ready? {:?}",
         tx.poll_ready(cx).unwrap().is_ready());

    // We should now be able to send another message since we've pulled
    // the first message into a result/value/variable.
    if !tx.poll_ready(cx).unwrap().is_ready() {
      panic!("transaction should be ready!");
    }

    tx.start_send(22).unwrap();
    let result = rx.poll_next(cx).unwrap();
    println!("new result for transaction is: {:?}", result);

    FINISHED
  });

  block_on(f).unwrap();
}
```

9. Just because we're using the futures crate doesn't mean everything has to be concurrent. Add the following example to demonstrate how to block with channels:

```
fn channel_threads_blocking() {
  let (tx, rx) = mpsc::channel::(0);
  let (tx_2, rx_2) = mpsc::channel::<()>(2);

  let t = thread::spawn(move || {
    let tx_2 = tx_2.sink_map_err(|_| panic!());
    let (a, b) =
    block_on(tx.send(10).join(tx_2.send(()))).unwrap();

    block_on(a.send(30).join(b.send(()))).unwrap();
  });

  let (_, rx_2) = block_on(rx_2.next()).ok().unwrap();
  let (result, rx) = block_on(rx.next()).ok().unwrap();
  println!("The first number that we sent was: {}",
  result.unwrap());

  drop(block_on(rx_2.next()).ok().unwrap());
  let (result, _) = block_on(rx.next()).ok().unwrap();
  println!("The second number that we sent was: {}",
  result.unwrap());

  t.join().unwrap();
}
```

10. Sometimes we'll need concepts such as unbounded channels; let's add our `channel_unbounded` function:

```
fn channel_unbounded() {
  const MAX_SENDS: u32 = 5;
  const MAX_THREADS: u32 = 4;
  let (tx, rx) = mpsc::unbounded::();

  let t = thread::spawn(move || {
    let result: Vec<_> = block_on(rx.collect()).unwrap();
    for item in result.iter() {
      println!("channel_unbounded: results on rx: {:?}", item);
    }
  });

  for _ in 0..MAX_THREADS {
    let tx = tx.clone();

    thread::spawn(move || {
      for _ in 0..MAX_SENDS {
        tx.unbounded_send(1).unwrap();
      }
    });
  }

  drop(tx);

  t.join().ok().unwrap();
}
```

11. And now we can add our `main` function:

```
fn main() {
  println!("quick_streams():");
  quick_streams();

  println!("\niterate_streams():");
  iterate_streams();

  println!("\nchannel_threads():");
  channel_threads();

  println!("\nchannel_error():");
  channel_error();

  println!("\nchannel_buffer():");
  channel_buffer();

  println!("\nchannel_threads_blocking():");
  channel_threads_blocking();

  println!("\nchannel_unbounded():");
  channel_unbounded();
}
```

How it works...

First, let's talk about the `QuickStream` structure:

- The `poll_next()` function will continuously be invoked, and with each iteration, `i`'s ticks attribute will be decremented by `1`
- Polling will stop when the ticks attribute reaches `0` and returns `futures::Async::Ready<None>`

Within the `quick_streams()` function:

- We build a `futures::task::Context` by using `futures::future::poll_on(f: FnMut(|cx: Context|))`, so that we can explicitly invoke `QuickStream`'s `poll_next()` function on lines 42 and 50
- Since we have declared `10` ticks on line 38, our first two `block_on`'s `poll_next()` calls should yield `9` and `8`
- The next `block_on` call, on line 57, will keep polling `QuickStream` until `futures::Async::Ready<None>` is returned from the ticks attribute equaling zero

Within `iterate_streams()`:

- `futures::stream::iter_ok` will convert an `Iterator` into a `Stream`, which will always be ready to return the next value
- `futures::stream::iter_result` does the same thing as `iter_ok`, except we use `Result` values instead of `Ok` values
- On lines 78 through 89, we iterate through the stream's results and print out some information depending on whether the value was `Ok` or an `Error` type. If the `None` type has been returned from our stream, then we will break the loop
- Lines 92 through 95 show an alternative way of iterating through a stream's `Ok` results by using the `into_iter()` calls
- Lines 97 through 99 show an alternative way of iterating through a stream's `Result` return types

TIP

Loops, iterated results, and `collect()` calls are synchronous. We used this functions for demonstrative/educational purposes only. Combinators such as `map()`, `filter()`, `and_then()`, etc. would be used in a real application for streams and channels.

The `channel_threads()` function:

- On line 107, we define the maximum number of sends we want to attempt.
- On line 108, we declare a channel to send messages to. Channel capacity is the `buffer size (the argument of futures::channel::mpsc::channel) + the number of senders` (each sender is guaranteed a slot within the channel). Channels will return a `futures::channel::mpsc::Receiver<T>`, which implements the `Stream` trait, and a `futures::channel::mpsc::Sender<T>`, which implements the `Sink` trait.
- Lines 110 through 120 is where we `spawn` a thread and attempt to send 10 signals, looping until each send is sent successfully.
- We collect, and display, our results on line 122 through 125, and join our threads on line 127.

The `channel_error()` section:

- On line 131, we declare our channel with a `0 usize` buffer as the argument, which gives us one slot for the initial sender
- We send the first message successfully on line 133
- Lines 136 through 139 should fail, since we are trying to send a message to a channel that is considered full (since we did not receive the value, drop the initial sender, flush the stream, and so on)
- On line 146, we use the sender's `futures::channel::mpsc::Sender::try_send(&mut self, msg: T)` functions, which won't block our thread unless we don't drop/invoke the sender's destroyer method using `drop(T)` on line 147
- Polling the stream any additional times after receiving the last value will always return `None`

Next, the `channel_buffer()` function:

- We set up a future closure with `poll_fn()` on line 162.
- We check to see if our sender is ready to be polled with its `futures::sink::poll_ready(&mut self, cx: &mut futures::task::Context)` method on lines 163 through 165.
- Sinks have a method called `futures::sink::start_send(&mut self, item: <Self as Sink>::SinkItem) -> Result<(), <Self as Sink>::SinkError>`, which prepares the message to be delivered, but won't until we flush or close the sink. `poll_flush()` is often used to guarantee that every message has been sent from the sink.

- Polling the stream for the next value will also alleviate space within the sink/sender using the `futures::stream::poll_next(&mut self, cx: &mut futures::task::Context)` method, as we have done on line 179.
- We can check if our sender is ready, as we have done on line 182 using the `futures::Async::is_ready(&self) -> bool` method.
- Our final value should be `22` and displayed to the console from line 192.

Then the `channel_threads_blocking()` function:

- First, we set up our channels on lines 201 and 202.
- Then we `spawn` a thread that will map all of `tx_2`'s errors into a `panic!` (line 205), and then we send the value of `10` to our first channel while joining a second sender with the `()` value (line 206). On line 208, we send the value of `30` and another empty value `()` to our second channel.
- On line 211 we poll the second channel, which would hold a value of `()`.
- On line 212 we poll the first channel, which would hold a value of `10`.
- We drop the second channel's receiver (line 215), since we need to close or flush for the second `tx_2.send()` call on line 208 (`tx_2` is known as variable `b` on this line).
- After performing the drop, we can finally return our second value from the first channel's sender, which should be `30`.

And the `channel_unbounded()` function:

- On line 225 we declare an `unbounded channel`, which means that sending messages to this channel will always succeed as long as the receiver is not closed. Messages will be buffered on an as-needed basis, and since this channel is unbounded, our application can exhaust our available memory.
- Lines 227 through 232 `spawn` a thread that collects all of the receiver's messages (line 228), and we iterate through them on line 229. The item on line 230 is a tuple of the index in which the message was received and the message's value (in our case, this is always 1).
- Lines 237 through 241 is what will `spawn` the number of threads (using the `MAX_THREADS` constant) as well as the number of times that we want to send per thread using the `MAX_THREADS` constant.
- Lines 244 we will drop (which closes) the channel's sender so that we may collect all of the messages from line 228.

- We join the spawned thread with our current thread on line 246, which will execute the collection and iterations commands (lines 228 through 231).

Using Sinks

Sinks are the *sending-side* of channels, sockets, pipes, and so on, in which messages can be sent from the sink asynchronously. Sinks communicate by initiating a send signal, and then the rest is polled. One thing to watch out for when using sinks is that they can run out of sending space, which will prevent more messages from being sent.

How to do it...

1. Inside the `bin` folder, create a new file called `sinks.rs`.
2. Add the following code and run it with `cargo run --bin sinks`:

```
1   extern crate futures;
2
3   use futures::prelude::*;
4   use futures::future::poll_fn;
5   use futures::executor::block_on;
6   use futures::sink::flush;
7   use futures::stream::iter_ok;
8   use futures::task::{Waker, Context};
9
10  use std::mem;
```

3. Let's add our examples with using vectors as `sinks`:

```
12  fn vector_sinks() {
13    let mut vector = Vec::new();
14    let result = vector.start_send(0);
15    let result2 = vector.start_send(7);
16
17    println!("vector_sink: results of sending should both be
      Ok(()): {:?} and {:?}",
18         result,
19         result2);
20    println!("The entire vector is now {:?}", vector);
21
22    // Now we need to flush our vector sink.
23    let flush = flush(vector);
24    println!("Our flush value: {:?}", flush);
```

```
  println!("Our vector value: {:?}", flush.into_inner().unwrap());

  let vector = Vec::new();
  let mut result = vector.send(2);
  // safe to unwrap since we know that we have not flushed the sink yet
  let result = result.get_mut().unwrap().send(4);

  println!("Result of send(): {:?}", result);
  println!("Our vector after send(): {:?}", result.get_ref().unwrap());

  let vector = block_on(result).unwrap();
  println!("Our vector should already have one element: {:?}", vector);

  let result = block_on(vector.send(2)).unwrap();
  println!("We can still send to our stick to ammend values: {:?}",
       result);

  let vector = Vec::new();
  let send_all = vector.send_all(iter_ok(vec![1, 2, 3]));
  println!("The value of vector's send_all: {:?}", send_all);

  // Add some more elements to our vector...
  let (vector, _) = block_on(send_all).unwrap();
  let (result, _) = block_on(vector.send_all(iter_ok(vec![0, 6, 7]))).unwrap();
  println!("send_all's return value: {:?}", result);
}
```

We can map/transform our `sinks` values. Let's add our `mapping_sinks` example:

```
fn mapping_sinks() {
  let sink = Vec::new().with(|elem: i32| Ok::<i32, Never>(elem * elem));

  let sink = block_on(sink.send(0)).unwrap();
  let sink = block_on(sink.send(3)).unwrap();
  let sink = block_on(sink.send(5)).unwrap();
  println!("sink with() value: {:?}", sink.into_inner());

  let sink = Vec::new().with_flat_map(|elem| iter_ok(vec![elem; elem].into_iter().map(|y| y * y)));

  let sink = block_on(sink.send(0)).unwrap();
  let sink = block_on(sink.send(3)).unwrap();
  let sink = block_on(sink.send(5)).unwrap();
  let sink = block_on(sink.send(7)).unwrap();
  println!("sink with_flat_map() value: {:?}",
  sink.into_inner());
}
```

We can even send messages to multiple `sinks`. Let's add our `fanout` function:

```
fn fanout() {
  let sink1 = vec![];
  let sink2 = vec![];
  let sink = sink1.fanout(sink2);
  let stream = iter_ok(vec![1, 2, 3]);
  let (sink, _) = block_on(sink.send_all(stream)).unwrap();
  let (sink1, sink2) = sink.into_inner();

  println!("sink1 values: {:?}", sink1);
  println!("sink2 values: {:?}", sink2);
}
```

Next, we'll want to implement a structure for a customized sink. Sometimes our application will require us to manually flush our `sinks` instead of doing it automatically. Let's add our `ManualSink` structure:

```
#[derive(Debug)]
struct ManualSink {
  data: Vec,
  waiting_tasks: Vec,
}

impl Sink for ManualSink {
  type SinkItem = Option; // Pass None to flush
  type SinkError = ();

  fn start_send(&mut self, op: Option) -> Result<(),
  Self::SinkError> {
    if let Some(item) = op {
      self.data.push(item);
    } else {
      self.force_flush();
    }

    Ok(())
  }

  fn poll_ready(&mut self, _cx: &mut Context) -> Poll<(), ()> {
    Ok(Async::Ready(()))
  }

  fn poll_flush(&mut self, cx: &mut Context) -> Poll<(), ()> {
    if self.data.is_empty() {
      Ok(Async::Ready(()))
    } else {
      self.waiting_tasks.push(cx.waker().clone());
      Ok(Async::Pending)
    }
  }

  fn poll_close(&mut self, _cx: &mut Context) -> Poll<(), ()> {
    Ok(().into())
  }
}

impl ManualSink {
  fn new() -> ManualSink {
    ManualSink {
      data: Vec::new(),
      waiting_tasks: Vec::new(),
    }
  }

  fn force_flush(&mut self) -> Vec {
    for task in self.waiting_tasks.clone() {
      println!("Executing a task before replacing our values");
      task.wake();
    }

    mem::replace(&mut self.data, vec![])
  }
}
```

And now for our `manual flush` function:

```
fn manual_flush() {
  let mut sink = ManualSink::new().with(|x| Ok::<Option, ()>(x));
  let _ = sink.get_mut().start_send(Some(3));
  let _ = sink.get_mut().start_send(Some(7));

  let f = poll_fn(move |cx| -> Poll<Option<_>, Never> {
    // Try to flush our ManualSink
    let _ = sink.get_mut().poll_flush(cx);
    let _ = flush(sink.get_mut());

    println!("Our sink after trying to flush: {:?}", sink.get_ref());

    let results = sink.get_mut().force_flush();
    println!("Sink data after manually flushing: {:?}",
         sink.get_ref().data);
    println!("Final results of sink: {:?}", results);

    Ok(Async::Ready(Some(())))
  });

  block_on(f).unwrap();
}
```

And lastly, we can add our `main` function:

```
fn main() {
  println!("vector_sinks():");
  vector_sinks();

  println!("\nmapping_sinks():");
  mapping_sinks();

  println!("\nfanout():");
  fanout();

  println!("\nmanual_flush():");
  manual_flush();
}
```

How it works...

First, let's take a look at the `futures::Sink` trait itself:

```
pub trait Sink {
    type SinkItem;
    type SinkError;

    fn poll_ready(
        &mut self,
        cx: &mut Context
    ) -> Result<Async<()>, Self::SinkError>;
    fn start_send(
        &mut self,
        item: Self::SinkItem
    ) -> Result<(), Self::SinkError>;
```

```
            fn poll_flush(
                &mut self,
                cx: &mut Context
            ) -> Result<Async<()>, Self::SinkError>;
            fn poll_close(
                &mut self,
                cx: &mut Context
            ) -> Result<Async<()>, Self::SinkError>;
        }
```

We are already familiar with the `Item` and `Error` concepts from futures and streams, so we will move on to the required functions:

- `poll_ready` must be invoked with the returning value of `Ok(futures::Async::Ready(()))` before each attempt at using `start_send`. If the sink receives an error, the sink will no longer be able to receive items.
- `start_send`, as stated previously, prepares the message to be delivered, but won't until we flush or close the sink. If the sink uses buffers, the `Sink::SinkItem` won't be processed until the buffer has been fully completed.
- `poll_flush` will flush the sink, which will allow us to collect items that are currently being processed. `futures::Async::Ready` will return if the sink does not have any more items within the buffer, otherwise, the sink will return `futures::Async::Pending`.
- `poll_close` will flush and close the sink, following the same return rules as `poll_flush`.

Now, onto our `vector_sinks()` function:

- Sinks are implemented for `Vec<T>` types, so we can declare a mutable vector and use the `start_send()` function, which will immediately poll our values into the vector on lines 13 through 15.
- On line 28 we use the `futures::SinkExt::send(self, item: Self::SinkItem)`, which will complete after the item has been processed and flushed through the sink. `futures::SinkExt::send_all` is recommended for batching multiple items to send through, versus having to manually flush between each send call (as demonstrated on line 43).

Our `mapping_sinks()` function:

- Line 51 demonstrates how you can map/manipulate elements within a sink using the `futures::SinkExt::with` function. This function produces a new sink that iterates through each item and sends the final value *as a future* to the *parent* sink.

- Line 60 illustrates the `futures::SinkExt::flat_with_map` function that has mostly the same functionality as the `futures::SinkExt::with` function except each iterated item is sent as a stream value to the *parent* sink and will return an `Iterator::flat_map` value instead of a `Iterator::map`.

Next, the `fanout()` function:

- The `futures::SinkExt::fanout` function allows us to send messages to multiple sinks at one time, as we have done on line 72.

And then `manual_flush()`:

- We first implement our own `Sink` trait with the `ManualSink<T>` construct (lines 81 through 135). Our `ManualSink`'s `poll_flush` method will only return `Async::Ready()` if our data vector is empty, otherwise, we are going to push the task (`futures::task::Waker`) into a queue line through the `waiting_tasks` attribute. We use the `waiting_tasks` attribute within our `force_flush()` function (line 128) in order to manually *wake up* our tasks (line 130).
- On lines 138 through 140, we build our `ManualSink<Option<i32>>` and start sending some values.
- We use `poll_fn` on line 142 in order to quickly build a `futures::task::Context` so that we may pass this value down to our underlying poll calls.
- On line 144 we manually call our `poll_flush()` function, which will not execute our actual tasks since they are placed within the `waiting_tasks` attribute.
- Until we invoke `force_flush()`, our sink will not return any values (as indicated on lines 150-151). Once this function has been called upon and the underlying `Waker` tasks have finished executing, then we can see the messages (line 152) that we sent earlier (lines 139 and 140).

Using the oneshot channel

Oneshot channels are useful for when you need to send only one message to a channel. The oneshot channel is applicable for tasks that really only need to be updated/notified once, such as whether or not a recipient has read your message, or as a final destination within a task pipeline to notify the end user that the task has been completed.

How to do it...

1. Inside the `bin` folder, create a new file called `oneshot.rs`.
2. Add the following code and run it with `cargo run --bin oneshot`:

```
extern crate futures;

use futures::prelude::*;
use futures::channel::oneshot::*;
use futures::executor::block_on;
use futures::future::poll_fn;
use futures::stream::futures_ordered;

const FINISHED: Result<Async<()>, Never> = Ok(Async::Ready(()));

fn send_example() {
  // First, we'll need to initiate some oneshot channels like so:
  let (tx_1, rx_1) = channel::();
  let (tx_2, rx_2) = channel::();
  let (tx_3, rx_3) = channel::();

  // We can decide if we want to sort our futures by FIFO (futures_ordered)
  // or if the order doesn't matter (futures_unordered)
  // Note: All futured_ordered()'ed futures must be set as a Box type
  let mut ordered_stream = futures_ordered(vec![
    Box::new(rx_1) as Box<Future>,
    Box::new(rx_2) as Box<Future>,
  ]);

  ordered_stream.push(Box::new(rx_3) as Box<Future>);

  // unordered example:
  // let unordered_stream = futures_unordered(vec![rx_1, rx_2, rx_3]);

  // Call an API, database, etc. and return the values (in our case we're typecasting to u32)
  tx_1.send(7).unwrap();
  tx_2.send(12).unwrap();
  tx_3.send(3).unwrap();

  let ordered_results: Vec<_> =
  block_on(ordered_stream.collect()).unwrap();
  println!("Ordered stream results: {:?}", ordered_results);
}

fn check_if_closed() {
  let (tx, rx) = channel::();

  println!("Is our channel canceled? {:?}", tx.is_canceled());
  drop(rx);

  println!("Is our channel canceled now? {:?}",
  tx.is_canceled());
}

fn check_if_ready() {
  let (mut tx, rx) = channel::();
  let mut rx = Some(rx);

  block_on(poll_fn(|cx| {
      println!("Is the transaction pending? {:?}",
           tx.poll_cancel(cx).unwrap().is_pending());
      drop(rx.take());

      let is_ready = tx.poll_cancel(cx).unwrap().is_ready();
      let is_pending =
      tx.poll_cancel(cx).unwrap().is_pending();

      println!("Are we ready? {:?} This means that the pending
      should be false: {:?}",
           is_ready,
           is_pending);
      FINISHED
    }))
    .unwrap();
}

fn main() {
  println!("send_example():");
  send_example();

  println!("\ncheck_if_closed():");
  check_if_closed();

  println!("\ncheck_if_ready():");
  check_if_ready();
}
```

How it works...

Within our `send_example()` function:

- On lines 13 through 15, we set up three `oneshot` channels.
- On lines 20 through 23 we use `futures::stream::futures_ordered`, which will convert a list (any `IntoIterator` value) of futures into a `Stream` yielding results on a first in, first out (FIFO) basis. If any underlying futures do not complete before the next future is invoked, this function will wait until the long-running future has been completed and will then internally re-sort that future into its proper order.
- Line 25 shows us that we can push additional futures into the `futures_ordered` iterator separately.
- Line 28 demonstrates another function that doesn't rely on sorting on a FIFO basis, called `futures::stream::futures_unordered`. This function will have better performance than its counterpart `futures_ordered`, but for our example, we are not sending enough values to make a difference.
- On lines 31 through 33 we send values to our channels, mimicking the process of returning values from an API, a database, and so on. If the send is successful then `Ok(())` will be returned, otherwise, an `Err` type will be returned.
- And on our last two lines (35 and 36), we collect our `futures_ordered` values and display them to the console.

Next, the `check_if_closed()` function:

- Our channel should remain open until we explicitly drop/destroy the receiver (or send a value to the channel). We can check the status of our receiver by invoking the `futures::channel::oneshot::Sender::is_canceled(&self) -> bool` function, which we have done on lines 42 and 45.

Then the `check_if_ready()` function:

- On line 50 we explicitly assign a value to the oneshot's receiver, which would put our receiver in a state of pending (since it already has a value).
- We drop our receiver on line 55 and we can check if our receiver is ready by using our sender's `futures::channel::oneshot::Sender::poll_cancel` function, which we use on lines 57 and 58. `poll_cancel` will return `Ok(Async::Ready)` if the receiver has been dropped or `Ok(Async::Pending)` if the receiver has not been dropped.

Returning futures

The `Future` trait relies on three main ingredients: a type, an error, and a `poll()` function that returns a `Result<Async<T>, E>` structure. The `poll()` method will never block the main thread, and `Async<T>` is an enumerator with two variants: `Ready(T)` and `Pending`. Periodically, the `poll()` method will be invoked by a task’s context's `waker()` trait, located in `futures::task::context::waker`, until a value is ready to be returned.

How to do it...

1. In the `src/bin` folder, create a file called `returning.rs`.
2. Add the following code and run it with `cargo run —bin returning`:

```
extern crate futures;

use futures::executor::block_on;
use futures::future::{join_all, Future, FutureResult, ok};
use futures::prelude::*;

#[derive(Clone, Copy, Debug, PartialEq)]
enum PlayerStatus {
  Loading,
  Default,
  Jumping,
}

#[derive(Clone, Copy, Debug)]
struct Player {
  name: &'static str,
  status: PlayerStatus,
  score: u32,
  ticks: usize,
}
```

3. Now comes the implementations for the structs:

```
impl Player {
  fn new(name: &'static str) -> Self {
    let mut ticks = 1;
    // Give Bob more ticks explicitly
    if name == "Bob" {
      ticks = 5;
    }

    Player {
      name: name,
      status: PlayerStatus::Loading,
      score: 0,
      ticks: ticks,
    }
  }

  fn set_status(&mut self, status: PlayerStatus) -> FutureResult<&mut Self, Never> {
    self.status = status;
    ok(self)
  }

  fn can_add_points(&mut self) -> bool {
    if self.status == PlayerStatus::Default {
      return true;
    }

    println!("We couldn't add any points for {}!", self.name);
    return false;
  }

  fn add_points(&mut self, points: u32) -> Async<&mut Self> {
    if !self.can_add_points() {
      Async::Ready(self)
    } else {
      let new_score = self.score + points;
      // Here we would send the new score to a remote server
      // but for now we will manaully increment the player's score.

      self.score = new_score;

      Async::Ready(self)
    }
  }
}

impl Future for Player {
  type Item = Player;
  type Error = ();

  fn poll(&mut self, cx: &mut task::Context) -> Poll<Self::Item, Self::Error> {
    // Presuming we fetch our player's score from a
    // server upon initial load.
    // After we perform the fetch send the Result value.

    println!("Player {} has been poll'ed!", self.name);

    if self.ticks == 0 {
      self.status = PlayerStatus::Default;
      Ok(Async::Ready(*self))
    } else {
      self.ticks -= 1;
      cx.waker().wake();
      Ok(Async::Pending)
    }
  }
}
```

4. Next, we'll want to add our `helper` functions and our `Async` function for adding points to our players:

```
fn async_add_points(player: &mut Player,
          points: u32)
          -> Box<Future + Send> {
  // Presuming that player.add_points() will send the points to a
  // database/server over a network and returns an updated
  // player score from the server/database.
  let _ = player.add_points(points);

  // Additionally, we may want to add logging mechanisms,
  // friend notifications, etc. here.

  return Box::new(ok(*player));
}

fn display_scoreboard(players: Vec<&Player>) {
  for player in players {
    println!("{}'s Score: {}", player.name, player.score);
  }
}
```

5. And finally, the actual usage:

```
fn main() {
  let mut player1 = Player::new("Bob");
  let mut player2 = Player::new("Alice");

  let tasks = join_all(vec![player1, player2]);

  let f = join_all(vec![
  async_add_points(&mut player1, 5),
  async_add_points(&mut player2, 2),
])
  .then(|x| {
    println!("First batch of adding points is done.");
    x
  });

block_on(f).unwrap();

let players = block_on(tasks).unwrap();
player1 = players[0];
player2 = players[1];

println!("Scores should be zero since no players were loaded");
display_scoreboard(vec![&player1, &player2]);

// In our minigame, a player cannot score if they are currently
// in the air or "jumping."
// Let's make one of our players' status set to the jumping status.

let f = player2.set_status(PlayerStatus::Jumping).and_then(move |mut new_player2| {
  async_add_points(&mut player1, 10)
    .and_then(move |_| {
      println!("Finished trying to give Player 1 points.");
      async_add_points(&mut new_player2, 2)
    })
    .then(move |new_player2| {
      println!("Finished trying to give Player 2 points.");
      println!("Player 1 (Bob) should have a score of 10 and Player 2 (Alice) should \
            have a score of 0");

      // unwrap is used here to since
      display_scoreboard(vec![&player1, &new_player2.unwrap()]);
      new_player2
    })
});

block_on(f).unwrap();

println!("All done!");
```

```
157 }
```

How it works...

Let's start by introducing the structures that participate in this example:

- `PlayerStatus` is an enumerator for maintaining a *global* state on the player's instance. The variants are:
    - `Loading`, which is the initial state
    - `Default`, which is applied after we are done loading the player's stats
    - `Jumping` is a special state that won't allow us to add points to the player's scoreboard due to the rules of the game
- `Player` holds our player's main attributes, along with a special attribute called ticks that stores the amount of cycles that we want to run through with `poll()` before assigning the player's status from `Loading` to `Default`.

Now, onto our implementations:

- Jumping down to the `fn set_status(&mut self, status: PlayerStatus) -> FutureResult<&mut Self, Never>` function on our `Player` structure, we will notice a return value of `FutureResult`, which tells futures that this function will immediately return a computed value from the `result()`, `ok()`, or `err()` functions from `futures::futures`. This is useful for quickly prototyping our application while being able to utilize our `executors` and futures combinators.
- At the `fn add_points(&mut self, points: u32) -> Async<&mut Self>` function we return our `Async` value immediately, since we currently do not have a server to use, but we would implement the `Async<T>` value over `FutureResult` for functions that require computations asynchronously.
- We mimic the time it takes for a network request using our player's `ticks` attribute. `Poll<I, E>` will keep executing as long as we are returning `Async::Pending` (line [x]). The executor needs to know whether or not a task needs to be polled again. The task's `Waker` trait is what handles these notifications, and we can manually invoke it using `cx.waker().wake()` on line 83 . Once our player's `ticks` attribute reaches zero we send an `Async::Ready(self)` signal, which tells the executor to no longer poll this function.

For our `async_add_points()` helper method:

- We return `Box<Future<Item = Player, Error = Never> + Send`, which tells futures that this function will eventually return a value of `Player` (since we `Never` return an error).
- The `+ Send` part of the return is not necessary for our current code base, but in the future, we may want to offload some of these tasks onto other threads which executors require. Spawning across threads requires us to return the `futures::prelude::Never` type as an error and a `'static` variable as well.
- When calling future functions with combinators (such as `then` and `and_then`), we will need to return a `Never` error type or the same error type as every other future function that is being called within the same combinator flow.

Finally, onto our main block:

- We use the `futures::future::join_all` function, which accepts any `IntoIterator` that contains all `InfoFuture` trait elements (which should be all future functions). This either collects and returns `Vec<T>` sorted FIFO, or cancels executing as soon as the first error returns from any of the future functions within the collection, which becomes the returning value for the `join_all()` call.
- `then()` and `and_then()` are combinators that internally use `future::Chain` and return a `Future` trait value, which allows us to add more combinators if we wanted. See the *Using combinators and utilities* section for more information on combinators.
- `block_on()` is an executor method that handles any future function or value as its input and returns a `Result<Future::Item, Future::Error>`. When running this method, the function containing the method will block until the future(s) have been completed. Spawned tasks will execute on the default executor, but they may not be completed before `block_on` finishes its task(s). If `block_on()` finishes before the spawned tasks, then those spawned tasks will be dropped.
- We can also use `block_on()` as a quick way to run our cycles/ticks and execute task(s), which invokes our `poll()` functions. We used this method on line 124 for *initially loading players* onto the game.

There's more...

The `box()` method for returning futures does cause an additional allocation to the heap. Another method of returning futures relies on using a `nightly` version of Rust or for this issue `https://github.com/rust-lang/rust/issues/34511` to be resolved. The new `async_add_points()` method would return an implied `Future` trait and would look as follows:

```
fn async_add_points<F>(f: F, player: &mut Player, points: u32) -> impl
Future<Item = Player, Error = F::Error>
where F: Future<Item = Player>,
{
    // Presuming that player.add_points() will send the points to a
    // database/server over a network and returns
    // an updated player score from the server/database.
    let _ = player.add_points(points).flatten();

    // Additionally, we may want to add logging mechanisms, friend
    notifications, etc. here.

    return f.map(player.clone());
}
```

Rust may cause `undefined behavior` if we were to call `poll()` more than once for a future. This problem can be mitigated by converting the future into a stream by using the `into_stream()` method or using the `fuse()` adapter, which adds a tiny bit of runtime overhead.

Tasks are usually executed/polled from using an `executor` such as the `block_on()` helper function. You can manually execute tasks by creating a `task::Context` and calling `poll()` directly from the task. As a general rule, it is recommended to not invoke `poll()` manually and to have an executor manage polling automatically.

See also

- *Boxing data* recipe in `Chapter 5`, *Advanced Data Structures*
- `Chapter 7`, *Parallelism and Rayon*

Locking resources with BiLocks

BiLocks are used when we need to store a value across multiple threads with up to two owners associated with that value. Applicable uses for a BiLock type would be splitting TCP/UDP data for reading and writing, or adding a layer between a sink and a stream (for logging, monitoring, and so on), or it can be a sink and a stream at the same time.

When using futures with an additional crate (such as tokio or hyper), knowing BiLocks can help us wrap data around the other crate's common methods. This would allow us to build futures and concurrency on top of existing crates without having to wait until the crate's maintainers support concurrency explicitly. BiLocks are a very low-level utility, but understanding how they work can help us further down the road with our (web) applications.

In the next chapter, we will mostly focus on networking with Rust, but we will also get to practice integrating futures with other crates. BiLocks can be used throughout these next examples, if you wanted to split a TCP/UDP stream in a mutex state, although it is not necessary to do so with the crates that we will be using.

How to do it...

1. In the `src/bin` folder, create a file called `bilocks.rs`.
2. Add the following code and run it with `cargo run —bin bilocks`:

```
extern crate futures;
extern crate futures_util;

use futures::prelude::*;
use futures::executor::LocalPool;
use futures::task::{Context, LocalMap, Wake, Waker};
use futures_util::lock::BiLock;

use std::sync::Arc;

struct FakeWaker;
impl Wake for FakeWaker {
  fn wake(_: &Arc) {}
}

struct Reader {
  lock: BiLock,
}

```

```
struct Writer {
  lock: BiLock,
}

fn split() -> (Reader, Writer) {
  let (a, b) = BiLock::new(0);
  (Reader { lock: a }, Writer { lock: b })
}
fn main() {
  let pool = LocalPool::new();
  let mut exec = pool.executor();
  let waker = Waker::from(Arc::new(FakeWaker));
  let mut map = LocalMap::new();
  let mut cx = Context::new(&mut map, &waker, &mut exec);

  let (reader, writer) = split();
  println!("Lock should be ready for writer: {}",
       writer.lock.poll_lock(&mut cx).is_ready());
  println!("Lock should be ready for reader: {}",
       reader.lock.poll_lock(&mut cx).is_ready());

  let mut writer_lock = match writer.lock.lock().poll(&mut cx).unwrap() {
    Async::Ready(t) => t,
    _ => panic!("We should be able to lock with writer"),
  };

  println!("Lock should now be pending for reader: {}",
       reader.lock.poll_lock(&mut cx).is_pending());
  *writer_lock = 123;

  let mut lock = reader.lock.lock();
  match lock.poll(&mut cx).unwrap() {
    Async::Ready(_) => {
      panic!("The lock should not be lockable since writer has already locked it!")
    }
    _ => println!("Couldn't lock with reader since writer has already initiated the lock"),
  };

  let writer = writer_lock.unlock();

  let reader_lock = match lock.poll(&mut cx).unwrap() {
    Async::Ready(t) => t,
    _ => panic!("We should be able to lock with reader"),
  };

66    println!("The new value for the lock is: {}", *reader_lock);
67
68    let reader = reader_lock.unlock();
69    let reunited_value = reader.reunite(writer).unwrap();
70
71    println!("After reuniting our locks, the final value is
      still: {}",
72        reunited_value);
73  }
```

How it works...

- First, we need to implement a fake `futures::task::Waker` for when we create a new context (this is what our `FakeWaker` structure is for on lines 11 through 14)
- Since BiLocks require two owners, we will divide the ownership into two different structures, called `Reader<T>` (on lines 16 through 18) and `Writer<T>` (on lines 20 through 22)
- Our `split() -> (Reader<u32>, Writer<u32>)` function is just to structure/organize our code a bit better, and when calling `BiLock::new(t: T)` the return type is a tuple of two `futures_util::lock::BiLock` elements

Now that the preliminary code has been explained, let's dive into our `main()` function:

- On lines 30 through 34 we set up a new `LocalPool`, `LocalExecutor`, `Waker` (`FakeWaker`), and a `LocalMap` (map storage of local data within tasks) for creating a new `Context`, since we will be polling our locks manually for demonstration purposes.
- Lines 38 and 40 use the `futures_util::lock::BiLock::poll_lock` function, which returns an `Async<futures_util::lock::BiLockGuard<T>>` value if the lock is available. If the lock is not available then the function will return `Async::Pending`. The lock (the `BiLockGuard<T>`) will unlock when the reference is dropped.
- On line 42 we execute `writer.lock.lock()`, which will block the lock and a `BiLockAcquire<T>` will be returned, which is a future that can be polled. When `BiLockAcquire` is polled, a `Poll<BiLockAcquired<T>, ()>` value is returned and that value can be dereferenced mutably.
- On line 48, we can now see that the lock is currently in an `Async::Pending` state, which would not allow us to lock the BiLock again, as shown on lines 51 through 57.

- After modifying our lock's value (line 49), we should now unlock it (line 59) so that the other owner can reference it (lines 61 through 64).
- When we call `BiLockAcquired::unlock()` (line 68), the original `BiLock<T>` is returned and the lock is officially unlocked.
- On line 69 we perform `futures_util::lock::BiLock::reunite(other: T)`, which recovers the value of the lock and destroys the *two halves* of the BiLock references (presuming that `T` is the other half of the BiLock from the `BiLock::new()` call).

9 Networking

In this chapter, we will cover the following recipes:

- Setting up a basic HTTP server
- Configuring an HTTP server to perform echoing and routing
- Configuring an HTTP server to perform file serving
- Making requests to APIs
- Setting up a basic UDP Socket
- Configuring a UDP socket to perform echoing
- Setting up a secure connection via TLS

Introduction

Through the internet, the world is getting smaller every day. The web connects people in amazing ways. Countless services are available at your fingertips for free. Millions of people can use your apps without even installing it.

As a developer wanting to take advantage of this, porting your app to the internet can be quite easy if you have set your architecture up in a clean way. The only thing you need to change is the layer that interacts with the outside world.

This chapter is going to show you how to create this layer by allowing your application to accept requests, respond to them, and show you how to create requests to other web services on your own.

Setting up a basic HTTP server

Let's start our chapter by bringing the famous Hello World program into the 21st century by hosting it on a server. We are going to use the `hyper` crate for this, which is a strongly typed wrapper around all things HTTP. In addition to being one of the fastest HTTP implementations in the world (`https://www.techempower.com/benchmarks/#section=data-r15hw=phtest=plaintext`), it is used by nearly *all* major high-level frameworks (`https://github.com/flosse/rust-web-framework-comparison#high-level-frameworks`), the only exception being the ones that reimplemented it all on the extremely basic *stringly-typed* TCP library that Rust provides under `std::net::TcpStream`.

Getting ready

All `hyper` recipes work with `futures`, so you should read all of `Chapter 8`, *Working with Futures*, before continuing.

At the time of writing, `hyper` has not yet upgraded to `futures v0.2` (tracking issue: `https://github.com/hyperium/hyper/issues/1448`), so we going to use `futures v0.1`. This should be no problem in the future (no pun intended), as all relevant code is written in a way that should be compatible with `0.2` when it's released.

If some unexpected API change breaks the recipes, you will be able to find a fixed version of them at the book's GitHub repository (`https://github.com/jnferner/rust-standard-library-cookbook/tree/master/chapter-nine/src/bin`), which will always be updated in order to work with the newest versions of all libraries.

How to do it...

1. Create a Rust project to work on during this chapter with `cargo new chapter-nine`
2. Navigate into the newly created `chapter-nine` folder. For the rest of this chapter, we will assume that your command line is currently in this directory
3. Open the `Cargo.toml` file that has been generated for you
4. Under `[dependencies]`, add the following lines:

```
futures = "0.1.18"
hyper = "0.11.21"
```

5. If you want, you can go to futures' (`https://crates.io/crates/futures`) and hyper's (`https://crates.io/crates/hyper`) *crates.io* pages to check for the newest version and use that one instead
6. Inside the folder `src`, create a new folder called `bin`
7. Delete the generated `lib.rs` file, as we are not creating a library
8. In the folder `src/bin`, create a file called `hello_world_server.rs`
9. Add the following code and run it with `cargo run --bin hello_world_server`:

```
extern crate futures;
extern crate hyper;

use futures::future::Future;
use hyper::header::{ContentLength, ContentType};
use hyper::server::{const_service, service_fn, Http, Request, Response, Service};
use std::net::SocketAddr;

const MESSAGE: &str = "Hello World!";

fn main() {
  // [::1] is the loopback address for IPv6, 3000 is a port
  let addr = "[::1]:3000".parse().expect("Failed to parse address");
  run_with_service_function(&addr).expect("Failed to run web server");
}
```

Run a server by creating a service with `service_fn`:

```
fn run_with_service_function(addr: &SocketAddr) -> Result<(), hyper::Error> {
  // Hyper is based on Services, which are construct that
  // handle how to respond to requests.
  // const_service and service_fn are convenience functions
  // that build a service out of a closure
  let hello_world = const_service(service_fn(|_| {
    println!("Got a connection!");
    // Return a Response with a body of type hyper::Body
    Ok(Response::::new()
      // Add header specifying content type as plain text
      .with_header(ContentType::plaintext())
      // Add header specifying the length of the message in bytes
      .with_header(ContentLength(MESSAGE.len() as u64))
      // Add body with our message
      .with_body(MESSAGE))
  }));

  let server = Http::new().bind(addr, hello_world)?;
  server.run()
}
```

Run a server by manually creating a `struct` that implements `Service`:

```
// The following function does the same, but uses an explicitely created
// struct HelloWorld that implements the Service trait
fn run_with_service_struct(addr: &SocketAddr) -> Result<(), hyper::Error> {
  let server = Http::new().bind(addr, || Ok(HelloWorld))?;
  server.run()
}

struct HelloWorld;
impl Service for HelloWorld {
  // Implementing a server requires specifying all involved types
  type Request = Request;
  type Response = Response;
  type Error = hyper::Error;
  // The future that wraps your eventual Response
  type Future = Box<Future>;

  fn call(&self, _: Request) -> Self::Future {
    // In contrast to service_fn, we need to explicitely return a future
    Box::new(futures::future::ok(
      Response::new()
        .with_header(ContentType::plaintext())
        .with_header(ContentLength(MESSAGE.len() as u64))
        .with_body(MESSAGE),
    ))
  }
}
```

How it works...

In `main`, we first parse a string representing our IPv6 loopback address (think `localhost`) as an `std::net::SocketAddr`, which is a type holding an IP address and a port [13]. Granted, we could have used a constant for our address, but we are showing how to parse it from a string, because in a real application you will probably fetch the address from an environment variable, as shown in `Chapter 1`, *Learning the Basics*; *Interacting with environment variables*.

We then run our `hyper` server, which we create in `run_with_service_function` [17]. Let's take a look at that function by learning a bit about `hyper`.

The most fundamental trait in `hyper` is the `Service`. It is defined as follows:

```
pub trait Service where
    <Self::Future as Future>::Item == Self::Response,
    <Self::Future as Future>::Error == Self::Error, {
    type Request;
    type Response;
    type Error;
    type Future: Future;
    fn call(&self, req: Self::Request) -> Self::Future;
}
```

It should be easy to read the signature of `call`: It takes a `Request` and returns a `Future` of a `Response`. `hyper` uses this trait to answer to an incoming request. We generally have two ways to define a `Service`:

- Manually create a `struct` that implements `Service`, explicitly setting its associated types to whatever `call` returns
- Let a `Service` be built for you by passing a closure that returns a `Result` to `service_fn`, which you wrap in a `const_service`

Both variants result in the exact same thing, so this example contains both versions to give you a taste of them.

run_with_service_function uses the second style [22]. It returns a Result of a Response, which service_fn converts to a Future of Response because Result implements Future. service_fn then does some type deduction for us and creates a kind of Service. But we aren't done yet. You see, when hyper receives a new connection, it will not call our Service directly with the Request, but first makes a copy of it in order to handle every connection with its very own Service. This means that our Service must have the ability to create new instances of itself, which is indicated by the NewService trait. Luckily, we don't need to implement it ourselves either. The closure at the heart of our Service doesn't manage any state, so we can call it a constant function. Constants are very easy to copy, as all copies are guaranteed to be identical. We can mark our Service as constant by calling const_service on it, which basically just wraps the Service in an Arc and then implements NewService by simply returning a copy of it. But what exactly is our Service returning anyways?

Response<hyper::Body> creates a new HTTP response [25] and manages its body as a hyper::Body, which is a future Stream<Chunk>. A Chunk is just a piece of an HTTP message. This Response is a builder, so we can change the contents of it by calling various methods. In our code, we set its Content-Type header to plaintext, which is a hyper shortcut for the MIME type text/plain [27].

A MIME type is a label for data served over HTTP. It tells the client how to treat the data it receives. For example, most browsers will not render the message <p>Hello World!</p> as HTML unless it comes with the header Content-Type: text/html.

We also set its Content-Length header to the length (in bytes) of our message so the client knows how much data they should expect [29]. Finally, we set the message's body to the message, which then gets sent to the client as "Hello World!" [31].

Our service can now be bound to a new instance of hyper::server::Http, which we then run [34 and 35]. You can now open your browser of choice and point it to http://localhost:3000. If everything went right, you should be greeted by a Hello World! message.

The same thing would happen if we called run_with_service_struct instead, which uses a manually created Service instead [40]. A quick inspection of its implementation shows us the key differences to the last approach [45 to 63]:

```
struct HelloWorld;
impl Service for HelloWorld {
    type Request = Request;
    type Response = Response;
```

```
        type Error = hyper::Error;
        type Future = Box<Future<Item = Self::Response, Error =
        Self::Error>>;

        fn call(&self, _: Request) -> Self::Future {
            Box::new(futures::future::ok(
                Response::new()
                    .with_header(ContentType::plaintext())
                    .with_header(ContentLength(MESSAGE.len() as u64))
                    .with_body(MESSAGE),
            ))
        }
    }
```

As you can see, we need to explicitly specify the concrete type of basically everything [48 to 52]. We also can't simply return a `Result` in our `call` method and need to return the actual `Future`, wrapped in a `Box` [56], so we don't need to think about which exact flavor of `Future` we are using.

On the other hand, this approach has one big advantage over the other: It can manage state in the form of members. Because all `hyper` recipes in this chapter work with constant Services, that is Services that will return the same `Response` to equal Requests, we will use the first variant to create Services. This is simply a stylistic decision based on simplicity, as they are all small enough that it wouldn't be worth it to extract them into an own `struct`. In your projects, use whichever form suits the current use case best.

See also

- *Using the builder pattern* and *Interacting with environment variables* recipe in `Chapter 1`, *Learning the Basics*

Configuring an HTTP server to perform echoing and routing

We learned how to serve the same response forever, but that would get pretty dull after a while. In this recipe, you are going to learn how to read requests and respond to them individually. For this, we will use routing to differentiate between requests to different endpoints.

Getting ready

To test this recipe, you will need a way to easily send HTTP requests. An excellent free tool for this is Postman (`https://www.getpostman.com/`), which features a nice and self-explanatory UI. If you'd rather not download anything, you can use your terminal for this. If you're on Windows, you can open PowerShell and enter the following to do an HTTP request:

```
Invoke-WebRequest -UseBasicParsing <Your URL> -Method <Your method in
CAPSLOCK> -Body <Your message as a string>
```

So, if you wanted to POST the message `hello there, my echoing friend` to `http://localhost:3000/echo`, as you will be asked to later in the recipe, you'd need to enter the following command:

```
Invoke-WebRequest -UseBasicParsing http://localhost:3000/echo -Method POST
-Body "Hello there, my echoing friend"
```

On Unix systems, you can use cURL for that (`https://curl.haxx.se/`). The analog command is the following:

```
curl -X <Your method> --data <Your message> -g <Your URL>
```

cURL will resolve `localhost` to its entry in `/etc/hosts`. In some configurations, this will only be the IPv4 loopback address (`127.0.0.1`). In some others, you will have to use `ip6-localhost`. Check your `/etc/hosts` to find out what to use. In any case, an explicit `[::1]` will always work. As an example, the following command will again POST the message `hello there, my echoing friend` to `http://localhost:3000/echo`:

```
curl -X POST --data "Hello there my echoing friend" -g
"http://[::1]:3000/echo"
```

How to do it...

1. Open the `Cargo.toml` file that has been generated for you
2. Under `[dependencies]`, if you didn't do so in the last recipe, add the following lines:

```
futures = "0.1.18"
hyper = "0.11.21"
```

3. If you want, you can go to futures' (`https://crates.io/crates/futures`) and hyper's (`https://crates.io/crates/hyper`) *crates.io* pages to check for the newest version and use that one instead
4. In the folder `src/bin`, create a file called `echo_server_with_routing.rs`
5. Add the following code and run it with `cargo run --bin echo_server_with_routing`:

```
extern crate hyper;

use hyper::{Method, StatusCode};
use hyper::server::{const_service, service_fn, Http, Request,
Response};
use hyper::header::{ContentLength, ContentType};
use std::net::SocketAddr;

fn main() {
  let addr = "[::1]:3000".parse().expect("Failed to parse
  address");
  run_echo_server(&addr).expect("Failed to run web server");
}

fn run_echo_server(addr: &SocketAddr) -> Result<(),
hyper::Error> {
  let echo_service = const_service(service_fn(|req: Request| {
    // An easy way to implement routing is
    // to simply match the request's path
    match (req.method(), req.path()) {
      (&Method::Get, "/") => handle_root(),
      (&Method::Post, "/echo") => handle_echo(req),
      _ => handle_not_found(),
    }
  }));

  let server = Http::new().bind(addr, echo_service)?;
  server.run()
}
```

The functions are handling the routes:

```
type ResponseResult = Result<Response, hyper::Error>;
fn handle_root() -> ResponseResult {
  const MSG: &str = "Try doing a POST at /echo";
  Ok(Response::new()
    .with_header(ContentType::plaintext())
    .with_header(ContentLength(MSG.len() as u64))
    .with_body(MSG))
35  }
36
37  fn handle_echo(req: Request) -> ResponseResult {
38    // The echoing is implemented by setting the response's
39    // body to the request's body
40    Ok(Response::new().with_body(req.body()))
41  }
42
43  fn handle_not_found() -> ResponseResult {
44    // Return a 404 for every unsupported route
45    Ok(Response::new().with_status(StatusCode::NotFound))
46  }
```

How it works...

This recipe begins like the last one, so let's skip straight into the definition of our `Service`[14 to 22]:

```
|req: Request| {
    match (req.method(), req.path()) {
        (&Method::Get, "/") => handle_root(),
        (&Method::Post, "/echo") => handle_echo(req),
        _ => handle_not_found(),
    }
}
```

We are now using the `Request` parameter that the last recipe simply ignored.

Because Rust allows us to pattern match on tuples, we can directly differentiate between HTTP methods and path combinations. We then pass on the control flow of our program to dedicated route handlers, which in turn are responsible for returning the response.

TIP In bigger programs with tons of routes, we would not specify them all in one function, but spread them across namespaces and split them into subrouters.

The code for `handle_root` [29] looks nearly identical to the hello world `Service` from the last chapter, but instructs the caller to POST at the `/post` route.

Our match for said POST leads to `handle_echo` [37], which simply returns the request's body as the response's body [40]. You can try this for yourself by POSTing a message to `http://localhost:3000/echo`, as described in the *Getting ready* section. If everything goes right, your message will come right back at you.

Last, but not least, handle_not_found [43] is called when no routes matched. This time, we don't send a message back, but instead, return the possible most famous status code of the world: 404 Not Found [45].

Configuring an HTTP server to perform file serving

The last recipes were really useful for building web services, but let's take a look at how to do the thing HTTP was originally created for: serving HTML files to the web.

How to do it...

1. Open the Cargo.toml file that has been generated for you.
2. Under [dependencies], if you didn't do so in the last recipe, add the following lines:

```
futures = "0.1.18"
hyper = "0.11.21"
```

3. If you want, you can go to futures' (https://crates.io/crates/futures) and hyper's (https://crates.io/crates/hyper) *crates.io* pages to check for the newest version and use that one instead.
4. In the folder chapter-nine, create a folder called files.
5. In the folder files, create a file called index.html and add the following code to it:

```
<!doctype html>
<html>

<head>
    <link rel="stylesheet" type="text/css" href="/style.css">
    <title>Home</title>
</head>

<body>
    <h1>Home</h1>
    <p>Welcome. You can access other files on this web server
    aswell! Available links:</p>
    <ul>
```

```
            <li>
                <a href="/foo.html">Foo!</a>
            </li>
            <li>
                <a href="/bar.html">Bar!</a>
            </li>
        </ul>
    </body>

    </html>
```

6. In the folder `files`, create a file called `foo.html` and add the following code to it:

```
<!doctype html>
<html>

<head>
    <link rel="stylesheet" type="text/css" href="/style.css">
    <title>Foo</title>
</head>

<body>
    <p>Foo!</p>
</body>

</html>
```

7. In the folder `files`, create a file called `bar.html` and add the following code to it:

```
<!doctype html>
<html>

<head>
    <link rel="stylesheet" type="text/css" href="/style.css">
    <title>Bar</title>
</head>

<body>
    <p>Bar!</p>
</body>

</html>
```

8. In the folder `files`, create a file called `not_found.html` and add the following code to it:

```
<!doctype html>
<html>

<head>
    <link rel="stylesheet" type="text/css" href="/style.css">
    <title>Page Not Found</title>
</head>

<body>
    <h1>Page Not Found</h1>
    <p>We're sorry, we couldn't find the page you requested.</p>
    <p>Maybe it was renamed or moved?</p>
    <p>Try searching at the
        <a href="/index.html">start page</a>
    </p>
</body>

</html>
```

9. In the folder `files`, create a file called `invalid_method.html` and add the following code to it:

```
<!doctype html>
<html>

<head>
    <link rel="stylesheet" type="text/css" href="/style.css">
    <title>Error 405 (Method Not Allowed)</title>
</head>

<body>
    <h1>Error 405</h1>
    <p>The method used is not allowed for this URL</p>
</body>

</html>
```

10. In the folder `src/bin`, create a file called `file_server.rs`.
11. Add the following code and run it with `cargo run --bin echo_server_with_routing`:

```
extern crate futures;
extern crate hyper;

```

```
use hyper::{Method, StatusCode};
use hyper::server::{const_service, service_fn, Http, Request, Response};
use hyper::header::{ContentLength, ContentType};
use hyper::mime;
use futures::Future;
use futures::sync::oneshot;
use std::net::SocketAddr;
use std::thread;
use std::fs::File;
use std::io::{self, copy};

fn main() {
  let addr = "[::1]:3000".parse().expect("Failed to parse address");
  run_file_server(&addr).expect("Failed to run web server");
}

fn run_file_server(addr: &SocketAddr) -> Result<(), hyper::Error> {
  let file_service = const_service(service_fn(|req: Request| {
    // Setting up our routes
    match (req.method(), req.path()) {
      (&Method::Get, "/") => handle_root(),
      (&Method::Get, path) => handle_get_file(path),
      _ => handle_invalid_method(),
    }
  }));

  let server = Http::new().bind(addr, file_service)?;
  server.run()
}
```

The following are the route handlers:

```
// Because we don't want the entire server to block when serving a file,
// we are going to return a response wrapped in a future
type ResponseFuture = Box<Future>;
fn handle_root() -> ResponseFuture {
  // Send the landing page
  send_file_or_404("index.html")
}

fn handle_get_file(file: &str) -> ResponseFuture {
  // Send whatever page was requested or fall back to a 404 page
  send_file_or_404(file)
}

fn handle_invalid_method() -> ResponseFuture {
  // Send a page telling the user that the method he used is not supported
  let response_future = send_file_or_404("invalid_method.html")
    // Set the correct status code
    .and_then(|response| Ok(response.with_status(StatusCode::MethodNotAllowed)));
  Box::new(response_future)
}
```

The following is the code for the functions returning the futures with our files:

```
// Send a future containing a response with the requested file or a 404 page
fn send_file_or_404(path: &str) -> ResponseFuture {
  // Sanitize the input to prevent unwanted data access
  let path = sanitize_path(path);

  let response_future = try_to_send_file(&path)
    // try_to_send_file returns a future of Result<Response, io::Error>
    // turn it into a future of a future of Response with an error of hyper::Error
    .and_then(|response_result| response_result.map_err(|error| error.into()))
    // If something went wrong, send the 404 page instead
    .or_else(|_| send_404());
  Box::new(response_future)
}

// Return a requested file in a future of Result<Response, io::Error>
// to indicate whether it exists or not
type ResponseResultFuture = Box<Future, Error = hyper::Error>>;
fn try_to_send_file(file: &str) -> ResponseResultFuture {
  // Prepend "files/" to the file
  let path = path_on_disk(file);
  // Load the file in a separate thread into memory.
  // As soon as it's done, send it back through a channel
  let (tx, rx) = oneshot::channel();
  thread::spawn(move || {
    let mut file = match File::open(&path) {
      Ok(file) => file,
      Err(err) => {
        println!("Failed to find file: {}", path);
        // Send error through channel
        tx.send(Err(err)).expect("Send error on file not found");
        return;
      }
    };

    // buf is our in-memory representation of the file
    let mut buf: Vec = Vec::new();
    match copy(&mut file, &mut buf) {
      Ok(_) => {
        println!("Sending file: {}", path);
        // Detect the content type by checking the file extension
        // or fall back to plaintext
        let content_type = get_content_type(&path).unwrap_or_else(ContentType::plaintext);
        let res = Response::new()
          .with_header(ContentLength(buf.len() as u64))
          .with_header(content_type)
          .with_body(buf);
        // Send file through channel
        tx.send(Ok(res))
          .expect("Send error on successful file read");
      }
      Err(err) => {
        // Send error through channel
        tx.send(Err(err)).expect("Send error on error reading file");
      }
    };
  });
  // Convert all encountered errors to hyper::Error
  Box::new(rx.map_err(|error| io::Error::new(io::ErrorKind::Other, error).into()))
}

fn send_404() -> ResponseFuture {
  // Try to send our 404 page
  let response_future = try_to_send_file("not_found.html").and_then(|response_result| {
    Ok(response_result.unwrap_or_else(|_| {
      // If the 404 page doesn't exist, sent fallback text instead
      const ERROR_MSG: &str = "Failed to find \"File not found\"
      page. How ironic\n";
      Response::new()
        .with_status(StatusCode::NotFound)
        .with_header(ContentLength(ERROR_MSG.len() as u64))
        .with_body(ERROR_MSG)
    }))
  });
  Box::new(response_future)
}
```

The following are some helper functions:

```
 fn sanitize_path(path: &str) -> String {
   // Normalize the separators for the next steps
   path.replace("\\", "/")
     // Prevent the user from going up the filesystem
     .replace("../", "")
     // If the path comes straigh from the router,
     // it will begin with a slash
     .trim_left_matches(|c| c == '/')
     // Remove slashes at the end as we only serve files
     .trim_right_matches(|c| c == '/')
     .to_string()
 }

 fn path_on_disk(path_to_file: &str) -> String {
   "files/".to_string() + path_to_file
 }

 fn get_content_type(file: &str) -> Option {
   // Check the file extension and return the respective MIME type
  let pos = file.rfind('.')? + 1;
  let mime_type = match &file[pos..] {
    "txt" => mime::TEXT_PLAIN_UTF_8,
    "html" => mime::TEXT_HTML_UTF_8,
    "css" => mime::TEXT_CSS,
    // This list can be extended for all types your server
    should support
    _ => return None,
  };
  Some(ContentType(mime_type))
}
```

How it works...

Wow! That was a lot of files. Of course, the exact content of the HTML and CSS doesn't matter for this recipe, as we're going to be focused on Rust. We've put them all in the `files` folder because we are going to make its contents publicly accessible by name for any client.

The basics of the server setup are the same as with the echoing recipe: create a `Service` with `const_service` and `service_fn` [21], match the request's method and path, and then handle the routes in different functions. When looking at our return type, however, we can notice a difference [36]:

```
type ResponseFuture = Box<Future<Item = Response, Error =
hyper::Error>>;
```

We are no longer returning a `Response` directly, but instead, wrapping it in a `Future`. This allows us to not block the server when loading a file into memory; we can continue handling requests in the main thread while the file serving `Future` is run in the background.

When looking at our route handlers, you can see that they all use the `send_file_or_404` function. Let's take a look at it [56]:

```
fn send_file_or_404(path: &str) -> ResponseFuture {
    let path = sanitize_path(path);

    let response_future = try_to_send_file(&path)
        .and_then(|response_result| response_result.map_err(|error|
         error.into()))
        .or_else(|_| send_404());
    Box::new(response_future)
}
```

First, the function sanitizes our input. The implementation of `sanitize_path` [130 to 141] should be pretty straightforward. It filters out potential troublemakers so that a malicious client cannot do any shenanigans, such as requesting the file `localhost:3000/../../../../home/admin/.ssh/id_rsa`.

We then call `try_to_send_file` on the sanitized path [72]. We are going to look at that function in a minute, but for now, it's enough to look at its signature. It tells us that it returns a `Future` of a `Result` that can be a `Response` or an `io::Error`, as that's the error encountered on invalid filesystem access. We cannot return this `Future` directly, since we already told `hyper` that we are going to return a `Future` of `Response`, so we need to convert the types. If the file retrieving `Future` generated from `try_to_send_file` succeeded, we act on its item, which is a `Result<Response, io::Error>`.

Because `hyper::Error` implements `From<io::Error>`, we can convert them easily by calling `.into()` [63] (see `Chapter 5`, *Advanced Data Structures*; *Converting types*, for an introduction to the `From` trait). This will return a `Result<Response, hyper::Error>`. Because a `Future` is constructable from a `Result`, it will be implicitly converted to a `Future<Response, hyper::Error>` for us, which is exactly what we want. A little cherry on top is our handling of `try_to_send_file` returning an error, in which case we can safely assume that the file doesn't exist, so we return a `Future` with a custom `404 Not Found` page by calling `send_404()` [65]. Before looking at its implementation, let's check out `try_to_send_file` first [72].

First, we convert the requested path into a local filesystem path with `path_on_disk` [74], which is simply implemented as follows [144]:

```
"files/".to_string() + path_to_file
```

We created an own function for this so it will be easy for you to extend the filesystem logic. For example, for Unix systems, it is usual to put all static HTML in `/var/www/`, while Windows web servers usually put all of their data in their own installation folder. Or you may want to read a configuration file provided by the user and store its value in a `lazy_static`, as shown in `Chapter 5`, *Advanced Data Structures*; *Using lazy static variable*, and use that path instead. You can implement all of those rules in this function.

Back in `try_to_send_file`, we create a `oneshot::channel` to send data as a `Future` [77]. This concept is explained in detail in `Chapter 8`, *Working with Futures*; *Using the oneshot channel*.
The rest of the function now creates a new thread to load the file into memory in the background [78]. We first open the file [79] and return an error through the channel if it doesn't exist. We then copy the entire file into a local vector of bytes [91] and again propagate any error that might occur [107]. If the process of copying into RAM succeeded, we return a `Response` with the content of the file as its body [100]. Along the way, we have to figure out the file's appropriate MIME type [96], as promised in the recipe *Setting up a basic HTTP server*. For that, we simply match the extension of the file [147 to 158]:

```
fn get_content_type(file: &str) -> Option<ContentType> {
    let pos = file.rfind('.')? + 1;
    let mime_type = match &file[pos..] {
        "txt" => mime::TEXT_PLAIN_UTF_8,
        "html" => mime::TEXT_HTML_UTF_8,
        "css" => mime::TEXT_CSS,
        _ => return None,
    };
    Some(ContentType(mime_type))
}
```

You may think this implementation is pretty lazy and that there should be a better way, but, trust me, this is exactly how all big web servers do it. Case in point, you can find `nginx` (`https://nginx.org/en/`) mime detection algorithm here: `https://github.com/nginx/nginx/blob/master/conf/mime.types`. If you plan on serving new file types, you can extend the `match` for their extensions. The `nginx` source is a good resource for this.

`get_content_type` returns `None` if there was no match [155] instead of a default content type, so that every caller can decide on a default for themselves. In `try_to_send_file`, we use `.unwrap_or_else(ContentType::plaintext);` [96] to set the fallback MIME type to `text/plain`.

The last unexplained function left in our example is `send_404`, which we use a lot as a fallback. You can see that all it really does is call `try_to_send_file` on the 404 page [117] and on error send a static message instead [124].

The fallback in `send_404` really shows us the beauty in Rust's error handling concept. Because strongly typed errors are part of a function's signature, as opposed to languages such as C++, where you never know who might throw an exception, you are forced to consciously handle the error cases. Try to remove `and_then` and its associated closure and you'll see that the compiler doesn't let you compile your program because you didn't handle the `Result` of `try_to_send_file` in any way.

Go ahead now and see the results of our file server with your own eyes by pointing your browser to `http://localhost:3000/`.

There's more...

Despite being relatively easy to understand, our implementation of `try_to_send_file` is not endlessly scalable. Imagine serving and loading huge files into memory for millions of clients at the same time. That would bring your RAM to its limits pretty quickly. A more scalable solution is to send the file in chunks, that is part by part, so that you only need to hold a small part of it in memory at any given time. To implement this, you'll need to copy the contents of your file to a limited `[u8]` buffer with a fixed size and send that through an additional channel as an instance of `hyper::Chunk`, which implements `From<Vec<T>>`.

See also

- *Converting types into each other* and *Creating lazy static variables* recipe in `Chapter 5`, *Advanced Data Structures*
- *Using the oneshot channel* recipe in `Chapter 8`, *Working with Futures*

Making requests to APIs

Our last destination in this chapter brings us away from the server in favor of the other party participating in internet communication: the client. We will use `reqwest`, which is built around `hyper`, to create HTTPS requests to web services and parse their data into nicely usable Rust structures. You can also use the content of this recipe to write integration tests for your own web services.

How to do it...

1. Open the `Cargo.toml` file that has been generated for you
2. Under `[dependencies]`, if you didn't do so in the last recipe, add the following lines:

```
reqwest = "0.8.5"
serde = "1.0.30"
serde_derive = "1.0.30"
```

3. If you want, you can go to `request`'s (`https://crates.io/crates/reqwest`), `serde`'s (`https://crates.io/crates/serde`), and `serde_derive`'s (`https://crates.io/crates/serde_derive`) *crates.io* pages to check for the newest versions and use those ones instead
4. In the folder `src/bin`, create a file called `making_requests.rs`
5. Add the following code and run it with `cargo run --bin making_requests`:

```
extern crate reqwest;
#[macro_use]
extern crate serde_derive;

use std::fmt;

#[derive(Serialize, Deserialize, Debug)]
// The JSON returned by the web service that hands posts out
// it written in camelCase, so we need to tell serde about that
#[serde(rename_all = "camelCase")]
struct Post {
  user_id: u32,
  id: u32,
  title: String,
  body: String,
}

#[derive(Serialize, Deserialize, Debug)]
#[serde(rename_all = "camelCase")]
struct NewPost {
  user_id: u32,
  title: String,
  body: String,
}

#[derive(Serialize, Deserialize, Debug)]
#[serde(rename_all = "camelCase")]
// The following struct could be rewritten with a builder
struct UpdatedPost {
  #[serde(skip_serializing_if = "Option::is_none")]
  user_id: Option,
  #[serde(skip_serializing_if = "Option::is_none")]
  title: Option,
  #[serde(skip_serializing_if = "Option::is_none")]
  body: Option,
}

struct PostCrud {
  client: reqwest::Client,
  endpoint: String,
}

impl fmt::Display for Post {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    write!(
      f,
      "User ID: {}\nID: {}\nTitle: {}\nBody: {}\n",
      self.user_id, self.id, self.title, self.body
    )
  }
}
```

The following code shows the requests being implemented:

```
impl PostCrud {
  fn new() -> Self {
    PostCrud {
      // Build an HTTP client. It's reusable!
      client: reqwest::Client::new(),
      // This is a link to a fake REST API service
      endpoint: "https://jsonplaceholder.typicode.com/posts".to_string(),
    }
  }

  fn create(&self, post: &NewPost) -> Result<Post, reqwest::Error> {
    let response = self.client.post(&self.endpoint).json(post).send()?.json()?;
    Ok(response)
  }

  fn read(&self, id: u32) -> Result<Post, reqwest::Error> {
    let url = format!("{}/{}", self.endpoint, id);
    let response = self.client.get(&url).send()?.json()?;
    Ok(response)
  }

  fn update(&self, id: u32, post: &UpdatedPost) -> Result<Post, reqwest::Error> {
    let url = format!("{}/{}", self.endpoint, id);
    let response = self.client.patch(&url).json(post).send()?.json()?;
    Ok(response)
  }

  fn delete(&self, id: u32) -> Result<(), reqwest::Error> {
    let url = format!("{}/{}", self.endpoint, id);
    self.client.delete(&url).send()?;
    Ok(())
  }
}
```

The following code shows us using our CRUD client:

```
fn main() {
  let post_crud = PostCrud::new();
  let post = post_crud.read(1).expect("Failed to read post");
  println!("Read a post:\n{}", post);

  let new_post = NewPost {
    user_id: 2,
    title: "Hello World!".to_string(),
    body: "This is a new post, sent to a fake JSON API
```

```
          server.\n".to_string(),
96      };
97      let post = post_crud.create(&new_post).expect("Failed to
        create post");
98      println!("Created a post:\n{}", post);
99
100     let updated_post = UpdatedPost {
101       user_id: None,
102       title: Some("New title".to_string()),
103       body: None,
104     };
105     let post = post_crud
106       .update(4, &updated_post)
107       .expect("Failed to update post");
108     println!("Updated a post:\n{}", post);
109
110     post_crud.delete(51).expect("Failed to delete post");
111 }
```

How it works...

At the top, we define our structures. `Post`[11], `NewPost` [20], and `UpdatedPost` [29] all just represent convenient ways to handle the different requirements of the API. The particular JSON API we are interacting with uses camelCase variables, so we need to specify this on every `struct`, otherwise `serde` won't be able to parse them correctly [10, 19 and 27].

Because the `PATCH` method we're communicating with doesn't accept null values on unchanged variables, we mark them all in `UpdatedPost` as not serialized when equal to `None` [30, 32 and 34]:

```
#[serde(skip_serializing_if = "Option::is_none")]
```

Additionally, we implement the `fmt::Display` trait on `Post`, so we can print it nicely [43 to 51].

But enough about our models; let's take a look at `PostCrud` [53]. Its purpose is to abstract a CRUD (Create, Read, Update, Delete) service. For this, it is equipped with a reusable HTTP client via `reqwest::Client` [57] and a mock JSON API endpoint from `https://jsonplaceholder.typicode.com/`

Its methods show you how easy `reqwest` is to use: you simply use the required HTTP method directly as a function on the client, pass optional data to it, which it will automatically deserialize for you with `.json()`, `.send()` the request, and then parse the response again as JSON with a second call to `.json()` [64].

There's more...

Of course, `reqwest` is able to work with non-JSON-based web services as well. It has various methods for this such as `query`, which adds an array of key-value queries to the URL, or `form`, which will add `url-encoded` form bodies to the request. While using all these methods, `reqwest` will manage the headers for you, but you can manage them however explicitly you want using the `headers` method.

See also

- *Using the builder pattern* recipe in `Chapter 1`, *Learning the Basics*
- *Serialization basics with Serde* recipe in `Chapter 4`, *Serialization*

10
Using Experimental Nightly Features

In this chapter, we will cover the following recipes:

- Iterating over an inclusive range
- Returning abstract types
- Composing functions
- Filtering strings efficiently
- Stepping through an iterator in regular intervals
- Benchmarking your code
- Using generators

Introduction

This final chapter leads us to the most important experimental features in Rust, provided on the newest `nightly` toolchain. As of the time of writing, this is `rustc 1.25.0-nightly`. If you are using `rustup` (`https://rustup.rs/`), you can set it as your default toolchain like this:

```
rustup default nightly
```

These recipes will ensure that you stay ahead in your knowledge of Rust, and are ready to use them effectively once they are stabilized, or right now in your own unstable apps.

All of the recipes in this chapter have varying stability guarantees. Many will undergo drastic changes before they land in the `stable` toolchain, and others are nearly done. This means that some of the example code provided is expected not to work on your newest `nightly`. When this happens, you can find help in two places:

- If the feature is still experimental, it will have an entry in *The Unstable Book* (`https://doc.rust-lang.org/unstable-book/`), and a link to the relevant GitHub issue and its surrounding discussion
- If the feature has been stabilized, there is a good chance you will find it listed in the appendix of *The Rust Programming Language, Second Edition* (`https://doc.rust-lang.org/stable/book/second-edition/appendix-06-newest-features.html`)

Iterating over an inclusive range

We begin the chapter with a small feature that can make your code a bit more readable. The inclusive range syntax (`..=`) will create a range up to a value *including* it. This helps you eliminate ugly instances of things like `n .. m+1` by rewriting them as `n ..= m`.

How to do it...

1. Create a Rust project to work on during this chapter with `cargo new chapter-ten`.
2. Navigate into the newly-created `chapter-ten` folder. For the rest of this chapter, we will assume that your command line is currently in this directory.
3. Delete the generated `lib.rs` file, as we are not creating a library.
4. Inside the `src` folder, create a new folder called `bin`.
5. In the `src/bin` folder, create a file called `inclusive_range.rs`.
6. Add the following code, and run it with `cargo run --bin inclusive_range`:

```
#![feature(inclusive_range_syntax)]

fn main() {
  // Retrieve the entire alphabet in lower and uppercase:
  let alphabet: Vec<_> = (b'A' .. b'z' + 1) // Start as u8
    .map(|c| c as char)      // Convert all to chars
    .filter(|c| c.is_alphabetic()) // Filter only alphabetic
    chars
8       .collect(); // Collect as Vec
9     println!("alphabet: {:?}", alphabet);
10
11    // Do the same, but using the inclusive range syntax:
12    let alphabet: Vec<_> = (b'A' ..= b'z') // Start as u8
13      .map(|c| c as char)       // Convert all to chars
14      .filter(|c| c.is_alphabetic()) // Filter only alphabetic
         chars
15      .collect(); // Collect as Vec
16    println!("alphabet: {:?}", alphabet);
17  }
```

How it works...

In this example, the code is the promised rewrite of a snippet in `Chapter 2`, *Working with Collections*; *Accessing collections as iterators*.

The traditional range syntax (`n .. m`) is exclusive, meaning that `0 .. 5` will only include the numbers 0, 1, 2, 3, and 4. This is good for uses cases where you count up to a length of something, but in our case, we want to iterate over the alphabet, including Z [5]. The inclusive range syntax (`n ..= m`) helps us by including the last element, so that `0 ..= 5` will yield 0, 1, 2, 3, 4, and 5.

See also

- *Accessing collections as iterators* recipe in `Chapter 2`, *Working with Collections*

Returning abstract types

Remember when we used `Box` to create trait objects in order to hide the exact implementation returned, and instead only give guarantees about implemented traits? That required us to accept some overhead, as a `Box` allocates its resources on the heap; however, on the current `nightly`, things are different. You can use the `impl trait` syntax introduced in this recipe to return objects as their trait directly on the stack, all without boxes. At the moment, this only works for returned types, but the syntax is planned to be extended to most places where you could write a concrete type.

How to do it...

1. Open the `Cargo.toml` file that has been generated earlier for you.
2. In the `bin` folder, create a file called `return_abstract.rs`.
3. Add the following code, and run it with `cargo run --bin return_abstract`:

```
#![feature(conservative_impl_trait)]

trait Animal {
  fn do_sound(&self);
}

struct Dog;
impl Animal for Dog {
  fn do_sound(&self) {
    println!("Woof");
  }
}

fn main() {
  // The caller doesn't know which exact object he gets
  // He knows only that it implements the Animal trait
  let animal = create_animal();
  animal.do_sound();

  for word in caps_words_iter("do you feel lucky, punk?") {
    println!("{}", word);
  }

  let multiplier = create_multiplier(23);
  let result = multiplier(3);
  println!("23 * 3 = {}", result);
}

// The impl trait syntax allows us to use abstract return types
// This means that we don't specify which exact struct we
return
// but which trait(s) it implements
fn create_animal() -> impl Animal {
  Dog {}
}

// Any iterator can be returned as an abstract return type
```

```
37  fn caps_words_iter<'a>(text: &'a str) -> impl Iterator + 'a {
38    // Return an iterator over every word converted into ALL_CAPS
39    text.trim().split(' ').map(|word| word.to_uppercase())
40  }
41
42  // Same goes for closures
43  fn create_multiplier(a: i32) -> impl Fn(i32) -> i32 {
44    move |b| a * b
45  }
```

How it works...

If you've read `Chapter 5`, *Advanced Data Structures*; *Boxing data*, this recipe doesn't need much explanation. By returning an `impl trait`, we tell the caller of a function to not care about the specific struct that is returned, and that its only guarantee about it is that it implements some trait. In this sense, abstract return types work like the trait objects discussed in the said recipe, with the added bonus of being way faster, as they don't have any overhead. This is useful for returning iterators [37] and closures [43], which we adapted from the recipe about boxes, but also to hide implementation details. Consider our function `create_animal`[32]. A caller will only care that it returns a struct that implements `Animal`, but not which exact animal. If a `Dog` [7] doesn't prove to be the right thing because of changing requirements, you can create a `Cat`, and return that one without touching the rest of the code, as it all just depends on `Animal`. This is a form of *dependency inversion* (`https://en.wikipedia.org/wiki/Dependency_inversion_principle`).

The `conservative` in `conservative_impl_trait` [1] tells us that this is just a part of a bigger feature. At the moment, you can only use it in return types of functions. In the future, you'll be able to use it in traits, constraints, and bindings as well.

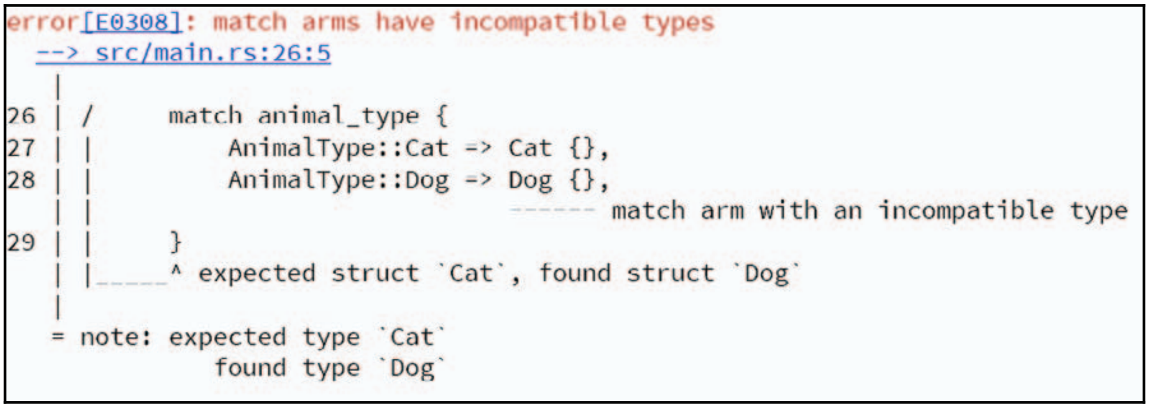
There's more...

Of all the recipes in this chapter, this one is probably the most stable, as it is being considered for immediate stabilization. The discussion can be found at `https://github.com/rust-lang/rust/issues/34511`.

While you can use abstract return types for some forms of dependency inversion, you cannot use them for traditional Java-style factories that return different objects depending on a parameter. This is because abstract return types only hide the specific struct returned on the outside of the function, but still internally rely on a specific return value. Because of this, the following code will not compile:

```
trait Animal {
    fn do_sound(&self);
}

struct Dog;
impl Animal for Dog {
    fn do_sound(&self) {
        println!("Woof");
    }
}
struct Cat;
impl Animal for Cat {
    fn do_sound(&self) {
        println!("Meow");
    }
}

enum AnimalType {
    Dog,
    Cat,
}

fn create_animal(animal_type: AnimalType) -> impl Animal {
    match animal_type {
        AnimalType::Cat => Cat {},
        AnimalType::Dog => Dog {},
    }
}
```

While the outside world doesn't know which animal is going to be returned by `create_animal`, the function itself needs a specific return type internally. Because the first possible return from our match is an instance of `Cat`, `create_animal` assumes that we are going to return no other type. We break that expectation in the next line by returning a `Dog`, so the compiler fails:



```
error[E0308]: match arms have incompatible types
  --> src/main.rs:26:5
   |
26 | /     match animal_type {
27 | |         AnimalType::Cat => Cat {},
28 | |         AnimalType::Dog => Dog {},
   | |                            ------ match arm with an incompatible type
29 | |     }
   | |_____^ expected struct `Cat`, found struct `Dog`
   |
   = note: expected type `Cat`
              found type `Dog`
```

If we want this factory to compile, we need to resort back to `Box` again:

```
fn create_animal(animal_type: AnimalType) -> Box<Animal> {
    match animal_type {
        AnimalType::Cat => Box::new(Cat {}),
        AnimalType::Dog => Box::new(Dog {}),
    }
}
```

By the way, this is also exactly what Java does under the hood.

For most purposes, you won't need a factory such as the one presented here. It is considered more idiomatic to use generics with trait bounds.

See also

- *Boxing data* recipe in `Chapter 5`, *Advanced Data Structures*

Composing functions

Because we have now learned how to return arbitrary closures with no overhead, we can combine that with macros that accept any number of parameters (`Chapter 1`, *Learning the Basics*; *Accepting a variable number of arguments*) to create an easy way to chain actions as you would be used to in functional languages such as *Haskell*.

How to do it...

1. Open the `Cargo.toml` file that has been generated earlier for you.
2. In the `bin` folder, create a file called `compose_functions.rs`.
3. Add the following code, and run it with `cargo run --bin compose_functions`:

```
#![feature(conservative_impl_trait)]

// The compose! macro takes a variadic amount of closures and returns
// a closure that applies them all one after another
macro_rules! compose {
  ( $last:expr ) => { $last };
  ( $head:expr, $ ($tail:expr), +) => {
    compose_two($head, compose!($ ($tail), +))
  };
}

// compose_two is a helper function used to
// compose only two closures into one
fn compose_two<FunOne, FunTwo, Input, Intermediate, Output>(
  fun_one: FunOne,
  fun_two: FunTwo,
) -> impl Fn(Input) -> Output
where
  FunOne: Fn(Input) -> Intermediate,
  FunTwo: Fn(Intermediate) -> Output,
{
  move |x| fun_two(fun_one(x))
}

fn main() {
  let add = |x| x + 2.0;
  let multiply = |x| x * 3.0;
  let divide = |x| x / 4.0;
  // itermediate(x) returns ((x + 2) * 3) / 4
  let intermediate = compose!(add, multiply, divide);

  let subtract = |x| x - 5.0;
  // finally(x) returns (((x + 2) * 3) / 4) - 5
  let finally = compose!(intermediate, subtract);

36    println!("(((10 + 2) * 3) / 4) - 5 is: {}", finally(10.0));
37  }
```

How it works...

By the usage of `compose!` in main [30 and 34], you should see clearly what it does: it takes as many closures as you want, and combines them into a new closure that runs them one by one. This is really useful for runtime user-driven functionality composition.

The macro is implemented similarly to the standard macro for variable arguments from `Chapter 1`, *Learning the Basics*; *Accepting a variable number of arguments*, with its edge case being a single closure [6]. When encountering more, it will recursively go through them and combine them in pairs by calling the helper function `compose_two` [14]. Usually, type parameters are written as a single character, but we are using full words for them in this recipe for readability reasons, as there are quite a number of types involved. The type constraints used should illustrate how the types are used pretty well [18 to 20]:

```
where
    FunOne: Fn(Input) -> Intermediate,
    FunTwo: Fn(Intermediate) -> Output,
```

`FunOne` is a closure that takes an `Input`, turns it into an `Intermediate`, and passes it to `FunTwo`, which returns an `Output`. As you can see from the implementation, the only thing we do is call `fun_one` on a value, and then call `fun_two` on its returned value [22]:

```
move |x| fun_two(fun_one(x))
```

See also

- *Accepting a variable number of arguments* recipe in `Chapter 1`, *Learning the Basics*

Filtering strings efficiently

While you can filter characters out of a `String` on the stable channel already, this requires creating a new `String` with the filtered characters. On `nightly`, you can do this in place of the same `String`, helping you a lot with performance if you need to perform this kind of action many, many times, or on very large strings.

How to do it...

1. Open the `Cargo.toml` file that has been generated earlier for you.
2. In the `bin` folder, create a file called `retain_string.rs`.
3. Add the following code, and run it with `cargo run --bin retain_string`:

```rust
#![feature(string_retain)]

fn main() {
  let mut some_text = "H_el_l__o_ ___Wo_r__l_d_".to_string();
  println!("Original text: {}", some_text);
  // retain() removes all chars that don't fulfill a
  // predicate in place, making it very efficient
  some_text.retain(|c| c != '_');
  println!("Text without underscores: {}", some_text);
  some_text.retain(char::is_lowercase);
  println!("Text with only lowercase letters: {}", some_text);

  // Before retain, you had to filter the string as an iterator
  over chars
  // This will however create a new String, generating overhead
  let filtered: String = "H_el_l__o_ ___Wo_r__l_d_"
    .chars()
    .filter(|c| *c != '_')
    .collect();
  println!("Text filtered by an iterator: {}", filtered);
}
```

How it works...

In `Chapter 2`, *Working with Collections*; *Using a Vector*, we learned about `Vec::retain`, which filters a vector in place. On the `nightly` toolchain, this functionality has arrived in `String` and works the same way, as if a `String` was a `Vec<char>`—which, if you think about it, it really is.

The functionality of filtering a `String` was always there, but it required going over the string as an `Iterator` and creating a new `String` with the filtered characters; or worse yet, converting a `String` to a newly-created `Vec<char>`, using `retain` on it, and then converting the chars back into another newly-created `String`.

See also

- *Using a vector* recipe in `Chapter 2`, *Working with Collections*

Stepping through an iterator in regular intervals

Have you ever wanted to step through data by only looking at every nth item? On stable Rust, the best solution to this problem is using the third-party crate `itertools` (`https://crates.io/crates/itertools`), which brings you a whole lot of iterator goodies, or allows you to code the functionality yourself; however, you have a built-in `step_by` method doing exactly this.

How to do it...

1. Open the `Cargo.toml` file that has been generated earlier for you.
2. In the `bin` folder, create a file called `iterator_step_by.rs`.
3. Add the following code, and run it with `cargo run --bin iterator_step_by`:

```
#![feature(iterator_step_by)]

fn main() {
  // step_by() will start on the first element of an iterator,
  // but then skips a certain number of elements on every
  iteration
  let even_numbers: Vec<_> = (0..100).step_by(2).collect();
  println!("The first one hundred even numbers: {:?}",
  even_numbers);

  // step_by() will always start at the beginning.
  // If you need to skip the first few elements as well, use
  skip()
  let some_data = ["Andrei", "Romania", "Giuseppe", "Italy",
  "Susan", "Britain"];
  let countries: Vec<_> =
  some_data.iter().skip(1).step_by(2).collect();
13    println!("Countries in the data: {:?}", countries);
14
15    let grouped_stream = "Aaron 182cm 70kg Alice 160cm 90kg Bob
      197cm 83kg";
16    let weights: Vec<_> = grouped_stream
17      .split_whitespace()
18      .skip(2)
19      .step_by(3)
20      .collect();
21    println!("The weights of the people are: {:?}", weights);
22  }
```

How it works...

This recipe shines when you are being handed an unstructured data stream that follows a certain pattern. For example, sometimes some old APIs that don't use JSON, or other programs you might want to interact with, hand you streams of data that are grouped by position, like the data we stored in `grouped_stream`[15], which follows the following pattern:

```
person0 height0 weight0 person1 height1 weight1 person2 height2 weight2
```

`step_by` lets us parse this structure very easily. It works by handing you current element, and then skipping a certain amount of elements on every iteration. In our example, we parse `grouped_stream` by first creating an iterator over every substring that is not whitespace with `split_whitespace` [17], then, because we are only interested in the weights, `skip` the first two elements (`"Aaron"` and `"182cm"`), which places our iterator at `"70kg"`. We then tell the iterator to only look at every third element from now on with `step_by(3)` [19], resulting in us iterating over `"70kg"`, `"90kg"`, and `"83kg"`. Finally, we `collect` the elements into a vector [20].

See also

- *Using a string* and *Accessing collections as iterators* recipes in `Chapter 2`, *Working with Collections*

Benchmarking your code

The Rust project has developed a testing crate for the compiler itself. Because it includes some quite useful features, most importantly a benchmarker, it is accessible on nightly builds as the built-in `test` crate. Because it gets shipped with every nightly build, you don't need to add it to your `Cargo.toml` to use it.

The `test` crate is marked unstable because of its tight coupling to the compiler.

How to do it...

1. Open the `Cargo.toml` file that has been generated earlier for you.
2. In the `bin` folder, create a file called `benchmarking.rs`.
3. Add the following code, and run it with `cargo bench`:

```
#![feature(test)]
// The test crate was primarily designed for
// the Rust compiler itself, so it has no stability guaranteed
extern crate test;

pub fn slow_fibonacci_recursive(n: u32) -> u32 {
  match n {
    0 => 0,
    1 => 1,
    _ => slow_fibonacci_recursive(n - 1) +
    slow_fibonacci_recursive(n - 2),
  }
}

pub fn fibonacci_imperative(n: u32) -> u32 {
  match n {
    0 => 0,
    1 => 1,
    _ => {
      let mut penultimate;
      let mut last = 1;
      let mut fib = 0;
      for _ in 0..n {
        penultimate = last;
        last = fib;
        fib = penultimate + last;
      }
      fib
    }
  }
}

pub fn memoized_fibonacci_recursive(n: u32) -> u32 {
  fn inner(n: u32, penultimate: u32, last: u32) -> u32 {
    match n {
      0 => penultimate,
      1 => last,
      _ => inner(n - 1, last, penultimate + last),
    }
  }
  inner(n, 0, 1)
}

pub fn fast_fibonacci_recursive(n: u32) -> u32 {
  fn inner(n: u32, penultimate: u32, last: u32) -> u32 {
    match n {
      0 => last,
      _ => inner(n - 1, last, penultimate + last),
    }
  }
  match n {
    0 => 0,
    _ => inner(n - 1, 0, 1),
  }
}
```

4. Running benchmarks:

```
#[cfg(test)]
mod tests {
  use super::*;
  use test::Bencher;

  // Functions annotated with the bench attribute will
  // undergo a performance evaluation when running "cargo bench"
  #[bench]
  fn bench_slow_fibonacci_recursive(b: &mut Bencher) {
    b.iter(|| {
      // test::block_box is "black box" for the compiler and LLVM
      // Telling them to not optimize a variable away
      let n = test::black_box(20);
      slow_fibonacci_recursive(n)
70      });
71    }
72
73    #[bench]
74    fn bench_fibonacci_imperative(b: &mut Bencher) {
75      b.iter(|| {
76        let n = test::black_box(20);
77        fibonacci_imperative(n)
78      });
79    }
80
81    #[bench]
82    fn bench_memoized_fibonacci_recursive(b: &mut Bencher) {
83      b.iter(|| {
84        let n = test::black_box(20);
85        memoized_fibonacci_recursive(n)
86      });
87    }
88
89    #[bench]
90    fn bench_fast_fibonacci_recursive(b: &mut Bencher) {
91      b.iter(|| {
92        let n = test::black_box(20);
93        fast_fibonacci_recursive(n)
94      });
95    }
96  }
```

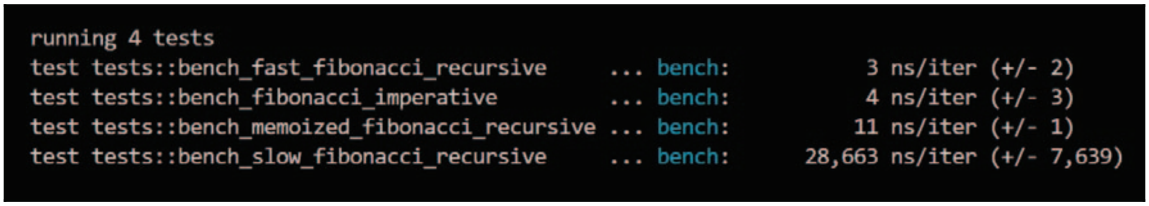
How it works...

The bread and butter of the `test` crate is the `Bencher` struct [59]. An instance of it is passed automatically to every function annotated with the `#[bench]` attribute [63] when running `cargo bench`. Its `iter` method takes a closure [65], and runs it multiple times to determine how long one iteration of it takes. While doing this, it also discards time measurements that are far off the others to eliminate one-off extremes.

Another useful part of the `test` crate is its `black_box` struct [68], which wraps any value and tells the compiler and LLVM to not optimize it away, no matter what. If we didn't use it in our benchmarks, they might get optimized away and result in a rather optimistic and unhelpful measurement of 0 ns/iter, or zero nanoseconds per execution of the closure.

We can use the tools at our disposal to test out some theories. Remember the recursive **Fibonacci** implementation discussed in `Chapter 7`, *Parallelism and Rayon*; *Running two operations together*? Well, it is repeated here as `slow_fibonacci_recursive`[6].

This implementation is slow because both calls to `slow_fibonacci_recursive(n - 1)` and `slow_fibonacci_recursive(n - 2)` need to recalculate *all* values individually. Worse yet, every call also splits up into a call to `slow_fibonacci_recursive(n - 1)` and `slow_fibonacci_recursive(n - 2)` once more, recalculating everything again and again! In terms of *Big O*, this is an efficiency of $O(1.6^n)$ (proof at `https://stackoverflow.com/a/360773/5903309`). In comparison, the imperative algorithm, `fibonacci_imperative` [14] is a simple loop, so it's at $O(n)$. By this theory, it should be *a lot* faster than the slow, recursive one. Running `cargo bench` lets us verify these assertions easily:



```
running 4 tests
test tests::bench_fast_fibonacci_recursive     ... bench:           3 ns/iter (+/- 2)
test tests::bench_fibonacci_imperative         ... bench:           4 ns/iter (+/- 3)
test tests::bench_memoized_fibonacci_recursive ... bench:          11 ns/iter (+/- 1)
test tests::bench_slow_fibonacci_recursive     ... bench:      28,663 ns/iter (+/- 7,639)
```

What a difference! On my computer, the slow, recursive implementation is more than 7,000 times slower than the imperative one! Surely, we can do better.

StackOverflow user **Boiethios** helpfully provided us with `memoized_fibonacci_recursive` at `https://stackoverflow.com/a/49052806/5903309`. As the name suggests, this implementation uses a concept known as *memoization*. This means that the algorithm has some way of passing around already-calculated values. An easy way of doing this would be to pass around a `HashMap` with all calculated values, but that would again bring its own overhead, as it operates on the heap. Instead, we go for the route of *accumulators*. This means that we just pass the relevant values directly as parameters, which in our case are `penultimate`, which represents the Fibonacci of `n-2`, and `last`, which represents the Fibonacci of `n-1`. If you want to read more about these functional concepts, check out `http://www.idryman.org/blog/2012/04/14/recursion-best-practices/`.

Checking the benchmarks, we can see that we managed to improve the algorithm quite a bit with `memoized_fibonacci_recursive`. But it's still a bit slower than `fibonacci_imperative`. One of many possible ways to further improve the algorithm is by extracting the `n == 1` match that would be checked in every recursive call outside, as demonstrated in `fast_fibonacci_recursive` [43], which clocks in at a great three nanoseconds per iteration!

There's more...

Our implementation also employs another optimization: *tail call optimization*, or *TCO* for short. In oversimplified terms, TCO happens when the compiler is able to rewrite a recursive algorithm into an imperative one. More generally, TCO is when the compiler can compile a recursive call into a form that doesn't add a new stack frame per call, and as a consequence, can't cause a stack overflow (not the website, but the error). For a good discussion on the topic, see `https://stackoverflow.com/questions/310974/what-is-tail-call-optimization`.

Although Rust doesn't support TCO per se (see the RFC at `https://github.com/rust-lang/rfcs/issues/271`), the lower-level LLVM does. It requires the last call of a function to be a call to itself. The last line of `inner` [40] is a call to `inner`, so it's eligible for TCO.

This is somewhat hard to guarantee in bigger Rust algorithms, though, as objects implementing the `Drop` trait will inject a call to `drop()` at the end of a function, removing any possibility of TCO.

See also

- *Running two operations together* recipe in `Chapter 7`, *Parallelism and Rayon*

Using generators

The biggest concept that is still not quite usable yet in Rust is *easy async*. One reason for that is a lack of compiler support for certain things, which is being worked on right now. One important part of the road to async is *generators*, which are implemented similarly to how they're used in C# or Python.

How to do it...

1. Open the `Cargo.toml` file that has been generated earlier for you.
2. In the `bin` folder, create a file called `generator.rs`.
3. Add the following code, and run it with `cargo run --bin generator`:

```
1   #![feature(generators, generator_trait,
    conservative_impl_trait)]
```

```

fn main() {
  // A closure that uses the keyword "yield" is called a generator
  // Yielding a value "remembers" where you left off
  // when calling .resume() on the generator
  let mut generator = || {
    yield 1;
    yield 2;
  };
  if let GeneratorState::Yielded(value) = generator.resume() {
    println!("The generator yielded: {}", value);
  }
  if let GeneratorState::Yielded(value) = generator.resume() {
    println!("The generator yielded: {}", value);
  }
  // When there is nothing left to yield,
  // a generator will automatically return an empty tuple
  if let GeneratorState::Complete(value) = generator.resume() {
    println!("The generator completed with: {:?}", value);
  }

  // At the moment, you can return a different type
  // than you yield, although this feature is considered for removal
  let mut generator = || {
    yield 100;
    yield 200;
    yield 300;
    "I'm a string"
  };
  loop {
    match generator.resume() {
      GeneratorState::Yielded(value) => println!("The generator yielded: {}", value),
      GeneratorState::Complete(value) => {
        println!("The generator completed with: {}", value);
        break;
      }
    }
  }

  // Generators are great for implementing iterators.
  // Eventually, all Rust iterators are going to be rewritten with generators
  let fib: Vec<_> = fibonacci().take(10).collect();
  println!("First 10 numbers of the fibonacci sequence: {:?}", fib);
}

// As of the time of writing, a generator does not have a
// direct conversion to an iterator yet, so we need a wrapper:
use std::ops::{Generator, GeneratorState};
struct GeneratorIterator(T);
impl Iterator for GeneratorIterator
where
  T: Generator,
{
  type Item = T::Yield;
  fn next(&mut self) -> Option {
    match self.0.resume() {
      GeneratorState::Yielded(value) => Some(value),
      GeneratorState::Complete(_) => None,
    }
  }
}

fn fibonacci() -> impl Iterator {
  // Using our wrapper
  GeneratorIterator(move || {
    let mut curr = 0;
    let mut next = 1;
    loop {
      yield curr;
      let old = curr;
      curr = next;
      next += old;
    }
  })
}
```

How it works...

A `Generator` is currently defined as any closure that uses the new `yield` keyword. When it is executed with `.resume()` [11], it will run until it hits a `yield`. If run again, the generator will continue where it left off until it reaches another `yield` or encounters a `return`. If there are no more yields left in the generator, it will simply return an empty tuple, behaving as if encountering `return ();`.

Because there are two scenarios of what a generator does (`yield` vs `return`), you have to check the result of `.resume()` every time you use it, as it could be `GeneratorState::Yielded` or `GeneratorState::Complete`.

At the time of writing, you can `return` a different type than you `yield`. The situation around this is somewhat unclear, as the reason for this quirk is the aforementioned convention to `return ();` when running out of yields. Maybe the final version of generators in Rust will not rely on this behavior, and only allow returning the same type as yielding. You can find the discussion about this and more at `https://github.com/rust-lang/rust/issues/43122`.

Besides async, another big use case for generators is iterators. So much so that the Rust standard library iterators are planned to eventually be rewritten with generators. At the moment, the exact way of how this transition should happen has not been figured out, so there is no generic implementation of `Iterator for Generator`. To work around this, you can create a little wrapper type like we did with `GeneratorIterator`[50], which implements `Iterator` for its wrapped `Generator`.

We illustrate how to use it by rewriting the Fibonacci iterator from `Chapter 2`, *Working with Collections*; *Creating an own iterator,* with a generator in the `fibonacci` function [64]. The implementation looks pretty clean, doesn't it? As a reminder, here is how we wrote the original implementation using the `Iterator` trait directly, which not only needed a function, but also a `struct` and a trait implementation:

```
fn fibonacci() -> Fibonacci {
    Fibonacci { curr: 0, next: 1 }
}
struct Fibonacci {
    curr: u32,
    next: u32,
}
impl Iterator for Fibonacci {
    type Item = u32;
    fn next(&mut self) -> Option<u32> {
        let old = self.curr;
        self.curr = self.next;
        self.next += old;
        Some(old)
    }
}
```

See also

- *Creating an own iterator* recipe in `Chapter 2`, *Working with Collections*