Vesa Kaihlavirta

Mastering Rust

Write safe, concurrent and reliable programs without compromising on performance

Packt

Title Page

Mastering Rust
Write safe, concurrent and reliable programs without compromising on performance
Vesa Kaihlavirta

BIRMINGHAM - MUMBAI

Copyright

Mastering Rust

First published: May 2017

Production reference: 1290517

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78588-530-3

www.packtpub.com

Credits

Author Vesa Kaihlavirta	**Copy Editor** Safis Editing
Reviewer Antonin Carette	**Project Coordinator** Ulhas Kambali
Commissioning Editor Kunal Parikh	**Proofreader** Safis Editing
Acquisition Editor	**Indexer**

Denim Pinto	Aishwarya Gangawane
Content Development Editor Nikhil Borkar	**Graphics** Abhinash Sahu
Technical Editor Madhunikita Sunil Chindarkar	**Production Coordinator** Aparna Bhagat

About the Author

Vesa Kaihlavirta has been programming since he was five, beginning with C64 Basic. His main professional goal in life is to increase awareness of programming languages and software quality in all industries that use software. He's an Arch Linux Developer Fellow, and has been working in the telecom and financial industry for a decade. Vesa lives in Jyväskylä, central Finland.

I'd like to thank my brother, Lasse, for helping with the chapters of the book, and for raising my wisdom over the years. Also my wife, the love of my life and my greatest support, Johanna.

About the Reviewer

Antonin Carette is a PhD student at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) of Luxembourg, working on automated ways to classify documents using machine learning.

He believes in Mozilla philosophy and tries to contribute to the Mozilla community, promoting the Rust programming language.

Antonin lives in Thionville, France.

I'd like to thank my love, Alice, for supporting me (and my absent-mindedness) during these nights of reviews.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

Mapt

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Table of Contents

Preface

Rust is a new programming language. It offers performance and safety that reaches or even surpasses modern C++, while being a modern language with a relatively low barrier of entry. Rust's momentum, combined with its active and friendly community, promise a great future for the language.

While modern and fluent, Rust is not an entirely easy language. The memory management system keeps track of the life of every entity that is used in your program, and is designed in such a way that this tracking can typically happen entirely at compile time. The Rust programmer's burden is to help the compiler when it cannot decide for itself what should happen. Since modern programming is possible to do without ever facing such responsibilities, a modern programmer may not immediately feel comfortable with it.

However, like all expertise and skills, the more difficult it is to attain, the more valuable it is, and this book is here to help you. We cover the basics of Rust briefly, then move to more advanced parts such as the aforementioned memory management, concurrency, and metaprogramming. After working through this book, you'll have a very decent foundation for building highly performant and safe software.

What this book covers

Chapter 1, *Getting Your Feet Wet*, deals with installing the Rust toolset and runs through basic language features in a speedy fashion.

Chapter 2, *Using Cargo to Build Your First Program*, focuses on the standard build tool, Cargo, and also other development tools and their editor integration.

Chapter 3, *Unit Testing and Benchmarking*, covers the standard testing tools and practices.

Chapter 4, *Types*, runs through details and practices related to Rust's type system. We touch the different string types in Rust, arrays and slices, traits, implementations, and generics.

Chapter 5, *Error Handling*, covers how Rust handles error conditions in a rather unique way. Rust does error handling through its generic type system, instead of relying on exceptions.

Chapter 6, *Memory, Lifetimes, and Borrowing*, is possibly the most important chapter of the whole book. We see how Rust manages memory and resources, in general, in a safe way without relying on garbage collection.

Chapter 7, *Concurrency,* covers concurrent and parallel programming in Rust, and a few of the standard primitives (threads, channels, mutexes, and atomic reference counting) that can be used to implement safe concurrency.

Chapter 8, *Macros,* is where we start looking at the compile-time metaprogramming features of Rust. The so-called macros-by-example is the oldest and most stable form of metaprogramming in Rust.

Chapter 9, *Compiler Plugins*, goes through more advanced and newer metaprogramming features, such as linter plugins, custom derives, and code generation. Much of the content here relies on the nightly compiler.

Chapter 10, *Unsafety and Interfacing with Other Languages*, covers what kind of safety checks Rust has and how to circumvent them if needed. Interfacing with other languages is one place where we must instruct the compiler to relax some of its stricter checks.

Chapter 11, *Parsing and Serialization*, is where we look at a few ways of writing parsers. This chapter also touches on the standard Serde serialization framework.

Chapter 12, *Web Programming*, takes a look at basic backend web programming in Rust. We cover the low-level Hyper library for both client and server usage and check on the web framework situation by building a simple server-side game in Rocket.

Chapter 13, *Data Storage*, covers a few data storage options. We see how to build software with SQLite and PostgreSQL as data backends. We'll cover connection pooling via the r2d2 library, and lastly, we go through the Diesel ORM, followed by the summary of this chapter.

Chapter 14, *Debugging*, investigates using external debuggers for finding errors in Rust programs at runtime. We cover GDB and LLDB, and also GDB integration into the Visual Studio Code editor.

Chapter 15, *Solutions and Final Words*, contains short summaries for all the previous chapters for review purposes, followed by solutions to all the exercises in the book.

What you need for this book

To really dive into the content of this book, you should write out the example code and solve the exercises. For that, you'll need a fairly recent computer: 1 GB of RAM should be enough for the purposes of this book, but the more you have the faster the builds will be.

Linux is the best-supported operating system here, but Rust itself is also a first-class citizen on macOS and recent versions of Windows, so all the examples should adapt well there.

Who this book is for

The book will appeal to application developers who would like to build concurrent applications with Rust. Basic knowledge of Rust is assumed but not absolutely required.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "If the commands are not in your `PATH`, add the default Cargo installation location `$HOME/.cargo/bin/` to your path and try again."

A block of code is set as follows:

```
fn main() {
  println!("Are you writing this or reading it?");
}
```

Any command-line input or output is written as follows:

```
cargo install rustfmt
cargo install racer
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "To configure Rusty Code, select File | Preferences | Settings, and you'll get a two-pane view."

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Mastering-Rust. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Getting Your Feet Wet

Since you're already an accomplished programmer, this chapter will go through the design philosophy of Rust and the basics of the language in a rather speedy fashion. Each subsection will contain example code and runs of the compiler, the output given by it (if any), and there will be more than a dozen code examples.

Programming is a unique combination of knowledge and craft, both being equally important. To get on the path of mastering a craft, you need practice, which is why I recommend that you write, not copy/paste, every piece of code you see here manually.

Here are the topics covered in this chapter:

- Installing the Rust compiler and the Cargo build tool
- Language features: variables, conditionals, loops, primitive types, compound types, and sequences
- A final exercise for honing your skills with the compiler

What is Rust and why should you care?

Rust is a programming language originally started by Graydon Hoare in 2006. It's currently an open source project, developed mainly by a team in Mozilla and other developers. The first stable version, 1.0, was released in 2015.

While being a general purpose language, it is aiming for the space where C and C++ have dominated. Its defining principles that underline many of its design decisions are **zero-cost abstractions** and **compiler-assisted resource safety**.

One example of zero-cost abstractions is seen in Rust's iterators. They are an abstraction over loops that go through sequences, in roughly the same level that a markedly higher-level language such as Ruby has. However, their runtime cost is zero; they compile down to the same (or better) assembler code as you would have gotten by writing the same loop by hand.

Resource safety means that in Rust code and your resources (memory, file handles, and database references) can be analyzed by the compiler as safe to use. A most typical error in a C program is the memory-access error, where memory is used after being freed or is forgotten to be freed. In other languages, you might be spared from memory bugs by automatic garbage collection, but that may or may not help you with other types of resources such as file pointers. It gets even worse if you introduce concurrency and shared memory.

Rust has a system of borrows and lifetimes; plus, it replaces the concept of a null pointer with error types. These decisions raise the complexity of the language by a fair bit but make many errors impossible to make.

Last but not least, Rust's community is quite unusually active and friendly. Stack Overflow's Developer Survey in 2016 selected it as the most-loved programming language, so it can be said that the overall programming community is very interested in it.

To summarize, you should care about Rust because you can write high performing software with less bugs in it while enjoying many modern language features and an awesome community!

Installing Rust compiler and Cargo

The Rust toolset has two major components: the compiler (`rustc`) and a combined build tool or dependency manager (Cargo). This toolset comes in three frequently released versions:

- **Nightly**: This is the daily successful build of the master development branch. This contains all the features, some of which are unstable.
- **Beta**: This is released every six weeks; a new beta branch is taken from nightly. It contains only features that are flagged as stable.
- **Stable**: This is released every six weeks; the previous beta branch becomes the new stable.

Developers are encouraged to mainly use stable. However, the nightly version enables many useful features, which is why some libraries and programs require it.

Using rustup.rs

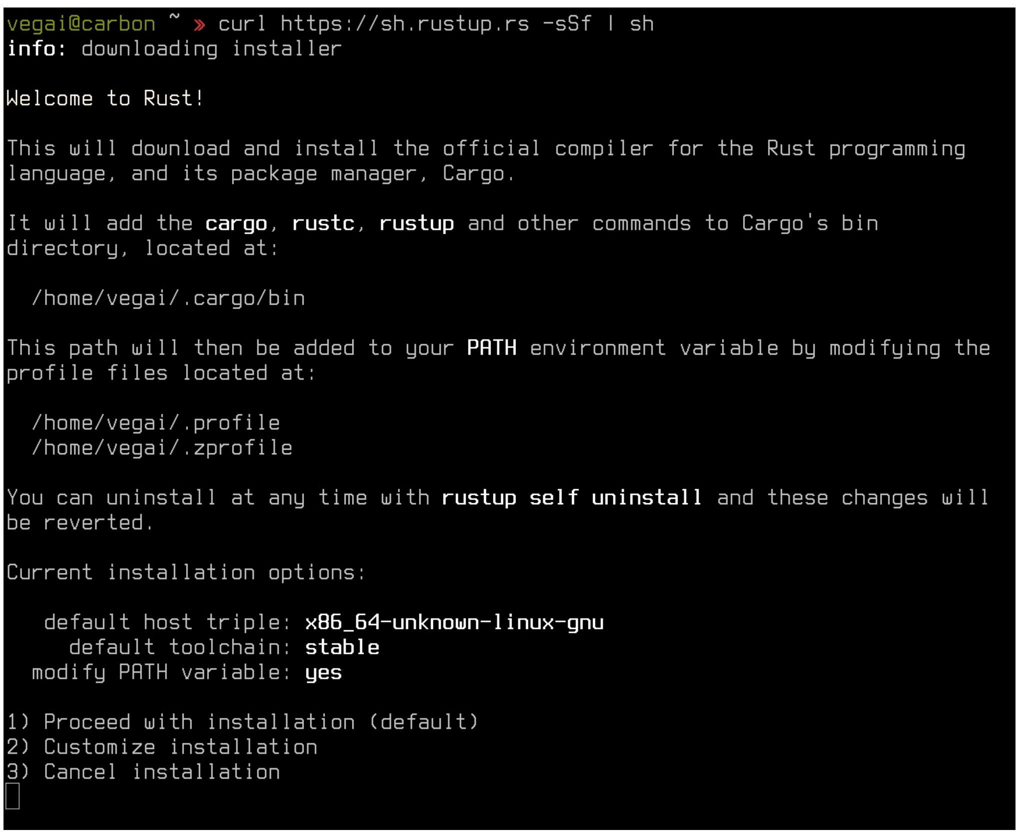
To make it easier for people in various platforms to download and install the standard tools, the Rust team developed rustup. The rustup tool provides a way to install prebuilt binaries of the Rust toolset (`rustc` and Cargo) easily for your local user. It also allows installing various other components, such as Rust source code and documentation.

The officially supported way to install Rust is to use `rustup.rs`:

```
curl https://sh.rustup.rs -sSf | sh
```

This command will download the installer and run it. The installer will, by default, install the stable version of the Rust compiler, the Cargo build tool, and the API documentation. They are installed by default for the current user under the `.cargo` directory, and rustup will also update your `PATH` environment variable to point there.

Here's how running the command should look:



If you need to make any changes to your installation, choose *2*. But these defaults are fine for us, so we'll go ahead and choose *1*. This is what the output should look like afterwards:

```
1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
1

info: syncing channel updates for 'stable-x86_64-unknown-linux-gnu'
info: downloading component 'rustc'
 37.2 MiB /  37.2 MiB (100 %)   2.2 MiB/s ETA:   0 s
info: downloading component 'rust-std'
 61.6 MiB /  61.6 MiB (100 %)   2.0 MiB/s ETA:   0 s
info: downloading component 'cargo'
  4.8 MiB /   4.8 MiB (100 %)   1.6 MiB/s ETA:   0 s
info: downloading component 'rust-docs'
 10.1 MiB /  10.1 MiB (100 %)   1.9 MiB/s ETA:   0 s
info: installing component 'rustc'
info: installing component 'rust-std'
info: installing component 'cargo'
info: installing component 'rust-docs'
info: default toolchain set to 'stable'

  stable installed - rustc 1.17.0 (56124baa9 2017-04-24)


Rust is installed now. Great!

To get started you need Cargo's bin directory in your PATH environment
variable. Next time you log in this will be done automatically.

To configure your current shell run source $HOME/.cargo/env
vegai@carbon ~ »
```

Now, you should have everything you need to compile and run programs written in Rust. Let's try it!

A tour of the language and trying it out

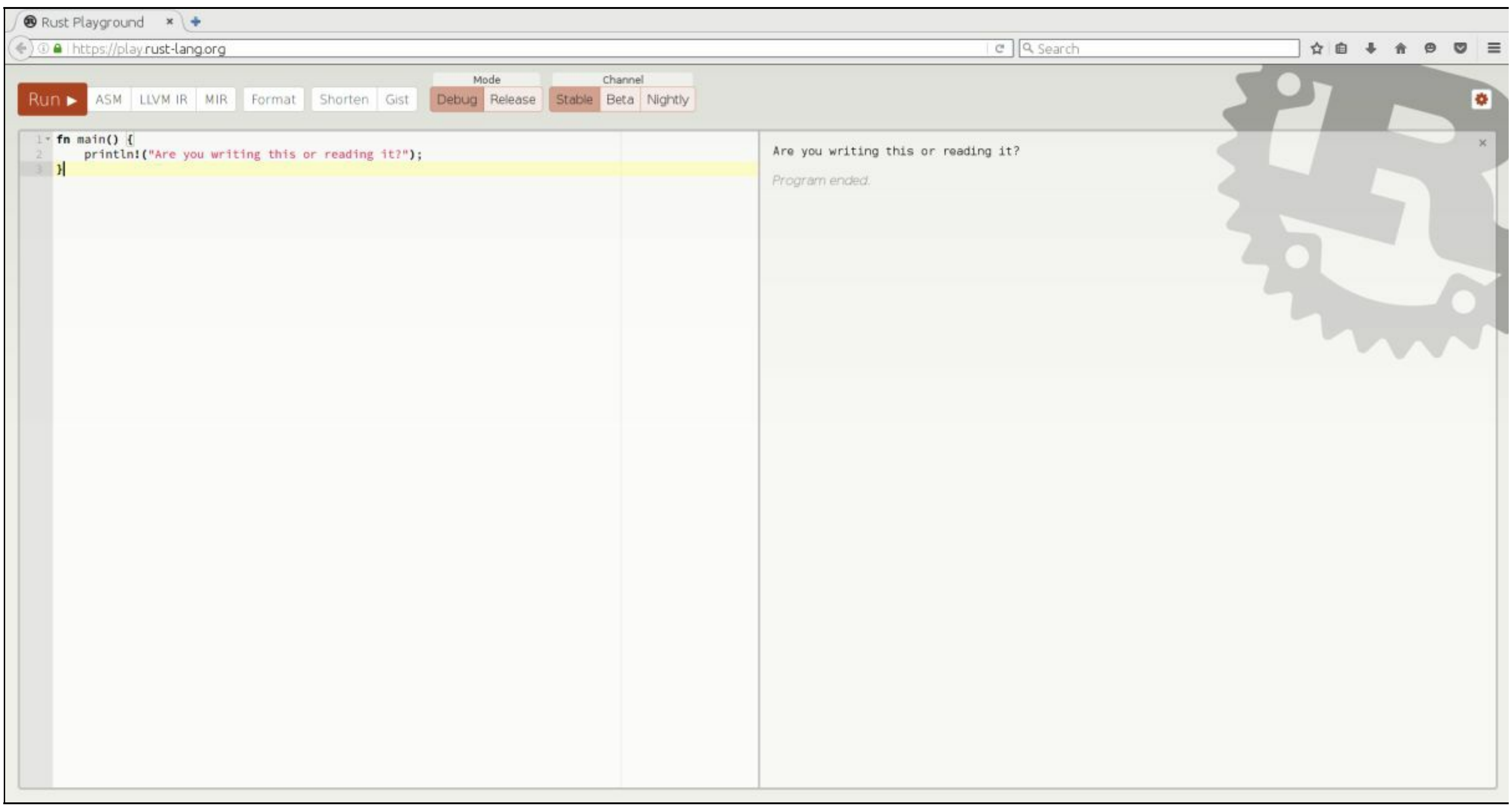
For the fundamental language features, Rust does not stray far from what you are used to. Programs are defined in modules; they contain functions, variables, and compound data structures. Here's how a minimal program looks:

```
fn main() {
  println!("Are you writing this or reading it?");
}
```

Try compiling and running this. Write it to a file called `main.rs` and then run the Rust compiler:

```
> rustc -o main main.rs
> ./main
Are you writing this or reading it?
```

Running `rustc` manually is not how you will do it for real programs, but it will do for these small programs. A fine alternative to running small pieces of code is to use the Rust Playground service in http://play.rust-lang.org:



The program itself is fairly simple: the `fn` keyword is used to define functions, followed by the function name, its arguments inside parentheses, and the function body inside curly braces. Nothing new (except some syntax) there. The exclamation mark after the print-line call means that it's actually not a function, but a macro. This just means that it performs some expansions at compile time rather than doing all the work at runtime. If you are familiar with macros from other languages such as C or LISP, Rust macros will be familiar as well. Macros will be covered more in Chapter 9, *Compiler Plugins*.

Variables are defined with the `let` keyword. Rust has a local type inference, which means that the types of function variables are figured out by the compiler, and the coder can almost always omit them. It can easily lead to improved readability of the source code, especially in the case of frequently used static strings:

```
// first-program.rs
fn main() {
    let target_inferred = "inferred world";
    // these two variables
    let target: &'static str = "non-inferred world"; // have identical types
    println!("Hi there, {}", target_inferred);
    println!("Hi there, {}", target);
}
```

The strings in this program are string literals or, more specifically, string slices with a static lifetime. Strings will be covered in Chapter 4, *Types*, and lifetimes in Chapter 6, *Memory, Lifetimes, and Borrowing*.

Comments in code are written like in C, `//` for single line comments, and `/* */` blocks for multiline comments.

Constants and variables

Rust deviates from the mainstream here by making constants the default variable type. If you need a variable that can be mutated, you use the `let mut` keyword:

```
// variables.rs
fn main() {
  let mut target = "world";
  println!("Howdy, {}", target);
  target = "mate";
  println!("Howdy, {}", target);
}
```

Conditionals should also look familiar; they follow the C-like `if...else` pattern. Since Rust is strongly-typed, the condition must be a Boolean type:

```
// conditionals.rs
fn main() {
  let condition = true;
  if condition {
    println!("Condition was true");
  } else {
    println!("Condition was false");
  }
}
```

In Rust, `if` is not a statement but an expression. This distinction means that `if` always returns a value. The value may be an empty type that you don't have to use, or it may be an actual value. This means that you can use the `if` expression as tertiary expressions are used in some languages:

```
// if-expression.rs
fn main() {
  let result = if 1 == 2 {
    "Nothing makes sense"
  } else {
    "Sanity reigns"
  };

  println!("Result of computation: {}", result);
}
```

Take a closer look at the preceding program; it highlights an important detail regarding the semicolon and blocks. The semicolon is not optional in Rust, but it has a specific meaning. The last expression of a block is the one whose value is returned out of a block, and the absence of the semicolon in the last line is important; if we were to add a semicolon after the strings in the `if` blocks, Rust would interpret it as you wanting to throw the value away:

```
// semicolon.rs
fn main() {
  let result = if 1 == 2 {
    "Nothing makes sense";
  } else {
    "Sanity reigns";
  };

  println!("Result of computation: {:?}", result);
}
```

In this case, the result will be empty, which is why we had to change the `println!` expression slightly; this type cannot be printed out in the regular way. More about that in Chapter 4, *Types*, where we talk about types.

Loops

Simple loops are programmed with either the `while` loop (if a condition for the looping is wanted) or with `loop` (if no condition is wanted). The `break` keyword gets you out of the loop. Here's an example of using the `loop` keyword:

```
// loop.rs
fn main() {
  let mut x = 1000;
  loop {
        if x < 0 {
                 break;
            }
            println!("{} more runs to go", x);
            x -= 1;
    }
}
```

An example of `while` loop is as follows:

```
// while.rs
fn main() {
  let mut x = 1000;
  while x > 0 {
            println!("{} more runs to go", x);
            x -= 1;
  }
}
```

Compound data

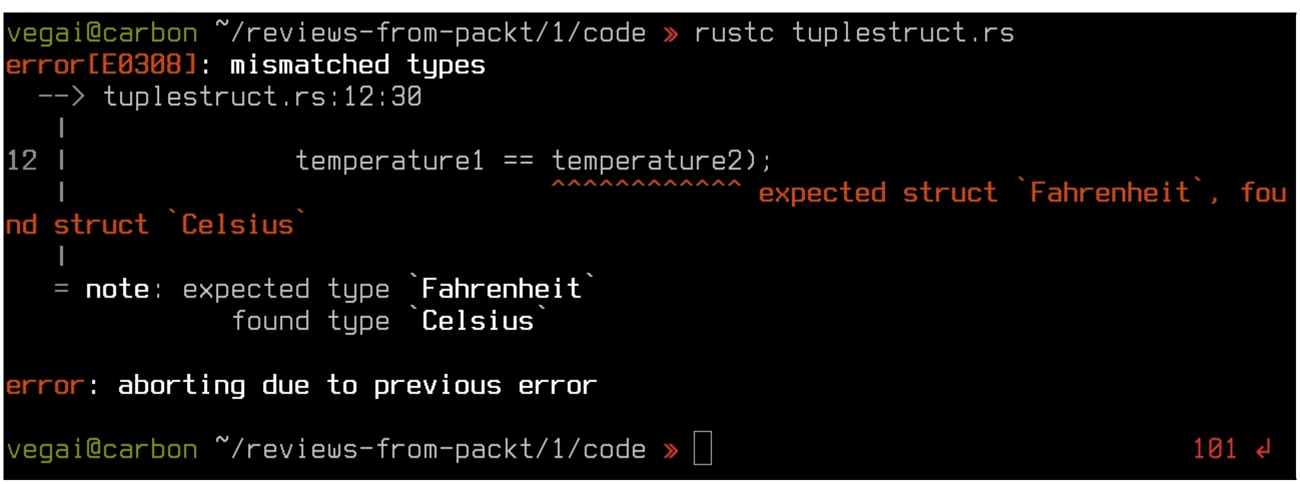
For defining custom data types, there are **structs**. The simpler form is called a **tuple struct**, where the individual fields are not named but are referred to by their position. This should mostly be used when your data consists of only one or a few fields to achieve better levels of type safety, such as here:

```
// tuplestruct.rs
#[derive(PartialEq)]
struct Fahrenheit(i64);

#[derive(PartialEq)]
struct Celsius(i64);

fn main() {
 let temperature1 = Fahrenheit(10);
 let temperature2 = Celsius(10);

 println!("Is temperature 1 the same as temperature 2? Answer: {}",
           temperature1 == temperature2);

 println!("Temperature 1 is {} fahrenheit", temperature1.0);
 println!("Temperature 2 is {} celsius", temperature2.0);
}
```

What is inside the tuple struct can be accessed by the `.<number>` operation, where the number refers to the position of the field in the struct.

This is the first piece of code in this book that fails to compile, and the reason is that while the two temperatures get the equals methods derived for them, they will only be defined for comparing the same types. Since comparing Fahrenheit with Celsius without any sort of conversion does not make sense, you can fix this piece of code by either removing the last `println!` invocation or by comparing `temperature1` against itself. The derive line before the structs generated code that allows `==` operation to work against the same type.

Here's how the compiler tells you this:



The other form of structs has named fields:

```
// struct.rs
```

```
struct Character {
  strength: u8,
  dexterity: u8,
  constitution: u8,
  wisdom: u8,
  intelligence: u8,
  charisma: u8,
  name: String
}

fn main() {
   let char = Character { strength: 9, dexterity: 9, constitution: 9,
   wisdom: 9, intelligence: 9, charisma: 9,
   name: "Generic AD&D Hero".to_string() };

   println!("Character's name is {}, and his/her strength is {}", char.name, char.strength);
}
```

In the preceding struct, you can see the usage of a primitive type, the unsigned 8-bit integer (**u8**). Primitive types by convention start with a lowercase character, whereas other types start with a capital letter (such as String up there). For reference, here's a full table of all primitive types:

Type	**Description**	**Possible values**
`bool`	Booleans	true, false
`u8/u16/u32/u64`	Fixed size unsigned integers	Unsigned range determined by bit size
`i8/i16/i32/i64`	Fixed size signed integers	Signed range determined by bit size
`f32/f64`	Fixed size floats	Float range determined by bit size (IEEE-754)
`usize`	Architecture-dependant unsigned integer	Depending on target machine, usually 32 or 64 bit value
`isize`	Architecture-dependant signed integer	Depending on target machine, usually 32 or 64 bit value
`char`	Single unicode character	4 bytes describing a unicode character
`str`	String slice	Unicode string

`[T; N]`	Fixed-size arrays	N number of type `T` values
`&[T]`	Slices	References to values of type `T`
`(T1, T2, ...)`	Tuples	Elements of types `T1`, `T2`, `...`
`fn(T1, T2, ...) → R`	Functions	Functions that take types `T1`, `T2`, `...` as parameters, returns value of type `R`

Enums and pattern matching

Whenever you need to model something that can be of several different types, `enums` may be a good choice. The enum variants in Rust can be defined with or without data inside them, and the data fields can be either named or anonymous:

```
enum Direction {
  N,
  NE,
  E,
  SE,
  S,
  SW,
  W,
  NW
}

enum PlayerAction {
  Move(direction: Direction, speed: u8),
  Wait,
  Attack(Direction)
}
```

This defines two `enum` types: `Direction` and `PlayerAction`. For each of these `enum` types, this also defines a number of namespaced enum variants: `Direction::N`, `Direction::NE`, and so on for the `Direction` type, and `PlayerAction::Move`, `PlayerAction::Wait`, and `PlayerAction::Attack` for the `PlayerAction` type.

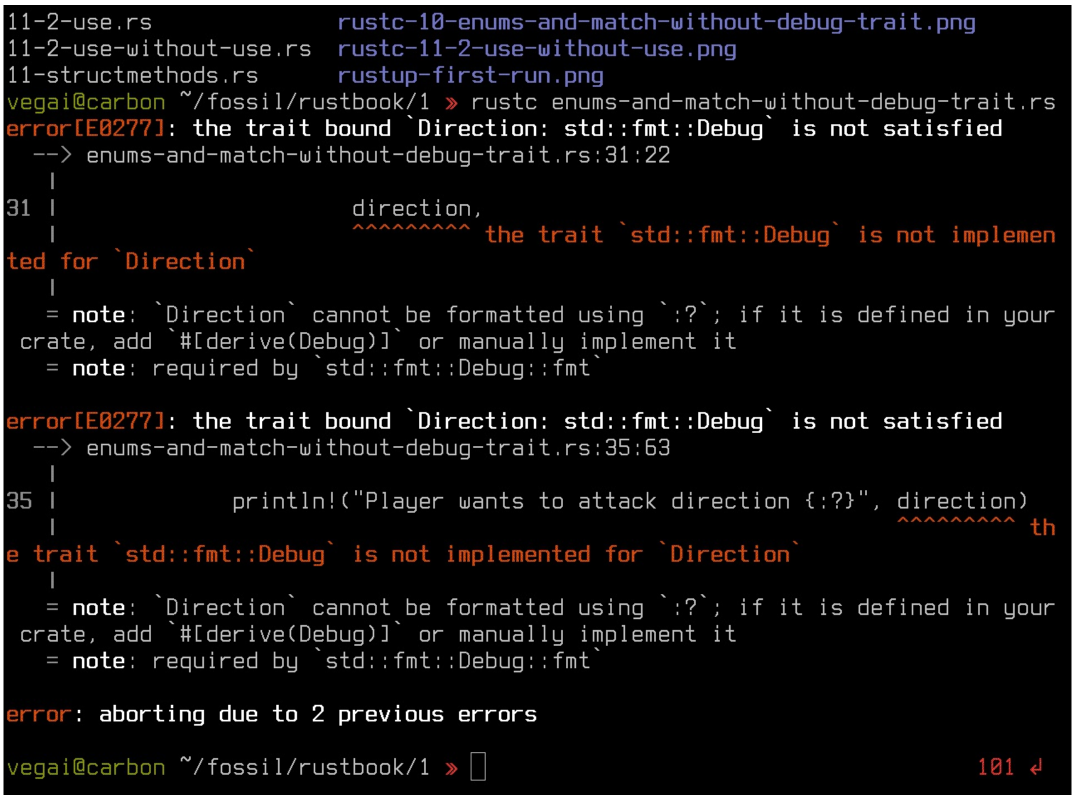
The most typical way of working with `enums` is pattern matching with the `match` expression:

```
#[derive(Debug)]
enum Direction {
  N,
  NE,
  E,
  SE,
  S,
  SW,
  W,
  NW,
}

enum PlayerAction {
  Move {
    direction: Direction,
    speed: u8,
  },
  Wait,
  Attack(Direction),
}

fn main() {
  let simulated_player_action = PlayerAction::Move {
    direction: Direction::NE,
    speed: 2,
  };

  match simulated_player_action {
    PlayerAction::Wait => println!("Player wants to wait"),
    PlayerAction::Move { direction, speed } => {
      println!("Player wants to move in direction {:?} with speed {}",
               direction, speed)
    }
    PlayerAction::Attack(direction) => {
      println!("Player wants to attack direction {:?}", direction)
```

```
        }
    };
}
```

Like `if`, `match` is also an expression, which means that it returns a value, and that value has to be of the same type in every branch. In the preceding example, it's what `println!()` returns, that is, the empty type.

The derive line above the first `enum` tells the compiler to generate code for a `Debug` trait. Traits will be covered more in Chapter 4, *Types*, but for now, we can just note that it makes the `println!` macro's `{:?}` syntax work properly. The compiler tells us if the `Debug` trait is missing and gives suggestions about how to fix it:



Struct methods

It's often the case that you wish to write functions that operate on a specific struct or return the values of a specific struct. That's when you write implementation blocks with the `impl` keyword.

For instance, we could extend the previously defined character struct with two methods: a constructor that takes a name and sets default values for all the character attributes and a getter method for character strength:

```
// structmethods.rs
struct Character {
  strength: u8,
  dexterity: u8,
  constitution: u8,
  wisdom: u8,
  intelligence: u8,
  charisma: u8,
  name: String,
}

impl Character {
  fn new_named(name: String) -> Character {
    Character {
      strength: 9,
      constitution: 9,
      dexterity: 9,
      wisdom: 9,
      intelligence: 9,
      charisma: 9,
      name: name,
    }
  }

  fn get_strength(&self) -> u8 {
    self.strength
  }
}
```

The `new_named` method is called an associated function because it does not take `self` as the first parameter. It is not far from what many other languages would call a static method. It is also a constructor method since it follows the convention of starting with the word, `new`, and because it returns a struct of the same type (`Character`) for which we're defining an implementation. Since `new_named` is an associated function, it can be called by prefixing the struct name and double colon:

```
Character::new_named("Dave")
```

The `self` parameter in `get_strength` is special in that its type is inferred to be the same as the `impl` block's type, and because it is the thing that makes `get_strength` a callable method on the struct. In other words, `get_strength` can be called on an already created instance of the struct:

```
let character = Character::new_named("Dave");
character.get_strength();
```

The ampersand before `self` means that `self` is borrowed for the duration of the method, which is exactly what we want here. Without the ampersand, the ownership would be moved to the method, which means that the value would be deallocated after leaving `get_strength`. Ownerships are a distinguishing feature of Rust, and will be dealt in depth in Chapter 6, *Memory, Lifetimes, and Borrowing*.

Using other pieces of code in your module

A quick word about how to include code from other places into the module you are writing. Rust's module system has its own pecularities, but it's enough to note now that the `use` statement brings code from another module into the current namespace. It does not load external pieces of code, it merely changes the visibility of things:

```
// use.rs
use std::ascii::AsciiExt;

fn main() {
  let lower_case_a = 'a';
  let upper_case_a = lower_case_a.to_ascii_uppercase();

  println!("{} upper cased is {}", lower_case_a, upper_case_a);
}
```

In this example, the `AsciiExt` module contains an implementation of the `to_ascii_uppercase` for the char type, so including that in this module makes it possible to use the method here. The compiler manages again to be quite helpful if you miss a particular `use` statement, like what happens here if we remove the first line and try to compile:

```
vegai@carbon ~/rustbook/1 » rustc use-without-use.rs
error: no method named `to_ascii_uppercase` found for type `char` in the curren
t scope
 --> use-without-use.rs:3:37
  |
3 |     let upper_case_a = lower_case_a.to_ascii_uppercase();
  |                                     ^^^^^^^^^^^^^^^^^^
  |
  = help: items from traits can only be used if the trait is in scope; the foll
owing trait is implemented but not in scope, perhaps add a `use` for it:
  = help: candidate #1: `use std::ascii::AsciiExt;`

error: aborting due to previous error

vegai@carbon ~/rustbook/1 »                                             101 ↲
```

Sequences

One more thing to cover and then we can wrap up the basics. Rust has a few built-in ways to construct sequences of data: `arrays` and `tuples`. Then, it has a way to take a view to a piece of that data: `slices`. Thirdly, it has several data structures as libraries, of which we will cover **Vectors** (for dynamically growable sequences) and **HashMaps** (for key/value data).

Arrays are C-like: they have a fixed length that you need to specify along with the type of the elements of the array when declaring it. The notation for array types is `[<type>, <size>]`:

```
// arrays.rs
fn main() {
  let numbers: [u8; 10] = [1, 2, 3, 4, 5, 7, 8, 9, 10, 11];
  let floats = [0.1, 0.2, 0.3];

  println!("The first number is {}", numbers[0]);

  for number in &numbers {
    println!("Number is {}", number);
  }

  for float_number in &floats {
    println!("Float is {}", float_number);
  }
}
```

As said before, Rust is able to infer the types of local variables, so writing them out is optional.

Slices offer a way to safely point to a continuous range in an existing data structure. The type of slices is `&[T]`. Its syntax looks similar to arrays:

```
// slices.rs
fn main() {
  let numbers: [u8; 4] = [1, 2, 4, 5];
  let all_numbers_slice: &[u8] = &numbers[..];
  let first_two_numbers: &[u8] = &numbers[0..2];

  println!("All numbers: {:?}", all_numbers_slice);
  println!("The second of the first two numbers: {}", first_two_numbers[1]);
}
```

Tuples differ from arrays in the way that arrays are sequences of the same type, while tuple elements have varying types:

```
// tuples.rs
fn main() {
  let number_and_string: (u8, &str) = (40, "a static string");

  println!("Number and string in a tuple: {:?}", number_and_string);
}
```

They are useful for simple, type-safe compounding of data, generally used when returning multiple values from a function.

Vectors are like arrays except that their contents or length don't have to be known in advance. They are created with either calling the constructor `Vec::new` or by using the `vec!` macro:

```
// vec.rs
fn main() {
  let mut numbers_vec: Vec<u8> = Vec::new();
  numbers_vec.push(1);
  numbers_vec.push(2);

  let mut numbers_vec_with_macro = vec![1];
  numbers_vec_with_macro.push(2);

  println!("Both vectors have equal contents: {}", numbers_vec ==
            numbers_vec_with_macro);
}
```

These are not the only ways to create vectors, and one typical way needs to be covered here. Rust defines **iterators**, things that can be iterated one by one, in a generic way. For instance, a program's runtime arguments are iterators, which would be a problem if you wanted to get the n^{th} argument. However, every iterator has a `collect` method, which gathers all the items in the iterator into a single collection, such as a vector, which *can* be indexed. There's an example of this usage in the chapter's exercise.

Finally, **HashMaps** can be used for key/value data. They are created with the `HashMap::new` constructor:

```
// hashmap.rs
use std::collections::HashMap;

fn main() {
  let mut configuration = HashMap::new();
  configuration.insert("path", "/home/user/".to_string());

  println!("Configured path is {:?}", configuration.get("path"));
}
```

Exercise - fix the word counter

Here's a program that counts instances of words in a text file, given to it as its first parameter. It is almost complete but has a few bugs that the compiler catches, and a couple of subtle ones. Go ahead and type the program text into a file, try to compile it, and try to fix all the bugs with the help of the compiler. The point of this exercise, in addition to covering the topics of this chapter, is to make you more comfortable with the error messages of the Rust compiler, which is an important skill for an aspiring Rust developer.

Try to make an effort even when things seem hopeless; *every drop of tear and sweat brings you a step closer to being a master*. Detailed answers to the task can be found in the Appendix section. Good luck!

```
// wordcounter.rs
use std::env;
use std::fs::File;
use std::io::prelude::BufRead;
use std::io::BufReader;

#[derive(Debug)]
struct WordStore (HashMap<String, u64>);

impl WordStore {
    fn new() {
        WordStore (HashMap::new())
    }

    fn increment(word: &str) {
        let key = word.to_string();
        let count = self.0.entry(key).or_insert(0);
        *count += 1;
    }

    fn display(self) {
        for (key, value) in self.0.iter() {
            println!("{}: {}", key, value);
        }
    }
}

fn main() {
    let arguments: Vec<String> = env::args().collect();
    println!("args 1 {}", arguments[1]);
    let filename = arguments[1].clone();

    let file = File::open(filename).expect("Could not open file");
    let reader = BufReader::new(file);

    let word_store = WordStore::new();

    for line in reader.lines() {
        let line = line.expect("Could not read line");
        let words = line.split(" ");
        for word in words {
            if word == "" {
                continue
            } else {
                word_store.increment(word);
            }
        }
    }

    word_store.display();
```

```
}
```

If you like extra challenges, here are a few ideas for you to try to flex your muscles a bit further:

1. Add a parameter to WordStore's `display` method for filtering the output based on the count. In other words, display a key/value pair only if the value is greater than that filtering value.
2. Since HashMaps store their values randomly, the output is also quite random. Try to sort the output. The HashMap's values method may be useful.
3. Think about the display method's `self` parameter. What is the implication of not using the ampersand (`&`) before `self`?

Summary

You now know the design principles and the basic language features of Rust, how to install the default implementation, and how to use the Rust compiler to build your own single-file pieces of code.

In the next chapter, we will take a look at editor integrations and the Cargo tool, and build the foundation for the project that we will extend during the course of the book.

Using Cargo to Build Your First Program

Now that we have some Rust under our belts, we can start with our project. Before we can do that, however, we will take a deeper look at the Cargo program and how it is used to declare project metadata and dependencies and build Rust projects.

This chapter will cover the following topics:

- The Cargo build tool
- `cargo init`: starting a project
- `cargo build`: building a project
- `cargo test`: running tests and benchmarks
- `cargo search`: searching crates: third-party libraries
- How to write `Cargo.toml` to configure your project
- Editor integrations

As a final exercise, we'll start the project that will be a basis for the rest of the book.

Cargo and crates

To write a larger program, you need some way of declaring what your program is about, how it should be built, and what its dependencies are. You will also need a way to act on those declarations. Furthermore, to support the whole programming ecosystem, you need a centralized place to find those dependencies.

For Rust, Cargo is the tool for doing all these things, and https://crates.io/ is the centralized place. If you ran `rustup` as it was described in the previous chapter, you have `cargo` installed along with `rustc`.

To see help on `cargo`, run it without parameters:

```
vegai@carbon ~ » cargo help
Rust's package manager

Usage:
    cargo <command> [<args>...]
    cargo [options]

Options:
    -h, --help              Display this message
    -V, --version           Print version info and exit
    --list                  List installed commands
    --explain CODE          Run `rustc --explain CODE`
    -v, --verbose ...       Use verbose output (-vv very verbose/build.rs output)
    -q, --quiet             No output printed to stdout
    --color WHEN            Coloring: auto, always, never
    --frozen                Require Cargo.lock and cache are up to date
    --locked                Require Cargo.lock is up to date

Some common cargo commands are (see all commands with --list):
    build       Compile the current project
    check       Analyze the current project and report errors, but don't build o
bject files
    clean       Remove the target directory
    doc         Build this project's and its dependencies' documentation
    new         Create a new cargo project
    init        Create a new cargo project in an existing directory
    run         Build and execute src/main.rs
    test        Run the tests
    bench       Run the benchmarks
    update      Update dependencies listed in Cargo.lock
    search      Search registry for crates
    publish     Package and upload this project to the registry
    install     Install a Rust binary

See 'cargo help <command>' for more information on a specific command.
vegai@carbon ~ »
```

Founding a project - cargo init

The `cargo init` command creates a new project structure: a `Cargo.toml` file with the essential metadata prefilled and a skeleton `src/main.rs` (for binary projects) or `src/lib.rs` (for library projects):

```
vegai@carbon ~ » cargo help init
Create a new cargo package in current directory

Usage:
    cargo init [options] [<path>]
    cargo init -h | --help

Options:
    -h, --help          Print this message
    --vcs VCS           Initialize a new repository for the given version
                        control system (git or hg) or do not initialize any vers
ion
                        control at all (none) overriding a global configuration.
    --bin               Use a binary (application) template
    --lib               Use a library template
    --name NAME         Set the resulting package name
    -v, --verbose ...   Use verbose output (-vv very verbose/build.rs output)
    -q, --quiet         No output printed to stdout
    --color WHEN        Coloring: auto, always, never
    --frozen            Require Cargo.lock and cache are up to date
    --locked            Require Cargo.lock is up to date
vegai@carbon ~ »
```

By default, `cargo init` creates a new library; the `--bin` parameter has to be used when creating a project that we want to run. Try it out and take a look at the directory structure it creates:

```
vegai@carbon ~ » cargo init --bin a_project
     Created binary (application) project
vegai@carbon ~ » tree a_project
a_project
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
vegai@carbon ~ »
```

Cargo created for you a Git repository and the files, `Cargo.toml` and `src/main.rs`. Let's take a look at `Cargo.toml`; this is the file that defines your project's metadata and dependencies:

```
[package]
name = "project"
version = "0.1.0"
authors = ["vegai <vegai@iki.fi>"]

[dependencies]
```

This is the minimal `Cargo.toml` file needed for a new project. **TOML** is short for **Tom's Obvious, Minimal Language**, a file format created by Tom Preston-Werner. It is reminiscent of standard INI files but adds several data types to it, which makes it an ideal modern format for configuration files. Let's keep it minimal for now; we will add things to it later.

Dependencies, building, and running

Before we can cover building and running, let's discuss a bit about dependency versioning. Cargo has two files that cover dependency versions: `Cargo.toml` is the file where you, as the coder, write dependencies and their wanted versions and `Cargo.lock` is a generated file that contains fixed versions of the said dependencies.

Depending on the stability requirements of your project, you might want your dependencies to be deterministic, never to change without you specifically requesting for it. With Cargo, you can define in rather broad strokes what version of dependencies you wish to include and then lock the dependency to a specific changeset or version.

For example, you might want to include the serialization library, **Serde**, in your project. At the time of writing this book, the latest version of Serde is 1.0, but you probably don't need to be fixed on that minor version. So, you define 1 as the version in your `Cargo.toml`, and `Cargo.lock` will fix it to 1. The next time you update `Cargo.lock` with the `cargo update` command, this version might get upgraded to 1.0.1 or whichever is the latest version in the 1.0.* match. If you don't care so much and just want the latest released version, you can use * as the version.

With that in mind, we can take a look at the `cargo build` command. It does the following:

- Runs `cargo update` for you if you don't yet have a `Cargo.lock` file
- Downloads all your dependencies defined in `Cargo.lock`
- Builds all those dependencies
- Builds your project and links it with the dependencies

Here's the help documentation for `cargo build`:

```
vegai@carbon ~ » cargo help build
Compile a local package and all of its dependencies

Usage:
    cargo build [options]

Options:
    -h, --help                   Print this message
    -p SPEC, --package SPEC ...  Package to build
    --all                        Build all packages in the workspace
    -j N, --jobs N               Number of parallel jobs, defaults to # of CPUs
    --lib                        Build only this package's library
    --bin NAME                   Build only the specified binary
    --example NAME               Build only the specified example
    --test NAME                  Build only the specified test target
    --bench NAME                 Build only the specified benchmark target
    --release                    Build artifacts in release mode, with optimizat
ions
    --features FEATURES          Space-separated list of features to also build
    --all-features               Build all available features
    --no-default-features        Do not build the `default` feature
    --target TRIPLE              Build for the target triple
    --manifest-path PATH         Path to the manifest to compile
    -v, --verbose ...            Use verbose output (-vv very verbose/build.rs o
utput)
    -q, --quiet                  No output printed to stdout
    --color WHEN                 Coloring: auto, always, never
    --message-format FMT         Error format: human, json [default: human]
    --frozen                     Require Cargo.lock and cache are up to date
    --locked                     Require Cargo.lock is up to date

If the --package argument is given, then SPEC is a package id specification
which indicates which package should be built. If it is not given, then the
current package is built. For more information on SPEC and its format, see the
`cargo help pkgid` command.

All packages in the workspace are built if the `--all` flag is supplied. The
`--all` flag may be supplied in the presence of a virtual manifest.

Compilation can be configured via the use of profiles which are configured in
the manifest. The default profile for this command is `dev`, but passing
the --release flag will use the `release` profile instead.
vegai@carbon ~ »
```

The default mode of `cargo build` is to build a debug version of the project without much optimization. The `-release` switch creates a production build, properly optimized. The difference in runtime speeds can be quite significant, but the optimized build is slower to compile. The resulting binary goes into target/debug or target/release, depending on your choice. Since that's tedious to remember or type out every time, the `cargo run` command builds and runs the binary for you.

Running tests - cargo test

Unit testing is one of the rare silver bullets of our industry that can make all the difference in keeping up high software quality. Rust supports unit testing and benchmark testing natively. Let's build a small library project to see how testing works from Cargo's side:

```
vegai@carbon ~/rustbook/2 » cargo init library-example
     Created library project
vegai@carbon ~/rustbook/2 » cd library-example
vegai@carbon ~/rustbook/2/library-example ±master⚡ » cat Cargo.toml
[package]
name = "library-example"
version = "0.1.0"
authors = ["vegai"]

[dependencies]
vegai@carbon ~/rustbook/2/library-example ±master⚡ » cat src/lib.rs
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
    }
}
vegai@carbon ~/rustbook/2/library-example ±master⚡ »
```

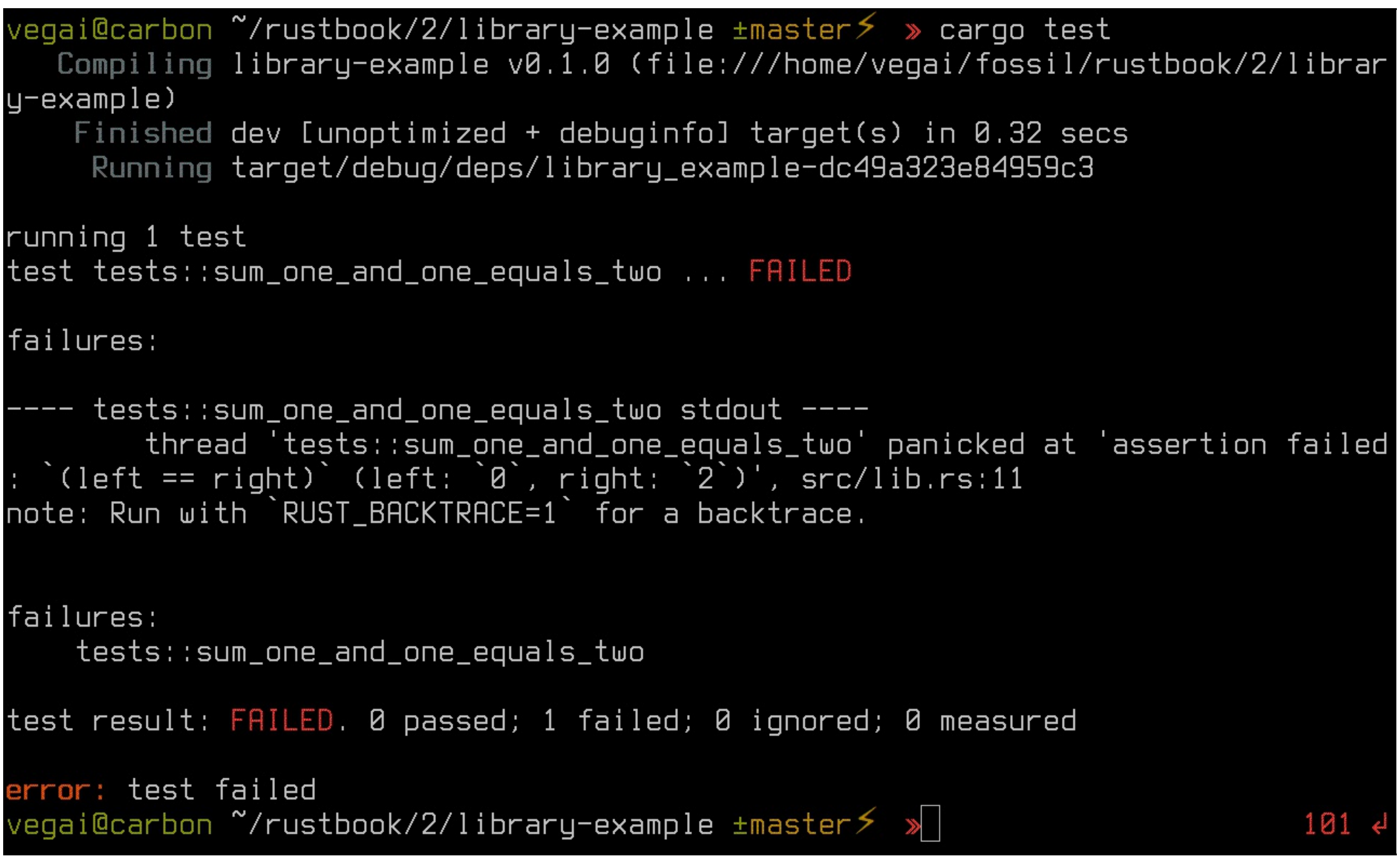
As you see, a library project is very similar to a binary project. The difference is that instead of `main.rs` and a main function inside it as an entry point, there is `lib.rs` with functions. Since Cargo has already created for us a skeleton `lib.rs` with a dummy unit test, we can run the tests right away:

```
vegai@carbon ~/rustbook/2/library-example ±master⚡ » cargo test
   Compiling library-example v0.1.0 (file:///home/vegai/fossil/rustbook/2/librar
y-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.47 secs
     Running target/debug/deps/library_example-dc49a323e84959c3

running 1 test
test tests::it_works ... ok

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

   Doc-tests library-example

running 0 tests

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

vegai@carbon ~/rustbook/2/library-example ±master⚡ »
```

Let's try a bit of **test-driven development** (**TDD**). We'll write a test for a functionality that we expect to fail and then fill in the implementation until it works. Here's the new `src/lib.rs`, featuring a `sum` function without proper implementation:

```
// library-example-1/src/lib.rs
#[allow(unused_variables)]
fn sum(a: i8, b: i8) -> i8 {
    return 0;
}

#[cfg(test)]
mod tests {
    use super::sum;
    #[test]
    fn sum_one_and_one_equals_two() {
        assert_eq!(sum(1, 1), 2);
    }
}
```

Don't worry about the details right now. There's a single `sum` function, and under the tests namespace, we have a single test function for it, which does a single assertion. An assertion checks that some condition is satisfied; in this case, the equality of two things. If the condition passes, the assertion passes; otherwise, it is an error and running the program stops. The unit tests fail due to an obvious flaw in `sum`:



Fix the problem in the `sum` function and try again:

```
fn sum(a: i8, b: i8) -> i8 {
    return a + b;
}
```

```
vegai@carbon ~/rustbook/2/library-example ±master⚡ » cargo test
   Compiling library-example v0.1.0 (file:///home/vegai/fossil/rustbook/2/librar
y-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
     Running target/debug/deps/library_example-dc49a323e84959c3

running 1 test
test tests::sum_one_and_one_equals_two ... ok

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

   Doc-tests library-example

running 0 tests

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

vegai@carbon ~/rustbook/2/library-example ±master⚡ »
```

That's good enough for now. We'll dive deeper into unit testing in the next chapter.

Cargo.toml - project metadata

Let's take a closer look at what `Cargo.toml` may contain. As you saw earlier, `cargo init` creates an almost empty `Cargo.toml` filled with just the most necessary fields so that a project can be built.

To see what `Cargo.toml` supports, here's an imagined `Cargo.toml` from a larger application:

```
# cargo-example/Cargo.toml
[package]
name = "cargo-metadata-example"
version = "1.2.3"
description = "An example of Cargo metadata"
license = "MIT"
readme = "README.md"
keywords = ["example", "cargo", "mastering"]
authors = ["Jack Daniels <jack@danie.ls>", "Iddie Ezzard <iddie@ezzy>"]
build = "build.rs"

[package.metadata.settings]
default-data-path = "/var/lib/example"

[features]
default=["mysql"]

[build-dependencies]
syntex = "^0.58"

[dependencies]
serde = "1.0"
serde_json = "1.0"
time = { git = "https://github.com/rust-lang/time", branch = "master" }
mysql = { version = "1.2", optional = "true" }
sqlite = { version = "2.5", optional = "true" }
```

Let's go through the parts that we haven't seen yet, starting from the `[package]` section:

- `description`: It contains a longer, free-form text field about the project.
- `license`: It supports software license identifiers listed in http://spdx.org/licenses/.
- `readme`: It allows you to link to a file in your project's repository that should be shown as the entry point to short documentation.
- `keywords`: It is a list of single words that help pinpoint your project's purpose.
- `authors`: It lists the project's key creators.
- `build`: It defines a file that is compiled and run before the rest of the program is compiled. This is often used to generate code.

Next is `[package.metadata.settings]`. Typically, Cargo complains about all keys and sections that it does not know about, but the sections with `metadata` in them are an exception. They are ignored by Cargo, so they can be used for any configurable key/value pairs you need for your program.

The `[features]`, `[dependencies]`, and `[build-dependencies]` sections tie in together. A dependency can be declared by a broad version number:

```
serde = "1.0"
```

This means that `serde` is a mandatory dependency (which is the default) and that we want the newest version, 1.0.*, of it but not for instance 1.1. The actual version will be fixed in `Cargo.lock`, updated by the `cargo update` command. Using the caret symbol broadens the versioning:

```
syntex = "^0.58"
```

Here, we're saying that we want the latest major version, 0.*.* but, at least, 0.58.*. Another way to define a dependency is to point to a Git repository:

```
time = { git = "https://github.com/rust-lang/time", branch = "master" }
```

That should be fairly self-explanatory. Again, the actual version (or in the case of Git, changeset revision) will be fixed in `Cargo.lock` by the `cargo update` command and not updated by any other command.

The example program has two optional dependencies, `mysql` and `sqlite`:

```
mysql = { version = "1.2", optional = "true" }
sqlite = { version = "2.5", optional = "true" }
```

This means that the program can be built without depending on either. The `[features]` section contains a list of the default features:

```
 default=["mysql"]
```

This means that if you do not manually override the feature set when building your program, only `mysql`, and not `sqlite`, will be depended on.

There's quite a lot more detail on how to configure your project with Cargo, but this will do for now. Take a look at https://crates.io for more.

Editor integrations

Rust's popularity pretty much guarantees that whichever coder's editor you are using, it has at least some preliminary out-of-the-box support for it. The Rust community has several tools that facilitate deeper support for text editors:

- **rustfmt**: It formats code according to conventions.
- **clippy**: It makes several additional checks that are beyond the scope of what a compiler should do. It can warn you of bad style and potential problems. Clippy relies on compiler plugins, so it is unfortunately available with nightly Rust only.
- **racer**: It can do lookups into Rust standard libraries, giving you code completion and tooltips.
- **rustsym**: It can query Rust code for symbols.

Of course, many editors integrate with **Cargo**.

All of these programs are Rust programs, so they can be found by a `cargo search` command and installed by `cargo install`. All of these tools can be used via the command line, but they become quite a bit more useful when properly integrated to an editor.

Currently, the popular text editors (Vim, Emacs, Visual Studio Code, Atom, and many others) have good Rust support. If your favorite editor is on this list, you're bound to have high-quality integration. For the sake of brevity, we'll focus on a single editor.

Visual Studio Code is a snappy and modern editor that has good and easy integration to these tools via the Rusty Code project. Let's go through the process of installing the stable tools (racer and rustfmt) and Visual Studio Code's Rusty Code. Installing the editor itself is beyond our scope here. See your operating system's package repositories and https://code.visualstudio.com for more information on the same.

To start, install the tools using Cargo by running the following commands:

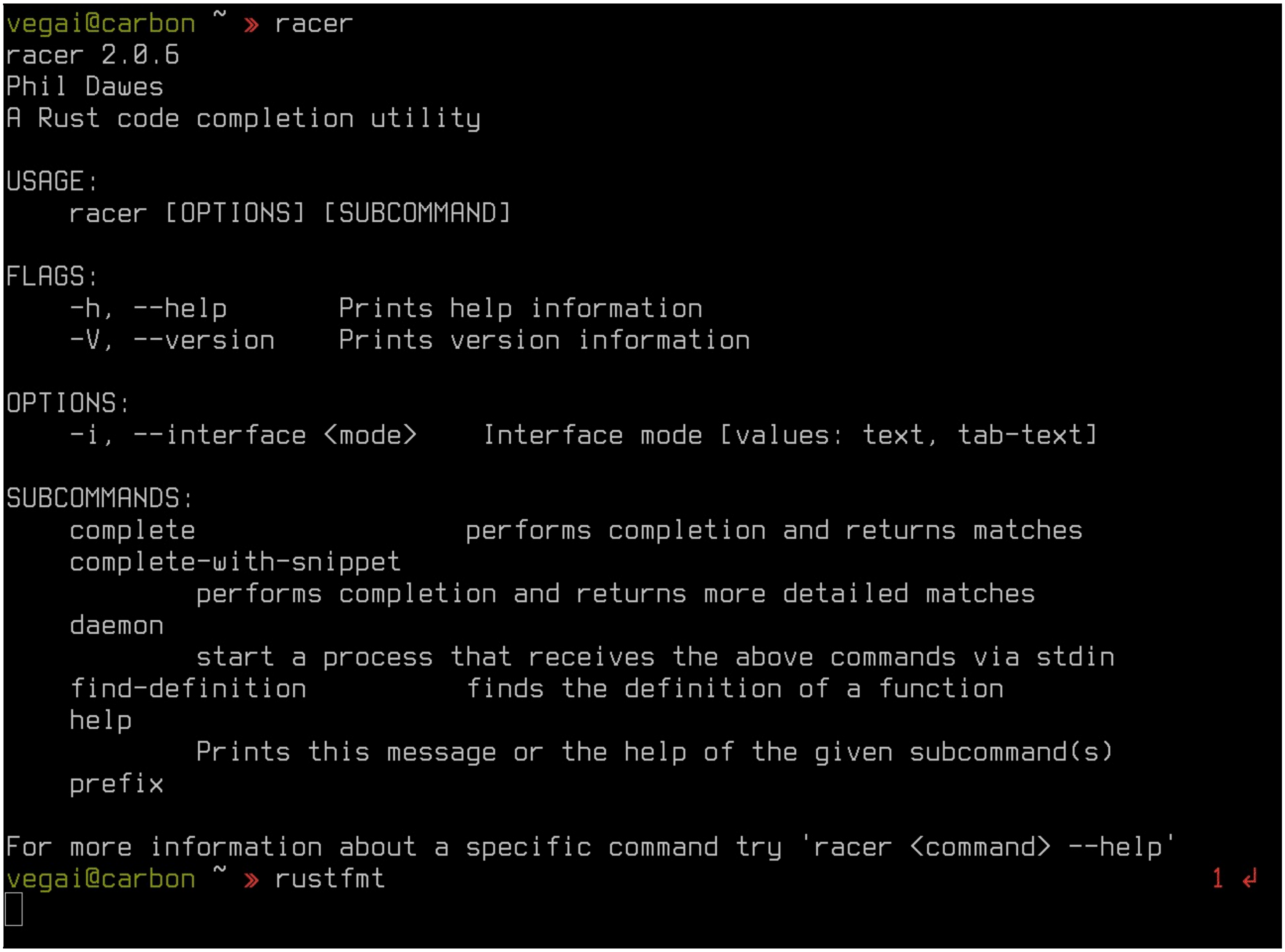
```
cargo install rustfmt
cargo install racer
```

I'll omit their output since if everything goes fine, it will be just a long list of dependencies downloaded and compiled for both of these programs. These are fairly large programs, so their installation will take 10-20 minutes, depending on your system. Of course, if your operating system's native package management system includes these packages, feel free to try those.

Racer requires Rust source code to be able to do lookups. You can get those locally with rustup using the following command:

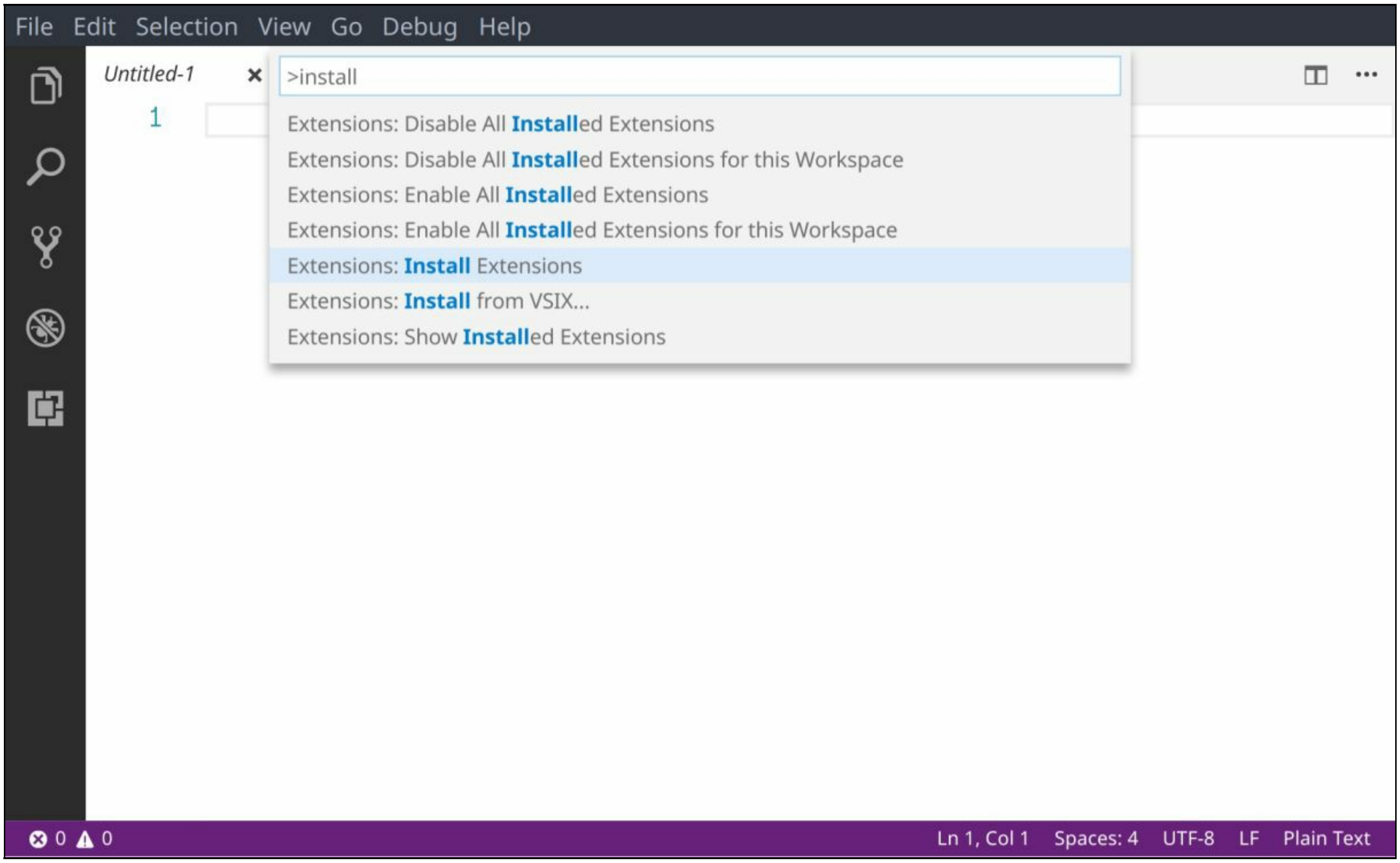
```
rustup component add rust-src
```

After installing the tools and getting the rustc source code, try both the commands on the command line to see that they are functioning properly and that they are in your `PATH`. Your output from trying them out should look something like this:

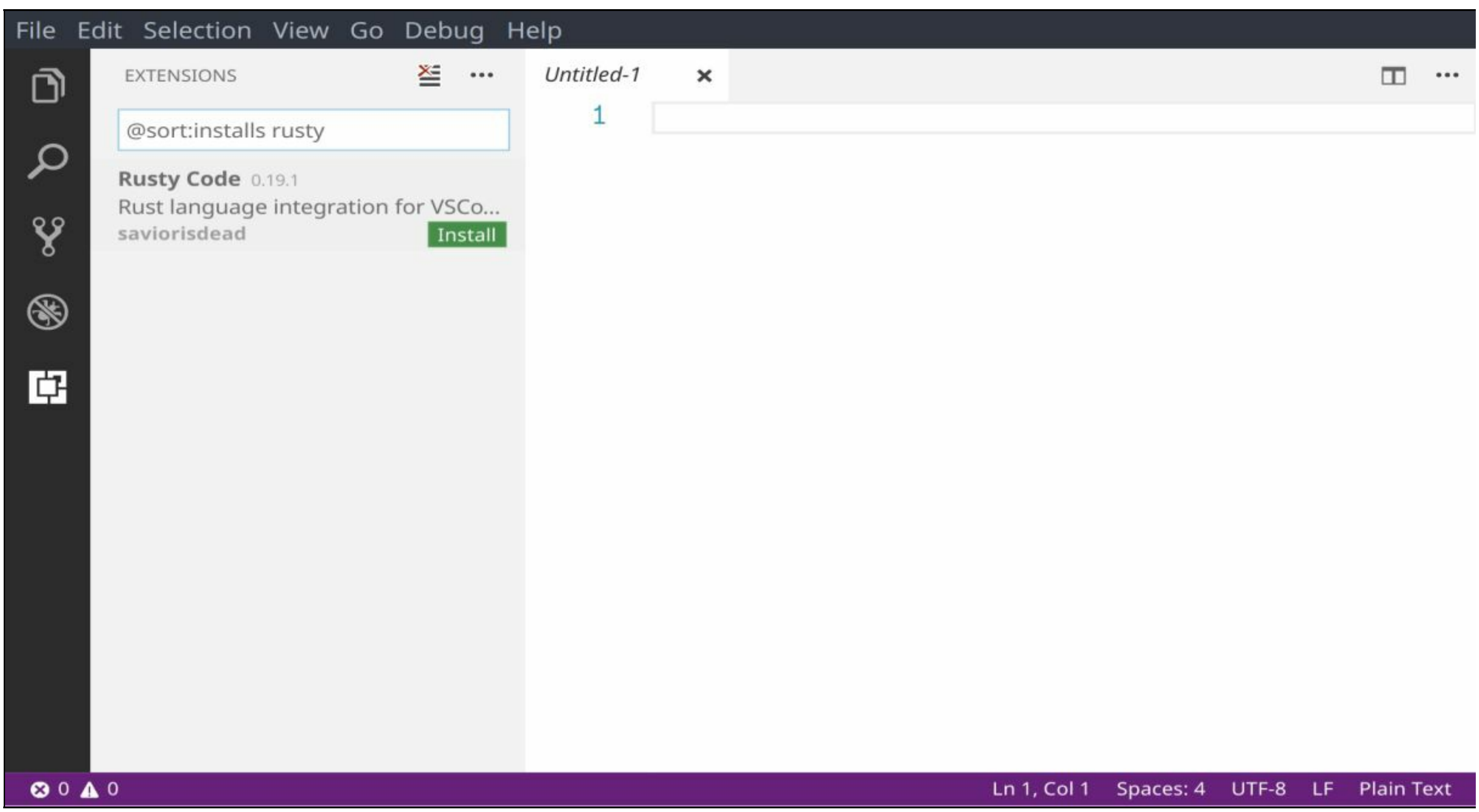


Since `rustfmt`, when run without parameters, takes Rust code from the standard input, it just waits there. Close the process with your typical *Ctrl* + *C*. If the commands are not in your `PATH`, add the default Cargo installation location `$HOME/.cargo/bin/` to your path and try again.

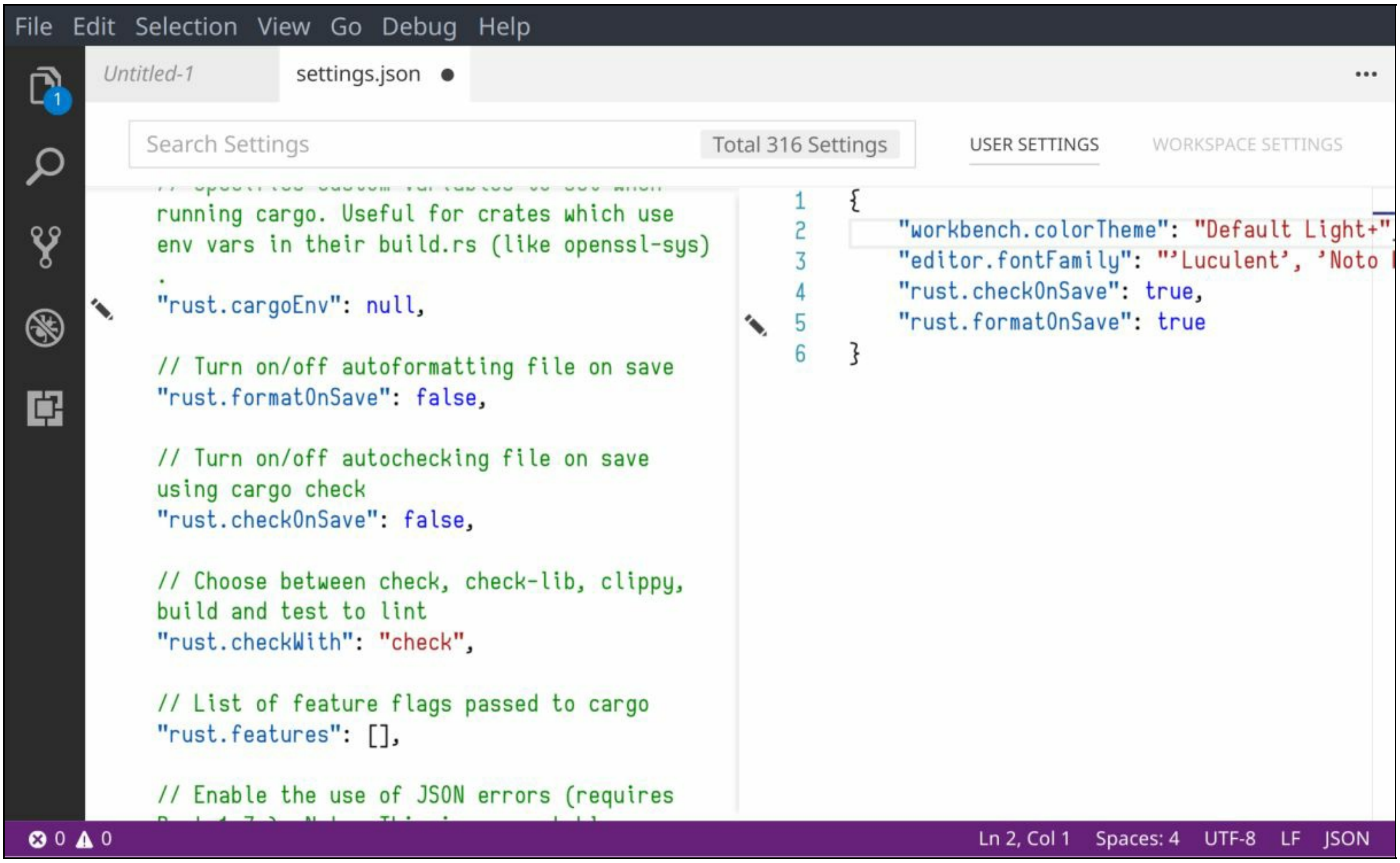
Next, fire up Visual Studio Code, use the command shortcut (*Ctrl* + *Shift* + *P*), type in `install`, and choose Extensions: Install Extensions:



Then, type in the word `Rusty` and let Visual Studio Code search for the extension for a while until you see something like this:

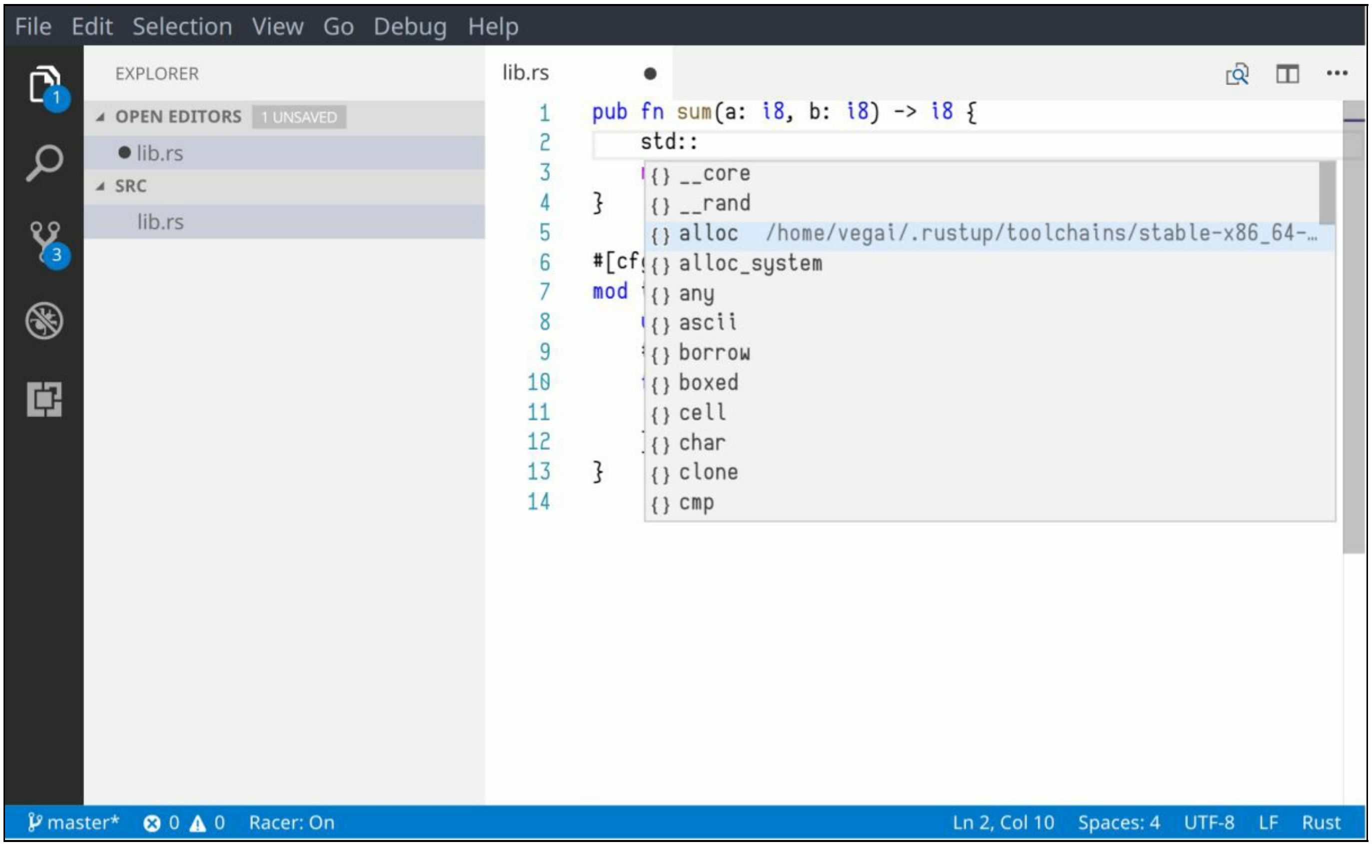


Then click on the Install icon. You may also need to reload the editor to enable the extension. To configure Rusty Code, select File | Preferences | Settings, and you'll get a two-pane view. The left side contains the global configuration and the right side contains your user's modifications to it. Browse down on the left pane a bit to find the Rusty Code section, and change the `checkOnSave` and `formatOnSave` variables by adding an override to your user's view on the right:

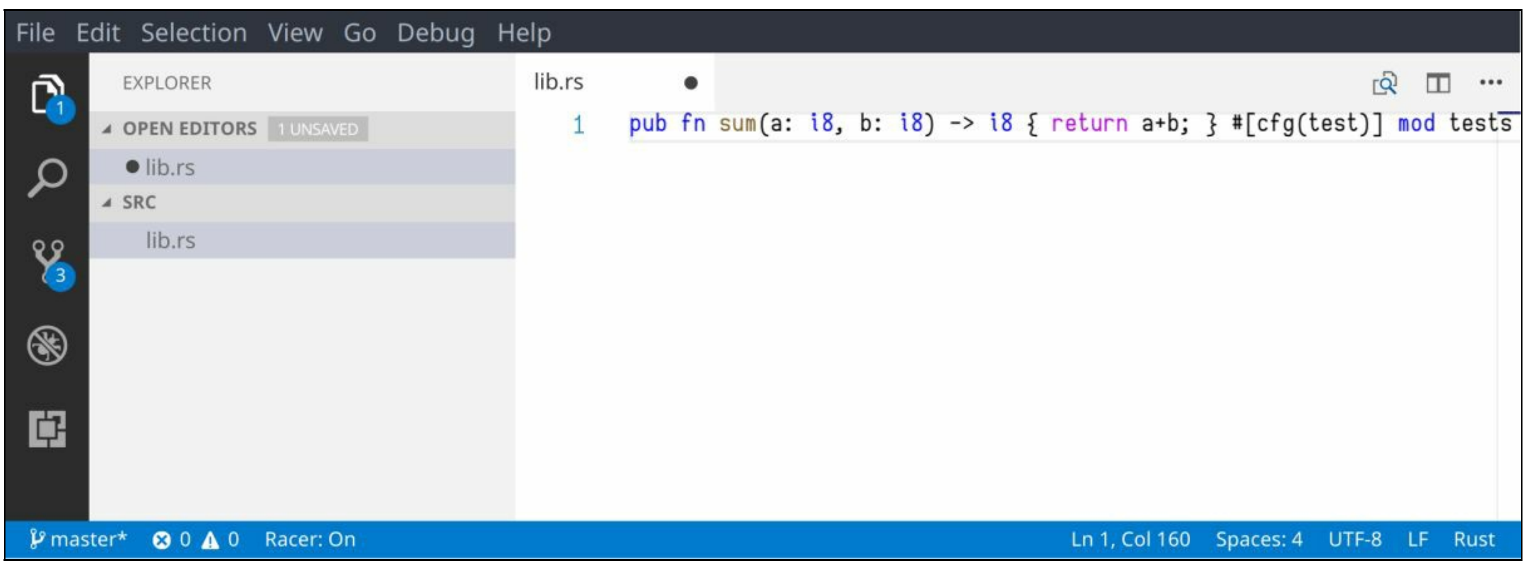


Remember to save the settings after making the changes. You can close the settings then.

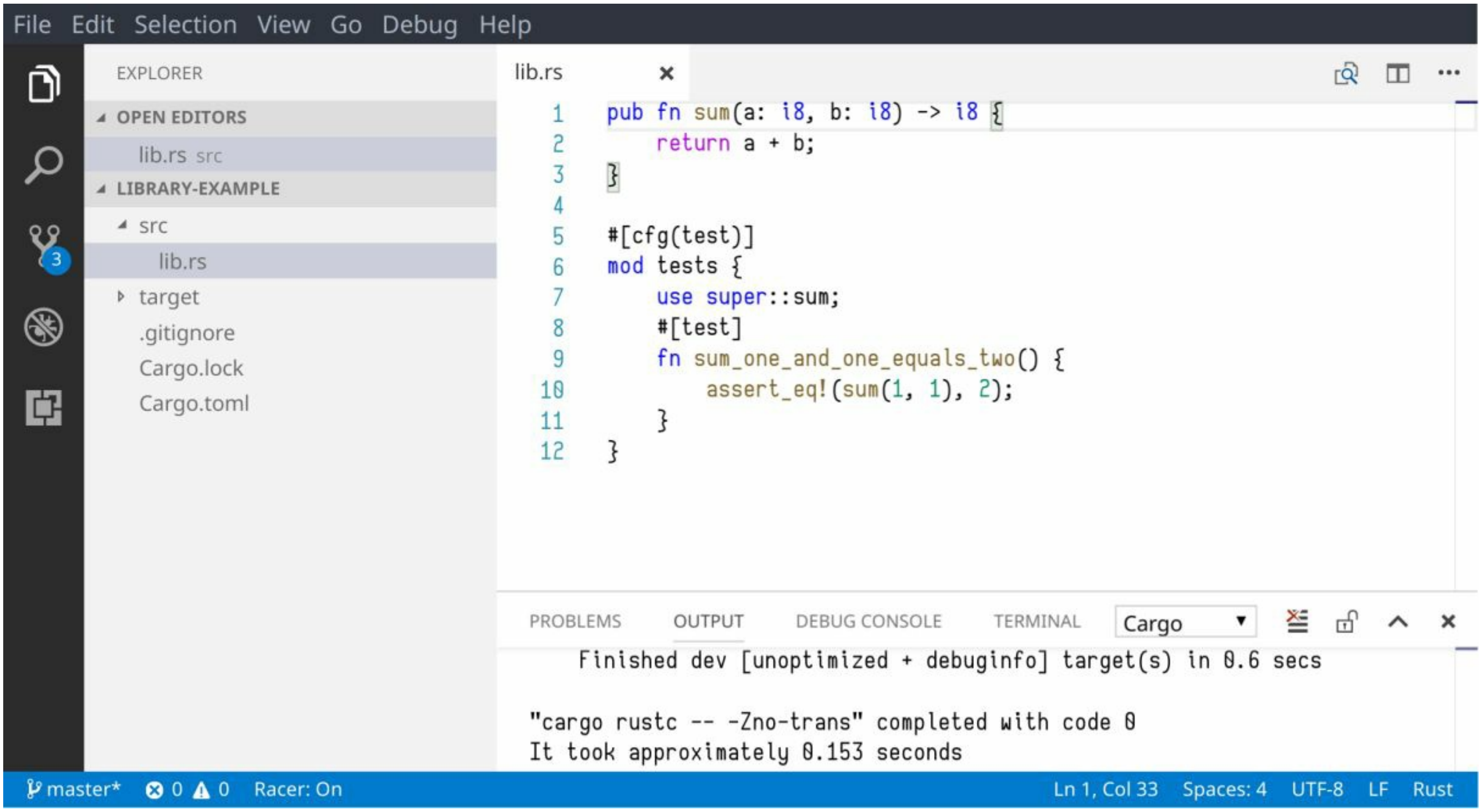
Next, let's take a cursory look at what Rusty Code does for us. Open up the project folder that you created earlier when trying out the unit tests. Then, find the `src/lib.rs` file and open it. All the standard libraries are in the standard namespace, so a nice demo will be to just start writing something starting with `std::` and following the suggestions:



Finally, let's take a look at how `formatOnSave` and `checkOnSave` settings work. We'll be a bit absurd and format the whole piece of code to be on a single line:



Then, save the file (*Ctrl* + *S*). Rusty Code will run your file through rustfmt, which formats your code if it can; if your syntax is bad enough, rustfmt cannot help you and will complain about it. Then, it will run `cargo check`, which runs rustc's build checks without generating the code:



That's enough introduction to editor integration. As for the other popular editors, here's a list of what is currently available:

- rust.vim for the **Vim** editor integrates with rustfmt and https://play.rust-lang.org.
- rust-mode for **Emacs** supports rustfmt integration. To get racer and other tools, you'll need separate packages. **Spacemacs** users can use the `rust` bundle to get them all.
- **Atom** has support via the `language-rust` package.
- **Sublime** has out-of-the-box support for Rust, with all the integrations.

Final exercise - starting our project

We now have a solid base on which to build some decent understanding. To drive it in, you should now build the foundation for the project that we will grow over the chapters.

The book's project will eventually be a multi-client strategy game. A game should be something where Rust's features, especially low resource usage, and multithread and memory safety, should benefit us greatly. Also, a game is open-ended enough in its specifications, so it will be rather easy to make use of all the features we need to cover. Last but not least, building games is fun, and mastering things that are fun is much easier.

Do the following:

1. Initialize a Cargo binary project. Call it whatever you want (but I will name the game `fanstr-buigam`, short for fantasy strategy building game).
2. Check `Cargo.toml` and fill in your name, email, and description of the project.
3. Try out the `cargo build`, `cargo test`, and `cargo run` commands.

Summary

In this chapter, you got acquainted with the standard Rust build tool, Cargo. We took a cursory look at initializing, building, and running the unit tests of your program. There are some tools beyond Cargo that can make the Rust coding experience smoother and safer, such as rustfmt, clippy, and racer. We saw how these may be integrated to a text editor by installing Visual Studio Code's Rusty Code extension. Finally, we founded a project that will be the basis for the game that we'll use to gain more skills.

The next chapter will be about unit testing. We'll start coding the book's project with a non-strict TDD style, finding out test cases where they are easy to find and filling in the implementation.

Unit Testing and Benchmarking

In this chapter, we will talk about the different methods of writing tests and benchmarks in Rust. Then, we'll put those skills to use and implement a few basic things in our project in a TDD style.

In this chapter, we will cover the following topics:

- Motivation for unit testing
- Test annotations
- Assert macros
- Integration tests
- Documentation tests
- Doing TDD with our project

Motivation and high-level view

As you may know already, automatic testing of small pieces of functionality is one of the most practical and effective ways of maintaining high code quality. It does not prove that bugs don't exist, but it hands out to your computer the most boring and repetitive task of checking known input-output pairs.

The consequences of the smart use of unit testing are profound. In the implementation phase, a well-written unit test becomes an informal specification for a tiny part of the system, which makes it very easy for the coder to recognize when a part has been completed, when its unit tests pass. Then, in the maintenance phase, the existing unit tests serve as a harness against regressions in the codebase, which again frees up those valuable mental reserves of the coder.

Rust has basic but robust built-in support for testing. It's made up of four things:

- Annotations like `#[test]`, `#[bench]`, `#[should_panic]`, `[cfg(test)]`, and `#[ignore]`
- Assert macros like `assert!` and `assert_eq!`
- Integration testing via tests/directory
- Documentation tests

Annotations

As you saw earlier, the test code can be included in the same file as the implementation code. Functions are marked as test functions by the `#[test]` annotation:

```
// test.rs
#[test]
fn test_case() {
    assert!(true)
}
```

The compiler ignores the test functions totally unless it's told to build in test mode. This can be achieved by using the `--test` parameter with `rustc` and executed by running the binary:

```
vegai@carbon ~/rustbook/3 » rustc --test test.rs
vegai@carbon ~/rustbook/3 » ./test

running 1 test
test test_case ... ok

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

vegai@carbon ~/rustbook/3 »
```

However, since Cargo has support for tests, this is usually done more easily via Cargo by commanding `cargo test` (which builds and runs the tests).

When your tests grow in complexity, it may be useful to isolate the test code from the real code. You can do this by encapsulating the test code inside a module, and tagging that module with the `#[cfg(test)]` annotation. The `#[cfg(...)]` annotation is more generally used for controlling compilation, for instance, including different code for different architectures or configurations. You might remember that the tests in the previous chapter were already using this form.

Say you want to programmatically generate test data for your tests but there's no reason to have that code in the release build. Here's how it will look:

```
// mod-test.rs
pub fn sum(a: i8, b: i8) -> i8 {
    return a+b;
}
#[cfg(test)]
mod tests {
    fn sum_inputs_and_outputs() -> Vec<((i8, i8), i8)> {
        vec![
            ((1, 1), 2),
            ((0, 0), 0),
            ((2, -2), 0),
        ]
    }

    #[test]
    fn test_sums() {
        for (input, output) in sum_inputs_and_outputs() {
            assert_eq!(::sum(input.0, input.1), output);
        }
```

```
    }
}
```

Here, we generate known input and output pairs in the `sum_inputs_and_outputs` function, which is used by the `test_sums` test function. The `#[test]` annotation is enough to keep the `test_sums` function away from our release binary. However, `sum_inputs_and_outputs` is not marked as `#[test]`, since it's not a test, so that would leak. Using `#[cfg(test)]` and encapsulating all the test code inside its own module, we get the benefit of keeping both the code and the resulting binary clean of the test code.

We use the double colon notation on the `assert_eq!` line:

```
assert_eq!(::sum(input.0, input.1), output);
```

The test code is running in the `tests` module, whereas the `sum` function is actually in the parent module. Previously, we worked around this by importing the function with the `use` expression, but this is another way to accomplish the same.

The `#[should_panic]` annotation can be paired with a `#[test]` annotation to signify that running the `test` function should cause a non-recoverable failure, which is called a **panic** in Rust. Here's a minimal passing test demonstrating `should_panic`:

```
// panic-test.rs
#[test]
#[should_panic]
fn test_panic() {
    panic!("Succeeded in failing!");
}
```

If your test code is exceedingly heavy, the `#[ignore]` annotation lets you make the test runner ignore such a test function by default. You can then choose to run such tests by supplying an `--ignore` parameter to either your test runner or the `cargo test` command. Here's the code containing a silly loop that is tested in a test that's ignored by default:

```
// silly-loop.rs
pub fn silly_loop() {
    for _ in 1..1_000_000_000 {};
}

#[cfg(test)]
mod tests {
    #[test] #[ignore]
    pub fn test_silly_loop() {
        ::silly_loop();
    }
}
```

Here's a sample run with the default test run and another that includes the ignored test:

```
vegai@carbon ~/rustbook/3 » rustc --test ignore-test.rs
vegai@carbon ~/rustbook/3 » time ./ignore-test

running 1 test
test tests::test_silly_loop ... ignored

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

./ignore-test  0.00s user 0.00s system 0% cpu 0.004 total
vegai@carbon ~/rustbook/3 » time ./ignore-test --ignored

running 1 test
test tests::test_silly_loop ... ok

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

./ignore-test --ignored  20.84s user 0.00s system 99% cpu 20.843 total
vegai@carbon ~/rustbook/3 »
```

Assert macros

The basic set has just two assertion macros for unit tests: `assert!` and `assert_eq!`. Both of these are quite simple. The `assert!` macro has two forms:

```
assert!(a==b);
assert!(a==b, "{} was not equal to {}", a, b);
```

The first form takes just a single Boolean value. If the value is `false`, the test run panics and shows the line where the failure happened.

The second form additionally takes in a format string and the corresponding number of variables. If the test fails, the test run panics, and in addition to showing the line number, displays the formatted text.

Even though `assert!` alone would be enough, comparing that two values are equal is such a typical case in unit tests. The `assert_eq!` macro does simply that: it takes two values and fails the test if they are not equal.

These macros are actually not specific to test code in any way; they are regular macros in the standard library, and you can use them just as well in your actual code to do assertions.

Integration or black box tests

The test/directory contains all the integration tests. These are tests that combine the usage of several larger pieces of code together. These tests are compiled as if they were separate crates. **Crate** is Rust's naming for external libraries, and the whole module system will be covered in the next chapter. The only thing we care about right now is that for an integration test, we need to specify all the crates we are using, even our program's own crate (which is `sum` in the next example).

In this example, I have created a project, `sum`, with the same contents in the library as in the previous unit test, and added this integration test:

```
sum-with-doctest/tests/sum.rs
extern crate sum;

#[test]
fn test_sum_integration() {
    assert_eq!(sum::sum(6, 8), 14);
}
```

Here's a view of the file tree of our `sum` example project with an integration test:

```
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── sum.rs
```

This looks similar to the unit test, but subtly different; in this case, we don't have any sort of a special view into the library. We are using it just like any user of our library would use it.

Documentation tests

It's often a good style to include examples in your documentation. There's a risk with such examples, though: your code might change and you might forget to update your documentation. Documentation tests help with that. They make the example code part of the unit test suite, which means that the examples run every time you run your unit tests, thus making forgetfulness painful earlier.

Documentation tests are included with your actual code, and come in two forms:

- Module-level at the start of the module, marked by `//!`
- Function-level before a function, marked by `///`

Doctests are executed via Cargo, so we'll need to make our `sum` function an actual project to try these out:

````
// sum-with-doctest/src/lib.rs
//! This crate has functionality for summing integers
//!
//! # Examples
//! ```
//! assert_eq!(sum::sum(2,2), 4);
//! ```

/// Sum two arguments
///
/// # Examples
///
/// ```
/// assert_eq!(sum::sum(1,1), 2);
/// ```
pub fn sum(a: i8, b: i8) -> i8 {
    return a+b;
}
````

As you see, the difference between module-level and function-level is more philosophical than technical. They are used in pretty much the same way but, of course, the module-level examples are typically higher level while the function-level examples cover just that one function.

Documentation tests run along with all the other tests when you run a `cargo test`. Here's how it looks when we run the test suite with the integration tests and the doctests:

```
vegai@carbon ~/rustbook/3/sum-with-doctest » cargo test
   Compiling sum v0.1.0 (file:///home/vegai/fossil/rustbook/3/sum-with-doctest)
    Finished dev [unoptimized + debuginfo] target(s) in 0.36 secs
     Running target/debug/deps/sum-73c1612b8e0590f1

running 0 tests

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

   Doc-tests sum

running 2 tests
test src/lib.rs - sum (line 12) ... ok
test src/lib.rs -  (line 4) ... ok

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

vegai@carbon ~/rustbook/3/sum-with-doctest »
```

Benchmarks

Just one more thing and we're done with the lecture portion. Benchmark tests allow us to bring automation to yet another boring and repetitive task: measuring the speed of our code. This is supported by two things:

- The `#[bench]` annotation marks a function as a benchmark
- The standard library module test has a `Bencher` type, which the `benchmark` function uses for benchmark iterations

Unfortunately, benchmark tests are an unstable feature, so we will have to use the nightly compiler for the following. Fortunately, with `rustup`, moving between different versions with the rustup compiler is easy. First, we'll make sure that the nightly compiler is installed:

```
vegai@carbon ~/rustbook/3 » rustup install nightly
info: syncing channel updates for 'nightly-x86_64-unknown-linux-gnu'
181.1 KiB / 181.1 KiB (100 %) 111.1 KiB/s ETA:   0 s
info: downloading component 'rustc'
 46.8 MiB /  46.8 MiB (100 %)   2.3 MiB/s ETA:   0 s
info: downloading component 'rust-std'
 75.8 MiB /  75.8 MiB (100 %)   2.0 MiB/s ETA:   0 s
info: downloading component 'cargo'
  4.9 MiB /   4.9 MiB (100 %)   3.7 MiB/s ETA:   0 s
info: downloading component 'rust-docs'
 11.8 MiB /  11.8 MiB (100 %)   2.9 MiB/s ETA:   0 s
info: installing component 'rustc'
info: installing component 'rust-std'
info: installing component 'cargo'
info: installing component 'rust-docs'

  nightly-x86_64-unknown-linux-gnu installed - rustc 1.18.0-nightly (1785bca51
2017-04-21)

vegai@carbon ~/rustbook/3 »
```

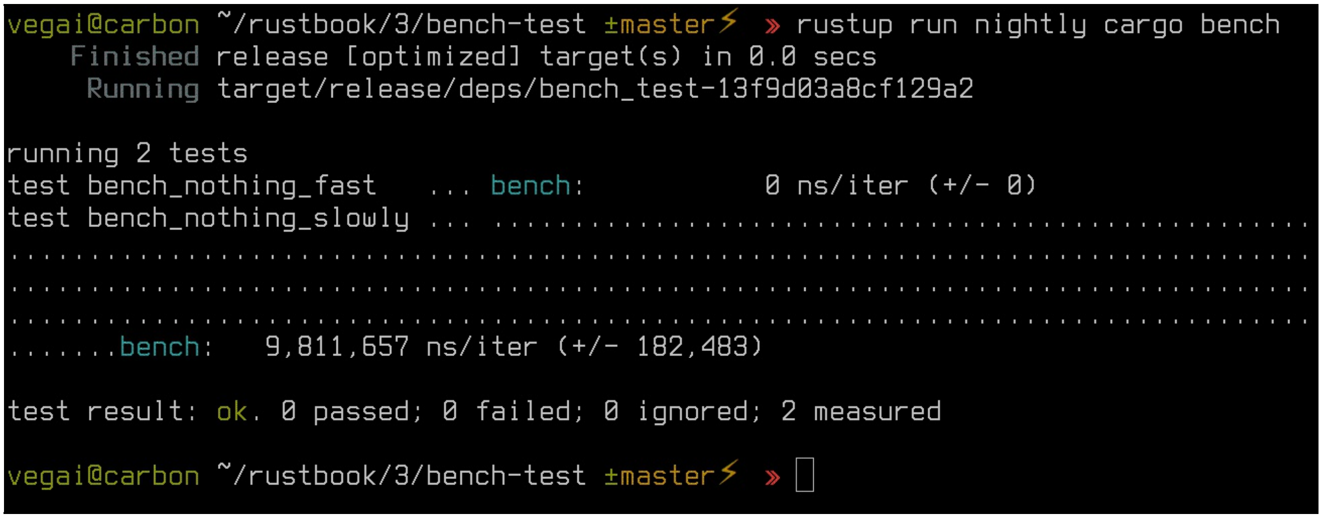
OK, now we can write and run a simple benchmark test. Benchmarks require Cargo, so we'll need a new project for this. Create a new library project using Cargo new. No changes to `Cargo.toml` are needed for this. The contents of `src/lib.rs` will be as follows:

```
// bench-test/src/lib.rs
#![feature(test)]

extern crate test;

use test::Bencher;

pub fn do_nothing_slowly() {
    print!(".");
    for _ in 1..10_000_000 {};
}

pub fn do_nothing_fast() {
}

#[bench]
fn bench_nothing_slowly(b: &mut Bencher) {
    b.iter(|| do_nothing_slowly());
}
```

```
#[bench]
fn bench_nothing_fast(b: &mut Bencher) {
    b.iter(|| do_nothing_fast());
}
```

The parameter to iterate is a closure with no parameters. If the closure had parameters, they would be inside the ||. This essentially means that `iter` is passed as a function that the benchmark test can run repeatedly. We print a single dot in the function so that Rust won't optimize the empty loop away.

Here's how it looks when we run it with Cargo bench. Note the usage of rustup to run this single command with the nightly compiler:



```
vegai@carbon ~/rustbook/3/bench-test ±master⚡ » rustup run nightly cargo bench
    Finished release [optimized] target(s) in 0.0 secs
     Running target/release/deps/bench_test-13f9d03a8cf129a2

running 2 tests
test bench_nothing_fast    ... bench:           0 ns/iter (+/- 0)
test bench_nothing_slowly ... .........................................................
.................................................................................
.................................................................................
.................................................................................
.......bench:   9,811,657 ns/iter (+/- 182,483)

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

vegai@carbon ~/rustbook/3/bench-test ±master⚡ »
```

Those are nanoseconds per iteration, with the figure inside the parentheses showing the variation between each run. Our slower implementation was, without surprise, quite a lot slower and not very stable (as shown by the large `+/-` variation).

Integrating with Travis

Travis is a public continuous integration service that allows you to run your project's unit tests automatically in the cloud. Continuous integration is a mechanism for doing various things against every new version of your code. It's generally used to maintain quality by automatically running builds and automatic tests, but can also be used for creating builds and even deploying them in staging or live environments. We'll focus on automatic running of unit tests here.

GitHub has integration to Travis, which runs arbitrary tests there for every commit. Here's what you need to make this happen:

- Your project in GitHub
- An account in Travis (you may use your GitHub account for this)
- Your project registered in Travis
- A `.travis.yml` file in your repository

The first step is going to https://travis-ci.org/ and logging in with your GitHub credentials. From there, you can add your project in GitHub to Travis.

Travis has good native support for Rust projects and keeps its Rust compiler continuously up-to-date. Here's what the `.travis.yml` file should contain:

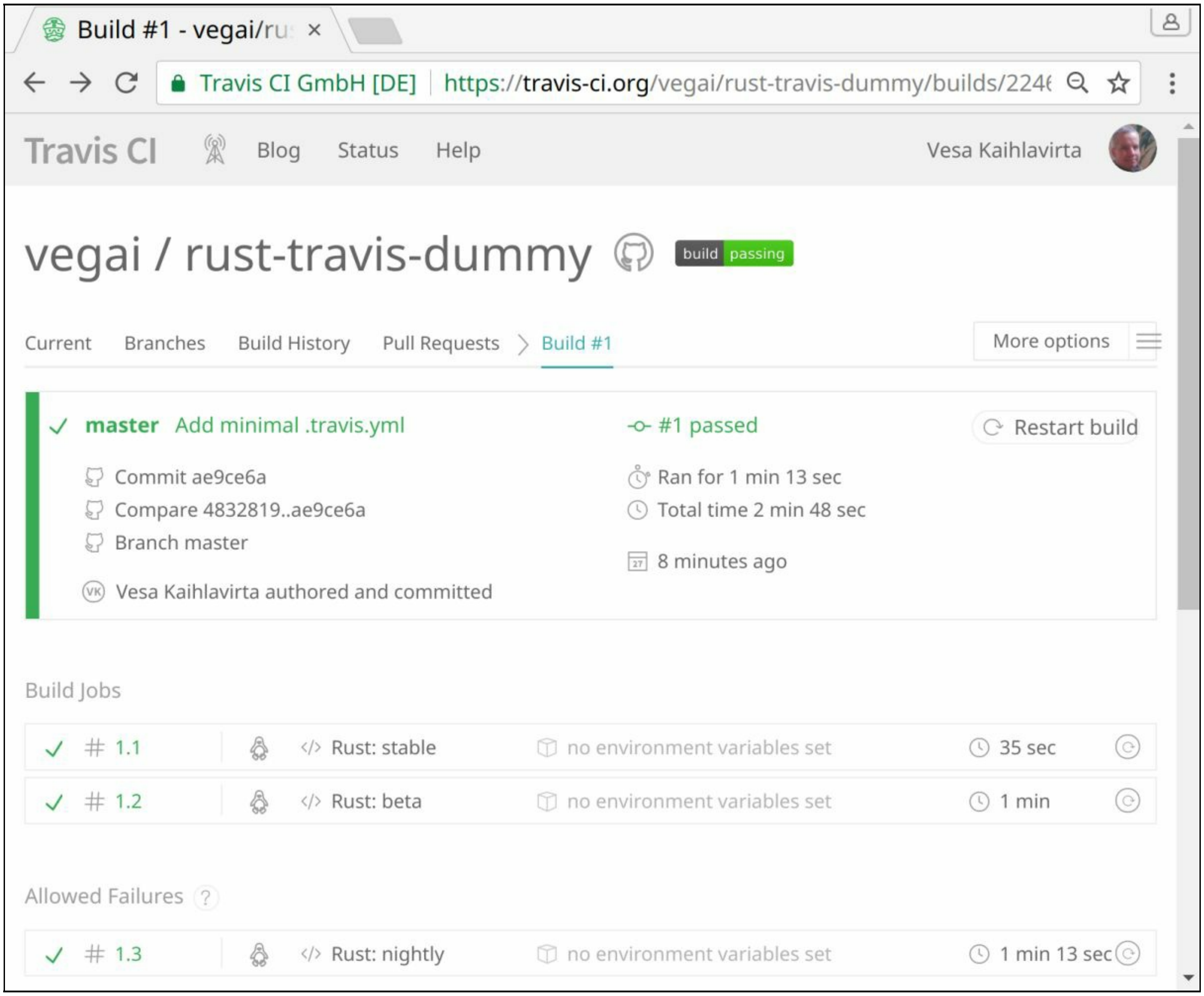
```
language: rust
rust:
  - stable
  - beta
  - nightly
matrix:
  allow_failures:
  - rust: nightly
```

The Rust project recommends testing against beta and nightly, but you may choose to target just a single version by removing some of the lines there. This recommended setup runs the tests on all three versions, but allows the fast-moving nightly compiler to fail.

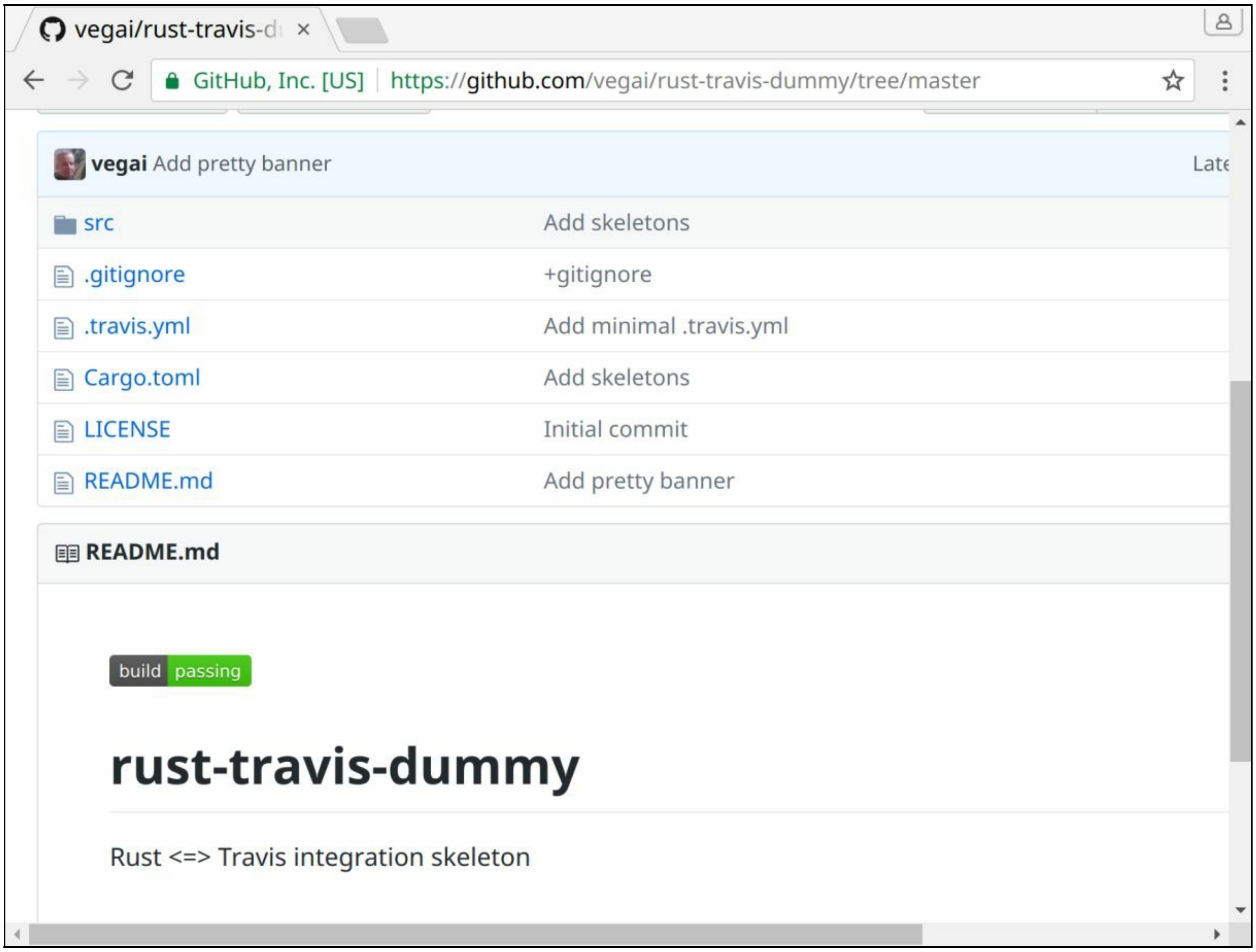
That's all you need. With the file in your repo, GitHub will inform Travis CI every time you push your code and run your tests. If you want to proudly tell the world about the latest test run results, add this line to your `README.md` (replacing `$user` with your `travis-ci user` and `$project` with your project name):

```
[![Build Status](https://travis-ci.org/$user/$project.svg?branch=master)](https://travis-ci.org/$user/$project)
```

Then your project will proudly show when its build and tests succeed! Here's an example of my dummy Travis project. First from the Travis side:



Now the corresponding GitHub page boasting the successful build in a banner:


vegai/rust-travis-d
GitHub, Inc. [US]
https://github.com/vegai/rust-travis-dummy/tree/master
vegai Add pretty banner
src
Add skeletons
.gitignore
+gitignore
.travis.yml
Add minimal .travis.yml
Cargo.toml
Add skeletons
LICENSE
Initial commit
README.md
Add pretty banner
README.md
build passing
rust-travis-dummy
Rust <=> Travis integration skeleton

Founding a city-builder game

Armed with all this knowledge, we can now try it out in practice!

In order to figure out what to test, we obviously need to have a clue about what we're trying to build. Here's the mile-high view of the project:

- Real-time city-building/strategy game
- Client/server architecture
- The game area is a 2D grid of squares, with each square having the following:
- A mandatory terrain ground
- An optional terrain block
- Objects
- Beings

Let's start by defining all the features of a square.

The terrain ground can be one of soil or stone. This unsurprisingly refers to the ground or floor.

The terrain block can be soil, stone, or tree. This refers to a non-passable block that can be left as a wall, or be mined or felled away.

Beings are living creatures, and each square may have one of them.

With these specifications, we can write the structs to `buistr/src/main.rs`:

```
// buistr/src/main.rs
[derive(PartialEq, Debug)]
enum TerrainGround {
  Soil,
  Stone
}

#[derive(PartialEq, Debug)]
enum TerrainBlock {
  Tree,
  Soil,
  Stone
}

#[derive(PartialEq, Debug)]
enum Being {
  Orc,
  Human
}

struct Square {
  ground: TerrainGround,
  block: Option<TerrainBlock>,
  beings: Option<Being>
}

struct Grid {
  size: (usize, usize),
  squares: Vec<Square>
}
```

`Option<T>` means that this is a thing of type `T` that might not exist. `Vector<T>` means a vector of things of type `T`. The type `T` is called a generic, which means that it can be any type. We'll cover generics more thoroughly in the next chapter.

All right, now we have the basic structures but no actual implementation yet. Let's start again with TDD by first defining with a test what we want to achieve. We'll want to be able to create an empty grid with some size. An empty grid would have nothing in it, except a soil ground in every square:

```
// buistr/src/main.rs
#[cfg(test)]
mod tests {
    #[test]
    fn test_empty_grid() {
        let grid = ::Grid::generate_empty(5, 13);
        assert_eq!(grid.size, (5, 13));
        let mut number_of_squares = 0;

        for square in &grid.squares {
          assert_eq!(square.ground, ::TerrainGround::Soil);
          assert_eq!(square.block, None);
          assert_eq!(square.beings, None);
          number_of_squares += 1;
        }

        assert_eq!(grid.squares.len(), 5*13);
        assert_eq!(number_of_squares, 5*13);
    }
}
```

So here's what the test code does:

1. Generates an empty grid with dimensions, *x=5*, *y=13*.
2. Asserts that the new grid gets its size set accordingly.
3. Goes through all the squares in the grid, tests that each contains a ground of soil, no blocks and no beings, and increments a counter that we check later.
4. Checks that the grid has as many squares as it should.
5. Double-checks that we checked that many squares in the loop.

Since we have no implementation yet, this test will fail before even starting:

```
vegai@carbon ~/rustbook/3/buist » cargo test
   Compiling buist v0.1.0 (file:///home/vegai/fossil/rustbook/3/buist)
error: no associated item named `generate_empty` found for type `Grid` in the cu
rrent scope
  --> src/main.rs:55:20
   |
55 |         let grid = ::Grid::generate_empty(5, 13);
   |                    ^^^^^^^^^^^^^^^^^^^^^^

error: aborting due to previous error

error: Could not compile `buist`.

To learn more, run the command again with --verbose.
vegai@carbon ~/rustbook/3/buist »                                       101 ↲
```

So, let's add the implementation:

```
// src/buistr/main.rs
impl Grid {
    fn generate_empty(size_x: usize, size_y: usize) -> Grid {
        let number_of_squares = size_x * size_y;
        let mut squares: Vec<Square> = Vec::with_capacity(number_of_squares);

        for _ in 1..number_of_squares {
            squares.push(Square{ground: TerrainGround::Stone, block: None, beings: None});
        }

        Grid {
            size: (size_x, size_y),
            squares: squares
        };
    }
}
```

Let's go through it:

1. We implement a `generate_empty` method for the `Grid` type.
2. It takes two variables, `size_x` and `size_y`, both of the `usize` type.
3. It returns `Grid`.
4. Compute the number of squares in the grid.
5. Create the vector that will hold the grid's squares via the `with_capacity` method of `Vec`. This preallocates the vector to our desired size.
6. Create all the square structures and set their ground, beings, and block values.
7. Return the grid.

Well, at least that's what we tried to do. Running `cargo test` gives us this:

```
vegai@carbon ~/rustbook/3/buist » cargo test
   Compiling buist v0.1.0 (file:///home/vegai/fossil/rustbook/3/buist)
error[E0308]: mismatched types
  --> src/main.rs:52:61
   |
52 |     fn generate_empty(size_x: usize, size_y: usize) -> Grid {
   |                                                             ^ expected struct
`Grid`, found ()
   |
   = note: expected type `Grid`
              found type `()`
help: consider removing this semicolon:
  --> src/main.rs:63:10
   |
63 |         };
   |          ^

error: aborting due to previous error

error: Could not compile `buist`.

To learn more, run the command again with --verbose.
vegai@carbon ~/rustbook/3/buist »                                       101 ↲
```

This works as a nice segue to the last section of this chapter.

Final exercise - fixing the tests

1. Fix the preceding compilation problem.
2. The code has a few other subtle problems, revealed by the tests. Fix those too.
3. After fixing the tests, the compiler warns about dead code. Find out how to suppress those warnings.

Summary

In this chapter, we went through writing unit tests, integration tests, documentation tests, and benchmarks using both `rustc` and `cargo`. You learned how to start using Travis CI for your public GitHub project. We wrote the first piece of actual code and a unit test for the book's project. In the next chapter, we'll go through how Rust's type system is built and how we can use it to keep our code safer.

Types

Rust's type system borrows a lot from the functional world with its structured types and traits. The type system is very strong: the types do not change under the hood and when something expects type X, you need to give it type X. Furthermore, the type system is static; nearly all of the type checks are done at runtime. These features give you very robust programs that rarely do the wrong thing during runtime, with the cost that writing the programs becomes a bit more constrained.

Rust's whole type system is not small, and we'll try to take a deep dive into it here. Expect lots of heavy material in this chapter, and be brave!

In this chapter, we will cover the following topics:

- String types
- Arrays and slices
- Generic types
- Traits and implementations
- Constants and statics

String types

Rust has two types of strings: string slices (`str`) and `String`. They are both guaranteed by runtime checks to be valid Unicode strings, and they are both internally coded as UTF-8. There are no separate non-Unicode character or string types; the primitive type `u8` is used for streams of bytes that may or may not be Unicode.

Why the two types? They exist mostly because of Rust's memory management and its philosophy of zero runtime cost. Passing string slices around your program is nearly free: it incurs nearly no allocation costs and no copying of memory. Unfortunately, nothing is actually free, and in this case, it means that you, the programmer, will need to pay some of that price. The concept of string slices is probably new to you and a tad tricky, but understanding it will bring you great benefits.

Let's dive in.

String slices

The `str` type is of a fixed size and its contents cannot be changed. Values of this type are typically used as borrowed types (`&str`) in one of these three ways:

- Pointing to a statically allocated string (`&'static str`)
- As arguments to a function
- As a view to a string inside another data structure

Let's see how each of these look in code. First, here are two types of statically allocated strings, one in the global scope and one in the function scope:

```
// string-slices.rs
const CONSTANT_STRING: &'static str = "This is a constant string";

fn main() {
    let another_string = "This string is local to the main function";

    println!("Constant string says: {}", CONSTANT_STRING);
    println!("Another string says: {}", another_string);
}
```

As you might remember, Rust has local type inference, which means that we can omit the types inside the body of functions when it suits us. In this case, it does suit us, indeed, since the type of both the strings is not pretty: `&'static str`. Let's read the type syntax piece by piece:

- `&` means that this is a reference
- `'static` means that the lifetime of the reference is static, that is, it lives for the whole duration of the program
- `str` means that this is a string slice

References and lifetimes are introduced more thoroughly in Chapter 6, *Memory, Lifetimes, and Borrowing*. Nevertheless, you can already understand that neither `CONSTANT_STRING` nor `another_string` is a string slice itself. Instead, they point to existing strings, and how those strings live during the execution of the program is explicitly specified by the lifetime, `'static`.

Let's then take a look at the second point: using string slices as arguments to functions. Unless you know what you are doing and specifically need something special, string slices is *the* way to pass strings into functions.

This is an important point, easy to miss, so it can be repeated for emphasis: if you are passing a string to a function, use the `&str` type. Here's an example:

```
// passing-string-slices.rs
fn say_hello(to_whom: &str) {
    println!("Hey {}!", to_whom)
}

fn main() {
    let string_slice: &'static str = "you";
```

```
    let string: String = string_slice.into();
    say_hello(string_slice);
    say_hello(&string);
}
```

Here, you can see why I stressed the point earlier. A string slice is an acceptable input parameter not only for actual string slice references but also for `String` references! So, once more: if you need to pass a string to your function, use the string slice, `&str`.

This brings us to the third point: using string slices as views into other data structures. In the preceding code, we did just that. The variable string is of the `String` type, but we can borrow its contents as a string slice simply by adding the reference operator `&`. This operation is very cheap, since no copying of data needs to be done.

The String type

OK, let's take a look at the higher level `String` type. Like the string slice, its contents are guaranteed to be Unicode. Unlike string slices, `String` is mutable and growable, it can be created during runtime, and it actually holds the data inside. Unfortunately, these great features have a downside. The `String` type is not of zero cost: it needs to be allocated in the heap and possibly reallocated when it grows. Heap allocation is a relatively expensive operation, but, fortunately for most applications, this cost is negligible. We'll cover memory allocation more thoroughly in Chapter 6, *Memory, Lifetimes, and Borrowing*.

A `String` type can be cast into `&str` rather transparently (as in the example we just saw) but not vice versa. If you need that, you have to explicitly request a new `String` type to be created from a string slice. That's what this line from the previous example does:

```
let string: String = string_slice.into();
```

Calling `into()` on anything is a generic way to convert a type from one value into another. Rust figures out that you want a `String` type because the type is (and must be) explicitly specified. Not all conversions are defined, of course, and if you attempt such a conversion, you will get a compiler error.

Let's take a look at the different ways of building and manipulating `String` types with the methods in the standard library. Here's a list of a few important ones:

- `String::new()` allocates an empty `String` type.
- `String::from(&str)` allocates a new `String` type and populates it from a string slice.
- `String::with_capacity(capacity: usize)` allocates an empty `String` with a preallocated size.
- `String::from_utf8(vec: Vec<u8>)` tries to allocate a new `String` from `bytestring`. The contents of the parameter must be UTF-8 or this will fail.
- The `len()` method gives you the length of the `String`, taking Unicode into account. As an example, a `String` containing the word *yö* has a length of 2, even though it takes 3 bytes in memory.
- The `push(ch: char)` and `push_str(string: &str)` methods add a character or a string slice to the String.

This is, of course, a non-exclusive list. A complete list of all the operations for `Strings` can be found at https://doc.rust-lang.org/std/string/struct.String.html.

Here's an example that uses all of the aforementioned methods:

```
// mutable-string.rs
fn main() {
    let mut empty_string = String::new();
    let empty_string_with_capacity = String::with_capacity(50);
    let string_from_bytestring: String = String::from_utf8(vec![82, 85, 83,
    84]).expect("Creating String from bytestring failed");

    println!("Length of the empty string is {}", empty_string.len());
    println!("Length of the empty string with capacity is {}",
    empty_string_with_capacity.len());
```

```
    println!("Length of the string from a bytestring is {}",
    string_from_bytestring.len());

    println!("Bytestring says {}", string_from_bytestring);

    empty_string.push('1');
    println!("1) Empty string now contains {}", empty_string);
    empty_string.push_str("2345");
    println!("2) Empty string now contains {}", empty_string);
    println!("Length of the previously empty string is now {}",
    empty_string.len());
}
```

Byte strings

The third form of strings is not actually a string but rather a stream of bytes. In Rust code, this is the unsigned 8-bit type, encapsulated in either a vector (`Vec<u8>`) or an array (`[u8]`). In rather the same way as string slices are usually used by references, so are arrays. So, the latter type is often used as `&[u8]`.

This is how we must work with strings when we're talking with the outside world. All your files are just bytes, just like the data we receive from and send to the internet. This might be a problem since not every array of bytes is valid UTF-8, which is why we need to handle any errors that might arise from the conversion. Recall the preceding conversion:

```
let string_from_bytestring: String = String::from_utf8(vec![82,
  85, 83, 84]).expect("Creating String from bytestring failed");
```

In this case, the conversion is OK, but had it not been, the execution of the program would have stopped right there, since, to repeat, strings in Rust are guaranteed to be Unicode.

Takeaways and tasks

Let's wrap up strings. Here's what to remember:

- There are two string types: `String` and `&str`
- Strings in Rust are guaranteed to be Unicode
- When passing strings to functions, favor the `&str` type
- When returning strings from functions, favor the `String` type
- Raw byte strings are arrays or vectors of 8-bit unsigned integers (`u8`)
- Strings are heap-allocated and dynamically grown, which makes them flexible but costlier

Here are a few tasks:

1. Create a few string slices and Strings, and print them. Use both `push` and `push_str` to populate a String with data.
2. Write a function that takes a string slice and prints it. Pass it a few static string slices and a few Strings.
3. Define byte strings with both UTF-8 strings and non-UTF-8 strings. Try to make Strings out of them and see what happens.
4. Make a String that contains the phrase `You are a Rust coder, Harry`. Split the String into words and print the second word. See https://doc.rust-lang.org/std/string/struct.String.html; you'll need to use the `collect()` method.

Arrays and slices

We've touched on arrays a couple of times. Let's look deeper.

Arrays contain a fixed number of elements of any single type. Their type is `[T; n]`, where `T` is the type of the contained values and `n` is the size of the array. Note that vector types (covered a bit later) give you dynamically sized arrays. This must be written out explicitly every time you wish to create an array yourself. Just like any other type, an array can be either mutable or immutable.

An array can be accessed by index by using the `[n]` syntax after the array name, very much like in other languages. This operation will cause a panic at runtime if you try to index beyond the length of the array.

Let's have a look at the following example:

```
// fixed-array-example.rs
fn main() {
    let mut integer_array_1 = [1, 2, 3];
    let integer_array_2: [u64; 3] = [2, 3, 4];
    let integer_array_3: [u64; 32] = [0; 32];
    let integer_array_4: [i32; 16438] = [-5; 16438];
    integer_array_1[1] = 255;

    println!("integer_array_1: {:?}", integer_array_1);
    println!("integer_array_2: {:?}", integer_array_2);
    println!("integer_array_3: {:?}", integer_array_3);
    // println!("integer_array_4: {:?}", integer_array_4);
    println!("integer_array_1[0]: {}", integer_array_1[0]);
    println!("integer_array_1[5]: {}", integer_array_1[5]);
}
```

That last line needs to be commented out, since only arrays equal to or less than the size of `32` get a `Debug` trait, which means that we cannot just go out and print larger ones. Here's what running this program looks like:

```
vegai@carbon ~/rustbook/4 » ./fixed-array-example
integer_array_1: [1, 255, 3]
integer_array_2: [2, 3, 4]
integer_array_3: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
integer_array_1[0]: 1
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 5'
, fixed-array-example.rs:13
note: Run with `RUST_BACKTRACE=1` for a backtrace.
vegai@carbon ~/rustbook/4 »                                                101 ↲
```

String slices have been mentioned already before, but slicing can be done for any array, not just strings. Slices are simple and cheap: they point to an existing data structure and contain a length. The type of a slice is close to that of arrays: `&[T]`. As you see, unlike arrays, this type does not have size information attached.

The syntax for slicing is `[n..m]`, where `n` is the inclusive starting point of the slice, and `m` is the non-inclusive endpoint. In other words, the element at `n` is included in the slice but the element at `m` is not. The index of the first element is `0`.

Here's an example of slice usage:

```
// array-slicing.rs
use std::fmt::Debug;

fn print_slice<T: Debug>(slice: &[T]) {
    println!("{:?}", slice);
}

fn main() {
    let array: [u8; 5] = [1, 2, 3, 4, 5];

    print!("Whole array just borrowed: ");
    print_slice(&array);

    print!("Whole array sliced: ");
    print_slice(&array[..]);

    print!("Without the first element: ");
    print_slice(&array[1..]);

    print!("One element from the middle: ");
    print_slice(&array[3..4]);

    print!("First three elements: ");
    print_slice(&array[..3]);

    //print!("Oops, going too far!: ");
    //print_slice(&array[..900]);
}
```

There's a `print_slice` function, which takes any values that implement the `Debug` trait. All such values can be fed into `println!` as parameters, and most internal types implement the `Debug` trait. Here's what running this program looks like:

```
vegai@carbon ~/rustbook/4 » ./array-slicing
Whole array just borrowed: [1, 2, 3, 4, 5]
Whole array sliced: [1, 2, 3, 4, 5]
Without the first element: [2, 3, 4, 5]
One element from the middle: [4]
First three elements: [1, 2, 3]
4 4
vegai@carbon ~/rustbook/4 »
```

So, you need to be rather careful when slicing. An overflow will cause a panic that will crash your program. It is also possible to make your own type indexable or sliceable by implementing a specific trait called Index, but we'll do that a bit later.

Takeaways and tasks

Here's what to remember about arrays and slices: arrays are of fixed size and the size needs to be known at compile time. The type is `[T; n]`, where `T` is the type of values in the array and `n` the size of the array:

- Slices are views into existing things and their size is more dynamic. The type is `&[T]`
- To pass sequences to functions, favor slices
- An `Index` trait can be used to make your own types indexable or sliceable

Here are some tasks you should try for yourself:

1. Make a 10-element fixed array of numbers.
2. Take a slice that contains all elements of the previous array except the first and the last.
3. Use for x in xs (shown briefly in Chapter 1, *Getting Your Feet Wet*) to sum all the numbers in both the array and the slice. Print the numbers.

Generic types

Imagine a situation where you need to encapsulate some values inside something else. Vectors, HashMaps, Ropes, all sorts of trees, and graphs... the amount of possible useful data structures is endless, and so is the amount of possible types of values you might want to put in them. Furthermore, a useful programming technique is to encapsulate your types inside others to enhance their semantic value, which may at best increase the clarity and safety of your code.

Now, imagine that you need to implement a method for such a type, such as fetching a specific key from a HashMap. HashMaps have keys, which point to values. Naively, you would need to write a specific method for each key-value type pair that you need in your program, even if all those methods are likely to be identical. That's what generic types make more convenient: they allow you to parameterize the type you are encapsulating, leading to dramatic decreases in code duplication and source maintenance.

There are two ways of making your own generic types: enums and structs. The usage is similar to what we had before, but now, we're including the generic type with the declaration. The type may be any uppercase letter enclosed in angle brackets. By default, the letter `T` is used when there's no reason to specify otherwise. Here are the examples of the `Option` and `Result` enums from the standard library, which are generic:

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

enum Result<T, E> {
   Ok(T),
   Err(E)
}
```

These are the types you will be using when you need values that are optionally empty, or when you have operations that might or might not succeed. They have several operations revolving around these concepts, such as `Option` types methods:

- `is_some` returns `true` if the `Option` type has a value, `false` otherwise
- `is_none` works the same as `is_some` but vice versa
- `expect` extracts the value from inside the `Option` type or panics if it was `none`

See the full list at https://doc.rust-lang.org/std/result/. The point is that none of these methods rely on what actual types are wrapped inside `Option`; they just operate on the wrappings. Therefore, they are perfect as generic types.

Here's a struct with generic parameters and examples of using it:

```
// generic-struct.rs
#[derive(Debug)]
struct Money<T> {
     amount: T,
```

```
        currency: String
}

fn main() {
    let whole_euros: Money<u8> = Money { amount: 42, currency: "EUR".to_string() };
    let floating_euros: Money<f32> = Money { amount: 24.312, currency: "EUR".to_string() };

    println!("Whole euros: {:?}", whole_euros);
    println!("Floating euros: {:?}", floating_euros);
}
```

Finally, we can define generic types for functions. Totally generic function parameters are rather constrained, though, but they can be enhanced with trait bounds, which we will cover a bit later. Here's an example that returns the first of the two parameters:

```
// generic-function.rs
fn select_first<T>(p1: T, _: T) -> T {
    p1
}

fn main() {
    let x = 1;
    let y = 2;

    let a = "meep";
    let b = "moop";

    println!("Selected first: {}", select_first(x, y));
    println!("Selected first: {}", select_first(a, b));
}
```

Since the function defines just a single type `T` for the function, the types need to match at the call site. In other words, the following call would not have been OK:

```
select_first(a, y);
```

This is because `T` has to be able to form a concrete type, and the type cannot be a string slice and a number at the same time.

Takeaways and tasks

Here are the key points from this section:

- The syntax for generic types is `<T>`, where `T` can be of any valid Rust type.
- In every block where it is used, it needs to be declared before it can be used. For instance, when declaring a function, `<T>` is declared just before the argument list.
- The generic type, `Option`, in the standard library is used to represent any value that might be nothing.
- The generic type, `Result`, is used to represent operations that may or may not succeed.

Here are a couple of tasks for you:

1. Take a look at the collection types, documented in https://doc.rust-lang.org/stable/std/collections/.
2. Use HashMap for any key-value type pairs you choose.
3. Use BTreeMap for any key-value type pairs.
4. Take a look at the `new` methods of various collections. Notice the difference in the generic type. Think about what they might mean.

Traits and implementations

Traits and **implementations** are similar to interfaces and classes that implement those interfaces in object-oriented languages. Even though object-oriented is a very liberal term, which might mean lots of different things, here are some key differences between typical OO languages and Rust:

- Even though traits have a form of inheritance in Rust, implementations do not. Therefore, composition is used instead of inheritance.
- You can write implementation blocks anywhere, without having access to the actual type.

The syntax for `trait` blocks defines a set of types and methods. A very simple `trait` declaration would look as follows:

```
trait TraitName {
   fn method(&self);
}
```

An implementation for this `trait` would need to specify something for all these things. Here's how it might look if we had a type called `MyType` and wanted to implement the preceding trait for it:

```
    impl TraitName for MyType {
        fn method(&self) {
              // implementation
        }
    }
```

Let's approach traits by taking a look at a few from the standard library and implementing them for our `Money<T>` type:

- The `std::ops::Add` trait lets us overload the `+` operator
- The `std::convert::Into` trait lets us specify conversion methods from and to arbitrary types
- The `Display` trait lets us specify how our type is formatted as a string

In particular, `Into` and `Display` are traits that you quite probably will want to implement for your own types.

Let's start with the implementation of the `Add` trait. To start, we'll have to check the documentation of `Add`, so we'll have some clue what is expected of us. The definition of the trait is provided at https://doc.rust-lang.org/std/ops/trait.Add.html:

```
pub trait Add<RHS = Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

Let's look at this line by line:

- `pub trait Add<RHS = Self>` says that the trait has a generic type `RHS` that needs to be equal to the `Self` type.

- `type Output` says that any implementation needs to declare an `Output` type.
- `fn add(self, rhs: RHS) -> Self::Output` says that any implementation needs to implement an add method that takes a right-hand side parameter that was declared on the first line to be the same as the `Self` type. In other words, the left-hand side and the right-hand side around the `+` operator need to be of the same type. Finally, it says that this `add` method must return the type that we declared on the second line.

All right, let's try it. Here's the code:

```
// std-trait-impls.rs
use std::ops::Add;

#[derive(Debug)]
struct Money<T> {
     amount: T,
        currency: String
}

impl<T: Add<T, Output=T>> Add for Money<T> {
    type Output = Money<T>;
    fn add(self, rhs: Money<T>) -> Self::Output {
        assert!(self.currency == rhs.currency);
        Money { currency: rhs.currency, amount: self.amount + rhs.amount }
    }
}

fn main() {
    let whole_euros_1: Money<u8> = Money { amount: 42, currency: "EUR".to_string() };
    let whole_euros_2: Money<u8> = Money { amount: 42, currency: "EUR".to_string() };
    let summed_euros = whole_euros_1 + whole_euros_2;

    println!("Summed euros: {:?}", summed_euros);
}
```

That's intimidating! But worry not; let's attack the beast at the `impl` block:

```
impl<T: Add<T, Output=T> Add for Money<T>
```

There's a new concept here called a **trait bound**. Trait bounds are for giving boundaries for generics. This lets us tell the compiler that we are not defining all types but only a certain subset of them. This is not just an optional stage but is needed for the type checking to pass. Let's split this into pieces.

The `impl<T: Add<T, Output=T>` line of code says that our implementation has a generic type `T`, but we are giving additional bounds to the type:

- `Add for Money<T>`: This says that we are implementing the `Add` trait for the type, `Money<T>`, where what `T` stands for was declared earlier on the line
- `T: Add`: This type has to implement the `Add` trait. If it does not, we can't use the `+` operator on it
- `<T, Output=T>`: Furthermore, the implementation of the `Add` trait must have its input and output types as the same

It's kind of obvious that this is exactly what we need. What is unfortunate and slightly complicated is that the compiler has no way of guessing these things for us and still maintains the strong and static guarantees that it needs to. We need to spell it out.

Then the `Into` trait; you have seen usages of this trait before: the trait declares an `into` method, giving us a general way to do conversions between types. The trait documentation is at https://doc.rust-lang.org/std/convert/trait.Into.html. Here's the trait:

```
    pub trait Into<T> {
        fn into(self) -> T;
    }
```

This is a bit simpler than the previous one. When we implement this, we just need to give the output type and then we can use the method on our type. Here's an implementation that converts our `Money<T>` to a new (and a bit silly) type, `CurrencylessMoney<T>`:

```
// into-impl.rs
use std::convert::Into;

struct Money<T> {
     amount: T,
    currency: String
}

#[derive(Debug)]
struct CurrencylessMoney<T> {
     amount: T
}

impl<T> Into<CurrencylessMoney<T>> for Money<T> {
    fn into(self) -> CurrencylessMoney<T> {
        CurrencylessMoney { amount: self.amount }
    }
}

fn main() {
    let money = Money { amount: 42, currency: "EUR".to_string() };
    let currencyless_money: CurrencylessMoney<u32> = money.into();

    println!("Money without currency: {:?}", currencyless_money);
}
```

Again, let's look at the `impl` line. This is similar to the `Add` trait, except that we don't have to bound the generic by any special output type, since `Into` does not have that:

```
impl<T> Into<CurrencylessMoney<T>> for Money<T>
```

The first `<T>` is a declaration of the generic type `T`, and the second and third are the usages of it. If you passed the `Add` trait with flying colors, this should be rather easy.

Finally, let's talk about the `Display` trait. It's documented at https://doc.rust-lang.org/std/fmt/trait.Display.html, and here's the trait:

```
    pub trait Display {
        fn fmt(&self, &mut Formatter) -> Result<(), Error>;
    }
```

Nothing fancy, but again, as we're making an implementation for a generic type, we'll have to spell some things out again. Here's an example implementation for the `Money<T>` type:

```
// display-trait.rs
use std::fmt::{Formatter, Display, Result};

struct Money<T> {
```

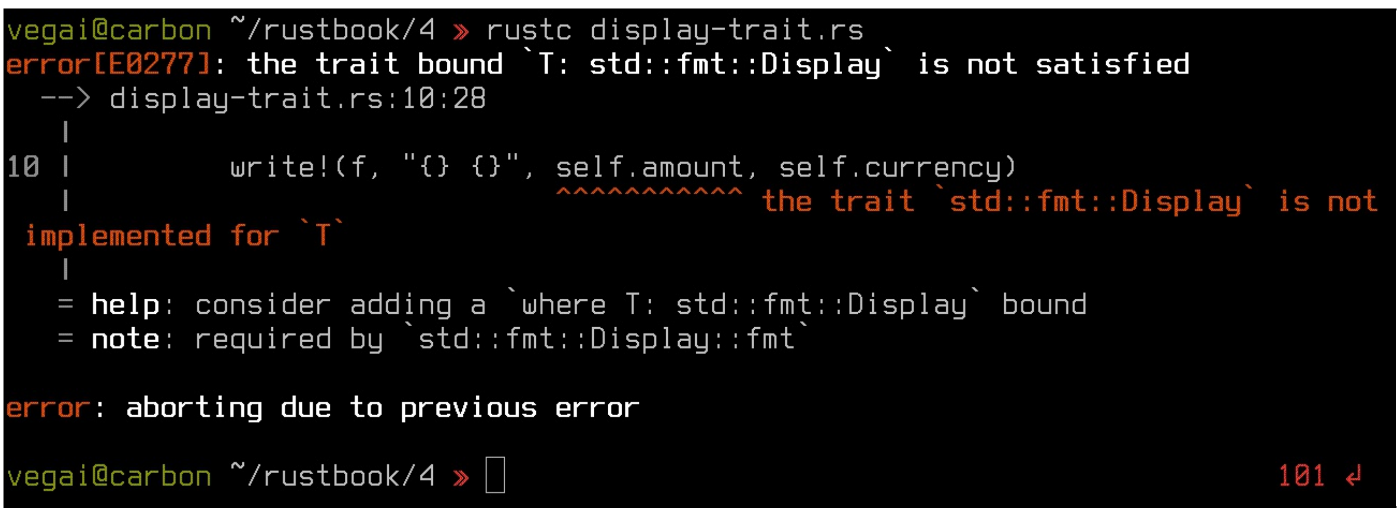
```
    amount: T,
        currency: String
}

impl<T: Display> Display for Money<T> {
    fn fmt(&self, f: &mut Formatter) -> Result {
        write!(f, "{} {}", self.amount, self.currency)
    }
}

fn main() {
    let money = Money { amount: 42, currency: "EUR".to_string() };
    println!("Displaying money: {}", money);
}
```

The `fmt` method that we need to implement in order to fulfill the `Display` trait takes in `Formatter`, which we write into using the `write!` macro. Like before, because our `Money<T>` type uses a generic type for the amount field, we need to specify that it also must satisfy the `Display` trait.

Let's see what happens if we don't specify that bound, that is, if our `impl` line looks like this:

```
impl<T> Display for Money<T>
```

This is akin to saying that we are trying to implement this display for any type `T`. However, that is not OK, since not all types implement the things we are using in our `fmt` method. The compiler will tell us about it as follows:



Lastly, we'll cover one more thing about traits: a concept called **trait objects**. A value can be given a type that is a trait, which means that the type can contain any object that implements that trait. This is a form of dynamic dispatch, since any decision about the real types of things can be only made at runtime. Here's an example of storing two different types inside a single `Debug` trait object:

```
// trait-object.rs
use std::fmt::Debug;

#[derive(Debug)]
struct Point {
    x: i8,
    y: i8
}

#[derive(Debug)]
struct ThreeDimPoint {
    x: i8,
```

```
    y: i8,
    z: i8
}

fn main() {
    let point = Point { x: 1, y: 3};
    let three_d_point = ThreeDimPoint { x: 3, y: 5, z: 9 };

    let mut x: &Debug = &point as &Debug;
    println!("1: {:?}", x);

    x = &three_d_point;
    println!("2: {:?}", x);
}
```

Takeaways and tasks

OK, time to wrap traits up for now and head for the summary and exercises. Here's what you need to remember from traits:

- Traits are like interfaces in OO languages: they allow giving types additional functionalities in a controlled way
- We can make types fulfill the traits by supplying `impl` blocks
- We can define `impl` blocks for types that have generics too, although we need to be careful about the type bounds
- Type bounds allow us to narrow down our generic types by declaring what traits need to be implemented for a type

And here's some extra work you can and should do to drive traits in:

1. Make your own type, without generics. Perhaps just strip off the generic type from our `Money<T>`. Implement some or all of the operators for it. For more information refer to https://doc.rust-lang.org/std/ops/index.html.
2. The same as the previous exercise but make your type have generics (or use the `Money<T>` type in from this section).
3. Implement a `Point` struct that describes a point in 2D space.
4. Implement a `Square` struct that uses the `Point` struct defined in the previous exercise for a coordinate.
5. Implement a `Rectangle` struct likewise.
6. Make a trait `Volume` that has a method for getting the size of something. Implement `Volume` for `Square` and `Rectangle` structs.

Congratulations on beating the toughest parts of this chapter! Now, we can relax a bit with a few lighter subjects.

Constants and statics

Rust allows defining global values that are visible everywhere in your program. However, since global variables are a breeding ground for the nastiest bugs out there, there are some safety mechanisms. Immutable globals are fine, but mutable globals need be used inside **unsafe** blocks. Before looking at the dangerous parts, let's first see how the immutable globals work.

The first form of global values is **constants.** These are good for giving descriptive names for your literal values that do not need to change during the lifetime of a program. As you might remember from before, Rust has local type inference, but not global. This means that the types of constants need to be written manually. Here's how the syntax is:

```
const THE_ANSWER: u32 = 42;
```

Now, you can use `THE_ANSWER` where you would use the literal `42`, otherwise. Note that constants are essentially just replaced into your code, so they do not actually exist as separate entities during the running of your program. Also note that constants are, by convention, written in all caps, and the compiler warns you if you don't follow this convention.

The other form is **statics.** Unlike constants, these are global values that actually exist during the runtime and can be made mutable. However, since mutable globals are inherently dangerous, all usage of them has to be enclosed in an unsafe block. Here's an example:

```
// statics.rs
static mut meep: u32 = 4;
static FUUP: u8 = 9;

fn main() {
    unsafe {
        println!("Meep is {}", meep);
        meep = 42;
        println!("Meep is now {}", meep);
    }

    println!("Fuup is {}", FUUP);
}
```

While immutable statics can be used everywhere, mutable statics can only be used (even if just for read access) inside unsafe blocks.

Generally, if you don't need the memory location of your global values for anything, you should prefer using `consts`. They allow the compiler to make better optimizations and are more straightforward to use.

Summary

Pat yourself in the back for a job well done. This chapter might not be the most advanced of the book, but the content was probably the heaviest. You now have a working knowledge of the different string types, collections (arrays, slices, and Vectors). You know about trait and implementation blocks, and you know how to work with constant values.

Armed with these, we can proceed to the next chapter, where we will talk about how error situations are handled in Rust.

Error Handling

In this chapter, we'll take a look at how unexpected situations are handled in Rust. Rust's error handling is based on generic types, such as `Option` and `Result`, which we saw in the previous chapter. There's also a mechanism called panicking, which is similar to exceptions, but unlike exceptions in other languages, panics are not used for recoverable error conditions.

The topics covered in this chapter include:

- The `Option` and `Result` types
- Matching against the `Option` and `Result` types
- Helper methods for handling errors
- The `try!` macro
- The `?` operator
- Panics
- Custom errors and the `Error` trait

Option and Result

A clear majority of error handling in Rust is done via these two generic types, or a type that looks and behaves very much like them. Operations that might fail do not throw exceptions but rather return one of these types. You may be curious as to why Rust chose this method instead of the more mainstream approach of exceptions and stack unwinding. Let's think about that for a moment.

First of all, exceptions have an overhead. In a nutshell, they can be implemented in languages in two ways:

- Make the code that runs without errors very cheap, but error cases very expensive
- Make error cases very cheap, but non-error cases expensive

Neither of these works well with Rust's central philosophy of zero runtime cost.

Secondly, exception-style error handling, as it is typically implemented, allows ignoring the errors via catch-all exception handlers. This creates a potential for fatal runtime errors, which goes against Rust's safety tenet.

We'll see how all this works in practice. Here's the `Result` type that we've seen a couple of times already:

```
enum Result<T, E> {
   Ok(T),
   Err(E),
}
```

A `Result` holds two types, `T` and `E`. `T` is the type that we use for a successful case and `E` is the error case. We'll try to open a file, read its contents into a `String`, and print those contents. Let's see what happens when we naively act as if we can just ignore all the error cases:

```
// result-1.rs
use std::io::Read;
use std::path::Path;
use std::fs::File;

fn main() {
    let path = Path::new("data.txt");
    let file = File::open(&path);
    let mut s = String::new();
    file.read_to_string(&mut s);

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

This is how the compiler responds:

```
vegai@carbon ~/rustbook/5 » rustc result-1.rs
error: no method named `read_to_string` found for type `std::result::Result<std::fs::File, std::io::Error>` in the current scope
 --> result-1.rs:9:10
  |
9 |     file.read_to_string(&mut s);
  |          ^^^^^^^^^^^^^^
  |

error: aborting due to previous error

vegai@carbon ~/rustbook/5 »                                              101 ↲
```

See the `Result` type in the error: ignoring the full namespaces, the type of the variable file is actually `Result<File, Error>`. We need that `File` type in order to read the contents of the file. As an aside, looking at the official documentation of this method may be a source of some confusion. Here's how it is documented:

```
fn open<P: AsRef<Path>>(path: P) -> Result<File>
```

`Result` looks like it's missing the second generic type, but it's merely hidden away by a type alias. This is not the `Result` type we have seen before, but a `Result` type specific to IO operations. It is defined in the `std::io::Result` module:

```
type Result<T> = Result<T, std::io::Error>;
```

The reason for this is conciseness: every IO operation uses that same error type, so this type alias saves developers from repeating it everywhere.

But back to what we were doing, we can use the `match` statement to get `File` from inside the `Result` type:

```
    let mut file = match File::open(&path) {
        Ok(file) => file,
        Err(err) => {
            println!("Error while opening file: {}", err);
            panic!();
        }
    };
```

So, we made two changes. First, we made the file variable mutable. Why? Because the function signature of `read_to_string` is as follows:

```
fn read_to_string(&mut self, buf: &mut String) -> Result<usize>
```

`&mut self` means that the variable we are calling this method on needs to be mutable because reading the file changes internal pointers of the file handle.

Secondly, we handled both the `Ok` case and the `Err` case by returning the actual file handle if everything was OK, and displaying an error and bailing out if not.

With this change, the program compiles and we can run it:

```
vegai@carbon ~/rustbook/5 » rustc result-2.rs
vegai@carbon ~/rustbook/5 » ./result-2
Error while opening file: No such file or directory (os error 2)
thread 'main' panicked at 'explicit panic', result-2.rs:11
note: Run with `RUST_BACKTRACE=1` for a backtrace.
vegai@carbon ~/rustbook/5 »                                     101 ↲
```

Panicking is always a tad ugly but works well for things that you would never expect to happen. Let's do something about that warning, though: warnings are always a sign of poor code quality, and we'll have none of that. The warning is there because `File::read_to_string` (which is a part of the implementation of the `Read` trait) returns a value whose type is Result<usize>. The value signifies how many bytes were read into String.

We have two ways of handling this warning:

- Handle both the `Ok` and `Err` cases like before
- Assign it to a special variable, essentially telling the compiler that we don't care about the return value

Since we already did the first one, and since it suits this purpose better anyway, let's do the second. The `read_to_string` line becomes as follows:

```
let _ = file.read_to_string(&mut s);
```

With that change, the code compiles without warnings.

Unwrapping

Writing the same `match` statements over and over again quickly becomes boilerplate code, which can make it nastier to read. The standard library contains a couple of helper methods that both `Result` and `Option` types implement. You can use them to simplify error cases where you really do not expect things to fail.

The methods are as follows:

- `unwrap(self): T` expects self to be `Ok`/`Some` and returns the value contained within. If it's `Err` or `None` instead, it raises a panic with the contents of the error displayed.
- `expect(self, msg: &str): T` behaves like unwrap, except that it outputs a custom message before panicking in addition to the contents of the error.
- `unwrap_or(self, opt_b: T): T` behaves like unwrap, except that instead of panicking, it returns `opt_b`.
- `unwrap_or_else(self, op: F): T` behaves like unwrap, except that instead of panicking, it calls `op`, which needs to be a function or a closure: more precisely, anything that implements the `FnOnce` trait.

Here's the previous code example that used `match` statements converted to using unwrapping:

```
// result-unwrapping.rs
use std::io::Read;
use std::path::Path;
use std::fs::File;

fn main() {
    let path = Path::new("data.txt");
    let mut file = File::open(&path)
        .expect("Error while opening data.txt");

    let mut s = String::new();
    file.read_to_string(&mut s)
        .expect("Error while reading file contents");

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

As you see, the error-handling code becomes dramatically nicer. Of course, this method should only be used when the error is so critical that panicking is a good option.

Mapping of the Option/Result values

The `map` and `map_err` methods provide a way to concisely apply mapping functions on the contents of the values inside `Ok`/`Some` and `Err` values. Since doing anything with `None` values would be pointless, `map_err` is not defined for `Option`. Here are the full types of these methods:

```
map<U, F>(self, f: F) -> Result<U, E>
   where F: FnOnce(T) -> U
map<U, F>(self, f: F) -> Option<U>
   where F: FnOnce(T) -> U

map_err<F, O>(self, f: O) -> Result<T, F>
   where O: FnOnce(E) -> F
```

Reading through the types carefully, we see that the `map` functions for both `Result` and `Option` types take a function that transforms a value of type `T` to a value of type `U`, that is, the `FnOnce` declaration. The return type tells us that the new value of type `U` is wrapped inside the returned `Result` or `Option`. In the case of `Result`, the error type is left untouched. For the `map_err` method, that is, vice versa, of course, the `Ok` type is left untouched, and the error type is mapped via the `f` function.

So, where would these methods be a good fit? An obvious place would be your own library method, where you'd like to make some modification to the `Ok`/`Some` value but propagate any possible `Err` or `None` values upwards to the caller. An example should make this clearer. We'll write two functions that take `bytestring`, try to convert it to `String`, and then convert the string to uppercase. As you might remember, the conversion might fail:

```
// mapping.rs
use std::string::FromUtf8Error;

fn bytestring_to_string_with_match(str: Vec<u8>) -> Result<String, FromUtf8Error> {
    match String::from_utf8(str) {
        Ok(str) => Ok(str.to_uppercase()),
        Err(err) => Err(err)
    }
}

fn bytestring_to_string(str: Vec<u8>) -> Result<String, FromUtf8Error> {
    String::from_utf8(str).map(|s| s.to_uppercase())
}

fn main() {
    let faulty_bytestring = vec!(130, 131, 132, 133);
    let ok_bytestring = vec!(80, 82, 84, 85, 86);

    let s1_faulty = bytestring_to_string_with_match(faulty_bytestring.clone());
    let s1_ok = bytestring_to_string_with_match(ok_bytestring.clone());

    let s2_faulty = bytestring_to_string(faulty_bytestring.clone());
    let s2_ok = bytestring_to_string(ok_bytestring.clone());

    println!("Read the string: {:?}", s1_faulty);
    println!("Read the string: {:?}", s1_ok);
    println!("Read the string: {:?}", s2_faulty);
    println!("Read the string: {:?}", s2_ok);
}
```

The two functions are functionally identical, so their output is the same as well. Here's the output:

```
vegai@carbon ~/rustbook/5 » ./mapping
Read the string: Err(FromUtf8Error { bytes: [130, 131, 132, 133], error: Utf8Err
or { valid_up_to: 0 } })
Read the string: Ok("PRTUV")
Read the string: Err(FromUtf8Error { bytes: [130, 131, 132, 133], error: Utf8Err
or { valid_up_to: 0 } })
Read the string: Ok("PRTUV")
vegai@carbon ~/rustbook/5 »
```

Early returns and the try! macro

Here's another pattern for error handling: returning early from a function when an error occurs in any operation. We'll modify the earlier code that converts `bytestring` to `String` into this pattern:

```
fn bytestring_to_string_with_match(str: Vec<u8>) -> Result<String, FromUtf8Error> {
    let ret = match String::from_utf8(str) {
        Ok(str) => str.to_uppercase(),
        Err(err) => return Err(err)
    };

    println!("Conversion succeeded: {}", ret);
    Ok(ret)
}
```

The `try!` macro abstracts this pattern, making it possible to write this in a more concise way:

```
fn bytestring_to_string_with_try(str: Vec<u8>) -> Result<String, FromUtf8Error> {
    let ret = try!(String::from_utf8(str));
    println!("Conversion succeeded: {}", ret);
    Ok(ret)
}
```

The `try!` macro has a caveat that may be non-obvious if you forget that it expands to an early return-since it might return a `Result` or an `Option` type, it cannot be used inside the main function at all. This is because the signature of the `main` function is simply this:

```
fn main()
```

It does not take any parameters and returns nothing, so it cannot return either an `Option` or a `Result` type. Here's a simple program to show you exactly what the compiler would think of this:

```
// try-main.rs
fn main() {
    let empty_ok_value = Ok(());
    try!(empty_ok_value);
}
```

The compiler outputs the following error when trying to build this:

```
vegai@carbon ~/rustbook/5 » rustc try-main.rs
error[E0308]: mismatched types
 --> try-main.rs:3:5
  |
3 |     try!(empty_ok_value);
  |     ^^^^^^^^^^^^^^^^^^^^^ expected (), found enum `std::result::Result`
  |
  = note: expected type `()`
             found type `std::result::Result<_, _>`
  = help: here are some functions which might fulfill your needs:
          - .unwrap()
          - .unwrap_err()
          - .unwrap_or_default()
  = note: this error originates in a macro outside of the current crate

error: aborting due to previous error

vegai@carbon ~/rustbook/5 »                                                101 ↲
```

The compilation errors that come from macros are always a bit hard to read because they have to stem from the generated code, and that is not something you, the coder, wrote.

The ? operator

A shorter way of writing `try!` macros is available with the `?` operator. With that, the preceding function can become even tidier:

```
// try.rs
fn bytestring_to_string_with_qmark(str: Vec<u8>) -> Result<String, FromUtf8Error> {
    let ret = String::from_utf8(str)?;
    println!("Conversion succeeded: {}", ret);
    Ok(ret)
}
```

This operator becomes even nicer if you have a combined statement of several operations, where a failure in each operator should mean a failure of the whole. For instance, we could merge the whole by opening a file, reading a file, and converting it to uppercase into a single line:

```
File::create("foo.txt")?.write_all(b"Hello world!")
```

This operator got into the stable release in version 1.13. In its current form, it works pretty much as a replacement for the `try!` macro, but there are some plans on making it more generic and usable for other cases too. If you're interested in the progress, the public RFC discussion can be found at https://github.com/rust-lang/rfcs/issues/1718.

Panicking

Even though Rust does not have an exception mechanism designed for general error-handling usage, panicking is not far from it. Panics are non-recoverable errors that crash your thread. If the current thread is the main one, then the whole program crashes.

On a more technical level, panicking happens in the same process as exceptions: unwinding the call stack. This means climbing up and out of the place in the code where the panicking happened till hitting the top, at which point, the thread in question aborts. Here's an example where we have two call stacks:

- `f1` spawns a new thread and calls `f2`, which calls `f3`, which panics
- `main` calls `f2`, which calls `f3`, which panics

Take a look at the following code snippet:

```
// panic.rs
use std::thread;

fn f1() -> thread::JoinHandle<()> {
    thread::spawn(move || {
        f2();
    })
}

fn f2() {
    f3();
}

fn f3() {
    panic!("Panicking in f3!");
}

fn main() {
    let child = f1();
    child.join().ok();

    f2();
    println!("This is unreachable code");
}
```

Here's how it looks on runtime:

```
vegai@carbon ~/rustbook/5 » ./panic
thread '<unnamed>' panicked at 'Panicking in f3!', panic.rs:14
note: Run with `RUST_BACKTRACE=1` for a backtrace.
thread 'main' panicked at 'Panicking in f3!', panic.rs:14
vegai@carbon ~/rustbook/5 »                                      101 ↲
```

Here, you can see that even though the child thread panicked, the main thread got to its own panic. Even though it's generally more recommended to handle errors properly via the `Option`/`Result` mechanism, you can use this method to handle fatal errors in worker threads; let the workers die, and restart them.

If you need more control on how your panics are handled, you can stop the unwinding at any point with the `std::panic::catch_unwind` function. As mentioned often before, `panic`/`catch_unwind` is not the recommended general error-handling method for Rust program, using the `Option`/`Result` return values is. The `catch_unwind` function takes a closure and handles any panics that happen inside it. Here's its type signature:

```
fn catch_unwind<F: FnOnce() -> R + UnwindSafe, R>(f: F) -> Result<R>
```

As you can see, the return value of `catch_unwind` has an additional constraint, `UnwindSafe`. It means that the variables in the closure are exception safe; most types are, but notable exceptions are mutable references (`&mut T`). There'll be more about those and their restrictions in the future chapters.

Here's an example of `catch_unwind`:

```
// catch-unwind.rs
use std::panic;

fn main() {
    panic::catch_unwind(|| {
        panic!("Panicking!");

    }).ok();
    println!("Survived that panic.");
}
```

And here's how it runs:

```
vegai@carbon ~/rustbook/5 » ./catch_unwind
thread 'main' panicked at 'Panicking!', catch_unwind.rs:5
note: Run with `RUST_BACKTRACE=1` for a backtrace.
Survived that panic.
vegai@carbon ~/rustbook/5 »
```

As you can see, `catch_unwind` does not prevent the panicking from happening; it just stops the unwinding and, thus, does not stop the thread. Note again that `catch_unwind` is not the recommended way of error management in Rust. It is not guaranteed to catch all panics. Double panicking may occur in certain situations, in which case the whole program would abort. Also, there's a compiler option which turns all panics into aborts, and aborts are not catchable by this method.

Custom errors and the Error trait

Quite often, we wish to separate the errors our programs might make from every other one. This is typically done in other languages by creating a new subclass of some exception based class, and possibly overriding some of the parent's methods.

Rust's approach is similar, but since we don't have classes or objects really, we use traits and implementations. Here's the `Error` trait from the standard library:

```
pub trait Error: Debug + Display + Reflect {
    fn description(&self) -> &str;
    fn cause(&self) -> Option<&Error> { None }
}
```

So, the new error type we're about to write requires these two methods. The `description` method returns a string slice reference, which tells in free form what the error is about. The `cause` method returns an optional reference to another `Error` trait, representing a possible lower-level reason for the error. Thus, the highest level `Error` trait has access to the lowest level, making a precise logging of the error possible.

Let's take an HTTP query as an example of a cause chain. We call `get` on a library that does the actual query. The query might fail due to a lot of different reasons:

- The DNS query might fail because of networking failures or because of a wrong address
- The actual transfer of data might fail
- The data might be received correctly, but there could be something wrong with the received HTTP headers, and so on and so forth

If it were the first case, we might imagine three levels of errors, chained together by the cause fields:

- The UDP connection failing due to the network being down (`cause=None`)
- The DNS lookup failing due to a UDP connection failure (`cause=UDPError`)
- The `get` query failing due to a DNS lookup failure (`cause=DNSError`)

In addition to requiring these two methods, the `Error` trait depends on the `Debug` and `Display` traits (`Reflect` can be ignored in this case), which means that any new error type needs to implement (or derive) those three as well.

Let's model money in a more-or-less arbitrary fashion by having it as simply a pairing of a currency and an amount. Currency will be an `enum` of either USD or EUR, and we'll have new methods for transforming arbitrary strings into moneys or currencies. The potential error here is in that conversion phase. Let's first look at the boilerplate, the structs and dummy implementations, before adding custom error types:

```
// custom-error-1.rs
#[derive(Debug)]
```

```
enum Currency { USD, EUR }

#[derive(Debug)]
struct CurrencyError;

impl Currency {
    fn new(currency: &str) -> Result<Self, CurrencyError> {
        match currency {
            "USD" => Ok(Currency::USD),
            "EUR" => Ok(Currency::EUR),
            _ => Err(CurrencyError{})
        }
    }
}

#[derive(Debug)]
struct Money {
    currency: Currency,
    amount: u64
}

#[derive(Debug)]
struct MoneyError;

impl Money {
    fn new(currency: &str, amount: u64) -> Result<Self, MoneyError> {
        let currency = match Currency::new(currency) {
            Ok(c) => c,
            Err(_) => panic!("Unimplemented!")
        };

        Ok(Money {
            currency: currency,
            amount: amount
        })
    }
}

fn main() {
    let money_1 = Money::new("EUR", 12345);
    let money_2 = Money::new("FIM", 600000);

    println!("Money_1 is {:?}", money_1);
    println!("Money_2 is {:?}", money_2);
}
```

You'll see that we have our dummy `Error` structs in place already, but they do not yet implement the `Error` traits. Let's see how that's done. Before anything else, we'll need a few additional `use` statements:

```
// custom-error-2.rs
use std::error::Error;
use std::fmt;
use std::fmt::Display;
```

Now, we can go forth and add a description field to `CurrencyError`, and implement `Display` and `Error` for it:

```
// custom-error-2.rs
#[derive(Debug)]
struct CurrencyError {
    description: String
}
impl Display for CurrencyError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "CurrencyError: {}", self.description)
    }
}

impl Error for CurrencyError {
```

```
    fn description(&self) -> &str {
        "CurrencyError"
    }
}
```

As `CurrencyError` does not have any lower-lever cause for the error, the default implementation that returns `None` for `cause` is fine. The `description` method is not terribly interesting either, but the details of the error are given by the `Display` implementation. We'll just need to change the last pattern match from `Currency::new` method to support this:

```
            _ => Err(CurrencyError{ description: format!("{} not a valid currency", currency)})
```

Next up, we'll augment `MoneyError`:

```
   // custom-error-2.rs
#[derive(Debug)]
struct MoneyError {
    cause: CurrencyError
}

impl Display for MoneyError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "MoneyError due to {}", self.cause)
    }
}

impl Error for MoneyError {
    fn description(&self) -> &str {
        "MoneyError"
    }
    fn cause(&self) -> Option<&Error> {
        Some(&self.cause)
    }
}
```

We're now holding the lower-level error in the cause field of `MoneyError`. Since the only known lower-level error in this case is `CurrencyError`, it is a concrete type. If there were more options, this could be an `enum` that encapsulates all the different possible errors. The new method of the `Money` class that used to panic in an error case can now become as follows:

```
    Err(e) => return Err(MoneyError { cause: e })
```

And there we are! Now we can see what caused the error by displaying `MoneyError` in the main function:

```
    let cause_for_money_2 = money_2.unwrap_err();
    println!("{}", cause_for_money_2);
```

Running the program now shows the error in both the debug formatted version, and in the code we just added, which outputs via the `Display` trait:

```
vegai@carbon ~/rustbook/5 » ./custom-error-2
Money_1 is Ok(Money { currency: EUR, amount: 12345 })
Money_2 is Err(MoneyError { cause: CurrencyError { description: "FIM not a valid curren
cy" } })
MoneyError due to CurrencyError: FIM not a valid currency
vegai@carbon ~/rustbook/5 »
```

The moral of this journey: Rust has a fine framework for defining your custom error types. Especially if you're writing your own libraries, you should define your own error types to make debugging easier.

Exercise

Let's go back to the game project. We'll add operations for the entities to move around the map, with all sorts of expected and unexpected error handling that might happen. Here are the data structures that we ended up with in Chapter 3, *Unit Testing and Benchmarking*:

```
#[derive(PartialEq, Debug)]
enum TerrainGround {
    Soil,
    Stone
}

#[derive(PartialEq, Debug)]
enum TerrainBlock {
    Tree,
    Soil,
    Stone
}

#[derive(PartialEq, Debug)]
enum Being {
    Orc,
    Human
}

struct Square {
    ground: TerrainGround,
    block: Option<TerrainBlock>,
    beings: Option<Being>
}

struct Grid {
    size: (usize, usize),
    squares: Vec<Square>
}
```

So, we'll want to make it possible to move `Being` in any direction on `Grid` with the following cases being errors:

- There is no `Being` in `Square`
- `Being` tries to fall off from the edge of `Grid`
- `Being` tries to move into `Square` where there is already `Being`
- `Being` tries to move to `Terrain`, which is `Stone`

Here's the first one as an example and you can fill in the rest. We'll implement the `move_being` method for the `Grid` struct, since that's the only one that has all the required information for this operation. The aforementioned structs are omitted:

```
enum Direction {
    West,
    East,
    North,
    South
}

#[derive(Debug, PartialEq)]
enum MovementError {
    NoBeingInSquare
}
```

```
impl Grid {
    fn move_being_in_coord(&self, coord: (usize, usize), dir: Direction) -> Result<(usize, usize), MovementErro
        let square = self.squares.get(coord.0 * self.size.0 + coord.1).expect("Index out of bounds trying to ge
        match square.being {
            Some(_) => Ok((0,0)), // XXX: fill in the implementations here
            None => Err(MovementError::NoBeingInSquare)
        }
    }

    fn generate_empty(size_x: usize, size_y: usize) -> Grid {
        let number_of_squares = size_x * size_y;
        let mut squares: Vec<Square> = Vec::with_capacity(number_of_squares);

        for _ in 0..number_of_squares {
            squares.push(Square{ground: TerrainGround::Soil, block: None, being: None});
        }

        Grid {
            size: (size_x, size_y),
            squares: squares
        }
    }

}

#[cfg(test)]
mod tests {
    #[test]
    fn test_empty_grid() {
        let grid = ::Grid::generate_empty(5, 13);
        assert_eq!(grid.size, (5, 13));
        let mut number_of_squares = 0;

        for square in &grid.squares {
          assert_eq!(square.ground, ::TerrainGround::Soil);
          assert_eq!(square.block, None);
          assert_eq!(square.being, None);
          number_of_squares += 1;
        }

        assert_eq!(grid.squares.len(), 5*13);
        assert_eq!(number_of_squares, 5*13);
    }

    #[test]
    fn test_move_being_without_being_in_square() {
        let grid = ::Grid::generate_empty(3, 3);
        assert_eq!(grid.move_being_in_coord((0, 0), ::Direction::West), Err(::MovementError::NoBeingInSquare));
    }
}
```

There we go. Now your part:

1. Implement the error case where `Being` tries to fall off from the edge of `Grid`.
2. Implement the error case where `Being` tries to move into a `Square` where there is already a `Being`.
3. Implement the error case where `Being` tries to move to `Terrain`, which is `Stone`.
4. Implement the happy case where no errors happen and `Being` successfully moves to the new `Square`.
5. Make `MovementError` implement the `Error` trait.

Summary

Here's what you should remember from this chapter:

- Error handling in Rust is explicit: operations that could fail have a two-part return value via the `Result` or `Option` generic types
- You must handle errors in some way, either by deconstructing the `Result`/`Option` values by a `match` statement, or by using the helper methods or macros
- You should usually `opt` for handling errors properly, that is, not resorting to operations that panic on failures
- Use the unwrapping methods or panicking when errors would be programming errors or so fatal that recovery would be impossible
- Panics are mostly non-recoverable, which means that they crash your thread
- Use the standard `Error` trait for your own error types

The next chapter will be about Rust's somewhat unique memory handling, including references, borrowing, and lifetimes.

Memory, Lifetimes, and Borrowing

Rust makes you, the developer, handle memory yourself. It helps you along the way, however, by having abstractions and language support for memory allocations. Its system of lifetimes, ownership, and borrowing may be familiar to you as concepts from the C++ world. Rust has all of these, not just as concepts but in the language along with compile-time checking, making this most difficult class of runtime problems an easier compile-time problem.

This chapter goes into the details of memory management in Rust. We give a short introduction to LLVM, the compiler framework that the Rust compiler uses, and its Intermediate Representation code.

The topics covered in this chapter are:

- LLVM
- Function variables, stack
- Heap allocations
- Moving, copying, and cloning
- Ownership
- Borrows
- Lifetimes
- Generic types box, `Cell`, `RefCell`, `Rc`

LLVM

Rust's compiler is based on LLVM, a compiler framework that allows easier and more robust writing of compilers. In its core is a language called **IR**, short for **Intermediate Representation**. It is sort of a middle ground between an actual programming language and a machine-specific assembler language.

Implementing a compiler for a new language with LLVM means writing a new frontend for your language: a program that takes in a program written in the new language and outputs LLVM IR codes. LLVM itself contains backends for several target architectures, which means that a developer of a new language will get more things for free.

The IR is not completely independent of the target machine, just less so. New frontends do have to make some choices concerning the target architecture, just far less than if they had written machine code backends.

Let's take a quick look at what the IR code looks like. Here's an addition function:

```
// add-1.ll
define i32 @add_unsigned(i32 %a, i32 %b) {
  %1 = add i32 %a, %b
  ret i32 %1
}
```

The values are typed and need to be repeated often. The `add_unsigned` function returns an `i32` and takes two `i32` parameters, `%a` and `%b`. It then calls the internal `add` function for those parameters and stores the answer in register `%1`. Then the value in that register is returned.

Next, we'll see what kind of assembler code this can be turned into. You'll need to install `llvm` locally if you want to run these as well. The compiler that turns IR into an assembler is called LLVM static compiler and its binary is usually `llc`. If the preceding code was in `add.ll`, we would run the following:

```
llc -march=x86_64 add.ll
```

The output of the command is saved in `add.s`. The assembler code is target-specific; this is what it looks like on Linux:

```
   .text
   .file "add.ll"
   .globl      add_unsigned
   .align      16, 0x90
   .type add_unsigned,@function
add_unsigned:                          # @add_unsigned
   .cfi_startproc
# BB#0:
   leal  (%rdi,%rsi), %eax
   retq
.Lfunc_end0:
   .size add_unsigned, .Lfunc_end0-add_unsigned
   .cfi_endproc
```

```
    .section    ".note.GNU-stack","",@progbits
```

Lots of boilerplate, but the actual function consists of the `leal` and `retq` instructions. We can also verify from this piece of IR code that we can generate assembler code for 32-bit x86 and ARM as well. Just change the `-march` parameter to either x86 or arm, and the function code becomes this for x86:

```
    movl  4(%esp), %eax
    addl  8(%esp), %eax
    retl
```

And this for ARM:

```
    add   r0, r0, r1
    mov   pc, lr
```

Getting LLVM IR output and the corresponding assembler output from a piece of Rust code is not much harder, although the output will contain much more boilerplate. Here's a simple piece of code in Rust:

```
// add-2.rs
fn add_one(a: u8) -> u8 {
   let g = a + 255;
   g
}

fn main() {
    let x = 1;
    let z = add_one(x);
    let _ = z;
}
```

To get IR from this piece of code, use the `--emit` parameter of `rustc`:

```
    rustc --emit=llvm-ir add-2.rs
```

It's important at this point to not optimize the code, since our program is so simple that it would probably be completely optimized away. The unfortunate part is that the resulting code is rather large and is full of uninteresting bits. Let's take a look at a few interesting ones, though. If you don't understand these fully, don't worry, just gloss through them briefly to get a bit more familiar with the syntax. First, locate the entry point, `main`:

```
// add-2.ll
define i64 @main(i64, i8**) unnamed_addr {
top:
  %2 = call i64 @_ZN3std2rt10lang_start17h162055cb2e4b9fe7E(i8* bitcast (void ()* @_ZN4llvm4main17ha9d0e54b0b6f
  ret i64 %2
}
```

Rust compiler mangles all the function names, but we can see the function name inside the mangled parts. For instance:

```
@_ZN4llvm4main17ha9d0e54b0b6fe32aE
```

The `main` function looks something like this:

```
define internal void @_ZN4llvm4main17ha9d0e54b0b6fe32aE() unnamed_addr #0 {
entry-block:
  %x = alloca i8
  %z = alloca i8
  store i8 1, i8* %x
  %0 = load i8, i8* %x
  %1 = call i8 @_ZN4llvm7add_one17h86509496e3ccd7f0E(i8 %0)
  store i8 %1, i8* %z
  ret void
}
```

Here we can see that there are two stack allocations (the `alloca` instructions). Next, assign the values in pretty much the same order as they were in the corresponding Rust code. Lastly, let's take a look at the `add_one` function:

```
// add-2.ll
define internal i8 @_ZN4llvm7add_one17h86509496e3ccd7f0E(i8) unnamed_addr #0 {
entry-block:
  %a = alloca i8
  %g = alloca i8
  store i8 %0, i8* %a
  %1 = load i8, i8* %a
  %2 = call { i8, i1 } @llvm.uadd.with.overflow.i8(i8 %1, i8 1)
  %3 = extractvalue { i8, i1 } %2, 0
  %4 = extractvalue { i8, i1 } %2, 1
  %5 = icmp eq i1 %4, true
  %6 = call i1 @llvm.expect.i1(i1 %5, i1 false)
  br i1 %6, label %cond, label %next

next:                                             ; preds = %entry-block
  store i8 %3, i8* %g
  %7 = load i8, i8* %g
  ret i8 %7

cond:                                             ; preds = %entry-block
  call void @_ZN4core9panicking5panic17heeca72c448510af4E({ %str_slice, %str_slice, i32 }* noalias readonly der
  unreachable
}
```

Here Rust generated quite a lot of extra code for the simple addition. The reason is mostly the overflow check: Rust checks integer overflows at runtime, so the corresponding code must be there. The LLVM internal function, `uadd.with.overflow`, returns two values and the second one is true if the calculation flowed over.

In the following sections, we won't be looking at LLVM anymore, but feel free to take a look every now and then what the generated code will look like. It might be a slightly complicated way to see how the programs behave, but nothing beats verifying things from actual compiler output when you're interested in the details.

Here are a few exercises that you can do:

1. Use rustc `-O` to generate optimized LLVM IR code. What happened to your code?
2. Make a new String value in `main` and see what kind of IR code gets generated.
3. Add a `println!` macro to your code. How did it affect the IR code?

Function variables and the stack

Rust's memory management hangs on to two concepts: the stack and the heap. Stacks are used for local variables: all the let bindings in your functions are stored in the stack, either as the values themselves or as references to other things. It is an extremely fast and reliable memory allocation scheme. It is fast because allocating and deallocating memory via a stack requires just one CPU instruction: moving the stack frame pointer. It is reliable because of its simplicity: when a function is finished, all its stack memory is released by restoring the stack frame pointer to where it was before entering the function. This makes the stack less versatile, however: there's no way for a thing in the stack to outlive its block.

Understanding the implications of the stack is not necessary in day-to-day work with Rust, but it provides a good foundation for the two different memory allocation schemes.

Here's the first example piece of code to illustrate how the stack works:

```
// stack-1.rs
fn f2(y: u8) -> u8 {
    let x = 2 + y;
    return x;
}

fn f1(x: u8) -> u8 {
    let z = f2(5);
    return z+x;
}

fn main() {
    println!("f1(9) is {}", f1(9));
}
```

OK, so what happens in the stack when this program runs? Glossing over the details of `println!` and focusing on how the stack lives, it goes something like this:

1. The `main` function calls `f1` with the parameter `9`, which goes on the stack. Stack is now `[9]`.
2. We enter `f1`, memory is reserved and zeroed for the `z` binding from the stack. Stack is now `[9, 0]`.
3. We call `f2` with the parameter `5`, that goes on the stack - it's now `[9, 0, 5]`.
4. We enter `f2`, memory is reserved for the `x` binding from the stack. It gets assigned `2+5`. Stack is now `[9, 0, 5, 7]`.
5. `f2` ends, returns the value `7`, and its stack frame (containing `[5, 7]`) is released. Stack is now `[9, 0]`.
6. Back at `f1`, `7` gets assigned to `z`. Stack is now `[9, 7]`.
7. `f1` ends, returns the value `9+7` and releases its stack frame. Stack is now empty.

To verify that, here's the output from running this program:

```
vegai@carbon ~/rustbook/6 » ./stack-1
f1(9) is 16
vegai@carbon ~/rustbook/6 »
```

You might expect this piece of code to not actually work given that we just said that things in the stack cannot outlive their block. You might, especially if you're used to higher level languages, interpret the `f2` function as actually returning the `x` variable from inside the function. Instead, it's actually just returning a copy of the value, and that is fine because a copy of the `x` variable is not the same as the `x` variable. Specifically, this works because the number types implement the `Copy` trait. More about that soon.

However, if we explicitly say that we want to return a reference to the actual variable, we run into all kinds of trouble. The ampersand character is used for references, so our naive attempt to do this might look like this:

```
// stack-2.rs
fn f1() -> &u8 {
    let x = 4;
    return &x;
}

fn main() {
    let f1s_x = f1();
    println!("f1 returned {:?}", f1s_x);
}
```

And here's the compiler's reply:

```
vegai@carbon ~/rustbook/6 » rustc stack-2.rs
error[E0106]: missing lifetime specifier
 --> stack-2.rs:1:12
  |
1 | fn f1() -> &u8 {
  |            ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no value
 for it to be borrowed from
  = help: consider giving it a 'static lifetime

error: aborting due to previous error

vegai@carbon ~/rustbook/6 »                                                      101 ↲
```

OK, let's do what the compiler suggests and add the static lifetime to the `return` parameter:

```
fn f1() → &'static u8
```

Now we get the more interesting error we were looking for:

```
vegai@carbon ~/rustbook/6 » rustc stack-3.rs
error: `x` does not live long enough
 --> stack-3.rs:3:13
  |
3 |     return &x;
  |             ^ does not live long enough
4 | }
  | - borrowed value only lives until here
  |
  = note: borrowed value must be valid for the static lifetime...

error: aborting due to previous error

vegai@carbon ~/rustbook/6 »                                          101 ↲
```

There we go! The compiler is confirming that local variables, those allocated in the stack, should not be returned from a function because they do not exist after the function call. There's no way this could work so Rust does not allow it.

The previous piece of code is roughly equivalent to this code in C:

```
/* stack-abuse.c */
#include <stdio.h>

int* f1() {
    int x = 4;
    return &x;
}

int main() {
    int *f1s_x = f1();
    printf("f1 returned %d", *f1s_x);
}
```

Even though it is distasteful, this code is valid C. It compiles with a warning and crashes to a segmentation fault at runtime:

```
vegai@carbon ~/rustbook/6 » gcc -o stack-abuse stack-abuse.c
stack-abuse.c: In function 'f1':
stack-abuse.c:5:12: warning: function returns address of local variable [-Wreturn-loca
l-addr]
     return &x;
            ^~
vegai@carbon ~/rustbook/6 » ./stack-abuse
[1]    8463 segmentation fault (core dumped)  ./stack-abuse
vegai@carbon ~/rustbook/6 »                                          139 ↲
```

Memory access problems in real C programs are not this flagrant, of course, but this works well to illustrate a certain difference in these two otherwise similar languages: C allows risky code, mitigating the risks by warnings generated by compiler heuristics. Rust has a robust system of lifetimes, which makes the risks more contained.

While the stack is simple and powerful, we obviously need also longer-living variables:

1. Take your favorite programming language, in case it's not Rust yet. Try to find out whether it does stack allocations and if there's any way you can control it.
2. Each process has a limited stack size, enforced by the operating system. The size varies over different systems, in Linux it's usually about 8MB. Imagine a few ways in which you could cause that limit to break.

The heap

The heap is for the more complicated and versatile memory allocation schemes. Values in the heap live more dynamically. Memory in the heap is allocated at some point of the program, released at some other point, and there does not have to be a strict boundary between these points like with the stack. In other words, values in the heap may live beyond a function where it was allocated but values in the stack may not.

Note that there is a tree-like data structure that is called the heap, but the heap related to programming language implementations is not the same. Rather, the heap we're talking about now is just a general term for a dynamically allocated pool of memory used in programming languages, and its design can vary.

Rust's heap is provided by either a memory allocator called **jemalloc**, which gives us good linear thread scalability, or by the system's own allocator. For instance, on Linux this would usually be the glibc's malloc.

The jemalloc allocator gets used by default when making binary builds, whereas the system allocator is the default when making library builds. The reason for these defaults is that when building binaries, the compiler has control of the whole program, so it does not have to consider external entities and can choose the more efficient jemalloc. A library, on the other hand, may be used in different circumstances that are not known when building the library, so the choice of using the system allocator is safer.

This distinction is usually not an important one, but in case you need to override these defaults, it is possible by using a feature tag and linking in a specific crate. In code, the top of your module would need to look as follows:

```
#![feature(alloc_system)]

extern crate alloc_system;
```

That would force the usage of the system allocator. To force jemalloc, you would say the following:

```
#![feature(alloc_jemalloc)]
#![crate_type = "dylib"]

extern crate alloc_jemalloc;
```

Every time you get a value that's not a primitive value, you get a heap allocation. For instance:

```
let s = String::new("foo")
```

`String::new` will allocate the string in the heap and return a reference to it. That reference goes into the variable `s`, which is allocated in the stack. The string in the heap lives for as long as it needs to: when `s` goes out of scope, the string does as well and it is then dropped.

If you need to allocate primitive values in the heap for some reason, there's a generic type `Box<T>` that does just that. It'll be covered a bit later.

Memory safety

In many modern languages the usages of stack and heap are abstracted away from the programmer: you declare and use the variables in your code and they are allocated based on the usage patterns. Usually the allocation happens in the heap, and some form of runtime garbage collection takes care of the deallocations. The end result is easy memory safety, but with a runtime cost: the allocation decisions happen automatically and may not always be optimal for your program.

In contrast, a low-level systems programming language such as C does nothing to hide these details from the programmer, and provides nearly no safety. A programmer can easily create hard-to-debug errors by allocating and deallocating things in the wrong order, or forgetting to deallocate. Such bugs lead to memory leaks, hard crashes in the form of segmentation faults, or in the worst case, security vulnerabilities. The upside is that an expert C programmer can be absolutely certain how memory is managed in the program and is thus free to create optimal solutions.

Here's a simple stack overflow in C:

```
// stack-overflow.c
int main() {
  char buf[3];
  buf[0] = 'a';
  buf[1] = 'b';
  buf[2] = 'c';
  buf[3] = 'd';
}
```

This compiles fine and even runs without errors, but the last assignment goes over the allocated buffer. Errors such as this happen in actual code in less obvious ways and frequently cause security problems.

Modern C++ safeguards against some of the problems associated with manual memory management by providing smart pointer types, but this does not completely eliminate them. Also, some virtual machines (Java's being the most prominent example) have several decades of work in them to make garbage collection highly efficient, giving more than fair performance for most workloads.

Rust's exceptional memory safety stands on three pillars:

1. No null pointers: `Option<T>` can be used safely if something might be nothing.
2. Optional library support for any kind of garbage collection.
3. Ownership, borrowing, and lifetimes: compile-time verification for almost all memory usage.

Firstly, null pointers are mournfully referred to as the "billion dollar mistake" by Tony Hoare, who implemented them the first time in 1965. The problem is not the null pointer per se, but the way they are implemented typically: any object may be assigned null, but those objects can be used without checking for null. Most programmers do not bother to check all usages, especially when it strongly looks like something cannot be null. Rust's `Option<T>` allows for null values, but makes the choice explicit and does not allow ignoring the null value.

Here's a silly simulation to show how it might work. Imagine a piece of code in Python, where an operation will succeed 99% of the time and return an object. The remaining 1% we just forget to check against and let the code fall through:

```
# meep.py
from random import random

class Meep:
    def exclaim(self):
        print("Holla!")

def probablyMakeMeep():
    if random() > 0.1:
        return Meep()
    # implicitly returns None

while True:
    meep = probablyMakeMeep()
    meep.exclaim()
```

These sorts of bugs are everywhere in programs written in languages that have unchecked null pointers. In Rust the same problem is also possible, but you will have to write explicit code saying that you don't care about the nulls by using the unwrapping methods of Option and Result types that we saw earlier.

Primitive garbage collection through reference counting is already in the standard library via the Rc and Arc ("A" referring to atomic, which means thread-safe) generic types. Support for advanced garbage collection (the `Gc<T>` type) is in the planning stage and may arrive at some point in the future.

The third point is the core of this whole chapter. Ownership, lifetimes, and borrowing gives us compile-time checked memory safety with zero runtime cost, without requiring garbage collection. We'll talk about each of these in the next three sections.

Ownership

When you use the `let` keyword, you create a temporary variable binding. Those bindings will own the things they bind. When the binding goes out of scope, the binding itself and whatever it points to gets freed. Going out of scope happens when a block ends: when a `{` gets closed by a `}`.

Here's an example:

```
// blocks.rs
fn main() {
    let level_0_str = String::from("foo");
    {
        let level_1_number = 9;
        {
            let level_2_vector = vec![1, 2, 3];
        } // level_2_vector goes out of scope here

        {
            let level_2_number = 9;
        } // level_2_number goes out of scope here
    } // level_1_number goes out of scope here
} // level_0_str goes out of scope here
```

No surprises there, certainly. Each let binding gets allocated in the stack, and the non-primitive parts (here the String and the Vector that gets created by the `vec!` macro) in the heap. This compiles just fine, although with warnings since we are not using these variables for anything.

The next piece should be more interesting:

```
// multiple-owners.rs
fn main() {
    let num1 = 1;
    let num2 = num1;

    let s1 = String::from("meep");
    let s2 = s1;

    println!("Number num1 is {}", num1);
    println!("Number num2 is {}", num2);

    println!("String s1 is {}", s1);
    println!("String s2 is {}", s2);
}
```

This one looks fairly simple as well. Both `num2` and `s2` get their contents from `num1` and `s1`. Nevertheless, this fails to compile:

```
vegai@carbon ~/rustbook/6 » rustc multiple-owners.rs
error[E0382]: use of moved value: `s1`
  --> multiple-owners.rs:11:33
   |
6  |     let s2 = s1;
   |         -- value moved here
...
11 |     println!("String s1 is {}", s1);
   |                                 ^^ value used here after move
   |
   = note: move occurs because `s1` has type `std::string::String`, which does not imp
lement the `Copy` trait

error: aborting due to previous error

vegai@carbon ~/rustbook/6 »                                                    101 ↲
```

Even odder still: the `num2` binding was fine! This is because types in Rust work differently based on how the types themselves are implemented:

- All types are movable by default
- All types that implement the `Copy` trait are copyable

As you figured out with some help from the compiler, the String type does not implement the `Copy` trait and is therefore a movable type. This means that every time you create a new binding with a reference to it, the original value gets invalidated and cannot be used again.

Copy trait

When a type implements the `Copy` trait (like all the primitive number types do), every new binding causes a new copy of the value instead of a move. This is why the `num2` binding in the example before was fine: it caused a new copy to be created and `num1` was left intact and still usable.

High level programming languages do similar things, but hide what actually happens behind the curtain of the implementation. For instance, check out these assignment operations followed by mutations in Python:

```
Python 3.6.0 (default, Jan 16 2017, 12:12:55)
[GCC 6.3.1 20170109] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> s1 = "string"
>>> s2 = s1
>>> s2 += " added"
>>> s2
'string added'
>>> l1 = [1, 2, 3]
>>> l2 = l1
>>> l1.append(4)
>>> l1
[1, 2, 3, 4]
>>> l2
[1, 2, 3, 4]
>>> 
```

Since strings in Python are immutable, the assignment operation must copy the string and the mutation makes yet a third copy. On the other hand, Python lists are mutable, so `l1` and `l2` will point to the same list and therefore have the same contents. Every python programmer knows these details and therefore it works very well in practice. Rust, however, has no such luxury because of the goals of zero cost thread-safe memory safety.

If we really wanted to do this, we could explicitly clone the String:

```
let s2 = s1.clone();
```

Cloning requires that the type implements the `Clone` trait, which Strings happen to do.

You're probably wondering what the difference between `Copy` and `Clone` traits is. Good question! Here are a few guidelines:

- If a type can be replicated extremely cheaply, that is, by simply copying the bits within, the `Copy` trait may be implemented for it.
- If the type depends only on other types that have `Copy` implemented on them, the `Copy` trait may be implemented for it.
- Otherwise, the `Clone` trait may be used. Its implementation may be more expensive.

- The `Copy` trait implicitly affects how the assignment operator `=` works.
- The `Clone` trait merely declares a `clone` method, which needs to be called explicitly.

The decision whether to make your own externally visible types obey the `Copy` trait requires some consideration due to how it affects the assignment operator. If at an early point of development your type is a `Copy` and you remove it afterwards, it affects every point where values of that type are assigned. You can easily break an API in that manner.

Function parameters and patterns

The same system of moves and copies works for other variable bindings besides just the `let` form. If you pass parameters to functions, the same rules are in effect:

```
// functions.rs
fn take_the_n(n: u8) {
}

fn take_the_s(s: String) {
}

fn main() {
    let n = 5;
    let s = String::from("string");

    take_the_n(n);
    take_the_s(s);

    println!("n is {}", n);
    println!("s is {}", s);
}
```

The compilation fails in a familiar way:

```
vegai@carbon ~/rustbook/6 » rustc functions.rs                                          101 ↲
error[E0382]: use of moved value: `s`
  --> functions.rs:16:25
   |
13 |     take_the_s(s);
   |                - value moved here
...
16 |     println!("s is {}", s);
   |                         ^ value used here after move
   |
   = note: move occurs because `s` has type `std::string::String`, which does not implem
ent the `Copy` trait

error: aborting due to previous error

vegai@carbon ~/rustbook/6 »                                                             101 ↲
```

The String type does not implement the `Copy` trait, so the ownership of the value is moved to the `take_the_s` function. When that function's block ends, the scope of the value is finished and it is freed. Therefore it cannot be used after the function call any more. The trivial fix is similar to before: add a `.clone()` call at the function call site:

```
    take_the_s(s.clone());
```

So, in effect you would have to clone all the function parameters that do not implement the `Copy` trait, and even those that do implement it get copied every time. As you might imagine, that doesn't fly well with the zero-cost promise, plus it's quite awkward. That's where the borrowing system comes in.

Here are a few exercises that you can do:

1. Take your second favorite programming language and try to figure out if ownership of variables plays any part. Perhaps behind the curtain, hidden?
2. Does the compiler/interpreter help the coder in that language with ownership issues or is it all in the hands of the programmer?

Borrowing

As you saw before, moving ownership when making function calls does not usually make much sense. Instead, you can define the function parameters as borrowed references with the ampersand `&`. We can fix the previous code example to pass the compiler without cloning like this:

```
// functions-with-borrows-1.rs
fn take_the_n(n: &u8) {
}

fn take_the_s(s: &String) {
}

fn main() {
    let n = 5;
    let s = String::from("string");

    take_the_n(&n);
    take_the_s(&s);

    println!("n is {}", n);
    println!("s is {}", s);
}
```

Note that the `&` needs be used in both the call site and in the parameter list. Similar to how variable bindings are by default, references are immutable by default.

In order to get to the actual value that's behind the reference, you use the asterisk operator `*`. For instance, if we want `take_the_n` to also output the number, it would look like this:

```
fn take_the_n(n: &u8) {
   println!("n is {}", *n);
}
```

To get a mutable reference, you will need to modify three things: the actual variable binding, the call site, and the function parameter list. First, the variable binding would have to be made mutable:

```
let mut n = 5;
```

Then, the function would change to this:

```
fn take_the_n(n: &mut u8) {
    *n = 10;
}
```

The call site would need to change to this form:

```
    take_the_n(&mut n);
```

Again, we see that everything in Rust is explicit. If they are particularly dangerous things, they are even more explicit. Mutable variables are, for obvious reasons, quite a lot more dangerous than immutable ones, especially when multiple threads come into play.

There are a couple of rules related to borrow references:

- A borrow reference may not live longer than what it referred to. Obviously, since if it did, it would be referring to a dead thing.
- If there's a mutable reference to a thing, no multiple references (mutable or immutable) are allowed to the same thing at the same time.
- If there is no mutable reference to a thing, any number of immutable references to the same thing at the same time are allowed.

These rules are verified at compile time by the compiler's borrow checker. Let's see examples of violations for each of these points:

1. We'll have a function that tries to return a reference to a value that goes away when the function exits:

```
// borrows-1.rs
fn get_a_borrowed_value<'a>() -> &'a u8 {
    let x = 1;
    &x
}

fn main() {
    let value = get_a_borrowed_value();
}
```

   `<'a>` is a lifetime specification; we'll get to those in a minute. This fails to pass the borrow checker:

```
vegai@carbon ~/rustbook/6 » rustc borrows-1.rs
error: `x` does not live long enough
 --> borrows-1.rs:3:6
  |
3 |     &x
  |      ^ does not live long enough
4 | }
  | - borrowed value only lives until here
  |
note: borrowed value must be valid for the lifetime 'a as defined on the body at 1:40...
 --> borrows-1.rs:1:41
  |
1 |   fn get_a_borrowed_value<'a>() -> &'a u8 {
  |  _________________________________________^ starting here...
2 | |     let x = 1;
3 | |     &x
4 | | }
  | |_^ ...ending here

error: aborting due to previous error

vegai@carbon ~/rustbook/6 »                                                    101 ↲
```

2. We can have any number of immutable references to something:

```
// borrows-2.rs
fn main() {
    let x=1;
```

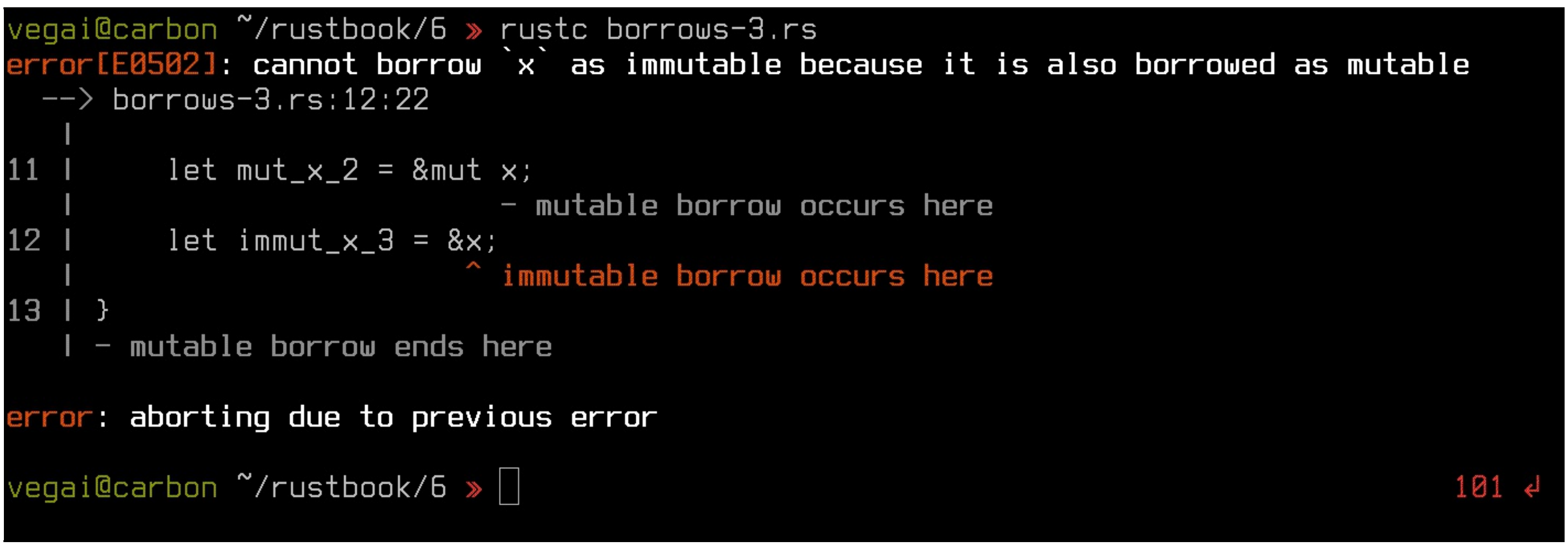
```
        let x1 = &x;
        let x2 = &x;

        println!("x1 says {}", *x1);
        println!("x2 says {}", *x2);
    }
```

This compiles and runs just as expected.

3. If there's an active mutable reference to something, there may be no other references to it:

```
    // borrows-3.rs
    fn main() {
        let mut x = 1;
        {
            let immut_x_1 = &x;
        }

        {
            let mut_x_1 = &mut x;
        }

        let mut_x_2 = &mut x;
        let immut_x_3 = &x;
    }
```

This fails to compile:



The first two borrows don't matter, since they are gone after blocks, but the last immutable borrow breaks the rules and breaks the code.

The motivation for this system is mainly to protect against variable misuse in multithreaded situations. The rules that allow many immutable references but only a single mutable one are similar to the rules in distributed systems: multiple read-only locks are fine, but even a single write lock affects everything.

Note that mutability or immutability is defined totally on the binding level. That is, a value is either mutable or immutable based on what the binding is. This also applies to things like structs and enums: either all of their fields are mutable or none of them are. That's not the whole story, though, since an immutable field in a struct may be a reference to something else, and while that reference cannot change, the thing it points to can. The `Cell` and `RefCell` types especially take advantage of this, and we'll cover them soon.

Note that this move versus the borrow mechanism works identically for `impl` blocks too, especially their `self` parameter. If you define a method that takes `self` as a non-borrowed variable, that means that ownership of `self` moves to the method, and when the method finishes, `self` goes out of scope and gets dropped! So unless you're deliberately writing a method that should drop `self` at the end, always use `&self` as a method parameter.

Lifetimes

The third piece in the Rust memory safety puzzle is lifetimes. If you have ever programmed in C, you should be acutely aware of the lifetime issue: every time you allocate some variable with malloc, it should have an explicit owner and that owner should reliably decide when that variable's life ends. It's not codified anywhere; rather it's the programmer's responsibility.

In Rust every reference has a lifetime attached to it. A lifetime defines how long the reference lives in relation to other references. Whenever it's able to, the Rust compiler juggles with them without the programmer's help via a mechanism called lifetime elision. Sometimes it's not able to, however, and then it needs our help.

Here's a list of all the places we may need to manually specify lifetimes:

- Global static and const
- Function signatures
- Structs and struct fields
- impl signatures

Let's go through each of them.

Globals

We've seen one of the cases multiple times before global string slices:

```
const MEEP: &'static str = "meep";
static SECOND_MEEP: &'static str = "meep2";
```

The lifetime needs to be specified here since Rust's type inference is only local, so we need to spell out the types for all globals. The static lifetime means that these values start existing when the program starts and go away when the program does. All the literal strings in Rust programs are static, and since `&'static str` is not the same type as `&str`, we get a type error if we don't explicitly specify the lifetime here.

References as function parameters

Whenever there's a reference in a function, either as input parameter or output values, that reference gets a lifetime. In many cases, the compiler is able to figure out the only possible lifetime so we don't have to. In other words, these two function signatures are identical:

```
fn f(x: &u8) → &u8
fn f<'a>(x: &'a u8) → &'a u8
```

I recommend looking at the lifetime syntax very slowly when you see it the first time. It may be daunting at first, but it gets easier rapidly. The first occurrence, just after the function name, is the lifetime declaration. It's saying that the `f` function contains parameters with lifetime `'a`. In the second occurrence we say that `x` has the lifetime `'a`, and the third says that the function returns a value with that same lifetime. You might notice that the syntax is similar to the generic type syntax, and that is not an accident: lifetimes are one kind of a generic type.

So what all this says is that you cannot return a primitive reference to a function, unless you brought that reference in as a parameter to the function. Also, if you bring in more than one reference, you need to specify which lifetime you're returning. In other words, this won't fly:

```
fn f(x: &u8, y: &u8) → &u8
```

With more than one reference and a returning reference, you have to explicitly define the lifetimes:

```
fn f<'a>(x: &'a u8, y: &'a u8) → &'a u8
```

Structs and struct fields

Whenever structs have references in them, we need to specify explicitly how long those references will live. The syntax is similar to that of the function signatures: we first declare the lifetime names on the struct line, and then use them in the fields.

Here's what the syntax looks like in the simplest form:

```
struct Number<'a> {
    num: &'a u8
}
```

What we are saying here is that the `num` field must not refer to any `u8` value that would live less long than the enclosing instance of the struct `Foo`. We are saying it explicitly again, as is the Rust way.

Impl signatures

When we create `impl` blocks for structs with references, we need to repeat the lifetime declarations and definitions again. For instance, if we made an implementation for the `Foo` struct we defined previously, the syntax would look like this:

```
// lifetime-structs.rs
impl<'a> Number<'a> {
    fn get_the_number(&self) -> &'a u8 {
        self.num
    }
    fn set_the_number(&mut self, new_number: &'a u8) {
   self.num = new_number
}
```

The Drop trait

The `Drop` trait is what you would call an object destructor method in other languages. It contains a single method `drop`, which gets called when the object goes out of scope. This is done in a strict order: **last in, first out**. That is, whatever was constructed the last, gets destructed the first. For example:

```
// drops.rs
struct Character {
    name: String
}

impl Drop for Character {
    fn drop(&mut self) {
        println!("{} went away", self.name)
    }
}

fn main() {
    let steve = Character { name: "Steve".into() };
    let john = Character { name: "John".into() };
}
```

And the output is as follows:

```
vegai@carbon ~/rustbook/6 » rustc drops.rs
warning: unused variable: `steve`, #[warn(unused_variables)] on by default
  --> drops.rs:12:9
   |
12 |     let steve = Character { name: "Steve".into() };
   |         ^^^^^

warning: unused variable: `john`, #[warn(unused_variables)] on by default
  --> drops.rs:13:9
   |
13 |     let john = Character { name: "John".into() };
   |         ^^^^

vegai@carbon ~/rustbook/6 » ./drops
John went away
Steve went away
vegai@carbon ~/rustbook/6 »
```

This mechanism is where you should put the cleanup code for your own structs, if they need any. It's especially handy for types where the cleanup is less clearly deterministic, such as when using reference counted values or garbage collectors.

Collector types

Next, we'll take a look at a few generic types by which we can control how the memory allocation in the heap is done. The types are as follows:

- `Box<T>`: This is the simplest form of heap allocation. The box owns the value inside it, and can thus be used for holding values inside structs or for returning them from functions.
- `Cell<T>`: This gives us internal mutability for types that implement the `Copy` trait. In other words, we gain the possibility to get multiple mutable references to something.
- `RefCell<T>`: This gives us internal mutability for types, without requiring the `Copy` trait. Uses runtime locking for safety.
- `Rc<T>`: This is for reference counting. It increments a counter whenever somebody takes a new reference, decrements it when someone releases a reference. When the counter hits zero, the value is dropped.
- `Arc<T>`: This is for atomic reference counting. Like the previous type, but with atomicity to guarantee multithread safety.
- Combinations of the previous types (such as `RefCell<Vec<T>>`).

Box<T>

The generic type box in the standard library gives us the simplest way to allocate values in the heap. If you're familiar with the concept of boxing and unboxing from other languages, this is the same.

The box itself does not implement the `Copy` trait, which makes it a move type. This means, like for other move types, that if you make a new binding to the existing box, the earlier binding gets invalidated.

To get a new box, do as you would for any other container type: call the static `new` method. To unbox, use the `*` operator:

```
let boxed_one = Box::new(1);
let unboxed_one = *boxed_one;
```

Interior mutability for Copy types - Cell<T>

As seen before, Rust protects us at compile time from the aliasing problem by allowing only a single mutable reference at the same time. However, there are cases where that is too restrictive, making code that we know is safe not pass the compiler because of the strict borrow checks.

Interior mutability allows us to bend the borrowing rules a bit. The standard library has two generic types for this: `Cell` and `RefCell`. Cell is zero-cost: the compiler generates code that is similar to primitive mutable references. The point is that, as we saw before, multiple mutable references are not acceptable. `Cell<T>` requires that the enclosed types implement the `Copy` trait.

Because `Cell` bends the rules, you should be wary when you use or see `Cell<T>`: there may be more than one mutable reference to the value inside. This, of course, means that the value you read from a `Cell` may change after you read it.

`Cells` work via three methods:

- `Cell::new` makes a new `Cell` with the value given as parameter inside
- The `get` method returns the value inside
- The `set` method replaces the value with a new one

In the simplest form, we could just take several mutable pointers to a single value:

```
fn main() {
    let x = 1;
    let ref_to_x_1 = &mut x;
    let ref_to_x_2 = &mut x;

    *ref_to_x_1 += 1;
    *ref_to_x_2 += 1;
}
```

But, of course, this does not compile due to the basic borrowing checks:

```
vegai@carbon ~/rustbook/6 » rustc multiple-mutref.rs
error: cannot borrow immutable local variable `x` as mutable
 --> multiple-mutref.rs:3:27
  |
2 |     let x = 1;
  |         - use `mut x` here to make mutable
3 |     let ref_to_x_1 = &mut x;
  |                           ^ cannot borrow mutably

error: cannot borrow immutable local variable `x` as mutable
 --> multiple-mutref.rs:4:27
  |
2 |     let x = 1;
  |         - use `mut x` here to make mutable
3 |     let ref_to_x_1 = &mut x;
4 |     let ref_to_x_2 = &mut x;
  |                           ^ cannot borrow mutably

error: aborting due to 2 previous errors

vegai@carbon ~/rustbook/6 »                                          101 ↲
```

You can make this work by encapsulating your value inside a `Cell` and using it via that:

```
// multiple-cells.rs
use std::cell::Cell;

fn main() {
    let x = Cell::new(1);
    let ref_to_x_1 = &x;
    let ref_to_x_2 = &x;

    ref_to_x_1.set(ref_to_x_1.get() + 1);
    ref_to_x_2.set(ref_to_x_2.get() + 1);

    println!("X is now {}", x.get());
}
```

This works as you would expect, and the only added cost is that the code is a slightly more awkward. The additional runtime cost is zero, though, and the references to the mutable things remain immutable. Internally, this works exactly due to the requirement of the internal value being a `Copy` type: Rust is free to copy the internal value without needing to worry that dropping the previous values would cause problems.

Interior mutability for move types - RefCell<T>

If you need `Cell`-like features for your non-copying types, `RefCell` can help. It uses read/write locks at runtime for you, which is convenient but not zero-cost. The other difference is that whereas `Cell` lets you handle actual values, `RefCell` handles references. This means that the same mutable borrow restrictions are in effect that work with the primitive `let` reference bindings, but `RefCell` restrictions are checked at runtime instead of compile-time.

This is what happens if you try to use a `Cell<T>` with a type that moves instead of copies. In other words, with a type that does not implement the `Copy` trait:

```
// multiple-move-types.rs
use std::cell::Cell;

struct Foo {
    number: u8
}

fn main() {
    let foo_one = Cell::new(Foo { number: 1 });
    let ref_to_foo_1 = &foo_one;
    let ref_to_foo_2 = &foo_one;

    foo_one.set( Foo { number: 2});
    foo_one.set( Foo { number: 3});
}
```

Quote the compiler:

```
vegai@carbon ~/rustbook/6 » rustc multiple-move-types.rs
error[E0277]: the trait bound `Foo: std::marker::Copy` is not satisfied
 --> multiple-move-types.rs:8:19
  |
8 |     let foo_one = Cell::new(Foo { number: 1 });
  |                   ^^^^^^^^^ the trait `std::marker::Copy` is not implemented for `Fo
o`
  |
  = note: required by `<std::cell::Cell<T>>::new`

error: no method named `set` found for type `std::cell::Cell<Foo>` in the current scope
  --> multiple-move-types.rs:12:13
   |
12 |     foo_one.set( Foo { number: 2});
   |             ^^^
   |
   = note: the method `set` exists but the following trait bounds were not satisfied: `F
oo : std::marker::Copy`

error: no method named `set` found for type `std::cell::Cell<Foo>` in the current scope
  --> multiple-move-types.rs:13:13
   |
13 |     foo_one.set( Foo { number: 3});
   |             ^^^
   |
   = note: the method `set` exists but the following trait bounds were not satisfied: `F
oo : std::marker::Copy`

error: aborting due to 3 previous errors

vegai@carbon ~/rustbook/6 »                                                      101 ↲
```

The `RefCell` API for the basic parts is the two borrowing methods:

- The `borrow` method takes a new immutable reference
- The `borrow_mut` method takes a new mutable reference

Let's try converting the preceding code to using `RefCells` instead:

```
// multiple-move-types-with-refcell-1.rs
use std::cell::RefCell;

struct Foo {
    number: u8
}

fn main() {
    let foo_one = RefCell::new(Foo { number: 1 });
    let mut ref_to_foo_1 = foo_one.borrow_mut();
    let mut ref_to_foo_2 = foo_one.borrow_mut();

    ref_to_foo_1.number = 2;
    ref_to_foo_2.number = 3;
}
```

This compiles just fine, but there is a problem here. We broke the *there can be only one mutable reference* rule, and we get a panic:

```
vegai@carbon ~/rustbook/6 » rustc multiple-move-types-with-refcell-1-broken.rs
vegai@carbon ~/rustbook/6 » ./multiple-move-types-with-refcell-1-broken
thread 'main' panicked at 'already borrowed: BorrowMutError', /buildslave/rust-buildbot
/slave/stable-dist-rustc-linux/build/src/libcore/result.rs:868
note: Run with `RUST_BACKTRACE=1` for a backtrace.
vegai@carbon ~/rustbook/6 »                                                      101 ↲
```

We need to let the `borrow` bindings go away somehow, either by enclosing them in blocks or by explicitly calling the `drop` function, like we do here:

```
// multiple-move-types-with-refcell-2.rs
let mut ref_to_foo_1 = foo_one.borrow_mut();
ref_to_foo_1.number = 2;
drop(ref_to_foo_1);

let mut ref_to_foo_2 = foo_one.borrow_mut();
ref_to_foo_2.number = 3;
```

This version runs without errors.

Practical uses of interior mutability

The examples of `Cell` and `RefCell` were simplified, and you would most probably not need to use them in that form in real code. Let's take a look at some actual benefits that these types would give us.

As mentioned before, bindings are not fine-grained: a value is either immutable or mutable, and that includes all its fields if it's a struct or an enum. `Cell` and `RefCell` can turn an immutable thing into mutable, allowing us to define parts of an immutable struct as mutable.

The following piece of code augments a struct with two integers and a `sum` method to cache the answer of the `sum` and return the cached value if it exists:

```
// interior-mutability.rs
use std::cell::Cell;

struct Point {
    x: u8,
    y: u8,
    cached_sum: Cell<Option<u8>>
}

impl Point {
    fn sum(&self) -> u8 {
        match self.cached_sum.get() {
            Some(sum) => {
                println!("Got from cache: {}", sum);
                sum
            },
            None => {
                let new_sum = self.x + self.y;
                self.cached_sum.set(Some(new_sum));
                println!("Set cache: {}", new_sum);
                new_sum
            }
        }
    }
}

fn main() {
    let p = Point { x: 8, y: 9, cached_sum: Cell::new(None) };

    println!("Summed result: {}", p.sum());
    println!("Summed result: {}", p.sum());
}
```

Running this code shows that the cache is working without needing to make the whole `p` mutable!

```
vegai@carbon ~/rustbook/6 » ./intbut-example
Set cache: 17
Summed result: 17
Got from cache: 17
Summed result: 17
vegai@carbon ~/rustbook/6 »
```

In addition to using `Cell` types with your own structs, there's a further pattern where we combine `Cell`/`RefCell` with another generic type that usually works with immutable types only. One such example is the `Rc<T>` type, which we'll look into next.

Reference collected memory: Rc<T> and Arc<T>

Reference counting is a simple form of garbage collection. The basic flow of events with Rc is as follows:

- Every time somebody takes a new reference, we increment an internal counter
- Every time somebody drops a reference, we decrement it
- When the internal counter hits zero, nobody refers to the object anymore, so it can be dropped

Using variables in reference counted containers gives us more flexibility in the implementation: we can hand out references to a value without having to keep exact track of when the references go out of scope.

Rc<T> is mostly used via two methods:

- The static method Rc::new makes a new reference collected container (you should start recognizing a pattern already!)
- The clone method increments the strong reference count and hands out a new Rc<T>

The reference counting system supports two kinds of references: strong (Rc<T>) and weak (Weak<T>). Both keep a count of how many references of each type have been handed out, but only when the strong references hit zero do the values get deallocated. The motivation for this is that an implementation of a data structure may need to point to the same thing multiple times. For instance, an implementation of a tree might have references to both the child nodes and the parent, but incrementing the counter for each such reference would not be correct. Instead, using weak references for the parent references would not corrupt the count.

As another example, a linked list might be implemented in such a way that it maintains links via reference counting to both the next item and to the previous. However, if we count for each direction, the count would be incorrect. A better way to do this would be to use strong references to one direction and weak references to the other.

Let's see how that might work. Here's a minimal implementation of possibly the worst practical but best learning data structure: singly linked list:

```
// rc-1.rs
use std::rc::Rc;

#[derive(Debug)]
struct LinkedList<T> {
    head: Option<Rc<LinkedListNode<T>>>
}

#[derive(Debug)]
struct LinkedListNode<T> {
    next: Option<Rc<LinkedListNode<T>>>,
```

```
    data: T
}

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

    fn append(&self, data: T) -> Self {
        LinkedList {
            head: Some(Rc::new(LinkedListNode {
                data: data,
                next: self.head.clone()
            }))
        }
    }
}

fn main() {
    let list_of_nums = LinkedList::new().append(1).append(2);
    println!("nums: {:?}", list_of_nums);

    let list_of_strs = LinkedList::new().append("foo").append("bar");
    println!("strs: {:?}", list_of_strs);
}
```

This linked list is formed of two structs: `LinkedList` provides a reference to the first element of the list and the list's public API, and `LinkedListNodes` contain the actual elements. Notice how we're using `Rc` and cloning the next data pointer on every append. Let's walkthrough what happens in the append case:

1. `LinkedList::new()` gives us a new list. Head is `None`.
2. We append `1` to the list. Head is now the node that contains `1` as data, next is the previous head: `None`.
3. We append `2` to the list. Head is now the node that contains `2` as data, next is the previous head, the node that contains `1` as data.

The debug output from `println!` confirms this:

```
vegai@carbon ~/rustbook/6 » ./rc-2
nums: LinkedList { head: Some(LinkedListNode { next: Some(LinkedListNode { next: None
, data: 1 }), data: 2 }) }
strs: LinkedList { head: Some(LinkedListNode { next: Some(LinkedListNode { next: None
, data: "foo" }), data: "bar" }) }
vegai@carbon ~/rustbook/6 »
```

This is a rather functional form of this structure: every append works by just adding data at the head, which means that we don't have to play with references and actual list reference can stay immutable. That changes a bit if we want to keep the structure this simple but still have a double-linked list, since then we actually have to change the existing structure.

You can downgrade an `Rc<T>` type into a `Weak<T>` type with the `downgrade` method, and similarly a `Weak<T>` type can be turned into `Rc<T>` using the `upgrade` method. The downgrade method will always work. In contrast, when calling upgrade on a weak reference, the actual value might have been dropped already, in which case you get a `None`.

So let's add a weak pointer to the previous node:

```
// rc-2.rs
use std::rc::Rc;
use std::rc::Weak;

#[derive(Debug)]
struct LinkedList<T> {
    head: Option<Rc<LinkedListNode<T>>>
}

#[derive(Debug)]
struct LinkedListNode<T> {
    next: Option<Rc<LinkedListNode<T>>>,
    prev: Option<Weak<LinkedListNode<T>>>,
    data: T
}

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

    fn append(&mut self, data: T) -> Self {
        let new_node = Rc::new(LinkedListNode {
            data: data,
            next: self.head.clone(),
            prev: None
        });

        match self.head.clone() {
            Some(node) => {
                node.prev = Some(Rc::downgrade(&new_node));
            },
            None => {
            }
        }

        LinkedList {
            head: Some(new_node)
        }
    }
}

fn main() {
    let list_of_nums = LinkedList::new().append(1).append(2).append(3);
    println!("nums: {:?}", list_of_nums);
}
```

The `append` method grew a bit: we now need to update the previous node of the current head before returning the newly created head. This is almost good enough, but not quite. The compiler doesn't let us do naughty things:

```
vegai@carbon ~/rustbook/6 » rustc rc-3.rs
error: cannot assign to immutable field
  --> rc-3.rs:30:17
   |
30 |                 node.prev = Some(Rc::downgrade(&new_node));
   |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |

error: aborting due to previous error

vegai@carbon ~/rustbook/6 »                                           101 ↲
```

We could make `append` take a mutable reference to `self`, but that would mean that we could only append to the list if all the nodes' bindings were mutable, forcing the whole structure to be mutable. What we really want is a way to make just one small part of the whole structure mutable, and fortunately we can do that with a single `RefCell`.

1. Add a `use` for the `RefCell`:

```
use std::cell::RefCell;
```

2. Wrap the previous field in `LinkedListNode` in a `RefCell`:

```
// rc-3.rs
#[derive(Debug)]
struct LinkedListNode<T> {
    next: Option<Rc<LinkedListNode<T>>>,
    prev: RefCell<Option<Weak<LinkedListNode<T>>>>,
    data: T
}
```

3. We change the `append` method to create a new `RefCell` and update the prev reference via the `RefCell` mutable borrow:

```
    // rc-3.rs
    fn append(&mut self, data: T) -> Self {
        let new_node = Rc::new(LinkedListNode {
            data: data,
            next: self.head.clone(),
            prev: RefCell::new(None)
        });

        match self.head.clone() {
            Some(node) => {
                let mut prev = node.prev.borrow_mut();
                *prev = Some(Rc::downgrade(&new_node));
            },
            None => {
            }
        }

        LinkedList {
            head: Some(new_node)
        }
    }
}
```

Whenever using `RefCell` borrows, it's good practice to think carefully that we're using it in a safe way, since making mistakes there may lead to runtime panics. In this implementation, however, it's easy to see that we have just the single `borrow`, and that the closing block immediately discards it.

Inspecting memory usage with std::mem

If you're interested in how much all these various collector types use memory, you don't have to guess. The `std::mem` module contains useful functions for checking that at runtime. Let's take a look at a couple:

- `size_of` returns the size of a type given via a generic type
- `size_of_val` returns the size of a value given as a reference

In case we were skeptical about the zero-cost claims of some of the preceding generic types, we can use these functions to check the overhead. The call style for `size_of` may be a bit peculiar if you're not familiar with it yet: we are not actually giving it anything as a parameter; we're just explicitly calling it against a type. Let's take a look at some sizes:

```
// mem-introspection.rs
use std::cell::Cell;
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    println!("type u8: {}", std::mem::size_of::<u8>());
    println!("type f64: {}", std::mem::size_of::<f64>());
    println!("value 4u8:  {}", std::mem::size_of_val(&4u8));
    println!("value 4:  {}", std::mem::size_of_val(&4));
    println!("value 'a': {}", std::mem::size_of_val(&'a'));

    println!("value \"Hello World\" as a static str slice: {}", std::mem::size_of_val("Hello World"));
    println!("value \"Hello World\" as a String: {}", std::mem::size_of_val("Hello World").to_string());

    println!("Cell(4)): {}", std::mem::size_of_val(&Cell::new(84)));
    println!("RefCell(4)): {}", std::mem::size_of_val(&RefCell::new(4)));

    println!("Rc(4): {}", std::mem::size_of_val(&Rc::new(4)));
    println!("Rc<RefCell(8)>): {}", std::mem::size_of_val(&Rc::new(RefCell::new(4))));
}
```

Here are a few exercises specifically on memory reflection:

1. Try to reason out the sizes of each of the preceding types.
2. Compile and run the code. Go through the differences between your guesses and reality.

Final exercises

1. Find blog posts about lifetimes, borrowing, and ownership. Read 'em!
2. Take a look at some more advanced projects; take a look at the data structures, paying special attention to the memory handling. Look for `Cells`, `RefCells`, and `Rc`. Look for lifetime annotations.

Summary

Rust takes a low-level systems programming approach to memory management, promising C-like performance. It does this without requiring a garbage collector by its system of memory ownership, borrowing, and lifetimes. The concepts are not new, but their combination and codification and the breadth of the safety given by them is.

We covered a whole lot of ground here in a subject that's probably the heaviest to grasp for a new Rust programmer. Getting fluent in all this takes quite an amount of work and various different approaches to the problem. The final exercises of this chapter are more free form, so as to give you a bit of breathing space after this grind.

Concurrency

Often, we'd like our programs to do several things at the same time:

- The user interface of a program continues working normally even though our program connects to the network
- A game updates the state of thousands of units at the same time, while playing a soundtrack in the background
- A scientific program splits computation in order to take full advantage of all the cores in the machine
- A web server handles more than one request at a time in order to maximize throughput

Rust has a safe form of concurrency, backed by the memory model system described in the previous chapter. In this chapter, we will go through how multithreading, sharing state, and message passing work in Rust.

The topics covered in this chapter are as follows:

- Problems of concurrency
- Closures
- Threads
- Channels
- Mutexes
- Atomic reference counting

Problems with concurrency

Concurrency means doing more than one independently happening thing during some time period. This is a general term, and the method of these different things happening might differ based on the circumstances. For instance, if you have a concurrent program running on a single-core machine, the execution of that program would jump between various tasks. If you had a multicore machine, the execution of the different parts might happen in *parallel*.

As a real-life experience, you could think of the process of preparing a dish. It is a concurrent process: you need to boil the rice, make a salad, fry the tofu, and make the salad dressing. Possibly, your child will need something, and you'll need to interact with them. If you're doing all this alone, you will be switching between these tasks, possibly finishing some before the others. Some of the tasks will have dependencies: for instance, you should be boiling the rice before even starting the other tasks. Some tasks may stay in a waiting state while not blocking the other tasks: you can start making the salad once the rice is boiling without your help. Some tasks will be completely independent: your child may have nothing to do with the tasks related to cooking.

If there is more than one of you doing it, you may benefit from **parallelism***:* your friend could be making the salad while you're boiling the rice and frying the tofu. Some of the dependencies between the tasks will still be there, but many of the things may be executed at the same time.

The core of the problem of concurrency comes from the indeterminism; you cannot accurately predict beforehand in what order things will start to happen and when they will finish. These problems are generally called **race conditions**, referring to multiple things behaving like race cars, trying to reach their own goals as fast as they can. Race conditions are not always catastrophic; just an unexpected order of things happening may be a race. In our cooking scenario, you might consider that for a perfect dish, you might want the rice and tofu to be ready at exactly the same time. However, the process that works fine when you cook by yourself now breaks slightly when your friend comes to help: the frying of the tofu will race to its goal too soon. To fix that, you may add **synchronization**: for instance, not let your friend start frying until the rice has been cooking for a while.

Another class of problems is called **data races**, where multiple things handle a shared resource without proper synchronization so that at least one of them has a write access to the resource. With a risk of stretching the cooking analogy too far, let's say that you decide to describe the process of food making on a single piece of paper, with both cooks having their own pens. When the water starts boiling, you want to report *Water boiling*. At the same time, your friend wants to report *Tofu frying*. Absurdly, you'll write on the paper at the exact same time, ending up with a gibberish phrase, *Wofer bryinging*.

A special case of race conditions are **deadlocks.** A deadlock happens when synchronization is added to one or more shared resource but without taking care that the resources are locked and released in the correct order. This may lead to a situation where multiple processes are cross-waiting for each other's resources. Your cooking situation could be further complicated by having a single bottle of oil and a single container of salt. Your recipe for cooking the rice might call for first adding the salt, then adding the oil, while the tofu cooking recipe would call for adding the ingredients in reverse order. Then, the following would happen:

1. You pick up the salt and add it to your rice. You start waiting for the oil bottle to be free.
2. At the same time, your friend picks up the oil and pours it on the pan. She starts waiting for the salt container to be free.
3. Deadlock happens when neither of you are able to let go of your own resource because you're waiting for the other.

Rust's type system almost completely protects against data races at compile time, with the exception that you may request unsafety if you need it for some reason. We'll cover a few of those cases in Chapter 10, *Unsafety and Interfacing with Other Languages*. More specifically, Rust gives us that protection by the restriction that only a single mutable reference to a thing may be active at the same time.

For other race conditions, there are mechanisms for helping with synchronization. We'll cover mutexes and atomic reference counting and the Send and Sync traits. But let's start by first bringing closures back to the front of our minds.

Closures

Closures give us a way to quickly define small, anonymous functions, which optionally grab some of the variables defined in the outer scope. That's where the name comes from: the variables from the outer scope are "closed over".

Threads are often launched via closures due to their terser syntax and features. The syntax should be familiar to any Ruby programmers, but there are a few Rusty additions.

In a simple form, closures are semantically identical to functions. Here are four similar functions or closures that take and return a value, and two that take no parameters:

```
fn square(x: u32) -> u32 {
 x * x
}

fn function_without_vars() {
 println!("Entered function without variables");
}

fn main() {
 let square_c1 = |x: u32| x*x;
 let square_c2 = |x: u32| { x*x };
 let square_c3 = |x: u32| -> u32 { x*x };

 let closure_without_vars = || println!("Entered closure without variables");

 println!("square of 4 = {}", square(4));
 println!("square of 4 = {}", square_c1(4));
 println!("square of 4 = {}", square_c2(4));
 println!("square of 4 = {}", square_c3(4));

 function_without_vars();
 closure_without_vars();
}
```

As you see from the preceding example, a local type inference sometimes allows the omission of the variable types, for instance, the return type for the square closures. Rust cannot infer the input type from the simple `x*x`, so it cannot be omitted. Curly brackets are optional if the return type is omitted and there's only a single expression in the closure body.

As mentioned before, one big feature of closures is that it closes over variables defined in the outer scope. They can do this in one of the two ways: using borrow semantics or using move semantics. Borrow semantics is the default, and move semantics can be requested with the `move` keyword:

```
// closures-2.rs
fn main() {
    let mut outer_scope_x = 42;

    {
        let mut closure = move || {
            outer_scope_x += 42;
            println!("Outer scope variable is {}", outer_scope_x);
        };

        closure();
    }
    println!("Outer_scope_x {}", outer_scope_x);
```

```
}
```

The effects of `move` semantics may be confusing because it depends on the traits that have been defined for the moving type. In this case, `outer_scope_x` gets closed with `move` semantics. It's a primitive type, which means that the `Copy` trait has been implemented for it, which in turn means that instead of moving the variable, it gets copied.

Therefore, even though the `outer_scope_x` variable inside the closure gets mutated, the variable outside does not. The output of this program looks as follows:

```
vegai@carbon ~/rustbook/7 » ./closures-2
Outer scope variable is 84
Outer_scope_x 42
vegai@carbon ~/rustbook/7 »
```

Exercises

1. Remove `mut` from the closure declaration line. Why does that make the compilation fail?
2. Remove `move` from the closure declaration line. What's the effect and why?
3. It looks like we don't need the block starting from line 4 and ending on line 11. Try to remove that and see if that's true.
4. Remove both the braces mentioned in the third exercise and use `move`. What's the effect and why?

Threads

The standard library contains the `spawn` function for launching new threads:

```
fn spawn<F, T>(f: F) -> JoinHandle<T>
   where F: FnOnce() -> T,
             F: Send + 'static,
                T: Send + 'static
```

Here's what the function declaration says:

- `spawn` takes a parameter `f`, which implements the `FnOnce` trait. In other words, `f` is a closure.
- The `f` closure must implement the `Send` trait, which means that all its parameters must implement the `Send` trait.
- The `T` return type from the closure must also implement the `Send` trait.
- `spawn` returns `JoinHandle` with a value of the `T` type enclosed.

`Send` is a **marker trait**. This means that it does not implement any methods; it is just used as a mark that says that the value is safe to be sent between threads: most types, and specifically all primitive types, are. In addition, `Send` is automatically derived, that is, if all the types in a struct are `Send`, then the struct also is. In summary, almost every type you will see and have seen is `Send`. A notable exception is `Rc<T>`, which we have already mentioned as not being thread-safe.

`JoinHandle` that the spawn function returns can be used to synchronize the execution of different threads. Namely, we can choose to join our current thread with the other one by calling the `join` method on `JoinHandle`. Joining another thread means waiting for it to end.

Here's an example of a thread that captures its outer value, this time using `String`, which does not implement the `Copy` trait:

```
// threads-1.rs
use std::thread;

fn main() {
    let outside_string = String::from("outside");

    let thread = thread::spawn( move || {
        println!("Inside thread with string '{}'", outside_string);
    });

    thread.join();
}
```

Here, the `move` semantics hit with full power; because `outside_string` is used inside the closure, it is moved in there, immediately invalidating the original reference to it. The thread returns a reference to the same `String`, so we get it back to the original thread after calling `join`.

There are cases where the `move` semantics are inferred by the compiler but, in this case, we need to specify it. This is what the compiler says if we omit the move:

```
vegai@carbon ~/rustbook/7 » rustc threads-2.rs
error[E0373]: closure may outlive the current function, but it borrows `outside_
string`, which is owned by the current function
 --> threads-2.rs:6:33
  |
6 |     let thread = thread::spawn( || {
  |                                 ^^ may outlive borrowed value `outside_strin
g`
7 |         println!("Inside thread with string '{}'", outside_string);
  |                                                    -------------- `outside_s
tring` is borrowed here
  |
help: to force the closure to take ownership of `outside_string` (and any other
referenced variables), use the `move` keyword, as shown:
  |     let thread = thread::spawn( move || {

error: aborting due to previous error

vegai@carbon ~/rustbook/7 »                                                101 ↲
```

In general, it is good practice to always explicitly state `move` whenever we want it.

Exercises

1. Change the closure so that `outside_string` is returned from it.
2. Grab `outside_string` in the `main` thread. You get it from the `join` method.
3. After the aforementioned changes, what happens when you omit the `move` annotation from the closure and why?

Sharing the Copy types

Types that have the `Copy` type implemented can be trivially shared between threads, but, of course, the values get copied:

```
// sharing-immutables.rs
use std::thread;
use std::time;

fn main() {
    let mut num = 4;

    for _ in 1..10 {
        thread::spawn(move || {
            num += 1;
            println!("String is {}", num);
        });
    }

    thread::sleep(time::Duration::from_secs(1));
    println!("In main thread: num is now {}", num);
}
```

The output shows us that the numbers in the threads are separate copies:

```
vegai@carbon ~/rustbook/7 » ./sharing-immutables
String is 5
String is 5
String is 5
String is 5
String is 5
String is 5
String is 5
String is 5
String is 5
In main thread: num is now 4
vegai@carbon ~/rustbook/7 »
```

Nevertheless, if immutable values are all you need between your threads, and runtime space efficiency is not a concern, this is fine. The `Copy` types have an extremely efficient method of copying, after all.

Channels

Communication between threads can be implemented in a safe way by the use of channels. Rust's standard library has two kinds of channels defined in `std::sync::mpsc`:

- `channel`: This is an asynchronous, infinite buffer
- `sync_channel`: This is a synchronous, bounded buffer

The acronym **mpsc** refers to **multi producer, single consumer**. That is, these channels may have multiple writers but only a single reader. Both of these functions return a pair of generic values: a sender and a receiver. The sender can be used to push new things into the channel, while receivers can be used to get things from the channel. The sender implements the `Clone` trait while the receiver does not. This, paired with Rust's regular ownership system, allows the compiler to enforce that channels are really used in multi producer, single consumer mode.

Here's an example:

```
// channels-1.rs
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
    let tx_clone = tx.clone();
    tx.send(0);

    thread::spawn(move || {
        tx.send(1)
    });

    thread::spawn(move || {
        tx_clone.send(2)
    });

    println!("Received {} via the channel", rx.recv().unwrap());
    println!("Received {} via the channel", rx.recv().unwrap());
}
```

The corresponding output may look like this:

```
vegai@carbon ~/rustbook/7 » ./channels-1
Received 0 via the channel
Received 1 via the channel
vegai@carbon ~/rustbook/7 »
```

There's a small possibility that `2` gets sent to the channel before `1`, so the output could differ. Note that we didn't receive all the values we sent. The remaining values just get dropped as the program ends.

With the default asynchronous channels, the `send` method *never* blocks. This is because the buffer channel is infinite, so there's always space for more. Of course, it's not really infinite, just conceptually so: your system may run out of memory if you send gigabytes to the channel without receiving anything.

Synchronous channels have a sized buffer, and when it's full, the `send` method blocks until there's more space in the channel. The usage is otherwise quite similar to asynchronous channels:

```
// channels-2.rs
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::sync_channel(1);
    let tx_clone = tx.clone();
    tx.send(0);

    thread::spawn(move || {
        tx.send(1)
    });

    thread::spawn(move || {
        tx_clone.send(2)
    });

    println!("Received {} via the channel", rx.recv().unwrap());
    println!("Received {} via the channel", rx.recv().unwrap());
}
```

The synchronous channel size is `1`, which means that we can't have two items in the channel; the send would block in such a case. However, in the preceding code, we don't get blocks (at least, the long ones) as the two sending threads work in the background and the main thread gets to the receiving bit.

For both these channel types, the `recv` call blocks if the channel is empty.

There is unstable support for multiplexing `select` calls on channels, that is, reading from several different channels at the same time, and acting upon the first one that gets data. Select multiplexing is the preferred way to make high-performance single-threaded servers, so they will be quite interesting when they reach stability.

Exercises

1. Change the synchronous buffer size to `0` and see what happens. Figure out a way to make the code work with a zero buffer.
2. Add a third receive call to the asynchronous code. Witness the block.
3. Take a look at the state of `select`. At the time of writing, there's a macro, `std::select!`, which is a rather concise way of defining select loops. Give it a try.

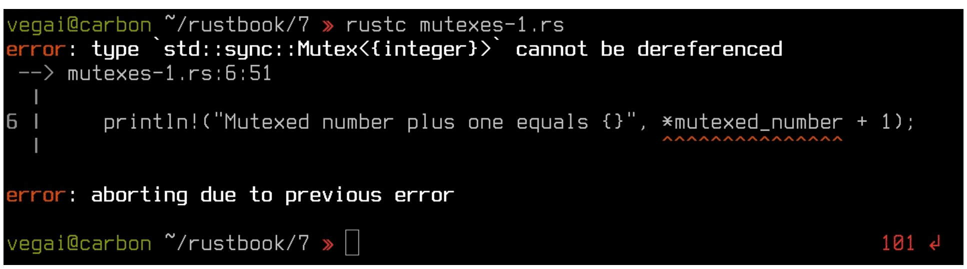
Locks and mutexes

When safe access to a shared resource is required, the access can be protected by the use of a mutex. **Mutex** is short for **mutual exclusion**, a widely used mechanism for ensuring that a piece of code is executed by only one thread at a time. It works by prohibiting access to a value from more than one thread at a time by locking the value.

Here's a piece of code that illustrates how this protection works at compile time:

```
// mutexes-1.rs
use std::sync::Mutex;


fn main() {
    let mutexed_number = Mutex::new(5);

    println!("Mutexed number plus one equals {}", *mutexed_number + 1);
}
```

This code fails to compile because we can get to the value protected by the mutex only by locking it first:



```
vegai@carbon ~/rustbook/7 » rustc mutexes-1.rs
error: type `std::sync::Mutex<{integer}>` cannot be dereferenced
 --> mutexes-1.rs:6:51
  |
6 |     println!("Mutexed number plus one equals {}", *mutexed_number + 1);
  |                                                   ^^^^^^^^^^^^^^^

error: aborting due to previous error

vegai@carbon ~/rustbook/7 »                                         101 ↲
```

This version works as expected:

```
// mutexes-2.rs
use std::sync::Mutex;

fn main() {
    let mutexed_number = Mutex::new(5);

    {
        let number = mutexed_number.lock().unwrap();
        println!("1 Mutexed number plus one equals {}", *number + 1);
    }
    let number = mutexed_number.lock().unwrap();
    println!("2 Mutexed number plus one equals {}", *number + 1);
}
```

The first lock gets released when the block ends because that's when the number variable gets dropped.

We'll need a bit more to be able to use this in a multithreaded context, though. If we just moved the mutex into a thread, we couldn't use it from any other thread. That's where reference counting can help us.

Exercises

1. Remove the inner block from the preceding code, compile, and run. What happens and why?
2. Try giving the mutex to multiple threads and using it from each. Why doesn't this work?

Atomic Rc

With computers, when something is atomic, it means that it is indivisible. In other words, it must happen as a whole or not at all. Here are a few examples:

- Databases: The **A** in **ACID** is short for **atomicity**. It means that a set of database operations either succeeds completely or not at all. Furthermore, outside observers (on another connection, for example) never see any of the intermediate states; the database immediately goes from not having any of the operations done to all of them having being done.
- Some file operations in Unix systems are atomic. For instance, the `mv` command that moves a file to another location is atomic. Like the previous example, this means that the move happens completely or not at all. No outside observer can see any intermediate steps, such as the file being at two places at the same time.

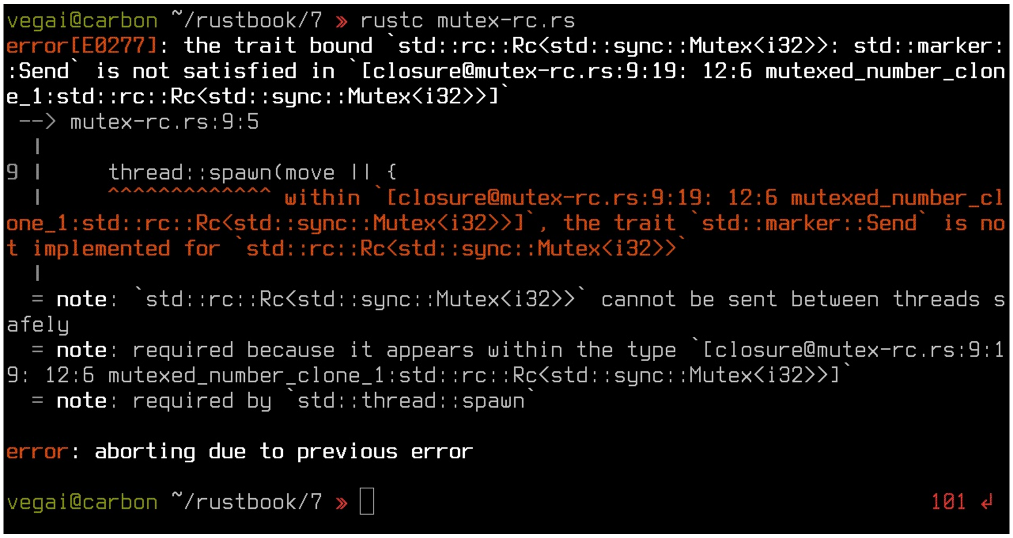
The power of atomic operations is that you can safely build on them without worrying about a certain class of concurrency issues, more specifically, the issues where one thread happens to barge on an operation that another thread is just performing.

We saw at the end of the previous section that mutex alone is not enough for sharing variables between threads. It just gives us mutability between threads. We'll need a way to share the same thing in several places, and a reference counted container is one answer.

You already know that the simple `Rc` type won't cut it, but for the sake of example, let's see what happens if we use it anyway:

```
// mutex-rc.rs
use std::sync::Mutex;
use std::thread;
use std::rc::Rc;

fn main() {
    let mutexed_number = Rc::new(Mutex::new(5));
    let mutexed_number_clone_1 = mutexed_number.clone();

    thread::spawn(move || {
        let number = mutexed_number_clone_1.lock().unwrap();
        println!("1 Rc/Mutexed number plus one equals {}", *number + 1);
    });
}
```

Thankfully, the compiler stops us from doing silly things:



Rust tries to close over the `mutexed_number_clone_1` variable and send it to the thread, but since `Rc` is not thread-safe, it does not implement the required `Send` trait, and we get a relatively nice error at compile time.

OK, enough horsing around. Let's bring out the atomics and try to make a proper mess of things this time. We'll launch 10,000... no... 1,000,000 threads and increment the value by one in each of them concurrently with proper reference counting and mutexes:

```
// mutex-arc.rs
use std::sync::Mutex;
use std::thread;
use std::sync::Arc;
use std::time;

const THREADS: u64 = 1_000_000;
const START_NUMBER: u64 = 1;

fn main() {
    let one_millisecond = time::Duration::from_millis(1);
    let one_second = time::Duration::from_millis(1000);

    let mutexed_number = Arc::new(Mutex::new(START_NUMBER));
    let mutexed_number_2 = mutexed_number.clone();

    thread::spawn(move || {
        for _ in 1..THREADS {
            let mutexed_number_clone = mutexed_number.clone();
            thread::spawn(move || {
                thread::sleep(one_millisecond);
                let mut number = mutexed_number_clone.lock().unwrap();
                *number += 1;
            });
        }
    });

    loop {
        thread::sleep(one_second);
        let number = mutexed_number_2.lock().unwrap();
        if *number != START_NUMBER + THREADS - 1 {
            println!("Not there yet, number is {}", *number);
        } else {
            println!("Got there! Number is {}", *number);
            break;
        }
    }
}
```

So, this piece of code essentially does the following:

1. Allocates a number inside an atomically reference counted mutex.
2. Starts a thread that starts a million threads, each incrementing the value by one.
3. Meanwhile, in the main thread, it inspects the value every second and exits when it reaches a goal.

And it works reliably too, although the overhead of mutex locking makes this implementation rather slow. Here's a sample run on a fairly modern Intel i7 processor:

```
vegai@carbon ~/rustbook/7 » time ./mutex-arc
Not there yet, number is 34351
Not there yet, number is 66538
Not there yet, number is 94204
Not there yet, number is 129404
Not there yet, number is 158148
Not there yet, number is 187417
Not there yet, number is 213452
Not there yet, number is 245594
Not there yet, number is 274302
Not there yet, number is 301901
Not there yet, number is 326312
Not there yet, number is 356794
Not there yet, number is 386599
Not there yet, number is 415977
Not there yet, number is 451629
Not there yet, number is 483627
Not there yet, number is 512690
Not there yet, number is 546687
Not there yet, number is 577471
Not there yet, number is 610927
Not there yet, number is 648616
Not there yet, number is 679141
Not there yet, number is 708716
Not there yet, number is 743982
Not there yet, number is 782080
Not there yet, number is 812135
Not there yet, number is 846964
Not there yet, number is 875546
Not there yet, number is 903947
Not there yet, number is 939215
Not there yet, number is 973633
Got there! Number is 1000000
./mutex-arc  19.83s user 57.33s system 240% cpu 32.081 total
vegai@carbon ~/rustbook/7 »
```

Exercises

1. Fiddle with the `move` declarations again. Consider the error messages given by the compiler.
2. Are all the `clone()` calls necessary? Try to remove a couple.
3. The threads completed on my machine at the speed of about 30,000-40,000 per second. Is that fast?

The final exercise

Here's one final exercise for the chapter:

1. Take a peek at the official library documentation:

- https://doc.rust-lang.org/std/thread/
- https://doc.rust-lang.org/std/sync/struct.Arc.html
- https://doc.rust-lang.org/stable/std/sync/mpsc/
- https://doc.rust-lang.org/stable/std/sync/struct.Mutex.html

2. Take the game code and implement at least two creatures moving in on the map in their own, independent threads.

Summary

Rust can easily give you safe and efficient concurrent programming with a rather minimal (although not zero) runtime cost. The safety comes from a combination of the language's memory safety trio (ownership, borrowing, and lifetimes) and the standard library that builds upon those.

During this chapter, you learned to launch threads with the standard Rust library and how `Copy` and `move` types work in the context of parallelism. We covered channels, the atomic reference counting type `Arc`, and how to use `Arc` with mutexes.

In the next few chapters, we'll dive into metaprogramming, starting with macros.

Macros

Rust has support for several forms of metaprogramming, which means writing programs that write programs. It can be a very powerful technique that helps surpass limitations of the language itself. It's a rather challenging way to program, however, and requires much more care and consideration than writing regular functions.

The oldest and most stable form of metaprogramming in Rust is syntactic macros. We'll cover those in this chapter.

This chapter will cover the following topics:

- Introduction to metaprogramming
- Dissecting `println!`
- Macro keywords
- Repeating constructs
- Building our own macros

Introduction to metaprogramming

In an ideal and simplified form, programming consists of two clearly separated things: program code and data. Once you are finished with your code, it is like carved in stone, non-malleable.

Metaprogramming means writing programs that write programs. It varies wildly how different programming languages do this. For instance, C has a preprocessor that reads specific tags starting with # and expands them before handing the result to the actual compiler. In C, those expansions are quite free-form; they are just simple text transformations without much safety. Specifically, macros written in C (and a few other languages) are not *hygienic*: they can refer to variables defined anywhere, as long as those variables are in scope at the macro invocation site. For instance, here's a macro that switches two parameters:

```
/* switcher.c */
#include <stdio.h>

#define SWITCH(a, b) { temp=b; b=a; a=temp; }

int main() {
    int x=1;
    int y=2;
    int temp=3;

    SWITCH(x, y);
    printf("x is now %d y is now %d temp is now %d\n", x, y, temp);
}
```

Since the macro invocation just replaces text, `SWITCH` using the temp variable works just fine. This unhygienic nature makes macros dangerous and brittle though; they can easily make a mess unless special precautions are taken. We're stressing the concept of hygiene here because Rust macros *are* hygienic and, also, a bit more structured than just simple string expansions.

Rust has several types of metaprogramming and a few more upcoming. The most stable forms are syntactic macros, also called **macros-by-example**, due to a paper by Kohlbecker and Wand that introduced the technique in 1986. They are defined by another macro, called `macro_rules!`. These macros are represented in the compiler's abstract syntax tree output along with other pieces of program code. This implies that these macros cannot be used everywhere in your code, but only in place of methods, statements, expressions, patterns, and items. It further implies that the parameters to macros must be well-formed within the AST; they must be well-formed token trees.

The other form of syntax extension is called **procedural macros** or **compiler plugins**. These are much more powerful than syntactic macros, allowing any custom Rust code to be run along the compilation process. The price is that they are way more complex to implement and rely on the compiler internals so much that they will probably never be fully implemented in stable Rust.

A limited form of procedural macros is being built, called **macros 1.1**. The motivation for this is that procedural macros are being effectively used by many high-profile libraries (such as the popular serialization framework, Serde), essentially making them work properly only in nightly Rust. It was discovered, however, that most of these libraries use only a limited subset of the whole procedural macro machine. Macros 1.1 tries to implement that subset, making it possible to use those libraries with full power in stable Rust. The work is already well on its way, and might end up in a stable release before the end of 2016.

We'll cover both forms of procedural macros in the next chapter.

To make the future even more fascinating, the macros-by-example system is also being revamped, but that work is expected to take a longer time to finish.

Metaprogramming, in general, and macros, in particular, are efficient tools but also dangerous; they can easily make the code more brittle, and less easy to read and debug. Therefore, these techniques should be used only after the more stable ones (such as functions, traits, and generics) have been considered and deemed insufficient. One instance that you've seen several times already is the `println!` macro. It has been implemented as a macro because it allows Rust to check at compile time that its arguments are valid. If `println!` were a regular function, that would not be possible. Consider the following example:

```
println("The result of 1+1 is {}");
println!("The result of 1+1 is {}");
```

As you already know, the second form will fail at compile time because it's missing an argument that matches the format string. This compile-time check could not be made in Rust for the function. Furthermore, Rust does not support functions with a variable number of arguments.

Dissecting println!

Let's start by diving into the deep end: we'll take our old friend `println!` apart. Here is its definition from the standard library, without the actual code body:

```
macro_rules! println {
    ($fmt:expr) => (print!(concat!($fmt, "\n")));
    ($fmt:expr, $($arg:tt)*) => (print!(concat!($fmt, "\n"), $($arg)*));
}
```

`macro_rules!` creates new macros. Its first parameter is the name of the new macro and then it follows the pattern-matched code bodies. Things that start with `$` (such as `$fmt:expr` in the preceding definition) get assigned whatever free-form string is in its place, and everything else (such as the comma in the preceding definition) is parsed verbatim. In the case of `println`, there are two matches:

- `($fmt:expr)` matches a single expression, which goes into the variable `$fmt`.
- `($fmt:expr, $($arg:tt)*)` matches a single expression, followed by a comma, followed by zero or more arguments. The arguments are stored in $arg.

Both `expr` and `tt` are special keywords, short for **expression** and **token tree**. We'll see later what they mean specifically. Let's take an example, an invocation of `println!`, that matches the second pattern. The first case will almost be the same, anyway. Here's where we'll begin:

```
println!("Help, I'm {} a {}!", "inside", "macro")
```

The patterns are tried in order, and the first one that matches gets selected. The first pattern does not match because our parameter to the macro has a comma after the first expression but the pattern does not. The second one will match just fine, so the whole expression expands to:

```
print!(concat!("Help, I'm {} a {}!", "\n"), "inside", "macro")
```

We'll need to take a look at the definition of `concat!` to see what happens next. Here it is, taken straight from the source code:

```
macro_rules! concat { ($($e:expr),*) => ({ /* compiler built-in */ }) }
```

OK, that doesn't help much. We can see that it takes an arbitrary number of expressions separated by commas, but its implementation is welded into the compiler. We'll just have to trust the documentation for `concat!`, which states that it concatenates all its literal parameters into a static string. This means that the next expansion becomes:

```
print!("Help, I'm {} a {}!\n", "inside", "macro")
```

The definition of the `print!` macro is:

```
macro_rules! print {
    ($($arg:tt)*) => ($crate::io::_print(format_args!($($arg)*)));
}
```

Further down the rabbit hole we go! Our expression becomes:

```
$crate::io::_print(format_args!("Help, I'm {} a {}!\n", "inside", "macro"))
```

We're almost there, since `format_args!` is again a compiler built-in:

```
macro_rules! format_args { ($fmt:expr, $($args:tt)*) => ({
        /* compiler built-in */
    }) }
```

Now we hit the bottom as far as the macro expansion goes. The `format_args!` macro is what all the other macros in the standard library needing formatting capabilities end up calling. It returns the formatted arguments in an `std::fmt::Arguments` type. Essentially, it will be a safely parsed version of the string, *Help, I'm inside a macro!*, but wrapped in that type.

The `format_args!` macro, and thus every macro that uses it, does a form of syntax checking. For instance, if we try to use `println!` with a wrong number of parameters, we'll get an error:

```
fn main() {
    println!("I have two parameters {} {}, but am only supplied one", 1);
}
```

The compiler complains:

```
vegai@carbon ~/rustbook/8 ±master⚡ » rustc faulty-println-1.rs
error: invalid reference to argument `1` (there is 1 argument)
 --> faulty-println-1.rs:2:5
  |
2 |     println!("I have two parameters {} {}, but am only supplied one", 1);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: this error originates in a macro outside of the current crate

error: aborting due to previous error

vegai@carbon ~/rustbook/8 ±master⚡ »                                    101 ↲
```

The compiler error does not exactly pinpoint what we did wrong but, at least, tells us that something is wrong. Also, when we have too many arguments, we get a compile-time error:

```
fn main() {
    println!("I have two parameters {} {}, but am supplied with three", 1, 2, 3);
}
```

The corresponding compiler output is:

```
vegai@carbon ~/rustbook/8 ±master⚡ » rustc faulty-println-2.rs
error: argument never used
 --> faulty-println-2.rs:2:79
  |
2 |     println!("I have two parameters {} {}, but am supplied with three", 1, 2
, 3);
  |
  ^

error: aborting due to previous error

vegai@carbon ~/rustbook/8 ±master⚡ »                                    101 ↲
```

The whole macro expansion process, with all the related error checks, happens at compile time.

There is nothing special and privileged about `println!` aside from the few compiler built-ins. The same mechanisms are available to you, and you'll see next how to build your own macros.

Exercises

1. Why did `println!` need two patterns?
2. Why is `println!` a macro instead of just a function?
3. Think about your second favorite compiled programming language. How does it do the same checks that `println!` does via a macro? Is either choice superior?

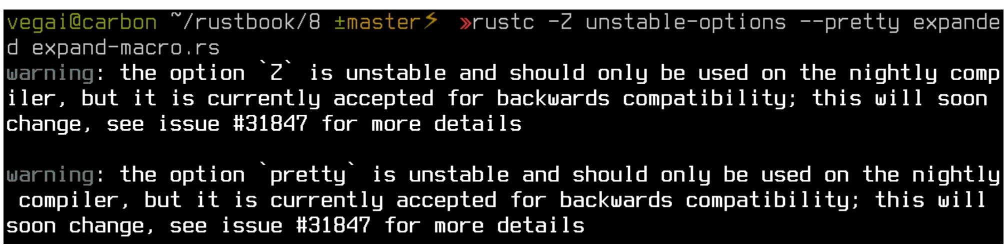
Debugging macros

Before we head over to macro keywords and building our own macros, let's take a look at what to do when our macros don't work.

The first technique is to ask the compiler to show us the code after the macro expansion has been done. Here's our macro that either takes nothing or a block. As a bonus, let's see what `println!` really becomes:

```
// expand-macro.rs
macro_rules! meep {
     () => (nothing);
     ($block:block) => ( make($block); );
}

fn main() {
    meep!();
    meep!({silly; things});
    println!("Just to show how fun println! really gets");
}
```

The expansion is requested from the compiler by using the parameter `--pretty expanded`. This is an unstable feature, but at the time of writing this book, it is still kind of supported by the stable compiler. That might change soon, as you will see from the compiler output, split into three portions:



```
vegai@carbon ~/rustbook/8 ±master⚡ »rustc -Z unstable-options --pretty expande
d expand-macro.rs
warning: the option `Z` is unstable and should only be used on the nightly comp
iler, but it is currently accepted for backwards compatibility; this will soon
change, see issue #31847 for more details

warning: the option `pretty` is unstable and should only be used on the nightly
 compiler, but it is currently accepted for backwards compatibility; this will
soon change, see issue #31847 for more details
```

Here is the error portion of the compiler output, showing unresolved names in both the macro code and its invocations:

```
error[E0425]: cannot find value `nothing` in this scope
 --> expand-macro.rs:2:12
  |
2 |     () => (nothing);
  |            ^^^^^^^ not found in this scope
...
7 |     meep!();
  |     -------- in this macro invocation

error[E0425]: cannot find function `make` in this scope
 --> expand-macro.rs:3:25
  |
3 |     ($block:block) => ( make($block); );
  |                         ^^^^ not found in this scope
...
8 |     meep!({silly; things});
  |     ----------------------- in this macro invocation

error[E0425]: cannot find value `silly` in this scope
 --> expand-macro.rs:8:12
  |
8 |     meep!({silly; things});
  |     -------^^^^^-----------
  |     |      |
  |     |      not found in this scope
  |     in this macro invocation

error[E0425]: cannot find value `things` in this scope
 --> expand-macro.rs:8:19
  |
8 |     meep!({silly; things});
  |     --------------^^^^^^---
  |     |             |
  |     |             not found in this scope
  |     in this macro invocation
```

Finally, this is the expanded macro output of `println!`:

```
#![feature(prelude_import)]
#![no_std]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std as std;

fn main() {
    nothing;
    make({ silly; things });
    ::io::_print(::std::fmt::Arguments::new_v1({
                                                   static __STATIC_FMTSTR:
                                                          &'static [&'static st
r]
                                                          =
                                                       &["Just to show how fun
println! really gets\n"];
                                                   __STATIC_FMTSTR
                                               }, &match () { () => [], }));
}
error: aborting due to 4 previous errors

vegai@carbon ~/rustbook/8 ±master⚡ »                                    101 ↲
```

So, we can see that as long as our macro and its invocation do not break the parsing rules too badly, we get to see the expanded code. Note how the compiler is nice enough to point to the actual macro invocation instead of the expanded code as the source of the error:

```
expand-macro.rs:8:12: 8:17 error: unresolved name `silly` [E0425]
expand-macro.rs:8     meep!({silly; things});
```

In case you need to do this for a whole Cargo project, this feature can also be used via a Cargo wrapper called `cargo-expand`. We won't go there, though, as it is essentially the same as the preceding invocation.

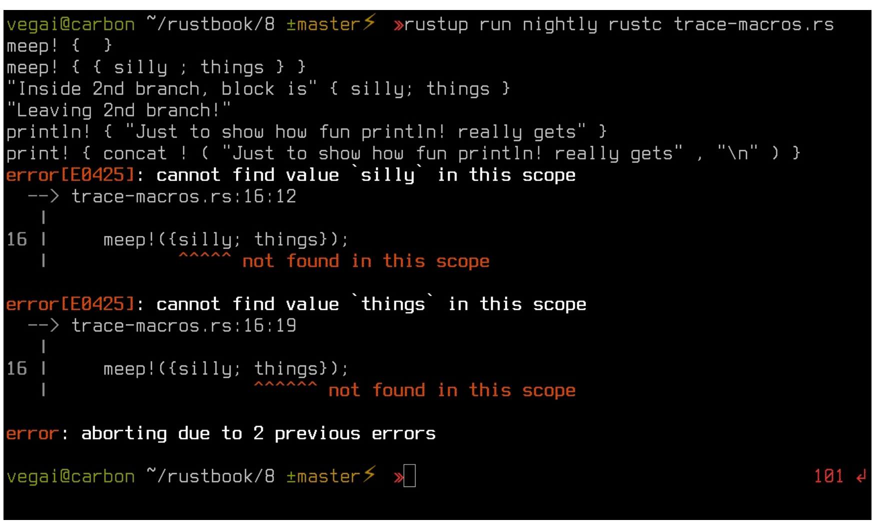
Trace macros are another way to debug macros. They are feature gated, which means that they can be used only in the nightly Rust side. There are two such macros:

- `trace_macros!` takes a Boolean and globally turns macro tracing on or off
- `log_syntax!` simply outputs all its arguments at compile time

Here's `expand-macro.rs` from before, modified to use both `trace_macros!` and `log_syntax!`, and with the unresolved names fixed:

```
// trace-macros.rs
#![feature(trace_macros, log_syntax)]

trace_macros!(true);

macro_rules! meep {
    () => ();
    ($block:block) => (
        log_syntax!("Inside 2nd branch, block is" $block);
        ($block);
        log_syntax!("Leaving 2nd branch!");
    );
}
```

The `main` function stays the same, so there's no need to repeat it. Here's what the nightly compiler gives us when we compile this:



As we can see, `log_syntax!` really outputs its parameters verbatim, with the quotes and all. Also, note how the `println!` expansion is a tad nicer compared with the output given by the `--pretty expanded` compiler output.

The trace macros are probably the best tool for debugging macros at the moment and for any foreseeable future. Therefore, be prepared to use nightly Rust if you decide to become a serious macro programmer.

Macro keywords

Let's start our journey of writing our own macros by checking out the list of different recognized keywords macro patterns may have:

- `block`: This is a sequence of statements
- `expr`: This is an expression
- `ident`: This is an identifier
- `item`: This is an item
- `meta`: This is a meta item
- `pat`: This is a pattern
- `path`: This is a qualified name
- `stmt`: This is a statement
- `tt`: This is a token tree
- `ty`: This is a type

block

We have already used `block` in the debugging example. It matches any sequence of statements, delimited by braces, such as what we were using before:

```
{ silly; things; }
```

This block has the statements `silly` and `things`.

expr

This matches a single expression, such as:

- `1`
- `x+1`
- `if x==4 { 1 } else { 2 }`

Notably, it does not match statements like `let x=1`, since it's not a single expression.

ident

Identifiers are any Unicode strings that are not keywords (such as `if` or `let`). As an exception, the underscore character alone is not an identifier in Rust. Examples of identifiers:

- `x`
- `longIdentifier`
- `SomeSortOfAStructType`

item

Top-level definitions are called **items.** These include functions, use declarations, type definitions, and so on. Here are some examples:

- `use std::io;`
- `fn main() { println!("hello") }`
- `const X: usize = 8;`

These do not have to be one-liners, of course. The `main` function would be a single item, even if it spanned several lines.

meta

The parameters inside attributes are called meta items, which are captured by `meta`. Attributes themselves look as follows:

- `#![foo]`
- `#[foo]`
- `#[foo(bar)]`
- `#[foo(bar="baz")]`

Meta items are the things inside the brackets. So, for each of the preceding attributes, the corresponding meta items are as follows:

- `foo`
- `foo`
- `foo(bar)`
- `foo(bar="baz")`

pat

Match expressions have **patterns** on the left-hand side of each match, which `pat` captures. Here are some examples:

- `1`
- `"x"`
- `t`
- `*t`
- `Some(t)`
- `1 | 2 | 3`
- `1 ... 3`
- `_`

path

Paths are qualified names, that is, names with a namespace attached to them. They're quite similar to identifiers, except that they allow the double colon. Here are some examples:

- `foo`
- `foo::bar`
- `Foo`
- `Foo::Bar::baz`

stmt

Statements are much like expressions, except that more patterns are accepted by `stmt`. The following are some examples:

- `foo`
- `1`
- `1+2`
- `let x = 1`

Especially, the last one wouldn't be accepted by `expr`.

tt

The `tt` keyword captures a single token tree. A token tree is either a single token (such as `1`, `+`, or `"foo bar"`) or several tokens surrounded by any of the braces, `()`, `[]`, or `{}`. The following are some examples:

- `foo`
- `{ bar; if x == 2 { 3 } else { 4 }; baz }`
- `{ bar; fi x == 2 ( 3 ] ulse ) 4 {; baz }`

As you can see, the insides of the token tree do not have to make semantic sense; they just have to be a sequence of tokens. Specifically, what does not match this are two or more tokens not enclosed in braces (such as `1 + 2`).

ty

The `ty` keyword captures things that look like types. Here are some examples:

- `u32`
- `u33`
- `String`
- `Strong`

No semantic checking that the type is actually a type is done in the macro expansion phase, so `"u33"` is accepted just as well as `"u32"`.

Repeating constructs

We just need one additional mechanism for writing our macros: a way to model repeating patterns. We've seen this in the `vec!` macro before:

```
vec![1, 2, 3]
```

This would create and return a new vector with three elements. Let's see how `vec!` does it. Here's its `macro_rules!` definition:

```
macro_rules! vec {
    ($elem:expr; $n:expr) => ($crate::vec::from_elem($elem, $n));
    ($($x:expr),*) => (<[_]>::into_vec(box [$($x),*]));
    ($($x:expr,)*) => (vec![$($x),*])
}
```

Let's ignore the right-hand side and focus on the last two patterns:

```
$($x:expr),*
$($x:expr,)*
```

The repeating pattern matches follow this pattern: `$($var:type)`. There may be any number of string literals sprinkled in there, depending on what you want your macro invocation to look like. In `vec!`, the string literal is the comma character. In the first match, the comma character is *outside* the repeating match. This is the typical case and will match sequences such as `1, 2, 3`. However, it won't match a sequence with a trailing comma, such as `1, 2, 3,`. Such a sequence makes more sense when formatted like this:

```
vec![
    1,
    2,
    3,
];
```

It frees the user of the macro from having to remember to remove the comma from the last item. The second pattern captures the comma *inside* the repeating match, which allows the preceding form. However, that pattern does not match `1, 2, 3`, hence we need them both.

The repeating construct requires either of the following two qualifiers, familiar from regular expressions:

- `*` means that the repeat needs to happen zero or more times
- `+` means that the repeat needs to happen one or more times

The patterns in `vec!` use `*`, which implies that `vec![]` is an allowed invocation of the macro. With `+`, it would not be.

Let's now look at how the repeats work on the right-hand side. There are two ways of using them. The `vec!` macro does not need to handle each of the captured elements of the sequence itself, so it just forwards them on using an identical syntax:

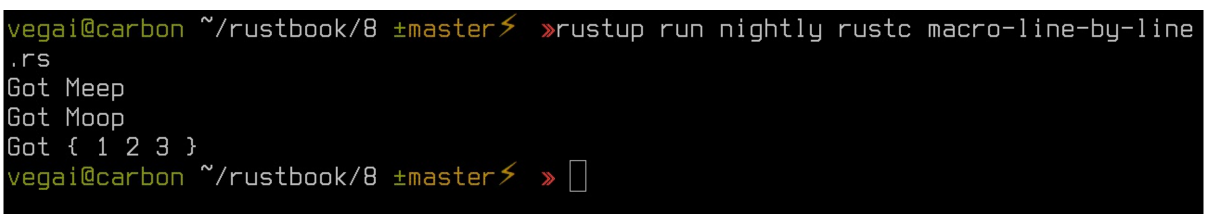
```
($($x:expr),*) => (<[_]>::into_vec(box [$($x),*]));
```

The only difference between the declaration on the left-hand side and the usage on the right-hand side is that the right-hand side does not include the type (`expr`) of the variable.

The second way of usage is to go through the elements one by one. The syntax for this is similar: we enclose the code we want to execute for each element by `$()` and qualify again. Here's a macro that outputs all the elements it has been given at compile time:

```
#![feature(log_syntax)]

macro_rules! m1 {
    ($($x:tt),*) => {
        $(
            log_syntax!(Got $x);
        )*
    };
}

fn main() {
    m1!(Meep, Moop, { 1 2 3 });
}
```

Note that we are capturing token trees in the macro pattern; compiling this code gives us the following output:



Now, we have covered pretty much everything needed to do metaprogramming via macros-by-example. Let's take a look at an example macro.

Example - an HTTP tester

Let's see how the macro expansion functions by working through a custom macro with overlapping patterns. This macro implements a small language, designed for describing simple HTTP `GET`/`POST` tests using the **hyper** library. Here's a sample of what the language looks like without the enclosing macro calls:

```
http://google.com GET => 302
http://google.com POST => 411
```

The first line makes a `GET` request to Google, and expects a return code, `302` (**Moved**). The second one makes a `POST` request to the same place, and expects a return code `411` (**Length Required**). This is very simplistic but quite sufficient for our purposes.

Hyper is Rust's de facto standard HTTP library, which supports both server and client operations. We're interested in the client portion. Since it is a library crate, we'll need to build a complete Rust application with Cargo, so we can declare the dependency. We'll call our program `http-tester`. Here's its `Cargo.toml`:

```
[package]
name = "http-tester"
version = "0.1.0"
authors = ["Vesa Kaihlavirta <vegai@iki.fi>"]

[dependencies]
hyper="0.9.*"
```

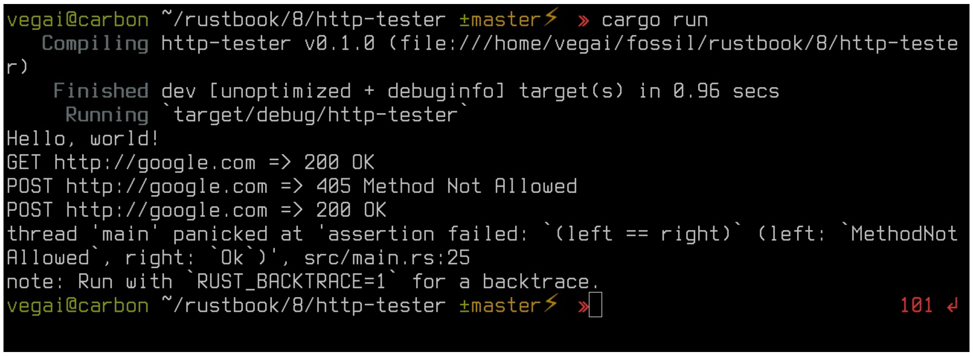
And here's `src/main.rs`:

```
extern crate hyper;

use hyper::client::Client;
use hyper::status::StatusCode;

macro_rules! http_test {
    ($url:tt GET => $code:expr) => {
        let client = Client::new();
        let res = client.get($url).send().unwrap();
        println!("GET {} => {}", $url, $code);
        assert_eq!(res.status, $code);
    };
    ($url:tt POST => $code:expr) => {
        let client = Client::new();
        let res = client.post($url).send().unwrap();
        println!("POST {} => {}", $url, $code);
        assert_eq!(res.status, $code);
    };
}

fn main() {
    println!("Hello, world!");
    http_test!("http://google.com" GET => StatusCode::Ok);
    http_test!("http://google.com" POST => StatusCode::MethodNotAllowed);
    http_test!("http://google.com" POST => StatusCode::Ok);
}
```

As you can see, we had to compromise on the syntax somewhat for a few reasons:

- The URL is a string instead of just a free-form identifier. This is due to the fact that we don't have complete freedom with macros-by-example.
- The hyper library prefers to use the StatusCode enum for the HTTP return codes, so we're just using that here.

Here's the output of running this program:



Take a moment to think what the benefit of using a macro here is. This could be implemented as a library call almost as well, but the macro has a few benefits, even in this basic form. One is that the HTTP verb is checked at compile time, so you're guaranteed that a successfully compiled program does not try to make a `POST` call, for instance. Also, we were able to implement this as a mini language, with the `=>` identifier signaling the separation between the command on the left-hand side and the expected return value on the right-hand side.

Note that Rust needs to read the macro input past the first match (`$url:tt`) and only at the first letter after the space (which is either the first letter of `GET` or the first letter of `POST` for any valid input) can it continue with just one of the pattern matches.

Exercises

1. Write a macro that takes an arbitrary number of elements and outputs an unordered HTML list in a literal string. For instance, `html_list!([1, 2]) => <ul><li>1/<li><li>2</li></ul>`.
2. Write a macro that accepts the following language:

```
language = HELLO recipient;
recipient = <String>;
```

   For instance, the following strings would be acceptable in this language:

```
HELLO world!
HELLO Rustaceans!
```

   Make the macro generate code that outputs a greeting directed to the recipient.

3. Write a macro that takes either of these two arbitrary sequences:

```
1, 2, 3
1 => 2; 2 => 3
```

   For the first pattern, it should generate a vector with all the values. For the second pattern, it should generate a HashMap with key-value pairs.

Summary

In this chapter, we gave a short introduction to metaprogramming, and took a cursory look at the many kinds of metaprogramming Rust supports and will support. The most supported form is macros-by-example, which fully works in stable Rust. They are defined by the `macro_rules!` macro. Macros-by-example work in the abstract syntax tree level, which means that they do not support arbitrary expansions but require that the macro expansions are well-formed in the AST.

We looked at ways to debug macros, first by asking our compiler to output the fully expanded form (`--pretty expanded`). The second way to debug macros, via the `macros log_syntax!` and `trace_macros!`, requires the nightly compiler but is quite a lot more convenient.

Macros are a powerful tool but not something that should be used lightly. Only when the more stable mechanisms such as functions, traits, and generics do not suffice should we turn to macros.

The next chapter shall cover the more powerful technique of procedural macros.

Compiler Plugins

The other form of metaprogramming, compiler plugins, enable arbitrary Rust code to be run at compile time. This feature is the only one in this book that has not hit the stable version of Rust yet (and perhaps never will in this form), but it is still quite widely used and an important differentiating feature that should be covered.

Expect a level of thickness and uncertainty in this chapter; compiler plugins are a challenging feature and their Rust implementation is still quite unstable. The concepts should be fairly stable, but the implementation details may very well be different even just a year after the publication date.

In this chapter, we will cover the following topics:

- Minimal compiler plugin
- Cargo integration
- Code generation
- Aster (a library for creating abstract syntax trees)
- Linter plugins
- Macros 1.1
- Macros 2.0

Basics of compiler plugins

The macros we saw in the previous chapter transformed a syntax into another syntax at compile time. That is a neat tool since it allows many forms of compile-time operations while providing a clean and stable API. However, there are a great number of desirable compile-time things we cannot do just by text manipulations. Here are some:

- Extra code validation checks (lints).
- Compile-time validations: database schema checks, hostname validations.
- Generating code depending on the environment. For example, creating data models from live database tables, filling in data structures from the environment, optimizing runtime performance by computing expensive things at compile time, and so on.

Whereas macros-by-example from the previous chapter had a special syntax for creating macros, with pattern matching being the central structure, compiler plugins are just Rust functions that we include in the compilation process. What makes them special is that they take predefined forms that describe code blocks as AST and return arbitrary blocks of code, again codified as AST.

Here's a list of currently existing libraries that make extensive use of compiler plugins:

- `diesel` is a safe, extensible database object-relational mapper. It uses compiler plugins to generate and validate database-interfacing model code by connecting to the actual database at compile time and reading the live schemas. At the time of writing this book, Diesel had just been ported to using macros 1.1, so by the time you read this, it is probably fully working on stable Rust.
- `serde` is a general serialization/deserialization framework. It uses compiler plugins to add new derive keywords, `Serialize` and `Deserialize`, which can generate the serialization and deserialization code from structs for several dozen different data formats. The framework makes it possible to fluently add new formats as separate libraries. Also, Serde will be targeting macros 1.1, which means that it should be fully usable on stable Rust already.
- `rust-clippy` extends the Rust compiler with hundreds of additional checks. This only works on nightly Rust, and macros 1.1 is not expected to be enough for these.

Compiler plugins are a highly unstable feature, so some parts of this chapter may fall out of date much sooner than other chapters. The concepts should be fairly stable, however, and the official Rust book nightly version should be expected to always contain the latest details at https://doc.rust-lang.org/nightly/book/compiler-plugins.html.

Let's dive in by first building a very simple compiler plugin.

The minimal compiler plugin

Compiler plugins are built as separate crates and they include a plugin registration function. The crate is then linked into the main application in a special way. Here's a compiler plugin that adds a new macro. The macro, when called in an application, prints a greeting at compile time:

```
// simplest-compiler-plugin.rs
#![feature(plugin_registrar, rustc_private)]

extern crate syntax;
extern crate rustc_plugin;

use syntax::tokenstream::TokenTree;
use syntax::ext::base::{ExtCtxt, DummyResult, MacResult};
use syntax::ext::quote::rt::Span;
use rustc_plugin::Registry;

fn hello(_: &mut ExtCtxt, sp: Span, _: &[TokenTree])
         -> Box<MacResult + 'static> {
    println!("Hello!");
    DummyResult::any(sp)
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_macro("hello", hello);
}
```

Quite a screenful for the simplest case, but such is the plugin writer's life. Most of this boilerplate comes from the fact that a function that we declare as a compiler plugin needs to comply exactly with the function definition. Let's investigate that:

```
fn hello(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
         -> Box<MacResult + 'static>
```

`ExtCtxt`, short for extension context, contains various methods for controlling the execution of an extension. For instance, we can call the `span_err` on the context to signal that a macro was not successful and we wish to abort compilation as soon as it is sensible.

`Span` refers to a region of code used internally for making error messages better. Many of the other plugin functions (such as `span_err` just mentioned previously) require a `Span` in order to display where an error happened in code.

The `args` parameter contains references to the arguments given to this macro as `TokenTree` structures.

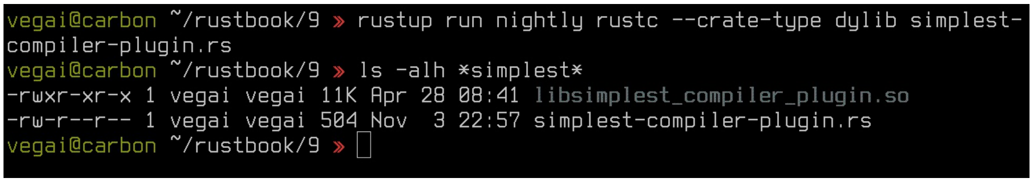
The return type is a boxed `MacResult` structure, which contains abstracted forms of Rust code. The end result of a macro invocation is an abstract structure of Rust code, inserted in place of the macro call. In our example, we insert `DummyResult`, an empty result that's usually used for error cases.

Other parts of the code are fairly regular: we declare our usage of unstable features on the first line, include needed crates, and pull in structs we refer to. As the last bit, we register our function as a macro.

Next, we'll need to build this as a separate crate. There are two ways to do that: first, manually via `rustc`, second via Cargo. We'll use `rustc` now and Cargo a bit later. The command to do this is as follows:

```
rustup run nightly rustc --crate-type dylib simplest-compiler-plugin.rs
```

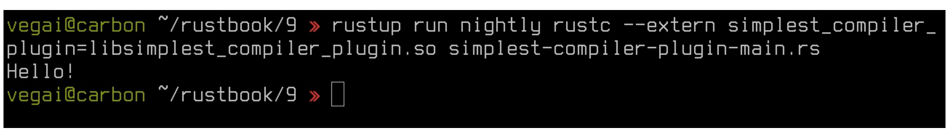
This invocation will output a dynamic library for us, as we can see here:



Next, write a `main` function that uses this plugin. This is much simpler; we just need to add the plugin to our program with a crate attribute:

```
// simplest-compiler-plugin-main.rs
#![feature(plugin)]
#![plugin(simplest_compiler_plugin)]

fn main() {
    hello!();
}
```

Lastly, we'll need to tell `rustc` to link the plugin along with the `main` program. This is done by the `--extern` parameter of `rustc` by giving it a logical name for the plugin (which is referred to in `main`) and a full path of the dynamic library built earlier. Here's how the compilation should look:



We get the greeting from the plugin at compile time!

Building a compiler plugin via Cargo

To compile the previous example with Cargo, we'll just need to include the crate containing the compiler plugin in `Cargo.toml` and define it as a plugin. Cargo will handle the rest. Modifying the preceding example into a Cargo project, we get this directory structure:

```
vegai@carbon ~/rustbook/9 » tree compiler-plugin
compiler-plugin
├── Cargo.lock
├── Cargo.toml
└── src
    ├── main.rs
    └── simplest_compiler_plugin.rs

1 directory, 4 files
vegai@carbon ~/rustbook/9 »
```

We'll still need to tell Cargo about the plugin. The lib section of `Cargo.toml` would look as follows:

```
// compiler-plugin/Cargo.toml
[lib]
name="simplest_compiler_plugin"
plugin=true
```

After this change, building with Cargo works and we can see the plugin in glorious action:

```
vegai@carbon ~/rustbook/9/compiler-plugin ±master⚡ » rustup run nightly cargo b
uild
   Compiling compiler-plugin v0.1.0 (file:///home/vegai/fossil/rustbook/9/compil
er-plugin)
Hello!
    Finished dev [unoptimized + debuginfo] target(s) in 0.83 secs
vegai@carbon ~/rustbook/9/compiler-plugin ±master⚡ »
```

Now that we are able to build a simple compiler plugin, let's try a bit more interesting example.

Code generation as a workaround

As mentioned before, compiler plugins are not a stable feature, but they are useful enough so that people want to use their features. There's a workaround that works already in the stable branch: code generation via a library called libsyntex. It works by essentially bundling the whole Rust compiler in a library and using it in order to implement the compiler plugins. This is not a drop-in replacement for the compiler plugins as they are in the nightly Rust; many small changes to the code base need to be done in order for the code generation method to work.

In practice this works by moving any code modules with compiler plugin functionality in a template file ending with .in. A separate build script (usually called build.rs) is then used before the compilation step to generate the same module with the extension code expanded. Let's try that with our hello example.

First, we'll need to reorganize the code tree into two Cargo projects, one holding the plugin and another using it. Here's how the tree will look:

The hello_plugin library needs to be changed slightly:

```
// compiler-plugin/stable/hello_plugin/src/lib.rs
extern crate syntex;
extern crate syntex_syntax;

use syntex_syntax::tokenstream::TokenTree;
use syntex_syntax::ext::base::{ExtCtxt, DummyResult, MacResult};
use syntex_syntax::ext::quote::rt::Span;
use syntex::Registry;

fn hello<'cx>(_: &'cx mut ExtCtxt, sp: Span, _: &[TokenTree])
         -> Box<MacResult + 'cx> {
    println!("Hello!");
    DummyResult::any(sp)
}

pub fn register(reg: &mut Registry) {
    reg.add_macro("hello", hello);
}
```

Mainly, the crates are named differently (syntex_syntax instead of just syntax), the lifetime of the macro is now <'cx> instead of <'static> (since the static lifetime isn't available when using syntex), and the Registry API call has changed from register_macro to add_macro. The API change is part of the general instability of compiler plugins; it might very well be something else by the time you're reading this.

`syntex` is a bit of a hack, and as the macros get stabilized, its usage should wane. It's a useful intermediate step that helps using unstable macros on the stable compiler today.

Aster

There's a library that abstracts some of the details of macro building called **AST builder** (`aster`). It's part of the serialization project, `serde`, and can be found at https://github.com/serde-rs/aster. Let's take a look at how to build extensions with it. To make it a tad more interesting, we'll write the code in such a way that both unstable and stable compilers are supported via configuration attributes.

This example will again be in a full Cargo project, since including and configuring an external crate is much easier this way. Plus, we can set up conditional compilation based on stable Rust this way. The project tree will be quite regular, with just the minimal required `Cargo.toml` and `src/main.rs`.

Here's how to set up `Cargo.toml` so that the default build will be for stable Rust and we can optionally use nightly features:

```
# aster/Cargo.toml
[package]
name = "aster"
version = "0.1.0"
authors = ["vegai"]

[features]
default = ["aster/with-syntex", "syntex_syntax"]
nightly = []

[dependencies]
aster = { version = "*", default_features = false }
syntex_syntax = { version = "*", optional = true }
```

The `features.default` key says that the features, `aster/with-syntex` and `syntex_syntax`, are enabled by default. The former is used inside Aster's `Cargo.toml`, while the latter is used in the dependencies section of this `Cargo.toml`. So, in order to enable nightly features with this setup, the build command would be as follows:

```
cargo build --no-default-features --features nightly
```

Building abstract syntax trees via Aster works by its root `AstBuilder` struct. It has methods for creating other leafs of the tree. Aster's documentation at https://docs.serde.rs/aster/ contains descriptions of all the different builders. We'll take a look at a couple of simple expressions here. First off, consider the following Rust expressions:

```
1 + 2 * 3
(1 + 2) * 3
```

These expressions will be obviously different in the AST representation, since any precedence rules get resolved when converting a piece of code to AST. In the first case, the addition is the root of the tree (written out as S-expressions here):

```
(+ 1 (* 2 3)
```

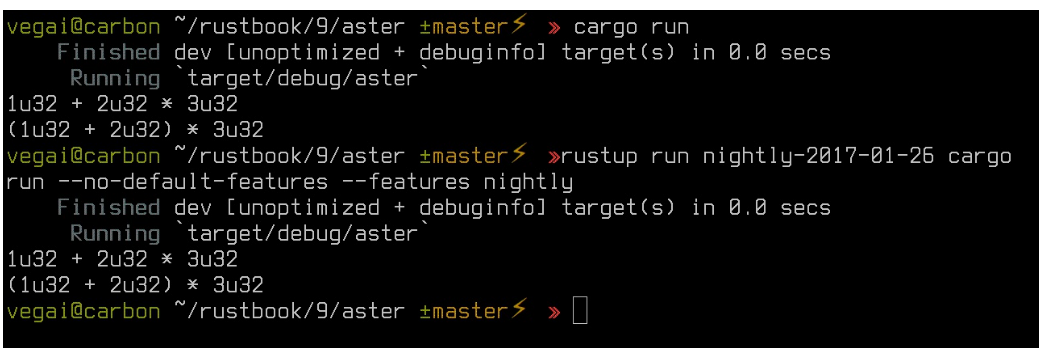
In the second expression, the multiplication is the root:

```
(* (+ 1 2) 3)
```

Building the AST manually means building the same tree with library calls to the `AstBuilder` object. Here's the full code of `main.rs` (with the required bits for conditional compilation on nightly), which builds both the preceding expressions:

```
// aster/src/main.rs
#![cfg_attr(feature = "nightly", feature(rustc_private))]

extern crate aster;

#[cfg(feature = "nightly")]
extern crate syntax;

#[cfg(not(feature = "nightly"))]
extern crate syntex_syntax as syntax;

fn main() {
    let builder = aster::AstBuilder::new();

    let expr1 = builder.expr()
        .add().u32(1).mul().u32(2).u32(3); // 1+2*3
    let expr2 = builder.expr()
        .mul().add().u32(1).u32(2).u32(3); // (1+2)*3

    println!("{}", syntax::print::pprust::expr_to_string(&expr1));
    println!("{}", syntax::print::pprust::expr_to_string(&expr2));
}
```

If you take a look at the second lines of both the expressions, you can see that they correspond quite directly to the preceding S-expressions. Here's the output of running this in both the modes:



We selected this particular nightly version of Rust because the latest Aster at the time of writing this book was 0.41.0, and the nightly build of January 26, 2017 was the closest release to it.

That should get you started at building ASTs with Aster. There's another library for doing similar things, called **Syn.** It uses a slightly different approach: it provides a macro that accepts a special syntax that looks more like Rust code. It will be covered a bit later in this chapter.

Linter plugins

Linter plugins can be used to add new validation checks for code, which the standard Rust compiler does not. Using linter plugins looks similar to using other compiler plugins but with minute differences. They also require a nightly version of the compiler.

To create a minimal linter plugin, you first create a custom struct and then implement the `LintPass` and `EarlyLintPass` traits for it. `LintPass` is for adding human-readable descriptions of the custom lints. `EarlyLintPass` is for creating the actual lint functionality. There is also a `LateLintPass`, the difference being that `EarlyLintPass` hooks up to an earlier phase in the compilation process and `LateLintPass` to a later one. They offer the same hooks but a later phase has access to additional information about types, whereas the earlier phase only has access to the AST. The rule of thumb when creating linter plugins is that you should use `EarlyLintPass` whenever you don't need the additional information since, that way, any errors happen earlier and faster.

Our custom linter will verify that all the functions have return values declared in the type. We can use the `check_fn` method to hook up to all the function definitions. Here's the full plugin code:

```
// lint/src/lint_fn.rs
#![feature(plugin_registrar)]
#![feature(box_syntax, rustc_private)]

extern crate syntax;
extern crate syntax_pos;

#[macro_use]
extern crate rustc;
extern crate rustc_plugin;

use rustc::lint::{EarlyContext, LintContext, LintPass, EarlyLintPass,
                  LintArray};
use rustc_plugin::Registry;
use syntax::ast::{NodeId, FnDecl, FunctionRetTy};
use syntax::visit::FnKind;
use syntax_pos::Span;

declare_lint!(TEST_FN_RETURN, Warn, "Warn about functions that have no return parameters");

struct Pass;

impl LintPass for Pass {
    fn get_lints(&self) -> LintArray {
        lint_array!(TEST_FN_RETURN)
    }
}

impl EarlyLintPass for Pass {
    fn check_fn(&mut self, cx: &EarlyContext, _: FnKind, fndecl: &FnDecl, span: Span, _: NodeId) {
        match fndecl.output {
            FunctionRetTy::Default(_) =>
                cx.span_lint(TEST_FN_RETURN, span, "function has no return parameters"),
            _ => {}
        }
    }
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    let pass = Box::new(Pass);
    reg.register_early_lint_pass(pass);
```

```
}
```

The `check_fn` method does all the work: this code is entered for every function declaration that is checked via the compiler when this plugin is active. It matches the function output type (`fndecl.output`) to the `FunctionRetTy::Default` enum choice, which corresponds to the empty output type `()`. If it matches, we signal an error via the `EarlyContext` object, which causes the compiler to stop compiling and output the error. Here's the main function that uses this lint:

```
// src/lint/main.rs
#![feature(plugin)]
#![plugin(lint_fn)]

fn return_the_answer() -> u8 {
    42
}

fn do_nothing() {

}

fn main() {
    return_the_answer();
    do_nothing();
}
```

Here's the output of building this module with nightly Rust:

```
vegai@carbon ~/rustbook/9/lint ±master⚡ » rustup run nightly cargo build
   Compiling linter v0.1.0 (file:///home/vegai/fossil/rustbook/9/lint)
warning: function has no return parameters
  --> src/main.rs:8:1
   |
8  | / fn do_nothing() {
9  | |
10 | | }
   | |_^
   |
   = note: #[warn(test_fn_return)] on by default

warning: function has no return parameters
  --> src/main.rs:12:1
   |
12 | / fn main() {
13 | |     return_the_answer();
14 | |     do_nothing();
15 | | }
   | |_^
   |
   = note: #[warn(test_fn_return)] on by default

    Finished dev [unoptimized + debuginfo] target(s) in 1.12 secs
vegai@carbon ~/rustbook/9/lint ±master⚡ »
```

Just as we wanted, we now get warnings from the two functions that are using the default return type.

The `rust-clippy` crate contains several custom lints, 176 at the time of writing this book. They can serve as great examples if you wish to see what kind of things linter plugins allow and also if you want to mechanically check for some things in your own code base. The `rust-clippy` source code can be found at https://github.com/Manishearth/rust-clippy.

Macros 1.1 - custom derives

The Rust team is working on stabilizing a significant subset of the compiler plugins mechanism, which should be large enough to be used for a majority of the things libraries usually use compiler plugins for, but small enough to be stabilized. This standardization attempt is called macros 1.1, and both the design and implementation have been stabilized and are available in the stable compiler since early 2017.

Macros 1.1 will give us the ability to write custom derives for structs and enums. This may not seem like much, but it is, in fact, exactly what the aforementioned major libraries needed. The implementation details consist of three things:

- A new crate type, `proc-macro`, which declares a crate as a macro crate
- A new attribute, `proc_macro_derive`, which declares a function as a custom derive attribute
- Library support for manipulating the tokens

Every crate that contains macros of this form must be of the type `proc-macro`. Furthermore, only functions that have the `proc_macro_derive` attribute are allowed to be exported from the crate. Let's dive right into how the simplest custom derive would look:

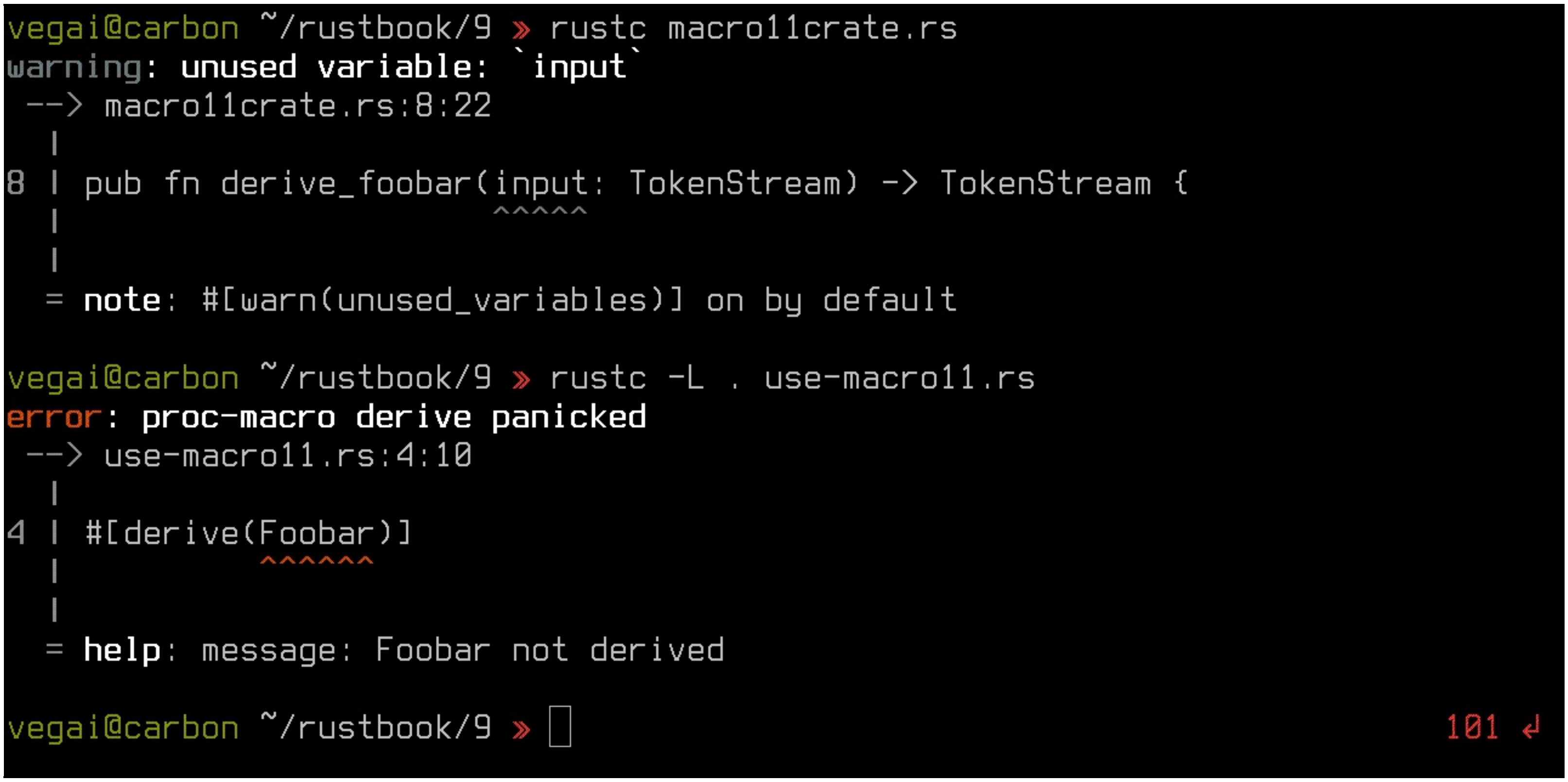
```
// macro11crate.rs
#![crate_type = "proc-macro"]

extern crate proc_macro;

use proc_macro::TokenStream;

#[proc_macro_derive(Foobar)]
pub fn derive_foobar(input: TokenStream) -> TokenStream {
    panic!("Foobar not derived")
}
```

As seen, the `derive` function needs to follow a certain pattern again: it takes a `TokenStream` and returns a `TokenStream`.

This piece of code defines a new derive, called `Foobar`. Using the derive just panics for now, but we can use it to verify that the custom derive is working. Here's a main program that uses this derive:

```
// use-macro11.rs
#[macro_use]
extern crate macro11crate;

#[derive(Foobar)]
struct Foo;

fn main() {
}
```

To build and link all this, we need to first build the macro crate and then have the main program link to it. `rustc` can find the crate if we help it a bit by adding the current path to the list of library paths via the `-L` flag:



All right, we get an error about the custom derive attribute panicking, just as expected. Now, let's try something more interesting.

The derives we saw earlier, and that are implemented internally in Rust, are most typically used to fulfill some traits automatically, such as `Debug`.

In that spirit, we'll add a `Countable` trait, which represents a struct from which we can query the number of fields. The trait will have a single method: `count_fields`. We'll add a custom derive that generates this method.

Here's the trait:

```
trait Countable {
    fn count_fields() -> u64;
}
```

To implement the custom derive, we'll use a library called `syn` to help us with manipulating the `TokenStream`.

Here's the code that implements the custom derive:

```
// countable/src/count_fields_plugin.rs
#![feature(proc_macro, proc_macro_lib)]

extern crate proc_macro;
use proc_macro::TokenStream;

extern crate syn;
use syn::{Field, Ident, Path, Ty};

#[macro_use]
extern crate quote;

#[proc_macro_derive(Countable)]
```

```
pub fn summable_fields(input: TokenStream) -> TokenStream {
    let source = input.to_string();
    let ast = syn::parse_macro_input(&source).unwrap();
    let expanded = expand_summable_fields(&ast);
    quote!(#ast #expanded).to_string().parse().unwrap()
}

fn expand_summable_fields(ast: &syn::MacroInput) -> quote::Tokens {
    let n = match ast.body {
        syn::Body::Struct(ref data) => data.fields().len(),
        syn::Body::Enum(_) => panic!("#[derive(Countable)] can only be used with structs"),
    };

    let name = &ast.ident;
    let (impl_generics, ty_generics, where_clause) = ast.generics.split_for_impl();

    quote! {
        impl #impl_generics ::Countable for #name #ty_generics #where_clause {
            fn count_fields(&self) -> usize {
                #n
            }
        }
    }
}
```

As you can see, `syn` allows binding values that can be used from inside a quoted piece of code. The variables starting with `#` inside the `quote!` macro get replaced by the contents of the variables at compile time. These variables can be any kind of AST representation.

Let's see how it handles a few structs:

```
#![feature(proc_macro)]
#![allow(dead_code)]

#[macro_use]
extern crate count_fields_plugin;

trait Countable {
    fn count_fields(&self) -> usize;
}
#[derive(Countable)]
struct S1 {
    x: u32,
    y: u8
}

#[derive(Countable)]
struct S2 {
    s: String,
    x: u64,
    y: i64
}

#[derive(Countable)]
struct S3 {
    s: String
}

fn main() {
    let s1 = S1 { x: 32, y: 8 };
    let s2 = S2 { s: "String".to_string(), x: 64, y: -64 };
    let s3 = S3 { s: "String".to_string() };
    println!("s1 has {} fields", s1.count_fields());
    println!("s2 has {} fields", s2.count_fields());
    println!("s3 has {} fields", s3.count_fields());
}
```

The output is as expected:

```
vegai@carbon ~/rustbook/9/countable ±master⚡ » cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/countable`
s1 has 2 fields
s2 has 3 fields
s3 has 1 fields
vegai@carbon ~/rustbook/9/countable ±master⚡ »
```

Exercises

1. Write a `Serializable` trait with `ser` and `deser` methods. Create a custom derive attribute using macros 1.1, which implements those functions automatically. You don't have to be able to load and save every kind of type; just a few primitives will be more than fine.
2. Write a compiler plugin that disallows too long functions.

Summary

In this chapter, we covered many of the advanced macro features of Rust. They allow a Rust programmer to run arbitrary Rust code at compile time for various effects:

- To generate Rust code based on some outside environmental state (such as database tables, time of date, and so on)
- To decorate structures with custom attributes, generating arbitrary code for them at compile time
- To create new linter plugins for making additional code checking passes that Rust itself does not support

Many of these features require the nightly version of the Rust compiler. There's a workaround via a code generation library called `syntex`, which enables many uses of nightly macros to work on the stable compiler, but it is slightly awkward to use.

The Rust community has two ongoing efforts for stabilizing macros. These attempts are called macros 1.1 (which contains stabilized support for custom attributes) and macros 2.0 (which should contain the remaining compiler plugin features). Of these, macros 1.1 is almost ready for inclusion in the stable Rust compiler at the time of writing this book, and 2.0 is still being designed.

As long as compiler plugins are a nightly feature, their documentation is not quite as easily available as the documentation of more stable features is. Fortunately, a prominent Rust compiler developer, Manish Goregaokar, maintains a collection of compiler internal documentations that can be very helpful in unraveling the mysteries of compiler plugins, among other things. The docs are located at https://manishearth.github.io/rust-internals-docs.

Unsafety and Interfacing with Other Languages

Rust's safety features protect the programmer from many serious mistakes, but there are times when it's appropriate to shake off the safety harnesses. One useful case is interfacing with other programming languages that are less safe, the most prominent target being C. We will also cover interfacing with some high-level languages where the community has created bridges, such as Ruby and JavaScript.

Here's the list of topics for this chapter:

- Unsafety
- Foreign function interface
- Interfacing with Ruby using ruru
- Interfacing with JavaScript/Node.js using Neon

Unsafety

There are many kinds of *unsafety* in programming. A practical way to think about it is to say that anything that can cause problems in a running program can be labeled as unsafe. The level of unsafety depends on how destructive the problems may be, how predictable they are, and how easy they are to find and fix. A few examples:

- A program uses floating-point numbers to store money. However, floating point numbers are not exact and may easily cause rounding errors. The impact depends on the situation: in some cases, being off by the thousands of cents may be OK, but, in accounting, the money values must be exact. This error is somewhat predictable (since, given the same input, it always manifests itself in the same way) and easy to fix. Rust offers no protection for such programming errors.
- A program for controlling a spaceship uses primitive numbers to store distances. However, in some pieces of code, the distances are interpreted in the metric system, and in some other places they are interpreted in the imperial system. This error has actually happened in spaceflight and has caused serious damage. Rust doesn't fully protect from such mistakes, although the enum concept allows easily separating different units from each other, making this error much less likely.
- A program writes to shared data from multiple places without appropriate locking mechanisms. The error manifests itself unpredictably, and finding it can be very difficult, since it is dependent on everything the host machine is doing at the same time. Rust fully protects against this problem with its resource borrowing and lifetimes system.
- A program accesses an object through a memory pointer, which, in some situations, is a null pointer, causing the program to crash. In the default mode, Rust fully protects against null pointers.

Since there are situations where the programmer really knows better than the compiler, some of the restrictions can be circumvented explicitly. The following three things are forbidden by default but allowed in unsafe mode:

- Updating a mutable static variable
- Accessing via a raw pointer
- Calling an unsafe function

There are four things that can be declared by using the `unsafe` keyword:

- Functions
- Blocks of code
- Traits
- Implementations

To mark a function as unsafe, prepend the keyword to the function declaration. Here's an `unsafe` function that takes in a raw pointer to an i32 and dereferences it:

```
unsafe fn unsafe_function(i: *const i32) -> i32 {
   *i
}
```

The `unsafe` function behaves like a regular function, except that the three aforementioned operations are allowed in it. Of course, declaring your function as unsafe makes it uncallable from regular, safe functions.

The `unsafe` blocks can be used inside functions to mark sections of code as unsafe. The preceding functions could also be written like this:

```
fn unsafe_function(i: *const i32) -> i32 {
   unsafe {
       *i
      }
}
```

This looks the same as before but contains a significant change. The function now does an essentially unsafe thing but wraps it inside a function that is marked as safe. Therefore, this function can be called without being in unsafe mode. This technique can be used to provide an interface that looks safe even though it is doing something unsafe internally. Obviously, if you do this, you should take special care that the `unsafe` blocks are correct.

Consider the preceding function; it is inherently not safe. Here are two ways to use it:

```
unsafe_function(&4 as *const i32);
unsafe_function(4 as *const i32);
```

Both are accepted by the compiler, but the second one causes a segmentation fault, which should never happen in safe Rust code. On the bright side, the `unsafe` block is at least marked as such in the function code, but that is hardly a consolation if that function is inside a third-party crate that you are just using. To reiterate, if you're writing libraries and need to reach for `unsafe`, be extra careful! If you're not certain that you have managed to create a safe wrapper around the `unsafe` part, better mark the function `unsafe`.

Implementation and `trait` blocks work the same, just prepend the `unsafe` keyword in the declaration:

```
unsafe trait UnsafeTrait {
   unsafe fn unsafe_function();
}
unsafe impl UnsafeTrait for MyType {
     unsafe fn unsafe_function() { }
}
```

An unsafety declaration in `traits` is required when any of the functions in them are marked as `unsafe`. Similarly, to implement an `unsafe trait`, the `impl` needs to be marked `unsafe`. Finally, a function marked as `unsafe` in the trait must be `unsafe` in the implementation as well.

Calling C code from Rust

Linking Rust code into C code requires the following two things at the minimum:

- The foreign function declared inside an `extern` block
- `std::os::raw` contains types that map directly to primitive C types and other functionalities that can be used from Rust

Let's look at a simple example to see how these things come together. Here's a piece of C code that measures the length of a C string. As you know, C strings are pointers to the first character of a contiguous block of memory whose end is signified by a zero byte:

```
/* count-string.c */
unsigned int count_string(char *str) {
  unsigned int c;
  for (c=0; *str != '\0'; c++, *str++);
  return c;
}
```

The primitive types are in `std::os::raw`, with names close to their C counterparts. A single letter before the type says whether the type is unsigned. For instance, the unsigned integer would be `c_uint`.

Because C strings are quite unlike Rust's, there's additional support for them in the `std::ffi` namespace:

- `std::ffi::CStr` represents a borrowed C string. It can be used to access a string that has been created in C.
- `std::ffi::CString` represents an owned string that is compatible with foreign C functions. It is often used to pass strings from Rust code to foreign C functions.

Since we want to pass a string from the Rust side to the function we just defined, we should construct and use the `CString` type here. Long story short, here's the Rust counterpart that calls the function in the preceding C code:

```
// count-string.rs
use std::os::raw::{c_char, c_uint};
use std::ffi::CString;

extern {
    fn count_string(str: *const c_char) -> c_uint;
}

fn main() {
    let c_string = CString::new("A string that can be passed to C").expect("Creating the C string failed");
    let count = unsafe {
        count_string(c_string.as_ptr())
    };
    println!("c_string's length is {}", count);
    let count = safe_count_string("12345").expect("goot");
    println!("c_string's safe length is {}", count);
}
```

Notice the `expect` call on the line where we create the string object? It highlights an important difference between Rust strings and C strings (C strings are terminated by the null byte whereas Rust strings contain the actual length parameter). This means that creating a C string may *not* contain a null byte as an element and a Rust string *may,* so a conversion from a Rust string will fail if it contains one.

To get this to work as a whole, we'll need to first build an object file out of the C code and then tell rustc to link against it. The first task depends a lot on what platform you are on, but we'll use GCC here. To link to the object file, we can use the `link_args` attribute to rustc through which we can add additional link arguments. Here's the manual build process and a demonstration of the resulting program:

```
vegai@carbon ~/rustbook/10 » gcc -c count-string.c
vegai@carbon ~/rustbook/10 » rustc -C link_args=count-string.o count-string.rs
vegai@carbon ~/rustbook/10 » ./count-string
c_string's length is 32
c_string's safe length is 5
vegai@carbon ~/rustbook/10 »
```

OK, now we have a rather minimal method of linking to a C module. There are a couple of ways to improve on this, however. First of all, it is awkward that we have to be in an `unsafe` block to call the function. To solve this, we can create a **safe wrapper** function in Rust. In a simplistic form, this just means creating a function that calls the external function inside an unsafe block:

```
// count-string.rs
fn safe_count_string(str: &str) -> Option<u32> {
    let c_string = match CString::new(str) {
        Ok(c) => c,
        Err(_) => return None
    };

    unsafe {
        Some(count_string(c_string.as_ptr()))
    }
}
```

We also moved the creation of the `CString` inside this `safe` function, and now, calling it feels exactly like calling any other Rust function. It is typically recommended to make safe wrappers around external functions, taking care that all exceptional cases happening inside the `unsafe` block are handled properly.

Connecting external libraries to Rust code

It's more likely that we'll want to link some pre-existing library written in C to our Rust code. The needed code on the Rust extends a bit on the previous example. We'll take the classic Unix `ncurses` library, which is used to display graphics, like text on the console. The first task is to define the external functions we'd like to use, supplying the `extern` block with a `link` attribute that tells which library the functions are coming from. The `ncurses` library has tons of functions, but we'll just use a couple in this example.

If you have a reasonable Unix-like operating system, you might want to look at the ncurses manual page by commanding `man ncurses`. Let's start by building a trivial program:

```
// ncurses-1.rs
use std::os::raw::c_char;
use std::ffi::CString;

#[link(name="ncurses")]
extern {
    fn initscr();
    fn printw(fmt: *const c_char, ...);
    fn refresh();
    fn getch();
    fn endwin();
}

fn main() {
    let the_message = CString::new("Rust and ncurses working together, and here is a number: %d").unwrap();
    unsafe {
        initscr();
        printw(the_message.as_ptr(), 1);
        refresh();
        getch();
        endwin();
    }
}
```

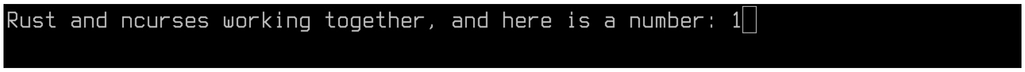
Couple of things to note here:

- The link attribute refers to the ncurses library. It needs to be installed in the system so that the linker can find it for this to work.
- The `printw` function's first parameter is a format string and the other parameters are variadic (the three dots). This means that after the first parameter, an arbitrary number of parameters may follow. We don't need to handle it differently in Rust, it just works.

The code first initializes the screen, prints the formatted string to it, refreshes the screen to make the text actually visible, and waits for any single keystroke to end the program. The compilation of this program requires nothing out of the ordinary, since the link directives are in the code, just the following:

```
rustc ncurses.rs
```

And running it works, too:



Although the variadic arguments work neatly, there's a caveat that there are no checks to verify whether you're using the correct number of arguments. If for instance, you changed the format string to this without any other changes, you would get a segmentation fault error at runtime, since the second parameter gets passed nothing:

```
    let the_message = CString::new("Rust and ncurses working together,
      and here is a number: %d %s").unwrap();
```

With that in mind, let's enhance the previous by making safe wrappers and using them to print the text in red on the console:

```
// ncurses-2.rs
use std::os::raw::{c_char, c_short};
use std::ffi::CString;

#[link(name="ncurses")]
extern {
    fn initscr();
    fn printw(fmt: *const c_char, ...);
    fn refresh();
    fn getch();
    fn endwin();
    fn start_color();
    fn init_pair(pair: c_short, f: c_short, b: c_short);
    fn attron(pair: c_short);
    fn COLOR_PAIR(pair: c_short) -> c_short;
}

struct Ncurses;
impl Ncurses {
    fn init_screen() {
        unsafe { initscr(); start_color(); }
    }
    fn refresh() {
        unsafe { refresh(); }
    }
    fn get_char() {
        unsafe { getch(); }
    }
    fn deinit_screen() {
        unsafe { endwin(); }
    }
    fn color_red() {
        unsafe { init_pair(1, 1, 0); attron(COLOR_PAIR(1)); }
    }
}

fn main() {
    let the_message = CString::new("Rust and ncurses working together, and here is a number: %d").unwrap();

    Ncurses::init_screen();
    Ncurses::color_red();
    unsafe {
        printw(the_message.as_ptr(), 1);
    }

    Ncurses::refresh();
    Ncurses::get_char();
    Ncurses::deinit_screen();
}
```

We namespaced the ncurses functionality inside an `Ncurses` implementation, which provides a nice and simple compartment for it. The `printw` external function remains as a direct call to the external function, since there's no simple way of handling variadic arguments to C from Rust code. Here's the screen you should see when running this program:

Creating Ruby extensions with Ruru

Rust has attracted many Ruby developers. Supposedly, they are the sort of people who like languages that start with the letters *ru*. Seriously, though, this is good news for anyone looking for strong interoperability between these languages. A project called *ruru* has a nice set of helpers to make this easier. In order for all this to work, you will need to have a fairly recent Ruby installed.

Here's a minimal example of a piece of Rust code that can be called from Ruby:

```
// ruru_test/src/lib.rs
#[macro_use]
extern crate ruru;

use ruru::{Float, Class, Object};

methods!(
    Float,
    itself,

    fn sum_floats(f2: Float) -> Float {
        let f1 = itself.to_f64();
        let f2 = f2.expect("f2 contained an error").to_f64();

        Float::new(f1 + f2)
    }
);

#[no_mangle]
pub extern fn initialize_sum_floats() {
    Class::from_existing("Float").define(|itself| {
        itself.def("+", sum_floats);
    });
}
```

The Ruby standard classes are defined in the `ruru` crate. The code that we want to be visible from the Ruby side is defined inside a `methods!` macro. The macro body starts by listing the required Ruby classes. The keyword `itself` is synonymous to `self` on the Ruby side, but since the word is a reserved keyword in Rust, `self` cannot be used. The function gets a Ruby float (which needs to be unwrapped by `expect` because there might be discrepancies) and constructs and returns a Ruby float.

The `initialize_sum_floats` function is the point of entry from the Ruby code, and we define in there that the `sum_floats` function is to replace the float class's `+` function. In essence, this will overload the internally defined float's plus function with ours.

There's one more thing to do for us to get a library that can be used from the Ruby side; the library needs to be defined as shared. This can be done via `Cargo.toml`. Here's the full `Cargo.toml`:

```
# ruru_test/Cargo.toml
[package]
name = "ruru_test"
version = "0.1.0"

[dependencies]
ruru="0.9"

[lib]
crate-type=["dylib"]
```

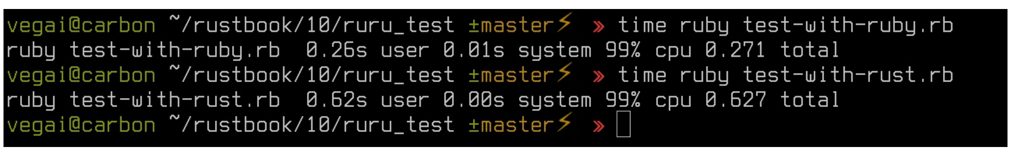
Now, a Cargo build will build the shared library in a target. The Ruby code uses the fiddle library that comes with the standard distribution. Here's a piece of code that runs a simple float sum five million times:

```
# ruru_test/test-with-rust.rb
require 'fiddle'

library = Fiddle::dlopen('target/release/libruru.so')
Fiddle::Function.new(library['initialize_sum_floats'], [], Fiddle::TYPE_VOIDP).call

5000000.times {
  puts 1.2+1.3
}
```

So, what do you think? Our superb Rust implementation of the plus function should be faster than the famously slow Ruby implementation. Let's build all this and make a simple benchmark. The other file, `test2.rb` here, will just contain the last three lines of `test.rb`:

```
# ruru_test/test-with-ruby.rb
5000000.times {
  puts 1.2+1.3
}
```

Here's what happened on my computer when running all these:



Our Rust version was about two times slower. This is because we are forced to convert the floats from Ruby representation to Rust and back again for every loop, whereas the Ruby version obviously doesn't have to do anything like that. This is what we should take away from this:

- Benchmark whether replacing modules with Rust actually helps at all
- Even though most things in Rust are zero-cost, it doesn't mean that all library calls are or even can be
- The larger chunks of work you can move to the Rust side, the more likely it is that you will see an improvement in speed

Let's try something that involves a bit more computing: a very naive algorithm for finding an approximate square root of a float. Here's the algorithm:

1. We will take the square root of S.
2. Define the accuracy of our algorithm. This is often called epsilon in math. Let's say *epsilon=0.00000001*, to make this especially unfair to Ruby.
3. Define *x=0*.
4. Increment *x* by epsilon.
5. If *x* squared is close enough to S (differs from it less than epsilon), that is our answer. Otherwise, go back to 4.

Here's the Ruby version of the code:

```
# ruru_test/sqr-with-ruby.rb
class Float
  def square_root(e=0.00000001)
    x=0
    while x**2 < (self-e) or x**2 > (self+e)
      x += e
    end
    x
  end
end

puts 9.0.square_root
```

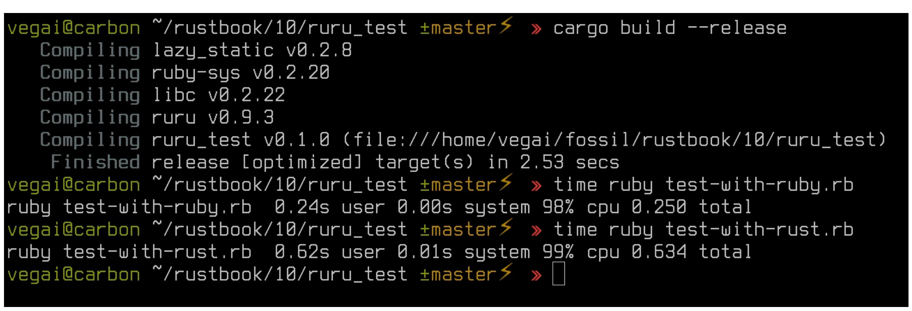
And here's the corresponding implementation in Rust:

```
    // ruru_test/src/lib.rs
    fn square_root(e: Float) -> Float {
        let e = match e {
            Ok(ruby_float) => ruby_float.to_f64(),
            Err(_) => 0.00000001
        };
        let mut x=0.0;
        let s = itself.to_f64();
        while x*x < s-e || x*x > s+e {
            x += e;
        };
        Float::new(x)
    }
```

Now, there are only three conversions needed between Ruby and Rust, and millions of iterations in pure Rust code.

In case you are wondering why we have to handle the error case before we can use the `e` parameter, it is because, on the Ruby side, the `e` parameter is optional. Since Rust does not have similar language support for optional parameters, it is done here via the result type and unwrapping.

I won't go through the rest of the boilerplate code here, since it does not differ from the previous example, and because it makes for a nice exercise. Here's the output of running both the Ruby implementation and the one augmented by Rust:



Now we're talking! Identical answers and a 37-time speedup from the Rust version. It is somewhat rare that glaring optimization opportunities such as this one present themselves to us. Still, when they do, it's nice to have a secret weapon in the toolbelt.

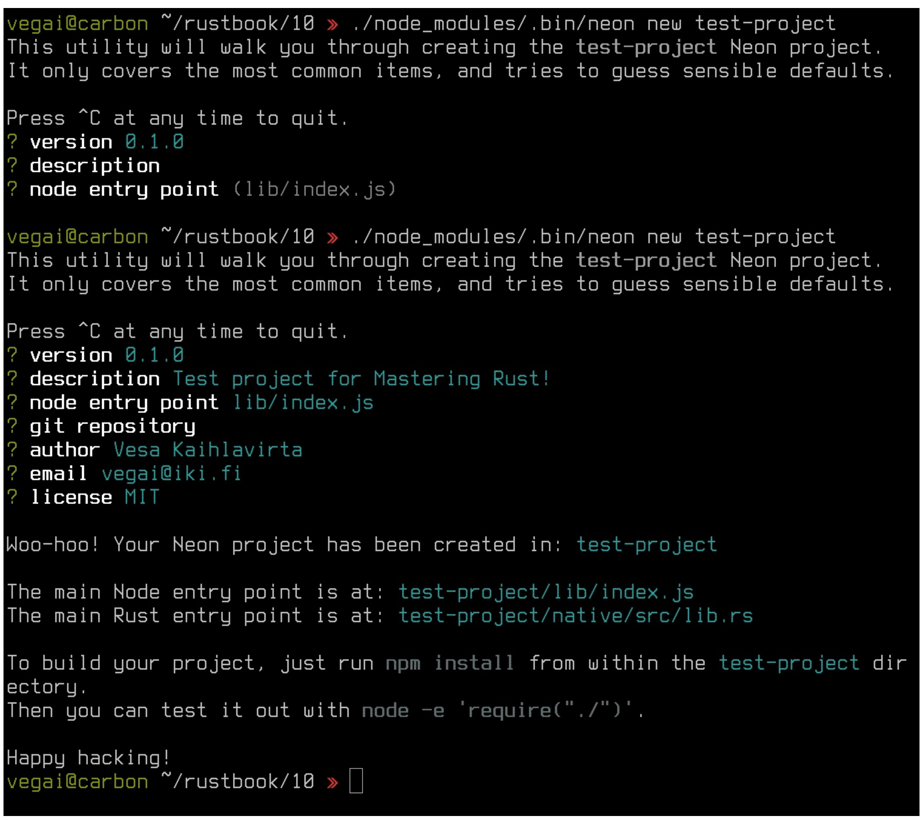
JavaScript/Node.js and Neon

The `neon` library present at http://neon.rustbridge.io/, contains an abstraction layer and assorted tools for writing Rust that behaves like a native Node.js module. It's partially written in JavaScript: there's a command-line program, `neon`, in the `neon-cli` package, a JavaScript side support library, and a Rust side support library. Node.js itself has good support for loading native modules, and neon uses that same support.

Following through these examples requires Node.js to be installed locally and its package manager `npm` to be found along the path. The command-line support tool, `neon-cli`, itself is installed via npm by running this command:

```
npm install neon-cli
```

You may opt for installing `neon-cli` globally by using the `-g` switch, but that requires root permissions and is therefore not absolutely recommended. If the preceding command succeeds, the command-line program Neon will be under your home directory in `./node_modules/.bin/neon`. You can add that path to your `PATH` environment variable to access it more easily. Neon can be used to found a JavaScript project with skeleton Neon support included. Here's an example run of neon:



Here's the directory tree this command created for us:

We can see that in addition to the Node.js skeleton, it also created a Cargo project for us with some initial code in it. Here's `index.js`:

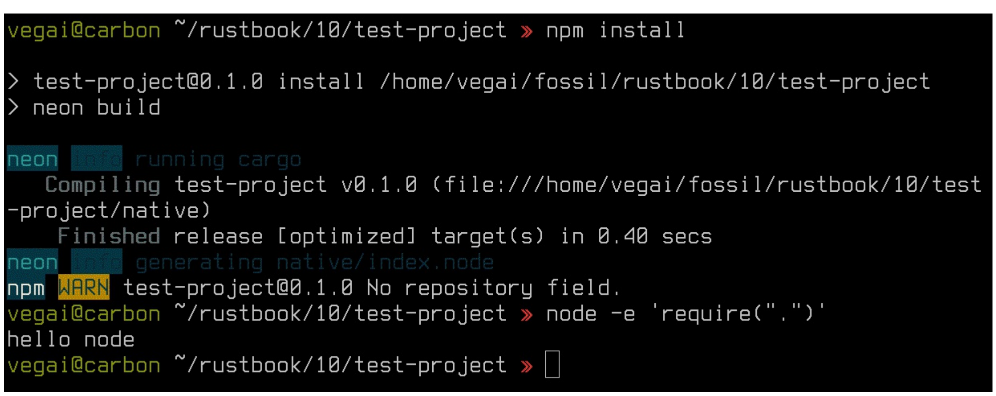
```
var addon = require('../native');

console.log(addon.hello());
```

This program first requires the native module, then calls the `hello` function from there, and logs its output to the console. Here's the code for that module:

```
// test-project/native/src/lib.rs
#[macro_use]
extern crate neon;

use neon::vm::{Call, JsResult, Module};
use neon::js::JsString;

fn hello(call: Call) -> JsResult<JsString> {
    let scope = call.scope;
    Ok(JsString::new(scope, "hello node").unwrap())
}

register_module!(m, {
    m.export("hello", hello)
});
```

We can see similar patterns as with the Ruby case. First, we define the function, taking in parameters as special structs and returning values in structs that are compatible with the target language, which is JavaScript in this case. Then, we register the new function so that the target language can see it.

To run this example, we'll need to first run `npm install` and then start the application via `node -e 'require(".")'`:



Parts of the output of `npm install` are omitted because a lot of it was uninteresting reports of installing tens of JavaScript dependencies. Let's go back to the Rust code in the example:

```
fn hello(call: Call) -> JsResult<JsString> {
    let scope = call.scope;
    Ok(JsString::new(scope, "hello node").unwrap())
}
```

The `call` parameter gives us all the context we need to manipulate the JavaScript side from Rust. JavaScript cannot figure the scope of things from Rust, so we need to specify it every time we create some object there.

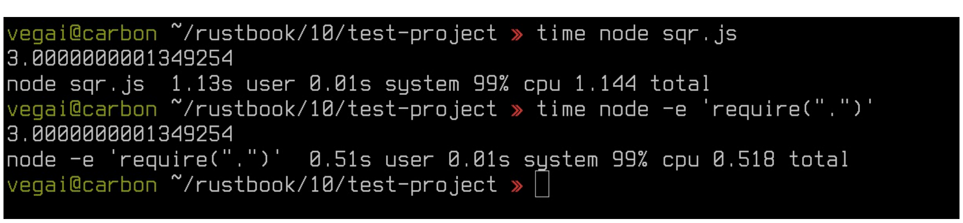
Let's try the naive square root algorithm again. JavaScript should offer a better challenge than Ruby, given the high performance of the v8 engine. Here's the JavaScript implementation:

```
function squareRoot(s, e=0.00000001) {
    var x=0;
    while (x*x < s-e || x*x > s+e) {
        x += e;
    }
    return x;
}

console.log(squareRoot(9.0));
```

Here's the corresponding Rust implementation, with the needed Neon boilerplate to get the values to and from Node.js:

```
#[macro_use]
extern crate neon;

use neon::vm::{Call, JsResult};
use neon::js::JsNumber;

fn square_root(call: Call) -> JsResult<JsNumber> {
    let scope = call.scope;
    let s = try!(try!(call.arguments.require(scope, 0)).check::<JsNumber>()).value();
    let e = match call.arguments.get(scope, 1) {
        Some(js_number) => try!(js_number.check::<JsNumber>()).value(),
        None => 0.00000001
    };
    let mut x=0.0;

    while x*x < s-e || x*x > s+e {
        x += e;
    };
    Ok(JsNumber::new(scope, x))
}

register_module!(m, {
    m.export("squareRoot", square_root)
});
```

Getting the arguments from the `call` is a bit more involved than with ruru, but otherwise the implementation is pretty much the same. `JsNumber` corresponds to the standard JavaScript number/float type. Here's the result of benchmarking both these implementations:



Even with the performance-tuned v8 engine, our Rust module still manages to be more than two times faster. The difference is not drastic, and may not warrant actually managing all the complexities of interfacing with another language. However, this technique might become more useful in the future, when more and more modules are written in Rust, making usability and not just performance be the driving cause.

Exercises

1. Write the Ruby module that runs the Rust version of the square root function. Additionally, find out if there's a combinator function in Result that does the unwrapping of the `e` parameter in a more concise way.
2. Extend the `ncurses` library by a few additional functions from the library. Create safe wrappers for them and use them.
3. Extend the safe wrappers of the `ncurses` library. Could a `macro-by-example` macro be used to make a safer `printw`? Could the initialization and deinitialization of the screen be made in a constructor and destructor implicitly?

Summary

Rust is a safe language, which means that the compiler does a great job of protecting us from many simple and complex mistakes, such as memory access errors and accessing mutable values from several places. Sometimes the compiler is too strict, however, and we need to tell it to drop some of the protections.

This is especially needed when integrating with legacy ecosystems, such as C, which have no such safety settings. Therefore, any calls to the functions written in C are unsafe and must be tagged as such to satisfy the compiler. If we are careful, we can write safe wrappers around these `unsafe` blocks, which makes the C functions look like ordinary Rust functions. The **Foreign Function Interface** (**FFI**), like many other Rust things, is zero-cost, which means that no runtime cost is paid when linking to the C code.

Rust has so-called bridges to other programming languages. Since many Ruby developers became interested in Rust, the most notable bridge, Helix, makes it trivial to access Rust code from Ruby and vice versa. Interfacing with other languages is always possible by using FFI in case abstractions haven't been written for them yet.

The remaining chapters will cover case studies of various Rust frameworks and libraries.

Parsing and Serialization

In this case study chapter, we'll take a look at a few current ways of writing custom parsers. We'll also take a look at the standard form of serialization through the Serde library.

The following topics will be covered in this chapter:

- Parsing basics
- nom
- Chomp
- Serde

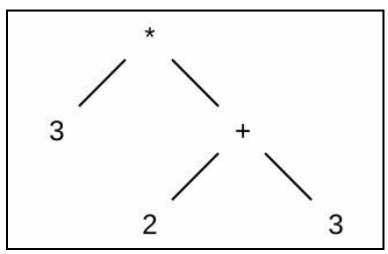
Parsing fundamentals

When we want to turn any sort of regular input into internal representations, we'll need to do some form of parsing. **Parsing** is one of the most researched topics in computer science, so a full coverage of the topic won't be feasible here. If you want to learn more about this very rich subject, I can recommend *Language Implementation Patterns* by *Terence Parr*. We'll cover some of the basics here.

It's important to learn early that while parsing and deserialization have similar themes, they are not synonymous concepts. Deserialization makes internal objects out of a stream of data (usually a string). Parsing is a more general idea of handling input data, where the end result does not have to be a set of objects.

In a simplified form, parsing is required when we get a linear sequence of input (usually characters but sometimes arbitrary binary data) and want to either perform actions based on the input or transform it into an internal representation. A typical example of this would be the transformation of program code into an abstract syntax tree. Consider the following expression: *3 * (2+3)*.

We want to get the following tree structure, with the implicit calculation ordering correctly interpreted:



There are several parsing techniques one can use, the simplest being just reading one character at a time and making decisions based on that. More complex techniques include:

- Reading ahead one or more characters at a time
- Trying several overlapping parsing routes and backtracking when they don't fully succeed
- Memorizing previous parsing matches when doing backtracking (also called **packrat parsers**)
- Add predicates (simple Boolean expressions) to parsers to help with parsing decisions

This is by no means an exhaustive list. Go ahead and think about the preceding mathematical expression. Would the simplest form of a parser be powerful enough to give us the correct tree form, not to mention correct tree forms for every other possible mathematical expression?

There are many ways to write this parser. One would be to use a left to right, leftmost derivative parsing with an arbitrary lookahead. Such parsers are called **LL(k)** parsers, with **LL** referring to the left to right and leftmost derivative, and **k** referring to the number of lookahead tokens. Why is the arbitrary lookahead required? Because a mathematical expression has implicit rules on calculation order, and so on, that cannot be derived correctly with just a static lookahead quantity.

But enough theory. Let's dive into a few Rust parser frameworks.

nom

Our first stop in checking out various Rust parsing libraries is **nom**. It's actually a parser combinator library, which refers to the fact that every parser can be combined with another. So, instead of writing a full parser for a language using some parser definition language such as EBNF, with parser combinators, you can write small and reusable parsers and combine them together.

nom plays into many of Rust's strengths, providing strong typing parsers with zero-copy semantics. This means that using nom will cause no more memory allocations than a corresponding optimal handwritten one would. nom uses macros quite lot to achieve more ergonomic usage and for optimizing code by precomputing things at compile-time.

It accepts input in any of these three granularities:

- Bit by bit
- Byte by byte
- UTF-8 string

nom generates very fast parsers, reportedly even beating parsers handwritten in C. The combination of all these features makes nom a good choice for practically every conceivable parsing need. The downside is that writing nom parsers is not simple, but fortunately, it comes with a set of libraries and macros that make it at least simpler. Even more fortunately, you have this book.

nom parsers are basically functions that take as input any of the aforementioned types (bits, bytes, or strings) and return a special `IResult` enumeration type, which can represent any of the following three situations:

- `Done(I, O)`: Parsing was successful, the first parameter `I` contains the remaining unparsed input and `O`, the result of the parsing
- `Error(Err)`: Parsing was unsuccessful, `Err` contains an enum that represents the reason why it happened
- `Incomplete(Needed)`: More input is needed, `Needed` optionally containing how much

nom contains an ever-growing list of pre-existing parsers that you can combine to make complete parsers. Here are a few examples:

- `Alpha` will match and return an array of alphabets
- `Digit` will match and return an array of numbers
- `Alphanumeric` combines the previous two
- `line_ending` matches a line end
- `Space` matches the longest array containing only spaces
- `eof` matches an end of input

You can find a current list of these from the documentation at http://rust.unhandledexpression.com/nom/. In order to build our own macros from these basic building blocks, nom offers a couple of helper macros. Here they are:

- `named!` creates a new parser.
- `tag!` matches a byte array.
- `take_while!` matches against bytes until a given predicate returns false.
- `do_parse!` applies parsers as a stream, separated by `>>`. There's a special colon syntax for storing intermediate parser results. The very last statement is what the whole parser returns.

The full list of these macros can be found from the aforementioned documentation site. To make all this a bit more concrete, let's take a look at an actual parser. Here's a piece of the `ISO8601` date parser, implemented in https://github.com/badboy/iso8601/blob/master/src/parsers.rs:

```
named!(day_zero  <u32>,
   do_parse!(tag!("0") >> s:char_between!('1', '9') >> (buf_to_u32(s))));
named!(day_one   <u32>,
   do_parse!(tag!("1") >> s:char_between!('0', '9') >> (10+buf_to_u32(s))));
named!(day_two   <u32>,
   do_parse!(tag!("2") >> s:char_between!('0', '9') >> (20+buf_to_u32(s))));
named!(day_three <u32>,
   do_parse!(tag!("3") >> s:char_between!('0', '1') >> (30+buf_to_u32(s))));
named!(day <u32>, alt!(day_zero | day_one | day_two | day_three));
```

This defines a day parser, which parses days of the months in number form and returns them as `u32`. The `char_between` macro is not nom's own; rather, it is defined in the same module and, as its name gives away, matches single digits that are between the two digits given. We'll include its implementation in a full example a bit later.

How this works is by defining four different parsers (`day_zero`, `day_one`, `day_two`, and `day_three`) for the different possible forms of days, depending on which number we start with. For instance, the parser that handles days of the month starting with number 3 has the following three parsers in sequence:

1. `tag!("3")` matches the digit `3`.
2. `s:char_between('0', '1')` matches any digit between `0` and `1`, that is, only those two digits. The result of the match gets stored in `s`.
3. `30+buf_to_u32(s)` converts the match from the previous step to `u32` and adds `30` to it.

The rest follow the same pattern and should be obvious now. We get the final parser on this line:

```
named!(day <u32>, alt!(day_zero | day_one | day_two | day_three));
```

The `alt!` macro tries each parser one at a time and returns the result of the first matching parser. If none of them match, the whole parser fails.

Now, we have a parser called day, which returns a `u32`. It can be used as any regular function in program code. Let's see how. nom is just a typical Rust crate, so we'll need to start a cargo project for this:

```
cargo init -bin nom-test
```

Then, add the nom library to `Cargo.toml`:

```
[dependencies]
nom = "2.*"
```

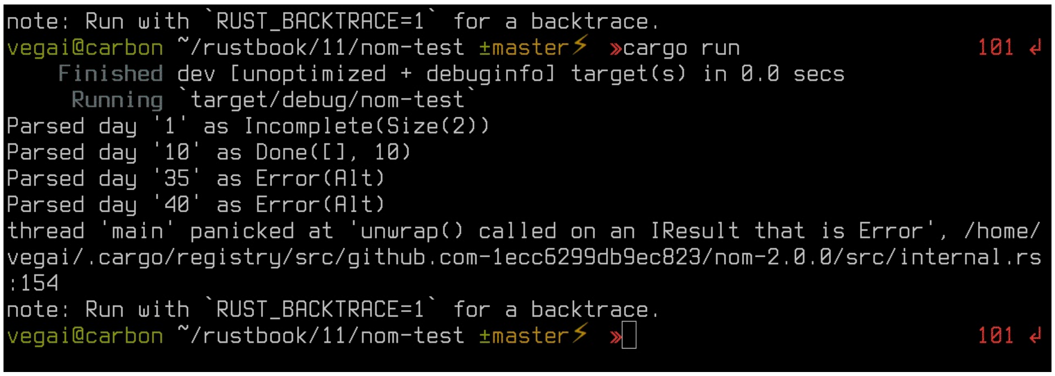
Here's the full main.rs, including, for the sake of completeness, the supporting macros and functions imported from https://github.com/badboy/iso8601:

```
// nom-test/src/main.rs
#[macro_use]
extern crate nom;

use std::str::{FromStr, from_utf8_unchecked};
macro_rules! check(
    ($input:expr, $submac:ident!( $($args:tt)* )) => (

    {
        let mut failed = false;
        for &idx in $input {
            if !$submac!(idx, $($args)*) {
                failed = true;
                break;
            }
        }
        if failed {
            nom::IResult::Error(nom::ErrorKind::Custom(20))
        } else {
            nom::IResult::Done(&b""[..], $input)
        }
    }
);
($input:expr, $f:expr) => (
check!($input, call!($f));
  );
);

macro_rules! char_between(
    ($input:expr, $min:expr, $max:expr) => (
    {
        fn f(c: u8) -> bool { c >= ($min as u8) && c <= ($max as u8)}
    flat_map!($input, take!(1), check!(f))
}
    );
);

fn to_string(s: &[u8]) -> &str {
    unsafe { from_utf8_unchecked(s) }
}

fn to_u32(s: &str) -> u32 {
    FromStr::from_str(s).unwrap()
}

fn buf_to_u32(s: &[u8]) -> u32 {
    to_u32(to_string(s))
}


named!(day_zero  <u32>,
   do_parse!(tag!("0") >> s:char_between!('1', '9') >> (buf_to_u32(s))));
named!(day_one   <u32>,
   do_parse!(tag!("1") >> s:char_between!('0', '9') >> (10+buf_to_u32(s))));
named!(day_two   <u32>,
   do_parse!(tag!("2") >> s:char_between!('0', '9') >> (20+buf_to_u32(s))));
named!(day_three <u32>,
   do_parse!(tag!("3") >> s:char_between!('0', '1') >> (30+buf_to_u32(s))));
named!(day <u32>, alt!(day_zero | day_one | day_two | day_three));
```

```
fn main() {
    println!("Parsed day '1' as {:?}", day(b"1"));
    println!("Parsed day '10' as {:?}", day(b"10"));
    println!("Parsed day '35' as {:?}", day(b"35"));
    println!("Parsed day '40' as {:?}", day(b"40"));
    println!("Parsed day 'abc' as {:?}", day(b"abc"));
}
```

Since nom works primarily with UTF-8 strings in various forms, the helper functions (`buf_to_u32`, `to_u32`, and `to_string`) are needed to get our output into `u32`. Note the use of `unsafe` in `to_string`: it should be fairly certain that strings coming to that function are already UTF-8 safe, so encapsulating the unsafety away should be fine and produces a nicer interface without an unneeded `Result` type.

Here's the output of running this code:



The `Alt` error comes from the fact that none of the different alternatives in our call to the `alt!` macro succeeded.

Chomp

Another parser combinator library is **Chomp**. It differs from nom by being designed around the monad pattern, which makes it a bit more concise and simple than nom but also slightly less efficient and powerful.

Being based on the monad pattern simply means that the parsers are built up by sequencing parsers with optional binding of return values and possibly returning early in the case of a parsing failure. This pattern enables flexible composition of functions, which in the case of Chomp are parsers. More in-depth understanding of monads is not required to use Chomp; the following examples should make it all clear.

Chomp parsers take custom input values (that fulfill the `Input` trait) and return custom output structs (`ParseResult` or `SimpleResult`). Like nom, building these parsers is done with a macro called `parse!`. For example, here's a parser that takes a single alphanumeric string as input and returns it:

```
// chomp-test/src/main.rs
#[macro_use]
extern crate chomp;

use chomp::prelude::{U8Input, Error, ParseResult, parse_only};
use chomp::parsers::take_while;
use chomp::ascii::{is_alphanumeric, is_whitespace};
use chomp::types::Buffer;

use std::str;

fn take_string<I: U8Input>(i: I) -> ParseResult<I, String, Error<u8>> {
    parse!{i;
           let str = take_while(|c| is_alphanumeric(c) | is_whitespace(c));
           ret (String::from_utf8(str.to_vec()).unwrap())
    }
}

fn main() {
    let parsing1 = parse_only(take_string, b"Abc string with alphanumerics 123 only");
    let parsing2 = parse_only(take_string, b"A string containing non-alphanumerics");

    println!("Parsing alphanumeric string: {:?}", parsing1);
    println!("Parsing non-alphanumeric string: {:?}", parsing2);
}
```

Let's go at it with a familiar method of taking a closer look at the potentially complicated parts. First, let's look at the type of the parser function:

```
fn take_string<I: U8Input>(i: I) -> ParseResult<I, String, Error<u8>>
```

Our parser function takes an input by a generic type that implements the `U8Input` trait. This would normally not accept Rust primitive `u8` slices as input, but we'll see soon why this works anyway. The function returns values in the `ParseResult` enum, with `String` being the type of the output value. The error type is not used for anything here, so we could just as well use the `SimpleResult` type. Then the function type would be as follows:

```
fn take_string<I: U8Input>(i: I) -> SimpleResult<I, String>
```

OK. Then, we enter the `parse!` macro. As mentioned earlier, this macro is here, so we can use a coding style similar to Haskell's Parsec. The macro invocation starts with declaring the input parameter:

```
    parse!{i;
```

Next, we use the `take_while` parser. It consumes input one token (in this case, a single `u8`) at a time, as long as the condition (a closure that returns a bool) inside returns `true`:

```
        let str = take_while(|c| is_alphanumeric(c) | is_whitespace(c));
```

Next, we use the special keyword, `ret`, which is defined in the `parse!` macro. It ends the parser and returns its parameter, just like a return call would:

```
        ret (String::from_utf8(str.to_vec()).unwrap())
```

All these intermediate values, such as the `str` binding in the previous example, are of the `Buffer` type, defined by Chomp. That's why a certain amount of conversions are needed, first to a `Vector<u8>`, which can be transformed into a `String` (after unwrapping the `Result`). That's the whole parser.

Unlike nom, Chomp parsers are not called directly. Rather, a higher-order function, `parse_only`, is used:

```
    let parsing1 = parse_only(take_string, b"Abc string with alphanumerics 123 only");
```

Below is the output of running the code. The second string gets cut up, since the `-` character does not match the condition for `take_while`, so processing the input ends there.

Now that we have the basics figured out, let's try something a bit more interesting: a parser for S-expressions, used by many programming languages in the Lisp family. You've certainly seen a few S-expressions before, also in this book, but for a refresher, they look like this:

```
()
a bcd c d
(a)
(a bcd c d (e f))
```

That is, they represent arbitrary nested lists in a neat form. There are countless ways to parse these, and here's our simple algorithm:

1. Parse an empty list: `()`.
2. Parse a single value, for instance, `bcd`.
3. Parse an inner list, for instance, `(e f)` recursively with this algorithm.
4. As long as there's more input, go to `1`.

If any of the steps match, we parse the next token using that rule; otherwise, we go forward to the next one. Step three matches the inner lists recursively using this same algorithm. Here's an implementation of the algorithm using Chomp:

```
// chomp-sexpr/src/main.rs
#[macro_use]
extern crate chomp;
```

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

use chomp::prelude::{string, SimpleResult, parse_only};
use chomp::ascii::{is_alphanumeric, is_whitespace};
use chomp::types::{Buffer, U8Input};
use chomp::parsers::{take_while, skip_while};

#[derive(Debug)]
enum Svalue {
    Sexpr(Rc<Sexpr>),
    String(String),
}

#[derive(Debug)]
struct Sexpr {
    values: Vec<Svalue>,
}

fn parse_empty_inner_list<I: U8Input>(i: I) -> SimpleResult<I, Svalue> {
    parse!{i;
        string(b"(");
        string(b")");
        ret (Svalue::Sexpr( Rc::new(Sexpr { values: vec!() })))
    }
}

fn parse_inner_list<I: U8Input>(i: I) -> SimpleResult<I, Svalue> {
    parse!{i;
        string(b"(");
        let values = parse_sexpr();
        string(b")");
        ret (Svalue::Sexpr(Rc::new(values)))
    }
}

fn parse_value<I: U8Input>(i: I) -> SimpleResult<I, Svalue> {
    parse!{i;
        let value = take_while(is_alphanumeric);
        ret (Svalue::String(String::from_utf8(value.to_vec()).unwrap()))
    }
}

fn parse_svalue<I: U8Input>(i: I) -> SimpleResult<I, Svalue> {
    parse!{i;
        let svalue = parse_empty_inner_list() <|> parse_inner_list() <|> parse_value();
        skip_while(is_whitespace);
        ret (svalue)
    }
}

fn parse_sexpr<I: U8Input>(i: I) -> SimpleResult<I, Sexpr> {
    parse!{i;
        let val1 = parse_svalue();
        let val2 = parse_svalue();
        let val3 = parse_svalue();
        let val4 = parse_svalue();
        let val5 = parse_svalue();
        ret (Sexpr { values: vec!(val1, val2, val3, val4, val5 )})
    }
}


fn main() {
    let sexpr = parse_only(parse_sexpr, b"()");
    println!("Parsed: {:?}", sexpr);

    let sexpr = parse_only(parse_sexpr, b"a bcd c d");
    println!("Parsed: {:?}", sexpr);

    let sexpr = parse_only(parse_sexpr, b"(a)");
    println!("Parsed: {:?}", sexpr);

    let sexpr = parse_only(parse_sexpr, b"(a bcd c d (e f))");
    println!("Parsed: {:?}", sexpr);
```

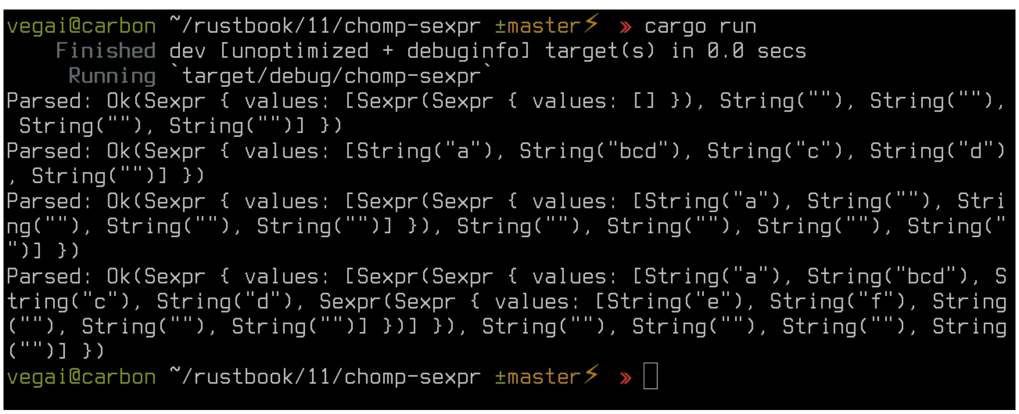
```
}
```

First, we define a few structures (enum `SValue` and struct `Sexpr`) that will hold the parsed data structure, gotten by running the parser against an arbitrary `u8` input. The values will go inside a vector, and the inner lists modeled by `SValue` point back to `Sexpr`.

Next are the individual parser parts (`parse_empty_inner_list`, `parse_value`, and `parse_inner_list`), which correspond to the different steps of the algorithm. Each individual parser returns an `SValue`, filled in with the data gotten from a parsing step. Then follows a parser that combines the previous three parsers by trying each one in turn. The <|> operator works in such a way that the first parser that matches the input is selected, and if none of the parsers match, the combined parsers is considered not to match. Note how there's a call to `parse_sexpr`, which can be thought of as being a recursive call in the context of the complete parser.

Finally, we build the root `Sexpr` structure by calling `parse_svalue` multiple times and gathering the results. That unfortunate implementation detail is due to Chomp's relative immaturity; there is a combinator called `many`, which would ideally be used to collect several parser results into a vector, but at this version, it causes an infinite loop and cannot therefore be used. Hopefully, it will be fixed in some future version.

Here's the output of running this code, showing the parsed list structures:



As you can see, another unfortunate symptom of a working `many` combinator is that our structure is littered with unneeded empty strings. Otherwise, the parser seems to work.

Other parser libraries

These are not the only parser libraries available for Rust. A brilliant source for up-to-date Rust libraries is the **Awesome Rust** project. For instance, a current list of parser libraries can be found at https://github.com/kud1ing/awesome-rust#parser.

Serde

Serde is the de facto standard generic serialization and deserialization library. Serialization means transforming a data structure into structured text (which could be, for instance, JSON, YAML, or a custom data format), while deserialization does the same in the other direction.

Note that there's an earlier library, bundled with the Rust compiler, called **rustc-serialize**. You might bump into it when looking at some older pieces of code. The rustc-serialize library is no longer needed for serialization, not since stable Rust gained macros 1.1, which fully implements the parts needed for Serde.

Using Serde is conceptually very simple. Serde has two traits: `Serialize` and `Deserialize`. Any struct that implements `Serialize` can be serialized by Serde and any struct that implements `Deserialize` can be deserialized. Serde contains code generation tools to derive these implementations, which can be used when a struct contains only types for which those traits have been already implemented. That includes at least all primitive Rust types, plus some standard reference types, such as `String`, `Vec`, and `HashMap`. This feature depends on macros 1.1, which fortunately should be stable by the time this book arrives.

Serde aligns with the Rust mantra of zero runtime cost; the serialization code has no runtime cost from reflection or from requiring type information at runtime. Therefore, a Serde implementation is probably as fast as a hand-written custom serialization method would be, but more generic and standard.

Serde's core does not support data formats but, rather, the design is such that people are free to implement support, decoupled from the core. Here's a table of some of the formats implemented by the Serde team or the Rust community:

Format	**Description**	**URL**
JSON	JavaScript Object Notation	https://github.com/serde-rs/json
YAML	A configuration language, widely used in the Ruby world	https://github.com/dtolnay/serde-yaml
TOML	An extended INI-file-like configuration language used by Cargo	https://github.com/alexcrichton/toml-rs
Msgpack	MessagePack binary serialization standard	https://github.com/3Hren/msgpack-rust

URL	URL encoding, traditionally used in HTTP requests to encode parameters to requests	https://github.com/nox/serde_urlencoded

The slightly amazing part about Serde is that in order to support all these data types, you just need to implement `Serialize` and `Deserialize`.

Let's first look at a simple example, dealing only with primitives. As a more difficult case after that, we'll write serialization support for a custom data type that cannot be derived.

Here's a pair of structs that contain only primitives that Serde can derive the serialization code from. The main function contains constructors for a few objects and then prints them out in both JSON and TOML formats:

```
#[macro_use]
extern crate serde_derive;

extern crate serde_json;
extern crate toml;

use std::collections::HashMap;

#[derive(Serialize, Deserialize, Debug)]
struct Foobar(String, f32);

#[derive(Serialize, Deserialize, Debug)]
struct DataContainer {
    #[serde(rename="SHORT NUMBER")]
    short_number: u8,
    long_number: u64,
    signed_number: i32,

    string_number_pairs: HashMap<String, u32>,
    list_of_foobars: Vec<Foobar>,
}

fn main() {
    let mut hm: HashMap<String, u32> = HashMap::new();
    hm.insert("derp".into(), 5);
    hm.insert("meep".into(), 12873);

    let foobar1 = Foobar("diip".into(), 41.12312398);
    let foobar2 = Foobar("moop".into(), 41.0);

    let dc = DataContainer {
        short_number: 43,
        long_number: 126387213,
        signed_number: -12367,

        string_number_pairs: hm,
        list_of_foobars: vec![foobar1, foobar2],
    };

    let serialized_json = serde_json::to_string_pretty(&dc).unwrap();
    let serialized_toml = toml::to_string(&dc).unwrap();
    println!("Serialized JSON:\n{}", serialized_json);
    println!("Serialized TOML:\n{}", serialized_toml);
}
```

Rather neat. We got serializations for both formats without needing to write any JSON- or TOML-specific code. Note the renamed field in the `DataContainer` struct; sometimes our serialized data is in a format that would not be pretty in Rust code, so Serde has a tag that allows renames. There are other tags for controlling how Serde behaves; a full list is available in the official documentation at https://serde.rs/attributes.html.

Here are the `Cargo.toml` dependencies that the preceding code was written with:

```
serde = "1.0"
serde_derive = "1.0"
serde_json = "1.0"
toml = { version = "0.4", default_features = false, features = ["serde"] }
```

The `toml` library needed to be configured manually to use Serde, since it defaults to using the legacy `rustc-serialize` library.

Deserialization is quite as easy. Here's how you could deserialize both the serialized forms back:

```
    let dc2: DataContainer = serde_json::from_str(&serialized_json).unwrap();
    let dc3: DataContainer = toml::from_str(&serialized_toml).unwrap();
```

Here, we can witness a detail of Serde; the format-specific serializer defines what types are supported. In this case, the `serde_json` deserializer supports our type just fine, but `toml` does not. Unfortunately this gets caught at runtime:

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/serde-test`
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: UnsupportedType', /checkout/src/libcore/result.rs:859
note: Run with `RUST_BACKTRACE=1` for a backtrace.
vegai@carbon ~/rustbook/11/serde-test ±master⚡ »                    101 ↲
```

The culprit is `Vec<Foobar>`; if we comment out everything related to it, `toml` can deserialize:

```
vegai@carbon ~/rustbook/11/serde-test ±master⚡ » cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/serde-test`
Serialized JSON:
{
  "SHORT NUMBER": 43,
  "long_number": 126387213,
  "signed_number": -12367,
  "string_number_pairs": {
    "meep": 12873,
    "derp": 5
  }
}
Serialized TOML:
"SHORT NUMBER" = 43
long_number = 126387213
signed_number = -12367

[string_number_pairs]
meep = 12873
derp = 5

vegai@carbon ~/rustbook/11/serde-test ±master⚡ »
```

That's all fine and dandy for things that we can derive the code easily for, but what happens when we have something that cannot be derived so easily? Here are the `Serialize` and `Deserialize` traits:

```
pub trait Serialize {
```

```
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer;
}

pub trait Deserialize<'de>: Sized {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer<'de>;
```

The `Serializer` and `Deserializer` traits that these methods take as parameters are exactly the traits that those who write serializers for custom data formats (such as JSON) will need to implement. Our `Serialize` and `Deserialize` (note the missing r at the end) implementations call out to those.

Let's consider writing a custom serializer for an enum that describes weekdays. This could be also derived, but Serde would end up using strings and we'd like a more compact integer representation. Check the following code first. We'll cover the details after the code:

```
// serde-custom/src/main.rs
extern crate serde;
extern crate serde_json;

use std::fmt;

use serde::ser::{Serialize, Serializer};
use serde::de;
use serde::de::{Deserialize, Deserializer, Visitor};

#[derive(Debug)]
enum Weekday {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday,
}

impl Serialize for Weekday {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer
    {
        let weekday_number = match *self {
            Weekday::Monday => 0,
            Weekday::Tuesday => 1,
            Weekday::Wednesday => 2,
            Weekday::Thursday => 3,
            Weekday::Friday => 4,
            Weekday::Saturday => 5,
            Weekday::Sunday => 6,
        };

        serializer.serialize_i32(weekday_number)
    }
}

struct WeekdayVisitor;
impl<'de> Visitor<'de> for WeekdayVisitor {
    type Value = Weekday;

    fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
       formatter.write_str("Integer between 0 and 6")
    }

    fn visit_u64<E>(self, value: u64) -> Result<Weekday, E>
        where E: de::Error
    {
        let weekday = match value {
            0 => Weekday::Monday,
```

```
            1 => Weekday::Tuesday,
            2 => Weekday::Wednesday,
            3 => Weekday::Thursday,
            4 => Weekday::Friday,
            5 => Weekday::Saturday,
            6 => Weekday::Sunday,
            _ => return Err(E::custom(format!("Number out of weekday range: {}", value))),
        };

        Ok(weekday)
    }
}

impl<'de> Deserialize<'de> for Weekday {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer<'de>
    {
        deserializer.deserialize_i32(WeekdayVisitor)
    }
}


fn main() {
    let weekday = Weekday::Tuesday;

    let serialized_json = serde_json::to_string_pretty(&weekday).unwrap();
    println!("Serialized {:?} into {}", weekday, serialized_json);

    let deserialized: Weekday = serde_json::from_str(&serialized_json).unwrap();
    println!("Got back weekday {:?}", deserialized);
}
```

The `Serializer` implementation is kind of easy: we just match against the weekday to get an `i32` and then we call the `serialize_i32` method in `Serializer`. This gives us an interesting problem for the `Deserializer` side, however. Take a look at it. `Deserializer` is based on a `Visitor` pattern, which means that `Deserializer` works by calling out to methods on the `Visitor` object it has been given, based on what sort of input it receives.

Now, in our case, we serialize the weekday into JSON, and JSON does not have any explicit type information for any of its fields. So, when we deserialize the value back to our `Weekday` struct, the deserializer makes a call to the `visit_u64` method, since that's its best guess. That's why we implement that method in the visitor.

And here again, we offer circumstantial evidence that the code is working:

```
vegai@carbon ~/rustbook/11/serde-custom » cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/serde-custom`
Serialized Tuesday into 1
Got back weekday Tuesday
vegai@carbon ~/rustbook/11/serde-custom »
```

Yay! Writing Serde serializers and, especially, deserializers seems a tad scary at first sight due to the `Visitor` pattern and generic trait types, but it's not quite so complex in practice.

Exercises

1. `combine` is a parser framework that is similar to Chomp. Translate the Chomp examples to `combine`.
2. Pest is a PEG parser generator. Implement the ISO 8601 date standard (or parts of it) using Pest.
3. Take a look at one of the Serde `Serializer`/`Deserializer` implementations, for instance, `serde_json`. How would you implement a new `Serializer` for a new data format?

Summary

In this chapter, we looked at a few parsing techniques: nom and Chomp. At the time of writing this book, it looks like nom might become some sort of a de facto standard in the Rust community, but it remains to be seen if that's true. nom focuses on efficiency, and some other parser frameworks that are more lax about that may be easier to use. For most usages, the efficiency of the parser is not very important, so an easier library may be a good idea. Be sure to check the current list of parser libraries for updates.

Serde is the de facto standard for writing serialization and deserialization code. We covered a few basic examples of using it and also covered a simple case of writing your own serializer and deserializer. Serde is an impressive piece of library that abstracts the serialization problem nicely, decoupling that code from data format-specific code neatly. On the flip side, this means that the abstraction level has to be quite high, which may make Serde a bit intimidating at first sight. Fortunately, Serde's deriving functionality makes it possible to evade most of the difficult parts.

The next chapter will be web services. We'll take a look at low-level HTTP and other web libraries, and also give examples of building services with web frameworks.

Web Programming

In this chapter, we'll look at how to work with web technologies, starting from the bottom by looking at the Hyper HTTP library. We'll use a chat service's bot API to build a simple chat bot that answers to a request. Then, we get slightly more advanced and cover Rocket, a promising new web framework that uses advanced Rust features in an attempt to compete with the more dynamic language frameworks.

This chapter covers:

- Introduction to web programming in Rust
- Hyper for client - building a Telegram Bot
- Hyper for server
- Web frameworks - Rocket and others

Introduction to Rust and web programming

Developing web programs in Rust works how one might expect for a compiled language. The application you build listens to HTTP requests. Routing logic inside the application directs those requests to handler functions, which return responses. Since Rust programs are statically linked by default, deploying such a program to a production environment means simply copying the executable over and restarting the application.

Hyper is the de facto HTTP library that most of the higher level frameworks build on. It's designed as a type-safe abstraction of raw HTTP, as opposed to a common theme in HTTP libraries: describing everything as strings. As an example, HTTP status codes are defined in `hyper::status::StatusCode`, an `enum` containing all standard status codes. The same goes for pretty much everything that can be strongly typed, such as HTTP methods, MIME types, HTTP headers, and so on.

Hyper has both client and server functionality. The client side allows what you would expect: building and executing HTTP requests with a configurable request body, headers, and lower-level settings. The server side allows opening a listening socket and attaching a single request handler to it. Notably, it does not include any request handler routing implementations; those are left to frameworks.

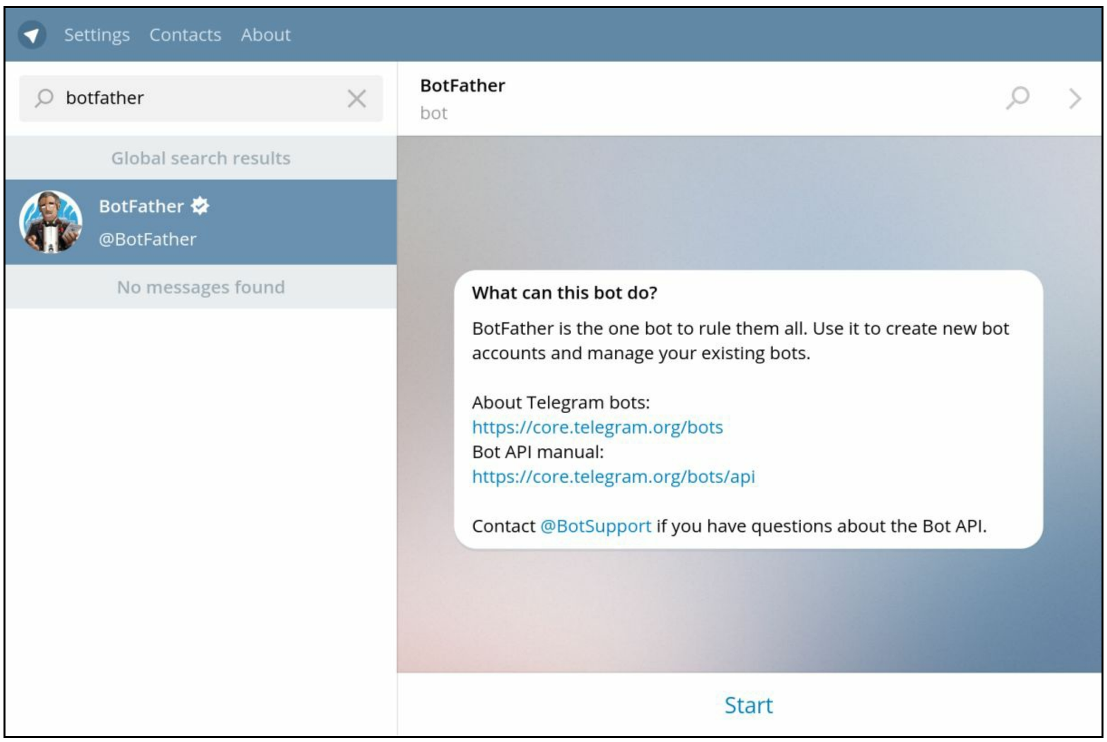
The higher level has plenty of options and is bound to have several new ones pop up at times. At the time of writing, the most prominent and stable framework is Iron, which is based on a chainable and pluggable middleware design: the core can easily be extended by third-party libraries that you plug into a router chain. Another stable one is Nickel, which is based on Express.js, and also follows a middleware architecture. The rest of the bunch are quite new and not as stable.

The data access layer is mostly handled by either low-level SQL or other datastore libraries and a few ORM-like systems, of which Diesel currently looks like the most promising. We'll cover data access a bit more in the next chapter.

Hyper as a client

We'll take a look at Hyper via something practical. Telegram is a widely used public chat platform that has an open HTTP API for creating both clients and bots. We'll use that to build an example for Hyper's client side. The full Telegram Bot API is documented in https://core.telegram.org/bots/api in case you wish to build more extended bots. Also, be sure to take a peek at https://crates.io/ for premade crates that implement the API. We'll cover just a minimal subset of it here.

First of all, we'll need to request the Telegram network to give us a new bot and a token for it. The token both identifies the bot and authenticates our usage of it, so it should be considered a secret. This is done by logging in to the Telegram network and talking to the BotFather bot. It can be found by using the search functionality of the Telegram client:



If you get this far, you're almost there. Click on Start, use the `/newbot` command, and then just follow the instructions. You should get a bot token after answering a few questions. In my case, the token I got was `276934321:AAG_4BHalBCTSIA4Z-3Auwv7MmoYC0rIK8k`.

Don't worry: by the time you're reading this, this token will not be valid anymore. Telegram API calls are always of a similar form: `https://api.telegram.org/bot<TOKEN>/<METHOD>`.

Here, `TOKEN` is exactly the string received from BotFather and `METHOD` is one of the several possible actions you can make your bot take. The parameters to the method can be passed in several ways, but we'll pass them via JSON in the HTTP request (`GET` or `POST`). We'll use structs to model that data and Serde to provide the back and forth JSON operations.

We'll start by implementing and calling the test method, `getMe`, which requires no parameters, in other words, a `POST` request to the following URL: `https://api.telegram.org/bot276934321:AAG_4BHalBCTSIA4Z-`

3Auwv7MmoYC0rIK8k/getMe.

OK, back to Rust. To make this call with Hyper, we'll first build the request and then call send on it. Here's the implementation:

```
// telegram-client/src/main.rs
extern crate hyper;
extern crate hyper_native_tls;

use hyper::client::Client;
use hyper::net::HttpsConnector;
use hyper_native_tls::NativeTlsClient;
use std::io::Read;

const API_URL: &'static str = "https://api.telegram.org";

fn main() {
    let ssl = NativeTlsClient::new().unwrap();
    let connector = HttpsConnector::new(ssl);
    let client = Client::with_connector(connector);

    let token = "276934321:AAG_4BHalBCTSIA4Z-3Auwv7MmoYC0rIK8k";

    let post_url = format!("{}/bot{}/{}", API_URL, token, "getMe");
    let request = client.post(&post_url);
    println!("PU: {}", post_url);
    let mut response = request.send().unwrap();

    let mut response_string = String::new();
    let _ = response.read_to_string(&mut response_string);

    println!("Response status: {}", response.status);
    println!("Response headers: {}", response.headers);
    println!("Response body: {}", response_string);
}
```

The only extraordinary thing here is the required use clause for the `std::io::Read trait`. This is because the `response`'s `read_to_string` method is an implementation of that trait, so the trait also needs to be in scope for all that to work. To confirm that the basis is working, here's the output from running this code:

```
vegai@carbon ~/rustbook/12/telegram-client » cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/telegram-client`
PU: https://api.telegram.org/bot276934321:AAG_4BHalBCTSIA4Z-3Auwv7MmoYC0rIK8k/ge
tMe
Response status: 200 OK
Response headers: Server: nginx/1.10.0
Date: Sat, 29 Apr 2017 14:25:10 GMT
Content-Type: application/json
Content-Length: 98
Connection: keep-alive
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET, POST, OPTIONS
Access-Control-Expose-Headers: Content-Length,Content-Type,Date,Server,Connectio
n
Strict-Transport-Security: max-age=31536000; includeSubdomains

Response body: {"ok":true,"result":{"id":276934321,"first_name":"Jeeves","userna
me":"jeeves_59832765983274_bot"}}
vegai@carbon ~/rustbook/12/telegram-client »
```

As stated previously, Telegram responses bodies are in the form of JSON.

Let's make a more interesting example: one that displays Rust's multithreading strengths a bit better.

We'll build a ChatBot command that implements a GitHub profile image search.

The Hyper client implements the `Sync` trait, which means that the same client can be shared between multiple threads. What makes this doubly useful is that Hyper's client does *connection pooling* by default. By sharing the client object between threads using Arc, Hyper opens just a single connection to the host even when sending from several threads. This is a quite useful feature for our program since it implements the chat bot part by communicating with a single host: api.telegram.org.

OK, to do all this, we'll need three things:

1. A way to receive messages from Telegram; this can be done by long-polling with the `getUpdates` method.
2. A way to make the HTTP request to GitHub and parse the return data.
3. A way to respond to the user on Telegram.

We can model this software so that each new message to the bot creates a new thread that constitutes a single chat session. In this simple case, the single chat session is launched by a new incoming message and does the following actions:

1. Make an API request to GitHub.
2. Make an image request to GitHub.
3. Respond to the message with the image.

Let's start by defining the structures that reflect the response for `getUpdates` from Telegram. The JSON response looks like this:

```
{
   "result" : [
      {
         "update_id" : 736617836,
         "message" : {
            "text" : "de",
            "from" : {
               "first_name" : "Vesa",
               "username" : "vegai",
               "id" : 64898733
            },
            "date" : 1483863182,
            "message_id" : 7,
            "chat" : {
               "type" : "private",
               "first_name" : "Vesa",
               "username" : "vegai",
               "id" : 64898733
            }
         }
      }
   ],
   "ok" : true
}
```

For our program, we can freely ignore most of that. Here are the fields we care about:

- The `update_id` field tells us the sequence of the update. We use that to tell Telegram the offset of the message we care about, so we only get unprocessed messages.

- The `message.text` field gives us the text that was sent to the bot.
- The `from.id` field is the ID that we can use for the reply message to direct it to the correct user or chat.

From that, we get the following structs:

```
// telegram-github-bot/src/main.rs
#[derive(Deserialize, Debug)]
struct TGUpdate {
    result: Vec<TGResult>
}

#[derive(Deserialize, Debug)]
struct TGResult {
    message: TGMessage,
    update_id: u64
}

#[derive(Deserialize, Debug)]
struct TGMessage {
    from: TGFrom,
    message_id: u64,
    text: Option<String>
}

#[derive(Deserialize, Debug)]
struct TGFrom {
    first_name: String,
    id: u64,
    username: String
}
```

As you might remember, we don't need to define all fields that the input has in our structs, just the ones we care about. The text fields of the `TGMessage` struct needs to be optional since many types of messages don't include it, and we don't want to crash the deserialization when that happens.

Next up, here's the function that fetches a single set of updates from Telegram:

```
// telegram-github-bot/src/main.rs
fn find_largest_offset(update: &TGUpdate) -> u64 {
    let mut i = 0;
    for r in &update.result {
        if r.update_id > i {
            i = r.update_id
        }
    }
    i
}

fn get_updates(client: &Client, token: &str, next_offset: u64) -> (TGUpdate, u64) {
    let post_url = format!("{}/bot{}/{}?offset={}&timeout={}",
                           API_URL,
                           token,
                           "getUpdates",
                           next_offset,
                           POLL_INTERVAL);
    let request = client.post(&post_url);
    let mut response = request.send().unwrap();

    let mut response_string = String::new();
    let _ = response.read_to_string(&mut response_string);
    let data: TGUpdate = serde_json::from_str(&response_string).unwrap();

    let last_seen_offset = find_largest_offset(&data);
    (data, last_seen_offset)
}
```

This function makes a request to `API_URL` (which is a const defined elsewhere in the file) and the `getUpdates` method. The parameters are passed as URL query parameters. `POLL_INTERVAL`, another const, defines in seconds how long the poll should be if nothing happens on the server side. Then the function deserializes the response to our `TGUpdate` struct, figures out the largest offset value (`update_id`) in that result set, and returns both the data and the found offset.

Next up, this piece of code gets the image from GitHub and replies to the user via Telegram:

```
// telegram-github-bot/src/main.rs
fn get_github_image_url(client: &Client, text: &str) -> String {
    let url = format!("https://api.github.com/users/{}", text);
    let request = client
        .get(&url)
        .header(UserAgent("JeevesMasteringRustBot-0.1".into()));
    let mut response = request.send().unwrap();

    let mut response_string = String::new();
    let _ = response.read_to_string(&mut response_string);
    let data: Value = serde_json::from_str(&response_string).unwrap();

    let github_url = data
        .as_object().unwrap().get("avatar_url").unwrap();
    github_url.as_str().unwrap().into()
}

fn reply_with_image(client: &Client, token: &str, id: u64, image_url: &str) {
    println!("Replying to {}", id);
    let post_url = format!("{}/bot{}/{}?chat_id={}&photo={}",
                           API_URL,
                           token,
                           "sendPhoto",
                           id,
                           image_url
    );

    let request = client.post(&post_url);
    let _ = request.send();
}
```

This should be quite familiar; the whole program is just a bunch of glue between different HTTP requests. The first function, `get_github_image_url`, makes a `GET` request to GitHub and extracts the `avatar_url` line from the response. GitHub, unlike Telegram, requires a `UserAgent` to be set, so we do that there. The second function then sends the photo by its URL.

The `main` function ties all this together:

```
// telegram-github-bot/src/main.rs
fn main() {
    let token = "276934321:AAG_4BHalBCTSIA4Z-3Auwv7MmoYC0rIK8k";
    let c = Arc::new(Client::new());
    let mut next_offset = 0u64;

    loop {
        let (updates, last_seen_offset) = get_updates(&c, token, next_offset);
        next_offset = last_seen_offset + 1;

        for result in updates.result {
            let c = c.clone();
            let github_user = result.message.text.clone().unwrap();

            thread::spawn(move || {
                let github_image_url = get_github_image_url(&c, &github_user);
                reply_with_image(
                    &c,
```

```
                    token,
                    result.message.from.id,
                    &github_image_url)
            });
        }
    }
}
```

Here, we get the new updates from Telegram in the main loop and launch separate threads to do the message handling. Notice how we have a single Hyper client object for all HTTP communications, and it's shared via an Arc. Rust guarantees that the usage is safe, and Hyper implements nice internal optimizations via its implementation of the `Sync` trait. The shared client uses concurrent client pools, so it may make connections to api.telegram.org and api.github.com concurrently, but not more than one for each.

The hyper pool size is configurable (see http://hyper.rs/hyper/v0.9.12/hyper/client/struct.Client.html#method.with_pool_config) in case you wish your clients to make more than one connection to a single host.

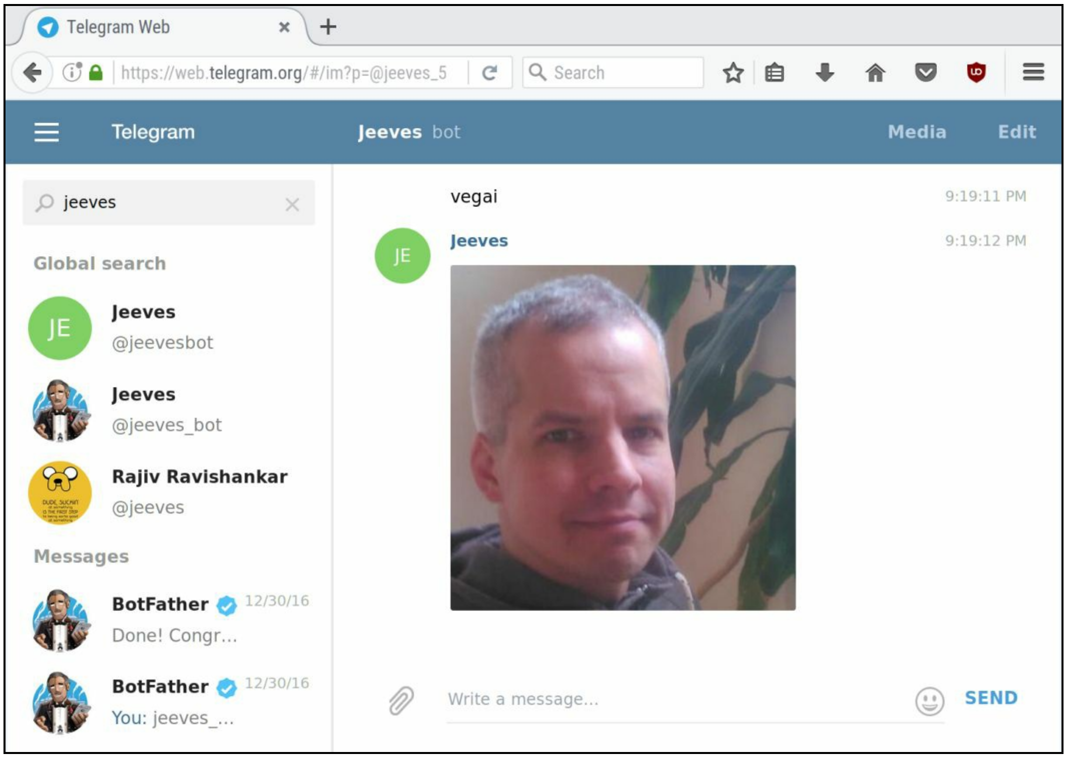
Here's the code in full:

```
// telegram-github-bot/src/main.rs
extern crate hyper;
extern crate serde_json;
#[macro_use]
extern crate serde_derive;
extern crate serde;

use std::thread;
use std::sync::Arc;
use hyper::client::Client;
use hyper::header::UserAgent;
use std::io::Read;
use serde_json::Value;

const API_URL: &'static str = "https://api.telegram.org";
const POLL_INTERVAL: u64 = 60;

#[derive(Deserialize, Debug)]
struct TGUpdate {
    result: Vec<TGResult>
}

#[derive(Deserialize, Debug, Clone)]
struct TGResult {
    message: TGMessage,
    update_id: u64
}

#[derive(Deserialize, Debug, Clone)]
struct TGMessage {
    from: TGFrom,
    message_id: u64,
    text: Option<String>,

}

#[derive(Deserialize, Debug, Clone)]
struct TGFrom {
    first_name: String,
    id: u64,
    username: String
}

fn get_github_image_url(client: &Client, text: &str) -> String {
    let url = format!("https://api.github.com/users/{}", text);
    let request = client
```

```
        .get(&url)
        .header(UserAgent("JeevesMasteringRustBot-0.1".into()));
    let mut response = request.send().unwrap();

    let mut response_string = String::new();
    let _ = response.read_to_string(&mut response_string);
    let data: Value = serde_json::from_str(&response_string).unwrap();

    let github_url = data
        .as_object().unwrap().get("avatar_url").unwrap();
    github_url.as_str().unwrap().into()
}

fn reply_with_image(client: &Client, token: &str, id: u64, image_url: &str) {
    println!("Replying to {}", id);
    let post_url = format!("{}/bot{}/{}?chat_id={}&photo={}",
                           API_URL,
                           token,
                           "sendPhoto",
                           id,
                           image_url
    );

    let request = client.post(&post_url);
    let _ = request.send();
}

fn find_largest_offset(update: &TGUpdate) -> u64 {
    let mut i = 0;
    for r in &update.result {
        if r.update_id > i {
            i = r.update_id
        }
    }
    i
}

fn get_updates(client: &Client, token: &str, next_offset: u64) -> (TGUpdate, u64) {
    let post_url = format!("{}/bot{}/{}?offset={}&timeout={}",
                           API_URL,
                           token,
                           "getUpdates",
                           next_offset,
                           POLL_INTERVAL);
    let request = client.post(&post_url);
    let mut response = request.send().unwrap();

    let mut response_string = String::new();
    let _ = response.read_to_string(&mut response_string);
    let data: TGUpdate = serde_json::from_str(&response_string).unwrap();

    let last_seen_offset = find_largest_offset(&data);
    (data, last_seen_offset)
}

fn main() {
    let token = "276934321:AAG_4BHalBCTSIA4Z-3Auwv7MmoYC0rIK8k";
    let c = Arc::new(Client::new());
    let mut next_offset = 0u64;

    loop {
        let (updates, last_seen_offset) = get_updates(&c, token, next_offset);
        next_offset = last_seen_offset + 1;

        for result in updates.result {
            let c = c.clone();
            let github_user = result.message.text.clone().unwrap();

            thread::spawn(move || {
                let github_image_url = get_github_image_url(&c, &github_user);
                reply_with_image(
                    &c,
                    token,
                    result.message.from.id,
```

```
                        &github_image_url)
                });
            }
        }
}
```

Here is an example of interacting with this bot in Telegram:



Hyper as a server

Let's go to the other aspect of Hyper: serving HTTP content. The basic usage is simple: you define a listening address and attach a handler to it. Since it may be called from several threads by Hyper, the handler that it needs to implement the `Sync` trait. Hyper provides a simple implementation for functions and closures, which takes a parameter for the `Request` and a `Response`. So, a simple logging HTTP server could look like this:

```
extern crate hyper;
extern crate chrono;

use hyper::server::{Server, Request, Response};
use hyper::header::UserAgent;
use hyper::status::StatusCode;
use chrono::*;

fn log_request(req: Request, mut res: Response) {
    let date = UTC::now();
    let user_agent = req.headers.get::<UserAgent>().unwrap();
    let mut status = res.status_mut();
    *status = StatusCode::Ok;

    println!("{} {} \"{} {} {}\" {} \"{}\"",
             req.remote_addr,

             date,
             req.method,
             req.uri,
             req.version,
             status,
             user_agent);
}

fn main() {
    Server::http("0.0.0.0:8123").unwrap().handle(log_request).unwrap();
}
```

Take note of the strong typing on these two lines, and the slightly peculiar way of getting headers:

```
let user_agent = req.headers.get::<UserAgent>().unwrap();
...
*status = StatusCode::Ok;
```

Since the headers and response status codes are defined by HTTP, they have all been declared in the `hyper::status::StatusCode` enum and as separate structs in `hyper::header`. In case you require non-standard status codes, the `StatusCode` enum has an `Unregistered(u16)` instance, which could be used like this in the preceding example:

```
    let mut status = res.status_mut();
    *status = StatusCode::Unregistered(420);
```

In the case of custom headers, you can fetch the headers struct from both the `Request` and the `Response`, and manipulate the headers via them with the `get_raw` and `set_raw -methods`, as follows:

```
   let user_agent = str::from_utf8(&req.headers.get_raw("User-Agent").unwrap()[0]).unwrap();
   ...
   let mut headers = res.headers_mut();
         headers.set_raw("XFoo-Header", vec![b"merp".to_vec()]);
```

It can be done, but as you can see, the usage is not exactly fun anymore, partly due to all the safety we have to circumvent and partly due to Hyper's API being less supportive in this case.

Let's head over to a higher level next and try out the Rocket web framework.

Rocket

The most promising web framework at the moment is called **Rocket**. It uses advanced Rust features in order to achieve a level of flexibility and agility found in more dynamic languages. We'll use Rocket to make a very simple RPG arena game using just basic HTML features.

Rocket currently requires nightly Rust since it uses several unstable features, such as compiler plugins, custom derives and attributes, and a few others we haven't talked about before. It will probably stay in the nightly territory for quite a while but is worth checking out, because it's wildly more ergonomic than the more stable frameworks.

Our game will have a global HashMap of `Character` structs, protected by the familiar Arc and Mutex combination to allow access from several threads. We use Rocket's dynamic ability to include the global game state inside request handlers by implementing its `FromRequest` trait for our struct.

Let's begin with the data structures. We initialize the global part of the game state by using the `lazy_static` crate. It has a macro that allows us to initialize a static global with a function. Here's the usage:

```
// arena-web/src/main.rs
lazy_static! {
    pub static ref CHARACTERS: Characters = Arc::new(Mutex::new(HashMap::new()));
}
```

The struct for a single `Character` is familiar:

```
// arena-web/src/main.rs
#[derive(Serialize, Clone)]
pub struct Character {
    name: String,
    strength: u8,
    dexterity: u8,
    hitpoints: u8
}
```

Then we have a structure for a `CharacterForm`. Rocket serializes the incoming HTML-form input into this structure. Note the `FromForm` derivation:

```
// arena-web/src/main.rs
#[derive(FromForm, Debug)]
struct CharacterForm {
    name: String
}
```

The global `Characters` type is a bit complex, so we use a type alias so that we don't have to type it out (not to mention reading it!) several times. The `GameState` struct wraps this type:

```
// arena-web/src/main.rs
type Characters = Arc<Mutex<HashMap<String, Character>>>;
struct GameState {
    players: Characters
}
```

We need to implement the `FromRequest` trait for `GameState`, so we can leverage Rocket's infrastructure, which allows passing arbitrary structs into the request handler. We just clone the players field from the global `CHARACTERS` Arc and wrap it in the `Outcome` struct that Rocket expects. The error type is moot here, since this operation cannot fail, but it needs to be there, nevertheless:

```
// arena-web/src/main.rs
impl<'a, 'r> FromRequest<'a, 'r> for GameState {
    type Error = std::fmt::Error;
    fn from_request(_: &'a Request<'r>) -> Outcome<Self, Self::Error> {
        Success(GameState{ players: CHARACTERS.clone() })
    }
}
```

Finally, the `GameViewData` struct models the dynamic showable data that we display on the main playing page. The flash string displays the effects of the previous combat action (if any), and the `CHARACTERS` HashMap is used to display information for all the existing characters. This needs to derive Serde's `Serialize` trait in order to be usable in the HTML template:

```
// arena-web/src/main.rs
#[derive(Serialize)]
pub struct GameViewData {
    flash: String,
    characters: HashMap<String, Character>
}
```

That's all the data we need. Next up are the request handlers. Rocket has a neat custom derive syntax for defining routes, familiar to anyone who has worked with the Python Flask framework. Here's the entry page handler:

```
// arena-web/src/main.rs
#[get("/")]
fn index(cookies: &Cookies) -> Redirect {
    match cookies.find("character_id") {
        Some(_) => Redirect::to("/game"),
        None => Redirect::to("/new")
    }
}
```

The route matches `GET /` requests. The `Cookies` structure implements the `FromRequest` trait, so we can just pass it in like this. The handler checks if a cookie with a `character_id` exists (and points to this player's character). If it does, we go right into the game, and if it doesn't, we redirect to a new character creation page. Its request handler is quite simple:

```
// arena-web/main.src
#[get("/new")]
fn new() -> Template {
    Template::render("index", &())
}
```

It just renders a template called `new`. By default, Rocket looks for template files from under the `templates` directory of your project. It supports both the **handlebars** and **tera templating** libraries, and it looks for both in that directory. Files ending with `.hbs` are interpreted as handlebars templates, while files ending with `.tera` are interpreted as tera templates. Our templates are handlebars files. Here's what `templates/index.hbs`, in all its simplicity, looks like:

```
<!-- arena-web/index.hbs -->
<!DOCTYPE html>
<html>
    <body>
        <h1>New character</h1>
        <form method="POST" action="/new">
            Name <input type="text" name="name" />
        </form>
    </body>
</html>
```

Nothing dynamic there yet, just static HTML.

Next up is the `POST` handler for new, which handles the form input from the previous page:

```
// arena-web/src/main.rs
#[post("/new", data = "<character_form>")]
fn post_new(cookies: &Cookies, character_form: Form<CharacterForm>, state: GameState) -> Result<Redirect, Strin
    let character = character_form.get();

    let mut rng = rand::thread_rng();
    let new_character_id: String = rng.gen::<u64>().to_string();

    let ref mut players = *state.players.lock().unwrap();
    players.insert(new_character_id.clone(), Character {
        name: character.name.clone(),
        strength: rng.gen::<u8>(),
        dexterity: rng.gen::<u8>(),
        hitpoints: rng.gen::<u8>()
    });

    cookies.add(Cookie::new("character_id".into(), new_character_id));
    Ok(Redirect::to("/game"))
}
```

We give an arbitrary data input name in the route declaration and refer to it in the function parameter list. We also get the `GameState` struct here and it all works out due to the `FromRequest` implementation we provided earlier. However, we'll have to manually lock the struct here to use it. Then we randomly generate a new `Character`, put in the given name from the HTML form, and set the `character_id` cookie, so it will be remembered in the next request. Then, we redirect to the game page. Here's its request handler:

```
// arena-web/src/main.rs
#[get("/game")]
fn game(state: GameState, flash: Option<FlashMessage>) -> Template {
 let players = state.players.clone();
 let characters = players.lock().unwrap();
 let flash: String = match flash {
 Some(f) => f.msg().into(),
 None => "".into()
 };
    Template::render("game", &GameViewData { flash: flash, characters: characters.clone() } )
}
```

Here, we take the global players struct, so we can display them and also display an optional flash message. The flash is used for describing the effect of a player attacking another player. The handlebars template is a bit more dynamic this time:

```
// arena-web/templates/game.hbs
<!DOCTYPE html>
<html>
    <body>
        {{#if flash}}
```

```
            {{flash}}
        {{/if}}
        <h1>Characters in game</h1>
        <ul>
            {{#each characters}}
                <li>
                    <p>
                    <form action="/attack/{{@key}}" method="POST">
 {{name}} str: {{strength}} dex: {{dexterity}} hp: {{hitpoints}}
                        <button>⚔</button>
                    </form>
                    </p>
                </li>
            {{/each}}
        </ul>
    </body>
</html>
```

We display the flash if it exists and then a list of all `Characters` in the game, each followed by an attack link. The attack handler is the most complicated one of this set:

```
// arena-web/src/main.rs
#[post("/attack/<id>")]
fn attack(cookies: &Cookies, state: GameState, id: &str) -> Flash<Redirect> {
    let ref mut players = *state.players.lock().unwrap();

    let attacker_id = cookies.find("character_id").unwrap().value;
    let attacker = players.get(&attacker_id).unwrap().clone();
    let defender = players.get(id).unwrap().clone();

    let mut rng = rand::thread_rng();
    let damage: i16 = attacker.strength as i16 - defender.dexterity as i16 + rng.gen::<i8>() as i16;

    let message = if damage < 1 {
        format!("{} missed {}", attacker.name, defender.name)
    } else if defender.hitpoints as i16 - damage < 1 {
        players.remove(id);
        format!("{} hits {}. {} is slain!", attacker.name, defender.name, defender.name)
    } else {
        let new_defender = Character {
            name: defender.name.clone(),
            strength: defender.strength,
            dexterity: defender.dexterity,
            hitpoints: defender.hitpoints - damage as u8
        };
        players.insert(id.into(), new_defender);
        format!("{} hits {}.", attacker.name, defender.name)
    };

    Flash::error(Redirect::to("/game"), message)
}
```

The target of the attack is given directly in the URL, and Rocket routes that into the function variable of the same name. We then fetch both the attacker and defender structs from the global map and clone them to circumvent potential problems with the borrow checker. Then, we do a naive calculation to figure out the inflicted damage and change the global structure based on the results. There's another point where we go around the borrow checker a bit; we need to change the defender's hitpoints in the second if branch, but it doesn't have interior mutability. So, we just create a new `Character`, instead, with the same data except for a changed hitpoints field, and we replace the old key in the HashMap.

Then, we do a redirect back to `/game`, with a flash that tells what happened on the next page.

Finally, we have the `main` function, which mounts all these routes to the root path and launches the listener:

```
// arena-web/src/main.rs
fn main() {
    rocket::ignite().
        mount("/", routes![index, new, post_new, game, attack, static_files]).
        launch();
}
```

Here's how the excitement looks in action:

Rocket gets quite near to the web frameworks of more dynamic languages such as Ruby and Python, and at the time of writing this book, its version is 0.1.15. This is an impressive feat even if the reliance on many nightly Rust features is a bit unfortunate.

Here's the full code listing of this program:

```
// arena-web/src/main.rs
#![feature(plugin, custom_derive)]
#![plugin(rocket_codegen)]

#[macro_use]
extern crate lazy_static;
extern crate rocket;
extern crate rocket_contrib;
extern crate rand;
extern crate time;
extern crate serde;
#[macro_use]
extern crate serde_derive;

use std::collections::HashMap;
use std::sync::{Arc, Mutex};

use rocket::request::{Outcome, Form, FromRequest, Request, FlashMessage};
use rocket::Outcome::Success;
use rocket::response::{Flash, Redirect, NamedFile};
use rocket_contrib::Template;
use rocket::http::{Cookie, Cookies};
use time::get_time;

use std::path::{Path, PathBuf};
use std::io;


use rand::Rng;
```

```
lazy_static! {
    pub static ref CHARACTERS: Characters = Arc::new(Mutex::new(HashMap::new()));
}

#[derive(Serialize, Clone)]
pub struct Character {
    name: String,
    strength: u8,
    dexterity: u8,
    hitpoints: u8
}
#[derive(FromForm, Debug)]
struct CharacterForm {
    name: String
}

type Characters = Arc<Mutex<HashMap<String, Character>>>;
struct GameState {
    players: Characters
}

impl<'a, 'r> FromRequest<'a, 'r> for GameState {
    type Error = std::fmt::Error;
    fn from_request(_: &'a Request<'r>) -> Outcome<Self, Self::Error> {
        Success(GameState{ players: CHARACTERS.clone() })
    }
}

#[derive(Serialize)]
struct GameViewData {
    flash: String,
    characters: HashMap<String, Character>
}
#[get("/")]
fn index(cookies: &Cookies) -> Redirect {
    match cookies.find("character_id") {
        Some(_) => Redirect::to("/game"),
        None => Redirect::to("/new")
    }
}

#[get("/new")]
fn new() -> Template {
    Template::render("index", &())
}

#[post("/new", data = "<character_form>")]
fn post_new(cookies: &Cookies, character_form: Form<CharacterForm>, state: GameState) -> Result<Redirect, Strin
    let character = character_form.get();
    println!("char: {:?}", character);

    let mut rng = rand::thread_rng();
    let new_character_id: String = rng.gen::<u64>().to_string();

    let ref mut players = *state.players.lock().unwrap();
    players.insert(new_character_id.clone(), Character {
        name: character.name.clone(),
        strength: rng.gen::<u8>(),
        dexterity: rng.gen::<u8>(),
        hitpoints: rng.gen::<u8>()
    });

    cookies.add(Cookie::new("character_id".into(), new_character_id));
cookies.add(Cookie::new("throttle", get_time().sec.to_string()));
    Ok(Redirect::to("/game"))
}


#[get("/game")]
fn game(state: GameState, flash: Option<FlashMessage>) -> Template {
  let players = state.players.clone();
  let characters = players.lock().unwrap();
  let flash: String = match flash {
```

```
    Some(f) => f.msg().into(),
    None => "".into()
  };

 let g_v_d = GameViewData { flash: flash, characters: characters.clone() };
 Template::render("game", &g_v_d)
}

#[post("/attack/<id>")]
fn attack(cookies: &Cookies, state: GameState, id: &str) -> Flash<Redirect> {
    let ref mut players = *state.players.lock().unwrap();
    let throttle: i64 = cookies.find("throttle").unwrap().value().parse().unwrap();
    println!("throttle is {}, current time is {}", throttle, get_time().sec);
    if throttle >= get_time().sec {
      return Flash::error(Redirect::to("/game"), "Attacking too fast!");
 }

    let attacker_cookie = cookies.find("character_id").unwrap();
    let attacker_id = cookies.find("character_id").unwrap().value;
    let attacker = players.get(&attacker_id).unwrap().clone();
    let defender = players.get(id).unwrap().clone();

    let mut rng = rand::thread_rng();
    let damage: i16 = attacker.strength as i16 - defender.dexterity as i16 + rng.gen::<i8>() as i16;

    let message = if damage < 1 {
        format!("{} missed {}", attacker.name, defender.name)
    } else if defender.hitpoints as i16 - damage < 1 {
        players.remove(id);
        format!("{} hits {}. {} is slain!", attacker.name, defender.name, defender.name)
    } else {
        let new_defender = Character {
            name: defender.name.clone(),
            strength: defender.strength,
            dexterity: defender.dexterity,
            hitpoints: defender.hitpoints - damage as u8
        };
        players.insert(id.into(), new_defender);
        format!("{} hits {}.", attacker.name, defender.name)
    };

    Flash::error(Redirect::to("/game"), message)
}
#[get("/<path..>", rank=1)]
fn static_files(path: PathBuf) -> io::Result<NamedFile> {
 NamedFile::open(Path::new("static/").join(path))
}
fn main() {
    rocket::ignite().
        mount("/", routes![index, new, post_new, game, attack, static_files]).
        launch();
}
```

`Cargo.toml` needs a few tweaks over the typical listing of dependencies to get the handlebars template support working alright:

```
[package]
name = "arena-web"
version = "0.1.0"
authors = ["Vesa Kaihlavirta <vegai@iki.fi>"]

[dependencies]
rocket = "0.2"
rocket_codegen = "0.2"
rocket_contrib = { version = "0.2", default-features = false, features = ["handlebars_templates"] }
lazy_static = "0.2"
rand = "0.3"
serde = "0.9"
serde_derive = "0.9"
```

After these, running the program is a simple invocation of `cargo run`. Remember, if you wish to benchmark Rust programs such as this one, it's imperative that you turn on optimizations by using the `--release` flag.

Other web frameworks

The first successful frameworks were Iron and Nickel.

Iron is designed around a strictly minimal core, with an extension mechanism around the middleware and simpler plugins and modifiers. Nearly everything it does is handled by these extensions, be it logging, body parsing, session handling, and so on. Iron uses no fancy Rust features, which means that it is sturdy, stable, and has no problems running in even an older version of the stable Rust compiler. Iron is at http://ironframework.io.

Nickel is designed with the same concept as Express.js, a prominent JavaScript framework. It also supports a middleware design but uses macros a bit heavier than Iron. Nickel can be found at http://nickel.rs.

There are a couple more frameworks out there and the list is growing larger all the time. One final point of interest is the upcoming WebAssembly standard, which is trying to set common guidelines for building a common low-level language for browser frontend programming. Rust's features, its lack of a mandatory garbage collector most of all, makes it resonate well with this enterprise, so this may be an interesting area to keep an eye on.

A complete overview (and an always current view of the current state of Rust web libraries and frameworks) is published at http://arewewebyet.org. Be sure to check that out to see what's available at any point in time.

Exercises

1. Extend the arena game by showing more information about all the characters in the main `GET /game` page.
2. The arena game loses all its state when the program is restarted. A typical technique for a web application is to store all the state in a relational database. Think of other ways. What would be the simplest way? Implement it.
3. Implement a pause in the area game so that people cannot just spam the attack link.
4. Make the game prettier: serve a static CSS and link it to the templates.
5. Take a look at http://arewewebyet.org. Go through and review the current libraries.
6. Try building the arena game on some of the more classic frameworks such as Iron or Nickel, or by using only Hyper and coding everything yourself.

Summary

Rust hasn't traditionally been a strong web language. It certainly will not give you the same development speed as the dynamic languages do, at least for now. In web programming, Rust's way of bringing errors to the programmer earlier is psychologically a difficult situation: any kind of progress feels better when you're working on a problem. Trying to appease an angry compiler can be a discouraging feeling, but we need to remember that the bugs we fix with the help of the compiler are bugs that we don't have to see at production.

Rust has the potential of giving higher performance, quality, and security to web applications.

In the next chapter, we'll take a look at data persistence through the Diesel ORM and a few other data-oriented libraries.

Data Storage

Rust does not offer any data storage natively, but the community has built several crates for accessing third-party data storages. In this chapter, we'll take a look at some of them.

We will cover the following topics in this chapter:

- SQLite
- The `rust-postgres`
- The r2d2 for connection pooling
- Object-relation mapping using Diesel
- Combining Diesel with Rocket

SQLite

There are a couple of options for connecting to SQLite. We'll go with John Gallagher's Rusqlite, which is available at https://github.com/jgallagher/rusqlite. Its API could be called mid-level: it helpfully hides many of the details of the actual SQLite API but is not laden with high-level abstractions such as struct mappings.

In contrast to many other relational database systems, SQLite's type system is dynamic. This means that columns do not have types but each individual value does. Technically, SQLite separates storage classes from data types, but this is mainly an implementation detail, and we may just think in terms of types without being too far from the truth.

Rusqlite provides **FromSql** and **ToSql** traits for converting objects between SQLite and Rust code. Also, it provides the following implementations out of the box:

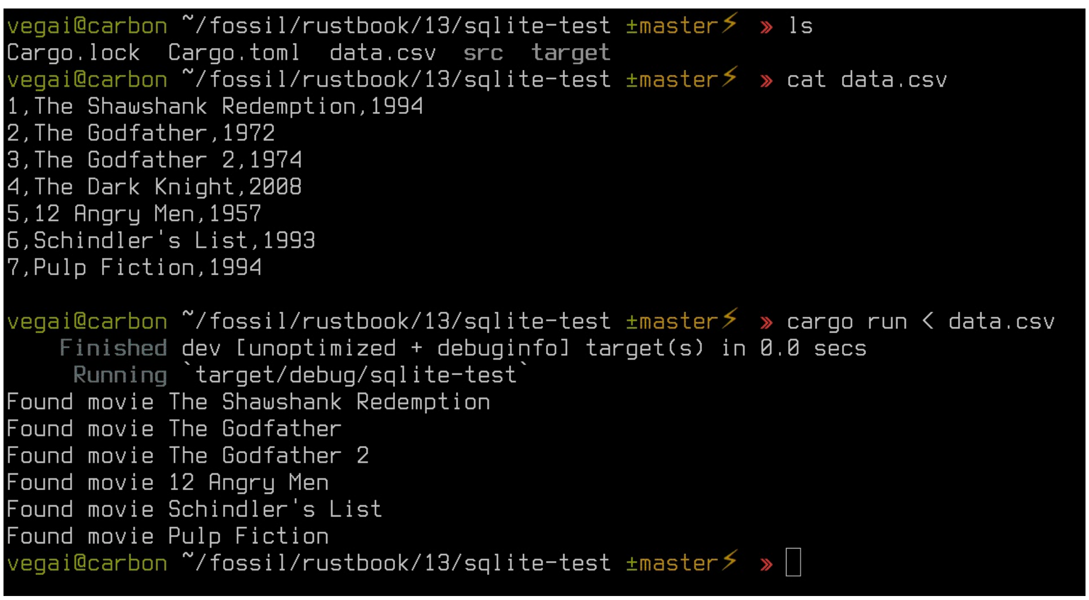
Description	**SQLite**	**Rust**
The null value	`NULL`	`rusqlite::types::Null`
1, 2, 3, 4, 6 or 8 byte signed integers	`INTEGER`	`i32` (with possible truncation) and `i64`
8-byte IEEE floats	`REAL`	`f64`
UTF-8, UTF-16BE or UTF-16LE strings	`TEXT`	`String` and `&str`
Bytestrings	`BLOB`	`Vec<u8>` and `&[u8]`

Here's a program that takes a properly formatted CVS file from standard input, stores it in SQLite, and then retrieves a subset of the data:

```
extern crate rusqlite;
use std::io::BufRead;
use std::io;

use rusqlite::Connection;

struct Movie {
    name: String
}

fn main() {
    let conn = Connection::open_in_memory().unwrap();
```

```
    conn.execute("CREATE TABLE movie (
                  id              INTEGER PRIMARY KEY,
                  name            TEXT NOT NULL,
                  year            INTEGER NOT NULL
                 )", &[]).unwrap();
    let stdin = io::stdin();
    for line in stdin.lock().lines() {
        let elems = line.unwrap();
        let elems: Vec<&str> = elems.split(",").collect();
        if elems.len() > 2 {
            let _ = conn.execute("INSERT INTO movie (id, name, year) VALUES (?1, ?2, ?3)",
                        &[&elems[0], &elems[1], &elems[2]]);
        }
    }

    let mut stmt = conn.prepare("SELECT name FROM movie WHERE year < ?1").unwrap();
    let movie_iter = stmt.query_map(&[&2000], |row|
                                Movie {
                                    name: row.get(0)
                                }
    ).unwrap();
    for movie in movie_iter {
        println!("Found movie {}", movie.unwrap().name);
    }
}
```

First, we create an in-memory SQLite table and read all the entries to it from the standard input. Then, we make a query to the database, getting all the movies from before the year 2000. Given the following input, here's the output of the program:



This is generally a rather neat technique for quickly transforming seemingly dumb data into a form where it can be more flexibly queried.

PostgreSQL

While SQLite is fine for the smaller problems, a real relational database can make the life of a business application coder much easier. The most popular PostgreSQL library is simply named `postgres`. It's a native Rust client, meaning that it does not ride on a `C` library but implements the whole protocol in Rust. If the API looks familiar, it is deliberate; the SQLite client's API is actually based on that of the PostgreSQL client.

The driver supports some of PostgreSQL's interesting features, such as bit vectors, time fields, JSON support, and UUIDs. Our example will use the time fields, a UUID for a primary key, and the JSON type for generic data.

We'll start by doing the shortest possible PostgreSQL initialization. How this works is dependent on the operating system: some systems create the initial databases automatically, and perhaps even start the server too. Anyway, this is how it's done from Arch Linux, with the PostgreSQL server already installed:

```
vegai@carbon ~/fossil/rustbook/13/sqlite-test ±master⚡ » sudo -u postgres -i
[postgres@carbon ~]$ initdb --locale $LANG -E UTF8 -D'/var/lib/postgres/data'
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "en_US.UTF-8".
The default text search configuration will be set to "english".

Data page checksums are disabled.

fixing permissions on existing directory /var/lib/postgres/data ... ok
creating subdirectories ... ok
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting dynamic shared memory implementation ... posix
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

WARNING: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the option -A, or
--auth-local and --auth-host, the next time you run initdb.

Success. You can now start the database server using:

    pg_ctl -D /var/lib/postgres/data -l logfile start

[postgres@carbon ~]$ exit
logout
vegai@carbon ~/fossil/rustbook/13/sqlite-test ±master⚡ » sudo systemctl start
pstgresql
vegai@carbon ~/fossil/rustbook/13/sqlite-test ±master⚡ »
```

In addition to starting the database, you might need to add a database user. We'll use the default `postgres` user in the following examples, but in case you have some trouble with the examples, go check PostgreSQL documentation on how to create a new user. You'll need to adapt the following examples to match your new user.

Then to our Rust project. Since we want to use the optional features, `Cargo.toml` needs to be a little different:

```
# postgres-test/Cargo.toml
[package]
name = "postgres-test"
version = "0.1.0"
authors = ["Vesa Kaihlavirta <vegai@iki.fi>"]

[dependencies.postgres]
git = "https://github.com/sfackler/rust-postgres"
features = ["with-bit-vec", "with-chrono", "with-uuid", "with-serde_json", "with-uuid"]

[dependencies]
uuid={ version="0.4", features = ["serde", "v4"] }
serde="0.9"
serde_json="0.9"
serde_derive="0.9"
chrono={ version="0.3", features = ["serde"] }
```

At the time of writing this book, we needed the `git` master version of `rust-postgres` to get the latest Serde support over UUID. By the time this book is out, using the latest stable release will probably work better. The `uuid` and `chrono` crates require explicitly declared Serde support, and the UUID crate additionally requires the v4 feature in order to gain the generation of version 4 UUIDs.

Here's the code:

```
// postgres-test/src/main.rs
extern crate postgres;
extern crate uuid;
#[macro_use]
extern crate serde_derive;
#[macro_use]
extern crate serde_json;
extern crate chrono;

use postgres::{Connection, TlsMode};
use uuid::Uuid;
use chrono::DateTime;
use chrono::offset::utc::UTC;

#[derive(Debug, Serialize, Deserialize)]
struct Person {
    id: Uuid,
    name: String,
    previous_time_of_lunch: DateTime<UTC>,
    data: Option<serde_json::Value>,
}

fn main() {
    let conn = Connection::connect("postgres://postgres@localhost", TlsMode::None).unwrap();
    conn.execute("CREATE TABLE IF NOT EXISTS person (
                    id                  UUID PRIMARY KEY,
                    name             VARCHAR NOT NULL,
                    previous_time_of_lunch  TIMESTAMP WITH TIME ZONE,
                    data                JSONB
                  )", &[]).unwrap();
    let me = Person {
        id: Uuid::new_v4(),
        name: "Steven".to_string(),
        previous_time_of_lunch: UTC::now(),
        data: Some(json!({
            "tags": ["employee", "future_ceo"],
            "contacts": {
                "email": "steven@employer.com"
            }
        }))
    };
    conn.execute("INSERT INTO person (id, name, data, previous_time_of_lunch) VALUES ($1, $2, $3, $4)",
```

```
            &[&me.id, &me.name, &me.data, &me.previous_time_of_lunch]).unwrap();
    for row in &conn.query("SELECT id, name, data, previous_time_of_lunch FROM person", &[]).unwrap() {
        let person = Person {
            id: row.get(0),
            name: row.get(1),
            data: row.get(2),
            previous_time_of_lunch: row.get(3)
        };
        println!("Found person {:?}", person);
        println!("{}'s email: {}", person.name, person.data.unwrap()["contacts"]["email"]);
        println!("{}'s last lunch: {}", person.name, person.previous_time_of_lunch);

    }
}
```

First, we connect to the database:

```
let conn = Connection::connect("postgres://postgres@localhost", TlsMode::None).unwrap();
```

We declare a non-encrypted connection to make the example simpler and then unwrap the object. This, of course, means that if the connection fails, our program ends right here.

Then, we create the table `person` in the database by executing SQL:

```
conn.execute("CREATE TABLE IF NOT EXISTS person (
                    id                  UUID PRIMARY KEY,
                    name              VARCHAR NOT NULL,
                    previous_time_of_lunch  TIMESTAMP WITH TIME ZONE,
                    data                JSONB
                  )", &[]).unwrap();
```

Here's how the PostgreSQL crate maps the PostgreSQL types into Rust types:

Field	PostgreSQL type	Rust type
id	UUID	uuid::Uuid
name	VARCHAR	String
previous_time_of_lunch	TIMESTAMP WITH TIMEZONE	DateTime<UTC>
data	JSONB	serde_json::Value

Both the `DateTime` and `UTC` structs originate from the `chrono` crate. Obviously, the `DateTime` generic could take other timezones too and the PostgreSQL type will be unchanged. For further information about the different types, take a look at both PostgreSQL's excellent documentation at https://www.postgresql.org/docs/current/static/datatype.html and the `rust-postgres` GitHub repo at https://github.com/sfackler/rust-postgres.

The `conn.execute` function takes two parameters, where the first one is the wanted SQL statement and the second one is an array of possible bindable arguments. Since Rust does not have optional arguments, we have to provide a reference to an empty list: `&[]`. Then, we unwrap the response, which again means that if the expression fails, we get a clean exit from the program at this point.

Then, we create a `Person` object that we want to insert into the database:

```
let me = Person {
    id: Uuid::new_v4(),
    name: "Steven".to_string(),
    previous_time_of_lunch: UTC::now(),
    data: Some(json!({
        "tags": ["employee", "future_ceo"],
        "contacts": {
            "email": "steven@employer.com"
        }
    }))
};
```

Because we are not using an **object-relational mapper** (**ORM**) here, but just a low-level interface, we'll need to unpack the values into the database query manually:

```
conn.execute("INSERT INTO person (id, name, data, previous_time_of_lunch) VALUES ($1, $2, $3, $4)", &[&me.id, &
```

Here we see how the binding of parameters works in the queries. After this, we run a query to get all the persons from the table, build `person` objects from them, and output some data:

```
for row in &conn.query("SELECT id, name, data, previous_time_of_lunch FROM person", &[]).unwrap() {
    let person = Person {
        id: row.get(0),
        name: row.get(1),
        data: row.get(2),
        previous_time_of_lunch: row.get(3)
    };
    println!("Found person {:?}", person);
    println!("{}'s email: {}", person.name, person.data.unwrap()["contacts"]["email"]);
    println!("{}'s last lunch: {}", person.name, person.previous_time_of_lunch);
}
```

Although the PostgreSQL crate is quite low level, note that the more specialized types are handled for you; `id`, `data`, and `previous_time_of_lunch` get converted properly.

Here's an output of the program, along with a `psql` query of the table to show the contents afterwards:

```
vegai@carbon ~/fossil/rustbook/13/postgres-test ±master⚡ » cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/postgres-test`
Found person Person { id: Uuid("8204c24d-00a6-48c6-b52e-9825298b1180"), name: "
Steven", previous_time_of_lunch: 2017-04-29T20:22:32.902248Z, data: Some(Object
({"contacts": Object({"email": String("steven@employer.com")}), "tags": Array([
String("employee"), String("future_ceo")])})) }
Steven's email: "steven@employer.com"
Steven's last lunch: 2017-04-29 20:22:32.902248 UTC
Found person Person { id: Uuid("1b8cef71-a05b-4971-b0f5-36c4facfe485"), name: "
Steven", previous_time_of_lunch: 2017-04-29T20:22:40.036284Z, data: Some(Object
({"contacts": Object({"email": String("steven@employer.com")}), "tags": Array([
String("employee"), String("future_ceo")])})) }
Steven's email: "steven@employer.com"
Steven's last lunch: 2017-04-29 20:22:40.036284 UTC
vegai@carbon ~/fossil/rustbook/13/postgres-test ±master⚡ »psql --user=postgres

psql (9.6.2)
Type "help" for help.

postgres=# \x on
Expanded display is on.
postgres=# select * from person;
-[ RECORD 1 ]----------+------------------------------------------------------
---------------------------
id                     | 8204c24d-00a6-48c6-b52e-9825298b1180
name                   | Steven
previous_time_of_lunch | 2017-04-29 23:22:32.902248+03
data                   | {"tags": ["employee", "future_ceo"], "contacts": {"ema
il": "steven@employer.com"}}
-[ RECORD 2 ]----------+------------------------------------------------------
---------------------------
id                     | 1b8cef71-a05b-4971-b0f5-36c4facfe485
name                   | Steven
previous_time_of_lunch | 2017-04-29 23:22:40.036284+03
data                   | {"tags": ["employee", "future_ceo"], "contacts": {"ema
il": "steven@employer.com"}}

postgres=#
```

That's good enough for low-level basics. Next, let's check out how we can make connection pools.

Connection pooling with r2d2

If an application requires any sort of efficiency, opening and closing a database connection every time something happens becomes a bottleneck really fast. A technique called connection pooling helps with this; when a process needs a new connection, a pool gives out an existing connection if any exists, and when a process no longer needs the connection, it hands it over back to the pool.

A crate called **r2d2** provides a generic way of maintaining such connections. Currently, it contains backend support for PostgreSQL, Redis, MySQL, SQLite, and a few lesser known options. We'll cover r2d2 here by first checking out how to connect to PostgreSQL using a pool and then by investigating how to implement a pool for our own simple data storage.

The architecture of r2d2 is formed of two parts: a generic part and a backend specific part. The backend code attaches to the generic part by implementing r2d2's `ManageConnection` trait and by adding a connection manager for the specific backend. The trait looks as follows:

```
pub trait ManageConnection: Send + Sync + 'static {
    type Connection: Send + 'static;
    type Error: Error + 'static;
    fn connect(&self) -> Result<Self::Connection, Self::Error>;
    fn is_valid(&self, conn: &mut Self::Connection) -> Result<(), Self::Error>;
    fn has_broken(&self, conn: &mut Self::Connection) -> bool;
}
```

That is, we need to specify a `Connection` type (that needs to be sendable to another thread), an `Error` type, and three methods. Connection types come from the underlying backend mechanism; for instance, this would be `postgres::Connection` for the PostgreSQL backend. The `Error` type is an enum that specifies all the possible `Error` cases that may happen during either the connection phase or while checking the validity of the connection.

We could just convert our example in the previous section to use a pool, but that would be slightly boring. The point of a pool is to allow efficient access to the database from several threads. So, what we'll do is change the example to using threads: one for creating the table, one for inserting, and several for reading the tables. For the sake of simplicity, we'll use a bad synchronization technique: sleeping. *Don't do that in real code!*

Unfortunately, since `r2d2-postgres` does not support all the optional features that the `postgres` crate does, we dumb down the types a bit and make the special types just `VARCHAR` characters in the database and strings in our code. Here's the full code of a pooled and threaded implementation using `r2d2-postgres`:

```
// postgres-r2d2-test/src/main.rs
extern crate postgres;
extern crate r2d2;
extern crate r2d2_postgres;
extern crate uuid;
#[macro_use]
extern crate serde_json;
extern crate chrono;

use r2d2_postgres::{TlsMode, PostgresConnectionManager};
use chrono::offset::utc::UTC;
```

```rust
use std::{thread, time};
use std::thread::sleep;

#[derive(Debug)]
struct Person {
    name: String,
    previous_time_of_lunch: String,
    data: String
}

type DbConnection = r2d2::PooledConnection<PostgresConnectionManager>;

fn create_table(conn: DbConnection) {
    conn.execute("CREATE TABLE IF NOT EXISTS person (
                    id              SERIAL PRIMARY KEY,
                    name            VARCHAR NOT NULL,
                    previous_time_of_lunch  VARCHAR NOT NULL,
                    data            VARCHAR NOT NULL
                  )", &[]).unwrap();
}

fn insert_person(conn: DbConnection) {
    let me = Person {
        name: "Steven".to_string(),
        previous_time_of_lunch: UTC::now().to_string(),
        data: json!({
            "tags": ["employee", "future_ceo"],
            "contacts": {
                "email": "steven@employer.com"
            }
        }).to_string()
    };
    conn.execute("INSERT INTO person (name, data, previous_time_of_lunch) VALUES ($1, $2, $3)",
                 &[&me.name, &me.data, &me.previous_time_of_lunch]).expect("Table creation");

}

fn main() {
    let config = r2d2::Config::builder()
        .pool_size(5)
        .build();
    let manager = PostgresConnectionManager::new("postgres://postgres@localhost", TlsMode::None).unwrap();
    let pool = r2d2::Pool::new(config, manager).unwrap();

    {
        let pool = pool.clone();
        thread::spawn(move || {
            let conn = pool.get().unwrap();
            create_table(conn);
            println!("Table creation thread finished.");
        });
    }

    {
        let pool = pool.clone();
        thread::spawn(move || {
            sleep(time::Duration::from_millis(500));
            let conn = pool.get().unwrap();
            insert_person(conn);
            println!("Person insertion thread finished.");
        });
    }

    sleep(time::Duration::from_millis(1000));
    {
        for _ in 0..1024 {
            let pool = pool.clone();
            thread::spawn(move || {
                let conn = pool.get().unwrap();
                for row in &conn.query("SELECT id, name, data, previous_time_of_lunch FROM person", &[]).unwrap
                    let person = Person {
                        name: row.get(1),
                        data: row.get(2),
                        previous_time_of_lunch: row.get(3)
```

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

The pool size is configured at `5`, which means that the `SELECT` query threads get to do five things concurrently, which is much better than one, which would happen without the pool.

Diesel

Writing a complex application using just low-level libraries is a recipe for a lot of mistakes. Diesel is an ORM and a query builder for Rust. It makes heavy use of derive macros as provided by Macros 1.1. Diesel detects most database interaction mistakes at compile time and is able to produce very efficient code in most cases, sometimes even beating low-level access with C. This is due to its ability to move checks that are typically made at runtime to compile time.

To start with Diesel, we'll need to add Diesel and its code generator library in our `Cargo.toml`, along with the `dotenv` library. This library handles the local configuration via dotfiles. We'll use Diesel to store the same person information as with the previous PostgreSQL example. Here's a `Cargo.toml` for our example application:

```
# diesel-test/Cargo.toml
[package]
name = "diesel-test"
version = "0.1.0"
authors = ["Vesa Kaihlavirta <vegai@iki.fi>"]

[dependencies]
diesel = { version = "0.10", features = ["postgres", "uuid", "serde_json"] }
diesel_codegen = { version = "0.10", features = ["postgres"] }
dotenv = "0.8"
uuid = { version="0.4", features = ["serde", "v4"] }
serde_json = "0.9"
```

As you can see, we'll need to specify which database we are targeting in the features for both `diesel` and `diesel_codegen` in addition to the features that we'll need from the database. We'll skip PostgreSQL timestamps and the chrono crate for now, since at the time of writing this book, Diesel support for these was not quite reliable yet.

Diesel has a command line application, `diesel_cli`, which will be essential here. This program is used to quickly set up and reset a development database, and to manipulate and execute database migrations. You can install this like any other Rust binary crate:

```
cargo install diesel_cli
```

`Cargo` will fetch and build `diesel_cli` and its dependencies, and install the binary to Cargo's default binary location for your user, which is usually the following:

```
~/.cargo/bin/
```

Next, we'll need to tell Diesel about the database credentials. By convention, this information will be stored in `.env` of the project root. Enter the following text there: `DATABASE_URL=postgres://postgres@localhost/diesel_test`.

More generally, the format for this URL is as follows: `<backend>://<user>[:<password>]@[address]/[database]`.

Since our database did not have a password, we could omit it. Now, run Diesel setup to create the database:

```
vegai@carbon ~/fossil/rustbook/13/diesel-test ±master⚡ » diesel setup
Creating database: diesel_test
Running migration 20170212154505
vegai@carbon ~/fossil/rustbook/13/diesel-test ±master⚡ »
```

Now that the basics are set up, let's take a breather here and talk a bit about the overall structure and idea of Diesel. It uses macros quite heavily to give us some good stuff, such as extra safety and performance. To be able to do this, it needs compile time access to the database. This is why the `.env` file is important: Diesel (or rather, its `infer_schema!` macro) connects to the database and generates bridging code between our Rust code and the database. This gives us compile-time guarantees that the database and our model code are in sync; plus, it allows the Diesel library to skip some runtime checks that other libraries require for general sanity.

The models are generally split into query and insert structs. Both use derive macros from Macros 1.1 to generate model code for different use cases.

In addition to this, a Diesel application needs code to connect to the database and a set of database migrations to build up and maintain the database tables.

Let's add a `migration` that creates our table now. As mentioned earlier, the `diesel` command helps here; it generates a version for a new migration and creates empty up and down migration files:

```
vegai@carbon ~/fossil/rustbook/13/diesel-test ±master⚡ » diesel migration genera
te create_person

Creating migrations/20170429222341_create_person/up.sql
Creating migrations/20170429222341_create_person/down.sql
vegai@carbon ~/fossil/rustbook/13/diesel-test ±master⚡ »
```

The migration files are just plain SQL, so we can just put in our earlier `CREATE TABLE` command:

```
CREATE TABLE person (
       id UUID PRIMARY KEY,
       name VARCHAR NOT NULL,
       previous_time_of_lunch TIMESTAMP WITH TIME ZONE,
       data JSONB
)
```

The down file should contain a corresponding `DROP TABLE`:

```
DROP TABLE person
```

OK, now we can write a model for this table. In Diesel, models could live in any visible module, but we'll go with the convention of putting them in `src/models.rs`. Here are its contents for our model person:

```
// diesel-test/src/models.rs
extern crate uuid;
extern crate serde_json;

use uuid::Uuid;
use serde_json;
use schema::person;
```

```
#[derive(Queryable)]
pub struct Person {
    pub id: Uuid,
    pub name: String,
    pub data: Option<serde_json::Value>,
}

#[derive(Insertable)]
#[table_name="person"]
pub struct NewPerson<'a> {
    pub id: Uuid,
    pub name: &'a str,
}
```

Note how we define two structs here: one for querying and another for inserting. The primary reason is that inserting and querying are quite different operations often; in this case, our insert side will need less information than the query side.

As a bit of a boilerplate, we'll need `src/schema.rs`, but the contents of that file get mostly generated by Diesel for us. That's why the file will only need this line:

```
infer_schema!("dotenv:DATABASE_URL");
```

Next, we'll have to write the method that creates a new connection to PostgreSQL and a method that inserts a new person into that database. This code goes into `src/lib.rs`, and here it is:

```
// diesel-test/src/lib.rs
#[macro_use] extern crate diesel;
#[macro_use] extern crate diesel_codegen;
#[macro_use] extern crate serde_json;
extern crate dotenv;
extern crate uuid;

pub mod schema;
pub mod models;

use diesel::prelude::*;
use diesel::pg::PgConnection;
use dotenv::dotenv;
use std::env;
use uuid::Uuid;

use self::models::{Person, NewPerson};

pub fn establish_connection() -> PgConnection {
    dotenv().ok();

    let database_url = env::var("DATABASE_URL")
        .expect("DATABASE_URL must be set");
    PgConnection::establish(&database_url)
        .expect(&format!("Error connecting to {}", database_url))
}

pub fn create_person<'a>(conn: &PgConnection, name: &'a str) -> Person {
    use schema::person;

    let new_person = NewPerson {
        id: Uuid::new_v4(),
        name: name
    };

    diesel::insert(&new_person).into(person::table)
        .get_result(conn)
        .expect("Error saving person")
}
```

Finally, we need a main program to tie all this together. In here, we have three private functions for inserting a person, updating it, and querying the result. All use their own connection to emphasize the fact that the database receives the changes properly:

```
// diesel-test/src/main.rs
extern crate diesel_test;
extern crate diesel;
extern crate uuid;
#[macro_use] extern crate serde_json;

use diesel::prelude::*;
use diesel_test::*;
use diesel_test::models::Person;
use diesel_test::schema::person::dsl::{person, name, data};
use uuid::Uuid;

fn new_person(new_name: &str) -> Uuid {
    let conn = establish_connection();
    let new_person = create_person(&conn, new_name);
    println!("Saved person: {}", new_person.name);
    new_person.id
}

fn add_data_to_person(uuid: Uuid, person_data: &serde_json::Value) {
    let conn = establish_connection();
    let steven_from_db = person.find(uuid);
    diesel::update(steven_from_db)
        .set(data.eq(person_data))
        .get_result::<Person>(&conn)
        .expect(&format!("Unable to change data on id {}", uuid));
}

fn find_all_with_name(needle: &str) -> Vec<Person> {
    let conn = establish_connection();
    person.filter(name.eq(needle))
        .load::<Person>(&conn)
        .expect(&format!("Error summoning {}", needle))
}

fn main() {
    let stevens_id = new_person("Steven");

    add_data_to_person(stevens_id, &json!({
        "tags": ["dangerous", "polite"]
    }));

    let all_stevens = find_all_with_name("Steven");
    println!("List of all found Stevens:");
    for steven in all_stevens {
        println!("id: {}", steven.id);
        match steven.data {
            Some(d) =>
                println!("data: {}", d),
            None =>
                println!("No data.")
        };
    }
}
```

Diesel is still under construction but already provides a nice basis for high-level database operations. Thanks to it only needing Macros 1.1 functionality, it works today on stable Rust, so it can be quite safely used for production code already. The barrier to entry is a tad high currently, but the situation should improve as more and more examples and literature come along.

Summary

We took a quick look at a few ways to do basic database interaction using Rust, by using the low-level SQLite and PostgreSQL library crates. We saw how to augment the database connectivity with a connection pool using r2d2. Finally, we made a small application with Diesel, the safe and performant ORM.

In the next chapter, we'll take a look at debugging Rust code.

Debugging

In this chapter, we'll cover debugging using external debuggers. Since Rust at runtime is close enough to how C programs work at runtime, we can leverage the debuggers used in those circles: GDB and LLDB. This will be a very practice-oriented chapter, where we walk through some of the basic debugging commands and workflows.

We'll also cover editor integration via Visual Studio Code using GDB.

The following topics will be covered in this chapter:

- Introduction to debugging
- GDB basics and practice
- GDB with threads
- LLDB basics
- GDB integration to Visual Studio Code

Introduction to debugging

Your program does not work, and you have no idea why. Worry not, for you are not alone: everyone who is a programmer has been there. How we fix this depends quite a lot on what kind of tools the ecosystem gives us.

- **Debug printing**: Just sprinkle `print` statements near the places where you suspect the bug is happening. Simple, crude, and often effective, but it is definitely supported in virtually every ecosystem. This technique requires no extra tools, and everybody knows how to do it.
- **REPL-based debugging**: The **read-eval-print loop** gives you a flexible interface where you can load any of your libraries and run code in them in a safe session. Extremely useful, especially if you've managed to properly compartmentalize your code into functions. Unfortunately, Rust does not have an official REPL, and its overall design does not really support one. Nevertheless, this is a popular request, and a tracking RFC at https://github.com/rust-lang/rfcs/issues/655 shows the current progress on it.
- **External debugger**: Without needing to do any actual code changes to your program, the program is compiled with debugging information included in the resulting binary. This instrumentation code allows an external program to attach to your program, manipulate it, and observe it while it's running.

The first option is so trivial that we don't need to go through it here. We shall also skip REPLs since, although there are third-party programs that implement it, they do not work with the latest compilers well enough to be practical development tools. This leaves us only the third item, debuggers, to focus on.

You'll need to have these debuggers installed locally on your computer. Usually, they are installed by default, but if they aren't yet, refer to your operating system's instructions on how to do it. Typically it's just a matter of running an installation command such as `apt-get install gdb` or `brew install gdb`.

Let's delve into it by investigating how the basic tooling works. Here's the program we will reflect on:

```
// broken-program-1/src/main.rs
extern crate num;
use num::Num;

fn multiply_numbers_by<N: Num + Copy>(nums: Vec<N>, multiplier: N) -> Vec<N> {
    let mut ret = vec!();
    for num in nums {
        ret.push(num * multiplier);
    }
    ret
}

fn main() {
    let nums: Vec<f64> = vec!(1.5, 2.192, 3.0, 4.898779, 5.5, -5.0);
    println!("Multiplied numbers: {:?}", multiply_numbers_by(nums, 3.141));
}
```

The `multiply_numbers_by` function is generic over most number types by using the generic type `Num` from the `num` crate. Remember to add the dependency to `Cargo.toml` when building this program.

The generic parameter `N` is required to implement the `Copy` type in addition to `Num`. This is because, without it, the multiplier parameter would get moved in the for loop's first iteration and it would not be usable in any future iterations. The main program feeds the function a list of floats and a multiplier of the same type.

We'll use the two most used debuggers of the open source world: `gdb` and `lldb`. Both of these tools are featureful enough to warrant several books of their own, so we will offer just a basic, Rust-related tour here.

GDB - basics

Let's start from the command line. A binary that we'll debug needs specific additional instrumentation that the tools latch on to. This instrumentation gives us essential runtime information such as the ability to match the source code to the running binary code, and so on. Running the debugger is possible against a release build too, but the selection of operations is very much limited. As you've seen several times before, the binaries Cargo and rustc builds for us by default reside in the `target/debug/` directory. So, that is covered for us already.

Rust comes with wrappers for both debuggers: `rust-gdb` and `rust-lldb`. Here's how to get to GDB's prompt after building the project:

```
vegai@carbon ~/rustbook/14/broken-program-1 ±master⚡ » rust-gdb target/debug/br
oken-program-1
GNU gdb (GDB) 7.12.1
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from target/debug/broken-program-1...done.
(gdb)
```

GDB now obediently awaits our commands. For a first sanity check, let's just run the program and see the outcome:

```
(gdb) run
Starting program: /home/vegai/fossil/rustbook/14/broken-program-1/target/debug/b
roken-program-1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
Multiplied numbers: [4.7115, 6.885072000000001, 9.423, 15.387064839, 17.2755, -1
5.705]
[Inferior 1 (process 6748) exited normally]
(gdb)
```

All right, the basics work. You should restart `rust-gdb` again after this command because it ran and fully returned from your program already.

Let's take a quick look at the scope of GDB's features. Try the commands, `help` (which displays high-level sections of commands) and `help all` (which displays a short help message for all available commands):

```
(gdb) help all

Command class: aliases

ni -- Step one instruction
rc -- Continue program being debugged but run it in reverse
rni -- Step backward one instruction
rsi -- Step backward exactly one instruction
si -- Step one instruction exactly
stepping -- Specify single-stepping behavior at a tracepoint
tp -- Set a tracepoint at specified location
tty -- Set terminal for future runs of program being debugged
where -- Print backtrace of all stack frames
ws -- Specify single-stepping behavior at a tracepoint

Command class: breakpoints

awatch -- Set a watchpoint for an expression
break -- Set breakpoint at specified location
break-range -- Set a breakpoint for an address range
catch -- Set catchpoints to catch events
catch assert -- Catch failed Ada assertions
catch catch -- Catch an exception
catch exception -- Catch Ada exceptions
catch exec -- Catch calls to exec
catch fork -- Catch calls to fork
catch load -- Catch loads of shared libraries
catch rethrow -- Catch an exception
catch signal -- Catch signals by their names and/or numbers
catch syscall -- Catch system calls by their names
catch throw -- Catch an exception
---Type <return> to continue, or q <return> to quit---
```

OK, so GDB can do a lot: there are 32 pages of these commands. Fortunately, we need only a couple: what we want it to do right now is step through the program, line by line, and potentially inspect the internal data structures at each point. To do that, we'll first need the list command, found in the files section of the `help` file. Here's its description:

```
(gdb) help list
List specified function or line.
With no argument, lists ten more lines after or around previous listing.
"list -" lists the ten lines before a previous ten-line listing.
One argument specifies a line, and ten lines are listed around that line.
Two arguments with comma between specify starting and ending lines to list.
Lines can be specified in these ways:
  LINENUM, to list around that line in current file,
  FILE:LINENUM, to list around that line in that file,
  FUNCTION, to list around beginning of that function,
  FILE:FUNCTION, to distinguish among like-named static functions.
  *ADDRESS, to list around the line containing that address.
With two args, if one is empty, it stands for ten lines away from
the other arg.

By default, when a single location is given, display ten lines.
This can be changed using "set listsize", and the current value
can be shown using "show listsize".
(gdb)
```

Note that Rust names are prefixed by their crate, so, to display the `multiply_numbers_by` function, listing by just that name doesn't work. You'll need the crate name, which in our case is the name of the program, `broken_program_1`. Another thing we need to realize here is that at runtime there are no generic functions. The compiler does monomorphization here, which means creating specialized functions from generic ones. In practice, this means that the full name of our function at runtime is `broken_program_1::multiply_numbers_by<f64>`:

```
(gdb) break 7
Breakpoint 1 at 0x928c: file /home/vegai/fossil/rustbook/14/broken-program-1/src
/main.rs, line 7.
(gdb) run
Starting program: /home/vegai/fossil/rustbook/14/broken-program-1/target/debug/b
roken-program-1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, broken_program_1::multiply_numbers_by<f64> (
    nums=Vec<f64>(len: 6, cap: 6) = {...}, multiplier=3.141)
    at /home/vegai/fossil/rustbook/14/broken-program-1/src/main.rs:7
7               ret.push(num * multiplier);
(gdb)
```

GDB gave us the lines of code for the function, plus a few lines around it. We then just pressed *Enter* a couple of times; doing this in GDB often means *repeat what I just did with a reasonable default*, which, in this case, meant displaying the following lines as long as there were any. Now, the line numbers we see here are absolute line numbers that we can use to refer to lines of code. We can use this information to insert breakpoints, that is, a place where the running of the code stops. Let's say we want to see what the `ret` vector is holding at every point of the loop, so we first insert a break before the loop and start the program:

```
(gdb) break 5
Breakpoint 1 at 0x9168: file /home/vegai/fossil/rustbook/14/broken-program-1/src
/main.rs, line 5.
(gdb) run
Starting program: /home/vegai/fossil/rustbook/14/broken-program-1/target/debug/b
roken-program-1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, broken_program_1::multiply_numbers_by<f64> (
    nums=Vec<f64>(len: 6, cap: 6) = {...}, multiplier=3.141)
    at /home/vegai/fossil/rustbook/14/broken-program-1/src/main.rs:5
5           let mut ret = vec!();
(gdb)
```

OK, we now stopped on the line we set the break point at. This means that that line has not been executed yet. We'll introduce three more commands: `next`, `print`, and `continue`. Next continues to the next line in the source code, without pausing to descend downwards in the call stack. There's also a command step, which also stops on each execution `print` in a function call, but that's not what we're interested in now. The `print` command tries to give a representation of its argument. The `continue` command executes the rest of the code to the end. See how they work:

```
(gdb) run
Starting program: /home/vegai/fossil/rustbook/14/broken-program-1/target/debug/b
roken-program-1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, broken_program_1::multiply_numbers_by<f64> (
    nums=Vec<f64>(len: 6, cap: 6) = {...}, multiplier=3.141)
    at /home/vegai/fossil/rustbook/14/broken-program-1/src/main.rs:5
5           let mut ret = vec!();
(gdb) print ret
No symbol 'ret' in current context
(gdb) next
6           for num in nums {
(gdb) print ret
$1 = Vec<f64>(len: 0, cap: 0)
(gdb) next
8           }
(gdb) next
6           for num in nums {
(gdb) print ret
$2 = Vec<f64>(len: 0, cap: 0)
(gdb) next
7               ret.push(num * multiplier);
(gdb) next
6           for num in nums {
(gdb) print ret
$3 = Vec<f64>(len: 1, cap: 4) = {4.7115}
(gdb) continue
Continuing.
Multiplied numbers: [4.7115, 6.885072000000001, 9.423, 15.387064839, 17.2755, -1
5.705]
[Inferior 1 (process 6966) exited normally]
(gdb)
```

As mentioned before, the execution was stopped before the mut variable was initialized, so we could not print it yet. Afterwards, the `print` command shows us how the vector grows, also conveniently displaying the internal `len` and `cap` values along with the content. When we have seen enough of this, we continue to the end of the program.

GDB - threads

The debuggers can swim in multithreaded programs quite as well as single-threaded ones. Here's the previous program, now doing its calculations in threads:

```
// broken-program-2/src/main.rs
extern crate num;
use num::Num;
use std::thread;
use std::sync::{Arc, Mutex};

fn multiply_and_store<N: Num + Copy + Send>(nums: Arc<Mutex<Vec<N>>>, num: N, multiplier: N) {
    let mut data = nums.lock().unwrap();
    data.push(num * multiplier);
}

fn multiply_numbers_by<N: 'static + Num + Copy + Send>(nums: Vec<N>, multiplier: N, ret: Arc<Mutex<Vec<N>>>) {
    let mut threads = vec!();

    for num in nums {
        let ret = ret.clone();
        threads.push(thread::spawn(move || {
            multiply_and_store(ret, num, multiplier);
        }));
    }

    for thread in threads {
        let _ = thread.join();
    }
}

fn main() {
    let nums: Vec<f64> = vec!(1.5, 2.192, 3.0, 4.898779, 5.5, -5.0);
    let multiplied_nums = Arc::new(Mutex::new(vec!()));
    multiply_numbers_by(nums, 3.141, multiplied_nums.clone());

    println!("Multiplied numbers: {:?}", multiplied_nums);
}
```

So, essentially the bound on the generic type `N` was augmented with the static lifetime and the `Send` trait. Also, we had to move the return value (now wrapped in good old Arc and Mutex for multithreaded safety) to main in order to prevent a lifetime of troubles with the return value.

OK, let's fire up the debugger again and run the program on it:

```
vegai@carbon ~/rustbook/14/broken-program-2 ±master⚡ » rust-gdb target/debug/br
ken-program-2
GNU gdb (GDB) 7.12.1
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from target/debug/broken-program-2...done.
(gdb) run
Starting program: /home/vegai/fossil/rustbook/14/broken-program-2/target/debug/b
roken-program-2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
[New Thread 0x7ffff6bff700 (LWP 7281)]
[New Thread 0x7ffff67fe700 (LWP 7282)]
[Thread 0x7ffff6bff700 (LWP 7281) exited]
[New Thread 0x7ffff61ff700 (LWP 7283)]
[Thread 0x7ffff67fe700 (LWP 7282) exited]
[New Thread 0x7ffff5ffe700 (LWP 7284)]
[Thread 0x7ffff61ff700 (LWP 7283) exited]
[New Thread 0x7ffff5dfd700 (LWP 7285)]
[Thread 0x7ffff5ffe700 (LWP 7284) exited]
[New Thread 0x7ffff5bfc700 (LWP 7286)]
[Thread 0x7ffff5dfd700 (LWP 7285) exited]
[Thread 0x7ffff5bfc700 (LWP 7286) exited]
Multiplied numbers: Mutex { data: [4.7115, 6.885072000000001, 9.423, 15.38706483
9, 17.2755, -15.705] }
[Inferior 1 (process 7277) exited normally]
(gdb) 
```

We see six new threads being created and finishing. LWP is short for light-weight process, which refers to threads created by your operating system's kernel. Now, we'll try some breakpoints, one before launching the threads, another one inside the threads:

```
(gdb) break 8
Breakpoint 1 at 0x111e7: file /home/vegai/fossil/rustbook/14/broken-program-2/sr
c/main.rs, line 8.
(gdb) break 13
Breakpoint 2 at 0x113a8: /home/vegai/fossil/rustbook/14/broken-program-2/src/mai
n.rs:13. (2 locations)
(gdb) run
Starting program: /home/vegai/fossil/rustbook/14/broken-program-2/target/debug/b
roken-program-2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 2, broken_program_2::multiply_numbers_by<f64> (
    nums=Vec<f64>(len: 6, cap: 6) = {...}, multiplier=3.141,
    ret=Arc<std::sync::mutex::Mutex<collections::vec::Vec<f64>>> = {...})
    at /home/vegai/fossil/rustbook/14/broken-program-2/src/main.rs:14
14          for num in nums {
(gdb) next
19          }
(gdb) next
14          for num in nums {
(gdb) next

Breakpoint 2, broken_program_2::multiply_numbers_by<f64> (
    nums=Vec<f64>(len: 6, cap: 6) = {...}, multiplier=3.141,
    ret=Arc<std::sync::mutex::Mutex<collections::vec::Vec<f64>>> = {...})
    at /home/vegai/fossil/rustbook/14/broken-program-2/src/main.rs:14
14          for num in nums {
(gdb) next
15              let ret = ret.clone();
(gdb) next
16              threads.push(thread::spawn(move || {
(gdb) next
[New Thread 0x7ffff6bff700 (LWP 7375)]
[Switching to Thread 0x7ffff6bff700 (LWP 7375)]

Thread 2 "broken-program-" hit Breakpoint 1, broken_program_2::multiply_and_stor
e<f64> (nums=Arc<std::sync::mutex::Mutex<collections::vec::Vec<f64>>> = {...},
    num=1.5, multiplier=3.141)
    at /home/vegai/fossil/rustbook/14/broken-program-2/src/main.rs:8
8           data.push(num * multiplier);
(gdb) 
```

Now, we can investigate the active threads with the `info threads` command, after which we remove the breakpoints and let the program run its course:

```
(gdb) info threads
  Id   Target Id         Frame
  1    Thread 0x7ffff7fd7100 (LWP 7445) "broken-program-" 0x0000555555565511 in br
oken_program_2::multiply_numbers_by<f64> (
    nums=Vec<f64>(len: 6, cap: 6) = {...}, multiplier=3.141,
    ret=Arc<std::sync::mutex::Mutex<collections::vec::Vec<f64>>> = {...})
    at /home/vegai/fossil/rustbook/14/broken-program-2/src/main.rs:18
* 2    Thread 0x7ffff6bff700 (LWP 7449) "broken-program-" broken_program_2::multip
ly_and_store<f64> (
    nums=Arc<std::sync::mutex::Mutex<collections::vec::Vec<f64>>> = {...},
    num=1.5, multiplier=3.141)
    at /home/vegai/fossil/rustbook/14/broken-program-2/src/main.rs:8
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) continue
Continuing.
[New Thread 0x7ffff65ff700 (LWP 7452)]
[Thread 0x7ffff6bff700 (LWP 7449) exited]
[New Thread 0x7ffff63fe700 (LWP 7453)]
[Thread 0x7ffff65ff700 (LWP 7452) exited]
[New Thread 0x7ffff61fd700 (LWP 7454)]
[Thread 0x7ffff63fe700 (LWP 7453) exited]
[New Thread 0x7ffff5ffc700 (LWP 7455)]
[Thread 0x7ffff61fd700 (LWP 7454) exited]
[New Thread 0x7ffff5dfb700 (LWP 7456)]
[Thread 0x7ffff5ffc700 (LWP 7455) exited]
Multiplied numbers: Mutex { data: [4.7115, 6.885072000000001, 9.423, 15.387064839,
 17.2755, -15.705] }
[Thread 0x7ffff5dfb700 (LWP 7456) exited]
[Inferior 1 (process 7445) exited normally]
(gdb) 
```

Since we inserted a breakpoint in the thread, when the execution reaches there, GDB switches to that thread and halts the program. We can also command the threads explicitly with the thread apply command. Here we switch back to thread 1 (the main thread), and command the second thread. The threads are referred to by their short IDs: 1, 2, and so on:

```
(gdb) break 8
Breakpoint 1 at 0x111e7: file /home/vegai/fossil/rustbook/14/broken-program-2/src/
main.rs, line 8.
(gdb) run
Starting program: /home/vegai/fossil/rustbook/14/broken-program-2/target/debug/bro
ken-program-2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
[New Thread 0x7ffff6bff700 (LWP 7495)]
[New Thread 0x7ffff69fe700 (LWP 7496)]
[New Thread 0x7ffff63ff700 (LWP 7497)]
[Switching to Thread 0x7ffff6bff700 (LWP 7495)]

Thread 2 "broken-program-" hit Breakpoint 1, broken_program_2::multiply_and_store<
f64> (nums=Arc<std::sync::mutex::Mutex<collections::vec::Vec<f64>>> = {...},
    num=1.5, multiplier=3.141)
    at /home/vegai/fossil/rustbook/14/broken-program-2/src/main.rs:8
8           data.push(num * multiplier);
(gdb) thread 1
[Switching to thread 1 (Thread 0x7ffff7fd7100 (LWP 7491))]
#0  0x00007ffff72e1541 in clone () from /usr/lib/libc.so.6
(gdb) thread apply 2 print data

Thread 2 (Thread 0x7ffff6bff700 (LWP 7495)):
$1 = MutexGuard<collections::vec::Vec<f64>> = {__lock = 0x7ffff6c29010,
  __poison = Guard = {panicking = false}}
(gdb) thread apply 2 next

Thread 2 (Thread 0x7ffff6bff700 (LWP 7495)):
[New Thread 0x7ffff5dff700 (LWP 7501)]
[New Thread 0x7ffff57ff700 (LWP 7502)]
[New Thread 0x7ffff51ff700 (LWP 7503)]
9       }
(gdb) 
```

This should be enough to get you started with GDB. It is quite an old tool and widely used, so the Internet is full of useful tidbits. Most of them apply to C, but fortunately Rust is so close to C at runtime that nearly every piece of knowledge from there can be applied here.

Next, we'll take a look at a slightly more modern but similar tool, `lldb`.

LLDB - quick overview

LLDB comes from the LLVM project, which Rust itself is written on. Let's go through the same steps as before, but this time with LLDB. Rust comes with a wrapper for LLDB as well, unsurprisingly enough called `rust-lldb`. Here's how running a program through it looks:

```
vegai@carbon ~/rustbook/14/broken-program-1 » rust-lldb target/debug/broken-progra
m-1
(lldb) command source -s 0 '/tmp/rust-lldb-commands.SSvzxV'
Executing commands in '/tmp/rust-lldb-commands.SSvzxV'.
(lldb) command script import "/home/vegai/.rustup/toolchains/stable-x86_64-unknown
-linux-gnu/lib/rustlib/etc/lldb_rust_formatters.py"
(lldb) type summary add --no-value --python-function lldb_rust_formatters.print_va
l -x ".*" --category Rust
(lldb) type category enable Rust
(lldb) target create "target/debug/broken-program-1"
Current executable set to 'target/debug/broken-program-1' (x86_64).
(lldb) 
```

LLDB is similar to GDB in many ways, but arguably quite a bit more modern and neat in its UI. For example, here's the `help` command for run:

```
(lldb) help run
     Launch the executable in the debugger.

Syntax:

Command Options Usage:
  run [<run-args>]


'run' is an abbreviation for 'process launch -c /bin/sh --'
(lldb) 
```

Listing the file is close enough to the GDB, syntax, but setting the breakpoint has a slightly different syntax:

```
(lldb) list 1
   1    extern crate num;
   2    use num::Num;
   3
   4    fn multiply_numbers_by<N: Num + Copy>(nums: Vec<N>, multiplier: N) -> Vec<N
> {
   5        let mut ret = vec!();
   6        for num in nums {
   7            ret.push(num * multiplier);
   8        }
   9        ret
   10   }
(lldb) breakpoint set -l 7
Breakpoint 1: where = broken-program-1`broken_program_1::multiply_numbers_by<f64> +
 428 at main.rs:7, address = 0x000000000000928c
(lldb) 
```

The `print`, `continue`, and `next` commands work in pretty much the same way as in `gdb`:

```
(lldb) run
Process 11094 launched: '/home/vegai/fossil/rustbook/14/broken-program-1/target/deb
ug/broken-program-1' (x86_64)
Process 11094 stopped
* thread #1, name = 'broken-program-', stop reason = breakpoint 1.1
    frame #0: broken-program-1`broken_program_1::multiply_numbers_by<f64>(nums=vec!
[1.5, 2.1920000000000002, 3, 4.8987790000000002, 5.5, -5], multiplier=3.141) at mai
n.rs:7
   4    fn multiply_numbers_by<N: Num + Copy>(nums: Vec<N>, multiplier: N) -> Vec<N
> {
   5        let mut ret = vec!();
   6        for num in nums {
-> 7            ret.push(num * multiplier);
   8        }
   9        ret
   10   }
(lldb) continue
Process 11094 resuming
Process 11094 stopped
* thread #1, name = 'broken-program-', stop reason = breakpoint 1.1
    frame #0: broken-program-1`broken_program_1::multiply_numbers_by<f64>(nums=vec!
[1.5, 2.1920000000000002, 3, 4.8987790000000002, 5.5, -5], multiplier=3.141) at mai
n.rs:7
   4    fn multiply_numbers_by<N: Num + Copy>(nums: Vec<N>, multiplier: N) -> Vec<N
> {
   5        let mut ret = vec!();
   6        for num in nums {
-> 7            ret.push(num * multiplier);
   8        }
   9        ret
   10   }
(lldb) print ret
(collections::vec::Vec<f64>) $0 = vec![4.7115]
(lldb)
```

If we switch over to our multithreaded program, we can see that `lldb` has similar thread functionality:

```
(lldb) breakpoint set -l 7
Breakpoint 1: where = broken-program-1`broken_program_1::multiply_numbers_by<f64> +
 428 at main.rs:7, address = 0x000000000000928c
(lldb) run
Process 11179 launched: '/home/vegai/fossil/rustbook/14/broken-program-1/target/deb
ug/broken-program-1' (x86_64)
Process 11179 stopped
* thread #1, name = 'broken-program-', stop reason = breakpoint 1.1
    frame #0: broken-program-1`broken_program_1::multiply_numbers_by<f64>(nums=vec!
[1.5, 2.1920000000000002, 3, 4.8987790000000002, 5.5, -5], multiplier=3.141) at mai
n.rs:7
   4    fn multiply_numbers_by<N: Num + Copy>(nums: Vec<N>, multiplier: N) -> Vec<N
> {
   5        let mut ret = vec!();
   6        for num in nums {
-> 7            ret.push(num * multiplier);
   8        }
   9        ret
   10   }
(lldb) thread list
Process 11179 stopped
* thread #1: tid = 11179, 0x000055555555d28c broken-program-1`broken_program_1::mul
tiply_numbers_by<f64>(nums=vec![1.5, 2.1920000000000002, 3, 4.8987790000000002, 5.5
, -5], multiplier=3.141) at main.rs:7, name = 'broken-program-', stop reason = brea
kpoint 1.1
(lldb)
```

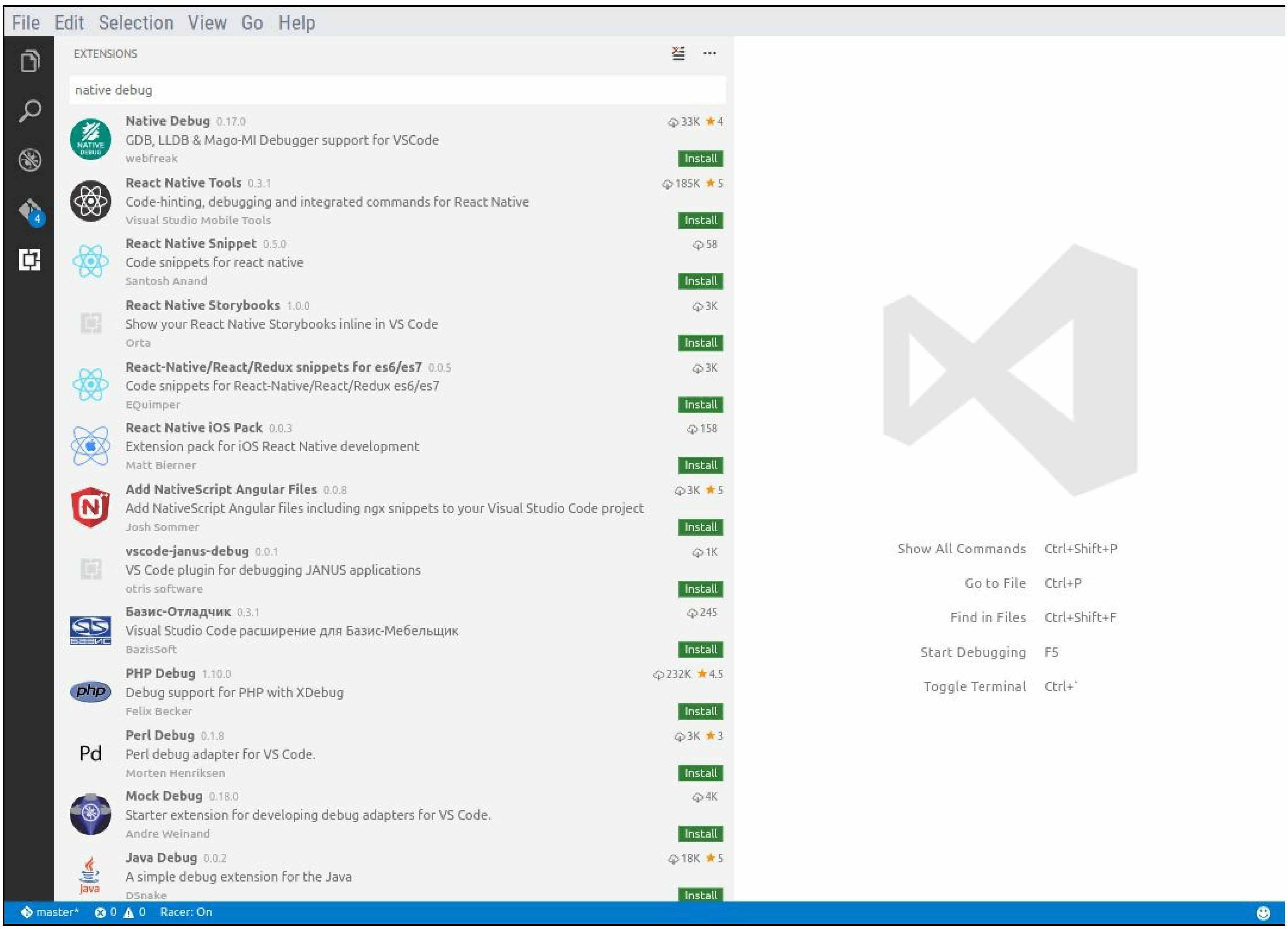
Take a look at the `lldb help` files and website for further information. Next, we'll take a look at how all this works more nicely inside a properly integrated text editor.

Editor integration with Visual Studio Code

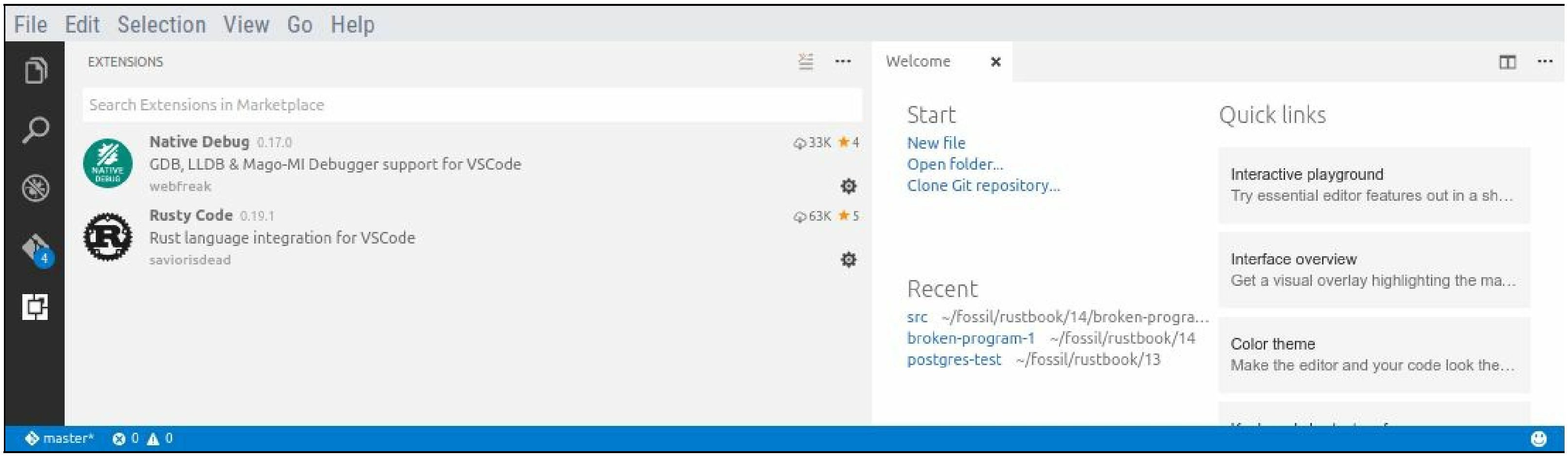
Using debuggers from the command line is a fine way to debug your software, and an important skill: you may easily wind up in a situation where your more advanced coding platform is not available. For instance, you may need to debug a program that is already running in production; attaching to a running process is possible with both `gdb` and `lldb`, but you may not be able to attach your editor.

Nevertheless, in a development environment you may enjoy a much smoother experience with a properly integrated editor. Let's see how this process would work with Visual Studio Code.

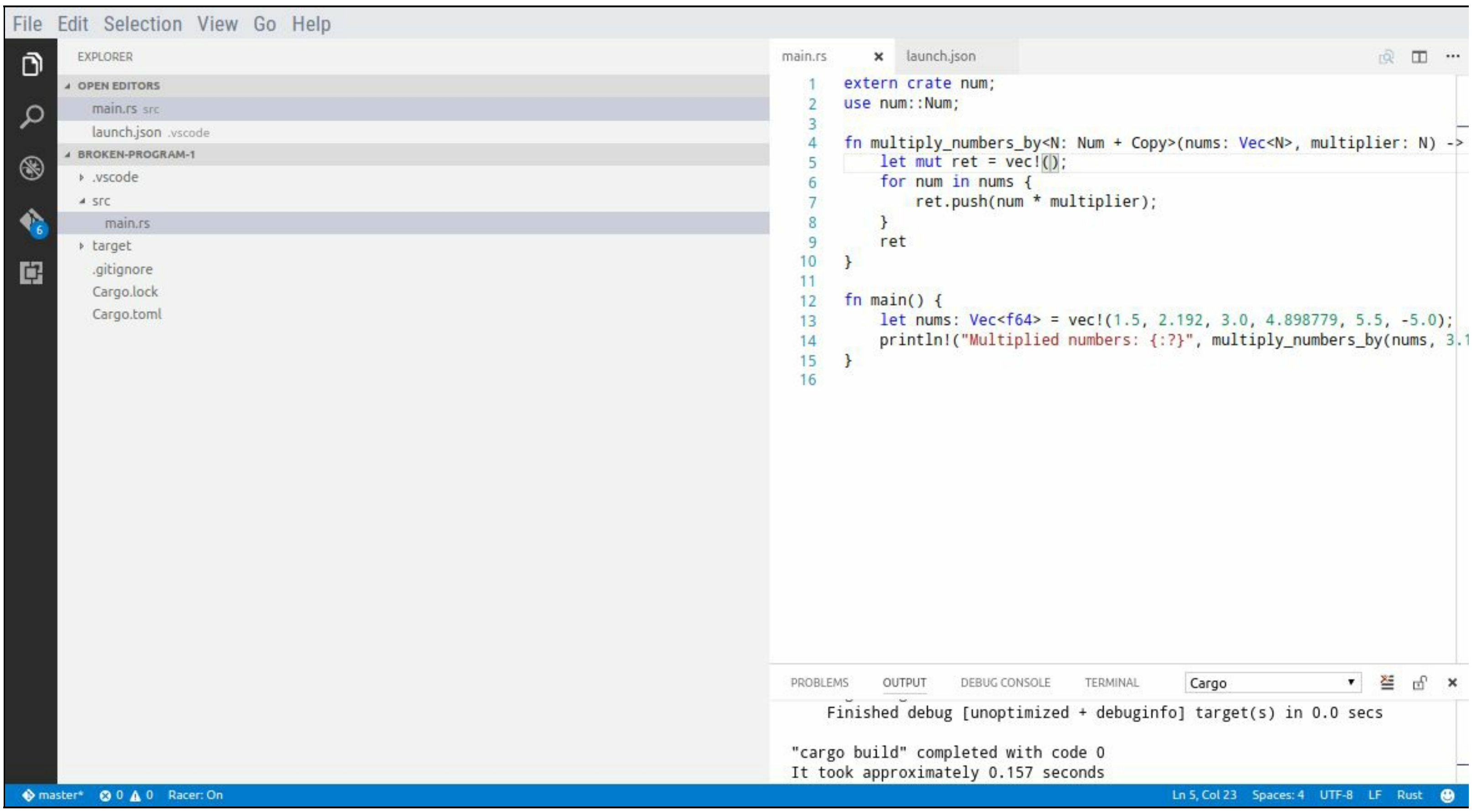
As a first step, you'll want the Rusty Code extension that we installed previously. In addition to that, we'll install a debugging extension called Native Debug, written by alias webfreak. Go to View | Extensions and search for it:



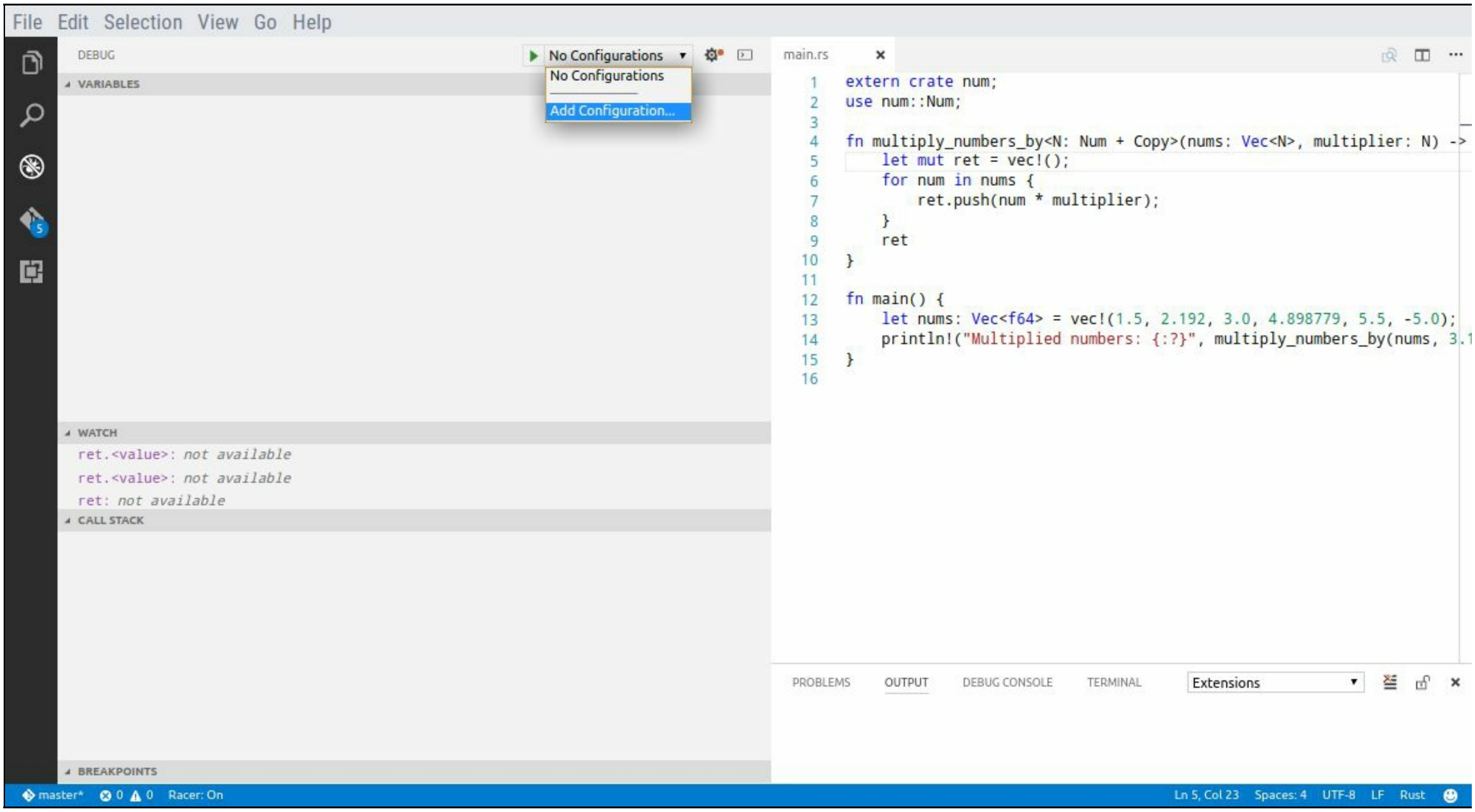
Click on Install and after installation is complete, reload to restart Visual Studio Code and enable the new extension. Your EXTENSIONS view should now look like this:



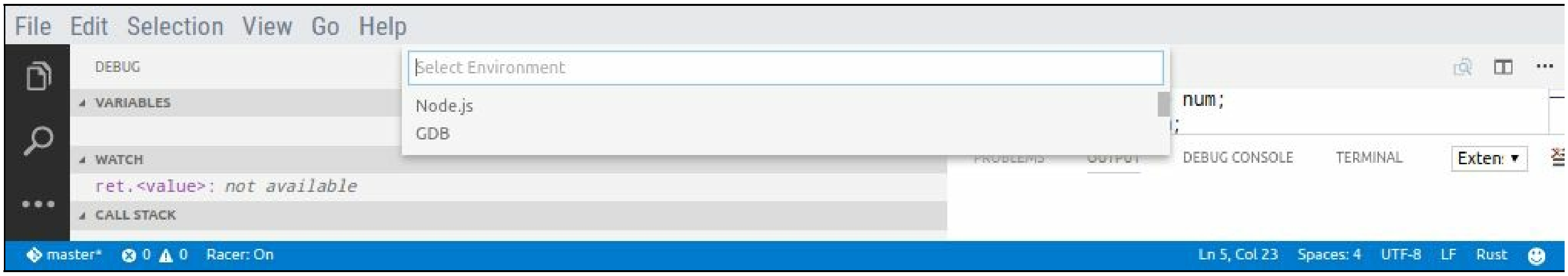
Next up, open our `broken-program-1` folder, and you should end up in this configuration of panels:



To enable the debugging panel, click on the icon showing a bug with a stop sign over it (third from the top on the left pane). Then, click on the dropbox labeled No Configurations, and from there, Add Configuration...:



You get a prompt that asks which premade debugging configuration you wish to base yours on:



Select GDB. Visual Studio Code will open up a file called `launch.json`, which will contain your project's debugging configuration.

We'll need to make two changes to this file:

1. Add a `gdbpath` variable pointing to `rust-gdb`. This will launch `gdb` via the `rust-gdb` wrapper, which gives us a pretty print of various Rust things.
2. Point target to our debugging binary.

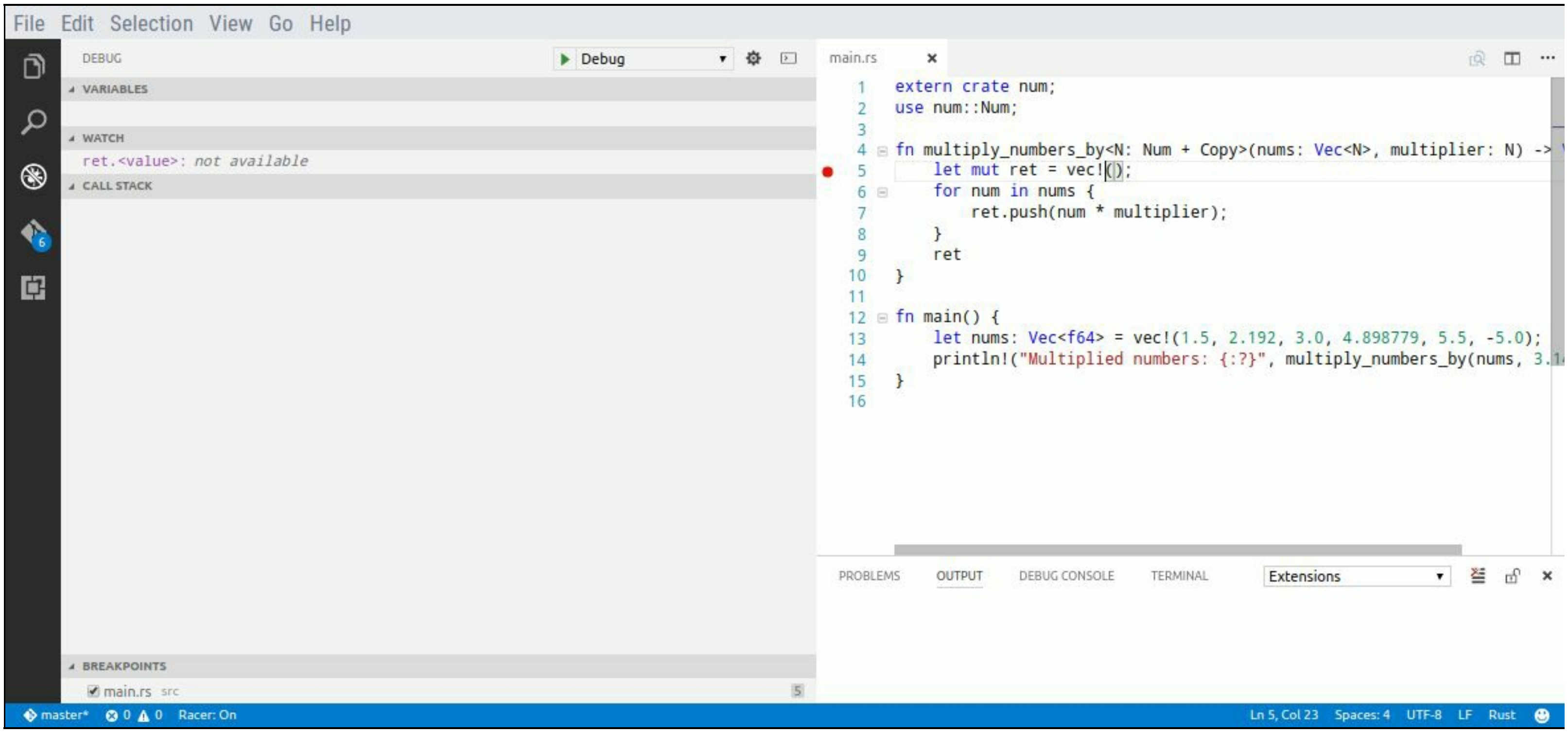
After these changes, the file should look something like this:

```
{
    "version": "0.2.0",
    "configurations": [

        {
            "name": "Debug",
            "type": "gdb",
            "gdbpath": "rust-gdb",
            "request": "launch",
```

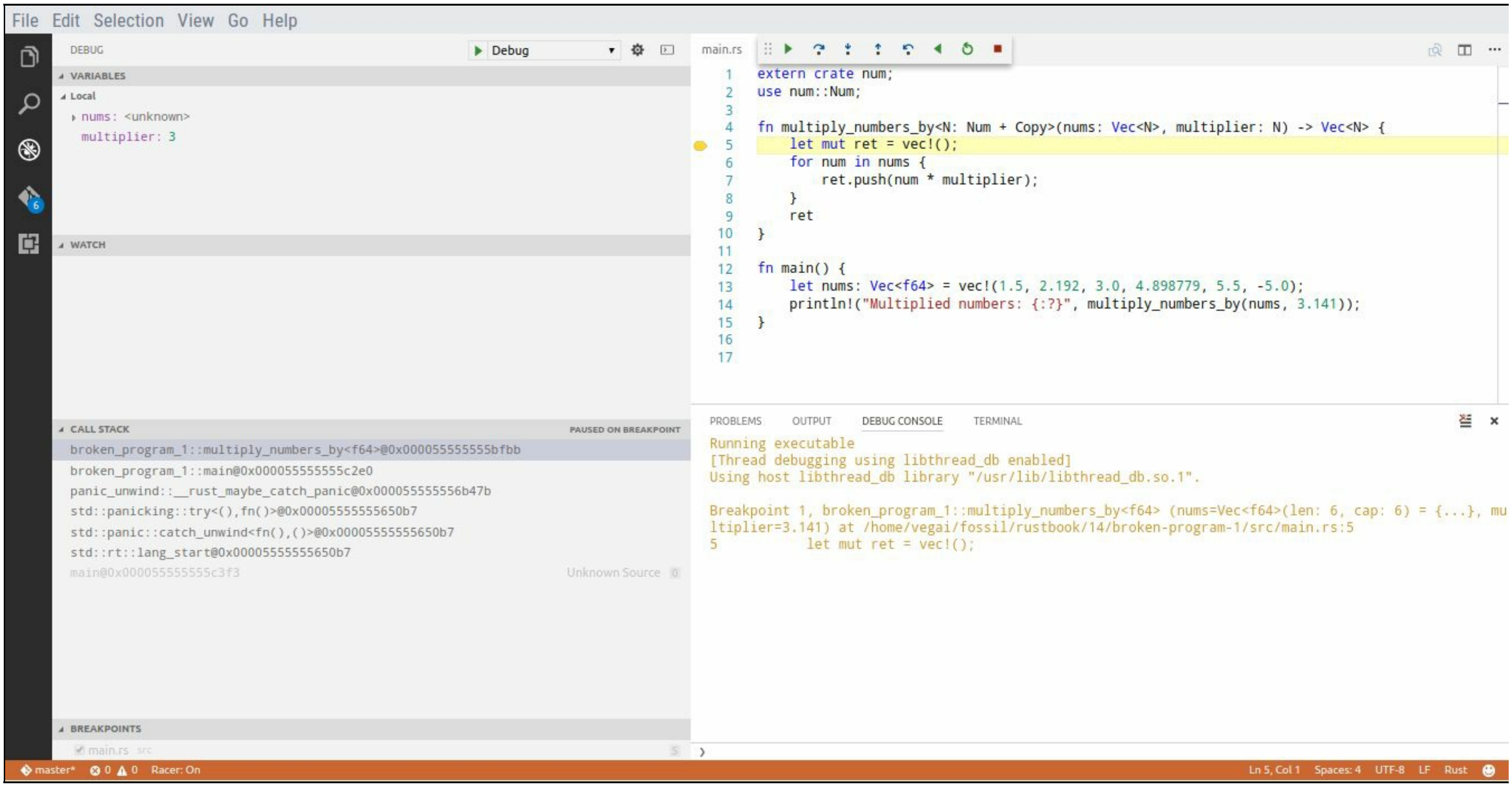
```
            "target": "./target/debug/broken-program-1",
            "cwd": "${workspaceRoot}"
        }
    ]
}
```

If you named your project something else besides `broken-program-1`, you should use that name here, of course.

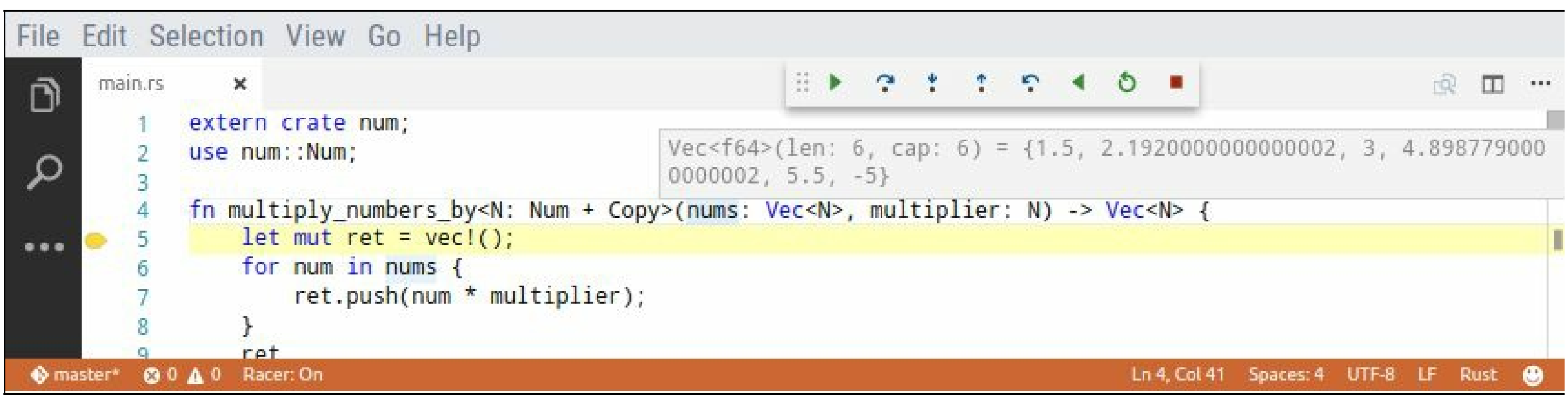
Now, the debugger is configured and we can insert a breakpoint to our program. This is done by clicking on the left-hand side of the line number on the line you wish to break at.



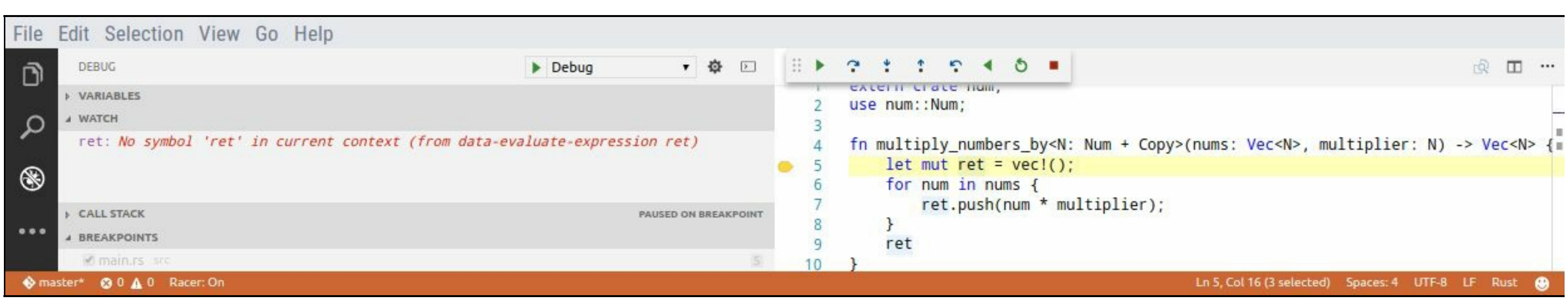
You should see a new red circle where you inserted the breakpoint. To remove it, click on it again. Now, we can run the program with debugging. Press the green Play button on top of the debugging panel. At the time of writing, Visual Code Studio integration is not quite perfect yet but usable. You may experience some weird output, however, as shown in the following screenshot:



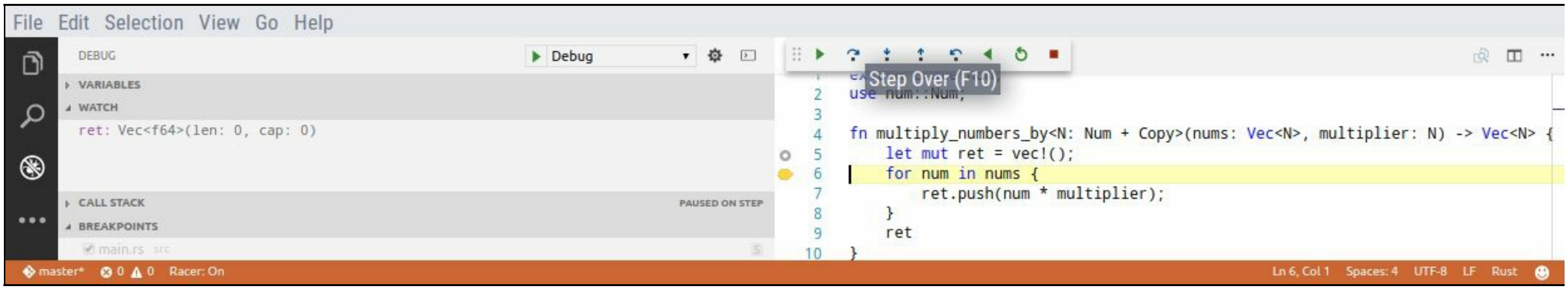
For instance, the multiplier in the local variables section is shown as an integer with the decimal part dropped off. From here, you can investigate the current values of all the variables by hovering over them in the code window:



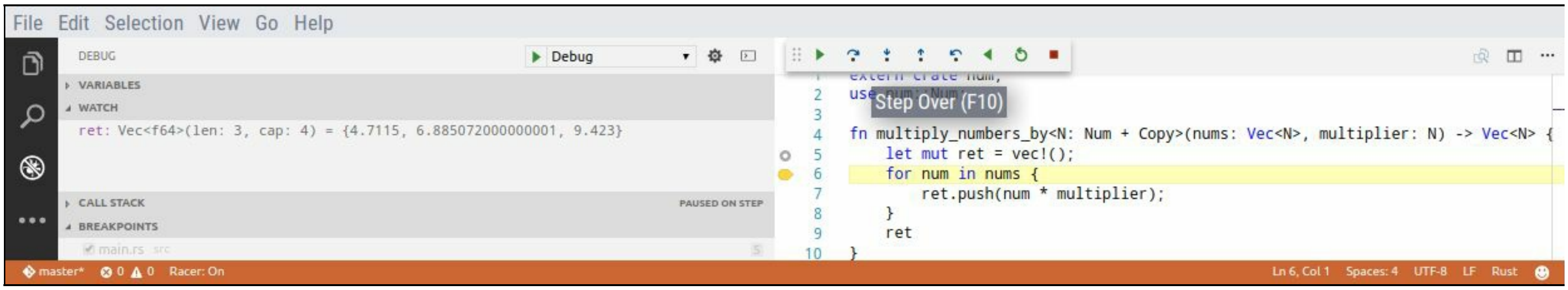
Also, you can set watches on variables, which makes them appear (and update as the program marches on) on the left debugging panel. To add a watch, paint over a variable name, click on your right mouse button (or equivalent), and select Debug: Add to Watch. For instance, after adding `nums` to the watch list, we get the following view:



We're still on the line that creates the `ret` variable, so the watch list reflects that fact. Step over a few iterations of the loop by pressing the Step Over button in the debug bar:



Witness how, on the watch list, the `ret` value gets populated:



That covers the basics of editor integration well enough to get you started. Be sure to check out the other features of this integration: it is just a regular `gdb` integration and since it's being heavily used with C and C++, you can gain leverage from the efforts of a large user base.

If you wish to use `lldb` instead of `gdb`, the process is pretty much identical to `gdb`. Just select `lldb` in the debug configuration phase instead of `gdb` and you should be set.

Exercises

1. Try the explore command of `gdb`. How does it differ from print?
2. Try the step command in `gdb` or `lldb`. How is it different? What happens between the function calls?
3. Both `gdb` and `lldb` support attaching to a running process. Make a program that doesn't quit, perhaps an erroneous never-ending loop, then attach to it via `rust-gdb` or `rust-lldb` and see what happens.
4. Could you fix a program running in production by attaching to it? What prerequisites need to be fulfilled beforehand?
5. Check out the disassembler command in either of the debuggers.
6. If Visual Code Studio isn't your preferred editor, try to find out if your favorite one has debugger integration for Rust.

Summary

In this chapter, we gained an overview of running a debugger against Rust code, both with GNU's `gdb` and the LLVM project's `lldb`. We showed how to integrate `gdb` into Visual Studio Code, giving you a nicer and easier view into debugging.

We're starting to reach the end of our journey; the next chapter will be about final conclusion, parting words, and solutions to exercises. If you've gotten this far, you should pat yourself on the back. Good work!

Solutions and Final Words

You've come a long way, well done! You've learned a new language that probably had quite a few paradigms you hadn't met before. In this chapter, we'll wrap up the book with a short recap of every chapter, along with sample solutions to all exercises.

Chapter 1 - Getting Your Feet Wet

Rust is a language stewarded by Mozilla Research. It focuses on zero-cost abstractions and compile-time safety. Zero-cost abstractions refer to having modern high-level programming techniques available without creating any runtime overhead. Access to resources is secured by the compiler by a system of lifetimes and borrowing, which makes memory access safe without requiring a garbage collector.

The official Rust implementation comes in three forms: nightly, beta, and stable. The nightly branch is built automatically every night from the latest source code. Every six weeks, a release happens: beta version becomes the new stable, and a new beta version is branched off the nightly version. This does not imply that all the features from nightly go to stable every six weeks, rather, only those that are deemed to be stable. People should generally reach for the stable version but nightly has several nice but unstable features, so it is sometimes used.

rustup is the official distribution system for the Rust compiler and the Cargo package manager. It supports fetching and locally installing various Rust components and toolchains, and also enables flexible switching between the nightly, beta, and stable versions.

Rust's main abstraction mechanism is functions, defined with the `fn` keyword. Variables default to non-mutable, and variable bindings are defined with the `let` keyword. Mutability can be explicitly requested with `let mut`. Conditional branching is done with `if`, which is quite similar to other languages. Low-level loops can be written using `loop` and `while`, and higher-level looping via iterators is done with `for`. Compound data with one or more fields is formed with `struct`, while compound data with single variants is defined with `enum`.

Primitive types in Rust include Booleans (`bool`), various integers (`usize`, `isize`, `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, and `i64`), floats (`f32` and `f64`), characters and string slices (`char` and `str`), fixed size arrays (`[T; N]`), slices (`&[T]`), tuples (`(T1, T2, ...)`), and functions (`fn(T1, T2, ...) → R`).

The standard library has two data structures for dynamic data: Vectors and HashMaps. Vectors offer dynamically sized arrays, while HashMaps are key-to-value mappings.

Exercise - fix the word counter

Here's a working version of the word counter. We have added the missing `use` statement and missing types, missing pointer syntax, and a missing mutability flag:

```
use std::env;
use std::fs::File;
use std::io::prelude::BufRead;
use std::io::BufReader;
use std::collections::HashMap;

#[derive(Debug)]
struct WordStore (HashMap<String, u64>);

impl WordStore {
    fn new() -> WordStore {
        WordStore (HashMap::new())
    }

    fn increment(&mut self, word: &str) {
        let key = word.to_string();
        let count = self.0.entry(key).or_insert(0);
        *count += 1;
    }

    fn display(&self) {
        for (key, value) in self.0.iter() {
            println!("{}: {}", key, value);
        }
    }
}

fn main() {
    let arguments: Vec<String> = env::args().collect();
    println!("args 1 {}", arguments[1]);
    let filename = arguments[1].clone();

    let file = File::open(filename).expect("Could not open file");
    let reader = BufReader::new(file);

    let mut word_store = WordStore::new();

    for line in reader.lines() {
        let line = line.expect("Could not read line");
        let words = line.split(" ");
        for word in words {
            if word == "" {
                continue
            } else {
                word_store.increment(word);
            }
        }
    }

    word_store.display();
}
```

Chapter 2 - Using Cargo to Build Your First Program

The Cargo tool is the official package manager and build tool. It allows configuring package dependencies, building, running, and testing Rust software. Cargo is extensible, so additional functionality can be added via third-party packages.

Projects are configured in a single file, `Cargo.toml`. TOML is a configuration language that is essentially an INI, a file format extended with tree forms, lists, and dictionaries.

The de facto Rust code formatter tool is rustfmt. Clippy is a tool that can make additional style and pedantic checks on your codebase. It requires the nightly compiler as it works via compiler plugins. Racer does lookups into library code, giving code completion and tooltips. It requires Rust's source code to be available, which can be installed locally with rustup. All of these programs are written in Rust, and can be installed with Cargo.

These programs form the backbone for editor integration. Visual Code Studio, Emacs, Vim, and Atom all have plugins that provide good Rust support, and the list of editors with good Rust support is growing. https://areweideyet.com should contain up-to-date information about the current situation.

Exercise - starting our project

1. Initialize a Cargo binary project, call it whatever you want (but I will name the game *fanstr-buigam*, short for fantasy strategy building game).

    Solution:

    ```
    cargo init fanstrbuigam -bin
    ```

2. Check `cargo.toml` and fill in your name, e-mail, and description of the project.

    Solution: `Cargo.toml` should have name, e-mail, description in the `[package]` section.

3. Try out the Cargo build, test, and run commands.

    Solution: `cargo build`/`cargo test`/`cargo run`.

Chapter 3 - Unit Testing and Benchmarking

Unit tests are a neat way to increase and maintain code quality. Rust supports basic unit testing in its core package. Test functions are annotated with `#[test]`. Two macros, `assert!` and `assert_eq!`, can be used to declare the expected function results. The `#[should_panic]` annotation can be used to define that a test should fail with a panic. Unit tests are placed in the same file as the code they test. The test code can be separated from the actual code by putting it in a separate module annotated with `#[cfg_test]`.

Integration tests go into a separate tests/directory in a Rust project. These are meant for testing larger portions of code. Unlike unit tests, integration tests run as if they were consumers of library code of the project. They are black box tests.

Documentation tests are a third form of tests. They are meant for making runnable test code in module documentation. These tests are marked in markdown style by enclosing the test code in ```. This can be used in module-level (`//! comments`) or function-level (`/// comments`).

The fourth form is benchmark tests. They work somewhat like unit tests. Benchmark functions are annotated with `#[bench]` and take a `Bencher` object, which comes from the test crate. Benchmarks are not a stable feature, so they require nightly Rust.

Exercise - fixing the tests

1. Fix the preceding compilation problem.

   Solution: Fixed by removing the one semicolon on line 63.

2. The code has a few other subtle problems, revealed by the tests, fix those too.

   Solution: The first one is that the Grid implementation creates a world with only Stone as its ground. Changing it to Soil fixes the test.

   The second one is in the `generate_empty` method. The loop should start from `0`, not `1`.

3. After fixing the tests, the compiler warns about dead code. Find out how to suppress those warnings.

   Solution: The `#[allow(dead_code)]` attribute for each enum would fix this. In a real software project, there's usually no reason to keep dead code around, however.

   Here's the code that contains all the three fixes:

```
#[allow(dead_code)]
#[derive(PartialEq, Debug)]
enum TerrainGround {
    Soil,
    Stone
}

#[allow(dead_code)]
#[derive(PartialEq, Debug)]
enum TerrainBlock {
    Tree,
    Soil,
    Stone
}

#[allow(dead_code)]
#[derive(PartialEq, Debug)]
enum Being {
    Orc,
    Human
}

struct Square {
    ground: TerrainGround,
    block: Option<TerrainBlock>,
    beings: Option<Being>
}

struct Grid {
    size: (usize, usize),
    squares: Vec<Square>
}

impl Grid {
    fn generate_empty(size_x: usize, size_y: usize) -> Grid {
        let number_of_squares = size_x * size_y;
        let mut squares: Vec<Square> = Vec::with_capacity(number_of_squares);
```

```
            for _ in 0..number_of_squares {
                squares.push(Square{ground: TerrainGround::Soil, block: None, beings: None});
            }

            Grid {
                size: (size_x, size_y),
                squares: squares
            }
        }
    }

    #[cfg(test)]
    mod tests {

        #[test]
        fn test_empty_grid() {
            let grid = ::Grid::generate_empty(5, 13);
            assert_eq!(grid.size, (5, 13));
            let mut number_of_squares = 0;

            for square in &grid.squares {
              assert_eq!(square.ground, ::TerrainGround::Soil);
              assert_eq!(square.block, None);
              assert_eq!(square.beings, None);
              number_of_squares += 1;
            }

            assert_eq!(grid.squares.len(), 5*13);
            assert_eq!(number_of_squares, 5*13);

        }
    }
```

Chapter 4 - Types

Rust has a primitive string slice type, `&str`, and a heap-allocated and growable `String` type. These types guarantee Unicode safety. Bytestrings need to be used for I/O, and the type for that is simply `[u8]`.

Arrays are fixed in size in Rust. The type for them is `[T; n]`, where `T` is the type of things contained and `n` is the size of the array. Slices are pointers to any existing sequence and the type for that is `&[T]`.

Traits are used to declare functionality. For example, the `Into` trait defines conversions between types. It can be implemented for arbitrary types.

Rust has generic types. The syntax for generic types is of the form `<T>`. This type can be declared in enums and structs, and then referred to in the enum or struct body. If the traits or implementation blocks define generic types, then that generic type has to be declared along with the trait or implementation before usage in the body. Generic types can be narrowed down by the usage of trait bounds.

Constants offer a safe form of global values. They can be declared at the top level with the `const` keyword, and their types must be defined explicitly every time. Constants essentially are just replaced with the contents wherever they are used. Statics are somewhat like constants, but are more like real values: they can also be mutable. Mutable static variables can only be used inside unsafe blocks.

Exercise - various throughout the chapter

1. Create a few string slices and Strings, and print them. Use both `push` and `push_str` to populate a String with data.

   Solution:

```
    let s1 = "string slice";
    let s2 = "another string slice";
    let str1 = "heap string".to_string();
    let str2 = String::from("another heap string");
    let mut str3 = String::new();

    str3.push_str("yet");
    str3.push(' ');
    str3.push_str("another");
    str3.push(' ');
    str3.push_str("heap string");

    println!("s1 {} s2 {} str1 {} str2 {} str3 {}", s1, s2, str1, str2, str3);
```

2. Write a function that takes a string slice and prints it. Pass it a few static string slices, and a few Strings.

   Solution:

```
  fn print_string_slice(s: &str) {
    println!("Printing {}", s);
  }

  fn main() {
    let s = "string slice";
    let str = String::from("another heap string");

    print_string_slice(&s);
    print_string_slice(&str);
  }
```

3. Define bytestrings with both UTF-8 strings and non-UTF-8 strings. Try to make Strings out of them and see what happens.

   Solution:

```
    let ok_bytestring = vec!(82, 85, 83, 84);
    let nok_bytestring = vec!(255, 254, 253);

    let str_from_ok = String::from_utf8(ok_bytestring);
    let str_from_nok = String::from_utf8(nok_bytestring);

    println!("{:?}", str_from_ok); // Ok("RUST")
    println!("{:?}", str_from_nok); // Err(FromUtf8Error...)
```

4. Make a String that contains the phrase *You are a Rust coder, Harry*. Split the string into words and print the second word. See https://doc.rust-lang.org/std/string/struct.String.html: you'll need to use the `collect()` method.

Solution: Note the lifetime problems due to temporary values if you try to do both the String creation and splitting on the same line:

```
let str = "You are a Rust coder, Harry".to_string();
let splitted_str: Vec<&str> = str.split(" ").collect();

println!("{}", splitted_str[1]);
```

5. Make a 10-element fixed array of numbers.

   Solution:

```
let numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
```

6. Take a slice that contains all elements of the previous array except the first and the last.

   Solution:

```
&numbers[1..9];
```

7. Use `for x in xs` (shown briefly in Chapter 1, *Getting Your Feet Wet*) to sum all the numbers in both the array and the slice. Print the numbers.

   Solution:

```
let numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
let nums_without_first_and_last = &numbers[1..9];

let mut sum1 = 0;
let mut sum2 = 0;

for n in numbers.iter() {
    sum1 +=n;
}

for n in nums_without_first_and_last.iter() {
    sum2 +=n;
}

println!("sum1 {} sum2 {}", sum1, sum2);
```

8. Take a look at the collection types, documented in https://doc.rust-lang.org/stable/std/collections/.

   Solution: Visit https://doc.rust-lang.org/stable/std/collections/.

9. Use HashMap for any key-value type pairs you choose.

   Solution:

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

fn main() {
    let mut hm = HashMap::new();
    hm.insert(14, "fourteen");
    hm.insert(25, "twentyfive");
}
```

10. Use BtreeMap for any key-value type pairs.

Solution: Nearly identical API to HashMaps. So like above.

11. Take a look at the `new` methods of various collections. Notice the difference in the generic type. Think about what they might mean.

Solution: HashMaps and BtreeMaps have generic types, `K` and `V`, which refer to keys and values. HashMap additionally requires a random number generator, so it has a `RandomState` generic. A user of the struct does not typically have to care about the RNG.

12. Make your own type, without generics. Perhaps just strip off the generic type from our `Money<T>`. Implement some or all of the operators for it: https://doc.rust-lang.org/std/ops/index.html.
13. Same as previous but make your type have generics (or use the `Money<T>` type in from this section).

Solution: The non-generic case is trivial. The generic case may be a bit tricky, since we need to tie the trait bound to the output, and the syntax is not obvious:

```
use std::ops::Add;

struct Money<T> {
     amount: T,
    currency: String
}

impl<T> Add for Money<T> where T: Add<Output=T> {
    type Output = Money<T>;

    fn add(self, other: Money<T>) -> Money<T> {
        let added = other.amount + self.amount;
        Money { amount: added, currency: self.currency }
    }
}
```

14. Implement a `Point` struct that describes a point in 2D space.
15. Implement a `Square` struct that uses the `Point` struct defined above for coordinate.
16. Implement a `Rectangle` struct likewise.

Solution for 14, 15, 16:

```
struct Point {
    x: i64,
    y: i64,
}

struct Square {
    point: Point,
    size: u64
}

struct Rectangle {
    point: Point,
    x_size: u64,
    y_size: u64
}
```

17. Make a trait, `Volume`, that has a method for getting the size of something. Implement `Volume` for `Square`

and `Rectangle` objects.

Solution:

```
trait Volume {
    fn get_volume(&self) -> u64;
}

impl Volume for Square {
    fn get_volume(&self) -> u64 {
        self.size * self.size
    }
}

impl Volume for Rectangle {
    fn get_volume(&self) -> u64 {
        self.x_size * self.y_size
    }
}
```

Chapter 5 - Error Handling

Rust's error handling primarily rests on two enums: `Option` and `Result`. Nearly all library code that could fail returns values wrapped inside one of those two or a variation of them.

`Option` replaces the `Null` type of other languages; it models things that are either something (`Some(N)`) or may be nothing (`None`). For instance, a query against a database might return an `Option`, since the result set of the query might be empty. Neither of these actions need to be considered an error.

The `Result` type models operations that may succeed (`Ok(T)`) or may fail (`Err(E)`). In case of an error, some form of reporting of the error is returned wrapped inside the `Err`.

To make operating with these types a bit easier, there are helpful unwrapping methods defined for them. In critical code, they should be used to make code easier to read, not just to ignore errors.

Rust has an exception-like mechanism called panic. It should primarily be used for aborting execution when something non-recoverable happens in the program.

Exercise solutions

1. Implement the error case where the `Being` tries to fall from the edge of the `Grid`.
2. Implement the error case where the `Being` tries to move into a `Square` where there is already a `Being`.
3. Implement the error case where the `Being` tries to move to a `Terrain` that is `Stone`.
4. Implement the happy case where no errors happen and the `Being` successfully moves to the new `Square`.

Solution: You may have noticed that 1-3 can be implemented in a straight-forward manner, but at 4, when we need to mutate the squares, we crash into some obstacles. We solved this by cloning the whole squares of the grid, which is obviously a tad inefficient, but works.

Here are the filled `move_being_in_coord` method and related unit tests:

```
    fn move_being_in_coord(&mut self, coord: (usize, usize), dir: Direction) -> Result<(usize, usize), I
        let copy_of_squares = self.squares.clone();
        let square = copy_of_squares.get(coord.0 * self.size.0 + coord.1).expect("Index out of bounds t

        if square.being == None {
            return Err(MovementError::NoBeingInSquare);

        }

        let new_coord = match dir {
            Direction::North => (coord.0 - 1, coord.1),
            Direction::East  => (coord.0, coord.1 + 1),
            Direction::South => (coord.0 + 1, coord.1),
            Direction::West  => (coord.0, coord.1 - 1),
        };

        if new_coord.0 >= self.size.0 || new_coord.1 >= self.size.1 {
            return Err(MovementError::FellOffTheGrid);
        }

        let new_square = copy_of_squares.get(new_coord.0 * self.size.0 + new_coord.1).unwrap();
        if new_square.being != None {
            return Err(MovementError::AnotherBeingInSquare);
        }
        if new_square.ground == TerrainGround::Stone {
            return Err(MovementError::MovedToBadTerrain);
        }

        // Everything checks out, let's mutate!
        // Easiest way is to just replace an existing square with a completely new one
        self.squares[new_coord.0 * self.size.0 + new_coord.1] =
            Square { ground: new_square.ground.clone(),
                     block: new_square.block.clone(),
                     being: square.being.clone()
            };
        self.squares[coord.0 * self.size.0 + coord.1] =
            Square { ground: square.ground.clone(),
                     block: square.block.clone(),
                     being: None
            };

        Ok(new_coord)
    }
...
    #[test]
    fn test_move_being_off_the_grid() {
        let mut grid = ::Grid::generate_empty(3, 3);
        let human = ::Being::Human;
```

```
        grid.squares[3*3-1].being = Some(human);

        assert_eq!(grid.move_being_in_coord((2,2), ::Direction::East),
                   Err(::MovementError::FellOffTheGrid));
    }

    #[test]
    fn test_move_being_on_another_being() {
        let mut grid = ::Grid::generate_empty(3, 3);
        let human = ::Being::Human;
        let orc = ::Being::Orc;

        grid.squares[0].being = Some(human);
        grid.squares[1].being = Some(orc);

        assert_eq!(grid.move_being_in_coord((0,0), ::Direction::East),
                    Err(::MovementError::AnotherBeingInSquare));
    }

    #[test]
    fn test_move_being_on_stone() {
        let mut grid = ::Grid::generate_empty(3, 3);
        let human = ::Being::Human;
        let stone = ::TerrainGround::Stone;

        grid.squares[0].being = Some(human);
        grid.squares[1].ground = stone;

        assert_eq!(grid.move_being_in_coord((0,0), ::Direction::East),
                    Err(::MovementError::MovedToBadTerrain));
    }

    #[test]
    fn test_move_successfully() {
        let mut grid = ::Grid::generate_empty(3, 3);
        let human = ::Being::Human;

        grid.squares[0].being = Some(human);
        assert_eq!(grid.move_being_in_coord((0,0), ::Direction::South),
                    Ok((1,0)));
        assert_eq!(grid.squares[0].being, None);
        assert!(grid.squares[3].being != None);
    }
```

5. Make `MovementError` implement the `Error` trait.

   Solution: The `Error` trait is declared as follows:

```
pub trait Error: Debug + Display
```

   So we need to supply a `Display` implementation for `MovementError` to be able to comply with the `Error` trait.

   The `cause` method is for reporting any lower-level cause of the error, but since `MovementError` has no such things, it can just return a `None`. Here's the code:

```
use std::error::Error;
use std::fmt;

impl fmt::Display for MovementError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "MovementError!")
    }
}

impl Error for MovementError {
```

```
    fn description(&self) -> &str {
        match self {
            NoBeingInSquare => "No being in square",
            FellOffTheGrid => "Tried to move off the grid",
            AnotherBeingInSquare => "Tried to move on another being",
            MovedToBadTerrain => "Tried to move to inaccessible terrain"
        }
    }

    fn cause(&self) -> Option<&Error> {
        None
    }
}
```

Chapter 6 - Memory, Lifetimes, and Borrowing

Rust's compiler is based on the compiler framework LLVM. The compiler emits LLVM IR codes, which LLVM translates into an executable binary. All that typically happens transparently, but it's possible to read into the intermediate IR codes and also Assembler code if a programmer wants to dive deep.

Local variables are stored in the stack. These values have strictly limited lifetimes: values in the stack do not outlive their blocks. Values that are expected to live longer than their blocks are stored in the heap.

Rust is a safe language due to the lack of null pointers and due to the ownership, borrowing, and lifetimes memory handling system.

Regarding ownership, Rust values have either move or copy semantics. Move is the default for new data types. As an example, when calling a function and passing parameters with move semantics to it, the ownership of those values goes to the function and therefore can no longer be used after the function call. Borrowing (using the `&` operator) helps circumvent this: that way, the ownership gets returned back.

Most Rust primitive types have copy semantics. This means that whenever a value is passed to another block, it gets copied. Copy semantics can be added to a new type by implementing the `Copy` trait. Opt-in copy semantics can be added by implementing the `Clone` trait. You can explicitly call the clone method on such values to get a new copy.

Collector types `Box<T>`, `Cell<T>`, `RefCell<T>`, `Rc<T>`, and `Arc<T>` can be used to make more fine-grained choices about the memory allocation and mutability of values.

Exercises

1. Use `rustc -O` to generate optimized LLVM IR code. What happened to your code?

   Solution: With `rustc -O`, quite a lot of boilerplate code vanishes. But also potentially useful things get lost in optimization, such as integer overflow checks.

2. Make a new String value in main and see what kind of IR code gets generated.

   Solution: There are heap allocations for String.

3. Add a `println!` macro to your code. How did it affect the IR code?

   Solution: The `println!` macro adds quite a lot of IR code due to formatting and console output.

4. Take your second favorite programming language and try to figure out if ownership of variables plays any part. Perhaps behind the curtain, hidden?

   Solution: C allocates everything in the stack by default, and relies on library calls (such as `malloc`) for heap allocations. Higher-level languages attempt to optimize for stack allocations when they detect that values are not used outside of blocks. But they also tend to default to heap allocations when they cannot, or when the implementors did not care about performance that much.

5. Each process has a limited stack size, enforced by the operating system. The size varies over different systems, in Linux it's usually about 8 MB. Imagine a few ways in which you could cause that limit to break.

   Solution: `let x = [0; 9999999]` should do it.

6. Take your second favorite programming language and try to figure out if ownership of variables plays any part. Perhaps behind the curtain, hidden?

   Solution: A good C programmer takes ownership into account, but it's not annotated in code and therefore cannot be easily checked. Higher-level languages try to hide this issue somewhat. For instance, in Python, most values are immutable, but things like lists, dictionaries, and objects are not. This is similar, but not identical, to the copy/move semantics separation of Rust.

7. Does the compiler/interpreter help the coder in that language with ownership issues or is it all in the hands of the programmer?

   Solution: Rust is a bit of a pioneer in the area of statically checked memory issues. High-level languages typically try to solve these issues under the hood so that the programmer doesn't have to.

8. Try to reason out the sizes of each of the preceding types.
9. Compile and run the code. Go through the differences between your guesses and reality.

   Solution: Run the code to see the sizes.

Chapter 7 - Concurrency

Concurrency means having more than one thing to do. Parallelism means those things being done at the same time. Concurrency comes with a plethora of potential problems, which are difficult to detect and fix. Rust protects against data races by the resource system, and helps with other concurrency problems by good concurrency primitives.

Closures are blocks that close over their surrounding environment. That means that variables are declared outside the block but used inside the block are captured inside the closure. Closure syntax allows differentiating between borrow and move semantics in these captures. Borrowing is the default, and its meaning is identical to before: variables are borrowed by the closure and then returned. Move semantics are also like before, with the added detail that if types have `Copy` semantics, they are copied instead.

Threads are launched with the standard spawn method. It takes a function or a closure, all of whose parameters have to implement the `Send` trait. Send has no methods, it's just a marker trait used to mark types that are safe to move between threads. Almost everything implements it, except types that are not thread-safe, such as `Rc<T>`.

Standard library has both asynchronous and synchronous channels for safely transferring data between threads. Synchronous channels have a configurable buffer, and if the buffer fills up, sending on that channel blocks. Asynchronous channels have an infinite buffer and thus never block. There are also mutexes for locking access to shared variables. Combining mutexes with the `Arc<T>` type gives you atomically reference counted types that can be safely and easily shared between different threads.

Exercises

1. Remove `mut` from the closure declaration line. Why does that make compilation fail?

   Solution: Because the closure becomes mutable when values it captures are mutated inside it, even when (like in our example) all of those values have copy semantics.

2. Remove move from the closure declaration line. What's the effect and why?

   Solution: The closure turns into having the default borrow semantics, which means that it receives a borrowing of the outer_scope_x variable, instead of a copy of it.

3. It looks like we wouldn't need the block starting from line 4 and ending on line 11. Try to remove that and see if that's true.

   Solution: We actually don't! The last `println!` wants access to `outer_scope_x`, which is being borrowed mutably by the closure. But, since we requested move semantics and `outer_scope_x` has a `Copy` trait, it is copied inside the closure, so the outside `outer_scope_x` is still accessible.

4. Remove both the braces mentioned in 3 and move. What's the effect and why?

   Solution: OK, this time we do need the braces. The last `println!` wants access to `outer_scope_x`, but this is actually borrowed mutably. So it won't be available because of this reason. By wrapping all that code inside a block, the closure and all its borrows get freed and thus `outer_scope_x` is again available.

5. Change the closure so that `outside_string` gets returned from it.
6. Grab the `outside_string` in the `main` thread. You get it from the `join` method.

   Solution for 5 and 6: The thread's `join` method gets the returned variable inside an `Option`.

```
use std::thread;

fn main() {
    let outside_string = String::from("outside");

    let thread = thread::spawn(move || {
        println!("Inside thread with string '{}'", outside_string);
        outside_string
    });

    let s = thread.join();
    println!("Result of the thread: {:?}", s);
}
```

7. After the changes above, what happens and why when you omit the move annotation from the closure?

   Solution: Nothing really happens in this case if we change the closure to borrow mode because we are not using the `outside_string` after the thread. If we are, however, all hell would break loose. Strings don't implement the `Copy` trait, so we'd need to clone it manually and use the cloned string inside the closure. Whether the semantics of the closure is move or borrow does not really matter at that point.

8. Change the synchronous buffer size to `0` and see what happens. Figure a way to make the code work with a zero buffer.

   Solution: Since a synchronous channel of size `0` blocks at the very first send, we'll need to have a receiving thread up and running before the first send, otherwise the code will block. A receiving thread might look like this:

```
let receiver = thread::spawn(move || {
    loop {
        println!("Received {} via the channel", rx.recv().unwrap());
    }
});
```

   Remember to join the thread at the end, otherwise the main function exits before the threads do.

9. Add a third receive call to the asynchronous code. Witness the block.

   Solution: Yep, that's what happens.

10. Take a look at the state of select. At the time of writing, there's a macro, `std::select!`, which is a rather concise way of defining select loops. Give it a try.

    Solution: The `select!` macro is used in a similar fashion as `match` blocks. It takes several `match` expressions that each receive on different channels. If none of the channels have things to receive, the macro blocks, otherwise it fires the block that first matches. Example:

```
select! {
        data1 = rx1.recv() => println!("thread 1 received data {}", data1),
        data2 = rx2.recv() => println!("thread 2 received data {}", data2),
    }
}
```

    `select!` was still a nightly feature at the time of writing this, so until that changes, it should not be relied on in production code.

11. Remove the inner block from the preceding code, compile and run. What happens and why?

    Solution: If we remove the block, the first lock never gets freed, so the program will hang at the second lock.

12. Try giving the mutex to multiple threads and using it from each. Why doesn't this work?

    Solution: Because you cannot borrow the same mutex from different threads due to the borrowing restrictions. And you cannot just clone it because then you'll have a different mutex between the threads. You'll need something like `Arc<T>` to share a mutex.

13. Fiddle with the move declarations again. Consider the error messages given by the compiler.

    Solution: If you remove any of the closure moves, the captured values become borrows. The compiler can no longer be sure that the values captured by the closure live long enough: they need to live at least as long as the closure, but it could be that the function is exited and its values freed while the threads still linger.

14. Are all the `clone()` calls necessary? Try to remove a couple.

    Solution: Without the clones, `mutexed_number` would be moved into the closure in the first iteration of the loop and hence not be available in the following iterations anymore.

15. The threads completed on my machine at the speed of about 30,000 to 40,000 per second. Is that fast?

    Solution: Depends on how you look at it. In the concurrent version, the 1 ms sleep (which is there to simulate a waiting state that is typical in programs that deal with I/O) is spread over all the threads, so it will not cause much overhead. If you had the same implementation without threads, you would spend 1,000 seconds just sleeping.

    Then again, if we just eliminate the sleep, a non-threaded version runs with a speed of about 45M iterations per second.

16. Take a peek at the official library documentation.

    Solution: Great docs, ain't they?

17. Take the game code and implement at least two creatures moving in on the map in their own, independent threads.

    Solution: Here's the simplest implementation, using `Arc` and mutex:

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

fn main() {
    let grid = Arc::new(Mutex::new(Grid::generate_empty(10,10)));
    let human = Being::Human;
    let orc = Being::Orc;

    grid.lock().unwrap().squares[0].being = Some(human);
    grid.lock().unwrap().squares[1].being = Some(orc);


    let grid2 = grid.clone();
    thread::spawn(move || {
        grid2.lock().unwrap().move_being_in_coord((0,0), Direction::East).expect("No being in thread 1"
    });

    let grid3 = grid.clone();
    thread::spawn(move || {
```

```
            grid3.lock().unwrap().move_being_in_coord((0,0), Direction::East).expect("No being in thread 2"
        });
    }
```

Chapter 8 - Macros

Metaprogramming can be used to augment the features of the programming language, or just to move some computation to compile time. The most stable form of metaprogramming in Rust is in the form of macros-by-example. These are created by the `macro!` keyword, and they allow limited generation of Rust code from templates at compile time.

The compiler can be instructed to output generated code (the `--pretty expanded` parameter). On the nightly compiler, there are further debugging macros which allow finer grained debugging of the macros.

Macros are used everywhere in Rust's standard library. For instance, the function for printing text to the screen is actually a macro, because it needs compile-time information about its parameters to do proper typechecking of format arguments.

Exercises

1. Why did println! need two patterns?

   Solution: The first pattern of println! matches a call where there is just the format string and no parameters. Without it, the following would match no rules:

```
println!("Hello world!")
```

2. Why is println! a macro instead of just a function?

   Solution: Having println! as a macro gives it the power of compile-time checking of its parameters. A print statement with a wrong number of arguments could never work, so it's nice that it fails at compile time. Furthermore, Rust does not have variable arguments (at least yet), so you would have to pass format arguments in a list or similar structure.

3. Think about your second favorite compiled programming language. How does it do the same checks that println! does via a macro? Is either choice superior?

   Solution: Python allows an arbitrary number of parameters in function calls, and has special syntax for formatting strings. This is quite flexible, but catches errors such as the following only at run-time:

```
print("This should have two parameters %s %s" % "but I gave only one")
```

   C allows doing the same as above using C's varargs, and does not even cause errors at runtime, but just prints garbage. A good compiler with all warnings turned on can warn about it at compile time, however.

4. Write a macro that takes an arbitrary number of elements and outputs an unordered HTML list in a literal string. For instance, html_list!([1, 2]) => 1/
2.

   Solution: This comes close, but inserts spaces between tokens:

```
macro_rules! html_list {
    ($($x:expr,)*) => { stringify!(<ul>$(<li>$x</li>)*) };
    ($($x:expr),*) => { stringify!(<ul>$(<li>$x</li>)*) };
}
```

   Unfortunately, there is no version of the stringify! macro that would not. Perhaps some day...

5. Write a macro that accepts the following language:

```
language = HELLO recipient;
 recipient = <String>;
```

   For instance, the following strings would be acceptable in this language:

```
HELLO world!
 HELLO Rustaceans!
```

Make the macro generate code that outputs a greeting directed at the recipient.

Solution: Need `stringify!` again.

```
macro_rules! greeting {
    (HELLO $x:tt!) => { println!("Hello, {}!", stringify!($x)) };
}
```

6. Write a macro that takes either of these two arbitrary sequences:

```
1, 2, 3
 1 => 2; 2 => 3
```

For the first pattern, it should generate a vector with all the values. For the second pattern, it should generate a HashMap with key=>value pairs.

Solution: Nothing odd going on here either. Just a lot of repeating expressions. The vector case is a single one liner that hands over to the `vec!` macro, while in the HashMap case, we first create it and then repeat inserts to fill it:

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

macro_rules! generate_vec_or_hash {
    ($($k:expr),*) => { vec!($($k,)*) };
    ($($k:expr => $v:expr),*) => {
        {
            let mut h = HashMap::new();
            $(
                h.insert($k, $v);
            )*
            h
        }
    }
}

fn main() {
    let vec = generate_vec_or_hash!(1, 2, 3);
    let dict = generate_vec_or_hash!("one" => 1, "two" => 2);

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

Chapter 9 - Compiler Plugins

Macros offer only limited forms of code generation, which is why other mechanisms have been invented and many are being designed. Most of these work only on nightly compiler.

Compiler plugins enable a wider range of compile-time computation. They are currently being used for extending compile-time validation: code linters, database schema checkers, and many others. They are pieces of Rust code, which are linked to the compiler process by special annotations.

Custom derives (structure and function annotations) are widely used in popular libraries such as the Diesel ORM and the serialization library, Serde. That's why they were quickly stabilized in late 2016 in the form of macros 1.1, and those libraries can therefore be used normally in stable Rust.

Code generation is a workaround for the stable Rust, which gives the same power as nightly's full compiler plugins. It works by an external crate syntex, which basically contains the same code generation tools, but does it in a separate step before the actual compilation.

Exercises

1. Write a trait, `Serializable`, with methods `ser` and `deser`. Create a custom derive attribute using macros 1.1 that implements those functions automatically. You don't have to be able to load and save every kind of type, just a few primitives will be more than fine.

   Solution: This was a deliberately open-ended question with vague specs. Here's possibly the simplest implementation:

```
trait Serializable {
    fn ser(&self) -> String;
    fn deser(s: &str) -> Self;
}

impl Serializable for u32 {
    fn ser(&self) -> String {
        self.to_string()
    }

    fn deser(s: &str) -> Self {
        s.parse().expect("Deserializing number failed")
    }
}
```

2. Write a compiler plugin that disallows functions that are too long.

   Solution: Here's an implementation that just counts characters of the debug string contents of a function block in a late linter pass:

```
impl<'a, 'ctx> LateLintPass<'a, 'ctx> for Pass {
    fn check_fn(&mut self, cx: &LateContext, _: FnKind, _: &FnDecl, body: &Body, span: Span, _: NodeId)
        let body_value = format!("{:?}", body.value);
        if body_value.len() > 200 {
            cx.span_lint(TEST_FN_RETURN, span, "function too large");
        }
    }
}
```

   Since this is unstable stuff, the API might very well be quite different when you're reading this. http://manishearth.github.io/rust-internals-docs/ may have documentation about the current state of things.

Chapter 10 - Unsafety and Interfacing with Other Languages

Rust has an unsafe mode which lifts the following restrictions: updating mutable static variables, accessing raw pointers, and calling unsafe functions. Unsafety can be requested for functions, blocks, traits, or implementations.

The typical case when unsafety is required is when interfacing with another language, such as using a library written in C from Rust.

For interfacing with other languages, there are several external crates, such as Ruru for interfacing with Ruby and Neon for JavaScript.

Exercises

1. Write the ruby module that runs the Rust version of the square root function. Additionally, find out if there's a combinator function in `Result` that does the unwrapping of the `e` parameter in a more concise way.

   Solution: Here's the ruby code that runs the Rust-made square root:

```
require 'fiddle'

library = Fiddle::dlopen('target/release/libruru_test.so')
Fiddle::Function.new(library['initialize_sum_floats'], [], Fiddle::TYPE_VOIDP).call

S=9.0
puts S.square_root
```

   And here's a oneliner for the unwrapping match block of `e`:

```
let e = e.unwrap_or(Float::new(0.00000001)).to_f64();
```

2. Extend the `ncurses` library by a few additional functions from the library. Create safe wrappers for them and use them.

   Solution: Check the main page for curses, and implement new wrappers in the `ncurses impl` block. For instance, here's a wrapper for `mvgetch`:

```
fn move_and_get_char(y: usize, x: usize) -> usize {
    unsafe {
        mvgetch(y, x)
    }
}
```

3. Extend the safe wrappers of `ncurses` library: could a macro-by-example macro be used to make a safer `printw`? Could the initialization and deinitialization of the screen be made in a constructor and destructor implicitly?

   Solution: You'll need to do three things:

1. Make all the calls methods (add `&self` to them) and call them that way in `main`.
2. Make a new method that calls `init_screen`, and use it in `main` to instantiate a new `ncurses` object.
3. Implement the `Drop` trait that calls `deinit_screen`. Then you can remove the `init` and `deinit` calls because they will be both automatically called. The `Drop` implementation is simple:

```
impl Drop for Ncurses {
    fn drop(&mut self) {
        self.deinit_screen();
    }
}
```

Chapter 11 - Parsing and Serialization

Parsing is a well-researched technique of making sense of a linear sequence of data, usually a string. Serialization is turning an internal representation of data to a linear sequence, and deserialization is vice versa. Parsing and serialization are easy to confuse, since parsing techniques are usually employed in deserialization.

Parser combinators are tools that allow making large parsers out of smaller ones. Examples of such libraries written in and for Rust include nom and Chomp. Another useful tool is parser expression grammar or PEG. One library that implements a PEG is called Pest.

Serde is the de facto standard library for creating (de)serializators in Rust.

Exercises

1. `combine` is a parser framework that is similar to chomp. Translate the chomp examples to `combine`.

   Solution: Here's the first example in `combine`:

```
extern crate combine;

use combine::{many, Parser};
use combine::char::{alpha_num, space};

fn main() {
    let (x, y): (String, &str) =
        many(alpha_num()
             .or(space()))
             .parse("String containing 123 non-alphanumerics").unwrap();
    println!("{:?} {:?}", x, y);
}
```

2. Pest is a PEG parser generator. Implement the ISO-8601 date standard (or parts of it) using Pest.

   Solution: Here's a Pest parser that can read 2016-11-20T19:50:49+02:00:

```
impl_rdp! {
    grammar! {
        expression = { date ~ ["T"] ~ time ~ ["+"] ~ timezone }
        date = { year ~ ["-"] ~ month ~ ["-"] ~ day }
        time = { hour ~ [":"] ~ minute ~ [":"] ~ second }
        timezone = { hour ~ [":"] ~ minute }

        year = @{ ['0'..'9']* }
        month = @{ ['0'..'1'] ~ ['0'..'9'] }
        day = @{ ['0'..'3'] ~ ['0'..'9'] }

        hour = @{ ['0'..'2'] ~ ['0'..'9'] }
        minute = @{ ['0'..'6'] ~ ['0'..'9'] }
        second = @{ ['0'..'6'] ~ ['0'..'9'] }
    }
}
```

   This just implements the parser. To manipulate the result further, check out the `process!` macro in Pest's documentation.

3. Take a look at one of the Serde serializer/deserializer implementations, for instance `serde_json`. How would you implement a new serializer for a new data format?

   Solution: Minimally, from `serde::de`, you would need to implement `Visitor` for your own `FoobarVisitor` struct. Then deserialize implementations for all the output datatypes you wish to support.

4. From `serde::ser`, you'll just need to implement the `Serialize` trait.

Chapter 12 - Web Programming

Rust's web programming story is getting better all the time. It's quite a bit stronger as a backend programming language, but there's some interesting development on the frontend as well. Rust has the potential of being in the top 10 of web platforms performance-wise due to its low-level nature and zero-cost philosophy.

The established HTTP library is Hyper, which gives both client and server-side support functionality. Hyper is strongly typed wherever it makes sense, so many kinds of accidental errors are not possible to sneak through the compiler.

Iron and Nickel are the oldest stabilized web frameworks for Rust. They're both designed around a middleware design and have been working on the stable compiler for a longer time. Rocket is a newer framework, which tries to be closer to frameworks found on more dynamic languages. It relies heavily on advanced metaprogramming features, and hence works only on the nightly Rust.

Exercises

1. Extend the arena game by showing more information about all the characters in the main GET /game page.

   Solution: Edit `game.hbs` and include the stats in the form:

```
            <form action="/attack/{{@key}}" method="POST">
                 {{name}} str: {{strength}} dex: {{dexterity}} hp: {{hitpoints}}
                      <button>⚔</button>
              </form>
```

2. The arena game loses all its state when the program is restarted. A typical way for a web application is to store all the state in a relational database. Think of other ways. What would be the simplest way? Implement it.

   Solution: We could use Serde to save JSON for us. Derive serialize and deserialize for all the data you want to save. Then, getting a JSON representation as a string is as simple as calling:

```
serde_json::to_string_pretty(self)
```

   This outputs a string which you can save to a file. At next boot, read it from the file and deserialize with:

```
serde_json::from_str(&input_string)
```

3. Implement a pause in the arena game so that people cannot just spam the attack link.

   Solution: The `Character` struct is already a `Mutex`/`Arc`-protected global HashMap, so we could use that. Store a timestamp in it, update it on every attack, and refuse to process the attack request if the timestamp is too close to the current time. Crate time has a `get_time` function that works well for this.

4. Make the game prettier: serve a static CSS and link it to the templates.

   Solution: Put your styles inside `static/css/style.css`. Then, link to them in the standard way in your templates (no static/ here, that will be added in the handler code):

```
<head>
    <link rel="stylesheet" href="/css/style.css">
</head>
Add a static file handler:
#[get("/<path..>", rank=1)]
fn static_files(path: PathBuf) -> io::Result<NamedFile> {
    NamedFile::open(Path::new("static/").join(path))
}
```

   And add a reference to it in your routes:

```
mount("/", routes![index, new, post_new, game, attack, static_files])
```

Chapter 13 - Data Storage

Rust has plenty of libraries to interact with external storage engines. All established open-source databases have support. A library called `r2d2` provides database pooling.

Several ORM-like libraries are being written, the most hip one at the time of writing being Diesel. Diesel uses macros 1.1, giving a query language and a statically checked access to the database: the models are verified at compile time to match with the actual database.

No exercises in this chapter.

Chapter 14 - Debugging

A compiled Rust program is close enough to a compiled C program that with just a small wrapper, the same debugging tools can be used effectively. Debugging practically requires instrumentation code in the binary, which the Rust compiler defaults to.

GDB and LLDB can both be used for debugging Rust binaries. Their primary use case is to step through the code line by line, while investigating the state of the program. Also threads can be manipulated in this way.

To make the debugging experience smoother, the debugging tools can be integrated to most text editors.

Exercises

1. Try the `explore` command of GDB. How does it differ from `print`?

   Solution: `print` gives you a single representation of a value. `explore` is more conversational: if there's just a single value, it displays that, but if used on a compound value (such as `nums` in the `example` function), it interactively allows diving as deep to the structure as it goes.

2. Try the `step` command in GDB or LLDB. How is it different? What happens between the function calls?

   Solution: They're pretty much the same. Both descend to the underlying library code. LLDB is a tad more verbose.

3. Both GDB and LLDB support attaching to a running process. Make a program that doesn't quit, perhaps an erroneous never-ending loop, then attach to it via `rust-gdb` or `rust-lldb` and see what happens.

   Solution: Run the program, find its PID (in most Unix systems, using a tool like **ps aux**), then run `rust-gdb -pid=PID`. You may need elevated (root) privileges to do this, since many systems deny attaching to a running process from regular users. This is unfortunate, since when using `rustup` (like recommended in this book), your root user won't have the same environments set up as your user, so `rust-gdb` and `rust-lldb` don't just work.

   You can try running regular GDB and LLDB as root against the PID, though. You'll just miss the Rust pretty prints.

4. Could you fix a program running in production by attaching to it? What prerequisites need to be fulfilled before?

   Solution: Theoretically possible. You could attach to the process with the process shown above, and change the variables on the fly. If you need to edit the actual running code, that's a bit more difficult: you can edit the assembler code. Not recommended.

   The code needs to be running in debug mode instead of release mode.

5. Check out the disassembler command in either of the debuggers.

   Solution: The disassembler command shows assembler code of a running program.

6. If Visual Code Studio isn't your preferred editor, try to find out if your favourite one has debugger integration for Rust.

   Solution: Keeping up with the *are we X yet* tradition, https://areweideyet.com/ should give you a good overview of the current text editor/IDE situation of Rust.