The Rust Programming Language

The paper used in this publication may meet the minimum requirements of the American National Standard for Information Sciences — Permanence of Paper for Printed Library Materials, ANSI Z39.48–1984.

10 9 8 7 6 5 4 3 2 1

Rust 1.0.0:	15 May 2015
Rust 1.1.0:	25 June 2015
Rust 1.2.0:	7 August 2015
Rust 1.3.0:	17 September 2015
Rust 1.4.0:	29 October 2015

This book was typeset with LaTeX, in
Adobe Source Serif Pro, *Heuristica*, Fira Sans, and Adobe Source Code Pro.
This version was generated on November 28, 2015, and corresponds to Rust 1.4.0.

Short Table of Contents

Table of Contents

Part I

The Rust Programming Language

Welcome! This book will teach you about the Rust Programming Language[1]. Rust is a systems programming language focused on three goals: safety, speed, and concurrency. It maintains these goals without having a garbage collector, making it a useful language for a number of use cases other languages aren't good at: embedding in other languages, programs with specific space and time requirements, and writing low-level code, like device drivers and operating systems. It improves on current languages targeting this space by having a number of compile-time safety checks that produce no runtime overhead, while eliminating all data races. Rust also aims to achieve 'zero-cost abstractions' even though some of these abstractions feel like those of a high-level language. Even then, Rust still allows precise control like a low-level language would.

"The Rust Programming Language" is split into eight sections. This introduction is the first. After this:

- Getting started[2] - Set up your computer for Rust development.
- Learn Rust[3] - Learn Rust programming through small projects.
- Effective Rust[4] - Higher-level concepts for writing excellent Rust code.
- Syntax and Semantics[5] - Each bit of Rust, broken down into small chunks.
- Nightly Rust[6] - Cutting-edge features that aren't in stable builds yet.
- Glossary[7] - A reference of terms used in the book.
- Bibliography[8] - Background on Rust's influences, papers about Rust.

After reading this introduction, you'll want to dive into either 'Learn Rust' or 'Syntax and Semantics', depending on your preference: 'Learn Rust' if you want to dive in with a project, or 'Syntax and Semantics' if you prefer to start small, and learn a single concept thoroughly before moving onto the next. Copious cross-linking connects these parts together.

Contributing

The source files from which this book is generated can be found on GitHub[9].

[1] https://www.rust-lang.org
[2] getting-started.html
[3] learn-rust.html
[4] effective-rust.html
[5] syntax-and-semantics.html
[6] nightly-rust.html
[7] glossary.html
[8] bibliography.html
[9] https://github.com/rust-lang/rust/tree/master/src/doc/book

Chapter 1

Getting Started

This first section of the book will get us going with Rust and its tooling. First, we'll install Rust. Then, the classic 'Hello World' program. Finally, we'll talk about Cargo, Rust's build system and package manager.

1.1 Installing Rust

The first step to using Rust is to install it. Generally speaking, you'll need an Internet connection to run the commands in this chapter, as we'll be downloading Rust from the internet.

We'll be showing off a number of commands using a terminal, and those lines all start with `$`. We don't need to type in the `$`s, they are there to indicate the start of each command. We'll see many tutorials and examples around the web that follow this convention: `$` for commands run as our regular user, and `#` for commands we should be running as an administrator.

1.1.1 Platform support

The Rust compiler runs on, and compiles to, a great number of platforms, though not all platforms are equally supported. Rust's support levels are organized into three tiers, each with a different set of guarantees.

Platforms are identified by their "target triple" which is the string to inform the compiler what kind of output should be produced. The columns below indicate whether the corresponding component works on the specified platform.

Tier 1

Tier 1 platforms can be thought of as "guaranteed to build and work". Specifically they will each satisfy the following requirements:

- Automated testing is set up to run tests for the platform.

- Landing changes to the `rust-lang/rust` repository's master branch is gated on tests passing.
- Official release artifacts are provided for the platform.
- Documentation for how to use and how to build the platform is available.

Target	std	rustc	cargo	notes
`x86_64-pc-windows-msvc`	✓	✓	✓	64-bit MSVC (Windows 7+)
`i686-pc-windows-gnu`	✓	✓	✓	32-bit MinGW (Windows 7+)
`x86_64-pc-windows-gnu`	✓	✓	✓	64-bit MinGW (Windows 7+)
`i686-apple-darwin`	✓	✓	✓	32-bit OSX (10.7+, Lion+)
`x86_64-apple-darwin`	✓	✓	✓	64-bit OSX (10.7+, Lion+)
`i686-unknown-linux-gnu`	✓	✓	✓	32-bit Linux (2.6.18+)
`x86_64-unknown-linux-gnu`	✓	✓	✓	64-bit Linux (2.6.18+)

Tier 2

Tier 2 platforms can be thought of as "guaranteed to build". Automated tests are not run so it's not guaranteed to produce a working build, but platforms often work to quite a good degree and patches are always welcome! Specifically, these platforms are required to have each of the following:

- Automated building is set up, but may not be running tests.
- Landing changes to the `rust-lang/rust` repository's master branch is gated on platforms **building**. Note that this means for some platforms only the standard library is compiled, but for others the full bootstrap is run.
- Official release artifacts are provided for the platform.

Target	std	rustc	cargo	notes
`i686-pc-windows-msvc`	✓	✓	✓	32-bit MSVC (Windows 7+)

Tier 3

Tier 3 platforms are those which Rust has support for, but landing changes is not gated on the platform either building or passing tests. Working builds for these platforms may be spotty as their reliability is often defined in terms of community contributions. Additionally, release artifacts and installers are not provided, but there may be community infrastructure producing these in unofficial locations.

Target	std	rustc	cargo	notes
`x86_64-unknown-linux-musl`	✓			64-bit Linux with MUSL
`arm-linux-androideabi`	✓			ARM Android
`i686-linux-android`	✓			32-bit x86 Android
`aarch64-linux-android`	✓			ARM64 Android

Target	std	rustc	cargo	notes
`arm-unknown-linux-gnueabi`	✓	✓		ARM Linux (2.6.18+)
`arm-unknown-linux-gnueabihf`	✓	✓		ARM Linux (2.6.18+)
`aarch64-unknown-linux-gnu`	✓			ARM64 Linux (2.6.18+)
`mips-unknown-linux-gnu`	✓			MIPS Linux (2.6.18+)
`mipsel-unknown-linux-gnu`	✓			MIPS (LE) Linux (2.6.18+)
`powerpc-unknown-linux-gnu`	✓			PowerPC Linux (2.6.18+)
`i386-apple-ios`	✓			32-bit x86 iOS
`x86_64-apple-ios`	✓			64-bit x86 iOS
`armv7-apple-ios`	✓			ARM iOS
`armv7s-apple-ios`	✓			ARM iOS
`aarch64-apple-ios`	✓			ARM64 iOS
`i686-unknown-freebsd`	✓	✓		32-bit FreeBSD
`x86_64-unknown-freebsd`	✓	✓		64-bit FreeBSD
`x86_64-unknown-openbsd`	✓	✓		64-bit OpenBSD
`x86_64-unknown-netbsd`	✓	✓		64-bit NetBSD
`x86_64-unknown-bitrig`	✓	✓		64-bit Bitrig
`x86_64-unknown-dragonfly`	✓	✓		64-bit DragonFlyBSD
`x86_64-rumprun-netbsd`	✓			64-bit NetBSD Rump Kernel
`i686-pc-windows-msvc` (XP)	✓			Windows XP support
`x86_64-pc-windows-msvc` (XP)	✓			Windows XP support

Note that this table can be expanded over time, this isn't the exhaustive set of tier 3 platforms that will ever be!

1.1.2 Installing on Linux or Mac

If we're on Linux or a Mac, all we need to do is open a terminal and type this:

```
$ curl -sSf https://static.rust-lang.org/rustup.sh | sh
```

This will download a script, and stat the installation. If it all goes well, you'll see this appear:

```
Welcome to Rust.

This script will download the Rust compiler and its package manager, Cargo, and
install them to /usr/local. You may install elsewhere by running this script
with the --prefix=<path> option.

The installer will run under 'sudo' and may ask you for your password. If you do
not want the script to run 'sudo' then pass it the --disable-sudo flag.

You may uninstall later by running /usr/local/lib/rustlib/uninstall.sh,
or by running this script again with the --uninstall flag.

Continue? (y/N)
```

From here, press `y` for 'yes', and then follow the rest of the prompts.

1.1.3 Installing on Windows

If you're on Windows, please download the appropriate installer[1].

1.1.4 Uninstalling

Uninstalling Rust is as easy as installing it. On Linux or Mac, just run the uninstall script:

```
$ sudo /usr/local/lib/rustlib/uninstall.sh
```

If we used the Windows installer, we can re-run the `.msi` and it will give us an uninstall option.

1.1.5 Troubleshooting

If we've got Rust installed, we can open up a shell, and type this:

```
$ rustc --version
```

You should see the version number, commit hash, and commit date.

If you do, Rust has been installed successfully! Congrats!

If you don't and you're on Windows, check that Rust is in your %PATH% system variable. If it isn't, run the installer again, select "Change" on the "Change, repair, or remove installation" page and ensure "Add to PATH" is installed on the local hard drive.

If not, there are a number of places where we can get help. The easiest is the #rust IRC channel on irc.mozilla.org[2], which we can access through Mibbit[3]. Click that link, and we'll be chatting with other Rustaceans (a silly nickname we call ourselves) who can help us out. Other great resources include the user's forum[4], and Stack Overflow[5].

This installer also installs a copy of the documentation locally, so we can read it offline. On UNIX systems, `/usr/local/share/doc/rust` is the location. On Windows, it's in a `share/doc` directory, inside the directory to which Rust was installed.

1.2 Hello, world!

Now that you have Rust installed, we'll help you write your first Rust program. It's traditional when learning a new language to write a little program to print the text "Hello, world!" to the screen, and in this section, we'll follow that tradition.

The nice thing about starting with such a simple program is that you can quickly verify that your compiler is installed, and that it's working properly. Printing information to the screen is also just a pretty common thing to do, so practicing it early on is good.

[1] `https://www.rust-lang.org/install.html`
[2] `irc://irc.mozilla.org/#rust`
[3] `http://chat.mibbit.com/?server=irc.mozilla.org&channel=%23rust`
[4] `https://users.rust-lang.org/`
[5] `http://stackoverflow.com/questions/tagged/rust`

> Note: This book assumes basic familiarity with the command line. Rust itself makes no specific demands about your editing, tooling, or where your code lives, so if you prefer an IDE to the command line, that's an option. You may want to check out SolidOak[6], which was built specifically with Rust in mind. There are a number of extensions in development by the community, and the Rust team ships plugins for various editors[7]. Configuring your editor or IDE is out of the scope of this tutorial, so check the documentation for your specific setup.

1.2.1 Creating a Project File

First, make a file to put your Rust code in. Rust doesn't care where your code lives, but for this book, I suggest making a *projects* directory in your home directory, and keeping all your projects there. Open a terminal and enter the following commands to make a directory for this particular project:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

> Note: If you're on Windows and not using PowerShell, the ~ may not work. Consult the documentation for your shell for more details.

1.2.2 Writing and Running a Rust Program

Next, make a new source file and call it *main.rs*. Rust files always end in a *.rs* extension. If you're using more than one word in your filename, use an underscore to separate them; for example, you'd use *hello_world.rs* rather than *helloworld.rs*.

Now open the *main.rs* file you just created, and type the following code:

```
fn main() {
    println!("Hello, world!");
}
```

Save the file, and go back to your terminal window. On Linux or OSX, enter the following commands:

```
$ rustc main.rs
$ ./main
Hello, world!
```

In Windows, just replace `main` with `main.exe`. Regardless of your operating system, you should see the string `Hello, world!` print to the terminal. If you did, then congratulations! You've officially written a Rust program. That makes you a Rust programmer! Welcome.

[6] https://github.com/oakes/SolidOak

[7] https://github.com/rust-lang/rust/blob/master/src/etc/CONFIGS.md

1.2.3 Anatomy of a Rust Program

Now, let's go over what just happened in your "Hello, world!" program in detail. Here's the first piece of the puzzle:

```
fn main() {

}
```

These lines define a *function* in Rust. The `main` function is special: it's the beginning of every Rust program. The first line says, "I'm declaring a function named `main` that takes no arguments and returns nothing." If there were arguments, they would go inside the parentheses (`(` and `)`), and because we aren't returning anything from this function, we can omit the return type entirely.

Also note that the function body is wrapped in curly braces (`{` and `}`). Rust requires these around all function bodies. It's considered good style to put the opening curly brace on the same line as the function declaration, with one space in between.

Inside the `main()` function:

```
    println!("Hello, world!");
```

This line does all of the work in this little program: it prints text to the screen. There are a number of details that are important here. The first is that it's indented with four spaces, not tabs.

The second important part is the `println!()` line. This is calling a Rust *macro*[8], which is how metaprogramming is done in Rust. If it were calling a function instead, it would look like this: `println()` (without the !). We'll discuss Rust macros in more detail later, but for now you just need to know that when you see a `!` that means that you're calling a macro instead of a normal function.

Next is `"Hello, world!"` which is a *string*. Strings are a surprisingly complicated topic in a systems programming language, and this is a *statically allocated*[9] string. We pass this string as an argument to `println!`, which prints the string to the screen. Easy enough!

The line ends with a semicolon (`;`). Rust is an *[expression oriented]* language, which means that most things are expressions, rather than statements. The `;` indicates that this expression is over, and the next one is ready to begin. Most lines of Rust code end with a `;`.

1.2.4 Compiling and Running Are Separate Steps

In "Writing and Running a Rust Program", we showed you how to run a newly created program. We'll break that process down and examine each step now.

Before running a Rust program, you have to compile it. You can use the Rust compiler by entering the `rustc` command and passing it the name of your source file, like this:

[8] `macros.html`
[9] `the-stack-and-the-heap.html`

```
$ rustc main.rs
```

If you come from a C or C++ background, you'll notice that this is similar to `gcc` or `clang`. After compiling successfully, Rust should output a binary executable, which you can see on Linux or OSX by entering the `ls` command in your shell as follows:

```
$ ls
main  main.rs
```

On Windows, you'd enter:

```
$ dir
main.exe  main.rs
```

This shows we have two files: the source code, with an `.rs` extension, and the executable (`main.exe` on Windows, `main` everywhere else). All that's left to do from here is run the `main` or `main.exe` file, like this:

```
$ ./main  # or main.exe on Windows
```

If *main.rs* were your "Hello, world!" program, this would print `Hello, world!` to your terminal.

If you come from a dynamic language like Ruby, Python, or JavaScript, you may not be used to compiling and running a program being separate steps. Rust is an *ahead-of-time compiled* language, which means that you can compile a program, give it to someone else, and they can run it even without Rust installed. If you give someone a `.rb` or `.py` or `.js` file, on the other hand, they need to have a Ruby, Python, or JavaScript implementation installed (respectively), but you only need one command to both compile and run your program. Everything is a tradeoff in language design.

Just compiling with `rustc` is fine for simple programs, but as your project grows, you'll want to be able to manage all of the options your project has, and make it easy to share your code with other people and projects. Next, I'll introduce you to a tool called Cargo, which will help you write real-world Rust programs.

1.3 Hello, Cargo!

Cargo is Rust's build system and package manager, and Rustaceans use Cargo to manage their Rust projects. Cargo manages three things: building your code, downloading the libraries your code depends on, and building those libraries. We call libraries your code needs 'dependencies' since your code depends on them.

The simplest Rust programs don't have any dependencies, so right now, you'd only use the first part of its functionality. As you write more complex Rust programs, you'll want to add dependencies, and if you start off using Cargo, that will be a lot easier to do.

As the vast, vast majority of Rust projects use Cargo, we will assume that you're using it for the rest of the book. Cargo comes installed with Rust itself, if you used the official installers. If you installed Rust through some other means, you can check if you have Cargo installed by typing:

```
$ cargo --version
```

Into a terminal. If you see a version number, great! If you see an error like '`command not found`', then you should look at the documentation for the system in which you installed Rust, to determine if Cargo is separate.

1.3.1 Converting to Cargo

Let's convert the Hello World program to Cargo. To Cargo-fy a project, you need to do three things:

1. Put your source file in the right directory.
2. Get rid of the old executable (`main.exe` on Windows, `main` everywhere else) and make a new one.
3. Make a Cargo configuration file.

Let's get started!

Creating a new Executable and Source Directory

First, go back to your terminal, move to your *hello_world* directory, and enter the following commands:

```
$ mkdir src
$ mv main.rs src/main.rs
$ rm main  # or 'del main.exe' on Windows
```

Cargo expects your source files to live inside a *src* directory, so do that first. This leaves the top-level project directory (in this case, *hello_world*) for READMEs, license information, and anything else not related to your code. In this way, using Cargo helps you keep your projects nice and tidy. There's a place for everything, and everything is in its place.

Now, copy *main.rs* to the *src* directory, and delete the compiled file you created with `rustc`. As usual, replace `main` with `main.exe` if you're on Windows.

This example retains `main.rs` as the source filename because it's creating an executable. If you wanted to make a library instead, you'd name the file `lib.rs`. This convention is used by Cargo to successfully compile your projects, but it can be overridden if you wish.

Creating a Configuration File

Next, create a new file inside your *hello_world* directory, and call it `Cargo.toml`.

Make sure to capitalize the `C` in `Cargo.toml`, or Cargo won't know what to do with the configuration file.

This file is in the *TOML*[10] (Tom's Obvious, Minimal Language) format. TOML is similar to INI, but has some extra goodies, and is used as Cargo's configuration format.

Inside this file, type the following information:

[10] `https://github.com/toml-lang/toml`

```
[package]

name = "hello_world"
version = "0.0.1"
authors = [ "Your name <you@example.com>" ]
```

The first line, `[package]`, indicates that the following statements are configuring a package. As we add more information to this file, we'll add other sections, but for now, we just have the package configuration.

The other three lines set the three bits of configuration that Cargo needs to know to compile your program: its name, what version it is, and who wrote it.

Once you've added this information to the *Cargo.toml* file, save it to finish creating the configuration file.

1.3.2 Building and Running a Cargo Project

With your *Cargo.toml* file in place in your project's root directory, you should be ready to build and run your Hello World program! To do so, enter the following commands:

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/yourname/projects/hello_world)
$ ./target/debug/hello_world
Hello, world!
```

Bam! If all goes well, `Hello, world!` should print to the terminal once more.

You just built a project with `cargo build` and ran it with `./target/debug/hello_world`, but you can actually do both in one step with `cargo run` as follows:

```
$ cargo run
     Running `target/debug/hello_world`
Hello, world!
```

Notice that this example didn't re-build the project. Cargo figured out that the file hasn't changed, and so it just ran the binary. If you'd modified your source code, Cargo would have rebuilt the project before running it, and you would have seen something like this:

```
$ cargo run
   Compiling hello_world v0.0.1 (file:///home/yourname/projects/hello_world)
     Running `target/debug/hello_world`
Hello, world!
```

Cargo checks to see if any of your project's files have been modified, and only rebuilds your project if they've changed since the last time you built it.

With simple projects, Cargo doesn't bring a whole lot over just using `rustc`, but it will become useful in future. With complex projects composed of multiple crates, it's much easier to let Cargo coordinate the build. With Cargo, you can just run `cargo build`, and it should work the right way.

1.3.3 Building for Release

When your project is finally ready for release, you can use `cargo build --release` to compile your project with optimizations. These optimizations make your Rust code run faster, but turning them on makes your program take longer to compile. This is why there are two different profiles, one for development, and one for building the final program you'll give to a user.

Running this command also causes Cargo to create a new file called *Cargo.lock*, which looks like this:

```
[root]
name = "hello_world"
version = "0.0.1"
```

Cargo uses the *Cargo.lock* file to keep track of dependencies in your application. This is the Hello World project's *Cargo.lock* file. This project doesn't have dependencies, so the file is a bit sparse. Realistically, you won't ever need to touch this file yourself; just let Cargo handle it.

That's it! If you've been following along, you should have successfully built `hello_world` with Cargo.

Even though the project is simple, it now uses much of the real tooling you'll use for the rest of your Rust career. In fact, you can expect to start virtually all Rust projects with some variation on the following commands:

```
$ git clone someurl.com/foo
$ cd foo
$ cargo build
```

1.3.4 Making A New Cargo Project the Easy Way

You don't have to go through that previous process every time you want to start a new project! Cargo can quickly make a bare-bones project directory that you can start developing in right away.

To start a new project with Cargo, enter `cargo new` at the command line:

```
$ cargo new hello_world --bin
```

This command passes `--bin` because the goal is to get straight to making an executable application, as opposed to a library. Executables are often called *binaries* (as in `/usr/bin`, if you're on a Unix system).

Cargo has generated two files and one directory for us: a `Cargo.toml` and a *src* directory with a *main.rs* file inside. These should look familliar, they're exactly what we created by hand, above.

This output is all you need to get started. First, open `Cargo.toml`. It should look something like this:

```
[package]

name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
```

Cargo has populated *Cargo.toml* with reasonable defaults based on the arguments you gave it and your `git` global configuration. You may notice that Cargo has also initialized the `hello_world` directory as a `git` repository.

Here's what should be in `src/main.rs`:

```
fn main() {
    println!("Hello, world!");
}
```

Cargo has generated a "Hello World!" for you, and you're ready to start coding!

> Note: If you want to look at Cargo in more detail, check out the official Cargo guide[11], which covers all of its features.

1.4 Closing Thoughts

This chapter covered the basics that will serve you well through the rest of this book, and the rest of your time with Rust. Now that you've got the tools down, we'll cover more about the Rust language itself.

You have two options: Dive into a project with 'Learn Rust[12]', or start from the bottom and work your way up with 'Syntax and Semantics[13]'. More experienced systems programmers will probably prefer 'Learn Rust', while those from dynamic backgrounds may enjoy either. Different people learn differently! Choose whatever's right for you.

[11] `http://doc.crates.io/guide.html`
[12] `learn-rust.html`
[13] `syntax-and-semantics.html`

Chapter 2

Learn Rust

Welcome! This section has a few tutorials that teach you Rust through building projects. You'll get a high-level overview, but we'll skim over the details.

If you'd prefer a more 'from the ground up'-style experience, check out Syntax and Semantics[1].

2.1 Guessing Game

For our first project, we'll implement a classic beginner programming problem: the guessing game. Here's how it works: Our program will generate a random integer between one and a hundred. It will then prompt us to enter a guess. Upon entering our guess, it will tell us if we're too low or too high. Once we guess correctly, it will congratulate us. Sounds good?

2.1.1 Set up

Let's set up a new project. Go to your projects directory. Remember how we had to create our directory structure and a `Cargo.toml` for `hello_world`? Cargo has a command that does that for us. Let's give it a shot:

```
$ cd ~/projects
$ cargo new guessing_game --bin
$ cd guessing_game
```

We pass the name of our project to `cargo new`, and then the `--bin` flag, since we're making a binary, rather than a library.

Check out the generated `Cargo.toml`:

```
[package]
```

[1] `syntax-and-semantics.html`

```
name = ”guessing_game”
version = ”0.1.0”
authors = [”Your Name <you@example.com>”]
```

Cargo gets this information from your environment. If it’s not correct, go ahead and fix that.

Finally, Cargo generated a ‘Hello, world!’ for us. Check out `src/main.rs`:

```
fn main() {
    println!(”Hello, world!”);
}
```

Let’s try compiling what Cargo gave us:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
```

Excellent! Open up your `src/main.rs` again. We’ll be writing all of our code in this file.

Before we move on, let me show you one more Cargo command: `run`. `cargo run` is kind of like `cargo build`, but it also then runs the produced executable. Try it out:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running ‘target/debug/guessing_game‘
Hello, world!
```

Great! The `run` command comes in handy when you need to rapidly iterate on a project. Our game is just such a project, we need to quickly test each iteration before moving on to the next one.

2.1.2 Processing a Guess

Let’s get to it! The first thing we need to do for our guessing game is allow our player to input a guess. Put this in your `src/main.rs`:

```
use std::io;

fn main() {
    println!(”Guess the number!”);

    println!(”Please input your guess.”);

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect(”Failed to read line”);

    println!(”You guessed: {}”, guess);
}
```

There's a lot here! Let's go over it, bit by bit.

```
use std::io;
```

We'll need to take user input, and then print the result as output. As such, we need the `io` library from the standard library. Rust only imports a few things by default into every program, the 'prelude'[2]. If it's not in the prelude, you'll have to `use` it directly. There is also a second 'prelude', the `io` prelude[3], which serves a similar function: you import it, and it imports a number of useful, `io`-related things.

```
fn main() {
```

As you've seen before, the `main()` function is the entry point into your program. The `fn` syntax declares a new function, the `()`s indicate that there are no arguments, and `{` starts the body of the function. Because we didn't include a return type, it's assumed to be `()`, an empty tuple[4].

```
    println!("Guess the number!");

    println!("Please input your guess.");
```

We previously learned that `println!()` is a macro[5] that prints a string[6] to the screen.

```
    let mut guess = String::new();
```

Now we're getting interesting! There's a lot going on in this little line. The first thing to notice is that this is a let statement[7], which is used to create 'variable bindings'. They take this form:

```
let foo = bar;
```

This will create a new binding named `foo`, and bind it to the value `bar`. In many languages, this is called a 'variable', but Rust's variable bindings have a few tricks up their sleeves.

For example, they're immutable[8] by default. That's why our example uses `mut`: it makes a binding mutable, rather than immutable. `let` doesn't take a name on the left hand side of the assignment, it actually accepts a 'pattern[9]'. We'll use patterns later. It's easy enough to use for now:

```
let foo = 5; // immutable.
let mut bar = 5; // mutable
```

[2] ../std/prelude/index.html
[3] ../std/io/prelude/index.html
[4] primitive-types.html#tuples
[5] macros.html
[6] strings.html
[7] variable-bindings.html
[8] mutability.html
[9] patterns.html

Oh, and `//` will start a comment, until the end of the line. Rust ignores everything in comments[10].

So now we know that `let mut guess` will introduce a mutable binding named `guess`, but we have to look at the other side of the `=` for what it's bound to: `String::new()`.

`String` is a string type, provided by the standard library. A `String`[11] is a growable, UTF-8 encoded bit of text.

The `::new()` syntax uses `::` because this is an 'associated function' of a particular type. That is to say, it's associated with `String` itself, rather than a particular instance of a `String`. Some languages call this a 'static method'.

This function is named `new()`, because it creates a new, empty `String`. You'll find a `new()` function on many types, as it's a common name for making a new value of some kind.

Let's move forward:

```
    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");
```

That's a lot more! Let's go bit-by-bit. The first line has two parts. Here's the first:

```
io::stdin()
```

Remember how we `used` `std::io` on the first line of the program? We're now calling an associated function on it. If we didn't `use std::io`, we could have written this line as `std::io::stdin()`.

This particular function returns a handle to the standard input for your terminal. More specifically, a std::io::Stdin[12].

The next part will use this handle to get input from the user:

```
.read_line(&mut guess)
```

Here, we call the `read_line()`[13] method on our handle. Methods[14] are like associated functions, but are only available on a particular instance of a type, rather than the type itself. We're also passing one argument to `read_line()`: `&mut guess`.

Remember how we bound `guess` above? We said it was mutable. However, `read_line` doesn't take a `String` as an argument: it takes a `&mut String`. Rust has a feature called 'references[15]', which allows you to have multiple references to one piece of data, which can reduce copying. References are a complex feature, as one of Rust's major selling points is how safe and easy it is to use references. We don't need to know a lot of those details to finish our program right now, though. For now, all we need to know is that like `let` bindings, references are immutable by default. Hence, we need to write `&mut guess`, rather than `&guess`.

[10] `comments.html`
[11] `../std/string/struct.String.html`
[12] `../std/io/struct.Stdin.html`
[13] `../std/io/struct.Stdin.html#method.read_line`
[14] `method-syntax.html`
[15] `references-and-borrowing.html`

Why does `read_line()` take a mutable reference to a string? Its job is to take what the user types into standard input, and place that into a string. So it takes that string as an argument, and in order to add the input, it needs to be mutable.

But we're not quite done with this line of code, though. While it's a single line of text, it's only the first part of the single logical line of code:

```
        .expect("Failed to read line");
```

When you call a method with the `.foo()` syntax, you may introduce a newline and other whitespace. This helps you split up long lines. We *could* have done:

```
    io::stdin().read_line(&mut guess).expect("failed to read line");
```

But that gets hard to read. So we've split it up, three lines for three method calls. We already talked about `read_line()`, but what about `expect()`? Well, we already mentioned that `read_line()` puts what the user types into the `&mut String` we pass it. But it also returns a value: in this case, an `io::Result`[16]. Rust has a number of types named `Result` in its standard library: a generic `Result`[17], and then specific versions for sub-libraries, like `io::Result`.

The purpose of these `Result` types is to encode error handling information. Values of the `Result` type, like any type, have methods defined on them. In this case, `io::Result` has an `expect()` method[18] that takes a value it's called on, and if it isn't a successful one, `panic!`[19]s with a message you passed it. A `panic!` like this will cause our program to crash, displaying the message.

If we leave off calling these two methods, our program will compile, but we'll get a warning:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
src/main.rs:10:5: 10:39 warning: unused result which must be used,
#[warn(unused_must_use)] on by default
src/main.rs:10     io::stdin().read_line(&mut guess);
                   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Rust warns us that we haven't used the `Result` value. This warning comes from a special annotation that `io::Result` has. Rust is trying to tell you that you haven't handled a possible error. The right way to suppress the error is to actually write error handling. Luckily, if we just want to crash if there's a problem, we can use these two little methods. If we can recover from the error somehow, we'd do something else, but we'll save that for a future project.

There's just one line of this first example left:

```
    println!("You guessed: {}", guess);
}
```

[16] ../std/io/type.Result.html
[17] ../std/result/enum.Result.html
[18] ../std/option/enum.Option.html#method.expect
[19] error-handling.html

This prints out the string we saved our input in. The `{}`s are a placeholder, and so we pass it `guess` as an argument. If we had multiple `{}`s, we would pass multiple arguments:

```
let x = 5;
let y = 10;

println!("x and y: {} and {}", x, y);
```

Easy.

Anyway, that's the tour. We can run what we have with `cargo run`:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

All right! Our first part is done: we can get input from the keyboard, and then print it back out.

2.1.3 Generating a secret number

Next, we need to generate a secret number. Rust does not yet include random number functionality in its standard library. The Rust team does, however, provide a `rand` crate[20]. A 'crate' is a package of Rust code. We've been building a 'binary crate', which is an executable. `rand` is a 'library crate', which contains code that's intended to be used with other programs.

Using external crates is where Cargo really shines. Before we can write the code using `rand`, we need to modify our `Cargo.toml`. Open it up, and add these few lines at the bottom:

```
[dependencies]

rand="0.3.0"
```

The `[dependencies]` section of `Cargo.toml` is like the `[package]` section: everything that follows it is part of it, until the next section starts. Cargo uses the dependencies section to know what dependencies on external crates you have, and what versions you require. In this case, we've specified version `0.3.0`, which Cargo understands to be any release that's compatible with this specific version. Cargo understands Semantic Versioning[21], which is a standard for writing version numbers. If we wanted to use only `0.3.0` exactly, we could use `=0.3.0`. If we wanted to use the latest version we

[20] `https://crates.io/crates/rand`
[21] `http://semver.org`

could use `*`; We could use a range of versions. Cargo's documentation[22] contains more details.

Now, without changing any of our code, let's build our project:

```
$ cargo build
    Updating registry ‘https://github.com/rust-lang/crates.io-index‘
 Downloading rand v0.3.8
 Downloading libc v0.1.6
   Compiling libc v0.1.6
   Compiling rand v0.3.8
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
```

(You may see different versions, of course.)

Lots of new output! Now that we have an external dependency, Cargo fetches the latest versions of everything from the registry, which is a copy of data from Crates.io[23]. Crates.io is where people in the Rust ecosystem post their open source Rust projects for others to use.

After updating the registry, Cargo checks our `[dependencies]` and downloads any we don't have yet. In this case, while we only said we wanted to depend on `rand`, we've also grabbed a copy of `libc`. This is because `rand` depends on `libc` to work. After downloading them, it compiles them, and then compiles our project.

If we run `cargo build` again, we'll get different output:

```
$ cargo build
```

That's right, no output! Cargo knows that our project has been built, and that all of its dependencies are built, and so there's no reason to do all that stuff. With nothing to do, it simply exits. If we open up `src/main.rs` again, make a trivial change, and then save it again, we'll just see one line:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
```

So, we told Cargo we wanted any `0.3.x` version of `rand`, and so it fetched the latest version at the time this was written, `v0.3.8`. But what happens when next week, version `v0.3.9` comes out, with an important bugfix? While getting bugfixes is important, what if `0.3.9` contains a regression that breaks our code?

The answer to this problem is the `Cargo.lock` file you'll now find in your project directory. When you build your project for the first time, Cargo figures out all of the versions that fit your criteria, and then writes them to the `Cargo.lock` file. When you build your project in the future, Cargo will see that the `Cargo.lock` file exists, and then use that specific version rather than do all the work of figuring out versions again. This lets you have a repeatable build automatically. In other words, we'll stay at `0.3.8` until we explicitly upgrade, and so will anyone who we share our code with, thanks to the lock file.

[22] `http://doc.crates.io/crates-io.html`
[23] `https://crates.io`

What about when we *do* want to use `v0.3.9`? Cargo has another command, `update`, which says 'ignore the lock, figure out all the latest versions that fit what we've specified. If that works, write those versions out to the lock file'. But, by default, Cargo will only look for versions larger than `0.3.0` and smaller than `0.4.0`. If we want to move to `0.4.x`, we'd have to update the `Cargo.toml` directly. When we do, the next time we `cargo build`, Cargo will update the index and re-evaluate our `rand` requirements.

There's a lot more to say about Cargo[24] and its ecosystem[25], but for now, that's all we need to know. Cargo makes it really easy to re-use libraries, and so Rustaceans tend to write smaller projects which are assembled out of a number of sub-packages.

Let's get on to actually *using* `rand`. Here's our next step:

```
extern crate rand;

use std::io;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("failed to read line");

    println!("You guessed: {}", guess);
}
```

The first thing we've done is change the first line. It now says `extern crate rand`. Because we declared `rand` in our `[dependencies]`, we can use `extern crate` to let Rust know we'll be making use of it. This also does the equivalent of a `use rand;` as well, so we can make use of anything in the `rand` crate by prefixing it with `rand::`.

Next, we added another `use` line: `use rand::Rng`. We're going to use a method in a moment, and it requires that `Rng` be in scope to work. The basic idea is this: methods are defined on something called 'traits', and for the method to work, it needs the trait to be in scope. For more about the details, read the traits[26] section.

There are two other lines we added, in the middle:

```
    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);
```

[24] `http://doc.crates.io`
[25] `http://doc.crates.io/crates-io.html`
[26] `traits.html`

We use the `rand::thread_rng()` function to get a copy of the random number generator, which is local to the particular thread[27] of execution we're in. Because we `use rand::Rng`'d above, it has a `gen_range()` method available. This method takes two arguments, and generates a number between them. It's inclusive on the lower bound, but exclusive on the upper bound, so we need `1` and `101` to get a number ranging from one to a hundred.

The second line just prints out the secret number. This is useful while we're developing our program, so we can easily test it out. But we'll be deleting it for the final version. It's not much of a game if it prints out the answer when you start it up!

Try running our new program a few times:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4
$ cargo run
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

Great! Next up: let's compare our guess to the secret guess.

2.1.4 Comparing guesses

Now that we've got user input, let's compare our guess to the random guess. Here's our next step, though it doesn't quite compile yet:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);
```

[27] `concurrency.html`

```
    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("failed to read line");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}
```

A few new bits here. The first is another `use`. We bring a type called `std::cmp::Ordering` into scope. Then, five new lines at the bottom that use it:

```
match guess.cmp(&secret_number) {
    Ordering::Less    => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal   => println!("You win!"),
}
```

The `cmp()` method can be called on anything that can be compared, and it takes a reference to the thing you want to compare it to. It returns the `Ordering` type we used earlier. We use a `match`[28] statement to determine exactly what kind of `Ordering` it is. `Ordering` is an `enum`[29], short for 'enumeration', which looks like this:

```
enum Foo {
    Bar,
    Baz,
}
```

With this definition, anything of type `Foo` can be either a `Foo::Bar` or a `Foo::Baz`. We use the `::` to indicate the namespace for a particular `enum` variant.

The `Ordering`[30] `enum` has three possible variants: `Less`, `Equal`, and `Greater`. The `match` statement takes a value of a type, and lets you create an 'arm' for each possible value. Since we have three types of `Ordering`, we have three arms:

```
match guess.cmp(&secret_number) {
    Ordering::Less    => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal   => println!("You win!"),
}
```

[28] `match.html`
[29] `enums.html`
[30] `../std/cmp/enum.Ordering.html`

If it's `Less`, we print `Too small!`, if it's `Greater`, `Too big!`, and if `Equal`, `You win!`. `match` is really useful, and is used often in Rust.

I did mention that this won't quite compile yet, though. Let's try it:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
src/main.rs:28:21: 28:35 error: mismatched types:
 expected ‘&collections::string::String‘,
    found ‘&_‘
(expected struct ‘collections::string::String‘,
    found integral variable) [E0308]
src/main.rs:28     match guess.cmp(&secret_number) {
                                   ^~~~~~~~~~~~~~
error: aborting due to previous error
Could not compile ‘guessing_game‘.
```

Whew! This is a big error. The core of it is that we have 'mismatched types'. Rust has a strong, static type system. However, it also has type inference. When we wrote `let guess = String::new()`, Rust was able to infer that `guess` should be a `String`, and so it doesn't make us write out the type. And with our `secret_number`, there are a number of types which can have a value between one and a hundred: `i32`, a thirty-two-bit number, or `u32`, an unsigned thirty-two-bit number, or `i64`, a sixty-four-bit number or others. So far, that hasn't mattered, and so Rust defaults to an `i32`. However, here, Rust doesn't know how to compare the `guess` and the `secret_number`. They need to be the same type. Ultimately, we want to convert the `String` we read as input into a real number type, for comparison. We can do that with three more lines. Here's our new program:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("failed to read line");

    let guess: u32 = guess.trim().parse()
```

```
        .expect("Please type a number!");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}
```

The new three lines:

```
    let guess: u32 = guess.trim().parse()
        .expect("Please type a number!");
```

Wait a minute, I thought we already had a `guess`? We do, but Rust allows us to 'shadow' the previous `guess` with a new one. This is often used in this exact situation, where `guess` starts as a `String`, but we want to convert it to an `u32`. Shadowing lets us re-use the `guess` name, rather than forcing us to come up with two unique names like `guess_str` and `guess`, or something else.

We bind `guess` to an expression that looks like something we wrote earlier:

```
guess.trim().parse()
```

Here, `guess` refers to the old `guess`, the one that was a `String` with our input in it. The `trim()` method on `String`s will eliminate any white space at the beginning and end of our string. This is important, as we had to press the 'return' key to satisfy `read_line()`. This means that if we type `5` and hit return, `guess` looks like this: `5\n`. The `\n` represents 'newline', the enter key. `trim()` gets rid of this, leaving our string with just the `5`. The `parse()` method on strings[31] parses a string into some kind of number. Since it can parse a variety of numbers, we need to give Rust a hint as to the exact type of number we want. Hence, `let guess: u32`. The colon (`:`) after `guess` tells Rust we're going to annotate its type. `u32` is an unsigned, thirty-two bit integer. Rust has a number of built-in number types[32], but we've chosen `u32`. It's a good default choice for a small positive number.

Just like `read_line()`, our call to `parse()` could cause an error. What if our string contained `A%`? There'd be no way to convert that to a number. As such, we'll do the same thing we did with `read_line()`: use the `expect()` method to crash if there's an error.

Let's try our program out!

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
```

[31] `../std/primitive.str.html#method.parse`
[32] `primitive-types.html#numeric-types`

```
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

Nice! You can see I even added spaces before my guess, and it still figured out that I guessed 76. Run the program a few times, and verify that guessing the number works, as well as guessing a number too small.

Now we've got most of the game working, but we can only make one guess. Let's change that by adding loops!

2.1.5 Looping

The `loop` keyword gives us an infinite loop. Let's add that in:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("failed to read line");

        let guess: u32 = guess.trim().parse()
            .expect("Please type a number!");

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => println!("You win!"),
        }
```

```
    }
}
```

And try it out. But wait, didn't we just add an infinite loop? Yup. Remember our discussion about `parse()`? If we give a non-number answer, we'll `return` and quit. Observe:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread '<main>' panicked at 'Please type a number!'
```

Ha! `quit` actually quits. As does any other non-number input. Well, this is suboptimal to say the least. First, let's actually quit when you win the game:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();
```

```
        io::stdin().read_line(&mut guess)
            .expect("failed to read line");

        let guess: u32 = guess.trim().parse()
            .expect("Please type a number!");

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                break;
            }
        }
    }
}
```

By adding the `break` line after the `You win!`, we'll exit the loop when we win. Exiting the loop also means exiting the program, since it's the last thing in `main()`. We have just one more tweak to make: when someone inputs a non-number, we don't want to quit, we just want to ignore it. We can do that like this:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };
```

```
        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                break;
            }
        }
    }
}
```

These are the lines that changed:

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```

This is how you generally move from ‘crash on error’ to ‘actually handle the returned by `parse()` is an `enum` just like `Ordering`, but in this case, each variant has some data associated with it: `Ok` is a success, and `Err` is a failure. Each contains more information: the successfully parsed integer, or an error type. In this case, we `match` on `Ok(num)`, which sets the inner value of the `Ok` to the name `num`, and then we just return it on the right-hand side. In the `Err` case, we don’t care what kind of error it is, so we just use _ instead of a name. This ignores the error, and `continue` causes us to go to the next iteration of the `loop`.

Now we should be good! Let’s try:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running ‘target/guessing_game‘
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

Awesome! With one tiny last tweak, we have finished the guessing game. Can you think of what it is? That's right, we don't want to print out the secret number. It was good for testing, but it kind of ruins the game. Here's our final source:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                break;
            }
        }
    }
}
```

2.1.6 Complete!

At this point, you have successfully built the Guessing Game! Congratulations!

This first project showed you a lot: `let`, `match`, methods, associated functions, using external crates, and more. Our next project will show off even more.

2.2 Dining Philosophers

For our second project, let's look at a classic concurrency problem. It's called 'the dining philosophers'. It was originally conceived by Dijkstra in 1965, but we'll use a lightly adapted version from this paper[33] by Tony Hoare in 1985.

> In ancient times, a wealthy philanthropist endowed a College to accommodate five eminent philosophers. Each philosopher had a room in which they could engage in their professional activity of thinking; there was also a common dining room, furnished with a circular table, surrounded by five chairs, each labelled by the name of the philosopher who was to sit in it. They sat anticlockwise around the table. To the left of each philosopher there was laid a golden fork, and in the center stood a large bowl of spaghetti, which was constantly replenished. A philosopher was expected to spend most of their time thinking; but when they felt hungry, they went to the dining room, sat down in their own chair, picked up their own fork on their left, and plunged it into the spaghetti. But such is the tangled nature of spaghetti that a second fork is required to carry it to the mouth. The philosopher therefore had also to pick up the fork on their right. When they were finished they would put down both their forks, get up from their chair, and continue thinking. Of course, a fork can be used by only one philosopher at a time. If the other philosopher wants it, they just have to wait until the fork is available again.

This classic problem shows off a few different elements of concurrency. The reason is that it's actually slightly tricky to implement: a simple implementation can deadlock. For example, let's consider a simple algorithm that would solve this problem:

1. A philosopher picks up the fork on their left.
2. They then pick up the fork on their right.
3. They eat.
4. They return the forks.

Now, let's imagine this sequence of events:

1. Philosopher 1 begins the algorithm, picking up the fork on their left.
2. Philosopher 2 begins the algorithm, picking up the fork on their left.
3. Philosopher 3 begins the algorithm, picking up the fork on their left.
4. Philosopher 4 begins the algorithm, picking up the fork on their left.
5. Philosopher 5 begins the algorithm, picking up the fork on their left.
6. ... ? All the forks are taken, but nobody can eat!

There are different ways to solve this problem. We'll get to our solution in the tutorial itself. For now, let's get started and create a new project with `cargo`:

```
$ cd ~/projects
$ cargo new dining_philosophers --bin
$ cd dining_philosophers
```

[33] `http://www.usingcsp.com/cspbook.pdf`

Now we can start modeling the problem itself. We'll start with the philosophers in `src/main.rs`:

```
struct Philosopher {
    name: String,
}

impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }
}

fn main() {
    let p1 = Philosopher::new("Judith Butler");
    let p2 = Philosopher::new("Gilles Deleuze");
    let p3 = Philosopher::new("Karl Marx");
    let p4 = Philosopher::new("Emma Goldman");
    let p5 = Philosopher::new("Michel Foucault");
}
```

Here, we make a `struct`[34] to represent a philosopher. For now, a name is all we need. We choose the `String`[35] type for the name, rather than `&str`. Generally speaking, working with a type which owns its data is easier than working with one that uses references.

Let's continue:

```
# struct Philosopher {
#     name: String,
# }
impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }
}
```

This `impl` block lets us define things on `Philosopher` structs. In this case, we define an 'associated function' called `new`. The first line looks like this:

```
# struct Philosopher {
#     name: String,
# }
```

[34] `structs.html`
[35] `strings.html`

```
# impl Philosopher {
fn new(name: &str) -> Philosopher {
#         Philosopher {
#             name: name.to_string(),
#         }
#     }
# }
```

We take one argument, a `name`, of type `&str`. This is a reference to another string. It returns an instance of our `Philosopher` struct.

```
# struct Philosopher {
#     name: String,
# }
# impl Philosopher {
#     fn new(name: &str) -> Philosopher {
Philosopher {
    name: name.to_string(),
}
#     }
# }
```

This creates a new `Philosopher`, and sets its `name` to our `name` argument. Not just the argument itself, though, as we call `.to_string()` on it. This will create a copy of the string that our `&str` points to, and give us a new `String`, which is the type of the `name` field of `Philosopher`.

Why not accept a `String` directly? It's nicer to call. If we took a `String`, but our caller had a `&str`, they'd have to call this method themselves. The downside of this flexibility is that we *always* make a copy. For this small program, that's not particularly important, as we know we'll just be using short strings anyway.

One last thing you'll notice: we just define a `Philosopher`, and seemingly don't do anything with it. Rust is an 'expression based' language, which means that almost everything in Rust is an expression which returns a value. This is true of functions as well, the last expression is automatically returned. Since we create a new `Philosopher` as the last expression of this function, we end up returning it.

This name, `new()`, isn't anything special to Rust, but it is a convention for functions that create new instances of structs. Before we talk about why, let's look at `main()` again:

```
# struct Philosopher {
#     name: String,
# }
#
# impl Philosopher {
#     fn new(name: &str) -> Philosopher {
#         Philosopher {
#             name: name.to_string(),
#         }
```

```
#     }
# }
#
fn main() {
    let p1 = Philosopher::new("Judith Butler");
    let p2 = Philosopher::new("Gilles Deleuze");
    let p3 = Philosopher::new("Karl Marx");
    let p4 = Philosopher::new("Emma Goldman");
    let p5 = Philosopher::new("Michel Foucault");
}
```

Here, we create five variable bindings with five new philosophers. These are my favorite five, but you can substitute anyone you want. If we *didn't* define that `new()` function, it would look like this:

```
# struct Philosopher {
#     name: String,
# }
fn main() {
    let p1 = Philosopher { name: "Judith Butler".to_string() };
    let p2 = Philosopher { name: "Gilles Deleuze".to_string() };
    let p3 = Philosopher { name: "Karl Marx".to_string() };
    let p4 = Philosopher { name: "Emma Goldman".to_string() };
    let p5 = Philosopher { name: "Michel Foucault".to_string() };
}
```

That's much noisier. Using `new` has other advantages too, but even in this simple case, it ends up being nicer to use.

Now that we've got the basics in place, there's a number of ways that we can tackle the broader problem here. I like to start from the end first: let's set up a way for each philosopher to finish eating. As a tiny step, let's make a method, and then loop through all the philosophers, calling it:

```
struct Philosopher {
    name: String,
}

impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }

    fn eat(&self) {
        println!("{} is done eating.", self.name);
    }
}
```

```
fn main() {
    let philosophers = vec![
        Philosopher::new(”Judith Butler”),
        Philosopher::new(”Gilles Deleuze”),
        Philosopher::new(”Karl Marx”),
        Philosopher::new(”Emma Goldman”),
        Philosopher::new(”Michel Foucault”),
    ];

    for p in &philosophers {
        p.eat();
    }
}
```

Let’s look at `main()` first. Rather than have five individual variable bindings for our philosophers, we make a `Vec<T>` of them instead. `Vec<T>` is also called a ‘vector’, and it’s a growable array type. We then use a `for`[36] loop to iterate through the vector, getting a reference to each philosopher in turn.

In the body of the loop, we call `p.eat()`, which is defined above:

```
fn eat(&self) {
    println!(”{} is done eating.”, self.name);
}
```

In Rust, methods take an explicit `self` parameter. That’s why `eat()` is a method, but `new` is an associated function: `new()` has no `self`. For our first version of `eat()`, we just print out the name of the philosopher, and mention they’re done eating. Running this program should give you the following output:

```
Judith Butler is done eating.
Gilles Deleuze is done eating.
Karl Marx is done eating.
Emma Goldman is done eating.
Michel Foucault is done eating.
```

Easy enough, they’re all done! We haven’t actually implemented the real problem yet, though, so we’re not done yet!

Next, we want to make our philosophers not just finish eating, but actually eat. Here’s the next version:

```
use std::thread;
use std::time::Duration;

struct Philosopher {
    name: String,
}
```

[36] `loops.html#for`

```
impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }

    fn eat(&self) {
        println!("{} is eating.", self.name);

        thread::sleep(Duration::from_millis(1000));

        println!("{} is done eating.", self.name);
    }
}

fn main() {
    let philosophers = vec![
        Philosopher::new("Judith Butler"),
        Philosopher::new("Gilles Deleuze"),
        Philosopher::new("Karl Marx"),
        Philosopher::new("Emma Goldman"),
        Philosopher::new("Michel Foucault"),
    ];

    for p in &philosophers {
        p.eat();
    }
}
```

Just a few changes. Let’s break it down.

```
use std::thread;
```

`use` brings names into scope. We’re going to start using the `thread` module from the standard library, and so we need to `use` it.

```
    fn eat(&self) {
        println!("{} is eating.", self.name);

        thread::sleep(Duration::from_millis(1000));

        println!("{} is done eating.", self.name);
    }
```

We now print out two messages, with a `sleep` in the middle. This will simulate the time it takes a philosopher to eat.

If you run this program, you should see each philosopher eat in turn:

```
Judith Butler is eating.
Judith Butler is done eating.
Gilles Deleuze is eating.
Gilles Deleuze is done eating.
Karl Marx is eating.
Karl Marx is done eating.
Emma Goldman is eating.
Emma Goldman is done eating.
Michel Foucault is eating.
Michel Foucault is done eating.
```

Excellent! We’re getting there. There’s just one problem: we aren’t actually operating in a concurrent fashion, which is a core part of the problem!

To make our philosophers eat concurrently, we need to make a small change. Here’s the next iteration:

```
use std::thread;
use std::time::Duration;

struct Philosopher {
    name: String,
}

impl Philosopher {
    fn new(name: &str) -> Philosopher {
        Philosopher {
            name: name.to_string(),
        }
    }

    fn eat(&self) {
        println!(”{} is eating.”, self.name);

        thread::sleep(Duration::from_millis(1000));

        println!(”{} is done eating.”, self.name);
    }
}

fn main() {
    let philosophers = vec![
        Philosopher::new(”Judith Butler”),
        Philosopher::new(”Gilles Deleuze”),
        Philosopher::new(”Karl Marx”),
        Philosopher::new(”Emma Goldman”),
        Philosopher::new(”Michel Foucault”),
    ];

    let handles: Vec<_> = philosophers.into_iter().map(|p| {
```

```
        thread::spawn(move || {
            p.eat();
        })
    }).collect();

    for h in handles {
        h.join().unwrap();
    }
}
```

All we've done is change the loop in `main()`, and added a second one! Here's the first change:

```
let handles: Vec<_> = philosophers.into_iter().map(|p| {
    thread::spawn(move || {
        p.eat();
    })
}).collect();
```

While this is only five lines, they're a dense five. Let's break it down.

```
let handles: Vec<_> =
```

We introduce a new binding, called `handles`. We've given it this name because we are going to make some new threads, and that will return some handles to those threads that let us control their operation. We need to explicitly annotate the type here, though, due to an issue we'll talk about later. The _ is a type placeholder. We're saying "`handles` is a vector of something, but you can figure out what that something is, Rust."

```
philosophers.into_iter().map(|p| {
```

We take our list of philosophers and call `into_iter()` on it. This creates an iterator that takes ownership of each philosopher. We need to do this to pass them to our threads. We take that iterator and call `map` on it, which takes a closure as an argument and calls that closure on each element in turn.

```
    thread::spawn(move || {
        p.eat();
    })
```

Here's where the concurrency happens. The `thread::spawn` function takes a closure as an argument and executes that closure in a new thread. This closure needs an extra annotation, `move`, to indicate that the closure is going to take ownership of the values it's capturing. Primarily, the `p` variable of the `map` function.

Inside the thread, all we do is call `eat()` on `p`. Also note that the call to `thread::spawn` lacks a trailing semicolon, making this an expression. This distinction is important, yielding the correct return value. For more details, read Expressions vs. Statements[37].

[37] `functions.html#expressions-vs-statements`

```
}).collect();
```

Finally, we take the result of all those `map` calls and collect them up. `collect()` will make them into a collection of some kind, which is why we needed to annotate the return type: we want a `Vec<T>`. The elements are the return values of the `thread::spawn` calls, which are handles to those threads. Whew!

```
for h in handles {
    h.join().unwrap();
}
```

At the end of `main()`, we loop through the handles and call `join()` on them, which blocks execution until the thread has completed execution. This ensures that the threads complete their work before the program exits.

If you run this program, you'll see that the philosophers eat out of order! We have multi-threading!

```
Judith Butler is eating.
Gilles Deleuze is eating.
Karl Marx is eating.
Emma Goldman is eating.
Michel Foucault is eating.
Judith Butler is done eating.
Gilles Deleuze is done eating.
Karl Marx is done eating.
Emma Goldman is done eating.
Michel Foucault is done eating.
```

But what about the forks? We haven't modeled them at all yet.

To do that, let's make a new `struct`:

```
use std::sync::Mutex;

struct Table {
    forks: Vec<Mutex<()>>,
}
```

This `Table` has a vector of `Mutex`es. A mutex is a way to control concurrency: only one thread can access the contents at once. This is exactly the property we need with our forks. We use an empty tuple, `()`, inside the mutex, since we're not actually going to use the value, just hold onto it.

Let's modify the program to use the `Table`:

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

struct Philosopher {
```

```
    name: String,
    left: usize,
    right: usize,
}

impl Philosopher {
    fn new(name: &str, left: usize, right: usize) -> Philosopher {
        Philosopher {
            name: name.to_string(),
            left: left,
            right: right,
        }
    }

    fn eat(&self, table: &Table) {
        let _left = table.forks[self.left].lock().unwrap();
        thread::sleep(Duration::from_millis(150));
        let _right = table.forks[self.right].lock().unwrap();

        println!("{} is eating.", self.name);

        thread::sleep(Duration::from_millis(1000));

        println!("{} is done eating.", self.name);
    }
}

struct Table {
    forks: Vec<Mutex<()>>,
}

fn main() {
    let table = Arc::new(Table { forks: vec![
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
    ]});

    let philosophers = vec![
        Philosopher::new("Judith Butler", 0, 1),
        Philosopher::new("Gilles Deleuze", 1, 2),
        Philosopher::new("Karl Marx", 2, 3),
        Philosopher::new("Emma Goldman", 3, 4),
        Philosopher::new("Michel Foucault", 0, 4),
    ];

    let handles: Vec<_> = philosophers.into_iter().map(|p| {
```

```
        let table = table.clone();

        thread::spawn(move || {
            p.eat(&table);
        })
    }).collect();

    for h in handles {
        h.join().unwrap();
    }
}
```

Lots of changes! However, with this iteration, we've got a working program. Let's go over the details:

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

We're going to use another structure from the `std::sync` package: `Arc<T>`. We'll talk more about it when we use it.

```
struct Philosopher {
    name: String,
    left: usize,
    right: usize,
}
```

We need to add two more fields to our `Philosopher`. Each philosopher is going to have two forks: the one on their left, and the one on their right. We'll use the `usize` type to indicate them, as it's the type that you index vectors with. These two values will be the indexes into the `forks` our `Table` has.

```
fn new(name: &str, left: usize, right: usize) -> Philosopher {
    Philosopher {
        name: name.to_string(),
        left: left,
        right: right,
    }
}
```

We now need to construct those `left` and `right` values, so we add them to `new()`.

```
fn eat(&self, table: &Table) {
    let _left = table.forks[self.left].lock().unwrap();
    thread::sleep(Duration::from_millis(150));
    let _right = table.forks[self.right].lock().unwrap();

    println!("{} is eating.", self.name);

    thread::sleep(Duration::from_millis(1000));
```

```
    println!("{} is done eating.", self.name);
}
```

We have three new lines. We've added an argument, `table`. We access the `Table`'s list of forks, and then use `self.left` and `self.right` to access the fork at that particular index. That gives us access to the `Mutex` at that index, and we call `lock()` on it. If the mutex is currently being accessed by someone else, we'll block until it becomes available. We have also a call to `thread::sleep` between the moment the first fork is picked and the moment the second forked is picked, as the process of picking up the fork is not immediate.

The call to `lock()` might fail, and if it does, we want to crash. In this case, the error that could happen is that the mutex is 'poisoned'[38], which is what happens when the thread panics while the lock is held. Since this shouldn't happen, we just use `unwrap()`.

One other odd thing about these lines: we've named the results `_left` and `_right`. What's up with that underscore? Well, we aren't planning on *using* the value inside the lock. We just want to acquire it. As such, Rust will warn us that we never use the value. By using the underscore, we tell Rust that this is what we intended, and it won't throw a warning.

What about releasing the lock? Well, that will happen when `_left` and `_right` go out of scope, automatically.

```
    let table = Arc::new(Table { forks: vec![
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
    ]});
```

Next, in `main()`, we make a new `Table` and wrap it in an `Arc<T>`. 'arc' stands for 'atomic reference count', and we need that to share our `Table` across multiple threads. As we share it, the reference count will go up, and when each thread ends, it will go back down.

```
let philosophers = vec![
    Philosopher::new("Judith Butler", 0, 1),
    Philosopher::new("Gilles Deleuze", 1, 2),
    Philosopher::new("Karl Marx", 2, 3),
    Philosopher::new("Emma Goldman", 3, 4),
    Philosopher::new("Michel Foucault", 0, 4),
];
```

We need to pass in our `left` and `right` values to the constructors for our `Philosophers`. But there's one more detail here, and it's *very* important. If you look at the pattern, it's

[38] `../std/sync/struct.Mutex.html#poisoning`

all consistent until the very end. Monsieur Foucault should have `4, 0` as arguments, but instead, has `0, 4`. This is what prevents deadlock, actually: one of our philosophers is left handed! This is one way to solve the problem, and in my opinion, it's the simplest. If you change the order of the parameters, you will be able to observe the deadlock taking place.

```
let handles: Vec<_> = philosophers.into_iter().map(|p| {
    let table = table.clone();

    thread::spawn(move || {
        p.eat(&table);
    })
}).collect();
```

Finally, inside of our `map()`/`collect()` loop, we call `table.clone()`. The `clone()` method on `Arc<T>` is what bumps up the reference count, and when it goes out of scope, it decrements the count. This is needed so that we know how many references to `table` exist across our threads. If we didn't have a count, we wouldn't know how to deallocate it.

You'll notice we can introduce a new binding to `table` here, and it will shadow the old one. This is often used so that you don't need to come up with two unique names.

With this, our program works! Only two philosophers can eat at any one time, and so you'll get some output like this:

```
Gilles Deleuze is eating.
Emma Goldman is eating.
Emma Goldman is done eating.
Gilles Deleuze is done eating.
Judith Butler is eating.
Karl Marx is eating.
Judith Butler is done eating.
Michel Foucault is eating.
Karl Marx is done eating.
Michel Foucault is done eating.
```

Congrats! You've implemented a classic concurrency problem in Rust.

2.3 Rust Inside Other Languages

For our third project, we're going to choose something that shows off one of Rust's greatest strengths: a lack of a substantial runtime.

As organizations grow, they increasingly rely on a multitude of programming languages. Different programming languages have different strengths and weaknesses, and a polyglot stack lets you use a particular language where its strengths make sense and a different one where it's weak.

A very common area where many programming languages are weak is in runtime performance of programs. Often, using a language that is slower, but offers greater

programmer productivity, is a worthwhile trade-off. To help mitigate this, they provide a way to write some of your system in C and then call that C code as though it were written in the higher-level language. This is called a 'foreign function interface', often shortened to 'FFI'.

Rust has support for FFI in both directions: it can call into C code easily, but crucially, it can also be called *into* as easily as C. Combined with Rust's lack of a garbage collector and low runtime requirements, this makes Rust a great candidate to embed inside of other languages when you need that extra oomph.

There is a whole chapter devoted to FFI[39] and its specifics elsewhere in the book, but in this chapter, we'll examine this particular use-case of FFI, with examples in Ruby, Python, and JavaScript.

2.3.1 The problem

There are many different projects we could choose here, but we're going to pick an example where Rust has a clear advantage over many other languages: numeric computing and threading.

Many languages, for the sake of consistency, place numbers on the heap, rather than on the stack. Especially in languages that focus on object-oriented programming and use garbage collection, heap allocation is the default. Sometimes optimizations can stack allocate particular numbers, but rather than relying on an optimizer to do its job, we may want to ensure that we're always using primitive number types rather than some sort of object type.

Second, many languages have a 'global interpreter lock' (GIL), which limits concurrency in many situations. This is done in the name of safety, which is a positive effect, but it limits the amount of work that can be done at the same time, which is a big negative.

To emphasize these two aspects, we're going to create a little project that uses these two aspects heavily. Since the focus of the example is to embed Rust into other languages, rather than the problem itself, we'll just use a toy example:

> Start ten threads. Inside each thread, count from one to five million. After all ten threads are finished, print out 'done!'.

I chose five million based on my particular computer. Here's an example of this code in Ruby:

```
threads = []

10.times do
  threads << Thread.new do
    count = 0

    5_000_000.times do
      count += 1
```

[39] ffi.html

```
    end

    count
  end
end

threads.each do |t|
  puts "Thread finished with count=#{t.value}"
end
puts "done!"
```

Try running this example, and choose a number that runs for a few seconds. Depending on your computer's hardware, you may have to increase or decrease the number.

On my system, running this program takes `2.156` seconds. And, if I use some sort of process monitoring tool, like `top`, I can see that it only uses one core on my machine. That's the GIL kicking in.

While it's true that this is a synthetic program, one can imagine many problems that are similar to this in the real world. For our purposes, spinning up a few busy threads represents some sort of parallel, expensive computation.

2.3.2 A Rust library

Let's rewrite this problem in Rust. First, let's make a new project with Cargo:

```
$ cargo new embed
$ cd embed
```

This program is fairly easy to write in Rust:

```
use std::thread;

fn process() {
    let handles: Vec<_> = (0..10).map(|_| {
        thread::spawn(|| {
            let mut x = 0;
            for _ in 0..5_000_000 {
                x += 1
            }
            x
        })
    }).collect();

    for h in handles {
        println!("Thread finished with count={}",
        h.join().map_err(|_| "Could not join a thread!").unwrap());
    }
}
```

Some of this should look familiar from previous examples. We spin up ten threads, collecting them into a `handles` vector. Inside of each thread, we loop five million times, and add one to `x` each time. Finally, we join on each thread.

Right now, however, this is a Rust library, and it doesn't expose anything that's callable from C. If we tried to hook this up to another language right now, it wouldn't work. We only need to make two small changes to fix this, though. The first is to modify the beginning of our code:

```
#[no_mangle]
pub extern fn process() {
```

We have to add a new attribute, `no_mangle`. When you create a Rust library, it changes the name of the function in the compiled output. The reasons for this are outside the scope of this tutorial, but in order for other languages to know how to call the function, we can't do that. This attribute turns that behavior off.

The other change is the `pub extern`. The `pub` means that this function should be callable from outside of this module, and the `extern` says that it should be able to be called from C. That's it! Not a whole lot of change.

The second thing we need to do is to change a setting in our `Cargo.toml`. Add this at the bottom:

```
[lib]
name = "embed"
crate-type = ["dylib"]
```

This tells Rust that we want to compile our library into a standard dynamic library. By default, Rust compiles an 'rlib', a Rust-specific format.

Let's build the project now:

```
$ cargo build --release
   Compiling embed v0.1.0 (file:///home/steve/src/embed)
```

We've chosen `cargo build --release`, which builds with optimizations on. We want this to be as fast as possible! You can find the output of the library in `target/release`:

```
$ ls target/release/
build  deps  examples  libembed.so  native
```

That `libembed.so` is our 'shared object' library. We can use this file just like any shared object library written in C! As an aside, this may be `embed.dll` (Microsoft Windows) or `libembed.dylib` (Mac OS X), depending on your operating system.

Now that we've got our Rust library built, let's use it from our Ruby.

2.3.3 Ruby

Open up an `embed.rb` file inside of our project, and do this:

```
require 'ffi'

module Hello
  extend FFI::Library
  ffi_lib 'target/release/libembed.so'
  attach_function :process, [], :void
end

Hello.process

puts 'done!'
```

Before we can run this, we need to install the `ffi` gem:

```
$ gem install ffi # this may need sudo
Fetching: ffi-1.9.8.gem (100%)
Building native extensions.  This could take a while...
Successfully installed ffi-1.9.8
Parsing documentation for ffi-1.9.8
Installing ri documentation for ffi-1.9.8
Done installing documentation for ffi after 0 seconds
1 gem installed
```

And finally, we can try running it:

```
$ ruby embed.rb
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
done!
done!
$
```

Whoa, that was fast! On my system, this took `0.086` seconds, rather than the two seconds the pure Ruby version took. Let's break down this Ruby code:

```
require 'ffi'
```

We first need to require the `ffi` gem. This lets us interface with our Rust library like a C library.

```
module Hello
  extend FFI::Library
  ffi_lib 'target/release/libembed.so'
```

The `Hello` module is used to attach the native functions from the shared library. Inside, we `extend` the necessary `FFI::Library` module and then call `ffi_lib` to load up our shared object library. We just pass it the path that our library is stored, which, as we saw before, is `target/release/libembed.so`.

```
attach_function :process, [], :void
```

The `attach_function` method is provided by the FFI gem. It's what connects our `process()` function in Rust to a Ruby function of the same name. Since `process()` takes no arguments, the second parameter is an empty array, and since it returns nothing, we pass `:void` as the final argument.

```
Hello.process
```

This is the actual call into Rust. The combination of our `module` and the call to `attach_function` sets this all up. It looks like a Ruby function but is actually Rust!

```
puts 'done!'
```

Finally, as per our project's requirements, we print out `done!`.

That's it! As we've seen, bridging between the two languages is really easy, and buys us a lot of performance.

Next, let's try Python!

2.3.4 Python

Create an `embed.py` file in this directory, and put this in it:

```
from ctypes import cdll

lib = cdll.LoadLibrary("target/release/libembed.so")

lib.process()

print("done!")
```

Even easier! We use `cdll` from the `ctypes` module. A quick call to `LoadLibrary` later, and we can call `process()`.

On my system, this takes `0.017` seconds. Speedy!

2.3.5 Node.js

Node isn't a language, but it's currently the dominant implementation of server-side JavaScript.

In order to do FFI with Node, we first need to install the library:

```
$ npm install ffi
```

After that installs, we can use it:

```
var ffi = require('ffi');

var lib = ffi.Library('target/release/libembed', {
  'process': ['void', []]
});

lib.process();

console.log("done!");
```

It looks more like the Ruby example than the Python example. We use the `ffi` module to get access to `ffi.Library()`, which loads up our shared object. We need to annotate the return type and argument types of the function, which are `void` for return and an empty array to signify no arguments. From there, we just call it and print the result.

On my system, this takes a quick `0.092` seconds.

2.3.6 Conclusion

As you can see, the basics of doing this are *very* easy. Of course, there's a lot more that we could do here. Check out the FFI[40] chapter for more details.

[40] `ffi.html`

Chapter 3

Effective Rust

So you've learned how to write some Rust code. But there's a difference between writing *any* Rust code and writing *good* Rust code.

This section consists of relatively independent tutorials which show you how to take your Rust to the next level. Common patterns and standard library features will be introduced. Read these sections in any order of your choosing.

3.1 The Stack and the Heap

As a systems language, Rust operates at a low level. If you're coming from a high-level language, there are some aspects of systems programming that you may not be familiar with. The most important one is how memory works, with a stack and a heap. If you're familiar with how C-like languages use stack allocation, this chapter will be a refresher. If you're not, you'll learn about this more general concept, but with a Rust-y focus.

As with most things, when learning about them, we'll use a simplified model to start. This lets you get a handle on the basics, without getting bogged down with details which are, for now, irrelevant. The examples we'll use aren't 100% accurate, but are representative for the level we're trying to learn at right now. Once you have the basics down, learning more about how allocators are implemented, virtual memory, and other advanced topics will reveal the leaks in this particular abstraction.

3.1.1 Memory management

These two terms are about memory management. The stack and the heap are abstractions that help you determine when to allocate and deallocate memory.

Here's a high-level comparison:

The stack is very fast, and is where memory is allocated in Rust by default. But the allocation is local to a function call, and is limited in size. The heap, on the other hand, is slower, and is explicitly allocated by your program. But it's effectively unlimited in size, and is globally accessible.

3.1.2 The Stack

Let's talk about this Rust program:

```
fn main() {
    let x = 42;
}
```

This program has one variable binding, `x`. This memory needs to be allocated from somewhere. Rust 'stack allocates' by default, which means that basic values 'go on the stack'. What does that mean?

Well, when a function gets called, some memory gets allocated for all of its local variables and some other information. This is called a 'stack frame', and for the purpose of this tutorial, we're going to ignore the extra information and just consider the local variables we're allocating. So in this case, when `main()` is run, we'll allocate a single 32-bit integer for our stack frame. This is automatically handled for you, as you can see; we didn't have to write any special Rust code or anything.

When the function exits, its stack frame gets deallocated. This happens automatically as well.

That's all there is for this simple program. The key thing to understand here is that stack allocation is very, very fast. Since we know all the local variables we have ahead of time, we can grab the memory all at once. And since we'll throw them all away at the same time as well, we can get rid of it very fast too.

The downside is that we can't keep values around if we need them for longer than a single function. We also haven't talked about what the word, 'stack', means. To do that, we need a slightly more complicated example:

```
fn foo() {
    let y = 5;
    let z = 100;
}

fn main() {
    let x = 42;

    foo();
}
```

This program has three variables total: two in `foo()`, one in `main()`. Just as before, when `main()` is called, a single integer is allocated for its stack frame. But before we can show what happens when `foo()` is called, we need to visualize what's going on with memory. Your operating system presents a view of memory to your program that's pretty simple: a huge list of addresses, from 0 to a large number, representing how much RAM your computer has. For example, if you have a gigabyte of RAM, your addresses go from `0` to `1,073,741,823`. That number comes from 230, the number of bytes in a gigabyte. [1]

[1] 'Gigabyte' can mean two things: 10^9, or 2^30. The SI standard resolved this by stating that 'gigabyte' is 10^9, and 'gibibyte' is 2^30. However, very few people use this terminology, and rely on context to differentiate. We follow in that tradition here.

This memory is kind of like a giant array: addresses start at zero and go up to the final number. So here's a diagram of our first stack frame:

Address	Name	Value
0	x	42

We've got `x` located at address `0`, with the value `42`.

When `foo()` is called, a new stack frame is allocated:

Address	Name	Value
2	z	100
1	y	5
0	x	42

Because `0` was taken by the first frame, `1` and `2` are used for `foo()`'s stack frame. It grows upward, the more functions we call.

There are some important things we have to take note of here. The numbers 0, 1, and 2 are all solely for illustrative purposes, and bear no relationship to the actual numbers the computer will actually use. In particular, the series of addresses are in reality going to be separated by some number of bytes that separate each address, and that separation may even exceed the size of the value being stored.

After `foo()` is over, its frame is deallocated:

Address	Name	Value
0	x	42

And then, after `main()`, even this last value goes away. Easy!

It's called a 'stack' because it works like a stack of dinner plates: the first plate you put down is the last plate to pick back up. Stacks are sometimes called 'last in, first out queues' for this reason, as the last value you put on the stack is the first one you retrieve from it.

Let's try a three-deep example:

```
fn bar() {
    let i = 6;
}

fn foo() {
    let a = 5;
    let b = 100;
    let c = 1;

    bar();
}
```

```
fn main() {
    let x = 42;

    foo();
}
```

Okay, first, we call `main()`:

Address	Name	Value
0	x	42

Next up, `main()` calls `foo()`:

Address	Name	Value
3	c	1
2	b	100
1	a	5
0	x	42

And then `foo()` calls `bar()`:

Address	Name	Value
4	i	6
3	c	1
2	b	100
1	a	5
0	x	42

Whew! Our stack is growing tall.

After `bar()` is over, its frame is deallocated, leaving just `foo()` and `main()`:

Address	Name	Value
3	c	1
2	b	100
1	a	5
0	x	42

And then `foo()` ends, leaving just `main()`:

Address	Name	Value
0	x	42

And then we're done. Getting the hang of it? It's like piling up dishes: you add to the top, you take away from the top.

3.1.3 The Heap

Now, this works pretty well, but not everything can work like this. Sometimes, you need to pass some memory between different functions, or keep it alive for longer than a single function's execution. For this, we can use the heap.

In Rust, you can allocate memory on the heap with the `Box<T>` type[2]. Here's an example:

```
fn main() {
    let x = Box::new(5);
    let y = 42;
}
```

Here's what happens in memory when `main()` is called:

Address	Name	Value
1	y	42
0	x	??????

We allocate space for two variables on the stack. `y` is `42`, as it always has been, but what about `x`? Well, `x` is a `Box<i32>`, and boxes allocate memory on the heap. The actual value of the box is a structure which has a pointer to 'the heap'. When we start executing the function, and `Box::new()` is called, it allocates some memory for the heap, and puts `5` there. The memory now looks like this:

Address	Name	Value
(230) - 1		5
...	...	...
1	y	42
0	x	→ (230) - 1

We have (230) - 1 addresses in our hypothetical computer with 1GB of RAM. And since our stack grows from zero, the easiest place to allocate memory is from the other end. So our first value is at the highest place in memory. And the value of the struct at `x` has a raw pointer[3] to the place we've allocated on the heap, so the value of `x` is (230) - 1, the memory location we've asked for.

We haven't really talked too much about what it actually means to allocate and deallocate memory in these contexts. Getting into very deep detail is out of the scope of this tutorial, but what's important to point out here is that the heap isn't just a stack that grows from the opposite end. We'll have an example of this later in the book, but

[2] `../std/boxed/index.html`
[3] `raw-pointers.html`

because the heap can be allocated and freed in any order, it can end up with 'holes'. Here's a diagram of the memory layout of a program which has been running for a while now:

Address	Name	Value
(230) - 1		5
(230) - 2		
(230) - 3		
(230) - 4		42
...	...	...
3	y	→ (230) - 4
2	y	42
1	y	42
0	x	→ (230) - 1

In this case, we've allocated four things on the heap, but deallocated two of them. There's a gap between (230) - 1 and (230) - 4 which isn't currently being used. The specific details of how and why this happens depends on what kind of strategy you use to manage the heap. Different programs can use different 'memory allocators', which are libraries that manage this for you. Rust programs use jemalloc[4] for this purpose.

Anyway, back to our example. Since this memory is on the heap, it can stay alive longer than the function which allocates the box. In this case, however, it doesn't.[5] When the function is over, we need to free the stack frame for `main()`. `Box<T>`, though, has a trick up its sleeve: Drop[6]. The implementation of `Drop` for `Box` deallocates the memory that was allocated when it was created. Great! So when `x` goes away, it first frees the memory allocated on the heap:

Address	Name	Value
1	y	42
0	x	??????

And then the stack frame goes away, freeing all of our memory.

3.1.4 Arguments and borrowing

We've got some basic examples with the stack and the heap going, but what about function arguments and borrowing? Here's a small Rust program:

```
fn foo(i: &i32) {
    let z = 42;
}
```

[4] `http://www.canonware.com/jemalloc/`

[5] We can make the memory live longer by transferring ownership, sometimes called 'moving out of the box'. More complex examples will be covered later.

[6] `drop.html`

```
fn main() {
    let x = 5;
    let y = &x;

    foo(y);
}
```

When we enter `main()`, memory looks like this:

Address	Name	Value
1	y	→ 0
0	x	5

`x` is a plain old `5`, and `y` is a reference to `x`. So its value is the memory location that `x` lives at, which in this case is `0`.

What about when we call `foo()`, passing `y` as an argument?

Address	Name	Value
3	z	42
2	i	→ 0
1	y	→ 0
0	x	5

Stack frames aren't just for local bindings, they're for arguments too. So in this case, we need to have both `i`, our argument, and `z`, our local variable binding. `i` is a copy of the argument, `y`. Since `y`'s value is `0`, so is `i`'s.

This is one reason why borrowing a variable doesn't deallocate any memory: the value of a reference is just a pointer to a memory location. If we got rid of the underlying memory, things wouldn't work very well.

3.1.5 A complex example

Okay, let's go through this complex program step-by-step:

```
fn foo(x: &i32) {
    let y = 10;
    let z = &y;

    baz(z);
    bar(x, z);
}

fn bar(a: &i32, b: &i32) {
    let c = 5;
```

```
    let d = Box::new(5);
    let e = &d;

    baz(e);
}

fn baz(f: &i32) {
    let g = 100;
}

fn main() {
    let h = 3;
    let i = Box::new(20);
    let j = &h;

    foo(j);
}
```

First, we call `main()`:

Address	Name	Value
(230) - 1		20
...	...	...
2	j	→ 0
1	i	→ (230) - 1
0	h	3

We allocate memory for `j`, `i`, and `h`. `i` is on the heap, and so has a value pointing there.

Next, at the end of `main()`, `foo()` gets called:

Address	Name	Value
(230) - 1		20
...	...	...
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (230) - 1
0	h	3

Space gets allocated for `x`, `y`, and `z`. The argument `x` has the same value as `j`, since that's what we passed it in. It's a pointer to the `0` address, since `j` points at `h`.

Next, `foo()` calls `baz()`, passing `z`:

Address	Name	Value
(230) - 1		20
...	...	...
7	g	100
6	f	→ 4
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (230) - 1
0	h	3

We've allocated memory for `f` and `g`. `baz()` is very short, so when it's over, we get rid of its stack frame:

Address	Name	Value
(230) - 1		20
...	...	...
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (230) - 1
0	h	3

Next, `foo()` calls `bar()` with `x` and `z`:

Address	Name	Value
(230) - 1		20
(230) - 2		5
...	...	...
10	e	→ 9
9	d	→ (230) - 2
8	c	5
7	b	→ 4
6	a	→ 0
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (230) - 1
0	h	3

We end up allocating another value on the heap, and so we have to subtract one from (230) - 1. It's easier to just write that than `1,073,741,822`. In any case, we set up the

variables as usual.
At the end of `bar()`, it calls `baz()`:

Address	Name	Value
(230) - 1		20
(230) - 2		5
...	...	...
12	g	100
11	f	→ 9
10	e	→ 9
9	d	→ (230) - 2
8	c	5
7	b	→ 4
6	a	→ 0
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (230) - 1
0	h	3

With this, we're at our deepest point! Whew! Congrats for following along this far.
After `baz()` is over, we get rid of `f` and `g`:

Address	Name	Value
(230) - 1		20
(230) - 2		5
...	...	...
10	e	→ 9
9	d	→ (230) - 2
8	c	5
7	b	→ 4
6	a	→ 0
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (230) - 1
0	h	3

Next, we return from `bar()`. `d` in this case is a `Box<T>`, so it also frees what it points to: (230) - 2.

Address	Name	Value
(230) - 1		20

Address	Name	Value
...	...	...
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (230) - 1
0	h	3

And after that, `foo()` returns:

Address	Name	Value
(230) - 1		20
...	...	...
2	j	→ 0
1	i	→ (230) - 1
0	h	3

And then, finally, `main()`, which cleans the rest up. When `i` is `Drop`ped, it will clean up the last of the heap too.

3.1.6 What do other languages do?

Most languages with a garbage collector heap-allocate by default. This means that every value is boxed. There are a number of reasons why this is done, but they're out of scope for this tutorial. There are some possible optimizations that don't make it true 100% of the time, too. Rather than relying on the stack and `Drop` to clean up memory, the garbage collector deals with the heap instead.

3.1.7 Which to use?

So if the stack is faster and easier to manage, why do we need the heap? A big reason is that Stack-allocation alone means you only have LIFO semantics for reclaiming storage. Heap-allocation is strictly more general, allowing storage to be taken from and returned to the pool in arbitrary order, but at a complexity cost.

Generally, you should prefer stack allocation, and so, Rust stack-allocates by default. The LIFO model of the stack is simpler, at a fundamental level. This has two big impacts: runtime efficiency and semantic impact.

Runtime Efficiency

Managing the memory for the stack is trivial: The machine just increments or decrements a single value, the so-called "stack pointer". Managing memory for the heap is non-trivial: heap-allocated memory is freed at arbitrary points, and each block of

heap-allocated memory can be of arbitrary size, the memory manager must generally work much harder to identify memory for reuse.

If you'd like to dive into this topic in greater detail, this paper[7] is a great introduction.

Semantic impact

Stack-allocation impacts the Rust language itself, and thus the developer's mental model. The LIFO semantics is what drives how the Rust language handles automatic memory management. Even the deallocation of a uniquely-owned heap-allocated box can be driven by the stack-based LIFO semantics, as discussed throughout this chapter. The flexibility (i.e. expressiveness) of non LIFO-semantics means that in general the compiler cannot automatically infer at compile-time where memory should be freed; it has to rely on dynamic protocols, potentially from outside the language itself, to drive deallocation (reference counting, as used by `Rc<T>` and `Arc<T>`, is one example of this).

When taken to the extreme, the increased expressive power of heap allocation comes at the cost of either significant runtime support (e.g. in the form of a garbage collector) or significant programmer effort (in the form of explicit memory management calls that require verification not provided by the Rust compiler).

3.2 Testing

> Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.
>
> Edsger W. Dijkstra, "The Humble Programmer" (1972)

Let's talk about how to test Rust code. What we will not be talking about is the right way to test Rust code. There are many schools of thought regarding the right and wrong way to write tests. All of these approaches use the same basic tools, and so we'll show you the syntax for using them.

3.2.1 The `test` attribute

At its simplest, a test in Rust is a function that's annotated with the `test` attribute. Let's make a new project with Cargo called `adder`:

```
$ cargo new adder
$ cd adder
```

Cargo will automatically generate a simple test when you make a new project. Here's the contents of `src/lib.rs`:

```
#[test]
fn it_works() {
}
```

[7] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.143.4688

Note the `#[test]`. This attribute indicates that this is a test function. It currently has no body. That's good enough to pass! We can run the tests with `cargo test`:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

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

   Doc-tests adder

running 0 tests

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

Cargo compiled and ran our tests. There are two sets of output here: one for the test we wrote, and another for documentation tests. We'll talk about those later. For now, see this line:

```
test it_works ... ok
```

Note the `it_works`. This comes from the name of our function:

```
fn it_works() {
# }
```

We also get a summary line:

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

So why does our do-nothing test pass? Any test which doesn't `panic!` passes, and any test that does `panic!` fails. Let's make our test fail:

```
#[test]
fn it_works() {
    assert!(false);
}
```

`assert!` is a macro provided by Rust which takes one argument: if the argument is `true`, nothing happens. If the argument is `false`, it `panic!`s. Let's run our tests again:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
```

```
test it_works ... FAILED

failures:

---- it_works stdout ----
        thread 'it_works' panicked at 'assertion failed: false', /home/steve/tmp/adder/src/lib



failures:
    it_works

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

thread '<main>' panicked at 'Some tests failed', /home/steve/src/rust/src/libtest/lib.rs:247
```

Rust indicates that our test failed:

```
test it_works ... FAILED
```

And that's reflected in the summary line:

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

We also get a non-zero status code. We can use `$?` on OS X and Linux:

```
$ echo $?
101
```

On Windows, if you're using `cmd`:

```
> echo %ERRORLEVEL%
```

And if you're using PowerShell:

```
> echo $LASTEXITCODE # the code itself
> echo $? # a boolean, fail or succeed
```

This is useful if you want to integrate `cargo test` into other tooling.
We can invert our test's failure with another attribute: `should_panic`:

```
#[test]
#[should_panic]
fn it_works() {
    assert!(false);
}
```

This test will now succeed if we `panic!` and fail if we complete. Let's try it:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

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

   Doc-tests adder

running 0 tests

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

Rust provides another macro, `assert_eq!`, that compares two arguments for equality:

```
#[test]
#[should_panic]
fn it_works() {
    assert_eq!(”Hello”, ”world”);
}
```

Does this test pass or fail? Because of the `should_panic` attribute, it passes:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

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

   Doc-tests adder

running 0 tests

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

`should_panic` tests can be fragile, as it’s hard to guarantee that the test didn’t fail for an unexpected reason. To help with this, an optional `expected` parameter can be added to the `should_panic` attribute. The test harness will make sure that the failure message contains the provided text. A safer version of the example above would be:

```
#[test]
#[should_panic(expected = ”assertion failed”)]
fn it_works() {
    assert_eq!(”Hello”, ”world”);
}
```

That's all there is to the basics! Let's write one 'real' test:

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}
```

This is a very common use of `assert_eq!`: call some function with some known arguments and compare it to the expected output.

3.2.2 The `ignore` attribute

Sometimes a few specific tests can be very time-consuming to execute. These can be disabled by default by using the `ignore` attribute:

```
#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}

#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

Now we run our tests and see that `it_works` is run, but `expensive_test` is not:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 2 tests
test expensive_test ... ignored
test it_works ... ok

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

   Doc-tests adder

running 0 tests

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

The expensive tests can be run explicitly using `cargo test -- --ignored`:

```
$ cargo test -- --ignored
     Running target/adder-91b3e234d4ed382a

running 1 test
test expensive_test ... ok

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

   Doc-tests adder

running 0 tests

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

The `--ignored` argument is an argument to the test binary, and not to Cargo, which is why the command is `cargo test -- --ignored`.

3.2.3 The `tests` module

There is one way in which our existing example is not idiomatic: it's missing the `tests` module. The idiomatic way of writing our example looks like this:

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

There's a few changes here. The first is the introduction of a `mod tests` with a `cfg` attribute. The module allows us to group all of our tests together, and to also define helper functions if needed, that don't become a part of the rest of our crate. The `cfg` attribute only compiles our test code if we're currently trying to run the tests. This can save compile time, and also ensures that our tests are entirely left out of a normal build.

The second change is the `use` declaration. Because we're in an inner module, we need to bring our test function into scope. This can be annoying if you have a large module, and so this is a common use of globs. Let's change our `src/lib.rs` to make use of it:

```
pub fn add_two(a: i32) -> i32 {
    a + 2
```

```
}

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

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

Note the different `use` line. Now we run our tests:

```
$ cargo test
    Updating registry 'https://github.com/rust-lang/crates.io-index'
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

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

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

   Doc-tests adder

running 0 tests

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

It works!

The current convention is to use the `tests` module to hold your "unit-style" tests. Anything that just tests one small bit of functionality makes sense to go here. But what about "integration-style" tests instead? For that, we have the `tests` directory.

3.2.4 The `tests` directory

To write an integration test, let's make a `tests` directory, and put a `tests/lib.rs` file inside, with this as its contents:

```
extern crate adder;

#[test]
fn it_works() {
    assert_eq!(4, adder::add_two(2));
}
```

This looks similar to our previous tests, but slightly different. We now have an `extern crate adder` at the top. This is because the tests in the `tests` directory are an entirely

separate crate, and so we need to import our library. This is also why `tests` is a suitable place to write integration-style tests: they use the library like any other consumer of it would.

Let's run them:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

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

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

     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

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

   Doc-tests adder

running 0 tests

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

Now we have three sections: our previous test is also run, as well as our new one.

That's all there is to the `tests` directory. The `tests` module isn't needed here, since the whole thing is focused on tests.

Let's finally check out that third section: documentation tests.

3.2.5 Documentation tests

Nothing is better than documentation with examples. Nothing is worse than examples that don't actually work, because the code has changed since the documentation has been written. To this end, Rust supports automatically running examples in your documentation (**note:** this only works in library crates, not binary crates). Here's a fleshed-out `src/lib.rs` with examples:

```
//! The ‘adder‘ crate provides functions that add numbers to other numbers.
//!
//! # Examples
//!
//! “‘
//! assert_eq!(4, adder::add_two(2));
//! “‘
```

```
/// This function adds two to its argument.
///
/// # Examples
///
/// ```
/// use adder::add_two;
///
/// assert_eq!(4, add_two(2));
/// ```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

Note the module-level documentation with `//!` and the function-level documentation with `///`. Rust's documentation supports Markdown in comments, and so triple graves mark code blocks. It is conventional to include the `# Examples` section, exactly like that, with examples following.

Let's run the tests again:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/adder-91b3e234d4ed382a

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

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

     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

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

   Doc-tests adder

running 2 tests
```

```
test add_two_0 ... ok
test _0 ... ok

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

Now we have all three kinds of tests running! Note the names of the documentation tests: the `_0` is generated for the module test, and `add_two_0` for the function test. These will auto increment with names like `add_two_1` as you add more examples.

We haven't covered all of the details with writing documentation tests. For more, please see the Documentation chapter[8]

One final note: documentation tests *cannot* be run on binary crates. To see more on file arrangement see the Crates and Modules[9] section.

3.3 Conditional Compilation

Rust has a special attribute, `#[cfg]`, which allows you to compile code based on a flag passed to the compiler. It has two forms:

```
#[cfg(foo)]
# fn foo() {}

#[cfg(bar = "baz")]
# fn bar() {}
```

They also have some helpers:

```
#[cfg(any(unix, windows))]
# fn foo() {}

#[cfg(all(unix, target_pointer_width = "32"))]
# fn bar() {}

#[cfg(not(foo))]
# fn not_foo() {}
```

These can nest arbitrarily:

```
#[cfg(any(not(unix), all(target_os="macos", target_arch = "powerpc")))]
# fn foo() {}
```

As for how to enable or disable these switches, if you're using Cargo, they get set in the `[features]` section[10] of your `Cargo.toml`:

[8] `documentation.html`
[9] `crates-and-modules.html`
[10] `http://doc.crates.io/manifest.html#the-features-section`

```
[features]
# no features by default
default = []

# The "secure-password" feature depends on the bcrypt package.
secure-password = ["bcrypt"]
```

When you do this, Cargo passes along a flag to `rustc`:

```
--cfg feature="${feature_name}"
```

The sum of these `cfg` flags will determine which ones get activated, and therefore, which code gets compiled. Let's take this code:

```
#[cfg(feature = "foo")]
mod foo {
}
```

If we compile it with `cargo build --features "foo"`, it will send the `--cfg feature="foo"` flag to `rustc`, and the output will have the `mod foo` in it. If we compile it with a regular `cargo build`, no extra flags get passed on, and so, no `foo` module will exist.

3.3.1 cfg_attr

You can also set another attribute based on a `cfg` variable with `cfg_attr`:

```
#[cfg_attr(a, b)]
# fn foo() {}
```

Will be the same as `#[b]` if `a` is set by `cfg` attribute, and nothing otherwise.

3.3.2 cfg!

The `cfg!` syntax extension[11] lets you use these kinds of flags elsewhere in your code, too:

```
if cfg!(target_os = "macos") || cfg!(target_os = "ios") {
    println!("Think Different!");
}
```

These will be replaced by a `true` or `false` at compile-time, depending on the configuration settings.

3.4 Documentation

Documentation is an important part of any software project, and it's first-class in Rust. Let's talk about the tooling Rust gives you to document your project.

[11] `compiler-plugins.html`

About `rustdoc`

The Rust distribution includes a tool, `rustdoc`, that generates documentation. `rustdoc` is also used by Cargo through `cargo doc`.

Documentation can be generated in two ways: from source code, and from standalone Markdown files.

Documenting source code

The primary way of documenting a Rust project is through annotating the source code. You can use documentation comments for this purpose:

```
/// Constructs a new ‘Rc<T>‘.
///
/// # Examples
///
/// “‘
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// “‘
pub fn new(value: T) -> Rc<T> {
    // implementation goes here
}
```

This code generates documentation that looks like this[12]. I've left the implementation out, with a regular comment in its place.

The first thing to notice about this annotation is that it uses `///` instead of `//`. The triple slash indicates a documentation comment.

Documentation comments are written in Markdown.

Rust keeps track of these comments, and uses them when generating documentation. This is important when documenting things like enums:

```
/// The ‘Option‘ type. See [the module level documentation](index.html) for more.
enum Option<T> {
    /// No value
    None,
    /// Some value ‘T‘
    Some(T),
}
```

The above works, but this does not:

```
/// The ‘Option‘ type. See [the module level documentation](index.html) for more.
enum Option<T> {
    None, /// No value
    Some(T), /// Some value ‘T‘
}
```

[12] `https://doc.rust-lang.org/nightly/std/rc/struct.Rc.html#method.new`

You'll get an error:

```
hello.rs:4:1: 4:2 error: expected ident, found `}`
hello.rs:4 }
           ^
```

This unfortunate error[13] is correct: documentation comments apply to the thing after them, and there's nothing after that last comment.

Writing documentation comments Anyway, let's cover each part of this comment in detail:

```
/// Constructs a new `Rc<T>`.
# fn foo() {}
```

The first line of a documentation comment should be a short summary of its functionality. One sentence. Just the basics. High level.

```
///
/// Other details about constructing `Rc<T>`s, maybe describing complicated
/// semantics, maybe additional options, all kinds of stuff.
///
# fn foo() {}
```

Our original example had just a summary line, but if we had more things to say, we could have added more explanation in a new paragraph.

Special sections Next, are special sections. These are indicated with a header, `#`. There are four kinds of headers that are commonly used. They aren't special syntax, just convention, for now.

```
/// # Panics
# fn foo() {}
```

Unrecoverable misuses of a function (i.e. programming errors) in Rust are usually indicated by panics, which kill the whole current thread at the very least. If your function has a non-trivial contract like this, that is detected/enforced by panics, documenting it is very important.

```
/// # Failures
# fn foo() {}
```

If your function or method returns a `Result<T, E>`, then describing the conditions under which it returns `Err(E)` is a nice thing to do. This is slightly less important than `Panics`, because failure is encoded into the type system, but it's still a good thing to do.

[13] `https://github.com/rust-lang/rust/issues/22547`

```
/// # Safety
# fn foo() {}
```

If your function is `unsafe`, you should explain which invariants the caller is responsible for upholding.

```
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
# fn foo() {}
```

Fourth, `Examples`. Include one or more examples of using your function or method, and your users will love you for it. These examples go inside of code block annotations, which we'll talk about in a moment, and can have more than one section:

```
/// # Examples
///
/// Simple `&str` patterns:
///
/// ```
/// let v: Vec<&str> = "Mary had a little lamb".split(' ').collect();
/// assert_eq!(v, vec!["Mary", "had", "a", "little", "lamb"]);
/// ```
///
/// More complex patterns with a lambda:
///
/// ```
/// let v: Vec<&str> = "abc1def2ghi".split(|c: char| c.is_numeric()).collect();
/// assert_eq!(v, vec!["abc", "def", "ghi"]);
/// ```
# fn foo() {}
```

Let's discuss the details of these code blocks.

Code block annotations To write some Rust code in a comment, use the triple graves:

```
/// ```
/// println!("Hello, world");
/// ```
# fn foo() {}
```

If you want something that's not Rust code, you can add an annotation:

```
/// "'c
/// printf("Hello, world\n");
/// "'
# fn foo() {}
```

This will highlight according to whatever language you're showing off. If you're just showing plain text, choose `text`.

It's important to choose the correct annotation here, because `rustdoc` uses it in an interesting way: It can be used to actually test your examples in a library crate, so that they don't get out of date. If you have some C code but `rustdoc` thinks it's Rust because you left off the annotation, `rustdoc` will complain when trying to generate the documentation.

Documentation as tests

Let's discuss our sample example documentation:

```
/// "'
/// println!("Hello, world");
/// "'
# fn foo() {}
```

You'll notice that you don't need a `fn main()` or anything here. `rustdoc` will automatically add a `main()` wrapper around your code, and in the right place. For example:

```
/// "'
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// "'
# fn foo() {}
```

This will end up testing:

```
fn main() {
    use std::rc::Rc;
    let five = Rc::new(5);
}
```

Here's the full algorithm rustdoc uses to preprocess examples:

1. Any leading `#![foo]` attributes are left intact as crate attributes.
2. Some common `allow` attributes are inserted, including `unused_variables`, `unused_assignments`, `unused_mut`, `unused_attributes`, and `dead_code`. Small examples often trigger these lints.
3. If the example does not contain `extern crate`, then `extern crate <mycrate>;` is inserted.

4. Finally, if the example does not contain `fn main`, the remainder of the text is wrapped in `fn main() { your_code }`

Sometimes, this isn't enough, though. For example, all of these code samples with `///` we've been talking about? The raw text:

```
/// Some documentation.
# fn foo() {}
```

looks different than the output:

```
/// Some documentation.
# fn foo() {}
```

Yes, that's right: you can add lines that start with `#`, and they will be hidden from the output, but will be used when compiling your code. You can use this to your advantage. In this case, documentation comments need to apply to some kind of function, so if I want to show you just a documentation comment, I need to add a little function definition below it. At the same time, it's just there to satisfy the compiler, so hiding it makes the example more clear. You can use this technique to explain longer examples in detail, while still preserving the testability of your documentation.

For example, imagine that we wanted to document this code:

```
let x = 5;
let y = 6;
println!(”{}”, x + y);
```

We might want the documentation to end up looking like this:

> First, we set `x` to five:
>
> ```
> let x = 5;
> # let y = 6;
> # println!(”{}”, x + y);
> ```
>
> Next, we set `y` to six:
>
> ```
> # let x = 5;
> let y = 6;
> # println!(”{}”, x + y);
> ```
>
> Finally, we print the sum of `x` and `y`:
>
> ```
> # let x = 5;
> # let y = 6;
> println!(”{}”, x + y);
> ```

To keep each code block testable, we want the whole program in each block, but we don’t want the reader to see every line every time. Here’s what we put in our source code:

```
First, we set ‘x‘ to five:

“‘text
let x = 5;
# let y = 6;
# println!(”{}”, x + y);
“‘

Next, we set ‘y‘ to six:

“‘text
# let x = 5;
let y = 6;
# println!(”{}”, x + y);
“‘

Finally, we print the sum of ‘x‘ and ‘y‘:

“‘text
# let x = 5;
# let y = 6;
println!(”{}”, x + y);
“‘
```

By repeating all parts of the example, you can ensure that your example still compiles, while only showing the parts that are relevant to that part of your explanation.

Documenting macros Here’s an example of documenting a macro:

```
/// Panic with a given message unless an expression evaluates to true.
///
/// # Examples
///
/// “‘
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(1 + 1 == 2, “Math is broken.”);
/// # }
/// “‘
///
/// “‘should_panic
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(true == false, “I’m broken.”);
```

```
/// # }
/// ```
#[macro_export]
macro_rules! panic_unless {
    ($condition:expr, $($rest:expr),+) => ({ if ! $condition { panic!($($rest),+); } });
}
# fn main() {}
```

You'll note three things: we need to add our own `extern crate` line, so that we can add the `#[macro_use]` attribute. Second, we'll need to add our own `main()` as well. Finally, a judicious use of `#` to comment out those two things, so they don't show up in the output.

Another case where the use of `#` is handy is when you want to ignore error handling. Lets say you want the following,

```
/// use std::io;
/// let mut input = String::new();
/// try!(io::stdin().read_line(&mut input));
```

The problem is that `try!` returns a `Result<T, E>` and test functions don't return anything so this will give a mismatched types error.

```
/// A doc test using try!
///
/// ```
/// use std::io;
/// # fn foo() -> io::Result<()> {
/// let mut input = String::new();
/// try!(io::stdin().read_line(&mut input));
/// # Ok(())
/// # }
/// ```
# fn foo() {}
```

You can get around this by wrapping the code in a function. This catches and swallows the `Result<T, E>` when running tests on the docs. This pattern appears regularly in the standard library.

Running documentation tests To run the tests, either:

```
$ rustdoc --test path/to/my/crate/root.rs
# or
$ cargo test
```

That's right, `cargo test` tests embedded documentation too. **However,** `cargo test` **will not test binary crates, only library ones.** This is due to the way `rustdoc` works: it links against the library to be tested, but with a binary, there's nothing to link to.

There are a few more annotations that are useful to help `rustdoc` do the right thing when testing your code:

```
/// "'ignore
/// fn foo() {
/// "'
# fn foo() {}
```

The `ignore` directive tells Rust to ignore your code. This is almost never what you want, as it's the most generic. Instead, consider annotating it with `text` if it's not code, or using `#`s to get a working example that only shows the part you care about.

```
/// "'should_panic
/// assert!(false);
/// "'
# fn foo() {}
```

`should_panic` tells `rustdoc` that the code should compile correctly, but not actually pass as a test.

```
/// "'no_run
/// loop {
///     println!("Hello, world");
/// }
/// "'
# fn foo() {}
```

The `no_run` attribute will compile your code, but not run it. This is important for examples such as "Here's how to start up a network service," which you would want to make sure compile, but might run in an infinite loop!

Documenting modules Rust has another kind of doc comment, `//!`. This comment doesn't document the next item, but the enclosing item. In other words:

```
mod foo {
    //! This is documentation for the 'foo' module.
    //!
    //! # Examples

    // ...
}
```

This is where you'll see `//!` used most often: for module documentation. If you have a module in `foo.rs`, you'll often open its code and see this:

```
//! A module for using 'foo's.
//!
//! The 'foo' module contains a lot of useful functionality blah blah blah
```

Documentation comment style Check out RFC 505[14] for full conventions around the style and format of documentation.

[14] https://github.com/rust-lang/rfcs/blob/master/text/0505-api-comment-conventions.md

Other documentation

All of this behavior works in non-Rust source files too. Because comments are written in Markdown, they're often `.md` files.

When you write documentation in Markdown files, you don't need to prefix the documentation with comments. For example:

```
/// # Examples
///
/// “‘
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// “‘
# fn foo() {}
```

is just

```
# Examples

“‘
use std::rc::Rc;

let five = Rc::new(5);
“‘
```

when it's in a Markdown file. There is one wrinkle though: Markdown files need to have a title like this:

```
% The title

This is the example documentation.
```

This `%` line needs to be the very first line of the file.

`doc` attributes

At a deeper level, documentation comments are syntactic sugar for documentation attributes:

```
/// this
# fn foo() {}

#[doc=”this”]
# fn bar() {}
```

are the same, as are these:

```
//! this

#![doc="this"]
```

You won't often see this attribute used for writing documentation, but it can be useful when changing some options, or when writing a macro.

Re-exports `rustdoc` will show the documentation for a public re-export in both places:

```
extern crate foo;

pub use foo::bar;
```

This will create documentation for `bar` both inside the documentation for the crate `foo`, as well as the documentation for your crate. It will use the same documentation in both places.

This behavior can be suppressed with `no_inline`:

```
extern crate foo;

#[doc(no_inline)]
pub use foo::bar;
```

Missing documentation

Sometimes you want to make sure that every single public thing in your project is documented, especially when you are working on a library. Rust allows you to to generate warnings or errors, when an item is missing documentation. To generate warnings you use `warn`:

```
#![warn(missing_docs)]
```

And to generate errors you use `deny`:

```
#![deny(missing_docs)]
```

There are cases where you want to disable these warnings/errors to explicitly leave something undocumented. This is done by using `allow`:

```
#[allow(missing_docs)]
struct Undocumented;
```

You might even want to hide items from the documentation completely:

```
#[doc(hidden)]
struct Hidden;
```

Controlling HTML You can control a few aspects of the HTML that `rustdoc` generates through the `#![doc]` version of the attribute:

```
#![doc(html_logo_url = "https://www.rust-lang.org/logos/rust-logo-128x128-blk-v2.png",
       html_favicon_url = "https://www.rust-lang.org/favicon.ico",
       html_root_url = "https://doc.rust-lang.org/")]
```

This sets a few different options, with a logo, favicon, and a root URL.

Configuring documentation tests You can also configure the way that `rustdoc` tests your documentation examples through the `#![doc(test(..))]` attribute.

```
#![doc(test(attr(allow(unused_variables), deny(warnings))))]
```

This allows unused variables within the examples, but will fail the test for any other lint warning thrown.

Generation options

`rustdoc` also contains a few other options on the command line, for further customization:

- `--html-in-header FILE`: includes the contents of FILE at the end of the `<head>...</head>` section.
- `--html-before-content FILE`: includes the contents of FILE directly after `<body>`, before the rendered content (including the search bar).
- `--html-after-content FILE`: includes the contents of FILE after all the rendered content.

Security note

The Markdown in documentation comments is placed without processing into the final webpage. Be careful with literal HTML:

```
/// <script>alert(document.cookie)</script>
# fn foo() {}
```

3.5 Iterators

Let's talk about loops.

Remember Rust's `for` loop? Here's an example:

```
for x in 0..10 {
    println!("{}", x);
}
```

Now that you know more Rust, we can talk in detail about how this works. Ranges (the `0..10`) are ‘iterators’. An iterator is something that we can call the `.next()` method on repeatedly, and it gives us a sequence of things.

Like this:

```
let mut range = 0..10;

loop {
    match range.next() {
        Some(x) => {
            println!(”{}”, x);
        },
        None => { break }
    }
}
```

We make a mutable binding to the range, which is our iterator. We then `loop`, with an inner `match`. This `match` is used on the result of `range.next()`, which gives us a reference to the next value of the iterator. `next` returns an `Option<i32>`, in this case, which will be `Some(i32)` when we have a value and `None` once we run out. If we get `Some(i32)`, we print it out, and if we get `None`, we `break` out of the loop.

This code sample is basically the same as our `for` loop version. The `for` loop is just a handy way to write this `loop`/`match`/`break` construct.

`for` loops aren’t the only thing that uses iterators, however. Writing your own iterator involves implementing the `Iterator` trait. While doing that is outside of the scope of this guide, Rust provides a number of useful iterators to accomplish various tasks. But first, a few notes about limitations of ranges.

Ranges are very primitive, and we often can use better alternatives. Consider the following Rust anti-pattern: using ranges to emulate a C-style `for` loop. Let’s suppose you needed to iterate over the contents of a vector. You may be tempted to write this:

```
let nums = vec![1, 2, 3];

for i in 0..nums.len() {
    println!(”{}”, nums[i]);
}
```

This is strictly worse than using an actual iterator. You can iterate over vectors directly, so write this:

```
let nums = vec![1, 2, 3];

for num in &nums {
    println!(”{}”, num);
}
```

There are two reasons for this. First, this more directly expresses what we mean. We iterate through the entire vector, rather than iterating through indexes, and then

indexing the vector. Second, this version is more efficient: the first version will have extra bounds checking because it used indexing, `nums[i]`. But since we yield a reference to each element of the vector in turn with the iterator, there's no bounds checking in the second example. This is very common with iterators: we can ignore unnecessary bounds checks, but still know that we're safe.

There's another detail here that's not 100% clear because of how `println!` works. `num` is actually of type `&i32`. That is, it's a reference to an `i32`, not an `i32` itself. `println!` handles the dereferencing for us, so we don't see it. This code works fine too:

```
let nums = vec![1, 2, 3];

for num in &nums {
    println!("{}", *num);
}
```

Now we're explicitly dereferencing `num`. Why does `&nums` give us references? Firstly, because we explicitly asked it to with `&`. Secondly, if it gave us the data itself, we would have to be its owner, which would involve making a copy of the data and giving us the copy. With references, we're just borrowing a reference to the data, and so it's just passing a reference, without needing to do the move.

So, now that we've established that ranges are often not what you want, let's talk about what you do want instead.

There are three broad classes of things that are relevant here: iterators, *iterator adaptors*, and *consumers*. Here's some definitions:

- *iterators* give you a sequence of values.
- *iterator adaptors* operate on an iterator, producing a new iterator with a different output sequence.
- *consumers* operate on an iterator, producing some final set of values.

Let's talk about consumers first, since you've already seen an iterator, ranges.

Consumers

A *consumer* operates on an iterator, returning some kind of value or values. The most common consumer is `collect()`. This code doesn't quite compile, but it shows the intention:

```
let one_to_one_hundred = (1..101).collect();
```

As you can see, we call `collect()` on our iterator. `collect()` takes as many values as the iterator will give it, and returns a collection of the results. So why won't this compile? Rust can't determine what type of things you want to collect, and so you need to let it know. Here's the version that does compile:

```
let one_to_one_hundred = (1..101).collect::<Vec<i32>>();
```

If you remember, the `::<>` syntax allows us to give a type hint, and so we tell it that we want a vector of integers. You don't always need to use the whole type, though. Using a _ will let you provide a partial hint:

```
let one_to_one_hundred = (1..101).collect::<Vec<_>>();
```

This says "Collect into a `Vec<T>`, please, but infer what the `T` is for me." _ is sometimes called a "type placeholder" for this reason.

`collect()` is the most common consumer, but there are others too. `find()` is one:

```
let greater_than_forty_two = (0..100)
                             .find(|x| *x > 42);

match greater_than_forty_two {
    Some(_) => println!("Found a match!"),
    None => println!("No match found :("),
}
```

`find` takes a closure, and works on a reference to each element of an iterator. This closure returns `true` if the element is the element we're looking for, and `false` otherwise. `find` returns the first element satisfying the specified predicate. Because we might not find a matching element, `find` returns an `Option` rather than the element itself.

Another important consumer is `fold`. Here's what it looks like:

```
let sum = (1..4).fold(0, |sum, x| sum + x);
```

`fold()` is a consumer that looks like this: `fold(base, |accumulator, element| ...)`. It takes two arguments: the first is an element called the *base*. The second is a closure that itself takes two arguments: the first is called the *accumulator*, and the second is an *element*. Upon each iteration, the closure is called, and the result is the value of the accumulator on the next iteration. On the first iteration, the base is the value of the accumulator.

Okay, that's a bit confusing. Let's examine the values of all of these things in this iterator:

base	accumulator	element	closure result
0	0	1	1
0	1	2	3
0	3	3	6

We called `fold()` with these arguments:

```
# (1..4)
.fold(0, |sum, x| sum + x);
```

So, `0` is our base, `sum` is our accumulator, and `x` is our element. On the first iteration, we set `sum` to `0`, and `x` is the first element of `nums`, `1`. We then add `sum` and `x`, which gives us `0 + 1 = 1`. On the second iteration, that value becomes our accumulator, `sum`, and the element is the second element of the array, `2`. `1 + 2 = 3`, and so that becomes the value of the accumulator for the last iteration. On that iteration, `x` is the last element, `3`, and `3 + 3 = 6`, which is our final result for our sum. `1 + 2 + 3 = 6`, and that's the result we got.

Whew. `fold` can be a bit strange the first few times you see it, but once it clicks, you can use it all over the place. Any time you have a list of things, and you want a single result, `fold` is appropriate.

Consumers are important due to one additional property of iterators we haven't talked about yet: laziness. Let's talk some more about iterators, and you'll see why consumers matter.

Iterators

As we've said before, an iterator is something that we can call the `.next()` method on repeatedly, and it gives us a sequence of things. Because you need to call the method, this means that iterators can be *lazy* and not generate all of the values upfront. This code, for example, does not actually generate the numbers `1-99`, instead creating a value that merely represents the sequence:

```
let nums = 1..100;
```

Since we didn't do anything with the range, it didn't generate the sequence. Let's add the consumer:

```
let nums = (1..100).collect::<Vec<i32>>();
```

Now, `collect()` will require that the range gives it some numbers, and so it will do the work of generating the sequence.

Ranges are one of two basic iterators that you'll see. The other is `iter()`. `iter()` can turn a vector into a simple iterator that gives you each element in turn:

```
let nums = vec![1, 2, 3];

for num in nums.iter() {
   println!("{}", num);
}
```

These two basic iterators should serve you well. There are some more advanced iterators, including ones that are infinite.

That's enough about iterators. Iterator adaptors are the last concept we need to talk about with regards to iterators. Let's get to it!

Iterator adaptors

Iterator adaptors take an iterator and modify it somehow, producing a new iterator. The simplest one is called `map`:

```
(1..100).map(|x| x + 1);
```

`map` is called upon another iterator, and produces a new iterator where each element reference has the closure it's been given as an argument called on it. So this would give us the numbers from `2-100`. Well, almost! If you compile the example, you'll get a warning:

```
warning: unused result which must be used: iterator adaptors are lazy and
         do nothing unless consumed, #[warn(unused_must_use)] on by default
(1..100).map(|x| x + 1);
 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Laziness strikes again! That closure will never execute. This example doesn't print any numbers:

```
(1..100).map(|x| println!("{}", x));
```

If you are trying to execute a closure on an iterator for its side effects, just use `for` instead.

There are tons of interesting iterator adaptors. `take(n)` will return an iterator over the next `n` elements of the original iterator. Let's try it out with an infinite iterator:

```
for i in (1..).take(5) {
    println!("{}", i);
}
```

This will print

```
1
2
3
4
5
```

`filter()` is an adapter that takes a closure as an argument. This closure returns `true` or `false`. The new iterator `filter()` produces only the elements that the closure returns `true` for:

```
for i in (1..100).filter(|&x| x % 2 == 0) {
    println!("{}", i);
}
```

This will print all of the even numbers between one and a hundred. (Note that because `filter` doesn't consume the elements that are being iterated over, it is passed a reference to each element, and thus the filter predicate uses the `&x` pattern to extract the integer itself.)

You can chain all three things together: start with an iterator, adapt it a few times, and then consume the result. Check it out:

```
(1..)
    .filter(|&x| x % 2 == 0)
    .filter(|&x| x % 3 == 0)
    .take(5)
    .collect::<Vec<i32>>();
```

This will give you a vector containing `6`, `12`, `18`, `24`, and `30`.

This is just a small taste of what iterators, iterator adaptors, and consumers can help you with. There are a number of really useful iterators, and you can write your own as well. Iterators provide a safe, efficient way to manipulate all kinds of lists. They're a little unusual at first, but if you play with them, you'll get hooked. For a full list of the different iterators and consumers, check out the iterator module documentation[15].

3.6 Concurrency

Concurrency and parallelism are incredibly important topics in computer science, and are also a hot topic in industry today. Computers are gaining more and more cores, yet many programmers aren't prepared to fully utilize them.

Rust's memory safety features also apply to its concurrency story too. Even concurrent Rust programs must be memory safe, having no data races. Rust's type system is up to the task, and gives you powerful ways to reason about concurrent code at compile time.

Before we talk about the concurrency features that come with Rust, it's important to understand something: Rust is low-level enough that the vast majority of this is provided by the standard library, not by the language. This means that if you don't like some aspect of the way Rust handles concurrency, you can implement an alternative way of doing things. mio[16] is a real-world example of this principle in action.

Background: `Send` and `Sync`

Concurrency is difficult to reason about. In Rust, we have a strong, static type system to help us reason about our code. As such, Rust gives us two traits to help us make sense of code that can possibly be concurrent.

[15] `../std/iter/index.html`
[16] `https://github.com/carllerche/mio`

`Send` The first trait we're going to talk about is `Send`[17]. When a type `T` implements `Send`, it indicates that something of this type is able to have ownership transferred safely between threads.

This is important to enforce certain restrictions. For example, if we have a channel connecting two threads, we would want to be able to send some data down the channel and to the other thread. Therefore, we'd ensure that `Send` was implemented for that type.

In the opposite way, if we were wrapping a library with FFI[18] that isn't threadsafe, we wouldn't want to implement `Send`, and so the compiler will help us enforce that it can't leave the current thread.

`Sync` The second of these traits is called `Sync`[19]. When a type `T` implements `Sync`, it indicates that something of this type has no possibility of introducing memory unsafety when used from multiple threads concurrently through shared references. This implies that types which don't have interior mutability[20] are inherently `Sync`, which includes simple primitive types (like `u8`) and aggregate types containing them.

For sharing references across threads, Rust provides a wrapper type called `Arc<T>`. `Arc<T>` implements `Send` and `Sync` if and only if `T` implements both `Send` and `Sync`. For example, an object of type `Arc<RefCell<U>` cannot be transferred across threads because `RefCell`[21] does not implement `Sync`, consequently `Arc<RefCell<U>` would not implement `Send`.

These two traits allow you to use the type system to make strong guarantees about the properties of your code under concurrency. Before we demonstrate why, we need to learn how to create a concurrent Rust program in the first place!

Threads

Rust's standard library provides a library for threads, which allow you to run Rust code in parallel. Here's a basic example of using `std::thread`:

```
use std::thread;

fn main() {
    thread::spawn(|| {
        println!(”Hello from a thread!”);
    });
}
```

The `thread::spawn()` method accepts a closure[22], which is executed in a new thread. It returns a handle to the thread, that can be used to wait for the child thread to finish and extract its result:

[17] `../std/marker/trait.Send.html`
[18] `ffi.html`
[19] `../std/marker/trait.Sync.html`
[20] `mutability.html`
[21] `choosing-your-guarantees.html#refcellt`
[22] `closures.html`

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        "Hello from a thread!"
    });

    println!("{}", handle.join().unwrap());
}
```

Many languages have the ability to execute threads, but it's wildly unsafe. There are entire books about how to prevent errors that occur from shared mutable state. Rust helps out with its type system here as well, by preventing data races at compile time. Let's talk about how you actually share things between threads.

Safe Shared Mutable State

Due to Rust's type system, we have a concept that sounds like a lie: "safe shared mutable state." Many programmers agree that shared mutable state is very, very bad.

Someone once said this:

> Shared mutable state is the root of all evil. Most languages attempt to deal with this problem through the 'mutable' part, but Rust deals with it by solving the 'shared' part.

The same ownership system[23] that helps prevent using pointers incorrectly also helps rule out data races, one of the worst kinds of concurrency bugs.

As an example, here is a Rust program that would have a data race in many languages. It will not compile:

```
use std::thread;
use std::time::Duration;

fn main() {
    let mut data = vec![1, 2, 3];

    for i in 0..3 {
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

This gives us an error:

[23] ownership.html

```
8:17 error: capture of moved value: ‘data‘
        data[i] += 1;
        ^~~~
```

Rust knows this wouldn't be safe! If we had a reference to `data` in each thread, and the thread takes ownership of the reference, we'd have three owners!

So, we need some type that lets us have more than one reference to a value and that we can share between threads, that is it must implement `Sync`.

We'll use `Arc<T>`, Rust's standard atomic reference count type, which wraps a value up with some extra runtime bookkeeping which allows us to share the ownership of the value between multiple references at the same time.

The bookkeeping consists of a count of how many of these references exist to the value, hence the reference count part of the name.

The Atomic part means `Arc<T>` can safely be accessed from multiple threads. To do this the compiler guarantees that mutations of the internal count use indivisible operations which can't have data races.

```
use std::thread;
use std::sync::Arc;
use std::time::Duration;

fn main() {
    let mut data = Arc::new(vec![1, 2, 3]);

    for i in 0..3 {
        let data = data.clone();
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

We now call `clone()` on our `Arc<T>`, which increases the internal count. This handle is then moved into the new thread.

And... still gives us an error.

```
<anon>:11:24 error: cannot borrow immutable borrowed content as mutable
<anon>:11                    data[i] += 1;
                             ^~~~
```

`Arc<T>` assumes one more property about its contents to ensure that it is safe to share across threads: it assumes its contents are `Sync`. This is true for our value if it's immutable, but we want to be able to mutate it, so we need something else to persuade the borrow checker we know what we're doing.

It looks like we need some type that allows us to safely mutate a shared value, for example a type that can ensure only one thread at a time is able to mutate the value inside it at any one time.

For that, we can use the `Mutex<T>` type!
Here's the working version:

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

fn main() {
    let data = Arc::new(Mutex::new(vec![1, 2, 3]));

    for i in 0..3 {
        let data = data.clone();
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

Note that the value of `i` is bound (copied) to the closure and not shared among the threads.
Also note that `lock`[24] method of `Mutex`[25] has this signature:

```
fn lock(&self) -> LockResult<MutexGuard<T>>
```

and because `Send` is not implemented for `MutexGuard<T>`, the guard cannot cross thread boundaries, ensuring thread-locality of lock acquire and release.
Let's examine the body of the thread more closely:

```
# use std::sync::{Arc, Mutex};
# use std::thread;
# use std::time::Duration;
# fn main() {
#     let data = Arc::new(Mutex::new(vec![1, 2, 3]));
#     for i in 0..3 {
#         let data = data.clone();
thread::spawn(move || {
    let mut data = data.lock().unwrap();
    data[i] += 1;
});
#     }
#     thread::sleep(Duration::from_millis(50));
# }
```

[24] ../std/sync/struct.Mutex.html#method.lock
[25] ../std/sync/struct.Mutex.html

First, we call `lock()`, which acquires the mutex's lock. Because this may fail, it returns an `Result<T, E>`, and because this is just an example, we `unwrap()` it to get a reference to the data. Real code would have more robust error handling here. We're then free to mutate it, since we have the lock.

Lastly, while the threads are running, we wait on a short timer. But this is not ideal: we may have picked a reasonable amount of time to wait but it's more likely we'll either be waiting longer than necessary or not long enough, depending on just how much time the threads actually take to finish computing when the program runs.

A more precise alternative to the timer would be to use one of the mechanisms provided by the Rust standard library for synchronizing threads with each other. Let's talk about one of them: channels.

Channels

Here's a version of our code that uses channels for synchronization, rather than waiting for a specific time:

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

fn main() {
    let data = Arc::new(Mutex::new(0));

    let (tx, rx) = mpsc::channel();

    for _ in 0..10 {
        let (data, tx) = (data.clone(), tx.clone());

        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            *data += 1;

            tx.send(());
        });
    }

    for _ in 0..10 {
        rx.recv();
    }
}
```

We use the `mpsc::channel()` method to construct a new channel. We just `send` a simple `()` down the channel, and then wait for ten of them to come back.

While this channel is just sending a generic signal, we can send any data that is `Send` over the channel!

```
use std::thread;
```

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    for i in 0..10 {
        let tx = tx.clone();

        thread::spawn(move || {
            let answer = i * i;

            tx.send(answer);
        });
    }

    for _ in 0..10 {
        println!(”{}”, rx.recv().unwrap());
    }
}
```

Here we create 10 threads, asking each to calculate the square of a number (`i` at the time of `spawn()`), and then `send()` back the answer over the channel.

Panics

A `panic!` will crash the currently executing thread. You can use Rust’s threads as a simple isolation mechanism:

```
use std::thread;

let handle = thread::spawn(move || {
    panic!(”oops!”);
});

let result = handle.join();

assert!(result.is_err());
```

`Thread.join()` gives us a `Result` back, which allows us to check if the thread has panicked or not.

3.7 Error Handling

Like most programming languages, Rust encourages the programmer to handle errors in a particular way. Generally speaking, error handling is divided into two broad categories: exceptions and return values. Rust opts for return values.

In this chapter, we intend to provide a comprehensive treatment of how to deal with errors in Rust. More than that, we will attempt to introduce error handling one piece at a time so that you'll come away with a solid working knowledge of how everything fits together.

When done naïvely, error handling in Rust can be verbose and annoying. This chapter will explore those stumbling blocks and demonstrate how to use the standard library to make error handling concise and ergonomic.

3.7.1 Table of Contents

This chapter is very long, mostly because we start at the very beginning with sum types and combinators, and try to motivate the way Rust does error handling incrementally. As such, programmers with experience in other expressive type systems may want to jump around.

3.7.2 The Basics

You can think of error handling as using *case analysis* to determine whether a computation was successful or not. As you will see, the key to ergonomic error handling is reducing the amount of explicit case analysis the programmer has to do while keeping code composable.

Keeping code composable is important, because without that requirement, we could `panic`[26] whenever we come across something unexpected. (`panic` causes the current task to unwind, and in most cases, the entire program aborts.) Here's an example:

```
// Guess a number between 1 and 10.
// If it matches the number we had in mind, return true. Else, return false.
fn guess(n: i32) -> bool {
    if n < 1 || n > 10 {
        panic!(”Invalid number: {}”, n);
    }
    n == 5
}

fn main() {
    guess(11);
}
```

If you try running this code, the program will crash with a message like this:

```
thread ’<main>’ panicked at ’Invalid number: 11’, src/bin/panic-simple.rs:5
```

Here's another example that is slightly less contrived. A program that accepts an integer as an argument, doubles it and prints it.

```
use std::env;

fn main() {
    let mut argv = env::args();
    let arg: String = argv.nth(1).unwrap(); // error 1
    let n: i32 = arg.parse().unwrap(); // error 2
    println!(”{}”, 2 * n);
}
```

If you give this program zero arguments (error 1) or if the first argument isn't an integer (error 2), the program will panic just like in the first example.

You can think of this style of error handling as similar to a bull running through a china shop. The bull will get to where it wants to go, but it will trample everything in the process.

[26] `../std/macro.panic!.html`

Unwrapping explained

In the previous example, we claimed that the program would simply panic if it reached one of the two error conditions, yet, the program does not include an explicit call to `panic` like the first example. This is because the panic is embedded in the calls to `unwrap`.

To "unwrap" something in Rust is to say, "Give me the result of the computation, and if there was an error, just panic and stop the program." It would be better if we just showed the code for unwrapping because it is so simple, but to do that, we will first need to explore the `Option` and `Result` types. Both of these types have a method called `unwrap` defined on them.

The `Option` type The `Option` type is defined in the standard library[27]:

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

The `Option` type is a way to use Rust's type system to express the *possibility of absence*. Encoding the possibility of absence into the type system is an important concept because it will cause the compiler to force the programmer to handle that absence. Let's take a look at an example that tries to find a character in a string:

```
// Searches ‘haystack‘ for the Unicode character ‘needle‘. If one is found, the
// byte offset of the character is returned. Otherwise, ‘None‘ is returned.
fn find(haystack: &str, needle: char) -> Option<usize> {
    for (offset, c) in haystack.char_indices() {
        if c == needle {
            return Some(offset);
        }
    }
    None
}
```

Notice that when this function finds a matching character, it doesn't just return the `offset`. Instead, it returns `Some(offset)`. `Some` is a variant or a *value constructor* for the `Option` type. You can think of it as a function with the type `fn<T>(value: T) -> Option<T>`. Correspondingly, `None` is also a value constructor, except it has no arguments. You can think of `None` as a function with the type `fn<T>() -> Option<T>`.

This might seem like much ado about nothing, but this is only half of the story. The other half is *using* the `find` function we've written. Let's try to use it to find the extension in a file name.

```
# fn find(_: &str, _: char) -> Option<usize> { None }
fn main() {
```

[27] `../std/option/enum.Option.html`

```
    let file_name = "foobar.rs";
    match find(file_name, '.') {
        None => println!("No file extension found."),
        Some(i) => println!("File extension: {}", &file_name[i+1..]),
    }
}
```

This code uses pattern matching[28] to do *case analysis* on the `Option<usize>` returned by the `find` function. In fact, case analysis is the only way to get at the value stored inside an `Option<T>`. This means that you, as the programmer, must handle the case when an `Option<T>` is `None` instead of `Some(t)`.

But wait, what about `unwrap`,which we used `previously`? There was no case analysis there! Instead, the case analysis was put inside the `unwrap` method for you. You could define it yourself if you want:

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

impl<T> Option<T> {
    fn unwrap(self) -> T {
        match self {
            Option::Some(val) => val,
            Option::None =>
              panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

The `unwrap` method *abstracts away the case analysis.* This is precisely the thing that makes `unwrap` ergonomic to use. Unfortunately, that `panic!` means that `unwrap` is not composable: it is the bull in the china shop.

Composing `Option<T>` values In an example from before, we saw how to use `find` to discover the extension in a file name. Of course, not all file names have a `.` in them, so it's possible that the file name has no extension. This *possibility of absence* is encoded into the types using `Option<T>`. In other words, the compiler will force us to address the possibility that an extension does not exist. In our case, we just print out a message saying as such.

Getting the extension of a file name is a pretty common operation, so it makes sense to put it into a function:

```
# fn find(_: &str, _: char) -> Option<usize> { None }
// Returns the extension of the given file name, where the extension is defined
// as all characters proceeding the first `.`.
```

[28] `../book/patterns.html`

```
// If ‘file_name‘ has no ‘.‘, then ‘None‘ is returned.
fn extension_explicit(file_name: &str) -> Option<&str> {
    match find(file_name, '.') {
        None => None,
        Some(i) => Some(&file_name[i+1..]),
    }
}
```

(Pro-tip: don't use this code. Use the `extension`[29] method in the standard library instead.)

The code stays simple, but the important thing to notice is that the type of `find` forces us to consider the possibility of absence. This is a good thing because it means the compiler won't let us accidentally forget about the case where a file name doesn't have an extension. On the other hand, doing explicit case analysis like we've done in `extension_explicit` every time can get a bit tiresome.

In fact, the case analysis in `extension_explicit` follows a very common pattern: *map* a function on to the value inside of an `Option<T>`, unless the option is `None`, in which case, just return `None`.

Rust has parametric polymorphism, so it is very easy to define a combinator that abstracts this pattern:

```
fn map<F, T, A>(option: Option<T>, f: F) -> Option<A> where F: FnOnce(T) -> A {
    match option {
        None => None,
        Some(value) => Some(f(value)),
    }
}
```

Indeed, `map` is defined as a method[30] on `Option<T>` in the standard library.

Armed with our new combinator, we can rewrite our `extension_explicit` method to get rid of the case analysis:

```
# fn find(_: &str, _: char) -> Option<usize> { None }
// Returns the extension of the given file name, where the extension is defined
// as all characters proceeding the first ‘.‘.
// If ‘file_name‘ has no ‘.‘, then ‘None‘ is returned.
fn extension(file_name: &str) -> Option<&str> {
    find(file_name, '.').map(|i| &file_name[i+1..])
}
```

One other pattern we commonly find is assigning a default value to the case when an `Option` value is `None`. For example, maybe your program assumes that the extension of a file is `rs` even if none is present. As you might imagine, the case analysis for this is not specific to file extensions - it can work with any `Option<T>`:

[29] ../std/path/struct.Path.html#method.extension
[30] ../std/option/enum.Option.html#method.map

```
fn unwrap_or<T>(option: Option<T>, default: T) -> T {
    match option {
        None => default,
        Some(value) => value,
    }
}
```

The trick here is that the default value must have the same type as the value that might be inside the `Option<T>`. Using it is dead simple in our case:

```
# fn find(haystack: &str, needle: char) -> Option<usize> {
#     for (offset, c) in haystack.char_indices() {
#         if c == needle {
#             return Some(offset);
#         }
#     }
#     None
# }
#
# fn extension(file_name: &str) -> Option<&str> {
#     find(file_name, '.').map(|i| &file_name[i+1..])
# }
fn main() {
    assert_eq!(extension("foobar.csv").unwrap_or("rs"), "csv");
    assert_eq!(extension("foobar").unwrap_or("rs"), "rs");
}
```

(Note that `unwrap_or` is defined as a method[31] on `Option<T>` in the standard library, so we use that here instead of the free-standing function we defined above. Don't forget to check out the more general `unwrap_or_else`[32] method.)

There is one more combinator that we think is worth paying special attention to: `and_then`. It makes it easy to compose distinct computations that admit the *possibility of absence*. For example, much of the code in this section is about finding an extension given a file name. In order to do this, you first need the file name which is typically extracted from a file *path*. While most file paths have a file name, not *all* of them do. For example, `.`, `..` or `/`.

So, we are tasked with the challenge of finding an extension given a file *path*. Let's start with explicit case analysis:

```
# fn extension(file_name: &str) -> Option<&str> { None }
fn file_path_ext_explicit(file_path: &str) -> Option<&str> {
    match file_name(file_path) {
        None => None,
        Some(name) => match extension(name) {
            None => None,
            Some(ext) => Some(ext),
```

[31] ../std/option/enum.Option.html#method.unwrap_or

[32] ../std/option/enum.Option.html#method.unwrap_or_else

```
        }
    }
}

fn file_name(file_path: &str) -> Option<&str> {
  // implementation elided
  unimplemented!()
}
```

You might think that we could just use the `map` combinator to reduce the case analysis, but its type doesn't quite fit. Namely, `map` takes a function that does something only with the inner value. The result of that function is then *always* rewrapped with `Some`. Instead, we need something like `map`, but which allows the caller to return another `Option`. Its generic implementation is even simpler than `map`:

```
fn and_then<F, T, A>(option: Option<T>, f: F) -> Option<A>
        where F: FnOnce(T) -> Option<A> {
    match option {
        None => None,
        Some(value) => f(value),
    }
}
```

Now we can rewrite our `file_path_ext` function without explicit case analysis:

```
# fn extension(file_name: &str) -> Option<&str> { None }
# fn file_name(file_path: &str) -> Option<&str> { None }
fn file_path_ext(file_path: &str) -> Option<&str> {
    file_name(file_path).and_then(extension)
}
```

The `Option` type has many other combinators defined in the standard library[33]. It is a good idea to skim this list and familiarize yourself with what's available—they can often reduce case analysis for you. Familiarizing yourself with these combinators will pay dividends because many of them are also defined (with similar semantics) for `Result`, which we will talk about next.

Combinators make using types like `Option` ergonomic because they reduce explicit case analysis. They are also composable because they permit the caller to handle the possibility of absence in their own way. Methods like `unwrap` remove choices because they will panic if `Option<T>` is `None`.

The `Result` type

The `Result` type is also defined in the standard library[34]:

[33] ../std/option/enum.Option.html
[34] ../std/result/

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

The `Result` type is a richer version of `Option`. Instead of expressing the possibility of *absence* like `Option` does, `Result` expresses the possibility of *error*. Usually, the *error* is used to explain why the execution of some computation failed. This is a strictly more general form of `Option`. Consider the following type alias, which is semantically equivalent to the real `Option<T>` in every way:

```
type Option<T> = Result<T, ()>;
```

This fixes the second type parameter of `Result` to always be `()` (pronounced "unit" or "empty tuple"). Exactly one value inhabits the `()` type: `()`. (Yup, the type and value level terms have the same notation!)

The `Result` type is a way of representing one of two possible outcomes in a computation. By convention, one outcome is meant to be expected or "`Ok`" while the other outcome is meant to be unexpected or "`Err`".

Just like `Option`, the `Result` type also has an `unwrap` method defined[35] in the standard library. Let's define it:

```
# enum Result<T, E> { Ok(T), Err(E) }
impl<T, E: ::std::fmt::Debug> Result<T, E> {
    fn unwrap(self) -> T {
        match self {
            Result::Ok(val) => val,
            Result::Err(err) =>
              panic!("called `Result::unwrap()` on an `Err` value: {:?}", err),
        }
    }
}
```

This is effectively the same as our definition for `Option::unwrap`, except it includes the error value in the `panic!` message. This makes debugging easier, but it also requires us to add a `Debug`[36] constraint on the `E` type parameter (which represents our error type). Since the vast majority of types should satisfy the `Debug` constraint, this tends to work out in practice. (`Debug` on a type simply means that there's a reasonable way to print a human readable description of values with that type.)

OK, let's move on to an example.

Parsing integers The Rust standard library makes converting strings to integers dead simple. It's so easy in fact, that it is very tempting to write something like the following:

[35] `../std/result/enum.Result.html#method.unwrap`
[36] `../std/fmt/trait.Debug.html`

```
fn double_number(number_str: &str) -> i32 {
    2 * number_str.parse::<i32>().unwrap()
}

fn main() {
    let n: i32 = double_number("10");
    assert_eq!(n, 20);
}
```

At this point, you should be skeptical of calling `unwrap`. For example, if the string doesn't parse as a number, you'll get a panic:

```
thread '<main>' panicked at 'called `Result::unwrap()` on an `Err` value: ParseIntError { kind
```

This is rather unsightly, and if this happened inside a library you're using, you might be understandably annoyed. Instead, we should try to handle the error in our function and let the caller decide what to do. This means changing the return type of `double_number`. But to what? Well, that requires looking at the signature of the `parse` method[37] in the standard library:

```
impl str {
    fn parse<F: FromStr>(&self) -> Result<F, F::Err>;
}
```

Hmm. So we at least know that we need to use a `Result`. Certainly, it's possible that this could have returned an `Option`. After all, a string either parses as a number or it doesn't, right? That's certainly a reasonable way to go, but the implementation internally distinguishes *why* the string didn't parse as an integer. (Whether it's an empty string, an invalid digit, too big or too small.) Therefore, using a `Result` makes sense because we want to provide more information than simply "absence." We want to say *why* the parsing failed. You should try to emulate this line of reasoning when faced with a choice between `Option` and `Result`. If you can provide detailed error information, then you probably should. (We'll see more on this later.)

OK, but how do we write our return type? The `parse` method as defined above is generic over all the different number types defined in the standard library. We could (and probably should) also make our function generic, but let's favor explicitness for the moment. We only care about `i32`, so we need to find its implementation of `FromStr`[38] (do a `CTRL-F` in your browser for "FromStr") and look at its associated type[39] `Err`. We did this so we can find the concrete error type. In this case, it's `std::num::ParseIntError`[40]. Finally, we can rewrite our function:

```
use std::num::ParseIntError;

fn double_number(number_str: &str) -> Result<i32, ParseIntError> {
```

[37] ../std/primitive.str.html#method.parse
[38] ../std/primitive.i32.html
[39] ../book/associated-types.html
[40] ../std/num/struct.ParseIntError.html

```
    match number_str.parse::<i32>() {
        Ok(n) => Ok(2 * n),
        Err(err) => Err(err),
    }
}

fn main() {
    match double_number("10") {
        Ok(n) => assert_eq!(n, 20),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

This is a little better, but now we've written a lot more code! The case analysis has once again bitten us.

Combinators to the rescue! Just like `Option`, `Result` has lots of combinators defined as methods. There is a large intersection of common combinators between `Result` and `Option`. In particular, `map` is part of that intersection:

```
use std::num::ParseIntError;

fn double_number(number_str: &str) -> Result<i32, ParseIntError> {
    number_str.parse::<i32>().map(|n| 2 * n)
}

fn main() {
    match double_number("10") {
        Ok(n) => assert_eq!(n, 20),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

The usual suspects are all there for `Result`, including `unwrap_or`[41] and `and_then`[42]. Additionally, since `Result` has a second type parameter, there are combinators that affect only the error type, such as `map_err`[43] (instead of `map`) and `or_else`[44] (instead of `and_then`).

The `Result` type alias idiom In the standard library, you may frequently see types like `Result<i32>`. But wait, we defined `Result` to have two type parameters. How can we get away with only specifying one? The key is to define a `Result` type alias that *fixes* one of the type parameters to a particular type. Usually the fixed type is the error type. For example, our previous example parsing integers could be rewritten like this:

[41] ../std/result/enum.Result.html#method.unwrap_or
[42] ../std/result/enum.Result.html#method.and_then
[43] ../std/result/enum.Result.html#method.map_err
[44] ../std/result/enum.Result.html#method.or_else

```
use std::num::ParseIntError;
use std::result;

type Result<T> = result::Result<T, ParseIntError>;

fn double_number(number_str: &str) -> Result<i32> {
    unimplemented!();
}
```

Why would we do this? Well, if we have a lot of functions that could return `ParseIntError`, then it's much more convenient to define an alias that always uses `ParseIntError` so that we don't have to write it out all the time.

The most prominent place this idiom is used in the standard library is with `io::Result`[45]. Typically, one writes `io::Result<T>`, which makes it clear that you're using the `io` module's type alias instead of the plain definition from `std::result`. (This idiom is also used for `fmt::Result`[46].)

A brief interlude: unwrapping isn't evil

If you've been following along, you might have noticed that I've taken a pretty hard line against calling methods like `unwrap` that could `panic` and abort your program. *Generally speaking*, this is good advice.

However, `unwrap` can still be used judiciously. What exactly justifies use of `unwrap` is somewhat of a grey area and reasonable people can disagree. I'll summarize some of my *opinions* on the matter.

- **In examples and quick 'n' dirty code.** Sometimes you're writing examples or a quick program, and error handling simply isn't important. Beating the convenience of `unwrap` can be hard in such scenarios, so it is very appealing.
- **When panicking indicates a bug in the program.** When the invariants of your code should prevent a certain case from happening (like, say, popping from an empty stack), then panicking can be permissible. This is because it exposes a bug in your program. This can be explicit, like from an `assert!` failing, or it could be because your index into an array was out of bounds.

This is probably not an exhaustive list. Moreover, when using an `Option`, it is often better to use its `expect`[47] method. `expect` does exactly the same thing as `unwrap`, except it prints a message you give to `expect`. This makes the resulting panic a bit nicer to deal with, since it will show your message instead of "called unwrap on a `None` value."

My advice boils down to this: use good judgment. There's a reason why the words "never do X" or "Y is considered harmful" don't appear in my writing. There are trade offs to all things, and it is up to you as the programmer to determine what is acceptable for your use cases. My goal is only to help you evaluate trade offs as accurately as possible.

[45] `../std/io/type.Result.html`
[46] `../std/fmt/type.Result.html`
[47] `../std/option/enum.Option.html#method.expect`

Now that we've covered the basics of error handling in Rust, and explained unwrapping, let's start exploring more of the standard library.

3.7.3 Working with multiple error types

Thus far, we've looked at error handling where everything was either an `Option<T>` or a `Result<T, SomeError>`. But what happens when you have both an `Option` and a `Result`? Or what if you have a `Result<T, Error1>` and a `Result<T, Error2>`? Handling *composition of distinct error types* is the next challenge in front of us, and it will be the major theme throughout the rest of this chapter.

Composing `Option` and `Result`

So far, I've talked about combinators defined for `Option` and combinators defined for `Result`. We can use these combinators to compose results of different computations without doing explicit case analysis.

Of course, in real code, things aren't always as clean. Sometimes you have a mix of `Option` and `Result` types. Must we resort to explicit case analysis, or can we continue using combinators?

For now, let's revisit one of the first examples in this chapter:

```
use std::env;

fn main() {
    let mut argv = env::args();
    let arg: String = argv.nth(1).unwrap(); // error 1
    let n: i32 = arg.parse().unwrap(); // error 2
    println!("{}", 2 * n);
}
```

Given our new found knowledge of `Option`, `Result` and their various combinators, we should try to rewrite this so that errors are handled properly and the program doesn't panic if there's an error.

The tricky aspect here is that `argv.nth(1)` produces an `Option` while `arg.parse()` produces a `Result`. These aren't directly composable. When faced with both an `Option` and a `Result`, the solution is *usually* to convert the `Option` to a `Result`. In our case, the absence of a command line parameter (from `env::args()`) means the user didn't invoke the program correctly. We could just use a `String` to describe the error. Let's try:

```
use std::env;

fn double_arg(mut argv: env::Args) -> Result<i32, String> {
    argv.nth(1)
        .ok_or("Please give at least one argument".to_owned())
        .and_then(|arg| arg.parse::<i32>().map_err(|err| err.to_string()))
        .map(|n| 2 * n)
```

```
}

fn main() {
    match double_arg(env::args()) {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

There are a couple new things in this example. The first is the use of the `Option::ok_or`[48] combinator. This is one way to convert an `Option` into a `Result`. The conversion requires you to specify what error to use if `Option` is `None`. Like the other combinators we've seen, its definition is very simple:

```
fn ok_or<T, E>(option: Option<T>, err: E) -> Result<T, E> {
    match option {
        Some(val) => Ok(val),
        None => Err(err),
    }
}
```

The other new combinator used here is `Result::map_err`[49]. This is just like `Result::map`, except it maps a function on to the *error* portion of a `Result` value. If the `Result` is an `Ok(...)` value, then it is returned unmodified.

We use `map_err` here because it is necessary for the error types to remain the same (because of our use of `and_then`). Since we chose to convert the `Option<String>` (from `argv.nth(1)`) to a `Result<String, String>`, we must also convert the `ParseIntError` from `arg.parse()` to a `String`.

The limits of combinators

Doing IO and parsing input is a very common task, and it's one that I personally have done a lot of in Rust. Therefore, we will use (and continue to use) IO and various parsing routines to exemplify error handling.

Let's start simple. We are tasked with opening a file, reading all of its contents and converting its contents to a number. Then we multiply it by `2` and print the output.

Although I've tried to convince you not to use `unwrap`, it can be useful to first write your code using `unwrap`. It allows you to focus on your problem instead of the error handling, and it exposes the points where proper error handling need to occur. Let's start there so we can get a handle on the code, and then refactor it to use better error handling.

```
use std::fs::File;
use std::io::Read;
use std::path::Path;
```

[48] ../std/option/enum.Option.html#method.ok_or
[49] ../std/result/enum.Result.html#method.map_err

```
fn file_double<P: AsRef<Path>>(file_path: P) -> i32 {
    let mut file = File::open(file_path).unwrap(); // error 1
    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap(); // error 2
    let n: i32 = contents.trim().parse().unwrap(); // error 3
    2 * n
}

fn main() {
    let doubled = file_double("foobar");
    println!("{}", doubled);
}
```

(N.B. The `AsRef<Path>` is used because those are the same bounds used on `std::fs::File::open`[50]. This makes it ergonomic to use any kind of string as a file path.)

There are three different errors that can occur here:

1. A problem opening the file.
2. A problem reading data from the file.
3. A problem parsing the data as a number.

The first two problems are described via the `std::io::Error`[51] type. We know this because of the return types of `std::fs::File::open`[52] and `std::io::Read::read_to_string`[53]. (Note that they both use the `Result` type alias idiom described previously. If you click on the `Result` type, you'll see the type alias[54], and consequently, the underlying `io::Error` type.) The third problem is described by the `std::num::ParseIntError`[55] type. The `io::Error` type in particular is *pervasive* throughout the standard library. You will see it again and again.

Let's start the process of refactoring the `file_double` function. To make this function composable with other components of the program, it should *not* panic if any of the above error conditions are met. Effectively, this means that the function should *return an error* if any of its operations fail. Our problem is that the return type of `file_double` is `i32`, which does not give us any useful way of reporting an error. Thus, we must start by changing the return type from `i32` to something else.

The first thing we need to decide: should we use `Option` or `Result`? We certainly could use `Option` very easily. If any of the three errors occur, we could simply return `None`. This will work *and it is better than panicking*, but we can do a lot better. Instead, we should pass some detail about the error that occurred. Since we want to express the *possibility of error*, we should use `Result<i32, E>`. But what should `E` be? Since two *different* types of errors can occur, we need to convert them to a common type. One such type is `String`. Let's see how that impacts our code:

[50] ../std/fs/struct.File.html#method.open
[51] ../std/io/struct.Error.html
[52] ../std/fs/struct.File.html#method.open
[53] ../std/io/trait.Read.html#method.read_to_string
[54] ../std/io/type.Result.html
[55] ../std/num/struct.ParseIntError.html

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    File::open(file_path)
         .map_err(|err| err.to_string())
         .and_then(|mut file| {
              let mut contents = String::new();
              file.read_to_string(&mut contents)
                  .map_err(|err| err.to_string())
                  .map(|_| contents)
         })
         .and_then(|contents| {
              contents.trim().parse::<i32>()
                      .map_err(|err| err.to_string())
         })
         .map(|n| 2 * n)
}

fn main() {
    match file_double(”foobar”) {
        Ok(n) => println!(”{}”, n),
        Err(err) => println!(”Error: {}”, err),
    }
}
```

This code looks a bit hairy. It can take quite a bit of practice before code like this becomes easy to write. The way we write it is by *following the types*. As soon as we changed the return type of `file_double` to `Result<i32, String>`, we had to start looking for the right combinators. In this case, we only used three different combinators: `and_then`, `map` and `map_err`.

`and_then` is used to chain multiple computations where each computation could return an error. After opening the file, there are two more computations that could fail: reading from the file and parsing the contents as a number. Correspondingly, there are two calls to `and_then`.

`map` is used to apply a function to the `Ok(...)` value of a `Result`. For example, the very last call to `map` multiplies the `Ok(...)` value (which is an `i32`) by `2`. If an error had occurred before that point, this operation would have been skipped because of how `map` is defined.

`map_err` is the trick that makes all of this work. `map_err` is just like `map`, except it applies a function to the `Err(...)` value of a `Result`. In this case, we want to convert all of our errors to one type: `String`. Since both `io::Error` and `num::ParseIntError` implement `ToString`, we can call the `to_string()` method to convert them.

With all of that said, the code is still hairy. Mastering use of combinators is important, but they have their limits. Let’s try a different approach: early returns.

Early returns

I'd like to take the code from the previous section and rewrite it using *early returns*. Early returns let you exit the function early. We can't return early in `file_double` from inside another closure, so we'll need to revert back to explicit case analysis.

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = match File::open(file_path) {
        Ok(file) => file,
        Err(err) => return Err(err.to_string()),
    };
    let mut contents = String::new();
    if let Err(err) = file.read_to_string(&mut contents) {
        return Err(err.to_string());
    }
    let n: i32 = match contents.trim().parse() {
        Ok(n) => n,
        Err(err) => return Err(err.to_string()),
    };
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

Reasonable people can disagree over whether this code is better that the code that uses combinators, but if you aren't familiar with the combinator approach, this code looks simpler to read to me. It uses explicit case analysis with `match` and `if let`. If an error occurs, it simply stops executing the function and returns the error (by converting it to a string).

Isn't this a step backwards though? Previously, we said that the key to ergonomic error handling is reducing explicit case analysis, yet we've reverted back to explicit case analysis here. It turns out, there are *multiple* ways to reduce explicit case analysis. Combinators aren't the only way.

The `try!` macro

A cornerstone of error handling in Rust is the `try!` macro. The `try!` macro abstracts case analysis just like combinators, but unlike combinators, it also abstracts *control flow*. Namely, it can abstract the *early return* pattern seen above.

Here is a simplified definition of a `try!` macro:

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(err),
    });
}
```

(The real definition[56] is a bit more sophisticated. We will address that later.)

Using the `try!` macro makes it very easy to simplify our last example. Since it does the case analysis and the early return for us, we get tighter code that is easier to read:

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = try!(File::open(file_path).map_err(|e| e.to_string()));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(|e| e.to_string()));
    let n = try!(contents.trim().parse::<i32>().map_err(|e| e.to_string()));
    Ok(2 * n)
}

fn main() {
    match file_double(”foobar”) {
        Ok(n) => println!(”{}”, n),
        Err(err) => println!(”Error: {}”, err),
    }
}
```

The `map_err` calls are still necessary given our definition of `try!`. This is because the error types still need to be converted to `String`. The good news is that we will soon learn how to remove those `map_err` calls! The bad news is that we will need to learn a bit more about a couple important traits in the standard library before we can remove the `map_err` calls.

Defining your own error type

Before we dive into some of the standard library error traits, I’d like to wrap up this section by removing the use of `String` as our error type in the previous examples.

Using `String` as we did in our previous examples is convenient because it’s easy to convert errors to strings, or even make up your own errors as strings on the spot. However, using `String` for your errors has some downsides.

The first downside is that the error messages tend to clutter your code. It’s possible to define the error messages elsewhere, but unless you’re unusually disciplined, it is very tempting to embed the error message into your code. Indeed, we did exactly this in a previous example.

[56] `../std/macro.try!.html`

The second and more important downside is that `String`s are *lossy*. That is, if all errors are converted to strings, then the errors we pass to the caller become completely opaque. The only reasonable thing the caller can do with a `String` error is show it to the user. Certainly, inspecting the string to determine the type of error is not robust. (Admittedly, this downside is far more important inside of a library as opposed to, say, an application.)

For example, the `io::Error` type embeds an `io::ErrorKind`[57], which is *structured data* that represents what went wrong during an IO operation. This is important because you might want to react differently depending on the error. (e.g., A `BrokenPipe` error might mean quitting your program gracefully while a `NotFound` error might mean exiting with an error code and showing an error to the user.) With `io::ErrorKind`, the caller can examine the type of an error with case analysis, which is strictly superior to trying to tease out the details of an error inside of a `String`.

Instead of using a `String` as an error type in our previous example of reading an integer from a file, we can define our own error type that represents errors with *structured data*. We endeavor to not drop information from underlying errors in case the caller wants to inspect the details.

The ideal way to represent *one of many possibilities* is to define our own sum type using `enum`. In our case, an error is either an `io::Error` or a `num::ParseIntError`, so a natural definition arises:

```
use std::io;
use std::num;

// We derive ‘Debug‘ because all types should probably derive ‘Debug‘.
// This gives us a reasonable human readable description of ‘CliError‘ values.
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}
```

Tweaking our code is very easy. Instead of converting errors to strings, we simply convert them to our `CliError` type using the corresponding value constructor:

```
# #[derive(Debug)]
# enum CliError { Io(::std::io::Error), Parse(::std::num::ParseIntError) }
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, CliError> {
    let mut file = try!(File::open(file_path).map_err(CliError::Io));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(CliError::Io));
    let n: i32 = try!(contents.trim().parse().map_err(CliError::Parse));
    Ok(2 * n)
```

[57] ../std/io/enum.ErrorKind.html

```
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

The only change here is switching `map_err(|e| e.to_string())` (which converts errors to strings) to `map_err(CliError::Io)` or `map_err(CliError::Parse)`. The *caller* gets to decide the level of detail to report to the user. In effect, using a `String` as an error type removes choices from the caller while using a custom `enum` error type like `CliError` gives the caller all of the conveniences as before in addition to *structured data* describing the error.

A rule of thumb is to define your own error type, but a `String` error type will do in a pinch, particularly if you're writing an application. If you're writing a library, defining your own error type should be strongly preferred so that you don't remove choices from the caller unnecessarily.

3.7.4 Standard library traits used for error handling

The standard library defines two integral traits for error handling: `std::error::Error`[58] and `std::convert::From`[59]. While `Error` is designed specifically for generically describing errors, the `From` trait serves a more general role for converting values between two distinct types.

The `Error` trait

The `Error` trait is defined in the standard library[60]:

```
use std::fmt::{Debug, Display};

trait Error: Debug + Display {
  /// A short description of the error.
  fn description(&self) -> &str;

  /// The lower level cause of this error, if any.
  fn cause(&self) -> Option<&Error> { None }
}
```

This trait is super generic because it is meant to be implemented for *all* types that represent errors. This will prove useful for writing composable code as we'll see later. Otherwise, the trait allows you to do at least the following things:

[58] ../std/error/trait.Error.html
[59] ../std/convert/trait.From.html
[60] ../std/error/trait.Error.html

- Obtain a `Debug` representation of the error.
- Obtain a user-facing `Display` representation of the error.
- Obtain a short description of the error (via the `description` method).
- Inspect the causal chain of an error, if one exists (via the `cause` method).

The first two are a result of `Error` requiring impls for both `Debug` and `Display`. The latter two are from the two methods defined on `Error`. The power of `Error` comes from the fact that all error types impl `Error`, which means errors can be existentially quantified as a trait object[61]. This manifests as either `Box<Error>` or `&Error`. Indeed, the `cause` method returns an `&Error`, which is itself a trait object. We'll revisit the `Error` trait's utility as a trait object later.

For now, it suffices to show an example implementing the `Error` trait. Let's use the error type we defined in the previous section:

```
use std::io;
use std::num;

// We derive ‘Debug‘ because all types should probably derive ‘Debug‘.
// This gives us a reasonable human readable description of ‘CliError‘ values.
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}
```

This particular error type represents the possibility of two types of errors occurring: an error dealing with I/O or an error converting a string to a number. The error could represent as many error types as you want by adding new variants to the `enum` definition.

Implementing `Error` is pretty straight-forward. It's mostly going to be a lot explicit case analysis.

```
use std::error;
use std::fmt;

impl fmt::Display for CliError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            // Both underlying errors already impl ‘Display‘, so we defer to
            // their implementations.
            CliError::Io(ref err) => write!(f, ”IO error: {}”, err),
            CliError::Parse(ref err) => write!(f, ”Parse error: {}”, err),
        }
    }
}

impl error::Error for CliError {
```

[61] `../book/trait-objects.html`

```
    fn description(&self) -> &str {
        // Both underlying errors already impl `Error`, so we defer to their
        // implementations.
        match *self {
            CliError::Io(ref err) => err.description(),
            CliError::Parse(ref err) => err.description(),
        }
    }

    fn cause(&self) -> Option<&error::Error> {
        match *self {
            // N.B. Both of these implicitly cast `err` from their concrete
            // types (either `&io::Error` or `&num::ParseIntError`)
            // to a trait object `&Error`. This works because both error types
            // implement `Error`.
            CliError::Io(ref err) => Some(err),
            CliError::Parse(ref err) => Some(err),
        }
    }
}
```

We note that this is a very typical implementation of `Error`: match on your different error types and satisfy the contracts defined for `description` and `cause`.

The `From` trait

The `std::convert::From` trait is defined in the standard library[62]:

```
trait From<T> {
    fn from(T) -> Self;
}
```

Deliciously simple, yes? `From` is very useful because it gives us a generic way to talk about conversion *from* a particular type `T` to some other type (in this case, "some other type" is the subject of the impl, or `Self`). The crux of `From` is the set of implementations provided by the standard library[63].

Here are a few simple examples demonstrating how `From` works:

```
let string: String = From::from("foo");
let bytes: Vec<u8> = From::from("foo");
let cow: ::std::borrow::Cow<str> = From::from("foo");
```

OK, so `From` is useful for converting between strings. But what about errors? It turns out, there is one critical impl:

```
impl<'a, E: Error + 'a> From<E> for Box<Error + 'a>
```

[62] ../std/convert/trait.From.html
[63] ../std/convert/trait.From.html

This impl says that for *any* type that impls `Error`, we can convert it to a trait object `Box<Error>`. This may not seem terribly surprising, but it is useful in a generic context.

Remember the two errors we were dealing with previously? Specifically, `io::Error` and `num::ParseIntError`. Since both impl `Error`, they work with `From`:

```
use std::error::Error;
use std::fs;
use std::io;
use std::num;

// We have to jump through some hoops to actually get error values.
let io_err: io::Error = io::Error::last_os_error();
let parse_err: num::ParseIntError = "not a number".parse::<i32>().unwrap_err();

// OK, here are the conversions.
let err1: Box<Error> = From::from(io_err);
let err2: Box<Error> = From::from(parse_err);
```

There is a really important pattern to recognize here. Both `err1` and `err2` have the *same type*. This is because they are existentially quantified types, or trait objects. In particular, their underlying type is *erased* from the compiler's knowledge, so it truly sees `err1` and `err2` as exactly the same. Additionally, we constructed `err1` and `err2` using precisely the same function call: `From::from`. This is because `From::from` is overloaded on both its argument and its return type.

This pattern is important because it solves a problem we had earlier: it gives us a way to reliably convert errors to the same type using the same function.

Time to revisit an old friend; the `try!` macro.

The real `try!` macro

Previously, we presented this definition of `try!`:

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(err),
    });
}
```

This is not its real definition. Its real definition is in the standard library[64]:

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(::std::convert::From::from(err)),
    });
}
```

[64] `../std/macro.try!.html`

There's one tiny but powerful change: the error value is passed through `From::from`. This makes the `try!` macro a lot more powerful because it gives you automatic type conversion for free.

Armed with our more powerful `try!` macro, let's take a look at code we wrote previously to read a file and convert its contents to an integer:

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = try!(File::open(file_path).map_err(|e| e.to_string()));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(|e| e.to_string()));
    let n = try!(contents.trim().parse::<i32>().map_err(|e| e.to_string()));
    Ok(2 * n)
}
```

Earlier, we promised that we could get rid of the `map_err` calls. Indeed, all we have to do is pick a type that `From` works with. As we saw in the previous section, `From` has an impl that lets it convert any error type into a `Box<Error>`:

```
use std::error::Error;
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, Box<Error>> {
    let mut file = try!(File::open(file_path));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents));
    let n = try!(contents.trim().parse::<i32>());
    Ok(2 * n)
}
```

We are getting very close to ideal error handling. Our code has very little overhead as a result from error handling because the `try!` macro encapsulates three things simultaneously:

1. Case analysis.
2. Control flow.
3. Error type conversion.

When all three things are combined, we get code that is unencumbered by combinators, calls to `unwrap` or case analysis.

There's one little nit left: the `Box<Error>` type is *opaque*. If we return a `Box<Error>` to the caller, the caller can't (easily) inspect underlying error type. The situation is certainly better than `String` because the caller can call methods like `description`[65] and

[65] `../std/error/trait.Error.html#tymethod.description`

`cause`[66], but the limitation remains: `Box<Error>` is opaque. (N.B. This isn't entirely true because Rust does have runtime reflection, which is useful in some scenarios that are beyond the scope of this chapter[67].)

It's time to revisit our custom `CliError` type and tie everything together.

Composing custom error types

In the last section, we looked at the real `try!` macro and how it does automatic type conversion for us by calling `From::from` on the error value. In particular, we converted errors to `Box<Error>`, which works, but the type is opaque to callers.

To fix this, we use the same remedy that we're already familiar with: a custom error type. Once again, here is the code that reads the contents of a file and converts it to an integer:

```
use std::fs::File;
use std::io::{self, Read};
use std::num;
use std::path::Path;

// We derive `Debug` because all types should probably derive `Debug`.
// This gives us a reasonable human readable description of `CliError` values.
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}

fn file_double_verbose<P: AsRef<Path>>(file_path: P) -> Result<i32, CliError> {
    let mut file = try!(File::open(file_path).map_err(CliError::Io));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(CliError::Io));
    let n: i32 = try!(contents.trim().parse().map_err(CliError::Parse));
    Ok(2 * n)
}
```

Notice that we still have the calls to `map_err`. Why? Well, recall the definitions of `try!` and `From`. The problem is that there is no `From` impl that allows us to convert from error types like `io::Error` and `num::ParseIntError` to our own custom `CliError`. Of course, it is easy to fix this! Since we defined `CliError`, we can impl `From` with it:

```
# #[derive(Debug)]
# enum CliError { Io(io::Error), Parse(num::ParseIntError) }
use std::io;
use std::num;

impl From<io::Error> for CliError {
```

[66] `../std/error/trait.Error.html#method.cause`

[67] `https://crates.io/crates/error`

```
    fn from(err: io::Error) -> CliError {
        CliError::Io(err)
    }
}

impl From<num::ParseIntError> for CliError {
    fn from(err: num::ParseIntError) -> CliError {
        CliError::Parse(err)
    }
}
```

All these impls are doing is teaching `From` how to create a `CliError` from other error types. In our case, construction is as simple as invoking the corresponding value constructor. Indeed, it is *typically* this easy.

We can finally rewrite `file_double`:

```
# use std::io;
# use std::num;
# enum CliError { Io(::std::io::Error), Parse(::std::num::ParseIntError) }
# impl From<io::Error> for CliError {
#     fn from(err: io::Error) -> CliError { CliError::Io(err) }
# }
# impl From<num::ParseIntError> for CliError {
#     fn from(err: num::ParseIntError) -> CliError { CliError::Parse(err) }
# }

use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, CliError> {
    let mut file = try!(File::open(file_path));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents));
    let n: i32 = try!(contents.trim().parse());
    Ok(2 * n)
}
```

The only thing we did here was remove the calls to `map_err`. They are no longer needed because the `try!` macro invokes `From::from` on the error value. This works because we've provided `From` impls for all the error types that could appear.

If we modified our `file_double` function to perform some other operation, say, convert a string to a float, then we'd need to add a new variant to our error type:

```
use std::io;
use std::num;

enum CliError {
```

```
    Io(io::Error),
    ParseInt(num::ParseIntError),
    ParseFloat(num::ParseFloatError),
}
```

And add a new `From` impl:

```
# enum CliError {
#     Io(::std::io::Error),
#     ParseInt(num::ParseIntError),
#     ParseFloat(num::ParseFloatError),
# }

use std::num;

impl From<num::ParseFloatError> for CliError {
    fn from(err: num::ParseFloatError) -> CliError {
        CliError::ParseFloat(err)
    }
}
```

And that's it!

Advice for library writers

If your library needs to report custom errors, then you should probably define your own error type. It's up to you whether or not to expose its representation (like `ErrorKind`[68]) or keep it hidden (like `ParseIntError`[69]). Regardless of how you do it, it's usually good practice to at least provide some information about the error beyond just its `String` representation. But certainly, this will vary depending on use cases.

At a minimum, you should probably implement the `Error`[70] trait. This will give users of your library some minimum flexibility for composing errors. Implementing the `Error` trait also means that users are guaranteed the ability to obtain a string representation of an error (because it requires impls for both `fmt::Debug` and `fmt::Display`).

Beyond that, it can also be useful to provide implementations of `From` on your error types. This allows you (the library author) and your users to compose more detailed errors. For example, `csv::Error`[71] provides `From` impls for both `io::Error` and `byteorder::Error`.

Finally, depending on your tastes, you may also want to define a `Result` type alias, particularly if your library defines a single error type. This is used in the standard library for `io::Result`[72] and `fmt::Result`[73].

[68] ../std/io/enum.ErrorKind.html
[69] ../std/num/struct.ParseIntError.html
[70] ../std/error/trait.Error.html
[71] http://burntsushi.net/rustdoc/csv/enum.Error.html
[72] ../std/io/type.Result.html
[73] ../std/fmt/type.Result.html

3.7.5 Case study: A program to read population data

This chapter was long, and depending on your background, it might be rather dense. While there is plenty of example code to go along with the prose, most of it was specifically designed to be pedagogical. So, we're going to do something new: a case study.

For this, we're going to build up a command line program that lets you query world population data. The objective is simple: you give it a location and it will tell you the population. Despite the simplicity, there is a lot that can go wrong!

The data we'll be using comes from the Data Science Toolkit[74]. I've prepared some data from it for this exercise. You can either grab the world population data[75] (41MB gzip compressed, 145MB uncompressed) or just the US population data[76] (2.2MB gzip compressed, 7.2MB uncompressed).

Up until now, we've kept the code limited to Rust's standard library. For a real task like this though, we'll want to at least use something to parse CSV data, parse the program arguments and decode that stuff into Rust types automatically. For that, we'll use the `csv`[77], and `rustc-serialize`[78] crates.

Initial setup

We're not going to spend a lot of time on setting up a project with Cargo because it is already covered well in the Cargo chapter[79] and Cargo's documentation[80].

To get started from scratch, run `cargo new --bin city-pop` and make sure your `Cargo.toml` looks something like this:

```
[package]
name = "city-pop"
version = "0.1.0"
authors = ["Andrew Gallant <jamslam@gmail.com>"]

[[bin]]
name = "city-pop"

[dependencies]
csv = "0.*"
rustc-serialize = "0.*"
getopts = "0.*"
```

You should already be able to run:

```
cargo build --release
./target/release/city-pop
# Outputs: Hello, world!
```

[74] https://github.com/petewarden/dstkdata
[75] http://burntsushi.net/stuff/worldcitiespop.csv.gz
[76] http://burntsushi.net/stuff/uscitiespop.csv.gz
[77] https://crates.io/crates/csv
[78] https://crates.io/crates/rustc-serialize
[79] ../book/hello-cargo.html
[80] http://doc.crates.io/guide.html

Argument parsing

Let's get argument parsing out of the way. We won't go into too much detail on Getopts, but there is some good documentation[81] describing it. The short story is that Getopts generates an argument parser and a help message from a vector of options (The fact that it is a vector is hidden behind a struct and a set of methods). Once the parsing is done, we can decode the program arguments into a Rust struct. From there, we can get information about the flags, for instance, whether they were passed in, and what arguments they had. Here's our program with the appropriate `extern crate` statements, and the basic argument setup for Getopts:

```
extern crate getopts;
extern crate rustc_serialize;

use getopts::Options;
use std::env;

fn print_usage(program: &str, opts: Options) {
    println!("{}", opts.usage(&format!("Usage: {} [options] <data-path> <city>", program)));
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let program = args[0].clone();

    let mut opts = Options::new();
    opts.optflag("h", "help", "Show this usage message.");

    let matches = match opts.parse(&args[1..]) {
        Ok(m)  => { m }
    Err(e) => { panic!(e.to_string()) }
    };
    if matches.opt_present("h") {
        print_usage(&program, opts);
    return;
    }
    let data_path = args[1].clone();
    let city = args[2].clone();

    // Do stuff with information
}
```

First, we get a vector of the arguments passed into our program. We then store the first one, knowing that it is our program's name. Once that's done, we set up our argument flags, in this case a simplistic help message flag. Once we have the argument flags set up, we use `Options.parse` to parse the argument vector (starting from index one, because index 0 is the program name). If this was successful, we assign matches to

[81] `http://doc.rust-lang.org/getopts/getopts/index.html`

the parsed object, if not, we panic. Once past that, we test if the user passed in the help flag, and if so print the usage message. The option help messages are constructed by Getopts, so all we have to do to print the usage message is tell it what we want it to print for the program name and template. If the user has not passed in the help flag, we assign the proper variables to their corresponding arguments.

Writing the logic

We all write code differently, but error handling is usually the last thing we want to think about. This isn't great for the overall design of a program, but it can be useful for rapid prototyping. Because Rust forces us to be explicit about error handling (by making us call `unwrap`), it is easy to see which parts of our program can cause errors.

In this case study, the logic is really simple. All we need to do is parse the CSV data given to us and print out a field in matching rows. Let's do it. (Make sure to add `extern crate csv;` to the top of your file.)

```
// This struct represents the data in each row of the CSV file.
// Type based decoding absolves us of a lot of the nitty gritty error
// handling, like parsing strings as integers or floats.
#[derive(Debug, RustcDecodable)]
struct Row {
    country: String,
    city: String,
    accent_city: String,
    region: String,

    // Not every row has data for the population, latitude or longitude!
    // So we express them as ‘Option‘ types, which admits the possibility of
    // absence. The CSV parser will fill in the correct value for us.
    population: Option<u64>,
    latitude: Option<f64>,
    longitude: Option<f64>,
}

fn print_usage(program: &str, opts: Options) {
    println!(”{}”, opts.usage(&format!(”Usage: {} [options] <data-path> <city>”, program)));
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let program = args[0].clone();

    let mut opts = Options::new();
    opts.optflag(”h”, ”help”, ”Show this usage message.”);

    let matches = match opts.parse(&args[1..]) {
        Ok(m)  => { m }
        Err(e) => { panic!(e.to_string()) }
```

```
    };

    if matches.opt_present("h") {
        print_usage(&program, opts);
        return;
    }

    let data_file = args[1].clone();
    let data_path = Path::new(&data_file);
    let city = args[2].clone();

    let file = fs::File::open(data_path).unwrap();
    let mut rdr = csv::Reader::from_reader(file);

    for row in rdr.decode::<Row>() {
        let row = row.unwrap();

        if row.city == city {
            println!("{}, {}: {:?}",
                row.city, row.country,
                row.population.expect("population count"));
        }
    }
}
```

Let's outline the errors. We can start with the obvious: the three places that `unwrap` is called:

1. `fs::File::open`[82] can return an `io::Error`[83].
2. `csv::Reader::decode`[84] decodes one record at a time, and decoding a record[85] (look at the `Item` associated type on the `Iterator` impl) can produce a `csv::Error`[86].
3. If `row.population` is `None`, then calling `expect` will panic.

Are there any others? What if we can't find a matching city? Tools like `grep` will return an error code, so we probably should too. So we have logic errors specific to our problem, IO errors and CSV parsing errors. We're going to explore two different ways to approach handling these errors.

I'd like to start with `Box<Error>`. Later, we'll see how defining our own error type can be useful.

Error handling with `Box<Error>`

`Box<Error>` is nice because it *just works*. You don't need to define your own error types and you don't need any `From` implementations. The downside is that since

[82] `../std/fs/struct.File.html#method.open`
[83] `../std/io/struct.Error.html`
[84] `http://burntsushi.net/rustdoc/csv/struct.Reader.html#method.decode`
[85] `http://burntsushi.net/rustdoc/csv/struct.DecodedRecords.html`
[86] `http://burntsushi.net/rustdoc/csv/enum.Error.html`

`Box<Error>` is a trait object, it *erases the type*, which means the compiler can no longer reason about its underlying type.

Previously we started refactoring our code by changing the type of our function from `T` to `Result<T, OurErrorType>`. In this case, `OurErrorType` is just `Box<Error>`. But what's `T`? And can we add a return type to `main`?

The answer to the second question is no, we can't. That means we'll need to write a new function. But what is `T`? The simplest thing we can do is to return a list of matching `Row` values as a `Vec<Row>`. (Better code would return an iterator, but that is left as an exercise to the reader.)

Let's refactor our code into its own function, but keep the calls to `unwrap`. Note that we opt to handle the possibility of a missing population count by simply ignoring that row.

```
struct Row {
    // unchanged
}

struct PopulationCount {
    city: String,
    country: String,
    // This is no longer an ‘Option‘ because values of this type are only
    // constructed if they have a population count.
    count: u64,
}

fn print_usage(program: &str, opts: Options) {
    println!(”{}”, opts.usage(&format!(”Usage: {} [options] <data-path> <city>”, program)));
}

fn search<P: AsRef<Path>>(file_path: P, city: &str) -> Vec<PopulationCount> {
    let mut found = vec![];
    let file = fs::File::open(file_path).unwrap();
    let mut rdr = csv::Reader::from_reader(file);
    for row in rdr.decode::<Row>() {
        let row = row.unwrap();
        match row.population {
            None => { } // skip it
            Some(count) => if row.city == city {
                found.push(PopulationCount {
                    city: row.city,
                    country: row.country,
                    count: count,
                });
            },
        }
    }
    found
}
```

```
fn main() {
    let args: Vec<String> = env::args().collect();
    let program = args[0].clone();

    let mut opts = Options::new();
    opts.optflag("h", "help", "Show this usage message.");

    let matches = match opts.parse(&args[1..]) {
        Ok(m)  => { m }
        Err(e) => { panic!(e.to_string()) }
    };
    if matches.opt_present("h") {
        print_usage(&program, opts);
        return;
    }

    let data_file = args[1].clone();
    let data_path = Path::new(&data_file);
    let city = args[2].clone();
    for pop in search(&data_path, &city) {
        println!("{}, {}: {:?}", pop.city, pop.country, pop.count);
    }
}
```

While we got rid of one use of `expect` (which is a nicer variant of `unwrap`), we still should handle the absence of any search results.

To convert this to proper error handling, we need to do the following:

1. Change the return type of `search` to be `Result<Vec<PopulationCount>, Box<Error>>`.
2. Use the `try!` macro so that errors are returned to the caller instead of panicking the program.
3. Handle the error in `main`.

Let's try it:

```
fn search<P: AsRef<Path>>
         (file_path: P, city: &str)
         -> Result<Vec<PopulationCount>, Box<Error+Send+Sync>> {
    let mut found = vec![];
    let file = try!(fs::File::open(file_path));
    let mut rdr = csv::Reader::from_reader(file);
    for row in rdr.decode::<Row>() {
        let row = try!(row);
        match row.population {
            None => { } // skip it
            Some(count) => if row.city == city {
                found.push(PopulationCount {
                    city: row.city,
```

```
                    country: row.country,
                    count: count,
                });
            },
        }
    }
    if found.is_empty() {
        Err(From::from("No matching cities with a population were found."))
    } else {
        Ok(found)
    }
}
```

Instead of `x.unwrap()`, we now have `try!(x)`. Since our function returns a `Result<T, E>`, the `try!` macro will return early from the function if an error occurs.

There is one big gotcha in this code: we used `Box<Error + Send + Sync>` instead of `Box<Error>`. We did this so we could convert a plain string to an error type. We need these extra bounds so that we can use the corresponding `From` impls[87]:

```
// We are making use of this impl in the code above, since we call `From::from`
// on a `&'static str`.
impl<'a, 'b> From<&'b str> for Box<Error + Send + Sync + 'a>

// But this is also useful when you need to allocate a new string for an
// error message, usually with `format!`.
impl From<String> for Box<Error + Send + Sync>
```

Since `search` now returns a `Result<T, E>`, `main` should use case analysis when calling `search`:

```
...
match search(&data_file, &city) {
    Ok(pops) => {
        for pop in pops {
            println!("{}, {}: {:?}", pop.city, pop.country, pop.count);
        }
    }
    Err(err) => println!("{}", err)
}
...
```

Now that we've seen how to do proper error handling with `Box<Error>`, let's try a different approach with our own custom error type. But first, let's take a quick break from error handling and add support for reading from `stdin`.

[87] `../std/convert/trait.From.html`

Reading from stdin

In our program, we accept a single file for input and do one pass over the data. This means we probably should be able to accept input on stdin. But maybe we like the current format too—so let's have both!

Adding support for stdin is actually quite easy. There are only three things we have to do:

1. Tweak the program arguments so that a single parameter—the city—can be accepted while the population data is read from stdin.
2. Modify the program so that an option `-f` can take the file, if it is not passed into stdin.
3. Modify the `search` function to take an *optional* file path. When `None`, it should know to read from stdin.

First, here's the new usage:

```
fn print_usage(program: &str, opts: Options) {
    println!("{}", opts.usage(&format!("Usage: {} [options] <city>", program)));
}
```

The next part is going to be only a little harder:

```
...
let mut opts = Options::new();
opts.optopt("f", "file", "Choose an input file, instead of using STDIN.", "NAME");
opts.optflag("h", "help", "Show this usage message.");
...
let file = matches.opt_str("f");
let data_file = file.as_ref().map(Path::new);

let city = if !matches.free.is_empty() {
    matches.free[0].clone()
} else {
    print_usage(&program, opts);
    return;
};

for pop in search(&data_file, &city) {
    println!("{}, {}: {:?}", pop.city, pop.country, pop.count);
}
...
```

In this piece of code, we take `file` (which has the type `Option<String>`), and convert it to a type that `search` can use, in this case, `&Option<AsRef<Path»`. To do this, we take a reference of file, and map `Path::new` onto it. In this case, `as_ref()` converts the `Option<String>` into an `Option<&str>`, and from there, we can execute `Path::new` to the content of the optional, and return the optional of the new value. Once we have that, it is a simple matter of getting the `city` argument and executing `search`.

Modifying `search` is slightly trickier. The `csv` crate can build a parser out of any type that implements `io::Read`[88]. But how can we use the same code over both types? There's actually a couple ways we could go about this. One way is to write `search` such that it is generic on some type parameter `R` that satisfies `io::Read`. Another way is to just use trait objects:

```
fn search<P: AsRef<Path>>
          (file_path: &Option<P>, city: &str)
          -> Result<Vec<PopulationCount>, Box<Error+Send+Sync>> {
    let mut found = vec![];
    let input: Box<io::Read> = match *file_path {
        None => Box::new(io::stdin()),
        Some(ref file_path) => Box::new(try!(fs::File::open(file_path))),
    };
    let mut rdr = csv::Reader::from_reader(input);
    // The rest remains unchanged!
}
```

Error handling with a custom type

Previously, we learned how to compose errors using a custom error type. We did this by defining our error type as an `enum` and implementing `Error` and `From`.

Since we have three distinct errors (IO, CSV parsing and not found), let's define an `enum` with three variants:

```
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Csv(csv::Error),
    NotFound,
}
```

And now for impls on `Display` and `Error`:

```
impl fmt::Display for CliError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            CliError::Io(ref err) => err.fmt(f),
            CliError::Csv(ref err) => err.fmt(f),
            CliError::NotFound => write!(f, "No matching cities with a \
                                             population were found."),
        }
    }
}

impl Error for CliError {
    fn description(&self) -> &str {
```

[88] `http://burntsushi.net/rustdoc/csv/struct.Reader.html#method.from_reader`

```
        match *self {
            CliError::Io(ref err) => err.description(),
            CliError::Csv(ref err) => err.description(),
            CliError::NotFound => "not found",
        }
    }
}
```

Before we can use our `CliError` type in our `search` function, we need to provide a couple `From` impls. How do we know which impls to provide? Well, we'll need to convert from both `io::Error` and `csv::Error` to `CliError`. Those are the only external errors, so we'll only need two `From` impls for now:

```
impl From<io::Error> for CliError {
    fn from(err: io::Error) -> CliError {
        CliError::Io(err)
    }
}

impl From<csv::Error> for CliError {
    fn from(err: csv::Error) -> CliError {
        CliError::Csv(err)
    }
}
```

The `From` impls are important because of how `try!` is defined. In particular, if an error occurs, `From::from` is called on the error, which in this case, will convert it to our own error type `CliError`.

With the `From` impls done, we only need to make two small tweaks to our `search` function: the return type and the "not found" error. Here it is in full:

```
fn search<P: AsRef<Path>>
          (file_path: &Option<P>, city: &str)
          -> Result<Vec<PopulationCount>, CliError> {
    let mut found = vec![];
    let input: Box<io::Read> = match *file_path {
        None => Box::new(io::stdin()),
        Some(ref file_path) => Box::new(try!(fs::File::open(file_path))),
    };
    let mut rdr = csv::Reader::from_reader(input);
    for row in rdr.decode::<Row>() {
        let row = try!(row);
        match row.population {
            None => { } // skip it
            Some(count) => if row.city == city {
                found.push(PopulationCount {
                    city: row.city,
                    country: row.country,
                    count: count,
```

```
                });
            },
        }
    }
    if found.is_empty() {
        Err(CliError::NotFound)
    } else {
        Ok(found)
    }
}
```

No other changes are necessary.

Adding functionality

Writing generic code is great, because generalizing stuff is cool, and it can then be useful later. But sometimes, the juice isn't worth the squeeze. Look at what we just did in the previous step:

1. Defined a new error type.
2. Added impls for `Error`, `Display` and two for `From`.

The big downside here is that our program didn't improve a whole lot. There is quite a bit of overhead to representing errors with `enum`s, especially in short programs like this.

One useful aspect of using a custom error type like we've done here is that the `main` function can now choose to handle errors differently. Previously, with `Box<Error>`, it didn't have much of a choice: just print the message. We're still doing that here, but what if we wanted to, say, add a `--quiet` flag? The `--quiet` flag should silence any verbose output.

Right now, if the program doesn't find a match, it will output a message saying so. This can be a little clumsy, especially if you intend for the program to be used in shell scripts.

So let's start by adding the flags. Like before, we need to tweak the usage string and add a flag to the Option variable. Once we've done that, Getopts does the rest:

```
...
let mut opts = Options::new();
opts.optopt("f", "file", "Choose an input file, instead of using STDIN.", "NAME");
opts.optflag("h", "help", "Show this usage message.");
opts.optflag("q", "quiet", "Silences errors and warnings.");
...
```

Now we just need to implement our "quiet" functionality. This requires us to tweak the case analysis in `main`:

```
match search(&args.arg_data_path, &args.arg_city) {
```

```
    Err(CliError::NotFound) if args.flag_quiet => process::exit(1),
    Err(err) => panic!("{}", err),
    Ok(pops) => for pop in pops {
        println!("{}, {}: {:?}", pop.city, pop.country, pop.count);
    }
}
```

Certainly, we don't want to be quiet if there was an IO error or if the data failed to parse. Therefore, we use case analysis to check if the error type is `NotFound` *and* if `--quiet` has been enabled. If the search failed, we still quit with an exit code (following `grep`'s convention).

If we had stuck with `Box<Error>`, then it would be pretty tricky to implement the `--quiet` functionality.

This pretty much sums up our case study. From here, you should be ready to go out into the world and write your own programs and libraries with proper error handling.

3.7.6 The Short Story

Since this chapter is long, it is useful to have a quick summary for error handling in Rust. These are some good "rules of thumb." They are emphatically *not* commandments. There are probably good reasons to break every one of these heuristics!

- If you're writing short example code that would be overburdened by error handling, it's probably just fine to use `unwrap` (whether that's `Result::unwrap`[89], `Option::unwrap`[90] or preferably `Option::expect`[91]). Consumers of your code should know to use proper error handling. (If they don't, send them here!)
- If you're writing a quick 'n' dirty program, don't feel ashamed if you use `unwrap`. Be warned: if it winds up in someone else's hands, don't be surprised if they are agitated by poor error messages!
- If you're writing a quick 'n' dirty program and feel ashamed about panicking anyway, then using either a `String` or a `Box<Error + Send + Sync>` for your error type (the `Box<Error + Send + Sync>` type is because of the available `From` impls[92]).
- Otherwise, in a program, define your own error types with appropriate `From`[93] and `Error`[94] impls to make the `try!`[95] macro more ergonomic.
- If you're writing a library and your code can produce errors, define your own error type and implement the `std::error::Error`[96] trait. Where appropriate, implement `From`[97] to make both your library code and the caller's code easier to write. (Because of Rust's coherence rules, callers will not be able to impl `From` on your error type, so your library should do it.)

[89] ../std/result/enum.Result.html#method.unwrap
[90] ../std/option/enum.Option.html#method.unwrap
[91] ../std/option/enum.Option.html#method.expect
[92] ../std/convert/trait.From.html
[93] ../std/convert/trait.From.html
[94] ../std/error/trait.Error.html
[95] ../std/macro.try!.html
[96] ../std/error/trait.Error.html
[97] ../std/convert/trait.From.html

- Learn the combinators defined on `Option`[98] and `Result`[99]. Using them exclusively can be a bit tiring at times, but I've personally found a healthy mix of `try!` and combinators to be quite appealing. `and_then`, `map` and `unwrap_or` are my favorites.

3.8 Choosing your Guarantees

One important feature of Rust is that it lets us control the costs and guarantees of a program.

There are various "wrapper type" abstractions in the Rust standard library which embody a multitude of tradeoffs between cost, ergonomics, and guarantees. Many let one choose between run time and compile time enforcement. This section will explain a few selected abstractions in detail.

Before proceeding, it is highly recommended that one reads about ownership[100] and borrowing[101] in Rust.

3.8.1 Basic pointer types

`Box<T>`

`Box<T>`[102] is an "owned" pointer, or a "box". While it can hand out references to the contained data, it is the only owner of the data. In particular, consider the following:

```
let x = Box::new(1);
let y = x;
// x no longer accessible here
```

Here, the box was *moved* into `y`. As `x` no longer owns it, the compiler will no longer allow the programmer to use `x` after this. A box can similarly be moved *out* of a function by returning it.

When a box (that hasn't been moved) goes out of scope, destructors are run. These destructors take care of deallocating the inner data.

This is a zero-cost abstraction for dynamic allocation. If you want to allocate some memory on the heap and safely pass around a pointer to that memory, this is ideal. Note that you will only be allowed to share references to this by the regular borrowing rules, checked at compile time.

`&T` and `&mut T`

These are immutable and mutable references respectively. They follow the "read-write lock" pattern, such that one may either have only one mutable reference to some

[98] `../std/option/enum.Option.html`
[99] `../std/result/enum.Result.html`
[100] `ownership.html`
[101] `references-and-borrowing.html`
[102] `../std/boxed/struct.Box.html`

data, or any number of immutable ones, but not both. This guarantee is enforced at compile time, and has no visible cost at runtime. In most cases these two pointer types suffice for sharing cheap references between sections of code.

These pointers cannot be copied in such a way that they outlive the lifetime associated with them.

`*const T` and `*mut T`

These are C-like raw pointers with no lifetime or ownership attached to them. They just point to some location in memory with no other restrictions. The only guarantee that these provide is that they cannot be dereferenced except in code marked `unsafe`.

These are useful when building safe, low cost abstractions like `Vec<T>`, but should be avoided in safe code.

`Rc<T>`

This is the first wrapper we will cover that has a runtime cost.

`Rc<T>`[103] is a reference counted pointer. In other words, this lets us have multiple "owning" pointers to the same data, and the data will be dropped (destructors will be run) when all pointers are out of scope.

Internally, it contains a shared "reference count" (also called "refcount"), which is incremented each time the `Rc` is cloned, and decremented each time one of the `Rc`s goes out of scope. The main responsibility of `Rc<T>` is to ensure that destructors are called for shared data.

The internal data here is immutable, and if a cycle of references is created, the data will be leaked. If we want data that doesn't leak when there are cycles, we need a garbage collector.

Guarantees The main guarantee provided here is that the data will not be destroyed until all references to it are out of scope.

This should be used when we wish to dynamically allocate and share some data (read-only) between various portions of your program, where it is not certain which portion will finish using the pointer last. It's a viable alternative to `&T` when `&T` is either impossible to statically check for correctness, or creates extremely unergonomic code where the programmer does not wish to spend the development cost of working with.

This pointer is *not* thread safe, and Rust will not let it be sent or shared with other threads. This lets one avoid the cost of atomics in situations where they are unnecessary.

There is a sister smart pointer to this one, `Weak<T>`. This is a non-owning, but also non-borrowed, smart pointer. It is also similar to `&T`, but it is not restricted in lifetime—a `Weak<T>` can be held on to forever. However, it is possible that an attempt to access the inner data may fail and return `None`, since this can outlive the owned `Rc`s. This is useful for cyclic data structures and other things.

[103] `../std/rc/struct.Rc.html`

Cost As far as memory goes, `Rc<T>` is a single allocation, though it will allocate two extra words (i.e. two `usize` values) as compared to a regular `Box<T>` (for "strong" and "weak" refcounts).

`Rc<T>` has the computational cost of incrementing/decrementing the refcount whenever it is cloned or goes out of scope respectively. Note that a clone will not do a deep copy, rather it will simply increment the inner reference count and return a copy of the `Rc<T>`.

3.8.2 Cell types

`Cell`s provide interior mutability. In other words, they contain data which can be manipulated even if the type cannot be obtained in a mutable form (for example, when it is behind an `&`-ptr or `Rc<T>`).

The documentation for the `cell` module has a pretty good explanation for these[104].

These types are *generally* found in struct fields, but they may be found elsewhere too.

`Cell<T>`

`Cell<T>`[105] is a type that provides zero-cost interior mutability, but only for `Copy` types. Since the compiler knows that all the data owned by the contained value is on the stack, there's no worry of leaking any data behind references (or worse!) by simply replacing the data.

It is still possible to violate your own invariants using this wrapper, so be careful when using it. If a field is wrapped in `Cell`, it's a nice indicator that the chunk of data is mutable and may not stay the same between the time you first read it and when you intend to use it.

```
use std::cell::Cell;

let x = Cell::new(1);
let y = &x;
let z = &x;
x.set(2);
y.set(3);
z.set(4);
println!("{}", x.get());
```

Note that here we were able to mutate the same value from various immutable references.

This has the same runtime cost as the following:

```
let mut x = 1;
let y = &mut x;
let z = &mut x;
```

[104] `../std/cell/`
[105] `../std/cell/struct.Cell.html`

```
x = 2;
*y = 3;
*z = 4;
println!("{}", x);
```

but it has the added benefit of actually compiling successfully.

Guarantees This relaxes the "no aliasing with mutability" restriction in places where it's unnecessary. However, this also relaxes the guarantees that the restriction provides; so if your invariants depend on data stored within `Cell`, you should be careful.

This is useful for mutating primitives and other `Copy` types when there is no easy way of doing it in line with the static rules of `&` and `&mut`.

`Cell` does not let you obtain interior references to the data, which makes it safe to freely mutate.

Cost There is no runtime cost to using `Cell<T>`, however if you are using it to wrap larger (`Copy`) structs, it might be worthwhile to instead wrap individual fields in `Cell<T>` since each write is otherwise a full copy of the struct.

`RefCell<T>`

`RefCell<T>`[106] also provides interior mutability, but isn't restricted to `Copy` types.

Instead, it has a runtime cost. `RefCell<T>` enforces the read-write lock pattern at runtime (it's like a single-threaded mutex), unlike `&T`/`&mut T` which do so at compile time. This is done by the `borrow()` and `borrow_mut()` functions, which modify an internal reference count and return smart pointers which can be dereferenced immutably and mutably respectively. The refcount is restored when the smart pointers go out of scope. With this system, we can dynamically ensure that there are never any other borrows active when a mutable borrow is active. If the programmer attempts to make such a borrow, the thread will panic.

```
use std::cell::RefCell;

let x = RefCell::new(vec![1,2,3,4]);
{
    println!("{:?}", *x.borrow())
}

{
    let mut my_ref = x.borrow_mut();
    my_ref.push(1);
}
```

Similar to `Cell`, this is mainly useful for situations where it's hard or impossible to satisfy the borrow checker. Generally we know that such mutations won't happen in a nested form, but it's good to check.

[106] `../std/cell/struct.RefCell.html`

For large, complicated programs, it becomes useful to put some things in `RefCell`s to make things simpler. For example, a lot of the maps in the `ctxt` struct[107] in the Rust compiler internals are inside this wrapper. These are only modified once (during creation, which is not right after initialization) or a couple of times in well-separated places. However, since this struct is pervasively used everywhere, juggling mutable and immutable pointers would be hard (perhaps impossible) and probably form a soup of `&`-ptrs which would be hard to extend. On the other hand, the `RefCell` provides a cheap (not zero-cost) way of safely accessing these. In the future, if someone adds some code that attempts to modify the cell when it's already borrowed, it will cause a (usually deterministic) panic which can be traced back to the offending borrow.

Similarly, in Servo's DOM there is a lot of mutation, most of which is local to a DOM type, but some of which crisscrosses the DOM and modifies various things. Using `RefCell` and `Cell` to guard all mutation lets us avoid worrying about mutability everywhere, and it simultaneously highlights the places where mutation is *actually* happening.

Note that `RefCell` should be avoided if a mostly simple solution is possible with `&` pointers.

Guarantees `RefCell` relaxes the *static* restrictions preventing aliased mutation, and replaces them with *dynamic* ones. As such the guarantees have not changed.

Cost `RefCell` does not allocate, but it contains an additional "borrow state" indicator (one word in size) along with the data.

At runtime each borrow causes a modification/check of the refcount.

3.8.3 Synchronous types

Many of the types above cannot be used in a threadsafe manner. Particularly, `Rc<T>` and `RefCell<T>`, which both use non-atomic reference counts (*atomic* reference counts are those which can be incremented from multiple threads without causing a data race), cannot be used this way. This makes them cheaper to use, but we need thread safe versions of these too. They exist, in the form of `Arc<T>` and `Mutex<T>`/`RwLock<T>`

Note that the non-threadsafe types *cannot* be sent between threads, and this is checked at compile time.

There are many useful wrappers for concurrent programming in the sync[108] module, but only the major ones will be covered below.

`Arc<T>`

`Arc<T>`[109] is just a version of `Rc<T>` that uses an atomic reference count (hence, "Arc"). This can be sent freely between threads.

[107] `../rustc/middle/ty/struct.ctxt.html`
[108] `../std/sync/index.html`
[109] `../std/sync/struct.Arc.html`

C++'s `shared_ptr` is similar to `Arc`, however in the case of C++ the inner data is always mutable. For semantics similar to that from C++, we should use `Arc<Mutex<T>`, `Arc<RwLock<T>`, or `Arc<UnsafeCell<T>`[110] (`UnsafeCell<T>` is a cell type that can be used to hold any data and has no runtime cost, but accessing it requires `unsafe` blocks). The last one should only be used if we are certain that the usage won't cause any memory unsafety. Remember that writing to a struct is not an atomic operation, and many functions like `vec.push()` can reallocate internally and cause unsafe behavior, so even monotonicity may not be enough to justify `UnsafeCell`.

Guarantees Like `Rc`, this provides the (thread safe) guarantee that the destructor for the internal data will be run when the last `Arc` goes out of scope (barring any cycles).

Cost This has the added cost of using atomics for changing the refcount (which will happen whenever it is cloned or goes out of scope). When sharing data from an `Arc` in a single thread, it is preferable to share `&` pointers whenever possible.

`Mutex<T>` and `RwLock<T>`

`Mutex<T>`[111] and `RwLock<T>`[112] provide mutual-exclusion via RAII guards (guards are objects which maintain some state, like a lock, until their destructor is called). For both of these, the mutex is opaque until we call `lock()` on it, at which point the thread will block until a lock can be acquired, and then a guard will be returned. This guard can be used to access the inner data (mutably), and the lock will be released when the guard goes out of scope.

```
{
    let guard = mutex.lock();
    // guard dereferences mutably to the inner type
    *guard += 1;
} // lock released when destructor runs
```

`RwLock` has the added benefit of being efficient for multiple reads. It is always safe to have multiple readers to shared data as long as there are no writers; and `RwLock` lets readers acquire a "read lock". Such locks can be acquired concurrently and are kept track of via a reference count. Writers must obtain a "write lock" which can only be obtained when all readers have gone out of scope.

Guarantees Both of these provide safe shared mutability across threads, however they are prone to deadlocks. Some level of additional protocol safety can be obtained via the type system.

[110] `Arc<UnsafeCell<T>` actually won't compile since `UnsafeCell<T>` isn't `Send` or `Sync`, but we can wrap it in a type and implement `Send`/`Sync` for it manually to get `Arc<Wrapper<T>` where `Wrapper` is `struct Wrapper<T>(UnsafeCell<T>)`.

[111] `../std/sync/struct.Mutex.html`

[112] `../std/sync/struct.RwLock.html`

Costs These use internal atomic-like types to maintain the locks, which are pretty costly (they can block all memory reads across processors till they're done). Waiting on these locks can also be slow when there's a lot of concurrent access happening.

3.8.4 Composition

A common gripe when reading Rust code is with types like `Rc<RefCell<Vec<T>>>` (or even more complicated compositions of such types). It's not always clear what the composition does, or why the author chose one like this (and when one should be using such a composition in one's own code)

Usually, it's a case of composing together the guarantees that you need, without paying for stuff that is unnecessary.

For example, `Rc<RefCell<T>>` is one such composition. `Rc<T>` itself can't be dereferenced mutably; because `Rc<T>` provides sharing and shared mutability can lead to unsafe behavior, so we put `RefCell<T>` inside to get dynamically verified shared mutability. Now we have shared mutable data, but it's shared in a way that there can only be one mutator (and no readers) or multiple readers.

Now, we can take this a step further, and have `Rc<RefCell<Vec<T>>>` or `Rc<Vec<RefCell<T>>>`. These are both shareable, mutable vectors, but they're not the same.

With the former, the `RefCell<T>` is wrapping the `Vec<T>`, so the `Vec<T>` in its entirety is mutable. At the same time, there can only be one mutable borrow of the whole `Vec` at a given time. This means that your code cannot simultaneously work on different elements of the vector from different `Rc` handles. However, we are able to push and pop from the `Vec<T>` at will. This is similar to an `&mut Vec<T>` with the borrow checking done at runtime.

With the latter, the borrowing is of individual elements, but the overall vector is immutable. Thus, we can independently borrow separate elements, but we cannot push or pop from the vector. This is similar to an `&mut [T]`[113], but, again, the borrow checking is at runtime.

In concurrent programs, we have a similar situation with `Arc<Mutex<T>>`, which provides shared mutability and ownership.

When reading code that uses these, go in step by step and look at the guarantees/costs provided.

When choosing a composed type, we must do the reverse; figure out which guarantees we want, and at which point of the composition we need them. For example, if there is a choice between `Vec<RefCell<T>>` and `RefCell<Vec<T>>`, we should figure out the tradeoffs as done above and pick one.

[113] `&[T]` and `&mut [T]` are *slices*; they consist of a pointer and a length and can refer to a portion of a vector or array. `&mut [T]` can have its elements mutated, however its length cannot be touched.

3.9 FFI

3.9.1 Introduction

This guide will use the snappy[114] compression/decompression library as an introduction to writing bindings for foreign code. Rust is currently unable to call directly into a C++ library, but snappy includes a C interface (documented in `snappy-c.h`[115]).

A note about libc

Many of these examples use the `libc` crate[116], which provides various type definitions for C types, among other things. If you're trying these examples yourself, you'll need to add `libc` to your `Cargo.toml`:

```
[dependencies]
libc = "0.2.0"
```

and add `extern crate libc;` to your crate root.

Calling foreign functions

The following is a minimal example of calling a foreign function which will compile if snappy is installed:

```
# #![feature(libc)]
extern crate libc;
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x);
}
```

The `extern` block is a list of function signatures in a foreign library, in this case with the platform's C ABI. The `#[link(...)]` attribute is used to instruct the linker to link against the snappy library so the symbols are resolved.

Foreign functions are assumed to be unsafe so calls to them need to be wrapped with `unsafe {}` as a promise to the compiler that everything contained within truly is safe. C libraries often expose interfaces that aren't thread-safe, and almost any function

[114] https://github.com/google/snappy
[115] https://github.com/google/snappy/blob/master/snappy-c.h
[116] https://crates.io/crates/libc

that takes a pointer argument isn't valid for all possible inputs since the pointer could be dangling, and raw pointers fall outside of Rust's safe memory model.

When declaring the argument types to a foreign function, the Rust compiler can not check if the declaration is correct, so specifying it correctly is part of keeping the binding correct at runtime.

The `extern` block can be extended to cover the entire snappy API:

```
# #![feature(libc)]
extern crate libc;
use libc::{c_int, size_t};

#[link(name = "snappy")]
extern {
    fn snappy_compress(input: *const u8,
                       input_length: size_t,
                       compressed: *mut u8,
                       compressed_length: *mut size_t) -> c_int;
    fn snappy_uncompress(compressed: *const u8,
                         compressed_length: size_t,
                         uncompressed: *mut u8,
                         uncompressed_length: *mut size_t) -> c_int;
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
    fn snappy_uncompressed_length(compressed: *const u8,
                                  compressed_length: size_t,
                                  result: *mut size_t) -> c_int;
    fn snappy_validate_compressed_buffer(compressed: *const u8,
                                         compressed_length: size_t) -> c_int;
}
# fn main() {}
```

3.9.2 Creating a safe interface

The raw C API needs to be wrapped to provide memory safety and make use of higher-level concepts like vectors. A library can choose to expose only the safe, high-level interface and hide the unsafe internal details.

Wrapping the functions which expect buffers involves using the `slice::raw` module to manipulate Rust vectors as pointers to memory. Rust's vectors are guaranteed to be a contiguous block of memory. The length is number of elements currently contained, and the capacity is the total size in elements of the allocated memory. The length is less than or equal to the capacity.

```
# #![feature(libc)]
# extern crate libc;
# use libc::{c_int, size_t};
# unsafe fn snappy_validate_compressed_buffer(_: *const u8, _: size_t) -> c_int { 0 }
# fn main() {}
pub fn validate_compressed_buffer(src: &[u8]) -> bool {
    unsafe {
```

```
        snappy_validate_compressed_buffer(src.as_ptr(), src.len() as size_t) == 0
    }
}
```

The `validate_compressed_buffer` wrapper above makes use of an `unsafe` block, but it makes the guarantee that calling it is safe for all inputs by leaving off `unsafe` from the function signature.

The `snappy_compress` and `snappy_uncompress` functions are more complex, since a buffer has to be allocated to hold the output too.

The `snappy_max_compressed_length` function can be used to allocate a vector with the maximum required capacity to hold the compressed output. The vector can then be passed to the `snappy_compress` function as an output parameter. An output parameter is also passed to retrieve the true length after compression for setting the length.

```
# #![feature(libc)]
# extern crate libc;
# use libc::{size_t, c_int};
# unsafe fn snappy_compress(a: *const u8, b: size_t, c: *mut u8,
#                           d: *mut size_t) -> c_int { 0 }
# unsafe fn snappy_max_compressed_length(a: size_t) -> size_t { a }
# fn main() {}
pub fn compress(src: &[u8]) -> Vec<u8> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen = snappy_max_compressed_length(srclen);
        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        snappy_compress(psrc, srclen, pdst, &mut dstlen);
        dst.set_len(dstlen as usize);
        dst
    }
}
```

Decompression is similar, because snappy stores the uncompressed size as part of the compression format and `snappy_uncompressed_length` will retrieve the exact buffer size required.

```
# #![feature(libc)]
# extern crate libc;
# use libc::{size_t, c_int};
# unsafe fn snappy_uncompress(compressed: *const u8,
#                             compressed_length: size_t,
#                             uncompressed: *mut u8,
#                             uncompressed_length: *mut size_t) -> c_int { 0 }
```

```
# unsafe fn snappy_uncompressed_length(compressed: *const u8,
#                                      compressed_length: size_t,
#                                      result: *mut size_t) -> c_int { 0 }
# fn main() {}
pub fn uncompress(src: &[u8]) -> Option<Vec<u8>> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen: size_t = 0;
        snappy_uncompressed_length(psrc, srclen, &mut dstlen);

        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        if snappy_uncompress(psrc, srclen, pdst, &mut dstlen) == 0 {
            dst.set_len(dstlen as usize);
            Some(dst)
        } else {
            None // SNAPPY_INVALID_INPUT
        }
    }
}
```

For reference, the examples used here are also available as a library on GitHub[117].

3.9.3 Destructors

Foreign libraries often hand off ownership of resources to the calling code. When this occurs, we must use Rust's destructors to provide safety and guarantee the release of these resources (especially in the case of panic).

For more about destructors, see the Drop trait[118].

3.9.4 Callbacks from C code to Rust functions

Some external libraries require the usage of callbacks to report back their current state or intermediate data to the caller. It is possible to pass functions defined in Rust to an external library. The requirement for this is that the callback function is marked as `extern` with the correct calling convention to make it callable from C code.

The callback function can then be sent through a registration call to the C library and afterwards be invoked from there.

A basic example is:

Rust code:

[117] https://github.com/thestinger/rust-snappy
[118] ../std/ops/trait.Drop.html

```
extern fn callback(a: i32) {
    println!("I'm called from C with value {0}", a);
}

#[link(name = "extlib")]
extern {
   fn register_callback(cb: extern fn(i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    unsafe {
        register_callback(callback);
        trigger_callback(); // Triggers the callback
    }
}
```

C code:

```
typedef void (*rust_callback)(int32_t);
rust_callback cb;

int32_t register_callback(rust_callback callback) {
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(7); // Will call callback(7) in Rust
}
```

In this example Rust's `main()` will call `trigger_callback()` in C, which would, in turn, call back to `callback()` in Rust.

Targeting callbacks to Rust objects

The former example showed how a global function can be called from C code. However it is often desired that the callback is targeted to a special Rust object. This could be the object that represents the wrapper for the respective C object.

This can be achieved by passing an raw pointer to the object down to the C library. The C library can then include the pointer to the Rust object in the notification. This will allow the callback to unsafely access the referenced Rust object.

Rust code:

```
#[repr(C)]
struct RustObject {
    a: i32,
    // other members
```

```
}

extern "C" fn callback(target: *mut RustObject, a: i32) {
    println!("I'm called from C with value {0}", a);
    unsafe {
        // Update the value in RustObject with the value received from the callback
        (*target).a = a;
    }
}

#[link(name = "extlib")]
extern {
   fn register_callback(target: *mut RustObject,
                        cb: extern fn(*mut RustObject, i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    // Create the object that will be referenced in the callback
    let mut rust_object = Box::new(RustObject { a: 5 });

    unsafe {
        register_callback(&mut *rust_object, callback);
        trigger_callback();
    }
}
```

C code:

```
typedef void (*rust_callback)(void*, int32_t);
void* cb_target;
rust_callback cb;

int32_t register_callback(void* callback_target, rust_callback callback) {
    cb_target = callback_target;
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(cb_target, 7); // Will call callback(&rustObject, 7) in Rust
}
```

Asynchronous callbacks

In the previously given examples the callbacks are invoked as a direct reaction to a function call to the external C library. The control over the current thread is switched from Rust to C to Rust for the execution of the callback, but in the end the callback is executed on the same thread that called the function which triggered the callback.

Things get more complicated when the external library spawns its own threads and invokes callbacks from there. In these cases access to Rust data structures inside the callbacks is especially unsafe and proper synchronization mechanisms must be used. Besides classical synchronization mechanisms like mutexes, one possibility in Rust is to use channels (in `std::sync::mpsc`) to forward data from the C thread that invoked the callback into a Rust thread.

If an asynchronous callback targets a special object in the Rust address space it is also absolutely necessary that no more callbacks are performed by the C library after the respective Rust object gets destroyed. This can be achieved by unregistering the callback in the object's destructor and designing the library in a way that guarantees that no callback will be performed after deregistration.

3.9.5 Linking

The `link` attribute on `extern` blocks provides the basic building block for instructing rustc how it will link to native libraries. There are two accepted forms of the link attribute today:

- `#[link(name = "foo")]`
- `#[link(name = "foo", kind = "bar")]`

In both of these cases, `foo` is the name of the native library that we're linking to, and in the second case `bar` is the type of native library that the compiler is linking to. There are currently three known types of native libraries:

- Dynamic - `#[link(name = "readline")]`
- Static - `#[link(name = "my_build_dependency", kind = "static")]`
- Frameworks - `#[link(name = "CoreFoundation", kind = "framework")]`

Note that frameworks are only available on OSX targets.

The different `kind` values are meant to differentiate how the native library participates in linkage. From a linkage perspective, the Rust compiler creates two flavors of artifacts: partial (rlib/staticlib) and final (dylib/binary). Native dynamic library and framework dependencies are propagated to the final artifact boundary, while static library dependencies are not propagated at all, because the static libraries are integrated directly into the subsequent artifact.

A few examples of how this model can be used are:

- A native build dependency. Sometimes some C/C++ glue is needed when writing some Rust code, but distribution of the C/C++ code in a library format is just a burden. In this case, the code will be archived into `libfoo.a` and then the Rust crate would declare a dependency via `#[link(name = "foo", kind = "static")]`.

Regardless of the flavor of output for the crate, the native static library will be included in the output, meaning that distribution of the native static library is not necessary.

- A normal dynamic dependency. Common system libraries (like `readline`) are available on a large number of systems, and often a static copy of these libraries cannot be found. When this dependency is included in a Rust crate, partial targets (like rlibs) will not link to the library, but when the rlib is included in a final target (like a binary), the native library will be linked in.

On OSX, frameworks behave with the same semantics as a dynamic library.

3.9.6 Unsafe blocks

Some operations, like dereferencing raw pointers or calling functions that have been marked unsafe are only allowed inside unsafe blocks. Unsafe blocks isolate unsafety and are a promise to the compiler that the unsafety does not leak out of the block.

Unsafe functions, on the other hand, advertise it to the world. An unsafe function is written like this:

```
unsafe fn kaboom(ptr: *const i32) -> i32 { *ptr }
```

This function can only be called from an `unsafe` block or another `unsafe` function.

3.9.7 Accessing foreign globals

Foreign APIs often export a global variable which could do something like track global state. In order to access these variables, you declare them in `extern` blocks with the `static` keyword:

```
# #![feature(libc)]
extern crate libc;

#[link(name = "readline")]
extern {
    static rl_readline_version: libc::c_int;
}

fn main() {
    println!("You have readline version {} installed.",
             rl_readline_version as i32);
}
```

Alternatively, you may need to alter global state provided by a foreign interface. To do this, statics can be declared with `mut` so we can mutate them.

```
# #![feature(libc)]
extern crate libc;

use std::ffi::CString;
use std::ptr;
```

```
#[link(name = ”readline”)]
extern {
    static mut rl_prompt: *const libc::c_char;
}

fn main() {
    let prompt = CString::new(”[my-awesome-shell] $”).unwrap();
    unsafe {
        rl_prompt = prompt.as_ptr();

        println!(”{:?}”, rl_prompt);

        rl_prompt = ptr::null();
    }
}
```

Note that all interaction with a `static mut` is unsafe, both reading and writing. Dealing with global mutable state requires a great deal of care.

3.9.8 Foreign calling conventions

Most foreign code exposes a C ABI, and Rust uses the platform’s C calling convention by default when calling foreign functions. Some foreign functions, most notably the Windows API, use other calling conventions. Rust provides a way to tell the compiler which convention to use:

```
# #![feature(libc)]
extern crate libc;

#[cfg(all(target_os = ”win32”, target_arch = ”x86”))]
#[link(name = ”kernel32”)]
#[allow(non_snake_case)]
extern ”stdcall” {
    fn SetEnvironmentVariableA(n: *const u8, v: *const u8) -> libc::c_int;
}
# fn main() { }
```

This applies to the entire `extern` block. The list of supported ABI constraints are:

- `stdcall`
- `aapcs`
- `cdecl`
- `fastcall`
- `Rust`
- `rust-intrinsic`
- `system`
- `C`

- `win64`

Most of the abis in this list are self-explanatory, but the `system` abi may seem a little odd. This constraint selects whatever the appropriate ABI is for interoperating with the target's libraries. For example, on win32 with a x86 architecture, this means that the abi used would be `stdcall`. On x86_64, however, windows uses the `C` calling convention, so `C` would be used. This means that in our previous example, we could have used `extern "system" { ... }` to define a block for all windows systems, not just x86 ones.

3.9.9 Interoperability with foreign code

Rust guarantees that the layout of a `struct` is compatible with the platform's representation in C only if the `#[repr(C)]` attribute is applied to it. `#[repr(C, packed)]` can be used to lay out struct members without padding. `#[repr(C)]` can also be applied to an enum.

Rust's owned boxes (`Box<T>`) use non-nullable pointers as handles which point to the contained object. However, they should not be manually created because they are managed by internal allocators. References can safely be assumed to be non-nullable pointers directly to the type. However, breaking the borrow checking or mutability rules is not guaranteed to be safe, so prefer using raw pointers (`*`) if that's needed because the compiler can't make as many assumptions about them.

Vectors and strings share the same basic memory layout, and utilities are available in the `vec` and `str` modules for working with C APIs. However, strings are not terminated with `\0`. If you need a NUL-terminated string for interoperability with C, you should use the `CString` type in the `std::ffi` module.

The `libc` crate on crates.io[119] includes type aliases and function definitions for the C standard library in the `libc` module, and Rust links against `libc` and `libm` by default.

3.9.10 The "nullable pointer optimization"

Certain types are defined to not be `null`. This includes references (`&T`, `&mut T`), boxes (`Box<T>`), and function pointers (`extern "abi" fn()`). When interfacing with C, pointers that might be null are often used. As a special case, a generic `enum` that contains exactly two variants, one of which contains no data and the other containing a single field, is eligible for the "nullable pointer optimization". When such an enum is instantiated with one of the non-nullable types, it is represented as a single pointer, and the non-data variant is represented as the null pointer. So `Option<extern "C" fn(c_int) -> c_int>` is how one represents a nullable function pointer using the C ABI.

3.9.11 Calling Rust code from C

You may wish to compile Rust code in a way so that it can be called from C. This is fairly easy, but requires a few things:

[119] `https://crates.io/crates/libc`

```
#[no_mangle]
pub extern fn hello_rust() -> *const u8 {
    "Hello, world!\0".as_ptr()
}
# fn main() {}
```

The `extern` makes this function adhere to the C calling convention, as discussed above in "Foreign Calling Conventions[120]". The `no_mangle` attribute turns off Rust's name mangling, so that it is easier to link to.

3.9.12 FFI and panics

It's important to be mindful of `panic!`s when working with FFI. A `panic!` across an FFI boundary is undefined behavior. If you're writing code that may panic, you should run it in another thread, so that the panic doesn't bubble up to C:

```
use std::thread;

#[no_mangle]
pub extern fn oh_no() -> i32 {
    let h = thread::spawn(|| {
        panic!("Oops!");
    });

    match h.join() {
        Ok(_) => 1,
        Err(_) => 0,
    }
}
# fn main() {}
```

3.9.13 Representing opaque structs

Sometimes, a C library wants to provide a pointer to something, but not let you know the internal details of the thing it wants. The simplest way is to use a `void *` argument:

```
void foo(void *arg);
void bar(void *arg);
```

We can represent this in Rust with the `c_void` type:

```
# #![feature(libc)]
extern crate libc;

extern "C" {
    pub fn foo(arg: *mut libc::c_void);
```

[120] `ffi.html#foreign-calling-conventions`

```
    pub fn bar(arg: *mut libc::c_void);
}
# fn main() {}
```

This is a perfectly valid way of handling the situation. However, we can do a bit better. To solve this, some C libraries will instead create a `struct`, where the details and memory layout of the struct are private. This gives some amount of type safety. These structures are called 'opaque'. Here's an example, in C:

```
struct Foo; /* Foo is a structure, but its contents are not part of the public interface */
struct Bar;
void foo(struct Foo *arg);
void bar(struct Bar *arg);
```

To do this in Rust, let's create our own opaque types with `enum`:

```
pub enum Foo {}
pub enum Bar {}

extern "C" {
    pub fn foo(arg: *mut Foo);
    pub fn bar(arg: *mut Bar);
}
# fn main() {}
```

By using an `enum` with no variants, we create an opaque type that we can't instantiate, as it has no variants. But because our `Foo` and `Bar` types are different, we'll get type safety between the two of them, so we cannot accidentally pass a pointer to `Foo` to `bar()`.

3.10 Borrow and AsRef

The `Borrow`[121] and `AsRef`[122] traits are very similar, but different. Here's a quick refresher on what these two traits mean.

3.10.1 Borrow

The `Borrow` trait is used when you're writing a datastructure, and you want to use either an owned or borrowed type as synonymous for some purpose.

For example, `HashMap`[123] has a `get` method[124] which uses `Borrow`:

```
fn get<Q: ?Sized>(&self, k: &Q) -> Option<&V>
    where K: Borrow<Q>,
          Q: Hash + Eq
```

[121] ../std/borrow/trait.Borrow.html
[122] ../std/convert/trait.AsRef.html
[123] ../std/collections/struct.HashMap.html
[124] ../std/collections/struct.HashMap.html#method.get

This signature is pretty complicated. The `K` parameter is what we're interested in here. It refers to a parameter of the `HashMap` itself:

```
struct HashMap<K, V, S = RandomState> {
```

The `K` parameter is the type of *key* the `HashMap` uses. So, looking at the signature of `get()` again, we can use `get()` when the key implements `Borrow<Q>`. That way, we can make a `HashMap` which uses `String` keys, but use `&str`s when we're searching:

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

let mut map = HashMap::new();
map.insert("Foo".to_string(), 42);

assert_eq!(map.get("Foo"), Some(&42));
```

This is because the standard library has `impl Borrow<str> for String`.

For most types, when you want to take an owned or borrowed type, a `&T` is enough. But one area where `Borrow` is effective is when there's more than one kind of borrowed value. This is especially true of references and slices: you can have both an `&T` or a `&mut T`. If we wanted to accept both of these types, `Borrow` is up for it:

```
use std::borrow::Borrow;
use std::fmt::Display;

fn foo<T: Borrow<i32> + Display>(a: T) {
    println!("a is borrowed: {}", a);
}

let mut i = 5;

foo(&i);
foo(&mut i);
```

This will print out `a is borrowed: 5` twice.

3.10.2 AsRef

The `AsRef` trait is a conversion trait. It's used for converting some value to a reference in generic code. Like this:

```
let s = "Hello".to_string();

fn foo<T: AsRef<str>>(s: T) {
    let slice = s.as_ref();
}
```

3.10.3 Which should I use?

We can see how they're kind of the same: they both deal with owned and borrowed versions of some type. However, they're a bit different.

Choose `Borrow` when you want to abstract over different kinds of borrowing, or when you're building a datastructure that treats owned and borrowed values in equivalent ways, such as hashing and comparison.

Choose `AsRef` when you want to convert something to a reference directly, and you're writing generic code.

3.11 Release Channels

The Rust project uses a concept called 'release channels' to manage releases. It's important to understand this process to choose which version of Rust your project should use.

3.11.1 Overview

There are three channels for Rust releases:

- Nightly
- Beta
- Stable

New nightly releases are created once a day. Every six weeks, the latest nightly release is promoted to 'Beta'. At that point, it will only receive patches to fix serious errors. Six weeks later, the beta is promoted to 'Stable', and becomes the next release of `1.x`.

This process happens in parallel. So every six weeks, on the same day, nightly goes to beta, beta goes to stable. When `1.x` is released, at the same time, `1.(x + 1)-beta` is released, and the nightly becomes the first version of `1.(x + 2)-nightly`.

3.11.2 Choosing a version

Generally speaking, unless you have a specific reason, you should be using the stable release channel. These releases are intended for a general audience.

However, depending on your interest in Rust, you may choose to use nightly instead. The basic tradeoff is this: in the nightly channel, you can use unstable, new Rust features. However, unstable features are subject to change, and so any new nightly release may break your code. If you use the stable release, you cannot use experimental features, but the next release of Rust will not cause significant issues through breaking changes.

3.11.3 Helping the ecosystem through CI

What about beta? We encourage all Rust users who use the stable release channel to also test against the beta channel in their continuous integration systems. This will help alert the team in case there's an accidental regression.

Additionally, testing against nightly can catch regressions even sooner, and so if you don't mind a third build, we'd appreciate testing against all channels.

As an example, many Rust programmers use Travis[125] to test their crates, which is free for open source projects. Travis supports Rust directly[126], and you can use a `.travis.yml` file like this to test on all channels:

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

matrix:
  allow_failures:
    - rust: nightly
```

With this configuration, Travis will test all three channels, but if something breaks on nightly, it won't fail your build. A similar configuration is recommended for any CI system, check the documentation of the one you're using for more details.

[125] `https://travis-ci.org/`
[126] `http://docs.travis-ci.com/user/languages/rust/`

Chapter 4

Syntax and Semantics

This section breaks Rust down into small chunks, one for each concept.

If you'd like to learn Rust from the bottom up, reading this in order is a great way to do that.

These sections also form a reference for each concept, so if you're reading another tutorial and find something confusing, you can find it explained somewhere in here.

4.1 Variable Bindings

Virtually every non-'Hello World' Rust program uses *variable bindings*. They bind some value to a name, so it can be used later. `let` is used to introduce a binding, just like this:

```
fn main() {
    let x = 5;
}
```

Putting `fn main() {` in each example is a bit tedious, so we'll leave that out in the future. If you're following along, make sure to edit your `main()` function, rather than leaving it off. Otherwise, you'll get an error.

4.1.1 Patterns

In many languages, a variable binding would be called a *variable*, but Rust's variable bindings have a few tricks up their sleeves. For example the left-hand side of a `let` expression is a 'pattern[1]', not just a variable name. This means we can do things like:

```
let (x, y) = (1, 2);
```

After this expression is evaluated, `x` will be one, and `y` will be two. Patterns are really powerful, and have their own section[2] in the book. We don't need those features for now, so we'll just keep this in the back of our minds as we go forward.

[1] `patterns.html`
[2] `patterns.html`

4.1.2 Type annotations

Rust is a statically typed language, which means that we specify our types up front, and they're checked at compile time. So why does our first example compile? Well, Rust has this thing called 'type inference'. If it can figure out what the type of something is, Rust doesn't require you to actually type it out.

We can add the type if we want to, though. Types come after a colon (`:`):

```
let x: i32 = 5;
```

If I asked you to read this out loud to the rest of the class, you'd say "`x` is a binding with the type `i32` and the value `five`."

In this case we chose to represent `x` as a 32-bit signed integer. Rust has many different primitive integer types. They begin with `i` for signed integers and `u` for unsigned integers. The possible integer sizes are 8, 16, 32, and 64 bits.

In future examples, we may annotate the type in a comment. The examples will look like this:

```
fn main() {
    let x = 5; // x: i32
}
```

Note the similarities between this annotation and the syntax you use with `let`. Including these kinds of comments is not idiomatic Rust, but we'll occasionally include them to help you understand what the types that Rust infers are.

4.1.3 Mutability

By default, bindings are *immutable*. This code will not compile:

```
let x = 5;
x = 10;
```

It will give you this error:

```
error: re-assignment of immutable variable ‘x‘
     x = 10;
     ^~~~~~~
```

If you want a binding to be mutable, you can use `mut`:

```
let mut x = 5; // mut x: i32
x = 10;
```

There is no single reason that bindings are immutable by default, but we can think about it through one of Rust's primary focuses: safety. If you forget to say `mut`, the compiler will catch it, and let you know that you have mutated something you may

not have intended to mutate. If bindings were mutable by default, the compiler would not be able to tell you this. If you *did* intend mutation, then the solution is quite easy: add `mut`.

There are other good reasons to avoid mutable state when possible, but they're out of the scope of this guide. In general, you can often avoid explicit mutation, and so it is preferable in Rust. That said, sometimes, mutation is what you need, so it's not verboten.

4.1.4 Initializing bindings

Rust variable bindings have one more aspect that differs from other languages: bindings are required to be initialized with a value before you're allowed to use them.

Let's try it out. Change your `src/main.rs` file to look like this:

```
fn main() {
    let x: i32;

    println!("Hello world!");
}
```

You can use `cargo build` on the command line to build it. You'll get a warning, but it will still print "Hello, world!":

```
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:2:9: 2:10 warning: unused variable: `x`, #[warn(unused_variable)]
   on by default
src/main.rs:2     let x: i32;
                      ^
```

Rust warns us that we never use the variable binding, but since we never use it, no harm, no foul. Things change if we try to actually use this `x`, however. Let's do that. Change your program to look like this:

```
fn main() {
    let x: i32;

    println!("The value of x is: {}", x);
}
```

And try to build it. You'll get an error:

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:4:39: 4:40 error: use of possibly uninitialized variable: `x`
src/main.rs:4     println!("The value of x is: {}", x);
                                                    ^
note: in expansion of format_args!
<std macros>:2:23: 2:77 note: expansion site
```

```
<std macros>:1:1: 3:2 note: in expansion of println!
src/main.rs:4:5: 4:42 note: expansion site
error: aborting due to previous error
Could not compile ‘hello_world‘.
```

Rust will not let us use a value that has not been initialized. Next, let’s talk about this stuff we’ve added to `println!`.

If you include two curly braces (`{}`, some call them moustaches...) in your string to print, Rust will interpret this as a request to interpolate some sort of value. *String interpolation* is a computer science term that means “stick in the middle of a string.” We add a comma, and then `x`, to indicate that we want `x` to be the value we’re interpolating. The comma is used to separate arguments we pass to functions and macros, if you’re passing more than one.

When you just use the curly braces, Rust will attempt to display the value in a meaningful way by checking out its type. If you want to specify the format in a more detailed manner, there are a wide number of options available[3]. For now, we’ll just stick to the default: integers aren’t very complicated to print.

4.1.5 Scope and shadowing

Let’s get back to bindings. Variable bindings have a scope - they are constrained to live in a block they were defined in. A block is a collection of statements enclosed by `{` and `}`. Function definitions are also blocks! In the following example we define two variable bindings, `x` and `y`, which live in different blocks. `x` can be accessed from inside the `fn main() {}` block, while `y` can be accessed only from inside the inner block:

```
fn main() {
    let x: i32 = 17;
    {
        let y: i32 = 3;
        println!(”The value of x is {} and value of y is {}”, x, y);
    }
    println!(”The value of x is {} and value of y is {}”, x, y); // This won’t work
}
```

The first `println!` would print “The value of x is 17 and the value of y is 3”, but this example cannot be compiled successfully, because the second `println!` cannot access the value of `y`, since it is not in scope anymore. Instead we get this error:

```
$ cargo build
   Compiling hello v0.1.0 (file:///home/you/projects/hello_world)
main.rs:7:62: 7:63 error: unresolved name ‘y‘. Did you mean ‘x‘? [E0425]
main.rs:7     println!(”The value of x is {} and value of y is {}”, x, y); // This won’t work
                                                                        ^
note: in expansion of format_args!
```

[3] `../std/fmt/index.html`

```
<std macros>:2:25: 2:56 note: expansion site
<std macros>:1:1: 2:62 note: in expansion of print!
<std macros>:3:1: 3:54 note: expansion site
<std macros>:1:1: 3:58 note: in expansion of println!
main.rs:7:5: 7:65 note: expansion site
main.rs:7:62: 7:63 help: run ‘rustc --explain E0425‘ to see a detailed explanation
error: aborting due to previous error
Could not compile ‘hello‘.

To learn more, run the command again with --verbose.
```

Additionally, variable bindings can be shadowed. This means that a later variable binding with the same name as another binding, that's currently in scope, will override the previous binding.

```
let x: i32 = 8;
{
    println!(”{}”, x); // Prints ”8”
    let x = 12;
    println!(”{}”, x); // Prints ”12”
}
println!(”{}”, x); // Prints ”8”
let x =  42;
println!(”{}”, x); // Prints ”42”
```

Shadowing and mutable bindings may appear as two sides of the same coin, but they are two distinct concepts that can't always be used interchangeably. For one, shadowing enables us to rebind a name to a value of a different type. It is also possible to change the mutability of a binding.

```
let mut x: i32 = 1;
x = 7;
let x = x; // x is now immutable and is bound to 7

let y = 4;
let y = ”I can also be bound to text!”; // y is now of a different type
```

4.2 Functions

Every Rust program has at least one function, the `main` function:

```
fn main() {
}
```

This is the simplest possible function declaration. As we mentioned before, `fn` says ‘this is a function’, followed by the name, some parentheses because this function takes no arguments, and then some curly braces to indicate the body. Here's a function named `foo`:

```
fn foo() {
}
```

So, what about taking arguments? Here's a function that prints a number:

```
fn print_number(x: i32) {
    println!("x is: {}", x);
}
```

Here's a complete program that uses `print_number`:

```
fn main() {
    print_number(5);
}

fn print_number(x: i32) {
    println!("x is: {}", x);
}
```

As you can see, function arguments work very similar to `let` declarations: you add a type to the argument name, after a colon.

Here's a complete program that adds two numbers together and prints them:

```
fn main() {
    print_sum(5, 6);
}

fn print_sum(x: i32, y: i32) {
    println!("sum is: {}", x + y);
}
```

You separate arguments with a comma, both when you call the function, as well as when you declare it.

Unlike `let`, you *must* declare the types of function arguments. This does not work:

```
fn print_sum(x, y) {
    println!("sum is: {}", x + y);
}
```

You get this error:

```
expected one of ‘¡, ‘:‘, or ‘@‘, found ‘)‘
fn print_number(x, y) {
```

This is a deliberate design decision. While full-program inference is possible, languages which have it, like Haskell, often suggest that documenting your types explicitly is a best-practice. We agree that forcing functions to declare types while allowing for inference inside of function bodies is a wonderful sweet spot between full inference and no inference.

What about returning a value? Here's a function that adds one to an integer:

```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

Rust functions return exactly one value, and you declare the type after an 'arrow', which is a dash (`-`) followed by a greater-than sign (`>`). The last line of a function determines what it returns. You'll note the lack of a semicolon here. If we added it in:

```
fn add_one(x: i32) -> i32 {
    x + 1;
}
```

We would get an error:

```
error: not all control paths return a value
fn add_one(x: i32) -> i32 {
     x + 1;
}

help: consider removing this semicolon:
     x + 1;
          ^
```

This reveals two interesting things about Rust: it is an expression-based language, and semicolons are different from semicolons in other 'curly brace and semicolon'-based languages. These two things are related.

Expressions vs. Statements

Rust is primarily an expression-based language. There are only two kinds of statements, and everything else is an expression.

So what's the difference? Expressions return a value, and statements do not. That's why we end up with 'not all control paths return a value' here: the statement `x + 1;` doesn't return a value. There are two kinds of statements in Rust: 'declaration statements' and 'expression statements'. Everything else is an expression. Let's talk about declaration statements first.

In some languages, variable bindings can be written as expressions, not just statements. Like Ruby:

```
x = y = 5
```

In Rust, however, using `let` to introduce a binding is *not* an expression. The following will produce a compile-time error:

```
let x = (let y = 5); // expected identifier, found keyword 'let'
```

The compiler is telling us here that it was expecting to see the beginning of an expression, and a `let` can only begin a statement, not an expression.

Note that assigning to an already-bound variable (e.g. `y = 5`) is still an expression, although its value is not particularly useful. Unlike other languages where an assignment evaluates to the assigned value (e.g. `5` in the previous example), in Rust the value of an assignment is an empty tuple `()` because the assigned value can have just one owner[4], and any other returned value would be too surprising:

```
let mut y = 5;

let x = (y = 6);  // x has the value `()`, not `6`
```

The second kind of statement in Rust is the *expression statement*. Its purpose is to turn any expression into a statement. In practical terms, Rust's grammar expects statements to follow other statements. This means that you use semicolons to separate expressions from each other. This means that Rust looks a lot like most other languages that require you to use semicolons at the end of every line, and you will see semicolons at the end of almost every line of Rust code you see.

What is this exception that makes us say "almost"? You saw it already, in this code:

```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

Our function claims to return an `i32`, but with a semicolon, it would return `()` instead. Rust realizes this probably isn't what we want, and suggests removing the semicolon in the error we saw before.

Early returns

But what about early returns? Rust does have a keyword for that, `return`:

```
fn foo(x: i32) -> i32 {
    return x;

    // we never run this code!
    x + 1
}
```

Using a `return` as the last line of a function works, but is considered poor style:

```
fn foo(x: i32) -> i32 {
    return x + 1;
}
```

The previous definition without `return` may look a bit strange if you haven't worked in an expression-based language before, but it becomes intuitive over time.

[4] `ownership.html`

Diverging functions

Rust has some special syntax for ‘diverging functions’, which are functions that do not return:

```
fn diverges() -> ! {
    panic!(”This function never returns!”);
}
```

`panic!` is a macro, similar to `println!()` that we’ve already seen. Unlike `println!()`, `panic!()` causes the current thread of execution to crash with the given message. Because this function will cause a crash, it will never return, and so it has the type ‘`!`’, which is read ‘diverges’.

If you add a main function that calls `diverges()` and run it, you’ll get some output that looks like this:

```
thread ‘<main>’ panicked at ‘This function never returns!’, hello.rs:2
```

If you want more information, you can get a backtrace by setting the `RUST_BACKTRACE` environment variable:

```
$ RUST_BACKTRACE=1 ./diverges
thread ’<main>’ panicked at ’This function never returns!’, hello.rs:2
stack backtrace:
   1:     0x7f402773a829 - sys::backtrace::write::h0942de78b6c02817K8r
   2:     0x7f402773d7fc - panicking::on_panic::h3f23f9d0b5f4c91bu9w
   3:     0x7f402773960e - rt::unwind::begin_unwind_inner::h2844b8c5e81e79558Bw
   4:     0x7f4027738893 - rt::unwind::begin_unwind::h4375279447423903650
   5:     0x7f4027738809 - diverges::h2266b4c4b850236beaa
   6:     0x7f40277389e5 - main::h19bb1149c2f00ecfBaa
   7:     0x7f402773f514 - rt::unwind::try::try_fn::h13186883479104382231
   8:     0x7f402773d1d8 - __rust_try
   9:     0x7f402773f201 - rt::lang_start::ha172a3ce74bb453aK5w
  10:     0x7f4027738a19 - main
  11:     0x7f402694ab44 - __libc_start_main
  12:     0x7f40277386c8 - <unknown>
  13:                0x0 - <unknown>
```

`RUST_BACKTRACE` also works with Cargo’s `run` command:

```
$ RUST_BACKTRACE=1 cargo run
     Running ‘target/debug/diverges‘
thread ’<main>’ panicked at ’This function never returns!’, hello.rs:2
stack backtrace:
   1:     0x7f402773a829 - sys::backtrace::write::h0942de78b6c02817K8r
   2:     0x7f402773d7fc - panicking::on_panic::h3f23f9d0b5f4c91bu9w
   3:     0x7f402773960e - rt::unwind::begin_unwind_inner::h2844b8c5e81e79558Bw
   4:     0x7f4027738893 - rt::unwind::begin_unwind::h4375279447423903650
```

```
   5:     0x7f4027738809 - diverges::h2266b4c4b850236beaa
   6:     0x7f40277389e5 - main::h19bb1149c2f00ecfBaa
   7:     0x7f402773f514 - rt::unwind::try::try_fn::h13186883479104382231
   8:     0x7f402773d1d8 - __rust_try
   9:     0x7f402773f201 - rt::lang_start::ha172a3ce74bb453aK5w
  10:     0x7f4027738a19 - main
  11:     0x7f402694ab44 - __libc_start_main
  12:     0x7f40277386c8 - <unknown>
  13:                0x0 - <unknown>
```

A diverging function can be used as any type:

```
# fn diverges() -> ! {
#    panic!("This function never returns!");
# }
let x: i32 = diverges();
let x: String = diverges();
```

Function pointers

We can also create variable bindings which point to functions:

```
let f: fn(i32) -> i32;
```

`f` is a variable binding which points to a function that takes an `i32` as an argument and returns an `i32`. For example:

```
fn plus_one(i: i32) -> i32 {
    i + 1
}

// without type inference
let f: fn(i32) -> i32 = plus_one;

// with type inference
let f = plus_one;
```

We can then use `f` to call the function:

```
# fn plus_one(i: i32) -> i32 { i + 1 }
# let f = plus_one;
let six = f(5);
```

4.3 Primitive Types

The Rust language has a number of types that are considered 'primitive'. This means that they're built-in to the language. Rust is structured in such a way that the standard library also provides a number of useful types built on top of these ones, as well, but these are the most primitive.

4.3.1 Booleans

Rust has a built in boolean type, named `bool`. It has two values, `true` and `false`:

```
let x = true;

let y: bool = false;
```

A common use of booleans is in `if` conditionals[5].
You can find more documentation for `bool`s in the standard library documentation[6].

4.3.2 `char`

The `char` type represents a single Unicode scalar value. You can create `char`s with a single tick: (')

```
let x = 'x';
let two_hearts = '';
```

Unlike some other languages, this means that Rust's `char` is not a single byte, but four.
You can find more documentation for `char`s in the standard library documentation[7].

4.3.3 Numeric types

Rust has a variety of numeric types in a few categories: signed and unsigned, fixed and variable, floating-point and integer.
These types consist of two parts: the category, and the size. For example, `u16` is an unsigned type with sixteen bits of size. More bits lets you have bigger numbers.
If a number literal has nothing to cause its type to be inferred, it defaults:

```
let x = 42; // x has type i32

let y = 1.0; // y has type f64
```

Here's a list of the different numeric types, with links to their documentation in the standard library:

- i8[8]
- i16[9]
- i32[10]
- i64[11]

[5] `if.html`
[6] `../std/primitive.bool.html`
[7] `../std/primitive.char.html`
[8] `../std/primitive.i8.html`
[9] `../std/primitive.i16.html`
[10] `../std/primitive.i32.html`
[11] `../std/primitive.i64.html`

- u8[12]
- u16[13]
- u32[14]
- u64[15]
- isize[16]
- usize[17]
- f32[18]
- f64[19]

Let's go over them by category:

Signed and Unsigned

Integer types come in two varieties: signed and unsigned. To understand the difference, let's consider a number with four bits of size. A signed, four-bit number would let you store numbers from `-8` to `+7`. Signed numbers use "two's complement representation". An unsigned four bit number, since it does not need to store negatives, can store values from `0` to `+15`.

Unsigned types use a `u` for their category, and signed types use `i`. The `i` is for 'integer'. So `u8` is an eight-bit unsigned number, and `i8` is an eight-bit signed number.

Fixed size types

Fixed size types have a specific number of bits in their representation. Valid bit sizes are `8`, `16`, `32`, and `64`. So, `u32` is an unsigned, 32-bit integer, and `i64` is a signed, 64-bit integer.

Variable sized types

Rust also provides types whose size depends on the size of a pointer of the underlying machine. These types have 'size' as the category, and come in signed and unsigned varieties. This makes for two types: `isize` and `usize`.

Floating-point types

Rust also has two floating point types: `f32` and `f64`. These correspond to IEEE-754 single and double precision numbers.

[12] ../std/primitive.u8.html
[13] ../std/primitive.u16.html
[14] ../std/primitive.u32.html
[15] ../std/primitive.u64.html
[16] ../std/primitive.isize.html
[17] ../std/primitive.usize.html
[18] ../std/primitive.f32.html
[19] ../std/primitive.f64.html

4.3.4 Arrays

Like many programming languages, Rust has list types to represent a sequence of things. The most basic is the *array*, a fixed-size list of elements of the same type. By default, arrays are immutable.

```
let a = [1, 2, 3]; // a: [i32; 3]
let mut m = [1, 2, 3]; // m: [i32; 3]
```

Arrays have type `[T; N]`. We'll talk about this `T` notation in the generics section[20]. The `N` is a compile-time constant, for the length of the array.

There's a shorthand for initializing each element of an array to the same value. In this example, each element of `a` will be initialized to `0`:

```
let a = [0; 20]; // a: [i32; 20]
```

You can get the number of elements in an array `a` with `a.len()`:

```
let a = [1, 2, 3];

println!("a has {} elements", a.len());
```

You can access a particular element of an array with *subscript notation*:

```
let names = ["Graydon", "Brian", "Niko"]; // names: [&str; 3]

println!("The second name is: {}", names[1]);
```

Subscripts start at zero, like in most programming languages, so the first name is `names[0]` and the second name is `names[1]`. The above example prints `The second name is: Brian`. If you try to use a subscript that is not in the array, you will get an error: array access is bounds-checked at run-time. Such errant access is the source of many bugs in other systems programming languages.

You can find more documentation for `array`s in the standard library documentation[21].

4.3.5 Slices

A 'slice' is a reference to (or "view" into) another data structure. They are useful for allowing safe, efficient access to a portion of an array without copying. For example, you might want to reference just one line of a file read into memory. By nature, a slice is not created directly, but from an existing variable binding. Slices have a defined length, can be mutable or immutable.

[20] `generics.html`
[21] `../std/primitive.array.html`

Slicing syntax

You can use a combo of `&` and `[]` to create a slice from various things. The `&` indicates that slices are similar to references, and the `[]`s, with a range, let you define the length of the slice:

```
let a = [0, 1, 2, 3, 4];
let complete = &a[..]; // A slice containing all of the elements in a
let middle = &a[1..4]; // A slice of a: just the elements 1, 2, and 3
```

Slices have type `&[T]`. We'll talk about that `T` when we cover generics[22].
You can find more documentation for slices in the standard library documentation[23].

4.3.6 `str`

Rust's `str` type is the most primitive string type. As an unsized type[24], it's not very useful by itself, but becomes useful when placed behind a reference, like `&str`[25]. As such, we'll just leave it at that.
You can find more documentation for `str` in the standard library documentation[26].

4.3.7 Tuples

A tuple is an ordered list of fixed size. Like this:

```
let x = (1, "hello");
```

The parentheses and commas form this two-length tuple. Here's the same code, but with the type annotated:

```
let x: (i32, &str) = (1, "hello");
```

As you can see, the type of a tuple looks just like the tuple, but with each position having a type name rather than the value. Careful readers will also note that tuples are heterogeneous: we have an `i32` and a `&str` in this tuple. In systems programming languages, strings are a bit more complex than in other languages. For now, just read `&str` as a *string slice*, and we'll learn more soon.
You can assign one tuple into another, if they have the same contained types and arity[27]. Tuples have the same arity when they have the same length.

```
let mut x = (1, 2); // x: (i32, i32)
let y = (2, 3); // y: (i32, i32)

x = y;
```

[22] `generics.html`
[23] `../std/primitive.slice.html`
[24] `unsized-types.html`
[25] `strings.html`
[26] `../std/primitive.str.html`
[27] `glossary.html#arity`

You can access the fields in a tuple through a *destructuring let*. Here's an example:

```
let (x, y, z) = (1, 2, 3);

println!("x is {}", x);
```

Remember before[28] when I said the left-hand side of a `let` statement was more powerful than just assigning a binding? Here we are. We can put a pattern on the left-hand side of the `let`, and if it matches up to the right-hand side, we can assign multiple bindings at once. In this case, `let` "destructures" or "breaks up" the tuple, and assigns the bits to three bindings.

This pattern is very powerful, and we'll see it repeated more later.

You can disambiguate a single-element tuple from a value in parentheses with a comma:

```
(0,); // single-element tuple
(0); // zero in parentheses
```

Tuple Indexing

You can also access fields of a tuple with indexing syntax:

```
let tuple = (1, 2, 3);

let x = tuple.0;
let y = tuple.1;
let z = tuple.2;

println!("x is {}", x);
```

Like array indexing, it starts at zero, but unlike array indexing, it uses a `.`, rather than `[]`s.

You can find more documentation for tuples in the standard library documentation[29].

4.3.8 Functions

Functions also have a type! They look like this:

```
fn foo(x: i32) -> i32 { x }

let x: fn(i32) -> i32 = foo;
```

In this case, `x` is a 'function pointer' to a function that takes an `i32` and returns an `i32`.

[28] `variable-bindings.html`
[29] `../std/primitive.tuple.html`

4.4 Comments

Now that we have some functions, it's a good idea to learn about comments. Comments are notes that you leave to other programmers to help explain things about your code. The compiler mostly ignores them.

Rust has two kinds of comments that you should care about: *line comments* and *doc comments*.

```
// Line comments are anything after '//' and extend to the end of the line.

let x = 5; // this is also a line comment.

// If you have a long explanation for something, you can put line comments next
// to each other. Put a space between the // and your comment so that it's
// more readable.
```

The other kind of comment is a doc comment. Doc comments use `///` instead of `//`, and support Markdown notation inside:

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let five = 5;
///
/// assert_eq!(6, add_one(5));
/// # fn add_one(x: i32) -> i32 {
/// #     x + 1
/// # }
/// ```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

There is another style of doc comment, `//!`, to comment containing items (e.g. crates, modules or functions), instead of the items following it. Commonly used inside crates root (lib.rs) or modules root (mod.rs):

```
//! # The Rust Standard Library
//!
//! The Rust Standard Library provides the essential runtime
//! functionality for building portable Rust software.
```

When writing doc comments, providing some examples of usage is very, very helpful. You'll notice we've used a new macro here: `assert_eq!`. This compares two values, and `panic!`s if they're not equal to each other. It's very helpful in documentation. There's another macro, `assert!`, which `panic!`s if the value passed to it is `false`.

You can use the `rustdoc`[30] tool to generate HTML documentation from these doc comments, and also to run the code examples as tests!

4.5 if

Rust's take on `if` is not particularly complex, but it's much more like the `if` you'll find in a dynamically typed language than in a more traditional systems language. So let's talk about it, to make sure you grasp the nuances.

`if` is a specific form of a more general concept, the 'branch'. The name comes from a branch in a tree: a decision point, where depending on a choice, multiple paths can be taken.

In the case of `if`, there is one choice that leads down two paths:

```
let x = 5;

if x == 5 {
    println!("x is five!");
}
```

If we changed the value of `x` to something else, this line would not print. More specifically, if the expression after the `if` evaluates to `true`, then the block is executed. If it's `false`, then it is not.

If you want something to happen in the `false` case, use an `else`:

```
let x = 5;

if x == 5 {
    println!("x is five!");
} else {
    println!("x is not five :(");
}
```

If there is more than one case, use an `else if`:

```
let x = 5;

if x == 5 {
    println!("x is five!");
} else if x == 6 {
    println!("x is six!");
} else {
    println!("x is not five or six :(");
}
```

This is all pretty standard. However, you can also do this:

[30] `documentation.html`

```
let x = 5;

let y = if x == 5 {
    10
} else {
    15
}; // y: i32
```

Which we can (and probably should) write like this:

```
let x = 5;

let y = if x == 5 { 10 } else { 15 }; // y: i32
```

This works because `if` is an expression. The value of the expression is the value of the last expression in whichever branch was chosen. An `if` without an `else` always results in `()` as the value.

4.6 Loops

Rust currently provides three approaches to performing some kind of iterative activity. They are: `loop`, `while` and `for`. Each approach has its own set of uses.

loop

The infinite `loop` is the simplest form of loop available in Rust. Using the keyword `loop`, Rust provides a way to loop indefinitely until some terminating statement is reached. Rust’s infinite `loop`s look like this:

```
loop {
    println!(”Loop forever!”);
}
```

while

Rust also has a `while` loop. It looks like this:

```
let mut x = 5; // mut x: i32
let mut done = false; // mut done: bool

while !done {
    x += x - 3;

    println!(”{}”, x);

    if x % 5 == 0 {
        done = true;
    }
}
```

`while` loops are the correct choice when you're not sure how many times you need to loop.

If you need an infinite loop, you may be tempted to write this:

```
while true {
```

However, `loop` is far better suited to handle this case:

```
loop {
```

Rust's control-flow analysis treats this construct differently than a `while true`, since we know that it will always loop. In general, the more information we can give to the compiler, the better it can do with safety and code generation, so you should always prefer `loop` when you plan to loop infinitely.

for

The `for` loop is used to loop a particular number of times. Rust's `for` loops work a bit differently than in other systems languages, however. Rust's `for` loop doesn't look like this "C-style" `for` loop:

```
for (x = 0; x < 10; x++) {
    printf( "%d\n", x );
}
```

Instead, it looks like this:

```
for x in 0..10 {
    println!("{}", x); // x: i32
}
```

In slightly more abstract terms,

```
for var in expression {
    code
}
```

The expression is an iterator[31]. The iterator gives back a series of elements. Each element is one iteration of the loop. That value is then bound to the name `var`, which is valid for the loop body. Once the body is over, the next value is fetched from the iterator, and we loop another time. When there are no more values, the `for` loop is over.

In our example, `0..10` is an expression that takes a start and an end position, and gives an iterator over those values. The upper bound is exclusive, though, so our loop will print `0` through `9`, not `10`.

Rust does not have the "C-style" `for` loop on purpose. Manually controlling each element of the loop is complicated and error prone, even for experienced C developers.

[31] `iterators.html`

Enumerate When you need to keep track of how many times you already looped, you can use the `.enumerate()` function.

On ranges:

```
for (i,j) in (5..10).enumerate() {
    println!(”i = {} and j = {}”, i, j);
}
```

Outputs:

```
i = 0 and j = 5
i = 1 and j = 6
i = 2 and j = 7
i = 3 and j = 8
i = 4 and j = 9
```

Don’t forget to add the parentheses around the range.

On iterators:

```
# let lines = ”hello\nworld”.lines();
for (linenumber, line) in lines.enumerate() {
    println!(”{}: {}”, linenumber, line);
}
```

Outputs:

```
0: Content of line one
1: Content of line two
2: Content of line three
3: Content of line four
```

Ending iteration early

Let’s take a look at that `while` loop we had earlier:

```
let mut x = 5;
let mut done = false;

while !done {
    x += x - 3;

    println!(”{}”, x);

    if x % 5 == 0 {
        done = true;
    }
}
```

We had to keep a dedicated `mut` boolean variable binding, `done`, to know when we should exit out of the loop. Rust has two keywords to help us with modifying iteration: `break` and `continue`.

In this case, we can write the loop in a better way with `break`:

```
let mut x = 5;

loop {
    x += x - 3;

    println!(”{}”, x);

    if x % 5 == 0 { break; }
}
```

We now loop forever with `loop` and use `break` to break out early. Issuing an explicit `return` statement will also serve to terminate the loop early.

`continue` is similar, but instead of ending the loop, goes to the next iteration. This will only print the odd numbers:

```
for x in 0..10 {
    if x % 2 == 0 { continue; }

    println!(”{}”, x);
}
```

Loop labels

You may also encounter situations where you have nested loops and need to specify which one your `break` or `continue` statement is for. Like most other languages, by default a `break` or `continue` will apply to innermost loop. In a situation where you would like to a `break` or `continue` for one of the outer loops, you can use labels to specify which loop the `break` or `continue` statement applies to. This will only print when both `x` and `y` are odd:

```
’outer: for x in 0..10 {
    ’inner: for y in 0..10 {
        if x % 2 == 0 { continue ’outer; } // continues the loop over x
        if y % 2 == 0 { continue ’inner; } // continues the loop over y
        println!(”x: {}, y: {}”, x, y);
    }
}
```

4.7 Ownership

This guide is one of three presenting Rust’s ownership system. This is one of Rust’s most unique and compelling features, with which Rust developers should become

quite acquainted. Ownership is how Rust achieves its largest goal, memory safety. There are a few distinct concepts, each with its own chapter:

- ownership, which you're reading now
- borrowing[32], and their associated feature 'references'
- lifetimes[33], an advanced concept of borrowing

These three chapters are related, and in order. You'll need all three to fully understand the ownership system.

4.7.1 Meta

Before we get to the details, two important notes about the ownership system.

Rust has a focus on safety and speed. It accomplishes these goals through many 'zero-cost abstractions', which means that in Rust, abstractions cost as little as possible in order to make them work. The ownership system is a prime example of a zero-cost abstraction. All of the analysis we'll talk about in this guide is *done at compile time*. You do not pay any run-time cost for any of these features.

However, this system does have a certain cost: learning curve. Many new users to Rust experience something we like to call 'fighting with the borrow checker', where the Rust compiler refuses to compile a program that the author thinks is valid. This often happens because the programmer's mental model of how ownership should work doesn't match the actual rules that Rust implements. You probably will experience similar things at first. There is good news, however: more experienced Rust developers report that once they work with the rules of the ownership system for a period of time, they fight the borrow checker less and less.

With that in mind, let's learn about ownership.

4.7.2 Ownership

Variable bindings[34] have a property in Rust: they 'have ownership' of what they're bound to. This means that when a binding goes out of scope, Rust will free the bound resources. For example:

```
fn foo() {
    let v = vec![1, 2, 3];
}
```

When `v` comes into scope, a new `Vec<T>`[35] is created. In this case, the vector also allocates space on the heap[36], for the three elements. When `v` goes out of scope at the end of `foo()`, Rust will clean up everything related to the vector, even the heap-allocated memory. This happens deterministically, at the end of the scope.

[32] `references-and-borrowing.html`
[33] `lifetimes.html`
[34] `variable-bindings.html`
[35] `../std/vec/struct.Vec.html`
[36] `the-stack-and-the-heap.html`

4.7.3 Move semantics

There's some more subtlety here, though: Rust ensures that there is *exactly one* binding to any given resource. For example, if we have a vector, we can assign it to another binding:

```
let v = vec![1, 2, 3];

let v2 = v;
```

But, if we try to use `v` afterwards, we get an error:

```
let v = vec![1, 2, 3];

let v2 = v;

println!("v[0] is: {}", v[0]);
```

It looks like this:

```
error: use of moved value: ‘v‘
println!("v[0] is: {}", v[0]);
                        ^
```

A similar thing happens if we define a function which takes ownership, and try to use something after we've passed it as an argument:

```
fn take(v: Vec<i32>) {
    // what happens here isn't important.
}

let v = vec![1, 2, 3];

take(v);

println!("v[0] is: {}", v[0]);
```

Same error: 'use of moved value'. When we transfer ownership to something else, we say that we've 'moved' the thing we refer to. You don't need some sort of special annotation here, it's the default thing that Rust does.

The details

The reason that we cannot use a binding after we've moved it is subtle, but important. When we write code like this:

```
let v = vec![1, 2, 3];

let v2 = v;
```

The first line allocates memory for the vector object, `v`, and for the data it contains. The vector object is stored on the stack[37] and contains a pointer to the content (`[1, 2, 3]`) stored on the heap[38]. When we move `v` to `v2`, it creates a copy of that pointer, for `v2`. Which means that there would be two pointers to the content of the vector on the heap. It would violate Rust's safety guarantees by introducing a data race. Therefore, Rust forbids using `v` after we've done the move.

It's also important to note that optimizations may remove the actual copy of the bytes on the stack, depending on circumstances. So it may not be as inefficient as it initially seems.

`Copy` types

We've established that when ownership is transferred to another binding, you cannot use the original binding. However, there's a trait[39] that changes this behavior, and it's called `Copy`. We haven't discussed traits yet, but for now, you can think of them as an annotation to a particular type that adds extra behavior. For example:

```
let v = 1;

let v2 = v;

println!("v is: {}", v);
```

In this case, `v` is an `i32`, which implements the `Copy` trait. This means that, just like a move, when we assign `v` to `v2`, a copy of the data is made. But, unlike a move, we can still use `v` afterward. This is because an `i32` has no pointers to data somewhere else, copying it is a full copy.

All primitive types implement the `Copy` trait and their ownership is therefore not moved like one would assume, following the ´ownership rules´. To give an example, the two following snippets of code only compile because the `i32` and `bool` types implement the `Copy` trait.

```
fn main() {
    let a = 5;

    let _y = double(a);
    println!("{}", a);
}

fn double(x: i32) -> i32 {
    x * 2
}

fn main() {
    let a = true;
```

[37] the-stack-and-the-heap.html
[38] the-stack-and-the-heap.html
[39] traits.html

```
    let _y = change_truth(a);
    println!(”{}”, a);
}

fn change_truth(x: bool) -> bool {
    !x
}
```

If we had used types that do not implement the `Copy` trait, we would have gotten a compile error because we tried to use a moved value.

```
error: use of moved value: ‘a‘
println!(”{}”, a);
               ^
```

We will discuss how to make your own types `Copy` in the traits[40] section.

4.7.4 More than ownership

Of course, if we had to hand ownership back with every function we wrote:

```
fn foo(v: Vec<i32>) -> Vec<i32> {
    // do stuff with v

    // hand back ownership
    v
}
```

This would get very tedious. It gets worse the more things we want to take ownership of:

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // do stuff with v1 and v2

    // hand back ownership, and the result of our function
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

Ugh! The return type, return line, and calling the function gets way more complicated. Luckily, Rust offers a feature, borrowing, which helps us solve this problem. It’s the topic of the next section!

[40] `traits.html`

4.8 References and Borrowing

This guide is two of three presenting Rust's ownership system. This is one of Rust's most unique and compelling features, with which Rust developers should become quite acquainted. Ownership is how Rust achieves its largest goal, memory safety. There are a few distinct concepts, each with its own chapter:

- ownership[41], the key concept
- borrowing, which you're reading now
- lifetimes[42], an advanced concept of borrowing

These three chapters are related, and in order. You'll need all three to fully understand the ownership system.

4.8.1 Meta

Before we get to the details, two important notes about the ownership system.

Rust has a focus on safety and speed. It accomplishes these goals through many 'zero-cost abstractions', which means that in Rust, abstractions cost as little as possible in order to make them work. The ownership system is a prime example of a zero cost abstraction. All of the analysis we'll talk about in this guide is *done at compile time*. You do not pay any run-time cost for any of these features.

However, this system does have a certain cost: learning curve. Many new users to Rust experience something we like to call 'fighting with the borrow checker', where the Rust compiler refuses to compile a program that the author thinks is valid. This often happens because the programmer's mental model of how ownership should work doesn't match the actual rules that Rust implements. You probably will experience similar things at first. There is good news, however: more experienced Rust developers report that once they work with the rules of the ownership system for a period of time, they fight the borrow checker less and less.

With that in mind, let's learn about borrowing.

4.8.2 Borrowing

At the end of the ownership[43] section, we had a nasty function that looked like this:

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // do stuff with v1 and v2

    // hand back ownership, and the result of our function
    (v1, v2, 42)
}
```

[41] ownership.html
[42] lifetimes.html
[43] ownership.html

```
let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

This is not idiomatic Rust, however, as it doesn't take advantage of borrowing. Here's the first step:

```
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
    // do stuff with v1 and v2

    // return the answer
    42
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let answer = foo(&v1, &v2);

// we can use v1 and v2 here!
```

Instead of taking `Vec<i32>`s as our arguments, we take a reference: `&Vec<i32>`. And instead of passing `v1` and `v2` directly, we pass `&v1` and `&v2`. We call the `&T` type a 'reference', and rather than owning the resource, it borrows ownership. A binding that borrows something does not deallocate the resource when it goes out of scope. This means that after the call to `foo()`, we can use our original bindings again.

References are immutable, just like bindings. This means that inside of `foo()`, the vectors can't be changed at all:

```
fn foo(v: &Vec<i32>) {
     v.push(5);
}

let v = vec![];

foo(&v);
```

errors with:

```
error: cannot borrow immutable borrowed content `*v` as mutable
v.push(5);
^
```

Pushing a value mutates the vector, and so we aren't allowed to do it.

4.8.3 &mut references

There’s a second kind of reference: `&mut T`. A ‘mutable reference’ allows you to mutate the resource you’re borrowing. For example:

```
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
}
println!(”{}”, x);
```

This will print `6`. We make `y` a mutable reference to `x`, then add one to the thing `y` points at. You’ll notice that `x` had to be marked `mut` as well. If it wasn’t, we couldn’t take a mutable borrow to an immutable value.

You’ll also notice we added an asterisk (`*`) in front of `y`, making it `*y`, this is because `y` is an `&mut` reference. You’ll also need to use them for accessing the contents of a reference as well.

Otherwise, `&mut` references are just like references. There *is* a large difference between the two, and how they interact, though. You can tell something is fishy in the above example, because we need that extra scope, with the `{` and `}`. If we remove them, we get an error:

```
error: cannot borrow ‘x‘ as immutable because it is also borrowed as mutable
    println!(”{}”, x);
                   ^
note: previous borrow of ‘x‘ occurs here; the mutable borrow prevents
subsequent moves, borrows, or modification of ‘x‘ until the borrow ends
        let y = &mut x;
                      ^
note: previous borrow ends here
fn main() {

}
^
```

As it turns out, there are rules.

4.8.4 The Rules

Here’s the rules about borrowing in Rust:

First, any borrow must last for a scope no greater than that of the owner. Second, you may have one or the other of these two kinds of borrows, but not both at the same time:

- one or more references (`&T`) to a resource,
- exactly one mutable reference (`&mut T`).

You may notice that this is very similar, though not exactly the same as, to the definition of a data race:

> There is a 'data race' when two or more pointers access the same memory location at the same time, where at least one of them is writing, and the operations are not synchronized.

With references, you may have as many as you'd like, since none of them are writing. However, as we can only have one `&mut` at a time, it is impossible to have a data race. This is how Rust prevents data races at compile time: we'll get errors if we break the rules.

With this in mind, let's consider our example again.

Thinking in scopes

Here's the code:

```
let mut x = 5;
let y = &mut x;

*y += 1;

println!("{}", x);
```

This code gives us this error:

```
error: cannot borrow 'x' as immutable because it is also borrowed as mutable
    println!("{}", x);
                   ^
```

This is because we've violated the rules: we have a `&mut T` pointing to `x`, and so we aren't allowed to create any `&T`s. One or the other. The note hints at how to think about this problem:

```
note: previous borrow ends here
fn main() {

}
^
```

In other words, the mutable borrow is held through the rest of our example. What we want is for the mutable borrow to end *before* we try to call `println!` and make an immutable borrow. In Rust, borrowing is tied to the scope that the borrow is valid for. And our scopes look like this:

```
let mut x = 5;

let y = &mut x;    // -+ &mut borrow of x starts here
```

```
                    //  |
*y += 1;            //  |
                    //  |
println!(”{}”, x); // -+ - try to borrow x here
                    // -+ &mut borrow of x ends here
```

The scopes conflict: we can’t make an `&x` while `y` is in scope.
So when we add the curly braces:

```
let mut x = 5;

{
    let y = &mut x; // -+ &mut borrow starts here
    *y += 1;        //  |
}                   // -+ ... and ends here

println!(”{}”, x);  // <- try to borrow x here
```

There’s no problem. Our mutable borrow goes out of scope before we create an immutable one. But scope is the key to seeing how long a borrow lasts for.

Issues borrowing prevents

Why have these restrictive rules? Well, as we noted, these rules prevent data races. What kinds of issues do data races cause? Here’s a few.

Iterator invalidation One example is ‘iterator invalidation’, which happens when you try to mutate a collection that you’re iterating over. Rust’s borrow checker prevents this from happening:

```
let mut v = vec![1, 2, 3];

for i in &v {
    println!(”{}”, i);
}
```

This prints out one through three. As we iterate through the vectors, we’re only given references to the elements. And `v` is itself borrowed as immutable, which means we can’t change it while we’re iterating:

```
let mut v = vec![1, 2, 3];

for i in &v {
    println!(”{}”, i);
    v.push(34);
}
```

Here’s the error:

```
error: cannot borrow ‘v‘ as mutable because it is also borrowed as immutable
    v.push(34);
    ^
note: previous borrow of ‘v‘ occurs here; the immutable borrow prevents
subsequent moves or mutable borrows of ‘v‘ until the borrow ends
for i in &v {
          ^
note: previous borrow ends here
for i in &v {
    println!(“{}”, i);
    v.push(34);
}
^
```

We can’t modify `v` because it’s borrowed by the loop.

use after free References must not live longer than the resource they refer to. Rust will check the scopes of your references to ensure that this is true.

If Rust didn’t check this property, we could accidentally use a reference which was invalid. For example:

```
let y: &i32;
{
    let x = 5;
    y = &x;
}

println!(”{}”, y);
```

We get this error:

```
error: ‘x‘ does not live long enough
    y = &x;
         ^
note: reference must be valid for the block suffix following statement 0 at
2:16...
let y: &i32;
{
    let x = 5;
    y = &x;
}

note: ...but borrowed value is only valid for the block suffix following
statement 0 at 4:18
    let x = 5;
    y = &x;
}
```

In other words, `y` is only valid for the scope where `x` exists. As soon as `x` goes away, it becomes invalid to refer to it. As such, the error says that the borrow ‘doesn’t live long enough’ because it’s not valid for the right amount of time.

The same problem occurs when the reference is declared *before* the variable it refers to. This is because resources within the same scope are freed in the opposite order they were declared:

```
let y: &i32;
let x = 5;
y = &x;

println!(”{}”, y);
```

We get this error:

```
error: ‘x‘ does not live long enough
y = &x;
     ^
note: reference must be valid for the block suffix following statement 0 at
2:16...
    let y: &i32;
    let x = 5;
    y = &x;

    println!(”{}”, y);
}

note: ...but borrowed value is only valid for the block suffix following
statement 1 at 3:14
    let x = 5;
    y = &x;

    println!(”{}”, y);
}
```

In the above example, `y` is declared before `x`, meaning that `y` lives longer than `x`, which is not allowed.

4.9 Lifetimes

This guide is three of three presenting Rust’s ownership system. This is one of Rust’s most unique and compelling features, with which Rust developers should become quite acquainted. Ownership is how Rust achieves its largest goal, memory safety. There are a few distinct concepts, each with its own chapter:

- ownership[44], the key concept

[44] `ownership.html`

- borrowing[45], and their associated feature 'references'
- lifetimes, which you're reading now

These three chapters are related, and in order. You'll need all three to fully understand the ownership system.

4.9.1 Meta

Before we get to the details, two important notes about the ownership system.

Rust has a focus on safety and speed. It accomplishes these goals through many 'zero-cost abstractions', which means that in Rust, abstractions cost as little as possible in order to make them work. The ownership system is a prime example of a zero-cost abstraction. All of the analysis we'll talk about in this guide is *done at compile time*. You do not pay any run-time cost for any of these features.

However, this system does have a certain cost: learning curve. Many new users to Rust experience something we like to call 'fighting with the borrow checker', where the Rust compiler refuses to compile a program that the author thinks is valid. This often happens because the programmer's mental model of how ownership should work doesn't match the actual rules that Rust implements. You probably will experience similar things at first. There is good news, however: more experienced Rust developers report that once they work with the rules of the ownership system for a period of time, they fight the borrow checker less and less.

With that in mind, let's learn about lifetimes.

4.9.2 Lifetimes

Lending out a reference to a resource that someone else owns can be complicated. For example, imagine this set of operations:

1. I acquire a handle to some kind of resource.
2. I lend you a reference to the resource.
3. I decide I'm done with the resource, and deallocate it, while you still have your reference.
4. You decide to use the resource.

Uh oh! Your reference is pointing to an invalid resource. This is called a dangling pointer or 'use after free', when the resource is memory.

To fix this, we have to make sure that step four never happens after step three. The ownership system in Rust does this through a concept called lifetimes, which describe the scope that a reference is valid for.

When we have a function that takes a reference by argument, we can be implicit or explicit about the lifetime of the reference:

```
// implicit
fn foo(x: &i32) {
```

[45] `references-and-borrowing.html`

```
}

// explicit
fn bar<'a>(x: &'a i32) {
}
```

The `'a` reads 'the lifetime a'. Technically, every reference has some lifetime associated with it, but the compiler lets you elide (i.e. omit, see "Lifetime Elision" below) them in common cases. Before we get to that, though, let's break the explicit example down:

```
fn bar<'a>(...)
```

We previously talked a little about function syntax[46], but we didn't discuss the `<>`s after a function's name. A function can have 'generic parameters' between the `<>`s, of which lifetimes are one kind. We'll discuss other kinds of generics later in the book[47], but for now, let's just focus on the lifetimes aspect.

We use `<>` to declare our lifetimes. This says that `bar` has one lifetime, `'a`. If we had two reference parameters, it would look like this:

```
fn bar<'a, 'b>(...)
```

Then in our parameter list, we use the lifetimes we've named:

```
...(x: &'a i32)
```

If we wanted an `&mut` reference, we'd do this:

```
...(x: &'a mut i32)
```

If you compare `&mut i32` to `&'a mut i32`, they're the same, it's just that the lifetime `'a` has snuck in between the `&` and the `mut i32`. We read `&mut i32` as 'a mutable reference to an `i32`' and `&'a mut i32` as 'a mutable reference to an `i32` with the lifetime `'a`'.

4.9.3 In `struct`s

You'll also need explicit lifetimes when working with `struct`[48]s that contain references:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5; // this is the same as `let _y = 5; let y = &_y;`
```

[46] `functions.html`
[47] `generics.html`
[48] `structs.html`

```
    let f = Foo { x: y };

    println!("{}", f.x);
}
```

As you can see, `structs` can also have lifetimes. In a similar way to functions,

```
struct Foo<'a> {
# x: &'a i32,
# }
```

declares a lifetime, and

```
# struct Foo<'a> {
x: &'a i32,
# }
```

uses it. So why do we need a lifetime here? We need to ensure that any reference to a `Foo` cannot outlive the reference to an `i32` it contains.

`impl` blocks

Let's implement a method on `Foo`:

```
struct Foo<'a> {
    x: &'a i32,
}

impl<'a> Foo<'a> {
    fn x(&self) -> &'a i32 { self.x }
}

fn main() {
    let y = &5; // this is the same as `let _y = 5; let y = &_y;`
    let f = Foo { x: y };

    println!("x is: {}", f.x());
}
```

As you can see, we need to declare a lifetime for `Foo` in the `impl` line. We repeat `'a` twice, just like on functions: `impl<'a>` defines a lifetime `'a`, and `Foo<'a>` uses it.

Multiple lifetimes

If you have multiple references, you can use the same lifetime multiple times:

```
fn x_or_y<'a>(x: &'a str, y: &'a str) -> &'a str {
#    x
# }
```

This says that `x` and `y` both are alive for the same scope, and that the return value is also alive for that scope. If you wanted `x` and `y` to have different lifetimes, you can use multiple lifetime parameters:

```
fn x_or_y<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
#    x
# }
```

In this example, `x` and `y` have different valid scopes, but the return value has the same lifetime as `x`.

Thinking in scopes

A way to think about lifetimes is to visualize the scope that a reference is valid for. For example:

```
fn main() {
    let y = &5;     // -+ y goes into scope
                    //  |
    // stuff        //  |
                    //  |
}                   // -+ y goes out of scope
```

Adding in our `Foo`:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5;           // -+ y goes into scope
    let f = Foo { x: y }; // -+ f goes into scope
    // stuff              //  |
                          //  |
}                         // -+ f and y go out of scope
```

Our `f` lives within the scope of `y`, so everything works. What if it didn't? This code won't work:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                    // -+ x goes into scope
                              //  |
    {                         //  |
        let y = &5;           // ---+ y goes into scope
```

```
        let f = Foo { x: y }; // ---+ f goes into scope
        x = &f.x;             //  | | error here
    }                         // ---+ f and y go out of scope
                              //  |
    println!(”{}”, x);        //  |
}                             // -+ x goes out of scope
```

Whew! As you can see here, the scopes of `f` and `y` are smaller than the scope of `x`. But when we do `x = &f.x`, we make `x` a reference to something that’s about to go out of scope.

Named lifetimes are a way of giving these scopes a name. Giving something a name is the first step towards being able to talk about it.

’static

The lifetime named ‘static’ is a special lifetime. It signals that something has the lifetime of the entire program. Most Rust programmers first come across `’static` when dealing with strings:

```
let x: &’static str = ”Hello, world.”;
```

String literals have the type `&’static str` because the reference is always alive: they are baked into the data segment of the final binary. Another example are globals:

```
static FOO: i32 = 5;
let x: &’static i32 = &FOO;
```

This adds an `i32` to the data segment of the binary, and `x` is a reference to it.

Lifetime Elision

Rust supports powerful local type inference in function bodies, but it’s forbidden in item signatures to allow reasoning about the types based on the item signature alone. However, for ergonomic reasons a very restricted secondary inference algorithm called “lifetime elision” applies in function signatures. It infers only based on the signature components themselves and not based on the body of the function, only infers lifetime parameters, and does this with only three easily memorizable and unambiguous rules. This makes lifetime elision a shorthand for writing an item signature, while not hiding away the actual types involved as full local inference would if applied to it.

When talking about lifetime elision, we use the term *input lifetime* and *output lifetime*. An *input lifetime* is a lifetime associated with a parameter of a function, and an *output lifetime* is a lifetime associated with the return value of a function. For example, this function has an input lifetime:

```
fn foo<’a>(bar: &’a str)
```

This one has an output lifetime:

```
fn foo<'a>() -> &'a str
```

This one has a lifetime in both positions:

```
fn foo<'a>(bar: &'a str) -> &'a str
```

Here are the three rules:

- Each elided lifetime in a function's arguments becomes a distinct lifetime parameter.
- If there is exactly one input lifetime, elided or not, that lifetime is assigned to all elided lifetimes in the return values of that function.
- If there are multiple input lifetimes, but one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to all elided output lifetimes.

Otherwise, it is an error to elide an output lifetime.

Examples Here are some examples of functions with elided lifetimes. We've paired each example of an elided lifetime with its expanded form.

```
fn print(s: &str); // elided
fn print<'a>(s: &'a str); // expanded

fn debug(lvl: u32, s: &str); // elided
fn debug<'a>(lvl: u32, s: &'a str); // expanded

// In the preceding example, `lvl` doesn't need a lifetime because it's not a
// reference (`&`). Only things relating to references (such as a `struct`
// which contains a reference) need lifetimes.

fn substr(s: &str, until: u32) -> &str; // elided
fn substr<'a>(s: &'a str, until: u32) -> &'a str; // expanded

fn get_str() -> &str; // ILLEGAL, no inputs

fn frob(s: &str, t: &str) -> &str; // ILLEGAL, two inputs
fn frob<'a, 'b>(s: &'a str, t: &'b str) -> &str; // Expanded: Output lifetime is ambiguous

fn get_mut(&mut self) -> &mut T; // elided
fn get_mut<'a>(&'a mut self) -> &'a mut T; // expanded

fn args<T:ToCStr>(&mut self, args: &[T]) -> &mut Command; // elided
fn args<'a, 'b, T:ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command; // expanded

fn new(buf: &mut [u8]) -> BufWriter; // elided
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a>; // expanded
```

4.10 Mutability

Mutability, the ability to change something, works a bit differently in Rust than in other languages. The first aspect of mutability is its non-default status:

```
let x = 5;
x = 6; // error!
```

We can introduce mutability with the `mut` keyword:

```
let mut x = 5;

x = 6; // no problem!
```

This is a mutable variable binding[49]. When a binding is mutable, it means you're allowed to change what the binding points to. So in the above example, it's not so much that the value at `x` is changing, but that the binding changed from one `i32` to another.

If you want to change what the binding points to, you'll need a mutable reference[50]:

```
let mut x = 5;
let y = &mut x;
```

`y` is an immutable binding to a mutable reference, which means that you can't bind `y` to something else (`y = &mut z`), but you can mutate the thing that's bound to `y` (`*y = 5`). A subtle distinction.

Of course, if you need both:

```
let mut x = 5;
let mut y = &mut x;
```

Now `y` can be bound to another value, and the value it's referencing can be changed.

It's important to note that `mut` is part of a pattern[51], so you can do things like this:

```
let (mut x, y) = (5, 6);

fn foo(mut x: i32) {
# }
```

4.10.1 Interior vs. Exterior Mutability

However, when we say something is 'immutable' in Rust, that doesn't mean that it's not able to be changed: we mean something has 'exterior mutability'. Consider, for example, `Arc<T>`[52]:

[49] `variable-bindings.html`
[50] `references-and-borrowing.html`
[51] `patterns.html`
[52] `../std/sync/struct.Arc.html`

```
use std::sync::Arc;

let x = Arc::new(5);
let y = x.clone();
```

When we call `clone()`, the `Arc<T>` needs to update the reference count. Yet we've not used any `mut`s here, `x` is an immutable binding, and we didn't take `&mut 5` or anything. So what gives?

To understand this, we have to go back to the core of Rust's guiding philosophy, memory safety, and the mechanism by which Rust guarantees it, the ownership[53] system, and more specifically, borrowing[54]:

> You may have one or the other of these two kinds of borrows, but not both at the same time:
>
> - one or more references (`&T`) to a resource,
> - exactly one mutable reference (`&mut T`).

So, that's the real definition of 'immutability': is this safe to have two pointers to? In `Arc<T>`'s case, yes: the mutation is entirely contained inside the structure itself. It's not user facing. For this reason, it hands out `&T` with `clone()`. If it handed out `&mut T`s, though, that would be a problem.

Other types, like the ones in the `std::cell`[55] module, have the opposite: interior mutability. For example:

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
```

RefCell hands out `&mut` references to what's inside of it with the `borrow_mut()` method. Isn't that dangerous? What if we do:

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
let z = x.borrow_mut();
# (y, z);
```

This will in fact panic, at runtime. This is what `RefCell` does: it enforces Rust's borrowing rules at runtime, and `panic!`s if they're violated. This allows us to get around another aspect of Rust's mutability rules. Let's talk about it first.

[53] `ownership.html`
[54] `references-and-borrowing.html#borrowing`
[55] `../std/cell/index.html`

Field-level mutability

Mutability is a property of either a borrow (`&mut`) or a binding (`let mut`). This means that, for example, you cannot have a `struct`[56] with some fields mutable and some immutable:

```
struct Point {
    x: i32,
    mut y: i32, // nope
}
```

The mutability of a struct is in its binding:

```
struct Point {
    x: i32,
    y: i32,
}

let mut a = Point { x: 5, y: 6 };

a.x = 10;

let b = Point { x: 5, y: 6};

b.x = 10; // error: cannot assign to immutable field ‘b.x‘
```

However, by using `Cell<T>`[57], you can emulate field-level mutability:

```
use std::cell::Cell;

struct Point {
    x: i32,
    y: Cell<i32>,
}

let point = Point { x: 5, y: Cell::new(6) };

point.y.set(7);

println!("y: {:?}", point.y);
```

This will print `y: Cell { value: 7 }`. We've successfully updated `y`.

4.11 Structs

`structs` are a way of creating more complex data types. For example, if we were doing calculations involving coordinates in 2D space, we would need both an `x` and a `y` value:

[56] `structs.html`
[57] `../std/cell/struct.Cell.html`

```
let origin_x = 0;
let origin_y = 0;
```

A `struct` lets us combine these two into a single, unified datatype:

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let origin = Point { x: 0, y: 0 }; // origin: Point

    println!("The origin is at ({}, {})", origin.x, origin.y);
}
```

There's a lot going on here, so let's break it down. We declare a `struct` with the `struct` keyword, and then with a name. By convention, `structs` begin with a capital letter and are camel cased: `PointInSpace`, not `Point_In_Space`.

We can create an instance of our `struct` via `let`, as usual, but we use a `key: value` style syntax to set each field. The order doesn't need to be the same as in the original declaration.

Finally, because fields have names, we can access the field through dot notation: `origin.x`.

The values in `structs` are immutable by default, like other bindings in Rust. Use `mut` to make them mutable:

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    point.x = 5;

    println!("The point is at ({}, {})", point.x, point.y);
}
```

This will print `The point is at (5, 0)`.

Rust does not support field mutability at the language level, so you cannot write something like this:

```
struct Point {
    mut x: i32,
    y: i32,
}
```

Mutability is a property of the binding, not of the structure itself. If you're used to field-level mutability, this may seem strange at first, but it significantly simplifies things. It even lets you make things mutable for a short time only:

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    point.x = 5;

    let point = point; // this new binding can't change now

    point.y = 6; // this causes an error
}
```

4.11.1 Update syntax

A `struct` can include `..` to indicate that you want to use a copy of some other `struct` for some of the values. For example:

```
struct Point3d {
    x: i32,
    y: i32,
    z: i32,
}

let mut point = Point3d { x: 0, y: 0, z: 0 };
point = Point3d { y: 1, .. point };
```

This gives `point` a new `y`, but keeps the old `x` and `z` values. It doesn't have to be the same `struct` either, you can use this syntax when making new ones, and it will copy the values you don't specify:

```
# struct Point3d {
#     x: i32,
#     y: i32,
#     z: i32,
# }
let origin = Point3d { x: 0, y: 0, z: 0 };
let point = Point3d { z: 1, x: 2, .. origin };
```

4.11.2 Tuple structs

Rust has another data type that's like a hybrid between a tuple[58] and a `struct`, called a 'tuple struct'. Tuple structs have a name, but their fields don't:

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);
```

These two will not be equal, even if they have the same values:

```
# struct Color(i32, i32, i32);
# struct Point(i32, i32, i32);
let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

It is almost always better to use a `struct` than a tuple struct. We would write `Color` and `Point` like this instead:

```
struct Color {
    red: i32,
    blue: i32,
    green: i32,
}

struct Point {
    x: i32,
    y: i32,
    z: i32,
}
```

Now, we have actual names, rather than positions. Good names are important, and with a `struct`, we have actual names.

There *is* one case when a tuple struct is very useful, though, and that's a tuple struct with only one element. We call this the 'newtype' pattern, because it allows you to create a new type, distinct from that of its contained value and expressing its own semantic meaning:

```
struct Inches(i32);

let length = Inches(10);

let Inches(integer_length) = length;
println!("length is {} inches", integer_length);
```

As you can see here, you can extract the inner integer type through a destructuring `let`, just as with regular tuples. In this case, the `let Inches(integer_length)` assigns `10` to `integer_length`.

[58] `primitive-types.html#tuples`

4.11.3 Unit-like structs

You can define a `struct` with no members at all:

```
struct Electron;

let x = Electron;
```

Such a `struct` is called 'unit-like' because it resembles the empty tuple, `()`, sometimes called 'unit'. Like a tuple struct, it defines a new type.

This is rarely useful on its own (although sometimes it can serve as a marker type), but in combination with other features, it can become useful. For instance, a library may ask you to create a structure that implements a certain trait[59] to handle events. If you don't have any data you need to store in the structure, you can just create a unit-like `struct`.

4.12 Enums

An `enum` in Rust is a type that represents data that could be one of several possible variants:

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}
```

Each variant can optionally have data associated with it. The syntax for defining variants resembles the syntaxes used to define structs: you can have variants with no data (like unit-like structs), variants with named data, and variants with unnamed data (like tuple structs). Unlike separate struct definitions, however, an `enum` is a single type. A value of the enum can match any of the variants. For this reason, an enum is sometimes called a 'sum type': the set of possible values of the enum is the sum of the sets of possible values for each variant.

We use the `::` syntax to use the name of each variant: they're scoped by the name of the `enum` itself. This allows both of these to work:

```
# enum Message {
#     Move { x: i32, y: i32 },
# }
let x: Message = Message::Move { x: 3, y: 4 };

enum BoardGameTurn {
    Move { squares: i32 },
```

[59] `traits.html`

```
    Pass,
}

let y: BoardGameTurn = BoardGameTurn::Move { squares: 1 };
```

Both variants are named `Move`, but since they're scoped to the name of the enum, they can both be used without conflict.

A value of an enum type contains information about which variant it is, in addition to any data associated with that variant. This is sometimes referred to as a 'tagged union', since the data includes a 'tag' indicating what type it is. The compiler uses this information to enforce that you're accessing the data in the enum safely. For instance, you can't simply try to destructure a value as if it were one of the possible variants:

```
fn process_color_change(msg: Message) {
    let Message::ChangeColor(r, g, b) = msg; // compile-time error
}
```

Not supporting these operations may seem rather limiting, but it's a limitation which we can overcome. There are two ways: by implementing equality ourselves, or by pattern matching variants with `match`[60] expressions, which you'll learn in the next section. We don't know enough about Rust to implement equality yet, but we'll find out in the `traits`[61] section.

4.12.1 Constructors as functions

An enum's constructors can also be used like functions. For example:

```
# enum Message {
# Write(String),
# }
let m = Message::Write("Hello, world".to_string());
```

Is the same as

```
# enum Message {
# Write(String),
# }
fn foo(x: String) -> Message {
    Message::Write(x)
}

let x = foo("Hello, world".to_string());
```

This is not immediately useful to us, but when we get to `closures`[62], we'll talk about passing functions as arguments to other functions. For example, with `iterators`[63], we can do this to convert a vector of `String`s into a vector of `Message::Write`s:

[60] `match.html`
[61] `traits.html`
[62] `closures.html`
[63] `iterators.html`

```
# enum Message {
# Write(String),
# }

let v = vec!["Hello".to_string(), "World".to_string()];

let v1: Vec<Message> = v.into_iter().map(Message::Write).collect();
```

4.13 Match

Often, a simple `if`[64]/`else` isn't enough, because you have more than two possible options. Also, conditions can get quite complex. Rust has a keyword, `match`, that allows you to replace complicated `if`/`else` groupings with something more powerful. Check it out:

```
let x = 5;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}
```

`match` takes an expression and then branches based on its value. Each 'arm' of the branch is of the form `val => expression`. When the value matches, that arm's expression will be evaluated. It's called `match` because of the term 'pattern matching', which `match` is an implementation of. There's an entire section on patterns[65] that covers all the patterns that are possible here.

So what's the big advantage? Well, there are a few. First of all, `match` enforces 'exhaustiveness checking'. Do you see that last arm, the one with the underscore (_)? If we remove that arm, Rust will give us an error:

```
error: non-exhaustive patterns: `_` not covered
```

In other words, Rust is trying to tell us we forgot a value. Because `x` is an integer, Rust knows that it can have a number of different values – for example, `6`. Without the _, however, there is no arm that could match, and so Rust refuses to compile the code. _ acts like a 'catch-all arm'. If none of the other arms match, the arm with _ will, and since we have this catch-all arm, we now have an arm for every possible value of `x`, and so our program will compile successfully.

`match` is also an expression, which means we can use it on the right-hand side of a `let` binding or directly where an expression is used:

[64] `if.html`
[65] `patterns.html`

```
let x = 5;

let number = match x {
    1 => ”one”,
    2 => ”two”,
    3 => ”three”,
    4 => ”four”,
    5 => ”five”,
    _ => ”something else”,
};
```

Sometimes it’s a nice way of converting something from one type to another.

4.13.1 Matching on enums

Another important use of the `match` keyword is to process the possible variants of an enum:

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}

fn quit() { /* ... */ }
fn change_color(r: i32, g: i32, b: i32) { /* ... */ }
fn move_cursor(x: i32, y: i32) { /* ... */ }

fn process_message(msg: Message) {
    match msg {
        Message::Quit => quit(),
        Message::ChangeColor(r, g, b) => change_color(r, g, b),
        Message::Move { x: x, y: y } => move_cursor(x, y),
        Message::Write(s) => println!(”{}”, s),
    };
}
```

Again, the Rust compiler checks exhaustiveness, so it demands that you have a match arm for every variant of the enum. If you leave one off, it will give you a compile-time error unless you use `_`.

Unlike the previous uses of `match`, you can’t use the normal `if` statement to do this. You can use the `if let`[66] statement, which can be seen as an abbreviated form of `match`.

[66] `if-let.html`

4.14 Patterns

Patterns are quite common in Rust. We use them in variable bindings[67], match statements[68], and other places, too. Let's go on a whirlwind tour of all of the things patterns can do!

A quick refresher: you can match against literals directly, and _ acts as an 'any' case:

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

This prints one.

There's one pitfall with patterns: like anything that introduces a new binding, they introduce shadowing. For example:

```
let x = 'x';
let c = 'c';

match c {
    x => println!("x: {} c: {}", x, c),
}

println!("x: {}", x)
```

This prints:

```
x: c c: c
x: x
```

In other words, `x =>` matches the pattern and introduces a new binding named `x` that's in scope for the match arm. Because we already have a binding named `x`, this new `x` shadows it.

4.14.1 Multiple patterns

You can match multiple patterns with `|`:

```
let x = 1;

match x {
```

[67] variable-bindings.html
[68] match.html

```
    1 | 2 => println!(”one or two”),
    3 => println!(”three”),
    _ => println!(”anything”),
}
```

This prints `one or two`.

4.14.2 Destructuring

If you have a compound data type, like a `struct`[69], you can destructure it inside of a pattern:

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x, y } => println!(”({},{})”, x, y),
}
```

We can use `:` to give a value a different name.

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x: x1, y: y1 } => println!(”({},{})”, x1, y1),
}
```

If we only care about some of the values, we don’t have to give them all names:

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x, .. } => println!(”x is {}”, x),
}
```

[69] `structs.html`

This prints `x is 0`.

You can do this kind of match on any member, not just the first:

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { y, .. } => println!(”y is {}”, y),
}
```

This prints `y is 0`.

This ‘destructuring’ behavior works on any compound data type, like tuples[70] or enums[71].

4.14.3 Ignoring bindings

You can use _ in a pattern to disregard the type and value. For example, here’s a `match` against a `Result<T, E>`:

```
# let some_value: Result<i32, &’static str> = Err(”There was an error”);
match some_value {
    Ok(value) => println!(”got a value: {}”, value),
    Err(_) => println!(”an error occurred”),
}
```

In the first arm, we bind the value inside the `Ok` variant to `value`. But in the `Err` arm, we use _ to disregard the specific error, and just print a general error message.

_ is valid in any pattern that creates a binding. This can be useful to ignore parts of a larger structure:

```
fn coordinate() -> (i32, i32, i32) {
    // generate and return some sort of triple tuple
# (1, 2, 3)
}

let (x, _, z) = coordinate();
```

Here, we bind the first and last element of the tuple to `x` and `z`, but ignore the middle element.

Similarly, you can use `..` in a pattern to disregard multiple values.

[70] `primitive-types.html#tuples`

[71] `enums.html`

```
enum OptionalTuple {
    Value(i32, i32, i32),
    Missing,
}

let x = OptionalTuple::Value(5, -2, 3);

match x {
    OptionalTuple::Value(..) => println!(”Got a tuple!”),
    OptionalTuple::Missing => println!(”No such luck.”),
}
```

This prints `Got a tuple!`.

4.14.4 ref and ref mut

If you want to get a reference[72], use the `ref` keyword:

```
let x = 5;

match x {
    ref r => println!(”Got a reference to {}”, r),
}
```

This prints `Got a reference to 5`.

Here, the `r` inside the `match` has the type `&i32`. In other words, the `ref` keyword *creates* a reference, for use in the pattern. If you need a mutable reference, `ref mut` will work in the same way:

```
let mut x = 5;

match x {
    ref mut mr => println!(”Got a mutable reference to {}”, mr),
}
```

4.14.5 Ranges

You can match a range of values with `...`:

```
let x = 1;

match x {
    1 ... 5 => println!(”one through five”),
    _ => println!(”anything”),
}
```

[72] `references-and-borrowing.html`

This prints `one through five`.
Ranges are mostly used with integers and `char`s:

```
let x = '';

match x {
    'a' ... 'j' => println!("early letter"),
    'k' ... 'z' => println!("late letter"),
    _ => println!("something else"),
}
```

This prints `something else`.

4.14.6 Bindings

You can bind values to names with `@`:

```
let x = 1;

match x {
    e @ 1 ... 5 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

This prints `got a range element 1`. This is useful when you want to do a complicated match of part of a data structure:

```
#[derive(Debug)]
struct Person {
    name: Option<String>,
}

let name = "Steve".to_string();
let mut x: Option<Person> = Some(Person { name: Some(name) });
match x {
    Some(Person { name: ref a @ Some(_), .. }) => println!("{:?}", a),
    _ => {}
}
```

This prints `Some("Steve")`: we've bound the inner `name` to `a`.
If you use `@` with `|`, you need to make sure the name is bound in each part of the pattern:

```
let x = 5;

match x {
    e @ 1 ... 5 | e @ 8 ... 10 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

4.14.7 Guards

You can introduce 'match guards' with `if`:

```
enum OptionalInt {
    Value(i32),
    Missing,
}

let x = OptionalInt::Value(5);

match x {
    OptionalInt::Value(i) if i > 5 => println!("Got an int bigger than five!"),
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing => println!("No such luck."),
}
```

This prints `Got an int!`.

If you're using `if` with multiple patterns, the `if` applies to both sides:

```
let x = 4;
let y = false;

match x {
    4 | 5 if y => println!("yes"),
    _ => println!("no"),
}
```

This prints `no`, because the `if` applies to the whole of `4 | 5`, and not to just the `5`. In other words, the precedence of `if` behaves like this:

```
(4 | 5) if y => ...
```

not this:

```
4 | (5 if y) => ...
```

4.14.8 Mix and Match

Whew! That's a lot of different ways to match things, and they can all be mixed and matched, depending on what you're doing:

```
match x {
    Foo { x: Some(ref name), y: None } => ...
}
```

Patterns are very powerful. Make good use of them.

4.15 Method Syntax

Functions are great, but if you want to call a bunch of them on some data, it can be awkward. Consider this code:

```
baz(bar(foo));
```

We would read this left-to-right, and so we see 'baz bar foo'. But this isn't the order that the functions would get called in, that's inside-out: 'foo bar baz'. Wouldn't it be nice if we could do this instead?

```
foo.bar().baz();
```

Luckily, as you may have guessed with the leading question, you can! Rust provides the ability to use this 'method call syntax' via the `impl` keyword.

4.15.1 Method calls

Here's how it works:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());
}
```

This will print `12.566371`.

We've made a `struct` that represents a circle. We then write an `impl` block, and inside it, define a method, `area`.

Methods take a special first parameter, of which there are three variants: `self`, `&self`, and `&mut self`. You can think of this first parameter as being the `foo` in `foo.bar()`. The three variants correspond to the three kinds of things `foo` could be: `self` if it's just a value on the stack, `&self` if it's a reference, and `&mut self` if it's a mutable reference. Because we took the `&self` parameter to `area`, we can use it just like any other parameter. Because we know it's a `Circle`, we can access the `radius` just like we would with any other `struct`.

We should default to using `&self`, as you should prefer borrowing over taking ownership, as well as taking immutable references over mutable ones. Here's an example of all three variants:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn reference(&self) {
       println!(”taking self by reference!”);
    }

    fn mutable_reference(&mut self) {
       println!(”taking self by mutable reference!”);
    }

    fn takes_ownership(self) {
       println!(”taking ownership of self!”);
    }
}
```

You can use as many `impl` blocks as you'd like. The previous example could have also been written like this:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn reference(&self) {
       println!(”taking self by reference!”);
    }
}

impl Circle {
    fn mutable_reference(&mut self) {
       println!(”taking self by mutable reference!”);
    }
}

impl Circle {
    fn takes_ownership(self) {
       println!(”taking ownership of self!”);
    }
}
```

4.15.2 Chaining method calls

So, now we know how to call a method, such as `foo.bar()`. But what about our original example, `foo.bar().baz()`? This is called ‘method chaining’. Let’s look at an example:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }

    fn grow(&self, increment: f64) -> Circle {
        Circle { x: self.x, y: self.y, radius: self.radius + increment }
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!(”{}”, c.area());

    let d = c.grow(2.0).area();
    println!(”{}”, d);
}
```

Check the return type:

```
# struct Circle;
# impl Circle {
fn grow(&self, increment: f64) -> Circle {
# Circle } }
```

We just say we’re returning a `Circle`. With this method, we can grow a new `Circle` to any arbitrary size.

4.15.3 Associated functions

You can also define associated functions that do not take a `self` parameter. Here’s a pattern that’s very common in Rust code:

```
struct Circle {
    x: f64,
    y: f64,
```

```
    radius: f64,
}

impl Circle {
    fn new(x: f64, y: f64, radius: f64) -> Circle {
        Circle {
            x: x,
            y: y,
            radius: radius,
        }
    }
}

fn main() {
    let c = Circle::new(0.0, 0.0, 2.0);
}
```

This 'associated function' builds a new `Circle` for us. Note that associated functions are called with the `Struct::function()` syntax, rather than the `ref.method()` syntax. Some other languages call associated functions 'static methods'.

4.15.4 Builder Pattern

Let's say that we want our users to be able to create `Circle`s, but we will allow them to only set the properties they care about. Otherwise, the `x` and `y` attributes will be `0.0`, and the `radius` will be `1.0`. Rust doesn't have method overloading, named arguments, or variable arguments. We employ the builder pattern instead. It looks like this:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct CircleBuilder {
    x: f64,
    y: f64,
    radius: f64,
}

impl CircleBuilder {
    fn new() -> CircleBuilder {
```

```
        CircleBuilder { x: 0.0, y: 0.0, radius: 1.0, }
    }

    fn x(&mut self, coordinate: f64) -> &mut CircleBuilder {
        self.x = coordinate;
        self
    }

    fn y(&mut self, coordinate: f64) -> &mut CircleBuilder {
        self.y = coordinate;
        self
    }

    fn radius(&mut self, radius: f64) -> &mut CircleBuilder {
        self.radius = radius;
        self
    }

    fn finalize(&self) -> Circle {
        Circle { x: self.x, y: self.y, radius: self.radius }
    }
}

fn main() {
    let c = CircleBuilder::new()
                .x(1.0)
                .y(2.0)
                .radius(2.0)
                .finalize();

    println!("area: {}", c.area());
    println!("x: {}", c.x);
    println!("y: {}", c.y);
}
```

What we've done here is make another `struct`, `CircleBuilder`. We've defined our builder methods on it. We've also defined our `area()` method on `Circle`. We also made one more method on `CircleBuilder`: `finalize()`. This method creates our final `Circle` from the builder. Now, we've used the type system to enforce our concerns: we can use the methods on `CircleBuilder` to constrain making `Circle`s in any way we choose.

4.16 Vectors

A 'vector' is a dynamic or 'growable' array, implemented as the standard library type `Vec<T>`[73]. The `T` means that we can have vectors of any type (see the chapter on

[73] `../std/vec/index.html`

generics[74] for more). Vectors always allocate their data on the heap. You can create them with the `vec!` macro:

```
let v = vec![1, 2, 3, 4, 5]; // v: Vec<i32>
```

(Notice that unlike the `println!` macro we've used in the past, we use square brackets `[]` with `vec!` macro. Rust allows you to use either in either situation, this is just convention.)

There's an alternate form of `vec!` for repeating an initial value:

```
let v = vec![0; 10]; // ten zeroes
```

Accessing elements

To get the value at a particular index in the vector, we use `[]`s:

```
let v = vec![1, 2, 3, 4, 5];

println!("The third element of v is {}", v[2]);
```

The indices count from `0`, so the third element is `v[2]`.

It's also important to note that you must index with the `usize` type:

```
let v = vec![1, 2, 3, 4, 5];

let i: usize = 0;
let j: i32 = 0;

// works
v[i];

// doesn't
v[j];
```

Indexing with a non-`usize` type gives an error that looks like this:

```
error: the trait `core::ops::Index<i32>` is not implemented for the type
`collections::vec::Vec<_>` [E0277]
v[j];
^~~~
note: the type `collections::vec::Vec<_>` cannot be indexed by `i32`
error: aborting due to previous error
```

There's a lot of punctuation in that message, but the core of it makes sense: you cannot index with an `i32`.

[74] `generics.html`

Iterating

Once you have a vector, you can iterate through its elements with `for`. There are three versions:

```
let mut v = vec![1, 2, 3, 4, 5];

for i in &v {
    println!("A reference to {}", i);
}

for i in &mut v {
    println!("A mutable reference to {}", i);
}

for i in v {
    println!("Take ownership of the vector and its element {}", i);
}
```

Vectors have many more useful methods, which you can read about in their API documentation[75].

4.17 Strings

Strings are an important concept for any programmer to master. Rust's string handling system is a bit different from other languages, due to its systems focus. Any time you have a data structure of variable size, things can get tricky, and strings are a re-sizable data structure. That being said, Rust's strings also work differently than in some other systems languages, such as C.

Let's dig into the details. A 'string' is a sequence of Unicode scalar values encoded as a stream of UTF-8 bytes. All strings are guaranteed to be a valid encoding of UTF-8 sequences. Additionally, unlike some systems languages, strings are not null-terminated and can contain null bytes.

Rust has two main types of strings: `&str` and `String`. Let's talk about `&str` first. These are called 'string slices'. A string slice has a fixed size, and cannot be mutated. It is a reference to a sequence of UTF-8 bytes.

```
let greeting = "Hello there."; // greeting: &'static str
```

`"Hello there."` is a string literal and its type is `&'static str`. A string literal is a string slice that is statically allocated, meaning that it's saved inside our compiled program, and exists for the entire duration it runs. The `greeting` binding is a reference to this statically allocated string. Any function expecting a string slice will also accept a string literal.

String literals can span multiple lines. There are two forms. The first will include the newline and the leading spaces:

[75] `../std/vec/index.html`

```
let s = "foo
    bar";

assert_eq!("foo\n        bar", s);
```

The second, with a `\`, trims the spaces and the newline:

```
let s = "foo\
    bar";

assert_eq!("foobar", s);
```

Rust has more than just `&str`s though. A `String`, is a heap-allocated string. This string is growable, and is also guaranteed to be UTF-8. `String`s are commonly created by converting from a string slice using the `to_string` method.

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

s.push_str(", world.");
println!("{}", s);
```

`String`s will coerce into `&str` with an `&`:

```
fn takes_slice(slice: &str) {
    println!("Got: {}", slice);
}

fn main() {
    let s = "Hello".to_string();
    takes_slice(&s);
}
```

This coercion does not happen for functions that accept one of `&str`'s traits instead of `&str`. For example, `TcpStream::connect`[76] has a parameter of type `ToSocketAddrs`. A `&str` is okay but a `String` must be explicitly converted using `&*`.

```
use std::net::TcpStream;

TcpStream::connect("192.168.0.1:3000"); // &str parameter

let addr_string = "192.168.0.1:3000".to_string();
TcpStream::connect(&*addr_string); // convert addr_string to &str
```

Viewing a `String` as a `&str` is cheap, but converting the `&str` to a `String` involves allocating memory. No reason to do that unless you have to!

[76] `../std/net/struct.TcpStream.html#method.connect`

Indexing

Because strings are valid UTF-8, strings do not support indexing:

```
let s = ”hello”;

println!(”The first letter of s is {}”, s[0]); // ERROR!!!
```

Usually, access to a vector with `[]` is very fast. But, because each character in a UTF-8 encoded string can be multiple bytes, you have to walk over the string to find the nth letter of a string. This is a significantly more expensive operation, and we don’t want to be misleading. Furthermore, ‘letter’ isn’t something defined in Unicode, exactly. We can choose to look at a string as individual bytes, or as codepoints:

```
let hachiko = ””;

for b in hachiko.as_bytes() {
    print!(”{}, ”, b);
}

println!(””);

for c in hachiko.chars() {
    print!(”{}, ”, c);
}

println!(””);
```

This prints:

```
229, 191, 160, 231, 138, 172, 227, 131, 143, 227, 131, 129, 229, 133, 172,
, , , , ,
```

As you can see, there are more bytes than `chars`.
You can get something similar to an index like this:

```
# let hachiko = ””;
let dog = hachiko.chars().nth(1); // kinda like hachiko[1]
```

This emphasizes that we have to walk from the beginning of the list of `chars`.

Slicing

You can get a slice of a string with slicing syntax:

```
let dog = ”hachiko”;
let hachi = &dog[0..5];
```

But note that these are *byte* offsets, not *character* offsets. So this will fail at runtime:

```
let dog = "";
let hachi = &dog[0..2];
```

with this error:

```
thread '<main>' panicked at 'index 0 and/or 2 in `` do not lie on
character boundary'
```

Concatenation

If you have a `String`, you can concatenate a `&str` to the end of it:

```
let hello = "Hello ".to_string();
let world = "world!";

let hello_world = hello + world;
```

But if you have two `String`s, you need an `&`:

```
let hello = "Hello ".to_string();
let world = "world!".to_string();

let hello_world = hello + &world;
```

This is because `&String` can automatically coerce to a `&str`. This is a feature called ‘`Deref` coercions[77]’.

4.18 Generics

Sometimes, when writing a function or data type, we may want it to work for multiple types of arguments. In Rust, we can do this with generics. Generics are called ‘parametric polymorphism’ in type theory, which means that they are types or functions that have multiple forms (‘poly’ is multiple, ‘morph’ is form) over a given parameter (‘parametric’).

Anyway, enough type theory, let’s check out some generic code. Rust’s standard library provides a type, `Option<T>`, that’s generic:

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

The `<T>` part, which you’ve seen a few times before, indicates that this is a generic data type. Inside the declaration of our `enum`, wherever we see a `T`, we substitute that type for the same type used in the generic. Here’s an example of using `Option<T>`, with some extra type annotations:

[77] `deref-coercions.html`

```
let x: Option<i32> = Some(5);
```

In the type declaration, we say `Option<i32>`. Note how similar this looks to `Option<T>`. So, in this particular `Option`, `T` has the value of `i32`. On the right-hand side of the binding, we make a `Some(T)`, where `T` is `5`. Since that's an `i32`, the two sides match, and Rust is happy. If they didn't match, we'd get an error:

```
let x: Option<f64> = Some(5);
// error: mismatched types: expected ‘core::option::Option<f64>‘,
// found ‘core::option::Option<_>‘ (expected f64 but found integral variable)
```

That doesn't mean we can't make `Option<T>`s that hold an `f64`! They just have to match up:

```
let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

This is just fine. One definition, multiple uses.

Generics don't have to only be generic over one type. Consider another type from Rust's standard library that's similar, `Result<T, E>`:

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

This type is generic over *two* types: `T` and `E`. By the way, the capital letters can be any letter you'd like. We could define `Result<T, E>` as:

```
enum Result<A, Z> {
    Ok(A),
    Err(Z),
}
```

if we wanted to. Convention says that the first generic parameter should be `T`, for 'type', and that we use `E` for 'error'. Rust doesn't care, however.

The `Result<T, E>` type is intended to be used to return the result of a computation, and to have the ability to return an error if it didn't work out.

Generic functions

We can write functions that take generic types with a similar syntax:

```
fn takes_anything<T>(x: T) {
    // do something with x
}
```

The syntax has two parts: the `<T>` says "this function is generic over one type, `T`", and the `x: T` says "x has the type `T`."

Multiple arguments can have the same generic type:

```
fn takes_two_of_the_same_things<T>(x: T, y: T) {
    // ...
}
```

We could write a version that takes multiple types:

```
fn takes_two_things<T, U>(x: T, y: U) {
    // ...
}
```

Generic structs

You can store a generic type in a `struct` as well:

```
struct Point<T> {
    x: T,
    y: T,
}

let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

Similar to functions, the `<T>` is where we declare the generic parameters, and we then use `x: T` in the type declaration, too.

When you want to add an implementation for the generic `struct`, you just declare the type parameter after the `impl`:

```
# struct Point<T> {
#     x: T,
#     y: T,
# }
#
impl<T> Point<T> {
    fn swap(&mut self) {
        std::mem::swap(&mut self.x, &mut self.y);
    }
}
```

So far you've seen generics that take absolutely any type. These are useful in many cases: you've already seen `Option<T>`, and later you'll meet universal container types like `Vec<T>`[78]. On the other hand, often you want to trade that flexibility for increased expressive power. Read about trait bounds[79] to see why and how.

[78] `../std/vec/struct.Vec.html`
[79] `traits.html`

4.19 Traits

A trait is a language feature that tells the Rust compiler about functionality a type must provide.

Do you remember the `impl` keyword, used to call a function with method syntax[80]?

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

Traits are similar, except that we define a trait with just the method signature, then implement the trait for that struct. Like this:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

As you can see, the `trait` block looks very similar to the `impl` block, but we don't define a body, just a type signature. When we `impl` a trait, we use `impl Trait for Item`, rather than just `impl Item`.

Trait bounds on generic functions

Traits are useful because they allow a type to make certain promises about its behavior. Generic functions can exploit this to constrain, or bound[81], the types they accept. Consider this function, which does not compile:

[80] `method-syntax.html`
[81] `glossary.html#bounds`

```
fn print_area<T>(shape: T) {
    println!(”This shape has an area of {}”, shape.area());
}
```

Rust complains:

```
error: no method named ‘area‘ found for type ‘T‘ in the current scope
```

Because `T` can be any type, we can’t be sure that it implements the `area` method. But we can add a trait bound to our generic `T`, ensuring that it does:

```
# trait HasArea {
#     fn area(&self) -> f64;
# }
fn print_area<T: HasArea>(shape: T) {
    println!(”This shape has an area of {}”, shape.area());
}
```

The syntax `<T: HasArea>` means “any type that implements the `HasArea` trait.” Because traits define function type signatures, we can be sure that any type which implements `HasArea` will have an `.area()` method.

Here’s an extended example of how this works:

```
trait HasArea {
    fn area(&self) -> f64;
}

struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct Square {
    x: f64,
    y: f64,
    side: f64,
}

impl HasArea for Square {
    fn area(&self) -> f64 {
        self.side * self.side
```

```
    }
}

fn print_area<T: HasArea>(shape: T) {
    println!(”This shape has an area of {}”, shape.area());
}

fn main() {
    let c = Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
    };

    let s = Square {
        x: 0.0f64,
        y: 0.0f64,
        side: 1.0f64,
    };

    print_area(c);
    print_area(s);
}
```

This program outputs:

```
This shape has an area of 3.141593
This shape has an area of 1
```

As you can see, `print_area` is now generic, but also ensures that we have passed in the correct types. If we pass in an incorrect type:

```
print_area(5);
```

We get a compile-time error:

```
error: the trait ‘HasArea‘ is not implemented for the type ‘_‘ [E0277]
```

Trait bounds on generic structs

Your generic structs can also benefit from trait bounds. All you need to do is append the bound when you declare type parameters. Here is a new type `Rectangle<T>` and its operation `is_square()`:

```
struct Rectangle<T> {
    x: T,
    y: T,
    width: T,
```

```
    height: T,
}

impl<T: PartialEq> Rectangle<T> {
    fn is_square(&self) -> bool {
        self.width == self.height
    }
}

fn main() {
    let mut r = Rectangle {
        x: 0,
        y: 0,
        width: 47,
        height: 47,
    };

    assert!(r.is_square());

    r.height = 42;
    assert!(!r.is_square());
}
```

`is_square()` needs to check that the sides are equal, so the sides must be of a type that implements the `core::cmp::PartialEq`[82] trait:

```
impl<T: PartialEq> Rectangle<T> { ... }
```

Now, a rectangle can be defined in terms of any type that can be compared for equality. Here we defined a new struct `Rectangle` that accepts numbers of any precision—really, objects of pretty much any type—as long as they can be compared for equality. Could we do the same for our `HasArea` structs, `Square` and `Circle`? Yes, but they need multiplication, and to work with that we need to know more about operator traits[83].

4.19.1 Rules for implementing traits

So far, we've only added trait implementations to structs, but you can implement a trait for any type. So technically, we *could* implement `HasArea` for `i32`:

```
trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for i32 {
    fn area(&self) -> f64 {
        println!("this is silly");
```

[82] `../core/cmp/trait.PartialEq.html`
[83] `operators-and-overloading.html`

```
        *self as f64
    }
}

5.area();
```

It is considered poor style to implement methods on such primitive types, even though it is possible.

This may seem like the Wild West, but there are two restrictions around implementing traits that prevent this from getting out of hand. The first is that if the trait isn't defined in your scope, it doesn't apply. Here's an example: the standard library provides a `Write`[84] trait which adds extra functionality to `File`s, for doing file I/O. By default, a `File` won't have its methods:

```
let mut f = std::fs::File::open("foo.txt").expect("Couldn't open foo.txt");
let buf = b"whatever"; // byte string literal. buf: &[u8; 8]
let result = f.write(buf);
# result.unwrap(); // ignore the error
```

Here's the error:

```
error: type `std::fs::File` does not implement any method in scope named `write`
let result = f.write(buf);
               ^~~~~~~~~~
```

We need to `use` the `Write` trait first:

```
use std::io::Write;

let mut f = std::fs::File::open("foo.txt").expect("Couldn't open foo.txt");
let buf = b"whatever";
let result = f.write(buf);
# result.unwrap(); // ignore the error
```

This will compile without error.

This means that even if someone does something bad like add methods to `i32`, it won't affect you, unless you `use` that trait.

There's one more restriction on implementing traits: either the trait, or the type you're writing the `impl` for, must be defined by you. So, we could implement the `HasArea` type for `i32`, because `HasArea` is in our code. But if we tried to implement `ToString`, a trait provided by Rust, for `i32`, we could not, because neither the trait nor the type are in our code.

One last thing about traits: generic functions with a trait bound use 'monomorphization' (mono: one, morph: form), so they are statically dispatched. What's that mean? Check out the chapter on trait objects[85] for more details.

[84] `../std/io/trait.Write.html`
[85] `trait-objects.html`

4.19.2 Multiple trait bounds

You’ve seen that you can bound a generic type parameter with a trait:

```
fn foo<T: Clone>(x: T) {
    x.clone();
}
```

If you need more than one bound, you can use `+`:

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

fn foo<T: Clone + Debug>(x: T) {
    x.clone();
    println!(”{:?}”, x);
}
```

`T` now needs to be both `Clone` as well as `Debug`.

4.19.3 Where clause

Writing functions with only a few generic types and a small number of trait bounds isn’t too bad, but as the number increases, the syntax gets increasingly awkward:

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

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!(”{:?}”, y);
}
```

The name of the function is on the far left, and the parameter list is on the far right. The bounds are getting in the way.

Rust has a solution, and it’s called a ‘`where` clause’:

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

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!(”{:?}”, y);
}

fn bar<T, K>(x: T, y: K) where T: Clone, K: Clone + Debug {
    x.clone();
    y.clone();
    println!(”{:?}”, y);
```

```
}

fn main() {
    foo(”Hello”, ”world”);
    bar(”Hello”, ”world”);
}
```

`foo()` uses the syntax we showed earlier, and `bar()` uses a `where` clause. All you need to do is leave off the bounds when defining your type parameters, and then add `where` after the parameter list. For longer lists, whitespace can be added:

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

fn bar<T, K>(x: T, y: K)
    where T: Clone,
          K: Clone + Debug {

    x.clone();
    y.clone();
    println!(”{:?}”, y);
}
```

This flexibility can add clarity in complex situations.

`where` is also more powerful than the simpler syntax. For example:

```
trait ConvertTo<Output> {
    fn convert(&self) -> Output;
}

impl ConvertTo<i64> for i32 {
    fn convert(&self) -> i64 { *self as i64 }
}

// can be called with T == i32
fn normal<T: ConvertTo<i64>>(x: &T) -> i64 {
    x.convert()
}

// can be called with T == i64
fn inverse<T>() -> T
        // this is using ConvertTo as if it were ”ConvertTo<i64>”
        where i32: ConvertTo<T> {
    42.convert()
}
```

This shows off the additional feature of `where` clauses: they allow bounds where the left-hand side is an arbitrary type (`i32` in this case), not just a plain type parameter (like `T`). In this example, `i32` must implement `ConvertTo<T>`. Rather than defining what `i32` is (since that’s obvious), the `where` clause here is a constraint on `T`.

4.19.4 Default methods

If you already know how a typical implementor will define a method, you can let your trait supply a default:

```
trait Foo {
    fn is_valid(&self) -> bool;

    fn is_invalid(&self) -> bool { !self.is_valid() }
}
```

Implementors of the `Foo` trait need to implement `is_valid()`, but they don't need to implement `is_invalid()`. They'll get this default behavior. They can override the default if they so choose:

```
# trait Foo {
#     fn is_valid(&self) -> bool;
#
#     fn is_invalid(&self) -> bool { !self.is_valid() }
# }
struct UseDefault;

impl Foo for UseDefault {
    fn is_valid(&self) -> bool {
        println!(”Called UseDefault.is_valid.”);
        true
    }
}

struct OverrideDefault;

impl Foo for OverrideDefault {
    fn is_valid(&self) -> bool {
        println!(”Called OverrideDefault.is_valid.”);
        true
    }

    fn is_invalid(&self) -> bool {
        println!(”Called OverrideDefault.is_invalid!”);
        true // this implementation is a self-contradiction!
    }
}

let default = UseDefault;
assert!(!default.is_invalid()); // prints ”Called UseDefault.is_valid.”

let over = OverrideDefault;
assert!(over.is_invalid()); // prints ”Called OverrideDefault.is_invalid!”
```

4.19.5 Inheritance

Sometimes, implementing a trait requires implementing another trait:

```
trait Foo {
    fn foo(&self);
}

trait FooBar : Foo {
    fn foobar(&self);
}
```

Implementors of `FooBar` must also implement `Foo`, like this:

```
# trait Foo {
#     fn foo(&self);
# }
# trait FooBar : Foo {
#     fn foobar(&self);
# }
struct Baz;

impl Foo for Baz {
    fn foo(&self) { println!(”foo”); }
}

impl FooBar for Baz {
    fn foobar(&self) { println!(”foobar”); }
}
```

If we forget to implement `Foo`, Rust will tell us:

```
error: the trait ‘main::Foo‘ is not implemented for the type ‘main::Baz‘ [E0277]
```

4.19.6 Deriving

Implementing traits like `Debug` and `Default` over and over again can become quite tedious. For that reason, Rust provides an attribute[86] that allows you to let Rust automatically implement traits for you:

```
#[derive(Debug)]
struct Foo;

fn main() {
    println!(”{:?}”, Foo);
}
```

[86] `attributes.html`

However, deriving is limited to a certain set of traits:

- `Clone`[87]
- `Copy`[88]
- `Debug`[89]
- `Default`[90]
- `Eq`[91]
- `Hash`[92]
- `Ord`[93]
- `PartialEq`[94]
- `PartialOrd`[95]

4.20 Drop

Now that we’ve discussed traits, let’s talk about a particular trait provided by the Rust standard library, `Drop`[96]. The `Drop` trait provides a way to run some code when a value goes out of scope. For example:

```
struct HasDrop;

impl Drop for HasDrop {
    fn drop(&mut self) {
        println!(”Dropping!”);
    }
}

fn main() {
    let x = HasDrop;

    // do stuff

} // x goes out of scope here
```

When `x` goes out of scope at the end of `main()`, the code for `Drop` will run. `Drop` has one method, which is also called `drop()`. It takes a mutable reference to `self`.

That’s it! The mechanics of `Drop` are very simple, but there are some subtleties. For example, values are dropped in the opposite order they are declared. Here’s another example:

[87] ../core/clone/trait.Clone.html
[88] ../core/marker/trait.Copy.html
[89] ../core/fmt/trait.Debug.html
[90] ../core/default/trait.Default.html
[91] ../core/cmp/trait.Eq.html
[92] ../core/hash/trait.Hash.html
[93] ../core/cmp/trait.Ord.html
[94] ../core/cmp/trait.PartialEq.html
[95] ../core/cmp/trait.PartialOrd.html
[96] ../std/ops/trait.Drop.html

```
struct Firework {
    strength: i32,
}

impl Drop for Firework {
    fn drop(&mut self) {
        println!(”BOOM times {}!!!”, self.strength);
    }
}

fn main() {
    let firecracker = Firework { strength: 1 };
    let tnt = Firework { strength: 100 };
}
```

This will output:

```
BOOM times 100!!!
BOOM times 1!!!
```

The TNT goes off before the firecracker does, because it was declared afterwards. Last in, first out.

So what is `Drop` good for? Generally, `Drop` is used to clean up any resources associated with a `struct`. For example, the `Arc<T>` type[97] is a reference-counted type. When `Drop` is called, it will decrement the reference count, and if the total number of references is zero, will clean up the underlying value.

4.21 if let

`if let` allows you to combine `if` and `let` together to reduce the overhead of certain kinds of pattern matches.

For example, let’s say we have some sort of `Option<T>`. We want to call a function on it if it’s `Some<T>`, but do nothing if it’s `None`. That looks like this:

```
# let option = Some(5);
# fn foo(x: i32) { }
match option {
    Some(x) => { foo(x) },
    None => {},
}
```

We don’t have to use `match` here, for example, we could use `if`:

```
# let option = Some(5);
# fn foo(x: i32) { }
```

[97] ../std/sync/struct.Arc.html

```
if option.is_some() {
    let x = option.unwrap();
    foo(x);
}
```

Neither of these options is particularly appealing. We can use `if let` to do the same thing in a nicer way:

```
# let option = Some(5);
# fn foo(x: i32) { }
if let Some(x) = option {
    foo(x);
}
```

If a pattern[98] matches successfully, it binds any appropriate parts of the value to the identifiers in the pattern, then evaluates the expression. If the pattern doesn’t match, nothing happens.

If you want to do something else when the pattern does not match, you can use `else`:

```
# let option = Some(5);
# fn foo(x: i32) { }
# fn bar() { }
if let Some(x) = option {
    foo(x);
} else {
    bar();
}
```

`while let`

In a similar fashion, `while let` can be used when you want to conditionally loop as long as a value matches a certain pattern. It turns code like this:

```
# let option: Option<i32> = None;
loop {
    match option {
        Some(x) => println!(”{}”, x),
        None => break,
    }
}
```

Into code like this:

```
# let option: Option<i32> = None;
while let Some(x) = option {
    println!(”{}”, x);
}
```

[98] `patterns.html`

4.22 Trait Objects

When code involves polymorphism, there needs to be a mechanism to determine which specific version is actually run. This is called 'dispatch'. There are two major forms of dispatch: static dispatch and dynamic dispatch. While Rust favors static dispatch, it also supports dynamic dispatch through a mechanism called 'trait objects'.

Background

For the rest of this chapter, we'll need a trait and some implementations. Let's make a simple one, `Foo`. It has one method that is expected to return a `String`.

```
trait Foo {
    fn method(&self) -> String;
}
```

We'll also implement this trait for `u8` and `String`:

```
# trait Foo { fn method(&self) -> String; }
impl Foo for u8 {
    fn method(&self) -> String { format!("u8: {}", *self) }
}

impl Foo for String {
    fn method(&self) -> String { format!("string: {}", *self) }
}
```

Static dispatch

We can use this trait to perform static dispatch with trait bounds:

```
# trait Foo { fn method(&self) -> String; }
# impl Foo for u8 { fn method(&self) -> String { format!("u8: {}", *self) } }
# impl Foo for String { fn method(&self) -> String { format!("string: {}", *self) } }
fn do_something<T: Foo>(x: T) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something(x);
    do_something(y);
}
```

Rust uses 'monomorphization' to perform static dispatch here. This means that Rust will create a special version of `do_something()` for both `u8` and `String`, and then replace the call sites with calls to these specialized functions. In other words, Rust generates something like this:

```
# trait Foo { fn method(&self) -> String; }
# impl Foo for u8 { fn method(&self) -> String { format!("u8: {}", *self) } }
# impl Foo for String { fn method(&self) -> String { format!("string: {}", *self) } }
fn do_something_u8(x: u8) {
    x.method();
}

fn do_something_string(x: String) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something_u8(x);
    do_something_string(y);
}
```

This has a great upside: static dispatch allows function calls to be inlined because the callee is known at compile time, and inlining is the key to good optimization. Static dispatch is fast, but it comes at a tradeoff: 'code bloat', due to many copies of the same function existing in the binary, one for each type.

Furthermore, compilers aren't perfect and may "optimize" code to become slower. For example, functions inlined too eagerly will bloat the instruction cache (cache rules everything around us). This is part of the reason that `#[inline]` and `#[inline(always)]` should be used carefully, and one reason why using a dynamic dispatch is sometimes more efficient.

However, the common case is that it is more efficient to use static dispatch, and one can always have a thin statically-dispatched wrapper function that does a dynamic dispatch, but not vice versa, meaning static calls are more flexible. The standard library tries to be statically dispatched where possible for this reason.

Dynamic dispatch

Rust provides dynamic dispatch through a feature called 'trait objects'. Trait objects, like `&Foo` or `Box<Foo>`, are normal values that store a value of *any* type that implements the given trait, where the precise type can only be known at runtime.

A trait object can be obtained from a pointer to a concrete type that implements the trait by *casting* it (e.g. `&x as &Foo`) or *coercing* it (e.g. using `&x` as an argument to a function that takes `&Foo`).

These trait object coercions and casts also work for pointers like `&mut T` to `&mut Foo` and `Box<T>` to `Box<Foo>`, but that's all at the moment. Coercions and casts are identical.

This operation can be seen as 'erasing' the compiler's knowledge about the specific type of the pointer, and hence trait objects are sometimes referred to as 'type erasure'.

Coming back to the example above, we can use the same trait to perform dynamic dispatch with trait objects by casting:

```
# trait Foo { fn method(&self) -> String; }
# impl Foo for u8 { fn method(&self) -> String { format!("u8: {}", *self) } }
# impl Foo for String { fn method(&self) -> String { format!("string: {}", *self) } }

fn do_something(x: &Foo) {
    x.method();
}

fn main() {
    let x = 5u8;
    do_something(&x as &Foo);
}
```

or by coercing:

```
# trait Foo { fn method(&self) -> String; }
# impl Foo for u8 { fn method(&self) -> String { format!("u8: {}", *self) } }
# impl Foo for String { fn method(&self) -> String { format!("string: {}", *self) } }

fn do_something(x: &Foo) {
    x.method();
}

fn main() {
    let x = "Hello".to_string();
    do_something(&x);
}
```

A function that takes a trait object is not specialized to each of the types that implements `Foo`: only one copy is generated, often (but not always) resulting in less code bloat. However, this comes at the cost of requiring slower virtual function calls, and effectively inhibiting any chance of inlining and related optimizations from occurring.

Why pointers? Rust does not put things behind a pointer by default, unlike many managed languages, so types can have different sizes. Knowing the size of the value at compile time is important for things like passing it as an argument to a function, moving it about on the stack and allocating (and deallocating) space on the heap to store it.

For `Foo`, we would need to have a value that could be at least either a `String` (24 bytes) or a `u8` (1 byte), as well as any other type for which dependent crates may implement `Foo` (any number of bytes at all). There's no way to guarantee that this last point can work if the values are stored without a pointer, because those other types can be arbitrarily large.

Putting the value behind a pointer means the size of the value is not relevant when we are tossing a trait object around, only the size of the pointer itself.

Representation The methods of the trait can be called on a trait object via a special record of function pointers traditionally called a 'vtable' (created and managed by the compiler).

Trait objects are both simple and complicated: their core representation and layout is quite straight-forward, but there are some curly error messages and surprising behaviors to discover.

Let's start simple, with the runtime representation of a trait object. The `std::raw` module contains structs with layouts that are the same as the complicated built-in types, including trait objects[99]:

```
# mod foo {
pub struct TraitObject {
    pub data: *mut (),
    pub vtable: *mut (),
}
# }
```

That is, a trait object like `&Foo` consists of a 'data' pointer and a 'vtable' pointer.

The data pointer addresses the data (of some unknown type `T`) that the trait object is storing, and the vtable pointer points to the vtable ('virtual method table') corresponding to the implementation of `Foo` for `T`.

A vtable is essentially a struct of function pointers, pointing to the concrete piece of machine code for each method in the implementation. A method call like `trait_object.method()` will retrieve the correct pointer out of the vtable and then do a dynamic call of it. For example:

```
struct FooVtable {
    destructor: fn(*mut ()),
    size: usize,
    align: usize,
    method: fn(*const ()) -> String,
}

// u8:

fn call_method_on_u8(x: *const ()) -> String {
    // the compiler guarantees that this function is only called
    // with `x` pointing to a u8
    let byte: &u8 = unsafe { &*(x as *const u8) };

    byte.method()
}

static Foo_for_u8_vtable: FooVtable = FooVtable {
    destructor: /* compiler magic */,
    size: 1,
```

[99] ../std/raw/struct.TraitObject.html

```
    align: 1,

    // cast to a function pointer
    method: call_method_on_u8 as fn(*const ()) -> String,
};


// String:

fn call_method_on_String(x: *const ()) -> String {
    // the compiler guarantees that this function is only called
    // with `x` pointing to a String
    let string: &String = unsafe { &*(x as *const String) };

    string.method()
}

static Foo_for_String_vtable: FooVtable = FooVtable {
    destructor: /* compiler magic */,
    // values for a 64-bit computer, halve them for 32-bit ones
    size: 24,
    align: 8,

    method: call_method_on_String as fn(*const ()) -> String,
};
```

The `destructor` field in each vtable points to a function that will clean up any resources of the vtable's type: for `u8` it is trivial, but for `String` it will free the memory. This is necessary for owning trait objects like `Box<Foo>`, which need to clean-up both the `Box` allocation as well as the internal type when they go out of scope. The `size` and `align` fields store the size of the erased type, and its alignment requirements; these are essentially unused at the moment since the information is embedded in the destructor, but will be used in the future, as trait objects are progressively made more flexible.

Suppose we've got some values that implement `Foo`. The explicit form of construction and use of `Foo` trait objects might look a bit like (ignoring the type mismatches: they're all just pointers anyway):

```
let a: String = "foo".to_string();
let x: u8 = 1;

// let b: &Foo = &a;
let b = TraitObject {
    // store the data
    data: &a,
    // store the methods
    vtable: &Foo_for_String_vtable
};
```

```
// let y: &Foo = x;
let y = TraitObject {
    // store the data
    data: &x,
    // store the methods
    vtable: &Foo_for_u8_vtable
};

// b.method();
(b.vtable.method)(b.data);

// y.method();
(y.vtable.method)(y.data);
```

Object Safety

Not every trait can be used to make a trait object. For example, vectors implement `Clone`, but if we try to make a trait object:

```
let v = vec![1, 2, 3];
let o = &v as &Clone;
```

We get an error:

```
error: cannot convert to a trait object because trait ‘core::clone::Clone‘ is not object-safe
let o = &v as &Clone;
        ^~
note: the trait cannot require that ‘Self : Sized‘
let o = &v as &Clone;
        ^~
```

The error says that `Clone` is not 'object-safe'. Only traits that are object-safe can be made into trait objects. A trait is object-safe if both of these are true:

- the trait does not require that `Self: Sized`
- all of its methods are object-safe

So what makes a method object-safe? Each method must require that `Self: Sized` or all of the following:

- must not have any type parameters
- must not use `Self`

Whew! As we can see, almost all of these rules talk about `Self`. A good intuition is "except in special circumstances, if your trait's method uses `Self`, it is not object-safe."

4.23 Closures

Sometimes it is useful to wrap up a function and *free variables* for better clarity and reuse. The free variables that can be used come from the enclosing scope and are 'closed over' when used in the function. From this, we get the name 'closures' and Rust provides a really great implementation of them, as we'll see.

4.23.1 Syntax

Closures look like this:

```
let plus_one = |x: i32| x + 1;

assert_eq!(2, plus_one(1));
```

We create a binding, `plus_one`, and assign it to a closure. The closure's arguments go between the pipes (`|`), and the body is an expression, in this case, `x + 1`. Remember that `{ }` is an expression, so we can have multi-line closures too:

```
let plus_two = |x| {
    let mut result: i32 = x;

    result += 1;
    result += 1;

    result
};

assert_eq!(4, plus_two(2));
```

You'll notice a few things about closures that are a bit different from regular named functions defined with `fn`. The first is that we did not need to annotate the types of arguments the closure takes or the values it returns. We can:

```
let plus_one = |x: i32| -> i32 { x + 1 };

assert_eq!(2, plus_one(1));
```

But we don't have to. Why is this? Basically, it was chosen for ergonomic reasons. While specifying the full type for named functions is helpful with things like documentation and type inference, the full type signatures of closures are rarely documented since they're anonymous, and they don't cause the kinds of error-at-a-distance problems that inferring named function types can.

The second is that the syntax is similar, but a bit different. I've added spaces here for easier comparison:

```
fn  plus_one_v1   (x: i32) -> i32 { x + 1 }
let plus_one_v2 = |x: i32| -> i32 { x + 1 };
let plus_one_v3 = |x: i32|          x + 1  ;
```

Small differences, but they're similar.

4.23.2 Closures and their environment

The environment for a closure can include bindings from its enclosing scope in addition to parameters and local bindings. It looks like this:

```
let num = 5;
let plus_num = |x: i32| x + num;

assert_eq!(10, plus_num(5));
```

This closure, `plus_num`, refers to a `let` binding in its scope: `num`. More specifically, it borrows the binding. If we do something that would conflict with that binding, we get an error. Like this one:

```
let mut num = 5;
let plus_num = |x: i32| x + num;

let y = &mut num;
```

Which errors with:

```
error: cannot borrow ‘num‘ as mutable because it is also borrowed as immutable
    let y = &mut num;
                ^~~
note: previous borrow of ‘num‘ occurs here due to use in closure; the immutable
  borrow prevents subsequent moves or mutable borrows of ‘num‘ until the borrow
  ends
    let plus_num = |x| x + num;
                    ^~~~~~~~~~
note: previous borrow ends here
fn main() {
    let mut num = 5;
    let plus_num = |x| x + num;

    let y = &mut num;
}
^
```

A verbose yet helpful error message! As it says, we can't take a mutable borrow on `num` because the closure is already borrowing it. If we let the closure go out of scope, we can:

```
let mut num = 5;
{
    let plus_num = |x: i32| x + num;

} // plus_num goes out of scope, borrow of num ends

let y = &mut num;
```

If your closure requires it, however, Rust will take ownership and move the environment instead. This doesn't work:

```
let nums = vec![1, 2, 3];

let takes_nums = || nums;

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

We get this error:

```
note: ‘nums‘ moved into closure environment here because it has type
  ‘[closure(()) -> collections::vec::Vec<i32>]‘, which is non-copyable
let takes_nums = || nums;
                 ^~~~~~~
```

`Vec<T>` has ownership over its contents, and therefore, when we refer to it in our closure, we have to take ownership of `nums`. It's the same as if we'd passed `nums` to a function that took ownership of it.

`move` closures

We can force our closure to take ownership of its environment with the `move` keyword:

```
let num = 5;

let owns_num = move |x: i32| x + num;
```

Now, even though the keyword is `move`, the variables follow normal move semantics. In this case, `5` implements `Copy`, and so `owns_num` takes ownership of a copy of `num`. So what's the difference?

```
let mut num = 5;

{
    let mut add_num = |x: i32| num += x;

    add_num(5);
}

assert_eq!(10, num);
```

So in this case, our closure took a mutable reference to `num`, and then when we called `add_num`, it mutated the underlying value, as we'd expect. We also needed to declare `add_num` as `mut` too, because we're mutating its environment.

If we change to a `move` closure, it's different:

```
let mut num = 5;

{
    let mut add_num = move |x: i32| num += x;

    add_num(5);
}

assert_eq!(5, num);
```

We only get `5`. Rather than taking a mutable borrow out on our `num`, we took ownership of a copy.

Another way to think about `move` closures: they give a closure its own stack frame. Without `move`, a closure may be tied to the stack frame that created it, while a `move` closure is self-contained. This means that you cannot generally return a non-`move` closure from a function, for example.

But before we talk about taking and returning closures, we should talk some more about the way that closures are implemented. As a systems language, Rust gives you tons of control over what your code does, and closures are no different.

4.23.3 Closure implementation

Rust's implementation of closures is a bit different than other languages. They are effectively syntax sugar for traits. You'll want to make sure to have read the traits chapter[100] before this one, as well as the chapter on trait objects[101].

Got all that? Good.

The key to understanding how closures work under the hood is something a bit strange: Using `()` to call a function, like `foo()`, is an overloadable operator. From this, everything else clicks into place. In Rust, we use the trait system to overload operators. Calling functions is no different. We have three separate traits to overload with:

```
# mod foo {
pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}

pub trait FnMut<Args> : FnOnce<Args> {
    extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait FnOnce<Args> {
    type Output;

    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
```

[100] `traits.html`
[101] `trait-objects.html`

```
}
# }
```

You'll notice a few differences between these traits, but a big one is `self`: `Fn` takes `&self`, `FnMut` takes `&mut self`, and `FnOnce` takes `self`. This covers all three kinds of `self` via the usual method call syntax. But we've split them up into three traits, rather than having a single one. This gives us a large amount of control over what kind of closures we can take.

The `|| {}` syntax for closures is sugar for these three traits. Rust will generate a struct for the environment, `impl` the appropriate trait, and then use it.

4.23.4 Taking closures as arguments

Now that we know that closures are traits, we already know how to accept and return closures: just like any other trait!

This also means that we can choose static vs dynamic dispatch as well. First, let's write a function which takes something callable, calls it, and returns the result:

```
fn call_with_one<F>(some_closure: F) -> i32
    where F : Fn(i32) -> i32 {

    some_closure(1)
}

let answer = call_with_one(|x| x + 2);

assert_eq!(3, answer);
```

We pass our closure, `|x| x + 2`, to `call_with_one`. It just does what it suggests: it calls the closure, giving it `1` as an argument.

Let's examine the signature of `call_with_one` in more depth:

```
fn call_with_one<F>(some_closure: F) -> i32
#    where F : Fn(i32) -> i32 {
#    some_closure(1) }
```

We take one parameter, and it has the type `F`. We also return a `i32`. This part isn't interesting. The next part is:

```
# fn call_with_one<F>(some_closure: F) -> i32
    where F : Fn(i32) -> i32 {
#   some_closure(1) }
```

Because `Fn` is a trait, we can bound our generic with it. In this case, our closure takes a `i32` as an argument and returns an `i32`, and so the generic bound we use is `Fn(i32) -> i32`.

There's one other key point here: because we're bounding a generic with a trait, this will get monomorphized, and therefore, we'll be doing static dispatch into the closure.

That's pretty neat. In many languages, closures are inherently heap allocated, and will always involve dynamic dispatch. In Rust, we can stack allocate our closure environment, and statically dispatch the call. This happens quite often with iterators and their adapters, which often take closures as arguments.

Of course, if we want dynamic dispatch, we can get that too. A trait object handles this case, as usual:

```
fn call_with_one(some_closure: &Fn(i32) -> i32) -> i32 {
    some_closure(1)
}

let answer = call_with_one(&|x| x + 2);

assert_eq!(3, answer);
```

Now we take a trait object, a `&Fn`. And we have to make a reference to our closure when we pass it to `call_with_one`, so we use `&||`.

4.23.5 Function pointers and closures

A function pointer is kind of like a closure that has no environment. As such, you can pass a function pointer to any function expecting a closure argument, and it will work:

```
fn call_with_one(some_closure: &Fn(i32) -> i32) -> i32 {
    some_closure(1)
}

fn add_one(i: i32) -> i32 {
    i + 1
}

let f = add_one;

let answer = call_with_one(&f);

assert_eq!(2, answer);
```

In this example, we don't strictly need the intermediate variable `f`, the name of the function works just fine too:

```
let answer = call_with_one(&add_one);
```

4.23.6 Returning closures

It's very common for functional-style code to return closures in various situations. If you try to return a closure, you may run into an error. At first, it may seem strange, but we'll figure it out. Here's how you'd probably try to return a closure from a function:

```
fn factory() -> (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

This gives us these long, related errors:

```
error: the trait ‘core::marker::Sized‘ is not implemented for the type
‘core::ops::Fn(i32) -> i32‘ [E0277]
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~
note: ‘core::ops::Fn(i32) -> i32‘ does not have a constant size known at compile-time
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~
error: the trait ‘core::marker::Sized‘ is not implemented for the type ‘core::ops::Fn(i32) ->
let f = factory();
    ^
note: ‘core::ops::Fn(i32) -> i32‘ does not have a constant size known at compile-time
let f = factory();
    ^
```

In order to return something from a function, Rust needs to know what size the return type is. But since `Fn` is a trait, it could be various things of various sizes: many different types can implement `Fn`. An easy way to give something a size is to take a reference to it, as references have a known size. So we'd write this:

```
fn factory() -> &(Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

But we get another error:

```
error: missing lifetime specifier [E0106]
fn factory() -> &(Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~
```

Right. Because we have a reference, we need to give it a lifetime. But our `factory()` function takes no arguments, so elision[102] doesn't kick in here. Then what choices do we have? Try `'static`:

```
fn factory() -> &'static (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

But we get another error:

```
error: mismatched types:
 expected ‘&'static core::ops::Fn(i32) -> i32‘,
    found ‘[closure@<anon>:7:9: 7:20]‘
(expected &-ptr,
    found closure) [E0308]
         |x| x + num
         ^~~~~~~~~~~
```

This error is letting us know that we don't have a `&'static Fn(i32) -> i32`, we have a `[closure@<anon>:7:9: 7:20]`. Wait, what?

Because each closure generates its own environment `struct` and implementation of `Fn` and friends, these types are anonymous. They exist just solely for this closure. So Rust shows them as `closure@<anon>`, rather than some autogenerated name.

The error also points out that the return type is expected to be a reference, but what we are trying to return is not. Further, we cannot directly assign a `'static` lifetime to an object. So we'll take a different approach and return a 'trait object' by `Box`ing up the `Fn`. This *almost* works:

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(|x| x + num)
}
# fn main() {
let f = factory();

let answer = f(1);
assert_eq!(6, answer);
# }
```

[102] `lifetimes.html#lifetime-elision`

There’s just one last problem:

```
error: closure may outlive the current function, but it borrows ‘num‘,
which is owned by the current function [E0373]
Box::new(|x| x + num)
         ^~~~~~~~~~
```

Well, as we discussed before, closures borrow their environment. And in this case, our environment is based on a stack-allocated `5`, the `num` variable binding. So the borrow has a lifetime of the stack frame. So if we returned this closure, the function call would be over, the stack frame would go away, and our closure is capturing an environment of garbage memory! With one last fix, we can make this work:

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(move |x| x + num)
}
# fn main() {
let f = factory();

let answer = f(1);
assert_eq!(6, answer);
# }
```

By making the inner closure a `move Fn`, we create a new stack frame for our closure. By `Box`ing it up, we’ve given it a known size, and allowing it to escape our stack frame.

4.24 Universal Function Call Syntax

Sometimes, functions can have the same names. Consider this code:

```
trait Foo {
    fn f(&self);
}

trait Bar {
    fn f(&self);
}

struct Baz;

impl Foo for Baz {
    fn f(&self) { println!(”Baz’s impl of Foo”); }
}

impl Bar for Baz {
```

```
    fn f(&self) { println!(”Baz’s impl of Bar”); }
}

let b = Baz;
```

If we were to try to call `b.f()`, we’d get an error:

```
error: multiple applicable methods in scope [E0034]
b.f();
  ^~~
note: candidate #1 is defined in an impl of the trait ‘main::Foo‘ for the type
‘main::Baz‘
    fn f(&self) { println!(”Baz’s impl of Foo”); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: candidate #2 is defined in an impl of the trait ‘main::Bar‘ for the type
‘main::Baz‘
    fn f(&self) { println!(”Baz’s impl of Bar”); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

We need a way to disambiguate which method we need. This feature is called ‘universal function call syntax’, and it looks like this:

```
# trait Foo {
#     fn f(&self);
# }
# trait Bar {
#     fn f(&self);
# }
# struct Baz;
# impl Foo for Baz {
#     fn f(&self) { println!(”Baz’s impl of Foo”); }
# }
# impl Bar for Baz {
#     fn f(&self) { println!(”Baz’s impl of Bar”); }
# }
# let b = Baz;
Foo::f(&b);
Bar::f(&b);
```

Let’s break it down.

```
Foo::
Bar::
```

These halves of the invocation are the types of the two traits: `Foo` and `Bar`. This is what ends up actually doing the disambiguation between the two: Rust calls the one from the trait name you use.

```
f(&b)
```

When we call a method like `b.f()` using method syntax[103], Rust will automatically borrow `b` if `f()` takes `&self`. In this case, Rust will not, and so we need to pass an explicit `&b`.

4.24.1 Angle-bracket Form

The form of UFCS we just talked about:

```
Trait::method(args);
```

Is a short-hand. There's an expanded form of this that's needed in some situations:

```
<Type as Trait>::method(args);
```

The `<>::` syntax is a means of providing a type hint. The type goes inside the `<>`s. In this case, the type is `Type as Trait`, indicating that we want `Trait`'s version of `method` to be called here. The `as Trait` part is optional if it's not ambiguous. Same with the angle brackets, hence the shorter form.

Here's an example of using the longer form.

```
trait Foo {
    fn foo() -> i32;
}

struct Bar;

impl Bar {
    fn foo() -> i32 {
        20
    }
}

impl Foo for Bar {
    fn foo() -> i32 {
        10
    }
}

fn main() {
    assert_eq!(10, <Bar as Foo>::foo());
    assert_eq!(20, Bar::foo());
}
```

Using the angle bracket syntax lets you call the trait method instead of the inherent one.

[103] `method-syntax.html`

4.25 Crates and Modules

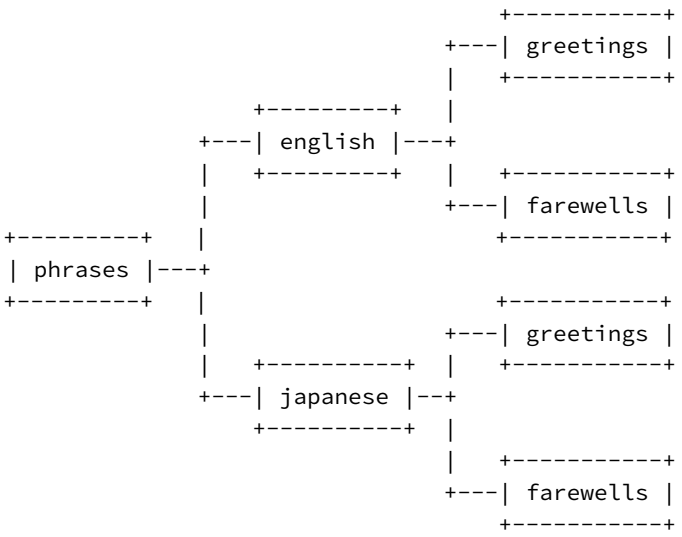
When a project starts getting large, it's considered good software engineering practice to split it up into a bunch of smaller pieces, and then fit them together. It's also important to have a well-defined interface, so that some of your functionality is private, and some is public. To facilitate these kinds of things, Rust has a module system.

4.25.1 Basic terminology: Crates and Modules

Rust has two distinct terms that relate to the module system: 'crate' and 'module'. A crate is synonymous with a 'library' or 'package' in other languages. Hence "Cargo" as the name of Rust's package management tool: you ship your crates to others with Cargo. Crates can produce an executable or a library, depending on the project.

Each crate has an implicit *root module* that contains the code for that crate. You can then define a tree of sub-modules under that root module. Modules allow you to partition your code within the crate itself.

As an example, let's make a *phrases* crate, which will give us various phrases in different languages. To keep things simple, we'll stick to 'greetings' and 'farewells' as two kinds of phrases, and use English and Japanese () as two languages for those phrases to be in. We'll use this module layout:



In this example, `phrases` is the name of our crate. All of the rest are modules. You can see that they form a tree, branching out from the crate *root*, which is the root of the tree: `phrases` itself.

Now that we have a plan, let's define these modules in code. To start, generate a new crate with Cargo:

```
$ cargo new phrases
$ cd phrases
```

If you remember, this generates a simple project for us:

```
$ tree .
.
├── Cargo.toml
└── src
    └── lib.rs

1 directory, 2 files
```

`src/lib.rs` is our crate root, corresponding to the `phrases` in our diagram above.

4.25.2 Defining Modules

To define each of our modules, we use the `mod` keyword. Let's make our `src/lib.rs` look like this:

```
mod english {
    mod greetings {
    }

    mod farewells {
    }
}

mod japanese {
    mod greetings {
    }

    mod farewells {
    }
}
```

After the `mod` keyword, you give the name of the module. Module names follow the conventions for other Rust identifiers: `lower_snake_case`. The contents of each module are within curly braces (`{}`).

Within a given `mod`, you can declare sub-mods. We can refer to sub-modules with double-colon (`::`) notation: our four nested modules are `english::greetings`, `english::farewells`, `japanese::greetings`, and `japanese::farewells`. Because these sub-modules are namespaced under their parent module, the names don't conflict: `english::greetings` and `japanese::greetings` are distinct, even though their names are both `greetings`.

Because this crate does not have a `main()` function, and is called `lib.rs`, Cargo will build this crate as a library:

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
$ ls target/debug
build  deps  examples  libphrases-a7448e02a0468eaa.rlib  native
```

`libphrases-hash.rlib` is the compiled crate. Before we see how to use this crate from another crate, let's break it up into multiple files.

4.25.3 Multiple file crates

If each crate were just one file, these files would get very large. It's often easier to split up crates into multiple files, and Rust supports this in two ways.

Instead of declaring a module like this:

```
mod english {
    // contents of our module go here
}
```

We can instead declare our module like this:

```
mod english;
```

If we do that, Rust will expect to find either a `english.rs` file, or a `english/mod.rs` file with the contents of our module.

Note that in these files, you don't need to re-declare the module: that's already been done with the initial `mod` declaration.

Using these two techniques, we can break up our crate into two directories and seven files:

```
$ tree .
.
├── Cargo.lock
├── Cargo.toml
├── src
│   ├── english
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   ├── japanese
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   └── lib.rs
└── target
    └── debug
        ├── build
        ├── deps
        ├── examples
        ├── libphrases-a7448e02a0468eaa.rlib
        └── native
```

`src/lib.rs` is our crate root, and looks like this:

```
mod english;
mod japanese;
```

These two declarations tell Rust to look for either `src/english.rs` and `src/japanese.rs`, or `src/english/mod.rs` and `src/japanese/mod.rs`, depending on our preference. In this case, because our modules have sub-modules, we've chosen the second. Both `src/english/mod.rs` and `src/japanese/mod.rs` look like this:

```
mod greetings;
mod farewells;
```

Again, these declarations tell Rust to look for either `src/english/greetings.rs` and `src/japanese/greetings.rs` or `src/english/farewells/mod.rs` and `src/japanese/farewells/mod.rs`. Because these sub-modules don't have their own sub-modules, we've chosen to make them `src/english/greetings.rs` and `src/japanese/farewells.rs`. Whew!

The contents of `src/english/greetings.rs` and `src/japanese/farewells.rs` are both empty at the moment. Let's add some functions.

Put this in `src/english/greetings.rs`:

```
fn hello() -> String {
    "Hello!".to_string()
}
```

Put this in `src/english/farewells.rs`:

```
fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

Put this in `src/japanese/greetings.rs`:

```
fn hello() -> String {
    "".to_string()
}
```

Of course, you can copy and paste this from this web page, or just type something else. It's not important that you actually put 'konnichiwa' to learn about the module system.

Put this in `src/japanese/farewells.rs`:

```
fn goodbye() -> String {
    "".to_string()
}
```

(This is 'Sayōnara', if you're curious.)

Now that we have some functionality in our crate, let's try to use it from another crate.

4.25.4 Importing External Crates

We have a library crate. Let’s make an executable crate that imports and uses our library.

Make a `src/main.rs` and put this in it (it won’t quite compile yet):

```
extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}", phrases::english::farewells::goodbye());

    println!("Hello in Japanese: {}", phrases::japanese::greetings::hello());
    println!("Goodbye in Japanese: {}", phrases::japanese::farewells::goodbye());
}
```

The `extern crate` declaration tells Rust that we need to compile and link to the `phrases` crate. We can then use `phrases`’ modules in this one. As we mentioned earlier, you can use double colons to refer to sub-modules and the functions inside of them.

(Note: when importing a crate that has dashes in its name “like-this”, which is not a valid Rust identifier, it will be converted by changing the dashes to underscores, so you would write `extern crate like_this;`.)

Also, Cargo assumes that `src/main.rs` is the crate root of a binary crate, rather than a library crate. Our package now has two crates: `src/lib.rs` and `src/main.rs`. This pattern is quite common for executable crates: most functionality is in a library crate, and the executable crate uses that library. This way, other programs can also use the library crate, and it’s also a nice separation of concerns.

This doesn’t quite work yet, though. We get four errors that look similar to this:

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:38: 4:72 error: function ‘hello‘ is private
src/main.rs:4     println!("Hello in English: {}", phrases::english::greetings::hello());
                                                   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: in expansion of format_args!
<std macros>:2:25: 2:58 note: expansion site
<std macros>:1:1: 2:62 note: in expansion of print!
<std macros>:3:1: 3:54 note: expansion site
<std macros>:1:1: 3:58 note: in expansion of println!
phrases/src/main.rs:4:5: 4:76 note: expansion site
```

By default, everything is private in Rust. Let’s talk about this in some more depth.

4.25.5 Exporting a Public Interface

Rust allows you to precisely control which aspects of your interface are public, and so private is the default. To make things public, you use the `pub` keyword. Let’s focus on the `english` module first, so let’s reduce our `src/main.rs` to just this:

```
extern crate phrases;

fn main() {
    println!(”Hello in English: {}”, phrases::english::greetings::hello());
    println!(”Goodbye in English: {}”, phrases::english::farewells::goodbye());
}
```

In our `src/lib.rs`, let's add `pub` to the `english` module declaration:

```
pub mod english;
mod japanese;
```

And in our `src/english/mod.rs`, let's make both `pub`:

```
pub mod greetings;
pub mod farewells;
```

In our `src/english/greetings.rs`, let's add `pub` to our `fn` declaration:

```
pub fn hello() -> String {
    ”Hello!”.to_string()
}
```

And also in `src/english/farewells.rs`:

```
pub fn goodbye() -> String {
    ”Goodbye.”.to_string()
}
```

Now, our crate compiles, albeit with warnings about not using the `japanese` functions:

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/japanese/greetings.rs:1:1: 3:2 warning: function is never used: ‘hello‘, #[warn(dead_code)
src/japanese/greetings.rs:1 fn hello() -> String {
src/japanese/greetings.rs:2     ””.to_string()
src/japanese/greetings.rs:3 }
src/japanese/farewells.rs:1:1: 3:2 warning: function is never used: ‘goodbye‘, #[warn(dead_cod
src/japanese/farewells.rs:1 fn goodbye() -> String {
src/japanese/farewells.rs:2     ””.to_string()
src/japanese/farewells.rs:3 }
     Running ‘target/debug/phrases‘
Hello in English: Hello!
Goodbye in English: Goodbye.
```

`pub` also applies to `structs` and their member fields. In keeping with Rust's tendency toward safety, simply making a `struct` public won't automatically make its members public: you must mark the fields individually with `pub`.

Now that our functions are public, we can use them. Great! However, typing out `phrases::english::greetings::hello()` is very long and repetitive. Rust has another keyword for importing names into the current scope, so that you can refer to them with shorter names. Let's talk about `use`.

4.25.6 Importing Modules with `use`

Rust has a `use` keyword, which allows us to import names into our local scope. Let's change our `src/main.rs` to look like this:

```
extern crate phrases;

use phrases::english::greetings;
use phrases::english::farewells;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());
}
```

The two `use` lines import each module into the local scope, so we can refer to the functions by a much shorter name. By convention, when importing functions, it's considered best practice to import the module, rather than the function directly. In other words, you *can* do this:

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::english::farewells::goodbye;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Goodbye in English: {}", goodbye());
}
```

But it is not idiomatic. This is significantly more likely to introduce a naming conflict. In our short program, it's not a big deal, but as it grows, it becomes a problem. If we have conflicting names, Rust will give a compilation error. For example, if we made the `japanese` functions public, and tried to do this:

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::japanese::greetings::hello;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Hello in Japanese: {}", hello());
}
```

Rust will give us a compile-time error:

```
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:5: 4:40 error: a value named ‘hello‘ has already been imported in this module [E
```

```
src/main.rs:4 use phrases::japanese::greetings::hello;
                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
error: aborting due to previous error
Could not compile ‘phrases‘.
```

If we’re importing multiple names from the same module, we don’t have to type it out twice. Instead of this:

```
use phrases::english::greetings;
use phrases::english::farewells;
```

We can use this shortcut:

```
use phrases::english::{greetings, farewells};
```

Re-exporting with `pub use`

You don’t just use `use` to shorten identifiers. You can also use it inside of your crate to re-export a function inside another module. This allows you to present an external interface that may not directly map to your internal code organization.

Let’s look at an example. Modify your `src/main.rs` to read like this:

```
extern crate phrases;

use phrases::english::{greetings,farewells};
use phrases::japanese;

fn main() {
    println!(”Hello in English: {}”, greetings::hello());
    println!(”Goodbye in English: {}”, farewells::goodbye());

    println!(”Hello in Japanese: {}”, japanese::hello());
    println!(”Goodbye in Japanese: {}”, japanese::goodbye());
}
```

Then, modify your `src/lib.rs` to make the `japanese` mod public:

```
pub mod english;
pub mod japanese;
```

Next, make the two functions public, first in `src/japanese/greetings.rs`:

```
pub fn hello() -> String {
    ””.to_string()
}
```

And then in `src/japanese/farewells.rs`:

```
pub fn goodbye() -> String {
    "".to_string()
}
```

Finally, modify your `src/japanese/mod.rs` to read like this:

```
pub use self::greetings::hello;
pub use self::farewells::goodbye;

mod greetings;
mod farewells;
```

The `pub use` declaration brings the function into scope at this part of our module hierarchy. Because we've `pub use`d this inside of our `japanese` module, we now have a `phrases::japanese::hello()` function and a `phrases::japanese::goodbye()` function, even though the code for them lives in `phrases::japanese::greetings::hello()` and `phrases::japanese::farewells::goodbye()`. Our internal organization doesn't define our external interface.

Here we have a `pub use` for each function we want to bring into the `japanese` scope. We could alternatively use the wildcard syntax to include everything from `greetings` into the current scope: `pub use self::greetings::*`.

What about the `self`? Well, by default, `use` declarations are absolute paths, starting from your crate root. `self` makes that path relative to your current place in the hierarchy instead. There's one more special form of `use`: you can `use super::` to reach one level up the tree from your current location. Some people like to think of `self` as `.` and `super` as `..`, from many shells' display for the current directory and the parent directory.

Outside of `use`, paths are relative: `foo::bar()` refers to a function inside of `foo` relative to where we are. If that's prefixed with `::`, as in `::foo::bar()`, it refers to a different `foo`, an absolute path from your crate root.

This will build and run:

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
     Running `target/debug/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
Hello in Japanese:
Goodbye in Japanese:
```

Complex imports

Rust offers several advanced options that can add compactness and convenience to your `extern crate` and `use` statements. Here is an example:

```
extern crate phrases as sayings;
```

```
use sayings::japanese::greetings as ja_greetings;
use sayings::japanese::farewells::*;
use sayings::english::{self, greetings as en_greetings, farewells as en_farewells};

fn main() {
    println!("Hello in English; {}", en_greetings::hello());
    println!("And in Japanese: {}", ja_greetings::hello());
    println!("Goodbye in English: {}", english::farewells::goodbye());
    println!("Again: {}", en_farewells::goodbye());
    println!("And in Japanese: {}", goodbye());
}
```

What's going on here?

First, both `extern crate` and `use` allow renaming the thing that is being imported. So the crate is still called "phrases", but here we will refer to it as "sayings". Similarly, the first `use` statement pulls in the `japanese::greetings` module from the crate, but makes it available as `ja_greetings` as opposed to simply `greetings`. This can help to avoid ambiguity when importing similarly-named items from different places.

The second `use` statement uses a star glob to bring in *all* symbols from the `sayings::japanese::farewells` module. As you can see we can later refer to the Japanese `goodbye` function with no module qualifiers. This kind of glob should be used sparingly.

The third `use` statement bears more explanation. It's using "brace expansion" globbing to compress three `use` statements into one (this sort of syntax may be familiar if you've written Linux shell scripts before). The uncompressed form of this statement would be:

```
use sayings::english;
use sayings::english::greetings as en_greetings;
use sayings::english::farewells as en_farewells;
```

As you can see, the curly brackets compress `use` statements for several items under the same path, and in this context `self` just refers back to that path. Note: The curly brackets cannot be nested or mixed with star globbing.

4.26 'const' and 'static'

Rust has a way of defining constants with the `const` keyword:

```
const N: i32 = 5;
```

Unlike `let`[104] bindings, you must annotate the type of a `const`.

Constants live for the entire lifetime of a program. More specifically, constants in Rust have no fixed address in memory. This is because they're effectively inlined to each place that they're used. References to the same constant are not necessarily guaranteed to refer to the same memory address for this reason.

[104] `variable-bindings.html`

4.26.1 `static`

Rust provides a 'global variable' sort of facility in static items. They're similar to constants, but static items aren't inlined upon use. This means that there is only one instance for each value, and it's at a fixed location in memory.

Here's an example:

```
static N: i32 = 5;
```

Unlike `let`[105] bindings, you must annotate the type of a `static`.

Statics live for the entire lifetime of a program, and therefore any reference stored in a constant has a `'static` lifetime[106]:

```
static NAME: &'static str = "Steve";
```

Mutability

You can introduce mutability with the `mut` keyword:

```
static mut N: i32 = 5;
```

Because this is mutable, one thread could be updating `N` while another is reading it, causing memory unsafety. As such both accessing and mutating a `static mut` is `unsafe`[107], and so must be done in an `unsafe` block:

```
# static mut N: i32 = 5;

unsafe {
    N += 1;

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

Furthermore, any type stored in a `static` must be `Sync`, and may not have a `Drop`[108] implementation.

4.26.2 Initializing

Both `const` and `static` have requirements for giving them a value. They may only be given a value that's a constant expression. In other words, you cannot use the result of a function call or anything similarly complex or at runtime.

[105] `variable-bindings.html`
[106] `lifetimes.html`
[107] `unsafe.html`
[108] `drop.html`

4.26.3 Which construct should I use?

Almost always, if you can choose between the two, choose `const`. It's pretty rare that you actually want a memory location associated with your constant, and using a const allows for optimizations like constant propagation not only in your crate but downstream crates.

4.27 Attributes

Declarations can be annotated with 'attributes' in Rust. They look like this:

```
#[test]
# fn foo() {}
```

or like this:

```
# mod foo {
#![test]
# }
```

The difference between the two is the `!`, which changes what the attribute applies to:

```
#[foo]
struct Foo;

mod bar {
    #![bar]
}
```

The `#[foo]` attribute applies to the next item, which is the `struct` declaration. The `#![bar]` attribute applies to the item enclosing it, which is the `mod` declaration. Otherwise, they're the same. Both change the meaning of the item they're attached to somehow.

For example, consider a function like this:

```
#[test]
fn check() {
    assert_eq!(2, 1 + 1);
}
```

It is marked with `#[test]`. This means it's special: when you run tests[109], this function will execute. When you compile as usual, it won't even be included. This function is now a test function.

Attributes may also have additional data:

[109] `testing.html`

```
#[inline(always)]
fn super_fast_fn() {
# }
```

Or even keys and values:

```
#[cfg(target_os = ”macos”)]
mod macos_only {
# }
```

Rust attributes are used for a number of different things. There is a full list of attributes in the reference[110]. Currently, you are not allowed to create your own attributes, the Rust compiler defines them.

4.28 ‘type‘ aliases

The `type` keyword lets you declare an alias of another type:

```
type Name = String;
```

You can then use this type as if it were a real type:

```
type Name = String;

let x: Name = ”Hello”.to_string();
```

Note, however, that this is an *alias*, not a new type entirely. In other words, because Rust is strongly typed, you’d expect a comparison between two different types to fail:

```
let x: i32 = 5;
let y: i64 = 5;

if x == y {
   // ...
}
```

this gives

```
error: mismatched types:
 expected ‘i32‘,
    found ‘i64‘
(expected i32,
    found i64) [E0308]
     if x == y {
             ^
```

[110] ../reference.html#attributes

But, if we had an alias:

```
type Num = i32;

let x: i32 = 5;
let y: Num = 5;

if x == y {
   // ...
}
```

This compiles without error. Values of a `Num` type are the same as a value of type `i32`, in every way.

You can also use type aliases with generics:

```
use std::result;

enum ConcreteError {
    Foo,
    Bar,
}

type Result<T> = result::Result<T, ConcreteError>;
```

This creates a specialized version of the `Result` type, which always has a `ConcreteError` for the `E` part of `Result<T, E>`. This is commonly used in the standard library to create custom errors for each subsection. For example, io::Result[111].

4.29 Casting between types

Rust, with its focus on safety, provides two different ways of casting different types between each other. The first, `as`, is for safe casts. In contrast, `transmute` allows for arbitrary casting, and is one of the most dangerous features of Rust!

4.29.1 `as`

The `as` keyword does basic casting:

```
let x: i32 = 5;

let y = x as i64;
```

It only allows certain kinds of casting, however:

[111] `../std/io/type.Result.html`

```
let a = [0u8, 0u8, 0u8, 0u8];

let b = a as u32; // four eights makes 32
```

This errors with:

```
error: non-scalar cast: `[u8; 4]` as `u32`
let b = a as u32; // four eights makes 32
        ^~~~~~~~
```

It's a 'non-scalar cast' because we have multiple values here: the four elements of the array. These kinds of casts are very dangerous, because they make assumptions about the way that multiple underlying structures are implemented. For this, we need something more dangerous.

4.29.2 `transmute`

The `transmute` function is provided by a compiler intrinsic[112], and what it does is very simple, but very scary. It tells Rust to treat a value of one type as though it were another type. It does this regardless of the typechecking system, and just completely trusts you.

In our previous example, we know that an array of four `u8`s represents a `u32` properly, and so we want to do the cast. Using `transmute` instead of `as`, Rust lets us:

```
use std::mem;

unsafe {
    let a = [0u8, 0u8, 0u8, 0u8];

    let b = mem::transmute::<[u8; 4], u32>(a);
}
```

We have to wrap the operation in an `unsafe` block for this to compile successfully. Technically, only the `mem::transmute` call itself needs to be in the block, but it's nice in this case to enclose everything related, so you know where to look. In this case, the details about `a` are also important, and so they're in the block. You'll see code in either style, sometimes the context is too far away, and wrapping all of the code in `unsafe` isn't a great idea.

While `transmute` does very little checking, it will at least make sure that the types are the same size. This errors:

```
use std::mem;

unsafe {
    let a = [0u8, 0u8, 0u8, 0u8];

    let b = mem::transmute::<[u8; 4], u64>(a);
}
```

[112] `intrinsics.html`

with:

```
error: transmute called with differently sized types: [u8; 4] (32 bits) to u64
(64 bits)
```

Other than that, you're on your own!

4.30 Associated Types

Associated types are a powerful part of Rust's type system. They're related to the idea of a 'type family', in other words, grouping multiple types together. That description is a bit abstract, so let's dive right into an example. If you want to write a `Graph` trait, you have two types to be generic over: the node type and the edge type. So you might write a trait, `Graph<N, E>`, that looks like this:

```
trait Graph<N, E> {
    fn has_edge(&self, &N, &N) -> bool;
    fn edges(&self, &N) -> Vec<E>;
    // etc
}
```

While this sort of works, it ends up being awkward. For example, any function that wants to take a `Graph` as a parameter now *also* needs to be generic over the `N`ode and `E`dge types too:

```
fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N) -> u32 { ... }
```

Our distance calculation works regardless of our `Edge` type, so the `E` stuff in this signature is just a distraction.

What we really want to say is that a certain `E`dge and `N`ode type come together to form each kind of `Graph`. We can do that with associated types:

```
trait Graph {
    type N;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
    // etc
}
```

Now, our clients can be abstract over a given `Graph`:

```
fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> u32 { ... }
```

No need to deal with the `E`dge type here!

Let's go over all this in more detail.

Defining associated types

Let's build that `Graph` trait. Here's the definition:

```
trait Graph {
    type N;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

Simple enough. Associated types use the `type` keyword, and go inside the body of the trait, with the functions.

These `type` declarations can have all the same thing as functions do. For example, if we wanted our `N` type to implement `Display`, so we can print the nodes out, we could do this:

```
use std::fmt;

trait Graph {
    type N: fmt::Display;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

Implementing associated types

Just like any trait, traits that use associated types use the `impl` keyword to provide implementations. Here's a simple implementation of Graph:

```
# trait Graph {
#     type N;
#     type E;
#     fn has_edge(&self, &Self::N, &Self::N) -> bool;
#     fn edges(&self, &Self::N) -> Vec<Self::E>;
# }
struct Node;

struct Edge;

struct MyGraph;

impl Graph for MyGraph {
    type N = Node;
    type E = Edge;
```

```
    fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
        true
    }

    fn edges(&self, n: &Node) -> Vec<Edge> {
        Vec::new()
    }
}
```

This silly implementation always returns `true` and an empty `Vec<Edge>`, but it gives you an idea of how to implement this kind of thing. We first need three `struct`s, one for the graph, one for the node, and one for the edge. If it made more sense to use a different type, that would work as well, we're just going to use `struct`s for all three here.

Next is the `impl` line, which is just like implementing any other trait.

From here, we use `=` to define our associated types. The name the trait uses goes on the left of the `=`, and the concrete type we're `impl`ementing this for goes on the right. Finally, we use the concrete types in our function declarations.

Trait objects with associated types

There's one more bit of syntax we should talk about: trait objects. If you try to create a trait object from an associated type, like this:

```
# trait Graph {
#     type N;
#     type E;
#     fn has_edge(&self, &Self::N, &Self::N) -> bool;
#     fn edges(&self, &Self::N) -> Vec<Self::E>;
# }
# struct Node;
# struct Edge;
# struct MyGraph;
# impl Graph for MyGraph {
#     type N = Node;
#     type E = Edge;
#     fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
#         true
#     }
#     fn edges(&self, n: &Node) -> Vec<Edge> {
#         Vec::new()
#     }
# }
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph>;
```

You'll get two errors:

```
error: the value of the associated type ‘E‘ (from the trait ‘main::Graph‘) must
be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
24:44 error: the value of the associated type ‘N‘ (from the trait
‘main::Graph‘) must be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

We can't create a trait object like this, because we don't know the associated types. Instead, we can write this:

```
# trait Graph {
#     type N;
#     type E;
#     fn has_edge(&self, &Self::N, &Self::N) -> bool;
#     fn edges(&self, &Self::N) -> Vec<Self::E>;
# }
# struct Node;
# struct Edge;
# struct MyGraph;
# impl Graph for MyGraph {
#     type N = Node;
#     type E = Edge;
#     fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
#         true
#     }
#     fn edges(&self, n: &Node) -> Vec<Edge> {
#         Vec::new()
#     }
# }
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph<N=Node, E=Edge>>;
```

The `N=Node` syntax allows us to provide a concrete type, `Node`, for the `N` type parameter. Same with `E=Edge`. If we didn't provide this constraint, we couldn't be sure which `impl` to match this trait object to.

4.31 Unsized Types

Most types have a particular size, in bytes, that is knowable at compile time. For example, an `i32` is thirty-two bits big, or four bytes. However, there are some types which are useful to express, but do not have a defined size. These are called 'unsized' or 'dynamically sized' types. One example is `[T]`. This type represents a certain number of `T` in sequence. But we don't know how many there are, so the size is not known.

Rust understands a few of these types, but they have some restrictions. There are three:

1. We can only manipulate an instance of an unsized type via a pointer. An `&[T]` works just fine, but a `[T]` does not.
2. Variables and arguments cannot have dynamically sized types.
3. Only the last field in a `struct` may have a dynamically sized type; the other fields must not. Enum variants must not have dynamically sized types as data.

So why bother? Well, because `[T]` can only be used behind a pointer, if we didn't have language support for unsized types, it would be impossible to write this:

```
impl Foo for str {
```

or

```
impl<T> Foo for [T] {
```

Instead, you would have to write:

```
impl Foo for &str {
```

Meaning, this implementation would only work for references[113], and not other types of pointers. With the `impl for str`, all pointers, including (at some point, there are some bugs to fix first) user-defined custom smart pointers, can use this `impl`.

4.31.1 ?Sized

If you want to write a function that accepts a dynamically sized type, you can use the special bound, `?Sized`:

```
struct Foo<T: ?Sized> {
    f: T,
}
```

This `?`, read as "T may be `Sized`", means that this bound is special: it lets us match more kinds, not less. It's almost like every `T` implicitly has `T: Sized`, and the `?` undoes this default.

4.32 Operators and Overloading

Rust allows for a limited form of operator overloading. There are certain operators that are able to be overloaded. To support a particular operator between types, there's a specific trait that you can implement, which then overloads the operator.

For example, the `+` operator can be overloaded with the `Add` trait:

[113] `references-and-borrowing.html`

```
use std::ops::Add;

#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point { x: self.x + other.x, y: self.y + other.y }
    }
}

fn main() {
    let p1 = Point { x: 1, y: 0 };
    let p2 = Point { x: 2, y: 3 };

    let p3 = p1 + p2;

    println!("{:?}", p3);
}
```

In `main`, we can use `+` on our two `Point`s, since we've implemented `Add<Output=Point>` for `Point`.

There are a number of operators that can be overloaded this way, and all of their associated traits live in the `std::ops`[114] module. Check out its documentation for the full list.

Implementing these traits follows a pattern. Let's look at `Add`[115] in more detail:

```
# mod foo {
pub trait Add<RHS = Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
# }
```

There's three types in total involved here: the type you `impl Add` for, `RHS`, which defaults to `Self`, and `Output`. For an expression `let z = x + y`, `x` is the `Self` type, `y` is the RHS, and `z` is the `Self::Output` type.

```
# struct Point;
# use std::ops::Add;
```

[114] `../std/ops/index.html`
[115] `../std/ops/trait.Add.html`

```
impl Add<i32> for Point {
    type Output = f64;

    fn add(self, rhs: i32) -> f64 {
        // add an i32 to a Point and get an f64
# 1.0
    }
}
```

will let you do this:

```
let p: Point = // ...
let x: f64 = p + 2i32;
```

4.32.1 Using operator traits in generic structs

Now that we know how operator traits are defined, we can define our `HasArea` trait and `Square` struct from the traits chapter[116] more generically:

```
use std::ops::Mul;

trait HasArea<T> {
    fn area(&self) -> T;
}

struct Square<T> {
    x: T,
    y: T,
    side: T,
}

impl<T> HasArea<T> for Square<T>
        where T: Mul<Output=T> + Copy {
    fn area(&self) -> T {
        self.side * self.side
    }
}

fn main() {
    let s = Square {
        x: 0.0f64,
        y: 0.0f64,
        side: 12.0f64,
    };

    println!("Area of s: {}", s.area());
}
```

[116] `traits.html`

For `HasArea` and `Square`, we just declare a type parameter `T` and replace `f64` with it. The `impl` needs more involved modifications:

```
impl<T> HasArea<T> for Square<T>
        where T: Mul<Output=T> + Copy { ... }
```

The `area` method requires that we can multiply the sides, so we declare that type `T` must implement `std::ops::Mul`. Like `Add`, mentioned above, `Mul` itself takes an `Output` parameter: since we know that numbers don't change type when multiplied, we also set it to `T`. `T` must also support copying, so Rust doesn't try to move `self.side` into the return value.

4.33 Deref coercions

The standard library provides a special trait, `Deref`[117]. It's normally used to overload `*`, the dereference operator:

```
use std::ops::Deref;

struct DerefExample<T> {
    value: T,
}

impl<T> Deref for DerefExample<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.value
    }
}

fn main() {
    let x = DerefExample { value: 'a' };
    assert_eq!('a', *x);
}
```

This is useful for writing custom pointer types. However, there's a language feature related to `Deref`: 'deref coercions'. Here's the rule: If you have a type `U`, and it implements `Deref<Target=T>`, values of `&U` will automatically coerce to a `&T`. Here's an example:

```
fn foo(s: &str) {
    // borrow a string for a second
}

// String implements Deref<Target=str>
```

[117] `../std/ops/trait.Deref.html`

```
let owned = "Hello".to_string();

// therefore, this works:
foo(&owned);
```

Using an ampersand in front of a value takes a reference to it. So `owned` is a `String`, `&owned` is an `&String`, and since `impl Deref<Target=str> for String`, `&String` will deref to `&str`, which `foo()` takes.

That's it. This rule is one of the only places in which Rust does an automatic conversion for you, but it adds a lot of flexibility. For example, the `Rc<T>` type implements `Deref<Target=T>`, so this works:

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

fn foo(s: &str) {
    // borrow a string for a second
}

// String implements Deref<Target=str>
let owned = "Hello".to_string();
let counted = Rc::new(owned);

// therefore, this works:
foo(&counted);
```

All we've done is wrap our `String` in an `Rc<T>`. But we can now pass the `Rc<String>` around anywhere we'd have a `String`. The signature of `foo` didn't change, but works just as well with either type. This example has two conversions: `Rc<String>` to `String` and then `String` to `&str`. Rust will do this as many times as possible until the types match.

Another very common implementation provided by the standard library is:

```
fn foo(s: &[i32]) {
    // borrow a slice for a second
}

// Vec<T> implements Deref<Target=[T]>
let owned = vec![1, 2, 3];

foo(&owned);
```

Vectors can `Deref` to a slice.

Deref and method calls

`Deref` will also kick in when calling a method. Consider the following example.

```
struct Foo;

impl Foo {
    fn foo(&self) { println!("Foo"); }
}

let f = &&Foo;

f.foo();
```

Even though `f` is a `&&Foo` and `foo` takes `&self`, this works. That's because these things are the same:

```
f.foo();
(&f).foo();
(&&f).foo();
(&&&&&&&&f).foo();
```

A value of type `&&&&&&&&&&&&&&&&Foo` can still have methods defined on `Foo` called, because the compiler will insert as many * operations as necessary to get it right. And since it's inserting `*`s, that uses `Deref`.

4.34 Macros

By now you've learned about many of the tools Rust provides for abstracting and reusing code. These units of code reuse have a rich semantic structure. For example, functions have a type signature, type parameters have trait bounds, and overloaded functions must belong to a particular trait.

This structure means that Rust's core abstractions have powerful compile-time correctness checking. But this comes at the price of reduced flexibility. If you visually identify a pattern of repeated code, you may find it's difficult or cumbersome to express that pattern as a generic function, a trait, or anything else within Rust's semantics.

Macros allow us to abstract at a syntactic level. A macro invocation is shorthand for an "expanded" syntactic form. This expansion happens early in compilation, before any static checking. As a result, macros can capture many patterns of code reuse that Rust's core abstractions cannot.

The drawback is that macro-based code can be harder to understand, because fewer of the built-in rules apply. Like an ordinary function, a well-behaved macro can be used without understanding its implementation. However, it can be difficult to design a well-behaved macro! Additionally, compiler errors in macro code are harder to interpret, because they describe problems in the expanded code, not the source-level form that developers use.

These drawbacks make macros something of a "feature of last resort". That's not to say that macros are bad; they are part of Rust because sometimes they're needed for truly concise, well-abstracted code. Just keep this tradeoff in mind.

4.34.1 Defining a macro

You may have seen the `vec!` macro, used to initialize a vector[118] with any number of elements.

```
let x: Vec<u32> = vec![1, 2, 3];
# assert_eq!(x, [1, 2, 3]);
```

This can't be an ordinary function, because it takes any number of arguments. But we can imagine it as syntactic shorthand for

```
let x: Vec<u32> = {
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
};
# assert_eq!(x, [1, 2, 3]);
```

We can implement this shorthand, using a macro: [119]

```
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
# fn main() {
#     assert_eq!(vec![1,2,3], [1, 2, 3]);
# }
```

Whoa, that's a lot of new syntax! Let's break it down.

```
macro_rules! vec { ... }
```

This says we're defining a macro named `vec`, much as `fn vec` would define a function named `vec`. In prose, we informally write a macro's name with an exclamation point, e.g. `vec!`. The exclamation point is part of the invocation syntax and serves to distinguish a macro from an ordinary function.

[118] `vectors.html`

[119] The actual definition of `vec!` in libcollections differs from the one presented here, for reasons of efficiency and reusability.

Matching

The macro is defined through a series of rules, which are pattern-matching cases. Above, we had

```
( $( $x:expr ),* ) => { ... };
```

This is like a `match` expression arm, but the matching happens on Rust syntax trees, at compile time. The semicolon is optional on the last (here, only) case. The “pattern” on the left-hand side of `=>` is known as a ‘matcher’. These have their own little grammar[120] within the language.

The matcher `$x:expr` will match any Rust expression, binding that syntax tree to the ‘metavariable’ `$x`. The identifier `expr` is a ‘fragment specifier’; the full possibilities are enumerated later in this chapter. Surrounding the matcher with `$(...),*` will match zero or more expressions, separated by commas.

Aside from the special matcher syntax, any Rust tokens that appear in a matcher must match exactly. For example,

```
macro_rules! foo {
    (x => $e:expr) => (println!(”mode X: {}”, $e));
    (y => $e:expr) => (println!(”mode Y: {}”, $e));
}

fn main() {
    foo!(y => 3);
}
```

will print

```
mode Y: 3
```

With

```
foo!(z => 3);
```

we get the compiler error

```
error: no rules expected the token ‘z‘
```

Expansion

The right-hand side of a macro rule is ordinary Rust syntax, for the most part. But we can splice in bits of syntax captured by the matcher. From the original example:

```
$(
    temp_vec.push($x);
)*
```

[120] `../reference.html#macros`

Each matched expression `$x` will produce a single `push` statement in the macro expansion. The repetition in the expansion proceeds in "lockstep" with repetition in the matcher (more on this in a moment).

Because `$x` was already declared as matching an expression, we don't repeat `:expr` on the right-hand side. Also, we don't include a separating comma as part of the repetition operator. Instead, we have a terminating semicolon within the repeated block.

Another detail: the `vec!` macro has *two* pairs of braces on the right-hand side. They are often combined like so:

```
macro_rules! foo {
    () => {{
        ...
    }}
}
```

The outer braces are part of the syntax of `macro_rules!`. In fact, you can use `()` or `[]` instead. They simply delimit the right-hand side as a whole.

The inner braces are part of the expanded syntax. Remember, the `vec!` macro is used in an expression context. To write an expression with multiple statements, including `let`-bindings, we use a block. If your macro expands to a single expression, you don't need this extra layer of braces.

Note that we never *declared* that the macro produces an expression. In fact, this is not determined until we use the macro as an expression. With care, you can write a macro whose expansion works in several contexts. For example, shorthand for a data type could be valid as either an expression or a pattern.

Repetition

The repetition operator follows two principal rules:

1. `$(...)*` walks through one "layer" of repetitions, for all of the `$name`s it contains, in lockstep, and
2. each `$name` must be under at least as many `$(...)*`s as it was matched against. If it is under more, it'll be duplicated, as appropriate.

This baroque macro illustrates the duplication of variables from outer repetition levels.

```
macro_rules! o_O {
    (
        $(
            $x:expr; [ $( $y:expr ),* ]
        );*
    ) => {
        &[ $($( $x + $y ),*),* ]
    }
```

```
}

fn main() {
    let a: &[i32]
        = o_O!(10; [1, 2, 3];
               20; [4, 5, 6]);

    assert_eq!(a, [11, 12, 13, 24, 25, 26]);
}
```

That's most of the matcher syntax. These examples use `$(...)*`, which is a "zero or more" match. Alternatively you can write `$(...)+` for a "one or more" match. Both forms optionally include a separator, which can be any token except `+` or `*`.

This system is based on "Macro-by-Example[121]" (PDF link).

4.34.2 Hygiene

Some languages implement macros using simple text substitution, which leads to various problems. For example, this C program prints `13` instead of the expected `25`.

```
#define FIVE_TIMES(x) 5 * x

int main() {
    printf("%d\n", FIVE_TIMES(2 + 3));
    return 0;
}
```

After expansion we have `5 * 2 + 3`, and multiplication has greater precedence than addition. If you've used C macros a lot, you probably know the standard idioms for avoiding this problem, as well as five or six others. In Rust, we don't have to worry about it.

```
macro_rules! five_times {
    ($x:expr) => (5 * $x);
}

fn main() {
    assert_eq!(25, five_times!(2 + 3));
}
```

The metavariable `$x` is parsed as a single expression node, and keeps its place in the syntax tree even after substitution.

Another common problem in macro systems is 'variable capture'. Here's a C macro, using a GNU C extension[122] to emulate Rust's expression blocks.

[121] https://www.cs.indiana.edu/ftp/techreports/TR206.pdf
[122] https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html

```
#define LOG(msg) ({ \
    int state = get_log_state(); \
    if (state > 0) { \
        printf(”log(%d): %s\n”, state, msg); \
    } \
})
```

Here’s a simple use case that goes terribly wrong:

```
const char *state = ”reticulating splines”;
LOG(state)
```

This expands to

```
const char *state = ”reticulating splines”;
int state = get_log_state();
if (state > 0) {
    printf(”log(%d): %s\n”, state, state);
}
```

The second variable named `state` shadows the first one. This is a problem because the print statement should refer to both of them.

The equivalent Rust macro has the desired behavior.

```
# fn get_log_state() -> i32 { 3 }
macro_rules! log {
    ($msg:expr) => {{
        let state: i32 = get_log_state();
        if state > 0 {
            println!(”log({}): {}”, state, $msg);
        }
    }};
}

fn main() {
    let state: &str = ”reticulating splines”;
    log!(state);
}
```

This works because Rust has a hygienic macro system[123]. Each macro expansion happens in a distinct ‘syntax context’, and each variable is tagged with the syntax context where it was introduced. It’s as though the variable `state` inside `main` is painted a different “color” from the variable `state` inside the macro, and therefore they don’t conflict.

This also restricts the ability of macros to introduce new bindings at the invocation site. Code such as the following will not work:

[123] https://en.wikipedia.org/wiki/Hygienic_macro

```
macro_rules! foo {
    () => (let x = 3);
}

fn main() {
    foo!();
    println!(”{}”, x);
}
```

Instead you need to pass the variable name into the invocation, so it’s tagged with the right syntax context.

```
macro_rules! foo {
    ($v:ident) => (let $v = 3);
}

fn main() {
    foo!(x);
    println!(”{}”, x);
}
```

This holds for `let` bindings and loop labels, but not for items[124]. So the following code does compile:

```
macro_rules! foo {
    () => (fn x() { });
}

fn main() {
    foo!();
    x();
}
```

4.34.3 Recursive macros

A macro’s expansion can include more macro invocations, including invocations of the very same macro being expanded. These recursive macros are useful for processing tree-structured input, as illustrated by this (simplistic) HTML shorthand:

```
# #![allow(unused_must_use)]
macro_rules! write_html {
    ($w:expr, ) => (());

    ($w:expr, $e:tt) => (write!($w, ”{}”, $e));

    ($w:expr, $tag:ident [ $($inner:tt)* ] $($rest:tt)*) => {{
        write!($w, ”<{}>”, stringify!($tag));
```

[124] ../reference.html#items

```
        write_html!($w, $($inner)*);
        write!($w, "</{}>", stringify!($tag));
        write_html!($w, $($rest)*);
    }};
}

fn main() {
#   // FIXME(#21826)
    use std::fmt::Write;
    let mut out = String::new();

    write_html!(&mut out,
        html[
            head[title["Macros guide"]]
            body[h1["Macros are the best!"]]
        ]);

    assert_eq!(out,
        "<html><head><title>Macros guide</title></head>\
         <body><h1>Macros are the best!</h1></body></html>");
}
```

4.34.4 Debugging macro code

To see the results of expanding macros, run `rustc --pretty expanded`. The output represents a whole crate, so you can also feed it back in to `rustc`, which will sometimes produce better error messages than the original compilation. Note that the `--pretty expanded` output may have a different meaning if multiple variables of the same name (but different syntax contexts) are in play in the same scope. In this case `--pretty expanded,hygiene` will tell you about the syntax contexts.

`rustc` provides two syntax extensions that help with macro debugging. For now, they are unstable and require feature gates.

- `log_syntax!(...)` will print its arguments to standard output, at compile time, and “expand” to nothing.
- `trace_macros!(true)` will enable a compiler message every time a macro is expanded. Use `trace_macros!(false)` later in expansion to turn it off.

4.34.5 Syntactic requirements

Even when Rust code contains un-expanded macros, it can be parsed as a full syntax tree[125]. This property can be very useful for editors and other tools that process code. It also has a few consequences for the design of Rust’s macro system.

One consequence is that Rust must determine, when it parses a macro invocation, whether the macro stands in for

[125] `glossary.html#abstract-syntax-tree`

- zero or more items,
- zero or more methods,
- an expression,
- a statement, or
- a pattern.

A macro invocation within a block could stand for some items, or for an expression / statement. Rust uses a simple rule to resolve this ambiguity. A macro invocation that stands for items must be either

- delimited by curly braces, e.g. `foo! { ... }`, or
- terminated by a semicolon, e.g. `foo!(...);`

Another consequence of pre-expansion parsing is that the macro invocation must consist of valid Rust tokens. Furthermore, parentheses, brackets, and braces must be balanced within a macro invocation. For example, `foo!([)` is forbidden. This allows Rust to know where the macro invocation ends.

More formally, the macro invocation body must be a sequence of 'token trees'. A token tree is defined recursively as either

- a sequence of token trees surrounded by matching `()`, `[]`, or `{}`, or
- any other single token.

Within a matcher, each metavariable has a 'fragment specifier', identifying which syntactic form it matches.

- `ident`: an identifier. Examples: `x`; `foo`.
- `path`: a qualified name. Example: `T::SpecialA`.
- `expr`: an expression. Examples: `2 + 2`; `if true { 1 } else { 2 }`; `f(42)`.
- `ty`: a type. Examples: `i32`; `Vec<(char, String)>`; `&T`.
- `pat`: a pattern. Examples: `Some(t)`; `(17, 'a')`; `_`.
- `stmt`: a single statement. Example: `let x = 3`.
- `block`: a brace-delimited sequence of statements. Example: `{ log(error, "hi"); return 12; }`.
- `item`: an item[126]. Examples: `fn foo() { }`; `struct Bar;`.
- `meta`: a "meta item", as found in attributes. Example: `cfg(target_os = "windows")`.
- `tt`: a single token tree.

There are additional rules regarding the next token after a metavariable:

- `expr` variables may only be followed by one of: `=> , ;`
- `ty` and `path` variables may only be followed by one of: `=> , : = > as`
- `pat` variables may only be followed by one of: `=> , = if in`
- Other variables may be followed by any token.

These rules provide some flexibility for Rust's syntax to evolve without breaking existing macros.

[126] `../reference.html#items`

The macro system does not deal with parse ambiguity at all. For example, the grammar `$($t:ty)* $e:expr` will always fail to parse, because the parser would be forced to choose between parsing `$t` and parsing `$e`. Changing the invocation syntax to put a distinctive token in front can solve the problem. In this case, you can write `$(T $t:ty)* E $e:exp`.

4.34.6 Scoping and macro import/export

Macros are expanded at an early stage in compilation, before name resolution. One downside is that scoping works differently for macros, compared to other constructs in the language.

Definition and expansion of macros both happen in a single depth-first, lexical-order traversal of a crate's source. So a macro defined at module scope is visible to any subsequent code in the same module, which includes the body of any subsequent child `mod` items.

A macro defined within the body of a single `fn`, or anywhere else not at module scope, is visible only within that item.

If a module has the `macro_use` attribute, its macros are also visible in its parent module after the child's `mod` item. If the parent also has `macro_use` then the macros will be visible in the grandparent after the parent's `mod` item, and so forth.

The `macro_use` attribute can also appear on `extern crate`. In this context it controls which macros are loaded from the external crate, e.g.

```
#[macro_use(foo, bar)]
extern crate baz;
```

If the attribute is given simply as `#[macro_use]`, all macros are loaded. If there is no `#[macro_use]` attribute then no macros are loaded. Only macros defined with the `#[macro_export]` attribute may be loaded.

To load a crate's macros without linking it into the output, use `#[no_link]` as well.

An example:

```
macro_rules! m1 { () => (()) }

// visible here: m1

mod foo {
    // visible here: m1

    #[macro_export]
    macro_rules! m2 { () => (()) }

    // visible here: m1, m2
}

// visible here: m1
```

```
macro_rules! m3 { () => (()) }

// visible here: m1, m3

#[macro_use]
mod bar {
    // visible here: m1, m3

    macro_rules! m4 { () => (()) }

    // visible here: m1, m3, m4
}

// visible here: m1, m3, m4
# fn main() { }
```

When this library is loaded with `#[macro_use] extern crate`, only `m2` will be imported.

The Rust Reference has a listing of macro-related attributes[127].

4.34.7 The variable `$crate`

A further difficulty occurs when a macro is used in multiple crates. Say that `mylib` defines

```
pub fn increment(x: u32) -> u32 {
    x + 1
}

#[macro_export]
macro_rules! inc_a {
    ($x:expr) => ( ::increment($x) )
}

#[macro_export]
macro_rules! inc_b {
    ($x:expr) => ( ::mylib::increment($x) )
}
# fn main() { }
```

`inc_a` only works within `mylib`, while `inc_b` only works outside the library. Furthermore, `inc_b` will break if the user imports `mylib` under another name.

Rust does not (yet) have a hygiene system for crate references, but it does provide a simple workaround for this problem. Within a macro imported from a crate named `foo`, the special macro variable `$crate` will expand to `::foo`. By contrast, when a macro is defined and then used in the same crate, `$crate` will expand to nothing. This means we can write

[127] `../reference.html#macro-related-attributes`

```
#[macro_export]
macro_rules! inc {
    ($x:expr) => ( $crate::increment($x) )
}
# fn main() { }
```

to define a single macro that works both inside and outside our library. The function name will expand to either `::increment` or `::mylib::increment`.

To keep this system simple and correct, `#[macro_use] extern crate ...` may only appear at the root of your crate, not inside `mod`. This ensures that `$crate` is a single identifier.

4.34.8 The deep end

The introductory chapter mentioned recursive macros, but it did not give the full story. Recursive macros are useful for another reason: Each recursive invocation gives you another opportunity to pattern-match the macro's arguments.

As an extreme example, it is possible, though hardly advisable, to implement the Bitwise Cyclic Tag[128] automaton within Rust's macro system.

```
macro_rules! bct {
    // cmd 0:  d ... => ...
    (0, $($ps:tt),* ; $_d:tt)
        => (bct!($($ps),*, 0 ; ));
    (0, $($ps:tt),* ; $_d:tt, $($ds:tt),*)
        => (bct!($($ps),*, 0 ; $($ds),*));

    // cmd 1p:  1 ... => 1 ... p
    (1, $p:tt, $($ps:tt),* ; 1)
        => (bct!($($ps),*, 1, $p ; 1, $p));
    (1, $p:tt, $($ps:tt),* ; 1, $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; 1, $($ds),*, $p));

    // cmd 1p:  0 ... => 0 ...
    (1, $p:tt, $($ps:tt),* ; $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; $($ds),*));

    // halt on empty data string
    ( $($ps:tt),* ; )
        => (());
}
```

Exercise: use macros to reduce duplication in the above definition of the `bct!` macro.

4.34.9 Common macros

Here are some common macros you'll see in Rust code.

[128] https://esolangs.org/wiki/Bitwise_Cyclic_Tag

panic!

This macro causes the current thread to panic. You can give it a message to panic with:

```
panic!("oh no!");
```

vec!

The `vec!` macro is used throughout the book, so you've probably seen it already. It creates `Vec<T>`s with ease:

```
let v = vec![1, 2, 3, 4, 5];
```

It also lets you make vectors with repeating values. For example, a hundred zeroes:

```
let v = vec![0; 100];
```

assert! and assert_eq!

These two macros are used in tests. `assert!` takes a boolean. `assert_eq!` takes two values and checks them for equality. `true` passes, `false` `panic!`s. Like this:

```
// A-ok!

assert!(true);
assert_eq!(5, 3 + 2);

// nope :(

assert!(5 < 3);
assert_eq!(5, 3);
```

try!

`try!` is used for error handling. It takes something that can return a `Result<T, E>`, and gives `T` if it's a `Ok<T>`, and `return`s with the `Err(E)` if it's that. Like this:

```
use std::fs::File;

fn foo() -> std::io::Result<()> {
    let f = try!(File::create("foo.txt"));

    Ok(())
}
```

This is cleaner than doing this:

```
use std::fs::File;

fn foo() -> std::io::Result<()> {
    let f = File::create("foo.txt");

    let f = match f {
        Ok(t) => t,
        Err(e) => return Err(e),
    };

    Ok(())
}
```

unreachable!

This macro is used when you think some code should never execute:

```
if false {
    unreachable!();
}
```

Sometimes, the compiler may make you have a different branch that you know will never, ever run. In these cases, use this macro, so that if you end up wrong, you'll get a `panic!` about it.

```
let x: Option<i32> = None;

match x {
    Some(_) => unreachable!(),
    None => println!("I know x is None!"),
}
```

unimplemented!

The `unimplemented!` macro can be used when you're trying to get your functions to typecheck, and don't want to worry about writing out the body of the function. One example of this situation is implementing a trait with multiple required methods, where you want to tackle one at a time. Define the others as `unimplemented!` until you're ready to write them.

4.34.10 Procedural macros

If Rust's macro system can't do what you need, you may want to write a compiler plugin[129] instead. Compared to `macro_rules!` macros, this is significantly more work, the interfaces are much less stable, and bugs can be much harder to track down. In exchange you get the flexibility of running arbitrary Rust code within the compiler. Syntax extension plugins are sometimes called 'procedural macros' for this reason.

[129] `compiler-plugins.html`

4.35 Raw Pointers

Rust has a number of different smart pointer types in its standard library, but there are two types that are extra-special. Much of Rust's safety comes from compile-time checks, but raw pointers don't have such guarantees, and are unsafe[130] to use.

`*const T` and `*mut T` are called 'raw pointers' in Rust. Sometimes, when writing certain kinds of libraries, you'll need to get around Rust's safety guarantees for some reason. In this case, you can use raw pointers to implement your library, while exposing a safe interface for your users. For example, `*` pointers are allowed to alias, allowing them to be used to write shared-ownership types, and even thread-safe shared memory types (the `Rc<T>` and `Arc<T>` types are both implemented entirely in Rust).

Here are some things to remember about raw pointers that are different than other pointer types. They:

- are not guaranteed to point to valid memory and are not even guaranteed to be non-null (unlike both `Box` and `&`);
- do not have any automatic clean-up, unlike `Box`, and so require manual resource management;
- are plain-old-data, that is, they don't move ownership, again unlike `Box`, hence the Rust compiler cannot protect against bugs like use-after-free;
- lack any form of lifetimes, unlike `&`, and so the compiler cannot reason about dangling pointers; and
- have no guarantees about aliasing or mutability other than mutation not being allowed directly through a `*const T`.

4.35.1 Basics

Creating a raw pointer is perfectly safe:

```
let x = 5;
let raw = &x as *const i32;

let mut y = 10;
let raw_mut = &mut y as *mut i32;
```

However, dereferencing one is not. This won't work:

```
let x = 5;
let raw = &x as *const i32;

println!("raw points at {}", *raw);
```

It gives this error:

[130] `unsafe.html`

```
error: dereference of raw pointer requires unsafe function or block [E0133]
     println!("raw points at {}", *raw);
                                  ^~~~
```

When you dereference a raw pointer, you're taking responsibility that it's not pointing somewhere that would be incorrect. As such, you need `unsafe`:

```
let x = 5;
let raw = &x as *const i32;

let points_at = unsafe { *raw };

println!("raw points at {}", points_at);
```

For more operations on raw pointers, see their API documentation[131].

4.35.2 FFI

Raw pointers are useful for FFI: Rust's `*const T` and `*mut T` are similar to C's `const T*` and `T*`, respectively. For more about this use, consult the FFI chapter[132].

4.35.3 References and raw pointers

At runtime, a raw pointer `*` and a reference pointing to the same piece of data have an identical representation. In fact, an `&T` reference will implicitly coerce to an `*const T` raw pointer in safe code and similarly for the `mut` variants (both coercions can be performed explicitly with, respectively, `value as *const T` and `value as *mut T`).
Going the opposite direction, from `*const` to a reference `&`, is not safe. A `&T` is always valid, and so, at a minimum, the raw pointer `*const T` has to point to a valid instance of type `T`. Furthermore, the resulting pointer must satisfy the aliasing and mutability laws of references. The compiler assumes these properties are true for any references, no matter how they are created, and so any conversion from raw pointers is asserting that they hold. The programmer *must* guarantee this.
The recommended method for the conversion is:

```
// explicit cast
let i: u32 = 1;
let p_imm: *const u32 = &i as *const u32;

// implicit coercion
let mut m: u32 = 2;
let p_mut: *mut u32 = &mut m;

unsafe {
    let ref_imm: &u32 = &*p_imm;
    let ref_mut: &mut u32 = &mut *p_mut;
}
```

[131] `../std/primitive.pointer.html`
[132] `ffi.html`

The `&*x` dereferencing style is preferred to using a `transmute`. The latter is far more powerful than necessary, and the more restricted operation is harder to use incorrectly; for example, it requires that `x` is a pointer (unlike `transmute`).

4.36 'unsafe'

Rust's main draw is its powerful static guarantees about behavior. But safety checks are conservative by nature: there are some programs that are actually safe, but the compiler is not able to verify this is true. To write these kinds of programs, we need to tell the compiler to relax its restrictions a bit. For this, Rust has a keyword, `unsafe`. Code using `unsafe` has less restrictions than normal code does.

Let's go over the syntax, and then we'll talk semantics. `unsafe` is used in four contexts. The first one is to mark a function as unsafe:

```
unsafe fn danger_will_robinson() {
    // scary stuff
}
```

All functions called from FFI[133] must be marked as `unsafe`, for example. The second use of `unsafe` is an unsafe block:

```
unsafe {
    // scary stuff
}
```

The third is for unsafe traits:

```
unsafe trait Scary { }
```

And the fourth is for `impl`ementing one of those traits:

```
# unsafe trait Scary { }
unsafe impl Scary for i32 {}
```

It's important to be able to explicitly delineate code that may have bugs that cause big problems. If a Rust program segfaults, you can be sure it's somewhere in the sections marked `unsafe`.

4.36.1 What does 'safe' mean?

Safe, in the context of Rust, means 'doesn't do anything unsafe'. It's also important to know that there are certain behaviors that are probably not desirable in your code, but are expressly *not* unsafe:

- Deadlocks

[133] `ffi.html`

- Leaks of memory or other resources
- Exiting without calling destructors
- Integer overflow

Rust cannot prevent all kinds of software problems. Buggy code can and will be written in Rust. These things aren't great, but they don't qualify as `unsafe` specifically.

In addition, the following are all undefined behaviors in Rust, and must be avoided, even when writing `unsafe` code:

- Data races
- Dereferencing a null/dangling raw pointer
- Reads of undef[134] (uninitialized) memory
- Breaking the pointer aliasing rules[135] with raw pointers.
- `&mut T` and `&T` follow LLVM's scoped noalias[136] model, except if the `&T` contains an `UnsafeCell<U>`. Unsafe code must not violate these aliasing guarantees.
- Mutating an immutable value/reference without `UnsafeCell<U>`
- Invoking undefined behavior via compiler intrinsics:
- Indexing outside of the bounds of an object with `std::ptr::offset` (`offset` intrinsic), with the exception of one byte past the end which is permitted.
- Using `std::ptr::copy_nonoverlapping_memory` (`memcpy32`/`memcpy64` intrinsics) on overlapping buffers
- Invalid values in primitive types, even in private fields/locals:
- Null/dangling references or boxes
- A value other than `false` (0) or `true` (1) in a `bool`
- A discriminant in an `enum` not included in its type definition
- A value in a `char` which is a surrogate or above `char::MAX`
- Non-UTF-8 byte sequences in a `str`
- Unwinding into Rust from foreign code or unwinding from Rust into foreign code.

4.36.2 Unsafe Superpowers

In both unsafe functions and unsafe blocks, Rust will let you do three things that you normally can not do. Just three. Here they are:

1. Access or update a static mutable variable[137].
2. Dereference a raw pointer.
3. Call unsafe functions. This is the most powerful ability.

That's it. It's important that `unsafe` does not, for example, 'turn off the borrow checker'. Adding `unsafe` to some random Rust code doesn't change its semantics, it won't just start accepting anything. But it will let you write things that *do* break some of the rules.

[134] `http://llvm.org/docs/LangRef.html#undefined-values`
[135] `http://llvm.org/docs/LangRef.html#pointer-aliasing-rules`
[136] `http://llvm.org/docs/LangRef.html#noalias`
[137] `const-and-static.html#static`

You will also encounter the `unsafe` keyword when writing bindings to foreign (non-Rust) interfaces. You're encouraged to write a safe, native Rust interface around the methods provided by the library.

Let's go over the basic three abilities listed, in order.

Access or update a `static mut`

Rust has a feature called '`static mut`' which allows for mutable global state. Doing so can cause a data race, and as such is inherently not safe. For more details, see the static[138] section of the book.

Dereference a raw pointer

Raw pointers let you do arbitrary pointer arithmetic, and can cause a number of different memory safety and security issues. In some senses, the ability to dereference an arbitrary pointer is one of the most dangerous things you can do. For more on raw pointers, see their section of the book[139].

Call unsafe functions

This last ability works with both aspects of `unsafe`: you can only call functions marked `unsafe` from inside an unsafe block.

This ability is powerful and varied. Rust exposes some compiler intrinsics[140] as unsafe functions, and some unsafe functions bypass safety checks, trading safety for speed.

I'll repeat again: even though you *can* do arbitrary things in unsafe blocks and functions doesn't mean you should. The compiler will act as though you're upholding its invariants, so be careful!

[138] `const-and-static.html#static`
[139] `raw-pointers.html`
[140] `intrinsics.html`

Chapter 5

Nightly Rust

Rust provides three distribution channels for Rust: nightly, beta, and stable. Unstable features are only available on nightly Rust. For more details on this process, see 'Stability as a deliverable[1]'.

To install nightly Rust, you can use `rustup.sh`:

```
$ curl -s https://static.rust-lang.org/rustup.sh | sh -s -- --channel=nightly
```

If you're concerned about the potential insecurity[2] of using `curl | sh`, please keep reading and see our disclaimer below. And feel free to use a two-step version of the installation and examine our installation script:

```
$ curl -f -L https://static.rust-lang.org/rustup.sh -O
$ sh rustup.sh --channel=nightly
```

If you're on Windows, please download either the 32-bit installer[3] or the 64-bit installer[4] and run it.

5.0.3 Uninstalling

If you decide you don't want Rust anymore, we'll be a bit sad, but that's okay. Not every programming language is great for everyone. Just run the uninstall script:

```
$ sudo /usr/local/lib/rustlib/uninstall.sh
```

If you used the Windows installer, just re-run the `.msi` and it will give you an uninstall option.

Some people, and somewhat rightfully so, get very upset when we tell you to `curl | sh`. Basically, when you do this, you are trusting that the good people who maintain

[1] `http://blog.rust-lang.org/2014/10/30/Stability.html`
[2] `http://curlpipesh.tumblr.com`
[3] `https://static.rust-lang.org/dist/rust-nightly-i686-pc-windows-gnu.msi`
[4] `https://static.rust-lang.org/dist/rust-nightly-x86_64-pc-windows-gnu.msi`

Rust aren't going to hack your computer and do bad things. That's a good instinct! If you're one of those people, please check out the documentation on building Rust from Source[5], or the official binary downloads[6].

Oh, we should also mention the officially supported platforms:

- Windows (7, 8, Server 2008 R2)
- Linux (2.6.18 or later, various distributions), x86 and x86-64
- OSX 10.7 (Lion) or greater, x86 and x86-64

We extensively test Rust on these platforms, and a few others, too, like Android. But these are the ones most likely to work, as they have the most testing.

Finally, a comment about Windows. Rust considers Windows to be a first-class platform upon release, but if we're honest, the Windows experience isn't as integrated as the Linux/OS X experience is. We're working on it! If anything does not work, it is a bug. Please let us know if that happens. Each and every commit is tested against Windows just like any other platform.

If you've got Rust installed, you can open up a shell, and type this:

```
$ rustc --version
```

You should see the version number, commit hash, commit date and build date:

```
rustc 1.0.0-nightly (f11f3e7ba 2015-01-04) (built 2015-01-06)
```

If you did, Rust has been installed successfully! Congrats!

This installer also installs a copy of the documentation locally, so you can read it offline. On UNIX systems, `/usr/local/share/doc/rust` is the location. On Windows, it's in a `share/doc` directory, inside wherever you installed Rust to.

If not, there are a number of places where you can get help. The easiest is the #rust IRC channel on irc.mozilla.org[7], which you can access through Mibbit[8]. Click that link, and you'll be chatting with other Rustaceans (a silly nickname we call ourselves), and we can help you out. Other great resources include the user's forum[9], and Stack Overflow[10].

5.1 Compiler Plugins

5.1.1 Introduction

`rustc` can load compiler plugins, which are user-provided libraries that extend the compiler's behavior with new syntax extensions, lint checks, etc.

[5] https://github.com/rust-lang/rust#building-from-source
[6] https://www.rust-lang.org/install.html
[7] irc://irc.mozilla.org/#rust
[8] http://chat.mibbit.com/?server=irc.mozilla.org&channel=%23rust
[9] https://users.rust-lang.org/
[10] http://stackoverflow.com/questions/tagged/rust

A plugin is a dynamic library crate with a designated *registrar* function that registers extensions with `rustc`. Other crates can load these extensions using the crate attribute `#![plugin(...)]`. See the `rustc::plugin`[11] documentation for more about the mechanics of defining and loading a plugin.

If present, arguments passed as `#![plugin(foo(... args ...))]` are not interpreted by rustc itself. They are provided to the plugin through the `Registry`'s `args` method[12].

In the vast majority of cases, a plugin should *only* be used through `#![plugin]` and not through an `extern crate` item. Linking a plugin would pull in all of libsyntax and librustc as dependencies of your crate. This is generally unwanted unless you are building another plugin. The `plugin_as_library` lint checks these guidelines.

The usual practice is to put compiler plugins in their own crate, separate from any `macro_rules!` macros or ordinary Rust code meant to be used by consumers of a library.

5.1.2 Syntax extensions

Plugins can extend Rust's syntax in various ways. One kind of syntax extension is the procedural macro. These are invoked the same way as ordinary macros[13], but the expansion is performed by arbitrary Rust code that manipulates syntax trees[14] at compile time.

Let's write a plugin `roman_numerals.rs`[15] that implements Roman numeral integer literals.

```
#![crate_type="dylib"]
#![feature(plugin_registrar, rustc_private)]

extern crate syntax;
extern crate rustc;

use syntax::codemap::Span;
use syntax::parse::token;
use syntax::ast::TokenTree;
use syntax::ext::base::{ExtCtxt, MacResult, DummyResult, MacEager};
use syntax::ext::build::AstBuilder;  // trait for expr_usize
use rustc::plugin::Registry;

fn expand_rn(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box<MacResult + 'static> {

    static NUMERALS: &'static [(&'static str, usize)] = &[
        ("M", 1000), ("CM", 900), ("D", 500), ("CD", 400),
        ("C",  100), ("XC",  90), ("L",  50), ("XL",  40),
```

[11] ../rustc/plugin/index.html
[12] ../rustc/plugin/registry/struct.Registry.html#method.args
[13] macros.html
[14] ../syntax/ast/index.html
[15] https://github.com/rust-lang/rust/tree/master/src/test/auxiliary/roman_numerals.rs

```
        (”X”,   10), (”IX”,   9), (”V”,   5), (”IV”,   4),
        (”I”,    1)];

    if args.len() != 1 {
        cx.span_err(
            sp,
            &format!(”argument should be a single identifier, but got {} arguments”, args.len(
        return DummyResult::any(sp);
    }

    let text = match args[0] {
        TokenTree::Token(_, token::Ident(s, _)) => s.to_string(),
        _ => {
            cx.span_err(sp, ”argument should be a single identifier”);
            return DummyResult::any(sp);
        }
    };

    let mut text = &*text;
    let mut total = 0;
    while !text.is_empty() {
        match NUMERALS.iter().find(|&&(rn, _)| text.starts_with(rn)) {
            Some(&(rn, val)) => {
                total += val;
                text = &text[rn.len()..];
            }
            None => {
                cx.span_err(sp, ”invalid Roman numeral”);
                return DummyResult::any(sp);
            }
        }
    }

    MacEager::expr(cx.expr_usize(sp, total))
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_macro(”rn”, expand_rn);
}
```

Then we can use `rn!()` like any other macro:

```
#![feature(plugin)]
#![plugin(roman_numerals)]

fn main() {
    assert_eq!(rn!(MMXV), 2015);
}
```

The advantages over a simple `fn(&str) -> u32` are:

- The (arbitrarily complex) conversion is done at compile time.
- Input validation is also performed at compile time.
- It can be extended to allow use in patterns, which effectively gives a way to define new literal syntax for any data type.

In addition to procedural macros, you can define new `derive`[16]-like attributes and other kinds of extensions. See `Registry::register_syntax_extension`[17] and the `SyntaxExtension` enum[18]. For a more involved macro example, see `regex_macros`[19].

Tips and tricks

Some of the macro debugging tips[20] are applicable.

You can use `syntax::parse`[21] to turn token trees into higher-level syntax elements like expressions:

```
fn expand_foo(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box<MacResult+'static> {

    let mut parser = cx.new_parser_from_tts(args);

    let expr: P<Expr> = parser.parse_expr();
```

Looking through `libsyntax` parser code[22] will give you a feel for how the parsing infrastructure works.

Keep the `Span`s[23] of everything you parse, for better error reporting. You can wrap `Spanned`[24] around your custom data structures.

Calling `ExtCtxt::span_fatal`[25] will immediately abort compilation. It's better to instead call `ExtCtxt::span_err`[26] and return `DummyResult`[27], so that the compiler can continue and find further errors.

To print syntax fragments for debugging, you can use `span_note`[28] together with `syntax::print::pprust::*_to_string`[29].

The example above produced an integer literal using `AstBuilder::expr_usize`[30]. As an alternative to the `AstBuilder` trait, `libsyntax` provides a set of quasiquote

[16] ../reference.html#derive
[17] ../rustc/plugin/registry/struct.Registry.html#method.register_syntax_extension
[18] https://doc.rust-lang.org/syntax/ext/base/enum.SyntaxExtension.html
[19] https://github.com/rust-lang/regex/blob/master/regex_macros/src/lib.rs
[20] macros.html#debugging-macro-code
[21] ../syntax/parse/index.html
[22] https://github.com/rust-lang/rust/blob/master/src/libsyntax/parse/parser.rs
[23] ../syntax/codemap/struct.Span.html
[24] ../syntax/codemap/struct.Spanned.html
[25] ../syntax/ext/base/struct.ExtCtxt.html#method.span_fatal
[26] ../syntax/ext/base/struct.ExtCtxt.html#method.span_err
[27] ../syntax/ext/base/struct.DummyResult.html
[28] ../syntax/ext/base/struct.ExtCtxt.html#method.span_note
[29] https://doc.rust-lang.org/syntax/print/pprust/index.html#functions
[30] ../syntax/ext/build/trait.AstBuilder.html#tymethod.expr_usize

macros[31]. They are undocumented and very rough around the edges. However, the implementation may be a good starting point for an improved quasiquote as an ordinary plugin library.

5.1.3 Lint plugins

Plugins can extend Rust's lint infrastructure[32] with additional checks for code style, safety, etc. Now let's write a plugin `lint_plugin_test.rs`[33] that warns about any item named `lintme`.

```
#![feature(plugin_registrar)]
#![feature(box_syntax, rustc_private)]

extern crate syntax;

// Load rustc as a plugin to get macros
#[macro_use]
extern crate rustc;

use rustc::lint::{EarlyContext, LintContext, LintPass, EarlyLintPass,
                  EarlyLintPassObject, LintArray};
use rustc::plugin::Registry;
use syntax::ast;

declare_lint!(TEST_LINT, Warn, "Warn about items named 'lintme'");

struct Pass;

impl LintPass for Pass {
    fn get_lints(&self) -> LintArray {
        lint_array!(TEST_LINT)
    }
}

impl EarlyLintPass for Pass {
    fn check_item(&mut self, cx: &EarlyContext, it: &ast::Item) {
        if it.ident.name.as_str() == "lintme" {
            cx.span_lint(TEST_LINT, it.span, "item is named 'lintme'");
        }
    }
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_early_lint_pass(box Pass as EarlyLintPassObject);
```

[31] ../syntax/ext/quote/index.html
[32] ../reference.html#lint-check-attributes
[33] https://github.com/rust-lang/rust/blob/master/src/test/auxiliary/lint_plugin_test.rs

```
}
```

Then code like

```
#![plugin(lint_plugin_test)]

fn lintme() { }
```

will produce a compiler warning:

```
foo.rs:4:1: 4:16 warning: item is named 'lintme', #[warn(test_lint)] on by default
foo.rs:4 fn lintme() { }
         ^~~~~~~~~~~~~~~
```

The components of a lint plugin are:

- one or more `declare_lint!` invocations, which define static `Lint`[34] structs;
- a struct holding any state needed by the lint pass (here, none);
- a `LintPass`[35] implementation defining how to check each syntax element. A single `LintPass` may call `span_lint` for several different `Lint`s, but should register them all through the `get_lints` method.

Lint passes are syntax traversals, but they run at a late stage of compilation where type information is available. `rustc`'s built-in lints[36] mostly use the same infrastructure as lint plugins, and provide examples of how to access type information.

Lints defined by plugins are controlled by the usual attributes and compiler flags[37], e.g. `#[allow(test_lint)]` or `-A test-lint`. These identifiers are derived from the first argument to `declare_lint!`, with appropriate case and punctuation conversion.

You can run `rustc -W help foo.rs` to see a list of lints known to `rustc`, including those provided by plugins loaded by `foo.rs`.

5.2 Inline Assembly

For extremely low-level manipulations and performance reasons, one might wish to control the CPU directly. Rust supports using inline assembly to do this via the `asm!` macro. The syntax roughly matches that of GCC & Clang:

```
asm!(assembly template
   : output operands
   : input operands
   : clobbers
   : options
   );
```

[34] ../rustc/lint/struct.Lint.html
[35] ../rustc/lint/trait.LintPass.html
[36] https://github.com/rust-lang/rust/blob/master/src/librustc/lint/builtin.rs
[37] ../reference.html#lint-check-attributes

Any use of `asm` is feature gated (requires `#![feature(asm)]` on the crate to allow) and of course requires an `unsafe` block.

> **Note**: the examples here are given in x86/x86-64 assembly, but all platforms are supported.

Assembly template

The `assembly template` is the only required parameter and must be a literal string (i.e. `""`)

```
#![feature(asm)]

#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn foo() {
    unsafe {
        asm!("NOP");
    }
}

// other platforms
#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
fn foo() { /* ... */ }

fn main() {
    // ...
    foo();
    // ...
}
```

(The `feature(asm)` and `#[cfg]`s are omitted from now on.)
Output operands, input operands, clobbers and options are all optional but you must add the right number of `:` if you skip them:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
asm!("xor %eax, %eax"
    :
    :
    : "{eax}"
   );
# } }
```

Whitespace also doesn't matter:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
asm!("xor %eax, %eax" ::: "{eax}");
# } }
```

Operands

Input and output operands follow the same format: `: "constraints1"(expr1), "constraints2"(expr2), ...`. Output operand expressions must be mutable lvalues, or not yet assigned:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn add(a: i32, b: i32) -> i32 {
    let c: i32;
    unsafe {
        asm!("add $2, $0"
             : "=r"(c)
             : "0"(a), "r"(b)
             );
    }
    c
}
# #[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
# fn add(a: i32, b: i32) -> i32 { a + b }

fn main() {
    assert_eq!(add(3, 14159), 14162)
}
```

If you would like to use real operands in this position, however, you are required to put curly braces `{}` around the register that you want, and you are required to put the specific size of the operand. This is useful for very low level programming, where which register you use is important:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# unsafe fn read_byte_in(port: u16) -> u8 {
let result: u8;
asm!("in %dx, %al" : "={al}"(result) : "{dx}"(port));
result
# }
```

Clobbers

Some instructions modify registers which might otherwise have held different values so we use the clobbers list to indicate to the compiler not to assume any values loaded into those registers will stay valid.

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
// Put the value 0x200 in eax
asm!("mov $$0x200, %eax" : /* no outputs */ : /* no inputs */ : "{eax}");
# } }
```

Input and output registers need not be listed since that information is already communicated by the given constraints. Otherwise, any other registers used either implicitly or explicitly should be listed.

If the assembly changes the condition code register `cc` should be specified as one of the clobbers. Similarly, if the assembly modifies memory, `memory` should also be specified.

Options

The last section, `options` is specific to Rust. The format is comma separated literal strings (i.e. `:"foo", "bar", "baz"`). It's used to specify some extra info about the inline assembly:

Current valid options are:

1. *volatile* - specifying this is analogous to `__asm__ __volatile__ (...)` in gcc/-clang.
2. *alignstack* - certain instructions expect the stack to be aligned a certain way (i.e. SSE) and specifying this indicates to the compiler to insert its usual stack alignment code
3. *intel* - use intel syntax instead of the default AT&T.

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() {
let result: i32;
unsafe {
   asm!("mov eax, 2" : "={eax}"(result) : : : "intel")
}
println!("eax is currently {}", result);
# }
```

More Information

The current implementation of the `asm!` macro is a direct binding to LLVM's inline assembler expressions[38], so be sure to check out their documentation as well[39] for more information about clobbers, constraints, etc.

5.3 No stdlib

By default, `std` is linked to every Rust crate. In some contexts, this is undesirable, and can be avoided with the `#![no_std]` attribute attached to the crate.

Obviously there's more to life than just libraries: one can use `#[no_std]` with an executable, controlling the entry point is possible in two ways: the `#[start]` attribute, or overriding the default shim for the C `main` function with your own.

[38] `http://llvm.org/docs/LangRef.html#inline-assembler-expressions`
[39] `http://llvm.org/docs/LangRef.html#inline-assembler-expressions`

The function marked `#[start]` is passed the command line parameters in the same format as C:

```
# #![feature(libc)]
#![feature(lang_items)]
#![feature(start)]
#![feature(no_std)]
#![no_std]

// Pull in the system libc library for what crt0.o likely requires
extern crate libc;

// Entry point for this program
#[start]
fn start(_argc: isize, _argv: *const *const u8) -> isize {
    0
}

// These functions and traits are used by the compiler, but not
// for a bare-bones hello world. These are normally
// provided by libstd.
#[lang = ”eh_personality”] extern fn eh_personality() {}
#[lang = ”panic_fmt”] fn panic_fmt() -> ! { loop {} }
# #[lang = ”eh_unwind_resume”] extern fn rust_eh_unwind_resume() {}
# #[no_mangle] pub extern fn rust_eh_register_frames () {}
# #[no_mangle] pub extern fn rust_eh_unregister_frames () {}
# // fn main() {} tricked you, rustdoc!
```

To override the compiler-inserted `main` shim, one has to disable it with `#![no_main]` and then create the appropriate symbol with the correct ABI and the correct name, which requires overriding the compiler’s name mangling too:

```
# #![feature(libc)]
#![feature(no_std)]
#![feature(lang_items)]
#![feature(start)]
#![no_std]
#![no_main]

extern crate libc;

#[no_mangle] // ensure that this symbol is called ‘main‘ in the output
pub extern fn main(argc: i32, argv: *const *const u8) -> i32 {
    0
}

#[lang = ”eh_personality”] extern fn eh_personality() {}
#[lang = ”panic_fmt”] fn panic_fmt() -> ! { loop {} }
# #[lang = ”eh_unwind_resume”] extern fn rust_eh_unwind_resume() {}
```

```
# #[no_mangle] pub extern fn rust_eh_register_frames () {}
# #[no_mangle] pub extern fn rust_eh_unregister_frames () {}
# // fn main() {} tricked you, rustdoc!
```

The compiler currently makes a few assumptions about symbols which are available in the executable to call. Normally these functions are provided by the standard library, but without it you must define your own.

The first of these two functions, `eh_personality`, is used by the failure mechanisms of the compiler. This is often mapped to GCC's personality function (see the libstd implementation[40] for more information), but crates which do not trigger a panic can be assured that this function is never called. The second function, `panic_fmt`, is also used by the failure mechanisms of the compiler.

Using libcore

> **Note**: the core library's structure is unstable, and it is recommended to use the standard library instead wherever possible.

With the above techniques, we've got a bare-metal executable running some Rust code. There is a good deal of functionality provided by the standard library, however, that is necessary to be productive in Rust. If the standard library is not sufficient, then libcore[41] is designed to be used instead.

The core library has very few dependencies and is much more portable than the standard library itself. Additionally, the core library has most of the necessary functionality for writing idiomatic and effective Rust code. When using `#![no_std]`, Rust will automatically inject the `core` crate, just like we do for `std` when we're using it.

As an example, here is a program that will calculate the dot product of two vectors provided from C, using idiomatic Rust practices.

```
# #![feature(libc)]
#![feature(lang_items)]
#![feature(start)]
#![feature(no_std)]
#![feature(core)]
#![feature(core_slice_ext)]
#![feature(raw)]
#![no_std]

extern crate libc;

use core::mem;

#[no_mangle]
pub extern fn dot_product(a: *const u32, a_len: u32,
                          b: *const u32, b_len: u32) -> u32 {
```

[40] ../std/rt/unwind/index.html
[41] ../core/index.html

```
    use core::raw::Slice;

    // Convert the provided arrays into Rust slices.
    // The core::raw module guarantees that the Slice
    // structure has the same memory layout as a &[T]
    // slice.
    //
    // This is an unsafe operation because the compiler
    // cannot tell the pointers are valid.
    let (a_slice, b_slice): (&[u32], &[u32]) = unsafe {
        mem::transmute((
            Slice { data: a, len: a_len as usize },
            Slice { data: b, len: b_len as usize },
        ))
    };

    // Iterate over the slices, collecting the result
    let mut ret = 0;
    for (i, j) in a_slice.iter().zip(b_slice.iter()) {
        ret += (*i) * (*j);
    }
    return ret;
}

#[lang = ”panic_fmt”]
extern fn panic_fmt(args: &core::fmt::Arguments,
                    file: &str,
                    line: u32) -> ! {
    loop {}
}

#[lang = ”eh_personality”] extern fn eh_personality() {}
# #[start] fn start(argc: isize, argv: *const *const u8) -> isize { 0 }
# #[lang = ”eh_unwind_resume”] extern fn rust_eh_unwind_resume() {}
# #[no_mangle] pub extern fn rust_eh_register_frames () {}
# #[no_mangle] pub extern fn rust_eh_unregister_frames () {}
# fn main() {}
```

Note that there is one extra lang item here which differs from the examples above, `panic_fmt`. This must be defined by consumers of libcore because the core library declares panics, but it does not define it. The `panic_fmt` lang item is this crate’s definition of panic, and it must be guaranteed to never return.

As can be seen in this example, the core library is intended to provide the power of Rust in all circumstances, regardless of platform requirements. Further libraries, such as liballoc, add functionality to libcore which make other platform-specific assumptions, but continue to be more portable than the standard library itself.

5.4 Intrinsics

> **Note**: intrinsics will forever have an unstable interface, it is recommended to use the stable interfaces of libcore rather than intrinsics directly.

These are imported as if they were FFI functions, with the special `rust-intrinsic` ABI. For example, if one was in a freestanding context, but wished to be able to `transmute` between types, and perform efficient pointer arithmetic, one would import those functions via a declaration like

```
#![feature(intrinsics)]
# fn main() {}

extern "rust-intrinsic" {
    fn transmute<T, U>(x: T) -> U;

    fn offset<T>(dst: *const T, offset: isize) -> *const T;
}
```

As with any other FFI functions, these are always `unsafe` to call.

5.5 Lang items

> **Note**: lang items are often provided by crates in the Rust distribution, and lang items themselves have an unstable interface. It is recommended to use officially distributed crates instead of defining your own lang items.

The `rustc` compiler has certain pluggable operations, that is, functionality that isn't hard-coded into the language, but is implemented in libraries, with a special marker to tell the compiler it exists. The marker is the attribute `#[lang = "..."]` and there are various different values of `...`, i.e. various different 'lang items'.

For example, `Box` pointers require two lang items, one for allocation and one for deallocation. A freestanding program that uses the `Box` sugar for dynamic allocations via `malloc` and `free`:

```
#![feature(lang_items, box_syntax, start, no_std, libc)]
#![no_std]

extern crate libc;

extern {
    fn abort() -> !;
}

#[lang = "owned_box"]
pub struct Box<T>(*mut T);
```

```
#[lang = ”exchange_malloc”]
unsafe fn allocate(size: usize, _align: usize) -> *mut u8 {
    let p = libc::malloc(size as libc::size_t) as *mut u8;

    // malloc failed
    if p as usize == 0 {
        abort();
    }

    p
}
#[lang = ”exchange_free”]
unsafe fn deallocate(ptr: *mut u8, _size: usize, _align: usize) {
    libc::free(ptr as *mut libc::c_void)
}

#[start]
fn main(argc: isize, argv: *const *const u8) -> isize {
    let x = box 1;

    0
}

#[lang = ”eh_personality”] extern fn eh_personality() {}
#[lang = ”panic_fmt”] fn panic_fmt() -> ! { loop {} }
# #[lang = ”eh_unwind_resume”] extern fn rust_eh_unwind_resume() {}
# #[no_mangle] pub extern fn rust_eh_register_frames () {}
# #[no_mangle] pub extern fn rust_eh_unregister_frames () {}
```

Note the use of `abort`: the `exchange_malloc` lang item is assumed to return a valid pointer, and so needs to do the check internally.

Other features provided by lang items include:

- overloadable operators via traits: the traits corresponding to the `==`, `<`, dereferencing (`*`) and `+` (etc.) operators are all marked with lang items; those specific four are `eq`, `ord`, `deref`, and `add` respectively.
- stack unwinding and general failure; the `eh_personality`, `fail` and `fail_bounds_checks` lang items.
- the traits in `std::marker` used to indicate types of various kinds; lang items `send`, `sync` and `copy`.
- the marker types and variance indicators found in `std::marker`; lang items `covariant_type`, `contravariant_lifetime`, etc.

Lang items are loaded lazily by the compiler; e.g. if one never uses `Box` then there is no need to define functions for `exchange_malloc` and `exchange_free`. `rustc` will emit an error when an item is needed but not found in the current crate or any that it depends on.

5.6 Advanced linking

The common cases of linking with Rust have been covered earlier in this book, but supporting the range of linking possibilities made available by other languages is important for Rust to achieve seamless interaction with native libraries.

5.6.1 Link args

There is one other way to tell `rustc` how to customize linking, and that is via the `link_args` attribute. This attribute is applied to `extern` blocks and specifies raw flags which need to get passed to the linker when producing an artifact. An example usage would be:

```
#![feature(link_args)]

#[link_args = "-foo -bar -baz"]
extern {}
# fn main() {}
```

Note that this feature is currently hidden behind the `feature(link_args)` gate because this is not a sanctioned way of performing linking. Right now `rustc` shells out to the system linker (`gcc` on most systems, `link.exe` on MSVC), so it makes sense to provide extra command line arguments, but this will not always be the case. In the future `rustc` may use LLVM directly to link native libraries, in which case `link_args` will have no meaning. You can achieve the same effect as the `link_args` attribute with the `-C link-args` argument to `rustc`.

It is highly recommended to *not* use this attribute, and rather use the more formal `#[link(...)]` attribute on `extern` blocks instead.

5.6.2 Static linking

Static linking refers to the process of creating output that contains all required libraries and so doesn't need libraries installed on every system where you want to use your compiled project. Pure-Rust dependencies are statically linked by default so you can use created binaries and libraries without installing Rust everywhere. By contrast, native libraries (e.g. `libc` and `libm`) are usually dynamically linked, but it is possible to change this and statically link them as well.

Linking is a very platform-dependent topic, and static linking may not even be possible on some platforms! This section assumes some basic familiarity with linking on your platform of choice.

Linux

By default, all Rust programs on Linux will link to the system `libc` along with a number of other libraries. Let's look at an example on a 64-bit Linux machine with GCC and `glibc` (by far the most common `libc` on Linux):

```
$ cat example.rs
fn main() {}
$ rustc example.rs
$ ldd example
        linux-vdso.so.1 =>  (0x00007ffd565fd000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fa81889c000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fa81867e000)
        librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007fa818475000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fa81825f000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa817e9a000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fa818cf9000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fa817b93000)
```

Dynamic linking on Linux can be undesirable if you wish to use new library features on old systems or target systems which do not have the required dependencies for your program to run.

Static linking is supported via an alternative `libc`, `musl`[42]. You can compile your own version of Rust with `musl` enabled and install it into a custom directory with the instructions below:

```
$ mkdir musldist
$ PREFIX=$(pwd)/musldist
$
$ # Build musl
$ curl -O http://www.musl-libc.org/releases/musl-1.1.10.tar.gz
$ tar xf musl-1.1.10.tar.gz
$ cd musl-1.1.10/
musl-1.1.10 $ ./configure --disable-shared --prefix=$PREFIX
musl-1.1.10 $ make
musl-1.1.10 $ make install
musl-1.1.10 $ cd ..
$ du -h musldist/lib/libc.a
2.2M    musldist/lib/libc.a
$
$ # Build libunwind.a
$ curl -O http://llvm.org/releases/3.7.0/llvm-3.7.0.src.tar.xz
$ tar xf llvm-3.7.0.src.tar.xz
$ cd llvm-3.7.0.src/projects/
llvm-3.7.0.src/projects $ curl http://llvm.org/releases/3.7.0/libunwind-3.7.0.src.tar.xz | tar
llvm-3.7.0.src/projects $ mv libunwind-3.7.0.src libunwind
llvm-3.7.0.src/projects $ mkdir libunwind/build
llvm-3.7.0.src/projects $ cd libunwind/build
llvm-3.7.0.src/projects/libunwind/build $ cmake -DLLVM_PATH=../../.. -DLIBUNWIND_ENABLE_SHARED
llvm-3.7.0.src/projects/libunwind/build $ make
llvm-3.7.0.src/projects/libunwind/build $ cp lib/libunwind.a $PREFIX/lib/
llvm-3.7.0.src/projects/libunwind/build $ cd ../../../../
$ du -h musldist/lib/libunwind.a
```

[42] `http://www.musl-libc.org`

```
164K    musldist/lib/libunwind.a
$
$ # Build musl-enabled rust
$ git clone https://github.com/rust-lang/rust.git muslrust
$ cd muslrust
muslrust $ ./configure --target=x86_64-unknown-linux-musl --musl-root=$PREFIX --prefix=$PREFIX
muslrust $ make
muslrust $ make install
muslrust $ cd ..
$ du -h musldist/bin/rustc
12K     musldist/bin/rustc
```

You now have a build of a `musl`-enabled Rust! Because we've installed it to a custom prefix we need to make sure our system can find the binaries and appropriate libraries when we try and run it:

```
$ export PATH=$PREFIX/bin:$PATH
$ export LD_LIBRARY_PATH=$PREFIX/lib:$LD_LIBRARY_PATH
```

Let's try it out!

```
$ echo 'fn main() { println!("hi!"); panic!("failed"); }' > example.rs
$ rustc --target=x86_64-unknown-linux-musl example.rs
$ ldd example
        not a dynamic executable
$ ./example
hi!
thread '<main>' panicked at 'failed', example.rs:1
```

Success! This binary can be copied to almost any Linux machine with the same machine architecture and run without issues.

`cargo build` also permits the `--target` option so you should be able to build your crates as normal. However, you may need to recompile your native libraries against `musl` before they can be linked against.

5.7 Benchmark Tests

Rust supports benchmark tests, which can test the performance of your code. Let's make our `src/lib.rs` look like this (comments elided):

```
#![feature(test)]

extern crate test;

pub fn add_two(a: i32) -> i32 {
    a + 2
}
```

```
#[cfg(test)]
mod tests {
    use super::*;
    use test::Bencher;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }

    #[bench]
    fn bench_add_two(b: &mut Bencher) {
        b.iter(|| add_two(2));
    }
}
```

Note the `test` feature gate, which enables this unstable feature.

We've imported the `test` crate, which contains our benchmarking support. We have a new function as well, with the `bench` attribute. Unlike regular tests, which take no arguments, benchmark tests take a `&mut Bencher`. This `Bencher` provides an `iter` method, which takes a closure. This closure contains the code we'd like to benchmark.

We can run benchmark tests with `cargo bench`:

```
$ cargo bench
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/release/adder-91b3e234d4ed382a

running 2 tests
test tests::it_works ... ignored
test tests::bench_add_two ... bench:         1 ns/iter (+/- 0)

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

Our non-benchmark test was ignored. You may have noticed that `cargo bench` takes a bit longer than `cargo test`. This is because Rust runs our benchmark a number of times, and then takes the average. Because we're doing so little work in this example, we have a `1 ns/iter (+/- 0)`, but this would show the variance if there was one.

Advice on writing benchmarks:

- Move setup code outside the `iter` loop; only put the part you want to measure inside
- Make the code do "the same thing" on each iteration; do not accumulate or change state
- Make the outer function idempotent too; the benchmark runner is likely to run it many times
- Make the inner `iter` loop short and fast so benchmark runs are fast and the calibrator can adjust the run-length at fine resolution

- Make the code in the `iter` loop do something simple, to assist in pinpointing performance improvements (or regressions)

Gotcha: optimizations

There's another tricky part to writing benchmarks: benchmarks compiled with optimizations activated can be dramatically changed by the optimizer so that the benchmark is no longer benchmarking what one expects. For example, the compiler might recognize that some calculation has no external effects and remove it entirely.

```
#![feature(test)]

extern crate test;
use test::Bencher;

#[bench]
fn bench_xor_1000_ints(b: &mut Bencher) {
    b.iter(|| {
        (0..1000).fold(0, |old, new| old ^ new);
    });
}
```

gives the following results

```
running 1 test
test bench_xor_1000_ints ... bench:         0 ns/iter (+/- 0)

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

The benchmarking runner offers two ways to avoid this. Either, the closure that the `iter` method receives can return an arbitrary value which forces the optimizer to consider the result used and ensures it cannot remove the computation entirely. This could be done for the example above by adjusting the `b.iter` call to

```
# struct X;
# impl X { fn iter<T, F>(&self, _: F) where F: FnMut() -> T {} } let b = X;
b.iter(|| {
    // note lack of ';' (could also use an explicit 'return').
    (0..1000).fold(0, |old, new| old ^ new)
});
```

Or, the other option is to call the generic `test::black_box` function, which is an opaque "black box" to the optimizer and so forces it to consider any argument as used.

```
#![feature(test)]

extern crate test;
```

```
# fn main() {
# struct X;
# impl X { fn iter<T, F>(&self, _: F) where F: FnMut() -> T {} } let b = X;
b.iter(|| {
    let n = test::black_box(1000);

    (0..n).fold(0, |a, b| a ^ b)
})
# }
```

Neither of these read or modify the value, and are very cheap for small values. Larger values can be passed indirectly to reduce overhead (e.g. `black_box(&huge_struct)`).

Performing either of the above changes gives the following benchmarking results

```
running 1 test
test bench_xor_1000_ints ... bench:       131 ns/iter (+/- 3)

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

However, the optimizer can still modify a testcase in an undesirable manner even when using either of the above.

5.8 Box Syntax and Patterns

Currently the only stable way to create a `Box` is via the `Box::new` method. Also it is not possible in stable Rust to destructure a `Box` in a match pattern. The unstable `box` keyword can be used to both create and destructure a `Box`. An example usage would be:

```
#![feature(box_syntax, box_patterns)]

fn main() {
    let b = Some(box 5);
    match b {
        Some(box n) if n < 0 => {
            println!(”Box contains negative number {}”, n);
        },
        Some(box n) if n >= 0 => {
            println!(”Box contains non-negative number {}”, n);
        },
        None => {
            println!(”No box”);
        },
        _ => unreachable!()
    }
}
```

Note that these features are currently hidden behind the `box_syntax` (box creation) and `box_patterns` (destructuring and pattern matching) gates because the syntax may still change in the future.

5.8.1 Returning Pointers

In many languages with pointers, you'd return a pointer from a function so as to avoid copying a large data structure. For example:

```
struct BigStruct {
    one: i32,
    two: i32,
    // etc
    one_hundred: i32,
}

fn foo(x: Box<BigStruct>) -> Box<BigStruct> {
    Box::new(*x)
}

fn main() {
    let x = Box::new(BigStruct {
        one: 1,
        two: 2,
        one_hundred: 100,
    });

    let y = foo(x);
}
```

The idea is that by passing around a box, you're only copying a pointer, rather than the hundred `i32`s that make up the `BigStruct`.

This is an antipattern in Rust. Instead, write this:

```
#![feature(box_syntax)]

struct BigStruct {
    one: i32,
    two: i32,
    // etc
    one_hundred: i32,
}

fn foo(x: Box<BigStruct>) -> BigStruct {
    *x
}

fn main() {
```

```
    let x = Box::new(BigStruct {
        one: 1,
        two: 2,
        one_hundred: 100,
    });

    let y: Box<BigStruct> = box foo(x);
}
```

This gives you flexibility without sacrificing performance.

You may think that this gives us terrible performance: return a value and then immediately box it up ?! Isn't this pattern the worst of both worlds? Rust is smarter than that. There is no copy in this code. `main` allocates enough room for the `box`, passes a pointer to that memory into `foo` as `x`, and then `foo` writes the value straight into the `Box<T>`.

This is important enough that it bears repeating: pointers are not for optimizing returning values from your code. Allow the caller to choose how they want to use your output.

5.9 Slice Patterns

If you want to match against a slice or array, you can use `&` with the `slice_patterns` feature:

```
#![feature(slice_patterns)]

fn main() {
    let v = vec!["match_this", "1"];

    match &v[..] {
        ["match_this", second] => println!("The second element is {}", second),
        _ => {},
    }
}
```

The `advanced_slice_patterns` gate lets you use `..` to indicate any number of elements inside a pattern matching a slice. This wildcard can only be used once for a given array. If there's an identifier before the `..`, the result of the slice will be bound to that name. For example:

```
#![feature(advanced_slice_patterns, slice_patterns)]

fn is_symmetric(list: &[u32]) -> bool {
    match list {
        [] | [_] => true,
        [x, inside.., y] if x == y => is_symmetric(inside),
        _ => false
```

```
    }
}

fn main() {
    let sym = &[0, 1, 4, 2, 4, 1, 0];
    assert!(is_symmetric(sym));

    let not_sym = &[0, 1, 7, 2, 4, 1, 0];
    assert!(!is_symmetric(not_sym));
}
```

5.10 Associated Constants

With the `associated_consts` feature, you can define constants like this:

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
    const ID: i32 = 1;
}

fn main() {
    assert_eq!(1, i32::ID);
}
```

Any implementor of `Foo` will have to define `ID`. Without the definition:

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
}
```

gives

```
error: not all trait items implemented, missing: `ID` [E0046]
     impl Foo for i32 {
     }
```

A default value can be implemented as well:

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32 = 1;
}

impl Foo for i32 {
}

impl Foo for i64 {
    const ID: i32 = 5;
}

fn main() {
    assert_eq!(1, i32::ID);
    assert_eq!(5, i64::ID);
}
```

As you can see, when implementing `Foo`, you can leave it unimplemented, as with `i32`. It will then use the default value. But, as in `i64`, we can also add our own definition.

Associated constants don't have to be associated with a trait. An `impl` block for a `struct` or an `enum` works fine too:

```
#![feature(associated_consts)]

struct Foo;

impl Foo {
    const FOO: u32 = 3;
}
```

5.11 Custom Allocators

Allocating memory isn't always the easiest thing to do, and while Rust generally takes care of this by default it often becomes necessary to customize how allocation occurs. The compiler and standard library currently allow switching out the default global allocator in use at compile time. The design is currently spelled out in RFC 1183[43] but this will walk you through how to get your own allocator up and running.

5.11.1 Default Allocator

The compiler currently ships two default allocators: `alloc_system` and `alloc_jemalloc` (some targets don't have jemalloc, however). These allocators are just normal Rust crates and contain an implementation of the routines to allocate and deallocate memory. The standard library is not compiled assuming either one, and the compiler will

[43] https://github.com/rust-lang/rfcs/blob/master/text/1183-swap-out-jemalloc.md

decide which allocator is in use at compile-time depending on the type of output artifact being produced.

Binaries generated by the compiler will use `alloc_jemalloc` by default (where available). In this situation the compiler "controls the world" in the sense of it has power over the final link. Primarily this means that the allocator decision can be left up the compiler.

Dynamic and static libraries, however, will use `alloc_system` by default. Here Rust is typically a 'guest' in another application or another world where it cannot authoritatively decide what allocator is in use. As a result it resorts back to the standard APIs (e.g. `malloc` and `free`) for acquiring and releasing memory.

5.11.2 Switching Allocators

Although the compiler's default choices may work most of the time, it's often necessary to tweak certain aspects. Overriding the compiler's decision about which allocator is in use is done simply by linking to the desired allocator:

```
#![feature(alloc_system)]

extern crate alloc_system;

fn main() {
    let a = Box::new(4); // allocates from the system allocator
    println!("{}", a);
}
```

In this example the binary generated will not link to jemalloc by default but instead use the system allocator. Conversely to generate a dynamic library which uses jemalloc by default one would write:

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

extern crate alloc_jemalloc;

pub fn foo() {
    let a = Box::new(4); // allocates from jemalloc
    println!("{}", a);
}
# fn main() {}
```

5.11.3 Writing a custom allocator

Sometimes even the choices of jemalloc vs the system allocator aren't enough and an entirely new custom allocator is required. In this you'll write your own crate which implements the allocator API (e.g. the same as `alloc_system` or `alloc_jemalloc`). As an example, let's take a look at a simplified and annotated version of `alloc_system`

```
# // only needed for rustdoc --test down below
# #![feature(lang_items)]
// The compiler needs to be instructed that this crate is an allocator in order
// to realize that when this is linked in another allocator like jemalloc should
// not be linked in
#![feature(allocator)]
#![allocator]

// Allocators are not allowed to depend on the standard library which in turn
// requires an allocator in order to avoid circular dependencies. This crate,
// however, can use all of libcore.
#![feature(no_std)]
#![no_std]

// Let's give a unique name to our custom allocator
#![crate_name = "my_allocator"]
#![crate_type = "rlib"]

// Our system allocator will use the in-tree libc crate for FFI bindings. Note
// that currently the external (crates.io) libc cannot be used because it links
// to the standard library (e.g. `#![no_std]` isn't stable yet), so that's why
// this specifically requires the in-tree version.
#![feature(libc)]
extern crate libc;

// Listed below are the five allocation functions currently required by custom
// allocators. Their signatures and symbol names are not currently typechecked
// by the compiler, but this is a future extension and are required to match
// what is found below.
//
// Note that the standard `malloc` and `realloc` functions do not provide a way
// to communicate alignment so this implementation would need to be improved
// with respect to alignment in that aspect.

#[no_mangle]
pub extern fn __rust_allocate(size: usize, _align: usize) -> *mut u8 {
    unsafe { libc::malloc(size as libc::size_t) as *mut u8 }
}

#[no_mangle]
pub extern fn __rust_deallocate(ptr: *mut u8, _old_size: usize, _align: usize) {
    unsafe { libc::free(ptr as *mut libc::c_void) }
}

#[no_mangle]
pub extern fn __rust_reallocate(ptr: *mut u8, _old_size: usize, size: usize,
                                _align: usize) -> *mut u8 {
    unsafe {
        libc::realloc(ptr as *mut libc::c_void, size as libc::size_t) as *mut u8
```

```
    }
}

#[no_mangle]
pub extern fn __rust_reallocate_inplace(_ptr: *mut u8, old_size: usize,
                                        _size: usize, _align: usize) -> usize {
    old_size // this api is not supported by libc
}

#[no_mangle]
pub extern fn __rust_usable_size(size: usize, _align: usize) -> usize {
    size
}

# // just needed to get rustdoc to test this
# fn main() {}
# #[lang = ”panic_fmt”] fn panic_fmt() {}
# #[lang = ”eh_personality”] fn eh_personality() {}
# #[lang = ”eh_unwind_resume”] extern fn eh_unwind_resume() {}
# #[no_mangle] pub extern fn rust_eh_register_frames () {}
# #[no_mangle] pub extern fn rust_eh_unregister_frames () {}
```

After we compile this crate, it can be used as follows:

```
extern crate my_allocator;

fn main() {
    let a = Box::new(8); // allocates memory via our custom allocator crate
    println!(”{}”, a);
}
```

5.11.4 Custom allocator limitations

There are a few restrictions when working with custom allocators which may cause compiler errors:

- Any one artifact may only be linked to at most one allocator. Binaries, dylibs, and staticlibs must link to exactly one allocator, and if none have been explicitly chosen the compiler will choose one. On the other hand rlibs do not need to link to an allocator (but still can).
- A consumer of an allocator is tagged with `#![needs_allocator]` (e.g. the `liballoc` crate currently) and an `#[allocator]` crate cannot transitively depend on a crate which needs an allocator (e.g. circular dependencies are not allowed). This basically means that allocators must restrict themselves to libcore currently.

Chapter 6

Glossary

Not every Rustacean has a background in systems programming, nor in computer science, so we've added explanations of terms that might be unfamiliar.

Abstract Syntax Tree

When a compiler is compiling your program, it does a number of different things. One of the things that it does is turn the text of your program into an 'abstract syntax tree', or 'AST'. This tree is a representation of the structure of your program. For example, `2 + 3` can be turned into a tree:

```
  +
 / \
2   3
```

And `2 + (3 * 4)` would look like this:

```
  +
 / \
2   *
   / \
  3   4
```

Arity

Arity refers to the number of arguments a function or operation takes.

```
let x = (2, 3);
let y = (4, 6);
let z = (8, 2, 6);
```

In the example above `x` and `y` have arity 2. `z` has arity 3.

Bounds

Bounds are constraints on a type or trait[1]. For example, if a bound is placed on the argument a function takes, types passed to that function must abide by that constraint.

DST (Dynamically Sized Type)

A type without a statically known size or alignment. (more info[2])

Expression

In computer programming, an expression is a combination of values, constants, variables, operators and functions that evaluate to a single value. For example, `2 + (3 * 4)` is an expression that returns the value 14. It is worth noting that expressions can have side-effects. For example, a function included in an expression might perform actions other than simply returning a value.

Expression-Oriented Language

In early programming languages, expressions[3] and statements[4] were two separate syntactic categories: expressions had a value and statements did things. However, later languages blurred this distinction, allowing expressions to do things and statements to have a value. In an expression-oriented language, (nearly) every statement is an expression and therefore returns a value. Consequently, these expression statements can themselves form part of larger expressions.

Statement

In computer programming, a statement is the smallest standalone element of a programming language that commands a computer to perform an action.

[1] `traits.html`
[2] `../nomicon/exotic-sizes.html#dynamically-sized-types-dsts`
[3] `glossary.html#expression`
[4] `glossary.html#statement`

Chapter 7

Syntax Index

7.0.5 Keywords

- `as`: primitive casting. See Casting Between Types (`as`)[1].
- `break`: break out of loop. See Loops (Ending Iteration Early)[2].
- `const`: constant items. See `const` and `static`[3].
- `continue`: continue to next loop iteration. See Loops (Ending Iteration Early)[4].
- `crate`: external crate linkage. See Crates and Modules (Importing External Crates)[5].
- `else`: fallback for `if` and `if let` constructs. See `if`[6], `if let`[7].
- `enum`: defining enumeration. See Enums[8].
- `extern`: external crate, function, and variable linkage. See Crates and Modules (Importing External Crates)[9], Foreign Function Interface[10].
- `false`: boolean false literal. See Primitive Types (Booleans)[11].
- `fn`: function definition and function pointer types. See Functions[12].
- `for`: iterator loop, part of trait `impl` syntax, and higher-ranked lifetime syntax. See Loops (`for`)[13], Method Syntax[14].
- `if`: conditional branching. See `if`[15], `if let`[16].
- `impl`: inherent and trait implementation blocks. See Method Syntax[17].

[1] `casting-between-types.html#as`
[2] `loops.html#ending-iteration-early`
[3] `const-and-static.html`
[4] `loops.html#ending-iteration-early`
[5] `crates-and-modules.html#importing-external-crates`
[6] `if.html`
[7] `if-let.html`
[8] `enums.html`
[9] `crates-and-modules.html#importing-external-crates`
[10] `ffi.html`
[11] `primitive-types.html#booleans`
[12] `functions.html`
[13] `loops.html#for`
[14] `method-syntax.html`
[15] `if.html`
[16] `if-let.html`
[17] `method-syntax.html`

- `in`: part of `for` loop syntax. See Loops (`for`)[18].
- `let`: variable binding. See Variable Bindings[19].
- `loop`: unconditional, infinite loop. See Loops (`loop`)[20].
- `match`: pattern matching. See Match[21].
- `mod`: module declaration. See Crates and Modules (Defining Modules)[22].
- `move`: part of closure syntax. See Closures (`move` closures)[23].
- `mut`: denotes mutability in pointer types and pattern bindings. See Mutability[24].
- `pub`: denotes public visibility in `struct` fields, `impl` blocks, and modules. See Crates and Modules (Exporting a Public Interface)[25].
- `ref`: by-reference binding. See Patterns (`ref` and `ref mut`)[26].
- `return`: return from function. See Functions (Early Returns)[27].
- `Self`: implementor type alias. See Traits[28].
- `self`: method subject. See Method Syntax (Method Calls)[29].
- `static`: global variable. See `const` and `static` (`static`)[30].
- `struct`: structure definition. See Structs[31].
- `trait`: trait definition. See Traits[32].
- `true`: boolean true literal. See Primitive Types (Booleans)[33].
- `type`: type alias, and associated type definition. See `type` Aliases[34], Associated Types[35].
- `unsafe`: denotes unsafe code, functions, traits, and implementations. See Unsafe[36].
- `use`: import symbols into scope. See Crates and Modules (Importing Modules with `use`)[37].
- `where`: type constraint clauses. See Traits (`where` clause)[38].
- `while`: conditional loop. See Loops (`while`)[39].

7.0.6 Operators and Symbols

- `!` (`expr!(…)`, `expr!{…}`, `expr![…]`): denotes macro expansion. See Macros[40].

[18] `loops.html#for`
[19] `variable-bindings.html`
[20] `loops.html#loop`
[21] `match.html`
[22] `crates-and-modules.html#defining-modules`
[23] `closures.html#move-closures`
[24] `mutability.html`
[25] `crates-and-modules.html#exporting-a-public-interface`
[26] `patterns.html#ref-and-ref-mut`
[27] `functions.html#early-returns`
[28] `traits.html`
[29] `method-syntax.html#method-calls`
[30] `const-and-static.html#static`
[31] `structs.html`
[32] `traits.html`
[33] `primitive-types.html#booleans`
[34] `type-aliases.html`
[35] `associated-types.html`
[36] `unsafe.html`
[37] `crates-and-modules.html#importing-modules-with-use`
[38] `traits.html#where-clause`
[39] `loops.html#while`
[40] `macros.html`

- `!` (`!expr`): bitwise or logical complement. Overloadable (`Not`).
- `%` (`expr % expr`): arithmetic remainder. Overloadable (`Rem`).
- `%=` (`var %= expr`): arithmetic remainder & assignment.
- `&` (`expr & expr`): bitwise and. Overloadable (`BitAnd`).
- `&` (`&expr`): borrow. See References and Borrowing[41].
- `&` (`&type`, `&mut type`, `&'a type`, `&'a mut type`): borrowed pointer type. See References and Borrowing[42].
- `&=` (`var &= expr`): bitwise and & assignment.
- `&&` (`expr && expr`): logical and.
- `*` (`expr * expr`): arithmetic multiplication. Overloadable (`Mul`).
- `*` (`*expr`): dereference.
- `*` (`*const type`, `*mut type`): raw pointer. See Raw Pointers[43].
- `*=` (`var *= expr`): arithmetic multiplication & assignment.
- `+` (`expr + expr`): arithmetic addition. Overloadable (`Add`).
- `+` (`trait + trait`, `'a + trait`): compound type constraint. See Traits (Multiple Trait Bounds)[44].
- `+=` (`var += expr`): arithmetic addition & assignment.
- `,`: argument and element separator. See Attributes[45], Functions[46], Structs[47], Generics[48], Match[49], Closures[50], Crates and Modules (Importing Modules with `use`)[51].
- `-` (`expr - expr`): arithmetic subtraction. Overloadable (`Sub`).
- `-` (`- expr`): arithmetic negation. Overloadable (`Neg`).
- `-=` (`var -= expr`): arithmetic subtraction & assignment.
- `->` (`fn(…) -> type`, `|…| -> type`): function and closure return type. See Functions[52], Closures[53].
- `-> !` (`fn(…) -> !`, `|…| -> !`): diverging function or closure. See Diverging Functions[54].
- `.` (`expr.ident`): member access. See Structs[55], Method Syntax[56].
- `..` (`..`, `expr..`, `..expr`, `expr..expr`): right-exclusive range literal.
- `..` (`..expr`): struct literal update syntax. See Structs (Update syntax)[57].
- `..` (`variant(x, ..)`, `struct_type { x, .. }`): "and the rest" pattern binding. See Patterns (Ignoring bindings)[58].
- `...` (`expr ... expr`): inclusive range pattern. See Patterns (Ranges)[59].

[41] `references-and-borrowing.html`
[42] `references-and-borrowing.html`
[43] `raw-pointers.html`
[44] `traits.html#multiple-trait-bounds`
[45] `attributes.html`
[46] `functions.html`
[47] `structs.html`
[48] `generics.html`
[49] `match.html`
[50] `closures.html`
[51] `crates-and-modules.html#importing-modules-with-use`
[52] `functions.html`
[53] `closures.html`
[54] `functions.html#diverging-functions`
[55] `structs.html`
[56] `method-syntax.html`
[57] `structs.html#update-syntax`
[58] `patterns.html#ignoring-bindings`
[59] `patterns.html#ranges`

- `/` (`expr / expr`): arithmetic division. Overloadable (`Div`).
- `/=` (`var /= expr`): arithmetic division & assignment.
- `:` (`pat: type`, `ident: type`): constraints. See Variable Bindings[60], Functions[61], Structs[62], Traits[63].
- `:` (`ident: expr`): struct field initializer. See Structs[64].
- `:` (`'a: loop {…}`): loop label. See Loops (Loops Labels)[65].
- `;`: statement and item terminator.
- `;` (`[…; len]`): part of fixed-size array syntax. See Primitive Types (Arrays)[66].
- `«` (`expr « expr`): left-shift. Overloadable (`Shl`).
- `«=` (`var «= expr`): left-shift & assignment.
- `<` (`expr < expr`): less-than comparison. Overloadable (`Cmp`, `PartialCmp`).
- `<=` (`var <= expr`): less-than or equal-to comparison. Overloadable (`Cmp`, `PartialCmp`).
- `=` (`var = expr`, `ident = type`): assignment/equivalence. See Variable Bindings[67], `type` Aliases[68], generic parameter defaults.
- `==` (`var == expr`): comparison. Overloadable (`Eq`, `PartialEq`).
- `=>` (`pat => expr`): part of match arm syntax. See Match[69].
- `>` (`expr > expr`): greater-than comparison. Overloadable (`Cmp`, `PartialCmp`).
- `>=` (`var >= expr`): greater-than or equal-to comparison. Overloadable (`Cmp`, `PartialCmp`).
- `»` (`expr » expr`): right-shift. Overloadable (`Shr`).
- `»=` (`var »= expr`): right-shift & assignment.
- `@` (`ident @ pat`): pattern binding. See Patterns (Bindings)[70].
- `^` (`expr ^ expr`): bitwise exclusive or. Overloadable (`BitXor`).
- `^=` (`var ^= expr`): bitwise exclusive or & assignment.
- `|` (`expr | expr`): bitwise or. Overloadable (`BitOr`).
- `|` (`pat | pat`): pattern alternatives. See Patterns (Multiple patterns)[71].
- `|=` (`var |= expr`): bitwise or & assignment.
- `||` (`expr || expr`): logical or.
- `_`: "ignored" pattern binding. See Patterns (Ignoring bindings)[72].

7.0.7 Other Syntax

- `'ident`: named lifetime or loop label. See Lifetimes[73], Loops (Loops Labels)[74].
- `…u8`, `…i32`, `…f64`, `…usize`, `…`: numeric literal of specific type.
- `"…"`: string literal. See Strings[75].

[60] `variable-bindings.html`
[61] `functions.html`
[62] `structs.html`
[63] `traits.html`
[64] `structs.html`
[65] `loops.html#loop-labels`
[66] `primitive-types.html#arrays`
[67] `variable-bindings.html`
[68] `type-aliases.html`
[69] `match.html`
[70] `patterns.html#bindings`
[71] `patterns.html#multiple-patterns`
[72] `patterns.html#ignoring-bindings`
[73] `lifetimes.html`
[74] `loops.html#loop-labels`
[75] `strings.html`

- `r"…"`, `r#"…"#`, `r##"…"##`, …: raw string literal, escape characters are not processed. See Reference (Raw String Literals)[76].
- `b"…"`: byte string literal, constructs a `[u8]` instead of a string. See Reference (Byte String Literals)[77].
- `br"…"`, `br#"…"#`, `br##"…"##`, …: raw byte string literal, combination of raw and byte string literal. See Reference (Raw Byte String Literals)[78].
- `'…'`: character literal. See Primitive Types (`char`)[79].
- `b'…'`: ASCII byte literal.

- `ident::ident`: path. See Crates and Modules (Defining Modules)[80].
- `::path`: path relative to the crate root (*i.e.* an explicitly absolute path). See Crates and Modules (Re-exporting with `pub use`)[81].
- `self::path`: path relative to the current module (*i.e.* an explicitly relative path). See Crates and Modules (Re-exporting with `pub use`)[82].
- `super::path`: path relative to the parent of the current module. See Crates and Modules (Re-exporting with `pub use`)[83].
- `type::ident`: associated constants, functions, and types. See Associated Types[84].
- `<type>::…`: associated item for a type which cannot be directly named (*e.g.* `<&T>::…`, `<[T]>::…`, *etc.*). See Associated Types[85].

- `path<…>` (*e.g.* `Vec<u8>`): specifies parameters to generic type *in a type.* See Generics[86].
- `path::<…>`, `method::<…>` (*e.g.* `"42".parse::<i32>()`): specifies parameters to generic type, function, or method *in an expression.*
- `fn ident<…> …`: define generic function. See Generics[87].
- `struct ident<…> …`: define generic structure. See Generics[88].
- `enum ident<…> …`: define generic enumeration. See Generics[89].
- `impl<…> …`: define generic implementation.
- `for<…> type`: higher-ranked lifetime bounds.
- `type<ident=type>` (*e.g.* `Iterator<Item=T>`): a generic type where one or more associated types have specific assignments. See Associated Types[90].

- `T: U`: generic parameter `T` constrained to types that implement `U`. See Traits[91].
- `T: 'a`: generic type `T` must outlive lifetime `'a`.
- `'b: 'a`: generic lifetime `'b` must outlive lifetime `'a`.

[76] ../reference.html#raw-string-literals
[77] ../reference.html#byte-string-literals
[78] ../reference.html#raw-byte-string-literals
[79] primitive-types.html#char
[80] crates-and-modules.html#defining-modules
[81] crates-and-modules.html#re-exporting-with-pub-use
[82] crates-and-modules.html#re-exporting-with-pub-use
[83] crates-and-modules.html#re-exporting-with-pub-use
[84] associated-types.html
[85] associated-types.html
[86] generics.html
[87] generics.html
[88] generics.html
[89] generics.html
[90] associated-types.html
[91] traits.html

- `T: ?Sized`: allow generic type parameter to be a dynamically-sized type. See Unsized Types (`?Sized`)[92].
- `'a + trait`, `trait + trait`: compound type constraint. See Traits (Multiple Trait Bounds)[93].

- `#[meta]`: outer attribute. See Attributes[94].
- `#![meta]`: inner attribute. See Attributes[95].
- `$ident`: macro substitution. See Macros[96].
- `$ident:kind`: macro capture. See Macros[97].
- `$(…)…`: macro repetition. See Macros[98].

- `//`: line comment. See Comments[99].
- `//!`: inner line doc comment. See Comments[100].
- `///`: outer line doc comment. See Comments[101].
- `/*…*/`: block comment. See Comments[102].
- `/*!…*/`: inner block doc comment. See Comments[103].
- `/**…*/`: outer block doc comment. See Comments[104].

- `()`: empty tuple (*a.k.a.* unit), both literal and type.
- `(expr)`: parenthesized expression.
- `(expr,)`: single-element tuple expression. See Primitive Types (Tuples)[105].
- `(type,)`: single-element tuple type. See Primitive Types (Tuples)[106].
- `(expr, …)`: tuple expression. See Primitive Types (Tuples)[107].
- `(type, …)`: tuple type. See Primitive Types (Tuples)[108].
- `expr(expr, …)`: function call expression. Also used to initialize tuple `structs` and tuple `enum` variants. See Functions[109].
- `ident!(…)`, `ident!{…}`, `ident![…]`: macro invocation. See Macros[110].
- `expr.0`, `expr.1`, …: tuple indexing. See Primitive Types (Tuple Indexing)[111].

- `{…}`: block expression.
- `Type {…}`: `struct` literal. See Structs[112].

[92] `unsized-types.html#?sized`
[93] `traits.html#multiple-trait-bounds`
[94] `attributes.html`
[95] `attributes.html`
[96] `macros.html`
[97] `macros.html`
[98] `macros.html`
[99] `comments.html`
[100] `comments.html`
[101] `comments.html`
[102] `comments.html`
[103] `comments.html`
[104] `comments.html`
[105] `primitive-types.html#tuples`
[106] `primitive-types.html#tuples`
[107] `primitive-types.html#tuples`
[108] `primitive-types.html#tuples`
[109] `functions.html`
[110] `macros.html`
[111] `primitive-types.html#tuple-indexing`
[112] `structs.html`

- `[…]`: array literal. See Primitive Types (Arrays)[113].
- `[expr; len]`: array literal containing `len` copies of `expr`. See Primitive Types (Arrays)[114].
- `[type; len]`: array type containing `len` instances of `type`. See Primitive Types (Arrays)[115].

[113] `primitive-types.html#arrays`
[114] `primitive-types.html#arrays`
[115] `primitive-types.html#arrays`

Chapter 8

Bibliography

This is a reading list of material relevant to Rust. It includes prior research that has - at one time or another - influenced the design of Rust, as well as publications about Rust.

Type system

- Region based memory management in Cyclone[1]
- Safe manual memory management in Cyclone[2]
- Typeclasses: making ad-hoc polymorphism less ad hoc[3]
- Macros that work together[4]
- Traits: composable units of behavior[5]
- Alias burying[6] - We tried something similar and abandoned it.
- External uniqueness is unique enough[7]
- Uniqueness and Reference Immutability for Safe Parallelism[8]
- Region Based Memory Management[9]

Concurrency

- Singularity: rethinking the software stack[10]
- Language support for fast and reliable message passing in singularity OS[11]
- Scheduling multithreaded computations by work stealing[12]
- Thread scheduling for multiprogramming multiprocessors[13]

[1] http://209.68.42.137/ucsd-pages/Courses/cse227.w03/handouts/cyclone-regions.pdf
[2] http://www.cs.umd.edu/projects/PL/cyclone/scp.pdf
[3] http://www.ps.uni-sb.de/courses/typen-ws99/class.ps.gz
[4] https://www.cs.utah.edu/plt/publications/jfp12-draft-fcdf.pdf
[5] http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf
[6] http://www.cs.uwm.edu/faculty/boyland/papers/unique-preprint.ps
[7] http://www.cs.uu.nl/research/techreps/UU-CS-2002-048.html
[8] https://research.microsoft.com/pubs/170528/msr-tr-2012-79.pdf
[9] http://www.cs.ucla.edu/~palsberg/tba/papers/tofte-talpin-iandc97.pdf
[10] https://research.microsoft.com/pubs/69431/osr2007_rethinkingsoftwarestack.pdf
[11] https://research.microsoft.com/pubs/67482/singsharp.pdf
[12] http://supertech.csail.mit.edu/papers/steal.pdf
[13] http://www.eecis.udel.edu/%7Ecavazos/cisc879-spring2008/papers/arora98thread.pdf

- The data locality of work stealing[14]
- Dynamic circular work stealing deque[15] - The Chase/Lev deque
- Work-first and help-first scheduling policies for async-finish task parallelism[16] - More general than fully-strict work stealing
- A Java fork/join calamity[17] - critique of Java's fork/join library, particularly its application of work stealing to non-strict computation
- Scheduling techniques for concurrent systems[18]
- Contention aware scheduling[19]
- Balanced work stealing for time-sharing multicores[20]
- Three layer cake for shared-memory programming[21]
- Non-blocking steal-half work queues[22]
- Reagents: expressing and composing fine-grained concurrency[23]
- Algorithms for scalable synchronization of shared-memory multiprocessors[24]
- Epoc-based reclamation[25].

Others

- Crash-only software[26]
- Composing High-Performance Memory Allocators[27]
- Reconsidering Custom Memory Allocation[28]

Papers *about* Rust

- GPU Programming in Rust: Implementing High Level Abstractions in a Systems Level Language[29]. Early GPU work by Eric Holk.
- Parallel closures: a new twist on an old idea[30]
- not exactly about Rust, but by nmatsakis
- Patina: A Formalization of the Rust Programming Language[31]. Early formalization of a subset of the type system, by Eric Reed.
- Experience Report: Developing the Servo Web Browser Engine using Rust[32]. By Lars Bergstrom.

[14] http://www.aladdin.cs.cmu.edu/papers/pdfs/y2000/locality_spaa00.pdf
[15] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.170.1097&rep=rep1&type=pdf
[16] http://www.cs.rice.edu/%7Eyguo/pubs/PID824943.pdf
[17] http://www.coopsoft.com/ar/CalamityArticle.html
[18] http://www.stanford.edu/~ouster/cgi-bin/papers/coscheduling.pdf
[19] http://www.blagodurov.net/files/a8-blagodurov.pdf
[20] http://www.cse.ohio-state.edu/hpcs/WWW/HTML/publications/papers/TR-12-1.pdf
[21] http://dl.acm.org/citation.cfm?id=1953616&dl=ACM&coll=DL&CFID=524387192&CFTOKEN=44362705
[22] http://www.cs.bgu.ac.il/%7Ehendlerd/papers/p280-hendler.pdf
[23] http://www.mpi-sws.org/~turon/reagents.pdf
[24] https://www.cs.rochester.edu/u/scott/papers/1991_TOCS_synch.pdf
[25] https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf
[26] https://www.usenix.org/legacy/events/hotos03/tech/full_papers/candea/candea.pdf
[27] http://people.cs.umass.edu/~emery/pubs/berger-pldi2001.pdf
[28] http://people.cs.umass.edu/~emery/pubs/berger-oopsla2002.pdf
[29] http://www.cs.indiana.edu/~eholk/papers/hips2013.pdf
[30] https://www.usenix.org/conference/hotpar12/parallel-closures-new-twist-old-idea
[31] ftp://ftp.cs.washington.edu/tr/2015/03/UW-CSE-15-03-02.pdf
[32] http://arxiv.org/abs/1505.07383

- Implementing a Generic Radix Trie in Rust[33]. Undergrad paper by Michael Sproul.
- Reenix: Implementing a Unix-Like Operating System in Rust[34]. Undergrad paper by Alex Light.
- [Evaluation of performance and productivity metrics of potential programming languages in the HPC environment] (http://octarineparrot.com/assets/mrfloya-thesis-ba.pdf). Bachelor's thesis by Florian Wilkens. Compares C, Go and Rust.
- Nom, a byte oriented, streaming, zero copy, parser combinators library in Rust[35]. By Geoffroy Couprie, research for VLC.
- Graph-Based Higher-Order Intermediate Representation[36]. An experimental IR implemented in Impala, a Rust-like language.
- Code Refinement of Stencil Codes[37]. Another paper using Impala.
- Parallelization in Rust with fork-join and friends[38]. Linus Farnstrand's master's thesis.
- Session Types for Rust[39]. Philip Munksgaard's master's thesis. Research for Servo.
- Ownership is Theft: Experiences Building an Embedded OS in Rust - Amit Levy, et. al.[40]

[33] https://michaelsproul.github.io/rust_radix_paper/rust-radix-sproul.pdf
[34] http://scialex.github.io/reenix.pdf
[35] http://spw15.langsec.org/papers/couprie-nom.pdf
[36] http://compilers.cs.uni-saarland.de/papers/lkh15_cgo.pdf
[37] http://compilers.cs.uni-saarland.de/papers/ppl14_web.pdf
[38] http://publications.lib.chalmers.se/records/fulltext/219016/219016.pdf
[39] http://munksgaard.me/papers/laumann-munksgaard-larsen.pdf
[40] http://amitlevy.com/papers/tock-plos2015.pdf

Part II

The Rustonomicon; or, the Advanced Rust Programming Language

8.1 NOTE: This is a draft document, and may contain serious errors

> Instead of the programs I had hoped for, there came only a shuddering blackness and ineffable loneliness; and I saw at last a fearful truth which no one had ever dared to breathe before — the unwhisperable secret of secrets — The fact that this language of stone and stridor is not a sentient perpetuation of Rust as London is of Old London and Paris of Old Paris, but that it is in fact quite unsafe, its sprawling body imperfectly embalmed and infested with queer animate things which have nothing to do with it as it was in compilation.

This book digs into all the awful details that are necessary to understand in order to write correct Unsafe Rust programs. Due to the nature of this problem, it may lead to unleashing untold horrors that shatter your psyche into a billion infinitesimal fragments of despair.

Should you wish a long and happy career of writing Rust programs, you should turn back now and forget you ever saw this book. It is not necessary. However if you intend to write unsafe code – or just want to dig into the guts of the language – this book contains invaluable information.

Unlike The Book[41] we will be assuming considerable prior knowledge. In particular, you should be comfortable with basic systems programming and Rust. If you don't feel comfortable with these topics, you should consider reading The Book[42] first. Though we will not be assuming that you have, and will take care to occasionally give a refresher on the basics where appropriate. You can skip straight to this book if you want; just know that we won't be explaining everything from the ground up.

To be clear, this book goes into deep detail. We're going to dig into exception-safety, pointer aliasing, memory models, and even some type-theory. We will also be spending a lot of time talking about the different kinds of safety and guarantees.

[41] ../book/
[42] ../book/

Chapter 9

Meet Safe and Unsafe

Programmers in safe "high-level" languages face a fundamental dilemma. On one hand, it would be *really* great to just say what you want and not worry about how it's done. On the other hand, that can lead to unacceptably poor performance. It may be necessary to drop down to less clear or idiomatic practices to get the performance characteristics you want. Or maybe you just throw up your hands in disgust and decide to shell out to an implementation in a less sugary-wonderful *unsafe* language.

Worse, when you want to talk directly to the operating system, you *have* to talk to an unsafe language: *C*. C is ever-present and unavoidable. It's the lingua-franca of the programming world. Even other safe languages generally expose C interfaces for the world at large! Regardless of why you're doing it, as soon as your program starts talking to C it stops being safe.

With that said, Rust is *totally* a safe programming language.

Well, Rust *has* a safe programming language. Let's step back a bit.

Rust can be thought of as being composed of two programming languages: *Safe Rust* and *Unsafe Rust*. Safe Rust is For Reals Totally Safe. Unsafe Rust, unsurprisingly, is *not* For Reals Totally Safe. In fact, Unsafe Rust lets you do some really crazy unsafe things.

Safe Rust is the *true* Rust programming language. If all you do is write Safe Rust, you will never have to worry about type-safety or memory-safety. You will never endure a null or dangling pointer, or any of that Undefined Behavior nonsense.

That's totally awesome.

The standard library also gives you enough utilities out-of-the-box that you'll be able to write awesome high-performance applications and libraries in pure idiomatic Safe Rust.

But maybe you want to talk to another language. Maybe you're writing a low-level abstraction not exposed by the standard library. Maybe you're *writing* the standard library (which is written entirely in Rust). Maybe you need to do something the type-system doesn't understand and just *frob some dang bits*. Maybe you need Unsafe Rust.

Unsafe Rust is exactly like Safe Rust with all the same rules and semantics. However Unsafe Rust lets you do some *extra* things that are Definitely Not Safe.

The only things that are different in Unsafe Rust are that you can:

- Dereference raw pointers
- Call `unsafe` functions (including C functions, intrinsics, and the raw allocator)
- Implement `unsafe` traits
- Mutate statics

That's it. The reason these operations are relegated to Unsafe is that misusing any of these things will cause the ever dreaded Undefined Behavior. Invoking Undefined Behavior gives the compiler full rights to do arbitrarily bad things to your program. You definitely *should not* invoke Undefined Behavior.

Unlike C, Undefined Behavior is pretty limited in scope in Rust. All the core language cares about is preventing the following things:

- Dereferencing null or dangling pointers
- Reading uninitialized memory[1]
- Breaking the pointer aliasing rules[2]
- Producing invalid primitive values:
  - dangling/null references
  - a `bool` that isn't 0 or 1
  - an undefined `enum` discriminant
  - a `char` outside the ranges [0x0, 0xD7FF] and [0xE000, 0x10FFFF]
  - A non-utf8 `str`
- Unwinding into another language
- Causing a data race[3]

That's it. That's all the causes of Undefined Behavior baked into Rust. Of course, unsafe functions and traits are free to declare arbitrary other constraints that a program must maintain to avoid Undefined Behavior. However, generally violations of these constraints will just transitively lead to one of the above problems. Some additional constraints may also derive from compiler intrinsics that make special assumptions about how code can be optimized.

Rust is otherwise quite permissive with respect to other dubious operations. Rust considers it "safe" to:

- Deadlock
- Have a race condition[4]
- Leak memory
- Fail to call destructors
- Overflow integers
- Abort the program
- Delete the production database

However any program that actually manages to do such a thing is *probably* incorrect. Rust provides lots of tools to make these things rare, but these problems are considered impractical to categorically prevent.

[1] `uninitialized.html`
[2] `references.html`
[3] `races.html`
[4] `races.html`

9.1 How Safe and Unsafe Interact

So what's the relationship between Safe and Unsafe Rust? How do they interact?

Rust models the separation between Safe and Unsafe Rust with the `unsafe` keyword, which can be thought as a sort of *foreign function interface* (FFI) between Safe and Unsafe Rust. This is the magic behind why we can say Safe Rust is a safe language: all the scary unsafe bits are relegated exclusively to FFI *just like every other safe language.*

However because one language is a subset of the other, the two can be cleanly intermixed as long as the boundary between Safe and Unsafe Rust is denoted with the `unsafe` keyword. No need to write headers, initialize runtimes, or any of that other FFI boiler-plate.

There are several places `unsafe` can appear in Rust today, which can largely be grouped into two categories:

- There are unchecked contracts here. To declare you understand this, I require you to write `unsafe` elsewhere:
  - On functions, `unsafe` is declaring the function to be unsafe to call. Users of the function must check the documentation to determine what this means, and then have to write `unsafe` somewhere to identify that they're aware of the danger.
  - On trait declarations, `unsafe` is declaring that *implementing* the trait is an unsafe operation, as it has contracts that other unsafe code is free to trust blindly. (More on this below.)
- I am declaring that I have, to the best of my knowledge, adhered to the unchecked contracts:
  - On trait implementations, `unsafe` is declaring that the contract of the `unsafe` trait has been upheld.
  - On blocks, `unsafe` is declaring any unsafety from an unsafe operation within to be handled, and therefore the parent function is safe.

There is also `#[unsafe_no_drop_flag]`, which is a special case that exists for historical reasons and is in the process of being phased out. See the section on drop flags[5] for details.

Some examples of unsafe functions:

- `slice::get_unchecked` will perform unchecked indexing, allowing memory safety to be freely violated.
- every raw pointer to sized type has intrinsic `offset` method that invokes Undefined Behavior if it is not "in bounds" as defined by LLVM.
- `mem::transmute` reinterprets some value as having the given type, bypassing type safety in arbitrary ways. (see conversions[6] for details)
- All FFI functions are `unsafe` because they can do arbitrary things. C being an obvious culprit, but generally any language can do something that Rust isn't happy about.

[5] `drop-flags.html`
[6] `conversions.html`

As of Rust 1.0 there are exactly two unsafe traits:

- `Send` is a marker trait (it has no actual API) that promises implementors are safe to send (move) to another thread.
- `Sync` is a marker trait that promises that threads can safely share implementors through a shared reference.

The need for unsafe traits boils down to the fundamental property of safe code:

No matter how completely awful Safe code is, it can't cause Undefined Behavior.

This means that Unsafe Rust, **the royal vanguard of Undefined Behavior**, has to be *super paranoid* about generic safe code. To be clear, Unsafe Rust is totally free to trust specific safe code. Anything else would degenerate into infinite spirals of paranoid despair. In particular it's generally regarded as ok to trust the standard library to be correct. `std` is effectively an extension of the language, and you really just have to trust the language. If `std` fails to uphold the guarantees it declares, then it's basically a language bug.

That said, it would be best to minimize *needlessly* relying on properties of concrete safe code. Bugs happen! Of course, I must reinforce that this is only a concern for Unsafe code. Safe code can blindly trust anyone and everyone as far as basic memory-safety is concerned.

On the other hand, safe traits are free to declare arbitrary contracts, but because implementing them is safe, unsafe code can't trust those contracts to actually be upheld. This is different from the concrete case because *anyone* can randomly implement the interface. There is something fundamentally different about trusting a particular piece of code to be correct, and trusting *all the code that will ever be written* to be correct.

For instance Rust has `PartialOrd` and `Ord` traits to try to differentiate between types which can "just" be compared, and those that actually implement a total ordering. Pretty much every API that wants to work with data that can be compared wants Ord data. For instance, a sorted map like BTreeMap *doesn't even make sense* for partially ordered types. If you claim to implement Ord for a type, but don't actually provide a proper total ordering, BTreeMap will get *really confused* and start making a total mess of itself. Data that is inserted may be impossible to find!

But that's okay. BTreeMap is safe, so it guarantees that even if you give it a completely garbage Ord implementation, it will still do something *safe*. You won't start reading uninitialized or unallocated memory. In fact, BTreeMap manages to not actually lose any of your data. When the map is dropped, all the destructors will be successfully called! Hooray!

However BTreeMap is implemented using a modest spoonful of Unsafe Rust (most collections are). That means that it's not necessarily *trivially true* that a bad Ord implementation will make BTreeMap behave safely. BTreeMap must be sure not to rely on Ord *where safety is at stake*. Ord is provided by safe code, and safety is not safe code's responsibility to uphold.

But wouldn't it be grand if there was some way for Unsafe to trust some trait contracts *somewhere*? This is the problem that unsafe traits tackle: by marking *the trait itself* as unsafe to implement, unsafe code can trust the implementation to uphold the

trait's contract. Although the trait implementation may be incorrect in arbitrary other ways.

For instance, given a hypothetical UnsafeOrd trait, this is technically a valid implementation:

```
# use std::cmp::Ordering;
# struct MyType;
# unsafe trait UnsafeOrd { fn cmp(&self, other: &Self) -> Ordering; }
unsafe impl UnsafeOrd for MyType {
    fn cmp(&self, other: &Self) -> Ordering {
        Ordering::Equal
    }
}
```

But it's probably not the implementation you want.

Rust has traditionally avoided making traits unsafe because it makes Unsafe pervasive, which is not desirable. The reason Send and Sync are unsafe is because thread safety is a *fundamental property* that unsafe code cannot possibly hope to defend against in the same way it would defend against a bad Ord implementation. The only way to possibly defend against thread-unsafety would be to *not use threading at all.* Making every load and store atomic isn't even sufficient, because it's possible for complex invariants to exist between disjoint locations in memory. For instance, the pointer and capacity of a Vec must be in sync.

Even concurrent paradigms that are traditionally regarded as Totally Safe like message passing implicitly rely on some notion of thread safety – are you really message-passing if you pass a pointer? Send and Sync therefore require some fundamental level of trust that Safe code can't provide, so they must be unsafe to implement. To help obviate the pervasive unsafety that this would introduce, Send (resp. Sync) is automatically derived for all types composed only of Send (resp. Sync) values. 99% of types are Send and Sync, and 99% of those never actually say it (the remaining 1% is overwhelmingly synchronization primitives).

9.2 Working with Unsafe

Rust generally only gives us the tools to talk about Unsafe Rust in a scoped and binary manner. Unfortunately, reality is significantly more complicated than that. For instance, consider the following toy function:

```
fn index(idx: usize, arr: &[u8]) -> Option<u8> {
    if idx < arr.len() {
        unsafe {
            Some(*arr.get_unchecked(idx))
        }
    } else {
        None
    }
}
```

Clearly, this function is safe. We check that the index is in bounds, and if it is, index into the array in an unchecked manner. But even in such a trivial function, the scope of the unsafe block is questionable. Consider changing the `<` to a `<=`:

```
fn index(idx: usize, arr: &[u8]) -> Option<u8> {
    if idx <= arr.len() {
        unsafe {
            Some(*arr.get_unchecked(idx))
        }
    } else {
        None
    }
}
```

This program is now unsound, and yet *we only modified safe code.* This is the fundamental problem of safety: it's non-local. The soundness of our unsafe operations necessarily depends on the state established by otherwise "safe" operations.

Safety is modular in the sense that opting into unsafety doesn't require you to consider arbitrary other kinds of badness. For instance, doing an unchecked index into a slice doesn't mean you suddenly need to worry about the slice being null or containing uninitialized memory. Nothing fundamentally changes. However safety *isn't* modular in the sense that programs are inherently stateful and your unsafe operations may depend on arbitrary other state.

Trickier than that is when we get into actual statefulness. Consider a simple implementation of `Vec`:

```
use std::ptr;

// Note this definition is insufficient. See the section on implementing Vec.
pub struct Vec<T> {
    ptr: *mut T,
    len: usize,
    cap: usize,
}

// Note this implementation does not correctly handle zero-sized types.
// We currently live in a nice imaginary world of only positive fixed-size
// types.
impl<T> Vec<T> {
    pub fn push(&mut self, elem: T) {
        if self.len == self.cap {
            // not important for this example
            self.reallocate();
        }
        unsafe {
            ptr::write(self.ptr.offset(self.len as isize), elem);
            self.len += 1;
        }
```

```
    }

    # fn reallocate(&mut self) { }
}

# fn main() {}
```

This code is simple enough to reasonably audit and verify. Now consider adding the following method:

```
fn make_room(&mut self) {
    // grow the capacity
    self.cap += 1;
}
```

This code is 100% Safe Rust but it is also completely unsound. Changing the capacity violates the invariants of Vec (that `cap` reflects the allocated space in the Vec). This is not something the rest of Vec can guard against. It *has* to trust the capacity field because there's no way to verify it.

`unsafe` does more than pollute a whole function: it pollutes a whole *module*. Generally, the only bullet-proof way to limit the scope of unsafe code is at the module boundary with privacy.

However this works *perfectly*. The existence of `make_room` is *not* a problem for the soundness of Vec because we didn't mark it as public. Only the module that defines this function can call it. Also, `make_room` directly accesses the private fields of Vec, so it can only be written in the same module as Vec.

It is therefore possible for us to write a completely safe abstraction that relies on complex invariants. This is *critical* to the relationship between Safe Rust and Unsafe Rust. We have already seen that Unsafe code must trust *some* Safe code, but can't trust *generic* Safe code. It can't trust an arbitrary implementor of a trait or any function that was passed to it to be well-behaved in a way that safe code doesn't care about.

However if unsafe code couldn't prevent client safe code from messing with its state in arbitrary ways, safety would be a lost cause. Thankfully, it *can* prevent arbitrary code from messing with critical state due to privacy.

Safety lives!

Chapter 10

Data Layout

Low-level programming cares a lot about data layout. It's a big deal. It also pervasively influences the rest of the language, so we're going to start by digging into how data is represented in Rust.

10.1 repr(Rust)

First and foremost, all types have an alignment specified in bytes. The alignment of a type specifies what addresses are valid to store the value at. A value of alignment `n` must only be stored at an address that is a multiple of `n`. So alignment 2 means you must be stored at an even address, and 1 means that you can be stored anywhere. Alignment is at least 1, and always a power of 2. Most primitives are generally aligned to their size, although this is platform-specific behavior. In particular, on x86 `u64` and `f64` may be only aligned to 32 bits.

A type's size must always be a multiple of its alignment. This ensures that an array of that type may always be indexed by offsetting by a multiple of its size. Note that the size and alignment of a type may not be known statically in the case of dynamically sized types[1].

Rust gives you the following ways to lay out composite data:

- structs (named product types)
- tuples (anonymous product types)
- arrays (homogeneous product types)
- enums (named sum types – tagged unions)

An enum is said to be *C-like* if none of its variants have associated data.

Composite structures will have an alignment equal to the maximum of their fields' alignment. Rust will consequently insert padding where necessary to ensure that all fields are properly aligned and that the overall type's size is a multiple of its alignment. For instance:

[1] `exotic-sizes.html#dynamically-sized-types-dsts`

```
struct A {
    a: u8,
    b: u32,
    c: u16,
}
```

will be 32-bit aligned on an architecture that aligns these primitives to their respective sizes. The whole struct will therefore have a size that is a multiple of 32-bits. It will potentially become:

```
struct A {
    a: u8,
    _pad1: [u8; 3], // to align ‘b‘
    b: u32,
    c: u16,
    _pad2: [u8; 2], // to make overall size multiple of 4
}
```

There is *no indirection* for these types; all data is stored within the struct, as you would expect in C. However with the exception of arrays (which are densely packed and in-order), the layout of data is not by default specified in Rust. Given the two following struct definitions:

```
struct A {
    a: i32,
    b: u64,
}

struct B {
    a: i32,
    b: u64,
}
```

Rust *does* guarantee that two instances of A have their data laid out in exactly the same way. However Rust *does not* currently guarantee that an instance of A has the same field ordering or padding as an instance of B, though in practice there's no reason why they wouldn't.

With A and B as written, this point would seem to be pedantic, but several other features of Rust make it desirable for the language to play with data layout in complex ways.

For instance, consider this struct:

```
struct Foo<T, U> {
    count: u16,
    data1: T,
    data2: U,
}
```

Now consider the monomorphizations of `Foo<u32, u16>` and `Foo<u16, u32>`. If Rust lays out the fields in the order specified, we expect it to pad the values in the struct to satisfy their alignment requirements. So if Rust didn't reorder fields, we would expect it to produce the following:

```
struct Foo<u16, u32> {
    count: u16,
    data1: u16,
    data2: u32,
}

struct Foo<u32, u16> {
    count: u16,
    _pad1: u16,
    data1: u32,
    data2: u16,
    _pad2: u16,
}
```

The latter case quite simply wastes space. An optimal use of space therefore requires different monomorphizations to have *different field orderings*.

Note: this is a hypothetical optimization that is not yet implemented in Rust 1.0

Enums make this consideration even more complicated. Naively, an enum such as:

```
enum Foo {
    A(u32),
    B(u64),
    C(u8),
}
```

would be laid out as:

```
struct FooRepr {
    data: u64, // this is either a u64, u32, or u8 based on `tag`
    tag: u8,   // 0 = A, 1 = B, 2 = C
}
```

And indeed this is approximately how it would be laid out in general (modulo the size and position of `tag`).

However there are several cases where such a representation is inefficient. The classic case of this is Rust's "null pointer optimization": an enum consisting of a single outer unit variant (e.g. `None`) and a (potentially nested) non- nullable pointer variant (e.g. `&T`) makes the tag unnecessary, because a null pointer value can safely be interpreted to mean that the unit variant is chosen instead. The net result is that, for example, `size_of::<Option<&T>>() == size_of::<&T>()`.

There are many types in Rust that are, or contain, non-nullable pointers such as `Box<T>`, `Vec<T>`, `String`, `&T`, and `&mut T`. Similarly, one can imagine nested enums pooling their tags into a single discriminant, as they are by definition known to have a

limited range of valid values. In principle enums could use fairly elaborate algorithms to cache bits throughout nested types with special constrained representations. As such it is *especially* desirable that we leave enum layout unspecified today.

10.2 Exotically Sized Types

Most of the time, we think in terms of types with a fixed, positive size. This is not always the case, however.

10.2.1 Dynamically Sized Types (DSTs)

Rust in fact supports Dynamically Sized Types (DSTs): types without a statically known size or alignment. On the surface, this is a bit nonsensical: Rust *must* know the size and alignment of something in order to correctly work with it! In this regard, DSTs are not normal types. Due to their lack of a statically known size, these types can only exist behind some kind of pointer. Any pointer to a DST consequently becomes a *fat* pointer consisting of the pointer and the information that "completes" them (more on this below).

There are two major DSTs exposed by the language: trait objects, and slices.

A trait object represents some type that implements the traits it specifies. The exact original type is *erased* in favor of runtime reflection with a vtable containing all the information necessary to use the type. This is the information that completes a trait object: a pointer to its vtable.

A slice is simply a view into some contiguous storage – typically an array or `Vec`. The information that completes a slice is just the number of elements it points to.

Structs can actually store a single DST directly as their last field, but this makes them a DST as well:

```
// Can't be stored on the stack directly
struct Foo {
    info: u32,
    data: [u8],
}
```

NOTE: As of Rust 1.0 struct DSTs are broken if the last field has a variable position based on its alignment[2].

10.2.2 Zero Sized Types (ZSTs)

Rust actually allows types to be specified that occupy no space:

```
struct Foo; // No fields = no size

// All fields have no size = no size
```

[2] https://github.com/rust-lang/rust/issues/26403

```
struct Baz {
    foo: Foo,
    qux: (),      // empty tuple has no size
    baz: [u8; 0], // empty array has no size
}
```

On their own, Zero Sized Types (ZSTs) are, for obvious reasons, pretty useless. However as with many curious layout choices in Rust, their potential is realized in a generic context: Rust largely understands that any operation that produces or stores a ZST can be reduced to a no-op. First off, storing it doesn't even make sense – it doesn't occupy any space. Also there's only one value of that type, so anything that loads it can just produce it from the aether – which is also a no-op since it doesn't occupy any space.

One of the most extreme example's of this is Sets and Maps. Given a `Map<Key, Value>`, it is common to implement a `Set<Key>` as just a thin wrapper around `Map<Key, UselessJunk>`. In many languages, this would necessitate allocating space for UselessJunk and doing work to store and load UselessJunk only to discard it. Proving this unnecessary would be a difficult analysis for the compiler.

However in Rust, we can just say that `Set<Key> = Map<Key, ()>`. Now Rust statically knows that every load and store is useless, and no allocation has any size. The result is that the monomorphized code is basically a custom implementation of a HashSet with none of the overhead that HashMap would have to support values.

Safe code need not worry about ZSTs, but *unsafe* code must be careful about the consequence of types with no size. In particular, pointer offsets are no-ops, and standard allocators (including jemalloc, the one used by default in Rust) may return `nullptr` when a zero-sized allocation is requested, which is indistinguishable from out of memory.

10.2.3 Empty Types

Rust also enables types to be declared that *cannot even be instantiated.* These types can only be talked about at the type level, and never at the value level. Empty types can be declared by specifying an enum with no variants:

```
enum Void {} // No variants = EMPTY
```

Empty types are even more marginal than ZSTs. The primary motivating example for Void types is type-level unreachability. For instance, suppose an API needs to return a Result in general, but a specific case actually is infallible. It's actually possible to communicate this at the type level by returning a `Result<T, Void>`. Consumers of the API can confidently unwrap such a Result knowing that it's *statically impossible* for this value to be an `Err`, as this would require providing a value of type `Void`.

In principle, Rust can do some interesting analyses and optimizations based on this fact. For instance, `Result<T, Void>` could be represented as just `T`, because the `Err` case doesn't actually exist. The following *could* also compile:

```
enum Void {}
```

```
let res: Result<u32, Void> = Ok(0);

// Err doesn't exist anymore, so Ok is actually irrefutable.
let Ok(num) = res;
```

But neither of these tricks work today, so all Void types get you is the ability to be confident that certain situations are statically impossible.

One final subtle detail about empty types is that raw pointers to them are actually valid to construct, but dereferencing them is Undefined Behavior because that doesn't actually make sense. That is, you could model C's `void *` type with `*const Void`, but this doesn't necessarily gain anything over using e.g. `*const ()`, which *is* safe to randomly dereference.

10.3 Other reprs

Rust allows you to specify alternative data layout strategies from the default.

10.3.1 repr(C)

This is the most important `repr`. It has fairly simple intent: do what C does. The order, size, and alignment of fields is exactly what you would expect from C or C++. Any type you expect to pass through an FFI boundary should have `repr(C)`, as C is the lingua-franca of the programming world. This is also necessary to soundly do more elaborate tricks with data layout such as reinterpreting values as a different type.

However, the interaction with Rust's more exotic data layout features must be kept in mind. Due to its dual purpose as "for FFI" and "for layout control", `repr(C)` can be applied to types that will be nonsensical or problematic if passed through the FFI boundary.

- ZSTs are still zero-sized, even though this is not a standard behavior in C, and is explicitly contrary to the behavior of an empty type in C++, which still consumes a byte of space.
- DSTs, tuples, and tagged unions are not a concept in C and as such are never FFI safe.
- Tuple structs are like structs with regards to `repr(C)`, as the only difference from a struct is that the fields aren't named.
- **If the type would have any drop flags[3], they will still be added**
- This is equivalent to one of `repr(u*)` (see the next section) for enums. The chosen size is the default enum size for the target platform's C ABI. Note that enum representation in C is implementation defined, so this is really a "best guess". In particular, this may be incorrect when the C code of interest is compiled with certain flags.

[3] `drop-flags.html`

10.3.2 repr(u8), repr(u16), repr(u32), repr(u64)

These specify the size to make a C-like enum. If the discriminant overflows the integer it has to fit in, it will produce a compile-time error. You can manually ask Rust to allow this by setting the overflowing element to explicitly be 0. However Rust will not allow you to create an enum where two variants have the same discriminant.

On non-C-like enums, this will inhibit certain optimizations like the null- pointer optimization.

These reprs have no effect on a struct.

10.3.3 repr(packed)

`repr(packed)` forces rust to strip any padding, and only align the type to a byte. This may improve the memory footprint, but will likely have other negative side-effects.

In particular, most architectures *strongly* prefer values to be aligned. This may mean the unaligned loads are penalized (x86), or even fault (some ARM chips). For simple cases like directly loading or storing a packed field, the compiler might be able to paper over alignment issues with shifts and masks. However if you take a reference to a packed field, it's unlikely that the compiler will be able to emit code to avoid an unaligned load.

As of Rust 1.0 this can cause undefined behavior.[4]

`repr(packed)` is not to be used lightly. Unless you have extreme requirements, this should not be used.

This repr is a modifier on `repr(C)` and `repr(rust)`.

[4] `https://github.com/rust-lang/rust/issues/27060`

Chapter 11

Ownership

Ownership is the breakout feature of Rust. It allows Rust to be completely memory-safe and efficient, while avoiding garbage collection. Before getting into the ownership system in detail, we will consider the motivation of this design.

We will assume that you accept that garbage collection (GC) is not always an optimal solution, and that it is desirable to manually manage memory in some contexts. If you do not accept this, might I interest you in a different language?

Regardless of your feelings on GC, it is pretty clearly a *massive* boon to making code safe. You never have to worry about things going away *too soon* (although whether you still wanted to be pointing at that thing is a different issue...). This is a pervasive problem that C and C++ programs need to deal with. Consider this simple mistake that all of us who have used a non-GC'd language have made at one point:

```
fn as_str(data: &u32) -> &str {
    // compute the string
    let s = format!("{}", data);

    // OH NO! We returned a reference to something that
    // exists only in this function!
    // Dangling pointer! Use after free! Alas!
    // (this does not compile in Rust)
    &s
}
```

This is exactly what Rust's ownership system was built to solve. Rust knows the scope in which the `&s` lives, and as such can prevent it from escaping. However this is a simple case that even a C compiler could plausibly catch. Things get more complicated as code gets bigger and pointers get fed through various functions. Eventually, a C compiler will fall down and won't be able to perform sufficient escape analysis to prove your code unsound. It will consequently be forced to accept your program on the assumption that it is correct.

This will never happen to Rust. It's up to the programmer to prove to the compiler that everything is sound.

Of course, Rust's story around ownership is much more complicated than just verifying that references don't escape the scope of their referent. That's because ensuring pointers are always valid is much more complicated than this. For instance in this code,

```
let mut data = vec![1, 2, 3];
// get an internal reference
let x = &data[0];

// OH NO! ‘push‘ causes the backing storage of ‘data‘ to be reallocated.
// Dangling pointer! User after free! Alas!
// (this does not compile in Rust)
data.push(4);

println!("{}", x);
```

naive scope analysis would be insufficient to prevent this bug, because `data` does in fact live as long as we needed. However it was *changed* while we had a reference into it. This is why Rust requires any references to freeze the referent and its owners.

11.1 References

This section gives a high-level view of the memory model that *all* Rust programs must satisfy to be correct. Safe code is statically verified to obey this model by the borrow checker. Unsafe code may go above and beyond the borrow checker while still satisfying this model. The borrow checker may also be extended to allow more programs to compile, as long as this more fundamental model is satisfied.

There are two kinds of reference:

- Shared reference: `&`
- Mutable reference: `&mut`

Which obey the following rules:

- A reference cannot outlive its referent
- A mutable reference cannot be aliased

That's it. That's the whole model. Of course, we should probably define what *aliased* means. To define aliasing, we must define the notion of *paths* and *liveness*.

NOTE: The model that follows is generally agreed to be dubious and have issues. It's ok-ish as an intuitive model, but fails to capture the desired semantics. We leave this here to be able to use notions introduced here in later sections. This will be significantly changed in the future. TODO: do that.

11.1.1 Paths

If all Rust had were values (no pointers), then every value would be uniquely owned by a variable or composite structure. From this we naturally derive a *tree* of ownership. The stack itself is the root of the tree, with every variable as its direct children. Each variable's direct children would be their fields (if any), and so on.

From this view, every value in Rust has a unique *path* in the tree of ownership. Of particular interest are *ancestors* and *descendants*: if `x` owns `y`, then `x` is an ancestor of `y`, and `y` is a descendant of `x`. Note that this is an inclusive relationship: `x` is a descendant and ancestor of itself.

We can then define references as simply *names* for paths. When you create a reference, you're declaring that an ownership path exists to this address of memory.

Tragically, plenty of data doesn't reside on the stack, and we must also accommodate this. Globals and thread-locals are simple enough to model as residing at the bottom of the stack (though we must be careful with mutable globals). Data on the heap poses a different problem.

If all Rust had on the heap was data uniquely owned by a pointer on the stack, then we could just treat such a pointer as a struct that owns the value on the heap. Box, Vec, String, and HashMap, are examples of types which uniquely own data on the heap.

Unfortunately, data on the heap is not *always* uniquely owned. Rc for instance introduces a notion of *shared* ownership. Shared ownership of a value means there is no unique path to it. A value with no unique path limits what we can do with it.

In general, only shared references can be created to non-unique paths. However mechanisms which ensure mutual exclusion may establish One True Owner temporarily, establishing a unique path to that value (and therefore all its children). If this is done, the value may be mutated. In particular, a mutable reference can be taken.

The most common way to establish such a path is through *interior mutability*, in contrast to the *inherited mutability* that everything in Rust normally uses. Cell, RefCell, Mutex, and RWLock are all examples of interior mutability types. These types provide exclusive access through runtime restrictions.

An interesting case of this effect is Rc itself: if an Rc has refcount 1, then it is safe to mutate or even move its internals. Note however that the refcount itself uses interior mutability.

In order to correctly communicate to the type system that a variable or field of a struct can have interior mutability, it must be wrapped in an UnsafeCell. This does not in itself make it safe to perform interior mutability operations on that value. You still must yourself ensure that mutual exclusion is upheld.

11.1.2 Liveness

Note: Liveness is not the same thing as a *lifetime*, which will be explained in detail in the next section of this chapter.

Roughly, a reference is *live* at some point in a program if it can be dereferenced. Shared references are always live unless they are literally unreachable (for instance,

they reside in freed or leaked memory). Mutable references can be reachable but *not* live through the process of *reborrowing*.

A mutable reference can be reborrowed to either a shared or mutable reference to one of its descendants. A reborrowed reference will only be live again once all reborrows derived from it expire. For instance, a mutable reference can be reborrowed to point to a field of its referent:

```
let x = &mut (1, 2);
{
    // reborrow x to a subfield
    let y = &mut x.0;
    // y is now live, but x isn't
    *y = 3;
}
// y goes out of scope, so x is live again
*x = (5, 7);
```

It is also possible to reborrow into *multiple* mutable references, as long as they are *disjoint*: no reference is an ancestor of another. Rust explicitly enables this to be done with disjoint struct fields, because disjointness can be statically proven:

```
let x = &mut (1, 2);
{
    // reborrow x to two disjoint subfields
    let y = &mut x.0;
    let z = &mut x.1;

    // y and z are now live, but x isn't
    *y = 3;
    *z = 4;
}
// y and z go out of scope, so x is live again
*x = (5, 7);
```

However it's often the case that Rust isn't sufficiently smart to prove that multiple borrows are disjoint. *This does not mean it is fundamentally illegal to make such a borrow,* just that Rust isn't as smart as you want.

To simplify things, we can model variables as a fake type of reference: *owned* references. Owned references have much the same semantics as mutable references: they can be re-borrowed in a mutable or shared manner, which makes them no longer live. Live owned references have the unique property that they can be moved out of (though mutable references *can* be swapped out of). This power is only given to *live* owned references because moving its referent would of course invalidate all outstanding references prematurely.

As a local lint against inappropriate mutation, only variables that are marked as `mut` can be borrowed mutably.

It is interesting to note that Box behaves exactly like an owned reference. It can be moved out of, and Rust understands it sufficiently to reason about its paths like a normal variable.

11.1.3 Aliasing

With liveness and paths defined, we can now properly define *aliasing*:

A mutable reference is aliased if there exists another live reference to one of its ancestors or descendants.

(If you prefer, you may also say the two live references alias *each other*. This has no semantic consequences, but is probably a more useful notion when verifying the soundness of a construct.)

That's it. Super simple right? Except for the fact that it took us two pages to define all of the terms in that definition. You know: Super. Simple.

Actually it's a bit more complicated than that. In addition to references, Rust has *raw pointers*: `*const T` and `*mut T`. Raw pointers have no inherent ownership or aliasing semantics. As a result, Rust makes absolutely no effort to track that they are used correctly, and they are wildly unsafe.

It is an open question to what degree raw pointers have alias semantics. However it is important for these definitions to be sound that the existence of a raw pointer does not imply some kind of live path.

11.2 Lifetimes

Rust enforces these rules through *lifetimes*. Lifetimes are effectively just names for scopes somewhere in the program. Each reference, and anything that contains a reference, is tagged with a lifetime specifying the scope it's valid for.

Within a function body, Rust generally doesn't let you explicitly name the lifetimes involved. This is because it's generally not really necessary to talk about lifetimes in a local context; Rust has all the information and can work out everything as optimally as possible. Many anonymous scopes and temporaries that you would otherwise have to write are often introduced to make your code Just Work.

However once you cross the function boundary, you need to start talking about lifetimes. Lifetimes are denoted with an apostrophe: `'a`, `'static`. To dip our toes with lifetimes, we're going to pretend that we're actually allowed to label scopes with lifetimes, and desugar the examples from the start of this chapter.

Originally, our examples made use of *aggressive* sugar – high fructose corn syrup even – around scopes and lifetimes, because writing everything out explicitly is *extremely noisy*. All Rust code relies on aggressive inference and elision of "obvious" things.

One particularly interesting piece of sugar is that each `let` statement implicitly introduces a scope. For the most part, this doesn't really matter. However it does matter for variables that refer to each other. As a simple example, let's completely desugar this simple piece of Rust code:

```
let x = 0;
let y = &x;
let z = &y;
```

The borrow checker always tries to minimize the extent of a lifetime, so it will likely desugar to the following:

```
// NOTE: ‘’a: {‘ and ‘&’b x‘ is not valid syntax!
’a: {
    let x: i32 = 0;
    ’b: {
        // lifetime used is ’b because that’s good enough.
        let y: &’b i32 = &’b x;
        ’c: {
            // ditto on ’c
            let z: &’c &’b i32 = &’c y;
        }
    }
}
```

Wow. That’s... awful. Let’s all take a moment to thank Rust for making this easier. Actually passing references to outer scopes will cause Rust to infer a larger lifetime:

```
let x = 0;
let z;
let y = &x;
z = y;

’a: {
    let x: i32 = 0;
    ’b: {
        let z: &’b i32;
        ’c: {
            // Must use ’b here because this reference is
            // being passed to that scope.
            let y: &’b i32 = &’b x;
            z = y;
        }
    }
}
```

11.2.1 Example: references that outlive referents

Alright, let’s look at some of those examples from before:

```
fn as_str(data: &u32) -> &str {
    let s = format!(”{}”, data);
    &s
}
```

desugars to:

```
fn as_str<’a>(data: &’a u32) -> &’a str {
    ’b: {
        let s = format!(”{}”, data);
```

```
        return &'a s;
    }
}
```

This signature of `as_str` takes a reference to a u32 with *some* lifetime, and promises that it can produce a reference to a str that can live *just as long*. Already we can see why this signature might be trouble. That basically implies that we're going to find a str somewhere in the scope the reference to the u32 originated in, or somewhere *even earlier*. That's a bit of a big ask.

We then proceed to compute the string `s`, and return a reference to it. Since the contract of our function says the reference must outlive `'a`, that's the lifetime we infer for the reference. Unfortunately, `s` was defined in the scope `'b`, so the only way this is sound if `'b` contains `'a` – which is clearly false since `'a` must contain the function call itself. We have therefore created a reference whose lifetime outlives its referent, which is *literally* the first thing we said that references can't do. The compiler rightfully blows up in our face.

To make this more clear, we can expand the example:

```
fn as_str<'a>(data: &'a u32) -> &'a str {
    'b: {
        let s = format!("{}", data);
        return &'a s
    }
}

fn main() {
    'c: {
        let x: u32 = 0;
        'd: {
            // An anonymous scope is introduced because the borrow does not
            // need to last for the whole scope x is valid for. The return
            // of as_str must find a str somewhere before this function
            // call. Obviously not happening.
            println!("{}", as_str::<'d>(&'d x));
        }
    }
}
```

Shoot!

Of course, the right way to write this function is as follows:

```
fn to_string(data: &u32) -> String {
    format!("{}", data)
}
```

We must produce an owned value inside the function to return it! The only way we could have returned an `&'a str` would have been if it was in a field of the `&'a u32`, which is obviously not the case.

(Actually we could have also just returned a string literal, which as a global can be considered to reside at the bottom of the stack; though this limits our implementation *just a bit*.)

11.2.2 Example: aliasing a mutable reference

How about the other example:

```
let mut data = vec![1, 2, 3];
let x = &data[0];
data.push(4);
println!("{}", x);

'a: {
    let mut data: Vec<i32> = vec![1, 2, 3];
    'b: {
        // 'b is as big as we need this borrow to be
        // (just need to get to `println;`)
        let x: &'b i32 = Index::index::<'b>(&'b data, 0);
        'c: {
            // Temporary scope because we don't need the
            // &mut to last any longer.
            Vec::push(&'c mut data, 4);
        }
        println!("{}", x);
    }
}
```

The problem here is a bit more subtle and interesting. We want Rust to reject this program for the following reason: We have a live shared reference `x` to a descendant of `data` when we try to take a mutable reference to `data` to `push`. This would create an aliased mutable reference, which would violate the *second* rule of references.

However this is *not at all* how Rust reasons that this program is bad. Rust doesn't understand that `x` is a reference to a subpath of `data`. It doesn't understand Vec at all. What it *does* see is that `x` has to live for `'b` to be printed. The signature of `Index::index` subsequently demands that the reference we take to `data` has to survive for `'b`. When we try to call `push`, it then sees us try to make an `&'c mut data`. Rust knows that `'c` is contained within `'b`, and rejects our program because the `&'b data` must still be live!

Here we see that the lifetime system is much more coarse than the reference semantics we're actually interested in preserving. For the most part, *that's totally ok*, because it keeps us from spending all day explaining our program to the compiler. However it does mean that several programs that are totally correct with respect to Rust's *true* semantics are rejected because lifetimes are too dumb.

11.3 Limits of Lifetimes

Given the following code:

```
struct Foo;

impl Foo {
    fn mutate_and_share(&mut self) -> &Self { &*self }
    fn share(&self) {}
}

fn main() {
    let mut foo = Foo;
    let loan = foo.mutate_and_share();
    foo.share();
}
```

One might expect it to compile. We call `mutate_and_share`, which mutably borrows `foo` temporarily, but then returns only a shared reference. Therefore we would expect `foo.share()` to succeed as `foo` shouldn't be mutably borrowed.

However when we try to compile it:

```
<anon>:11:5: 11:8 error: cannot borrow ‘foo‘ as immutable because it is also borrowed as mutab
<anon>:11     foo.share();
              ^~~
<anon>:10:16: 10:19 note: previous borrow of ‘foo‘ occurs here; the mutable borrow prevents sub
<anon>:10     let loan = foo.mutate_and_share();
                         ^~~
<anon>:12:2: 12:2 note: previous borrow ends here
<anon>:8 fn main() {
<anon>:9     let mut foo = Foo;
<anon>:10     let loan = foo.mutate_and_share();
<anon>:11     foo.share();
<anon>:12 }
          ^
```

What happened? Well, we got the exact same reasoning as we did for Example 2 in the previous section[1]. We desugar the program and we get the following:

```
struct Foo;

impl Foo {
    fn mutate_and_share<'a>(&'a mut self) -> &'a Self { &'a *self }
    fn share<'a>(&'a self) {}
}

fn main() {
    'b: {
        let mut foo: Foo = Foo;
        'c: {
            let loan: &'c Foo = Foo::mutate_and_share::<'c>(&'c mut foo);
```

[1] `lifetimes.html#example-aliasing-a-mutable-reference`

```
            'd: {
                Foo::share::<'d>(&'d foo);
            }
        }
    }
}
```

The lifetime system is forced to extend the `&mut foo` to have lifetime `'c`, due to the lifetime of `loan` and mutate_and_share's signature. Then when we try to call `share`, and it sees we're trying to alias that `&'c mut foo` and blows up in our face!

This program is clearly correct according to the reference semantics we actually care about, but the lifetime system is too coarse-grained to handle that.

TODO: other common problems? SEME regions stuff, mostly?

11.4 Lifetime Elision

In order to make common patterns more ergonomic, Rust allows lifetimes to be *elided* in function signatures.

A *lifetime position* is anywhere you can write a lifetime in a type:

```
&'a T
&'a mut T
T<'a>
```

Lifetime positions can appear as either "input" or "output":

- For `fn` definitions, input refers to the types of the formal arguments in the `fn` definition, while output refers to result types. So `fn foo(s: &str) -> (&str, &str)` has elided one lifetime in input position and two lifetimes in output position. Note that the input positions of a `fn` method definition do not include the lifetimes that occur in the method's `impl` header (nor lifetimes that occur in the trait header, for a default method).
- In the future, it should be possible to elide `impl` headers in the same manner.

Elision rules are as follows:

- Each elided lifetime in input position becomes a distinct lifetime parameter.
- If there is exactly one input lifetime position (elided or not), that lifetime is assigned to *all* elided output lifetimes.
- If there are multiple input lifetime positions, but one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to *all* elided output lifetimes.
- Otherwise, it is an error to elide an output lifetime.

Examples:

```
fn print(s: &str);                                    // elided
fn print<'a>(s: &'a str);                             // expanded

fn debug(lvl: uint, s: &str);                         // elided
fn debug<'a>(lvl: uint, s: &'a str);                  // expanded

fn substr(s: &str, until: uint) -> &str;              // elided
fn substr<'a>(s: &'a str, until: uint) -> &'a str;    // expanded

fn get_str() -> &str;                                 // ILLEGAL

fn frob(s: &str, t: &str) -> &str;                    // ILLEGAL

fn get_mut(&mut self) -> &mut T;                      // elided
fn get_mut<'a>(&'a mut self) -> &'a mut T;            // expanded

fn args<T:ToCStr>(&mut self, args: &[T]) -> &mut Command                  // elided
fn args<'a, 'b, T:ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command // expanded

fn new(buf: &mut [u8]) -> BufWriter;                  // elided
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a>        // expanded
```

11.5 Unbounded Lifetimes

Unsafe code can often end up producing references or lifetimes out of thin air. Such lifetimes come into the world as *unbounded*. The most common source of this is dereferencing a raw pointer, which produces a reference with an unbounded lifetime. Such a lifetime becomes as big as context demands. This is in fact more powerful than simply becoming `'static`, because for instance `&'static &'a T` will fail to typecheck, but the unbound lifetime will perfectly mold into `&'a &'a T` as needed. However for most intents and purposes, such an unbounded lifetime can be regarded as `'static`.

Almost no reference is `'static`, so this is probably wrong. `transmute` and `transmute_copy` are the two other primary offenders. One should endeavor to bound an unbounded lifetime as quick as possible, especially across function boundaries.

Given a function, any output lifetimes that don't derive from inputs are unbounded. For instance:

```
fn get_str<'a>() -> &'a str;
```

will produce an `&str` with an unbounded lifetime. The easiest way to avoid unbounded lifetimes is to use lifetime elision at the function boundary. If an output lifetime is elided, then it *must* be bounded by an input lifetime. Of course it might be bounded by the *wrong* lifetime, but this will usually just cause a compiler error, rather than allow memory safety to be trivially violated.

Within a function, bounding lifetimes is more error-prone. The safest and easiest way to bound a lifetime is to return it from a function with a bound lifetime. However if

this is unacceptable, the reference can be placed in a location with a specific lifetime. Unfortunately it’s impossible to name all lifetimes involved in a function.

11.6 Higher-Rank Trait Bounds

Rust’s `Fn` traits are a little bit magic. For instance, we can write the following code:

```
struct Closure<F> {
    data: (u8, u16),
    func: F,
}

impl<F> Closure<F>
    where F: Fn(&(u8, u16)) -> &u8,
{
    fn call(&self) -> &u8 {
        (self.func)(&self.data)
    }
}

fn do_it(data: &(u8, u16)) -> &u8 { &data.0 }

fn main() {
    let clo = Closure { data: (0, 1), func: do_it };
    println!("{}", clo.call());
}
```

If we try to naively desugar this code in the same way that we did in the lifetimes section, we run into some trouble:

```
struct Closure<F> {
    data: (u8, u16),
    func: F,
}

impl<F> Closure<F>
    // where F: Fn(&'??? (u8, u16)) -> &'??? u8,
{
    fn call<'a>(&'a self) -> &'a u8 {
        (self.func)(&self.data)
    }
}

fn do_it<'b>(data: &'b (u8, u16)) -> &'b u8 { &'b data.0 }

fn main() {
    'x: {
```

```
        let clo = Closure { data: (0, 1), func: do_it };
        println!("{}", clo.call());
    }
}
```

How on earth are we supposed to express the lifetimes on `F`'s trait bound? We need to provide some lifetime there, but the lifetime we care about can't be named until we enter the body of `call`! Also, that isn't some fixed lifetime; `call` works with *any* lifetime `&self` happens to have at that point.

This job requires The Magic of Higher-Rank Trait Bounds (HRTBs). The way we desugar this is as follows:

```
where for<'a> F: Fn(&'a (u8, u16)) -> &'a u8,
```

(Where `Fn(a, b, c) -> d` is itself just sugar for the unstable *real* `Fn` trait)

`for<'a>` can be read as "for all choices of `'a`", and basically produces an *infinite list* of trait bounds that F must satisfy. Intense. There aren't many places outside of the `Fn` traits where we encounter HRTBs, and even for those we have a nice magic sugar for the common cases.

11.7 Subtyping and Variance

Although Rust doesn't have any notion of structural inheritance, it *does* include subtyping. In Rust, subtyping derives entirely from lifetimes. Since lifetimes are scopes, we can partially order them based on the *contains* (outlives) relationship. We can even express this as a generic bound.

Subtyping on lifetimes is in terms of that relationship: if `'a: 'b` ("a contains b" or "a outlives b"), then `'a` is a subtype of `'b`. This is a large source of confusion, because it seems intuitively backwards to many: the bigger scope is a *subtype* of the smaller scope.

This does in fact make sense, though. The intuitive reason for this is that if you expect an `&'a u8`, then it's totally fine for me to hand you an `&'static u8`, in the same way that if you expect an Animal in Java, it's totally fine for me to hand you a Cat. Cats are just Animals *and more,* just as `'static` is just `'a` *and more.*

(Note, the subtyping relationship and typed-ness of lifetimes is a fairly arbitrary construct that some disagree with. However it simplifies our analysis to treat lifetimes and types uniformly.)

Higher-ranked lifetimes are also subtypes of every concrete lifetime. This is because taking an arbitrary lifetime is strictly more general than taking a specific one.

11.7.1 Variance

Variance is where things get a bit complicated.

Variance is a property that *type constructors* have with respect to their arguments. A type constructor in Rust is a generic type with unbound arguments. For instance

`Vec` is a type constructor that takes a `T` and returns a `Vec<T>`. `&` and `&mut` are type constructors that take two inputs: a lifetime, and a type to point to.

A type constructor's *variance* is how the subtyping of its inputs affects the subtyping of its outputs. There are two kinds of variance in Rust:

- F is *variant* over `T` if `T` being a subtype of `U` implies `F<T>` is a subtype of `F<U>` (subtyping "passes through")
- F is *invariant* over `T` otherwise (no subtyping relation can be derived)

(For those of you who are familiar with variance from other languages, what we refer to as "just" variance is in fact *covariance*. Rust does not have contravariance. Historically Rust did have some contravariance but it was scrapped due to poor interactions with other features. If you experience contravariance in Rust call your local compiler developer for medical advice.)

Some important variances:

- `&'a T` is variant over `'a` and `T` (as is `*const T` by metaphor)
- `&'a mut T` is variant with over `'a` but invariant over `T`
- `Fn(T) -> U` is invariant over `T`, but variant over `U`
- `Box`, `Vec`, and all other collections are variant over the types of their contents
- `UnsafeCell<T>`, `Cell<T>`, `RefCell<T>`, `Mutex<T>` and all other interior mutability types are invariant over T (as is `*mut T` by metaphor)

To understand why these variances are correct and desirable, we will consider several examples.

We have already covered why `&'a T` should be variant over `'a` when introducing subtyping: it's desirable to be able to pass longer-lived things where shorter-lived things are needed.

Similar reasoning applies to why it should be variant over T. It is reasonable to be able to pass `&&'static str` where an `&&'a str` is expected. The additional level of indirection does not change the desire to be able to pass longer lived things where shorted lived things are expected.

However this logic doesn't apply to `&mut`. To see why `&mut` should be invariant over T, consider the following code:

```
fn overwrite<T: Copy>(input: &mut T, new: &mut T) {
    *input = *new;
}

fn main() {
    let mut forever_str: &'static str = "hello";
    {
        let string = String::from("world");
        overwrite(&mut forever_str, &mut &*string);
    }
    // Oops, printing free'd memory
    println!("{}", forever_str);
}
```

The signature of `overwrite` is clearly valid: it takes mutable references to two values of the same type, and overwrites one with the other. If `&mut T` was variant over T, then `&mut &'static str` would be a subtype of `&mut &'a str`, since `&'static str` is a subtype of `&'a str`. Therefore the lifetime of `forever_str` would successfully be "shrunk" down to the shorter lifetime of `string`, and `overwrite` would be called successfully. `string` would subsequently be dropped, and `forever_str` would point to freed memory when we print it! Therefore `&mut` should be invariant.

This is the general theme of variance vs invariance: if variance would allow you to store a short-lived value into a longer-lived slot, then you must be invariant.

However it *is* sound for `&'a mut T` to be variant over `'a`. The key difference between `'a` and T is that `'a` is a property of the reference itself, while T is something the reference is borrowing. If you change T's type, then the source still remembers the original type. However if you change the lifetime's type, no one but the reference knows this information, so it's fine. Put another way: `&'a mut T` owns `'a`, but only *borrows* T.

`Box` and `Vec` are interesting cases because they're variant, but you can definitely store values in them! This is where Rust gets really clever: it's fine for them to be variant because you can only store values in them *via a mutable reference*! The mutable reference makes the whole type invariant, and therefore prevents you from smuggling a short-lived type into them.

Being variant allows `Box` and `Vec` to be weakened when shared immutably. So you can pass a `&Box<&'static str>` where a `&Box<&'a str>` is expected.

However what should happen when passing *by-value* is less obvious. It turns out that, yes, you can use subtyping when passing by-value. That is, this works:

```
fn get_box<'a>(str: &'a str) -> Box<&'a str> {
    // string literals are '&'static str's
    Box::new("hello")
}
```

Weakening when you pass by-value is fine because there's no one else who "remembers" the old lifetime in the Box. The reason a variant `&mut` was trouble was because there's always someone else who remembers the original subtype: the actual owner.

The invariance of the cell types can be seen as follows: `&` is like an `&mut` for a cell, because you can still store values in them through an `&`. Therefore cells must be invariant to avoid lifetime smuggling.

`Fn` is the most subtle case because it has mixed variance. To see why `Fn(T) -> U` should be invariant over T, consider the following function signature:

```
// 'a is derived from some parent scope
fn foo(&'a str) -> usize;
```

This signature claims that it can handle any `&str` that lives at least as long as `'a`. Now if this signature was variant over `&'a str`, that would mean

```
fn foo(&'static str) -> usize;
```

could be provided in its place, as it would be a subtype. However this function has a stronger requirement: it says that it can only handle `&'static str`s, and nothing else. Giving `&'a str`s to it would be unsound, as it's free to assume that what it's given lives forever. Therefore functions are not variant over their arguments.

To see why `Fn(T) -> U` should be variant over U, consider the following function signature:

```
// 'a is derived from some parent scope
fn foo(usize) -> &'a str;
```

This signature claims that it will return something that outlives `'a`. It is therefore completely reasonable to provide

```
fn foo(usize) -> &'static str;
```

in its place. Therefore functions are variant over their return type.

`*const` has the exact same semantics as `&`, so variance follows. `*mut` on the other hand can dereference to an `&mut` whether shared or not, so it is marked as invariant just like cells.

This is all well and good for the types the standard library provides, but how is variance determined for type that *you* define? A struct, informally speaking, inherits the variance of its fields. If a struct `Foo` has a generic argument `A` that is used in a field `a`, then Foo's variance over `A` is exactly `a`'s variance. However this is complicated if `A` is used in multiple fields.

- If all uses of A are variant, then Foo is variant over A
- Otherwise, Foo is invariant over A

```
use std::cell::Cell;

struct Foo<'a, 'b, A: 'a, B: 'b, C, D, E, F, G, H> {
    a: &'a A,     // variant over 'a and A
    b: &'b mut B, // invariant over 'b and B
    c: *const C,  // variant over C
    d: *mut D,    // invariant over D
    e: Vec<E>,    // variant over E
    f: Cell<F>,   // invariant over F
    g: G,         // variant over G
    h1: H,        // would also be variant over H except...
    h2: Cell<H>,  // invariant over H, because invariance wins
}
```

11.8 Drop Check

We have seen how lifetimes provide us some fairly simple rules for ensuring that we never read dangling references. However up to this point we have only ever interacted with the *outlives* relationship in an inclusive manner. That is, when we talked about

`'a: 'b`, it was ok for `'a` to live *exactly* as long as `'b`. At first glance, this seems to be a meaningless distinction. Nothing ever gets dropped at the same time as another, right? This is why we used the following desugaring of `let` statements:

```
let x;
let y;

{
    let x;
    {
        let y;
    }
}
```

Each creates its own scope, clearly establishing that one drops before the other. However, what if we do the following?

```
let (x, y) = (vec![], vec![]);
```

Does either value strictly outlive the other? The answer is in fact *no*, neither value strictly outlives the other. Of course, one of x or y will be dropped before the other, but the actual order is not specified. Tuples aren't special in this regard; composite structures just don't guarantee their destruction order as of Rust 1.0.

We *could* specify this for the fields of built-in composites like tuples and structs. However, what about something like Vec? Vec has to manually drop its elements via pure-library code. In general, anything that implements Drop has a chance to fiddle with its innards during its final death knell. Therefore the compiler can't sufficiently reason about the actual destruction order of the contents of any type that implements Drop.

So why do we care? We care because if the type system isn't careful, it could accidentally make dangling pointers. Consider the following simple program:

```
struct Inspector<'a>(&'a u8);

fn main() {
    let (inspector, days);
    days = Box::new(1);
    inspector = Inspector(&days);
}
```

This program is totally sound and compiles today. The fact that `days` does not *strictly* outlive `inspector` doesn't matter. As long as the `inspector` is alive, so is days.

However if we add a destructor, the program will no longer compile!

```
struct Inspector<'a>(&'a u8);

impl<'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
```

```
        println!("I was only {} days from retirement!", self.0);
    }
}

fn main() {
    let (inspector, days);
    days = Box::new(1);
    inspector = Inspector(&days);
    // Let's say `days` happens to get dropped first.
    // Then when Inspector is dropped, it will try to read free'd memory!
}
```

```
<anon>:12:28: 12:32 error: `days` does not live long enough
<anon>:12     inspector = Inspector(&days);
                                    ^~~~
<anon>:9:11: 15:2 note: reference must be valid for the block at 9:10...
<anon>:9 fn main() {
<anon>:10     let (inspector, days);
<anon>:11     days = Box::new(1);
<anon>:12     inspector = Inspector(&days);
<anon>:13     // Let's say `days` happens to get dropped first.
<anon>:14     // Then when Inspector is dropped, it will try to read free'd memory!
          ...
<anon>:10:27: 15:2 note: ...but borrowed value is only valid for the block suffix following sta
<anon>:10     let (inspector, days);
<anon>:11     days = Box::new(1);
<anon>:12     inspector = Inspector(&days);
<anon>:13     // Let's say `days` happens to get dropped first.
<anon>:14     // Then when Inspector is dropped, it will try to read free'd memory!
<anon>:15 }
```

Implementing Drop lets the Inspector execute some arbitrary code during its death. This means it can potentially observe that types that are supposed to live as long as it does actually were destroyed first.

Interestingly, only generic types need to worry about this. If they aren't generic, then the only lifetimes they can harbor are `'static`, which will truly live *forever*. This is why this problem is referred to as *sound generic drop*. Sound generic drop is enforced by the *drop checker*. As of this writing, some of the finer details of how the drop checker validates types is totally up in the air. However The Big Rule is the subtlety that we have focused on this whole section:

For a generic type to soundly implement drop, its generics arguments must strictly outlive it.

Obeying this rule is (usually) necessary to satisfy the borrow checker; obeying it is sufficient but not necessary to be sound. That is, if your type obeys this rule then it's definitely sound to drop.

The reason that it is not always necessary to satisfy the above rule is that some Drop implementations will not access borrowed data even though their type gives them the capability for such access.

For example, this variant of the above `Inspector` example will never accessed borrowed data:

```
struct Inspector<'a>(&'a u8, &'static str);

impl<'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("Inspector(_, {}) knows when *not* to inspect.", self.1);
    }
}

fn main() {
    let (inspector, days);
    days = Box::new(1);
    inspector = Inspector(&days, "gadget");
    // Let's say `days` happens to get dropped first.
    // Even when Inspector is dropped, its destructor will not access the
    // borrowed `days`.
}
```

Likewise, this variant will also never access borrowed data:

```
use std::fmt;

struct Inspector<T: fmt::Display>(T, &'static str);

impl<T: fmt::Display> Drop for Inspector<T> {
    fn drop(&mut self) {
        println!("Inspector(_, {}) knows when *not* to inspect.", self.1);
    }
}

fn main() {
    let (inspector, days): (Inspector<&u8>, Box<u8>);
    days = Box::new(1);
    inspector = Inspector(&days, "gadget");
    // Let's say `days` happens to get dropped first.
    // Even when Inspector is dropped, its destructor will not access the
    // borrowed `days`.
}
```

However, *both* of the above variants are rejected by the borrow checker during the analysis of `fn main`, saying that `days` does not live long enough.

The reason is that the borrow checking analysis of `main` does not know about the internals of each Inspector's Drop implementation. As far as the borrow checker knows while it is analyzing `main`, the body of an inspector's destructor might access that borrowed data.

Therefore, the drop checker forces all borrowed data in a value to strictly outlive that value.

11.8.1 An Escape Hatch

The precise rules that govern drop checking may be less restrictive in the future.

The current analysis is deliberately conservative and trivial; it forces all borrowed data in a value to outlive that value, which is certainly sound.

Future versions of the language may make the analysis more precise, to reduce the number of cases where sound code is rejected as unsafe. This would help address cases such as the two Inspectors above that know not to inspect during destruction.

In the meantime, there is an unstable attribute that one can use to assert (unsafely) that a generic type's destructor is *guaranteed* to not access any expired data, even if its type gives it the capability to do so.

That attribute is called `unsafe_destructor_blind_to_params`. To deploy it on the Inspector example from above, we would write:

```
struct Inspector<'a>(&'a u8, &'static str);

impl<'a> Drop for Inspector<'a> {
    #[unsafe_destructor_blind_to_params]
    fn drop(&mut self) {
        println!("Inspector(_, {}) knows when *not* to inspect.", self.1);
    }
}
```

This attribute has the word `unsafe` in it because the compiler is not checking the implicit assertion that no potentially expired data (e.g. `self.0` above) is accessed.

It is sometimes obvious that no such access can occur, like the case above. However, when dealing with a generic type parameter, such access can occur indirectly. Examples of such indirect access are:

- invoking a callback,
- via a trait method call.

(Future changes to the language, such as impl specialization, may add other avenues for such indirect access.)

Here is an example of invoking a callback:

```
struct Inspector<T>(T, &'static str, Box<for <'r> fn(&'r T) -> String>);

impl<T> Drop for Inspector<T> {
    fn drop(&mut self) {
        // The `self.2` call could access a borrow e.g. if `T` is `&'a _`.
        println!("Inspector({}, {}) unwittingly inspects expired data.",
                 (self.2)(&self.0), self.1);
    }
}
```

Here is an example of a trait method call:

```
use std::fmt;

struct Inspector<T: fmt::Display>(T, &'static str);

impl<T: fmt::Display> Drop for Inspector<T> {
    fn drop(&mut self) {
        // There is a hidden call to `<T as Display>::fmt` below, which
        // could access a borrow e.g. if `T` is `&'a _`
        println!("Inspector({}, {}) unwittingly inspects expired data.",
                 self.0, self.1);
    }
}
```

And of course, all of these accesses could be further hidden within some other method invoked by the destructor, rather than being written directly within it.

In all of the above cases where the `&'a u8` is accessed in the destructor, adding the `#[unsafe_destructor_blind_to_params]` attribute makes the type vulnerable to misuse that the borrower checker will not catch, inviting havoc. It is better to avoid adding the attribute.

11.8.2 Is that all about drop checker?

It turns out that when writing unsafe code, we generally don't need to worry at all about doing the right thing for the drop checker. However there is one special case that you need to worry about, which we will look at in the next section.

11.9 PhantomData

When working with unsafe code, we can often end up in a situation where types or lifetimes are logically associated with a struct, but not actually part of a field. This most commonly occurs with lifetimes. For instance, the `Iter` for `&'a [T]` is (approximately) defined as follows:

```
struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
}
```

However because `'a` is unused within the struct's body, it's *unbounded*. Because of the troubles this has historically caused, unbounded lifetimes and types are *forbidden* in struct definitions. Therefore we must somehow refer to these types in the body. Correctly doing this is necessary to have correct variance and drop checking.

We do this using `PhantomData`, which is a special marker type. `PhantomData` consumes no space, but simulates a field of the given type for the purpose of static analysis. This was deemed to be less error-prone than explicitly telling the type-system the kind of variance that you want, while also providing other useful such as the information needed by drop check.

Iter logically contains a bunch of `&'a T`s, so this is exactly what we tell the PhantomData to simulate:

```
use std::marker;

struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
    _marker: marker::PhantomData<&'a T>,
}
```

and that's it. The lifetime will be bounded, and your iterator will be variant over `'a` and `T`. Everything Just Works.

Another important example is Vec, which is (approximately) defined as follows:

```
struct Vec<T> {
    data: *const T, // *const for variance!
    len: usize,
    cap: usize,
}
```

Unlike the previous example it *appears* that everything is exactly as we want. Every generic argument to Vec shows up in the at least one field. Good to go!

Nope.

The drop checker will generously determine that Vec does not own any values of type T. This will in turn make it conclude that it doesn't need to worry about Vec dropping any T's in its destructor for determining drop check soundness. This will in turn allow people to create unsoundness using Vec's destructor.

In order to tell dropck that we *do* own values of type T, and therefore may drop some T's when *we* drop, we must add an extra PhantomData saying exactly that:

```
use std::marker;

struct Vec<T> {
    data: *const T, // *const for covariance!
    len: usize,
    cap: usize,
    _marker: marker::PhantomData<T>,
}
```

Raw pointers that own an allocation is such a pervasive pattern that the standard library made a utility for itself called `Unique<T>` which:

- wraps a `*const T` for variance
- includes a `PhantomData<T>`,
- auto-derives Send/Sync as if T was contained
- marks the pointer as NonZero for the null-pointer optimization

11.10 Splitting Borrows

The mutual exclusion property of mutable references can be very limiting when working with a composite structure. The borrow checker understands some basic stuff, but will fall over pretty easily. It does understand structs sufficiently to know that it’s possible to borrow disjoint fields of a struct simultaneously. So this works today:

```
struct Foo {
    a: i32,
    b: i32,
    c: i32,
}

let mut x = Foo {a: 0, b: 0, c: 0};
let a = &mut x.a;
let b = &mut x.b;
let c = &x.c;
*b += 1;
let c2 = &x.c;
*a += 10;
println!(”{} {} {} {}”, a, b, c, c2);
```

However borrowck doesn’t understand arrays or slices in any way, so this doesn’t work:

```
let mut x = [1, 2, 3];
let a = &mut x[0];
let b = &mut x[1];
println!(”{} {}”, a, b);
```

```
<anon>:4:14: 4:18 error: cannot borrow ‘x[..]‘ as mutable more than once at a time
<anon>:4 let b = &mut x[1];
                      ^~~~
<anon>:3:14: 3:18 note: previous borrow of ‘x[..]‘ occurs here; the mutable borrow prevents sub
<anon>:3 let a = &mut x[0];
                      ^~~~
<anon>:6:2: 6:2 note: previous borrow ends here
<anon>:1 fn main() {
<anon>:2 let mut x = [1, 2, 3];
<anon>:3 let a = &mut x[0];
<anon>:4 let b = &mut x[1];
<anon>:5 println!(”{} {}”, a, b);
<anon>:6 }
         ^
error: aborting due to 2 previous errors
```

While it was plausible that borrowck could understand this simple case, it’s pretty clearly hopeless for borrowck to understand disjointness in general container types like a tree, especially if distinct keys actually *do* map to the same value.

In order to "teach" borrowck that what we're doing is ok, we need to drop down to unsafe code. For instance, mutable slices expose a `split_at_mut` function that consumes the slice and returns two mutable slices. One for everything to the left of the index, and one for everything to the right. Intuitively we know this is safe because the slices don't overlap, and therefore alias. However the implementation requires some unsafety:

```
fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T]) {
    let len = self.len();
    let ptr = self.as_mut_ptr();
    assert!(mid <= len);
    unsafe {
        (from_raw_parts_mut(ptr, mid),
         from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

This is actually a bit subtle. So as to avoid ever making two `&mut`'s to the same value, we explicitly construct brand-new slices through raw pointers.

However more subtle is how iterators that yield mutable references work. The iterator trait is defined as follows:

```
trait Iterator {
    type Item;

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

Given this definition, Self::Item has *no* connection to `self`. This means that we can call `next` several times in a row, and hold onto all the results *concurrently*. This is perfectly fine for by-value iterators, which have exactly these semantics. It's also actually fine for shared references, as they admit arbitrarily many references to the same thing (although the iterator needs to be a separate object from the thing being shared).

But mutable references make this a mess. At first glance, they might seem completely incompatible with this API, as it would produce multiple mutable references to the same object!

However it actually *does* work, exactly because iterators are one-shot objects. Everything an IterMut yields will be yielded at most once, so we don't actually ever yield multiple mutable references to the same piece of data.

Perhaps surprisingly, mutable iterators don't require unsafe code to be implemented for many types!

For instance here's a singly linked list:

```
# fn main() {}
type Link<T> = Option<Box<Node<T>>>;
```

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

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

pub struct IterMut<'a, T: 'a>(Option<&'a mut Node<T>>);

impl<T> LinkedList<T> {
    fn iter_mut(&mut self) -> IterMut<T> {
        IterMut(self.head.as_mut().map(|node| &mut **node))
    }
}

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

    fn next(&mut self) -> Option<Self::Item> {
        self.0.take().map(|node| {
            self.0 = node.next.as_mut().map(|node| &mut **node);
            &mut node.elem
        })
    }
}
```

Here's a mutable slice:

```
# fn main() {}
use std::mem;

pub struct IterMut<'a, T: 'a>(&'a mut[T]);

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

    fn next(&mut self) -> Option<Self::Item> {
        let slice = mem::replace(&mut self.0, &mut []);
        if slice.is_empty() { return None; }

        let (l, r) = slice.split_at_mut(1);
        self.0 = r;
        l.get_mut(0)
    }
}

impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
```

```
    fn next_back(&mut self) -> Option<Self::Item> {
        let slice = mem::replace(&mut self.0, &mut []);
        if slice.is_empty() { return None; }

        let new_len = slice.len() - 1;
        let (l, r) = slice.split_at_mut(new_len);
        self.0 = l;
        r.get_mut(0)
    }
}
```

And here's a binary tree:

```
# fn main() {}
use std::collections::VecDeque;

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

struct Node<T> {
    elem: T,
    left: Link<T>,
    right: Link<T>,
}

pub struct Tree<T> {
    root: Link<T>,
}

struct NodeIterMut<'a, T: 'a> {
    elem: Option<&'a mut T>,
    left: Option<&'a mut Node<T>>,
    right: Option<&'a mut Node<T>>,
}

enum State<'a, T: 'a> {
    Elem(&'a mut T),
    Node(&'a mut Node<T>),
}

pub struct IterMut<'a, T: 'a>(VecDeque<NodeIterMut<'a, T>>);

impl<T> Tree<T> {
    pub fn iter_mut(&mut self) -> IterMut<T> {
        let mut deque = VecDeque::new();
        self.root.as_mut().map(|root| deque.push_front(root.iter_mut()));
        IterMut(deque)
    }
}
```

```
impl<T> Node<T> {
    pub fn iter_mut(&mut self) -> NodeIterMut<T> {
        NodeIterMut {
            elem: Some(&mut self.elem),
            left: self.left.as_mut().map(|node| &mut **node),
            right: self.right.as_mut().map(|node| &mut **node),
        }
    }
}


impl<'a, T> Iterator for NodeIterMut<'a, T> {
    type Item = State<'a, T>;

    fn next(&mut self) -> Option<Self::Item> {
        match self.left.take() {
            Some(node) => Some(State::Node(node)),
            None => match self.elem.take() {
                Some(elem) => Some(State::Elem(elem)),
                None => match self.right.take() {
                    Some(node) => Some(State::Node(node)),
                    None => None,
                }
            }
        }
    }
}

impl<'a, T> DoubleEndedIterator for NodeIterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        match self.right.take() {
            Some(node) => Some(State::Node(node)),
            None => match self.elem.take() {
                Some(elem) => Some(State::Elem(elem)),
                None => match self.left.take() {
                    Some(node) => Some(State::Node(node)),
                    None => None,
                }
            }
        }
    }
}

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;
    fn next(&mut self) -> Option<Self::Item> {
        loop {
            match self.0.front_mut().and_then(|node_it| node_it.next()) {
                Some(State::Elem(elem)) => return Some(elem),
```

```
                Some(State::Node(node)) => self.0.push_front(node.iter_mut()),
                None => if let None = self.0.pop_front() { return None },
            }
        }
    }
}

impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        loop {
            match self.0.back_mut().and_then(|node_it| node_it.next_back()) {
                Some(State::Elem(elem)) => return Some(elem),
                Some(State::Node(node)) => self.0.push_back(node.iter_mut()),
                None => if let None = self.0.pop_back() { return None },
            }
        }
    }
}
```

All of these are completely safe and work on stable Rust! This ultimately falls out of the simple struct case we saw before: Rust understands that you can safely split a mutable reference into subfields. We can then encode permanently consuming a reference via Options (or in the case of slices, replacing with an empty slice).

Chapter 12

Type Conversions

At the end of the day, everything is just a pile of bits somewhere, and type systems are just there to help us use those bits right. There are two common problems with typing bits: needing to reinterpret those exact bits as a different type, and needing to change the bits to have equivalent meaning for a different type. Because Rust encourages encoding important properties in the type system, these problems are incredibly pervasive. As such, Rust consequently gives you several ways to solve them.

First we'll look at the ways that Safe Rust gives you to reinterpret values. The most trivial way to do this is to just destructure a value into its constituent parts and then build a new type out of them. e.g.

```
struct Foo {
    x: u32,
    y: u16,
}

struct Bar {
    a: u32,
    b: u16,
}

fn reinterpret(foo: Foo) -> Bar {
    let Foo { x, y } = foo;
    Bar { a: x, b: y }
}
```

But this is, at best, annoying. For common conversions, Rust provides more ergonomic alternatives.

12.1 Coercions

Types can implicitly be coerced to change in certain contexts. These changes are generally just *weakening* of types, largely focused around pointers and lifetimes. They mostly exist to make Rust "just work" in more cases, and are largely harmless.

Here's all the kinds of coercion:
Coercion is allowed between the following types:

- Transitivity: `T_1` to `T_3` where `T_1` coerces to `T_2` and `T_2` coerces to `T_3`
- Pointer Weakening:
  - `&mut T` to `&T`
  - `*mut T` to `*const T`
  - `&T` to `*const T`
  - `&mut T` to `*mut T`
- Unsizing: `T` to `U` if `T` implements `CoerceUnsized<U>`

`CoerceUnsized<Pointer<U» for Pointer<T> where T: Unsize<U>` is implemented for all pointer types (including smart pointers like Box and Rc). Unsize is only implemented automatically, and enables the following transformations:

- `[T; n]` => `[T]`
- `T` => `Trait` where `T: Trait`
- `Foo<..., T, ...>` => `Foo<..., U, ...>` where:
  - `T: Unsize<U>`
  - `Foo` is a struct
  - Only the last field of `Foo` has type `T`
  - `T` is not part of the type of any other fields

Coercions occur at a *coercion site*. Any location that is explicitly typed will cause a coercion to its type. If inference is necessary, the coercion will not be performed. Exhaustively, the coercion sites for an expression `e` to type `U` are:

- let statements, statics, and consts: `let x: U = e`
- Arguments to functions: `takes_a_U(e)`
- Any expression that will be returned: `fn foo() -> U { e }`
- Struct literals: `Foo { some_u: e }`
- Array literals: `let x: [U; 10] = [e, ..]`
- Tuple literals: `let x: (U, ..) = (e, ..)`
- The last expression in a block: `let x: U = { ..; e }`

Note that we do not perform coercions when matching traits (except for receivers, see below). If there is an impl for some type `U` and `T` coerces to `U`, that does not constitute an implementation for `T`. For example, the following will not type check, even though it is OK to coerce `t` to `&T` and there is an impl for `&T`:

```
trait Trait {}

fn foo<X: Trait>(t: X) {}

impl<'a> Trait for &'a i32 {}
```

```
fn main() {
    let t: &mut i32 = &mut 0;
    foo(t);
}
```

```
<anon>:10:5: 10:8 error: the trait ‘Trait‘ is not implemented for the type ‘&mut i32‘ [E0277]
<anon>:10     foo(t);
              ^~~
```

12.2 The Dot Operator

The dot operator will perform a lot of magic to convert types. It will perform auto-referencing, auto-dereferencing, and coercion until types match.

TODO: steal information from http://stackoverflow.com/questions/28519997/what-are-rusts-exact-auto-dereferencing-rules/28552082#28552082

12.3 Casts

Casts are a superset of coercions: every coercion can be explicitly invoked via a cast. However some conversions require a cast. While coercions are pervasive and largely harmless, these "true casts" are rare and potentially dangerous. As such, casts must be explicitly invoked using the `as` keyword: `expr as Type`.

True casts generally revolve around raw pointers and the primitive numeric types. Even though they're dangerous, these casts are infallible at runtime. If a cast triggers some subtle corner case no indication will be given that this occurred. The cast will simply succeed. That said, casts must be valid at the type level, or else they will be prevented statically. For instance, `7u8 as bool` will not compile.

That said, casts aren't `unsafe` because they generally can't violate memory safety *on their own*. For instance, converting an integer to a raw pointer can very easily lead to terrible things. However the act of creating the pointer itself is safe, because actually using a raw pointer is already marked as `unsafe`.

Here's an exhaustive list of all the true casts. For brevity, we will use `*` to denote either a `*const` or `*mut`, and `integer` to denote any integral primitive:

- `*T as *U` where `T, U: Sized`
- `*T as *U` TODO: explain unsized situation
- `*T as integer`
- `integer as *T`
- `number as number`
- `C-like-enum as integer`
- `bool as integer`
- `char as integer`
- `u8 as char`
- `&[T; n] as *const T`
- `fn as *T` where `T: Sized`

- `fn as integer`

Note that lengths are not adjusted when casting raw slices - `*const [u16] as *const [u8]` creates a slice that only includes half of the original memory.

Casting is not transitive, that is, even if `e as U1 as U2` is a valid expression, `e as U2` is not necessarily so.

For numeric casts, there are quite a few cases to consider:

- casting between two integers of the same size (e.g. i32 -> u32) is a no-op
- casting from a larger integer to a smaller integer (e.g. u32 -> u8) will truncate
- casting from a smaller integer to a larger integer (e.g. u8 -> u32) will
  - zero-extend if the source is unsigned
  - sign-extend if the source is signed
- casting from a float to an integer will round the float towards zero
  - **NOTE: currently this will cause Undefined Behavior if the rounded value cannot be represented by the target integer type**[1]. This includes Inf and NaN. This is a bug and will be fixed.
- casting from an integer to float will produce the floating point representation of the integer, rounded if necessary (rounding strategy unspecified)
- casting from an f32 to an f64 is perfect and lossless
- casting from an f64 to an f32 will produce the closest possible value (rounding strategy unspecified)
  - **NOTE: currently this will cause Undefined Behavior if the value is finite but larger or smaller than the largest or smallest finite value representable by f32**[2]. This is a bug and will be fixed.

12.4 Transmutes

Get out of our way type system! We're going to reinterpret these bits or die trying! Even though this book is all about doing things that are unsafe, I really can't emphasize that you should deeply think about finding Another Way than the operations covered in this section. This is really, truly, the most horribly unsafe thing you can do in Rust. The railguards here are dental floss.

`mem::transmute<T, U>` takes a value of type `T` and reinterprets it to have type `U`. The only restriction is that the `T` and `U` are verified to have the same size. The ways to cause Undefined Behavior with this are mind boggling.

- First and foremost, creating an instance of *any* type with an invalid state is going to cause arbitrary chaos that can't really be predicted.
- Transmute has an overloaded return type. If you do not specify the return type it may produce a surprising type to satisfy inference.
- Making a primitive with an invalid value is UB

[1] https://github.com/rust-lang/rust/issues/10184
[2] https://github.com/rust-lang/rust/issues/15536

- Transmuting between non-repr(C) types is UB
- Transmuting an & to &mut is UB
  - Transmuting an & to &mut is *always* UB
  - No you can't do it
  - No you're not special
- Transmuting to a reference without an explicitly provided lifetime produces an unbounded lifetime[3]

`mem::transmute_copy<T, U>` somehow manages to be *even more* wildly unsafe than this. It copies `size_of<U>` bytes out of an `&T` and interprets them as a `U`. The size check that `mem::transmute` has is gone (as it may be valid to copy out a prefix), though it is Undefined Behavior for `U` to be larger than `T`.

Also of course you can get most of the functionality of these functions using pointer casts.

[3] `unbounded-lifetimes.html`

Chapter 13

Uninitialized Memory

All runtime-allocated memory in a Rust program begins its life as *uninitialized*. In this state the value of the memory is an indeterminate pile of bits that may or may not even reflect a valid state for the type that is supposed to inhabit that location of memory. Attempting to interpret this memory as a value of *any* type will cause Undefined Behavior. Do Not Do This.

Rust provides mechanisms to work with uninitialized memory in checked (safe) and unchecked (unsafe) ways.

13.1 Checked

Like C, all stack variables in Rust are uninitialized until a value is explicitly assigned to them. Unlike C, Rust statically prevents you from ever reading them until you do:

```
fn main() {
    let x: i32;
    println!(”{}”, x);
}
```

```
src/main.rs:3:20: 3:21 error: use of possibly uninitialized variable: ‘x‘
src/main.rs:3     println!(”{}”, x);
                                 ^
```

This is based off of a basic branch analysis: every branch must assign a value to `x` before it is first used. Interestingly, Rust doesn’t require the variable to be mutable to perform a delayed initialization if every branch assigns exactly once. However the analysis does not take advantage of constant analysis or anything like that. So this compiles:

```
fn main() {
    let x: i32;

    if true {
```

```
        x = 1;
    } else {
        x = 2;
    }

    println!(”{}”, x);
}
```

but this doesn’t:

```
fn main() {
    let x: i32;
    if true {
        x = 1;
    }
    println!(”{}”, x);
}

src/main.rs:6:17: 6:18 error: use of possibly uninitialized variable: ‘x‘
src/main.rs:6   println!(”{}”, x);
```

while this does:

```
fn main() {
    let x: i32;
    if true {
        x = 1;
        println!(”{}”, x);
    }
    // Don’t care that there are branches where it’s not initialized
    // since we don’t use the value in those branches
}
```

Of course, while the analysis doesn’t consider actual values, it does have a relatively sophisticated understanding of dependencies and control flow. For instance, this works:

```
let x: i32;

loop {
    // Rust doesn’t understand that this branch will be taken unconditionally,
    // because it relies on actual values.
    if true {
        // But it does understand that it will only be taken once because
        // we unconditionally break out of it. Therefore ‘x‘ doesn’t
        // need to be marked as mutable.
        x = 0;
        break;
    }
```

```
}
// It also knows that it's impossible to get here without reaching the break.
// And therefore that `x` must be initialized here!
println!("{}", x);
```

If a value is moved out of a variable, that variable becomes logically uninitialized if the type of the value isn't Copy. That is:

```
fn main() {
    let x = 0;
    let y = Box::new(0);
    let z1 = x; // x is still valid because i32 is Copy
    let z2 = y; // y is now logically uninitialized because Box isn't Copy
}
```

However reassigning `y` in this example *would* require `y` to be marked as mutable, as a Safe Rust program could observe that the value of `y` changed:

```
fn main() {
    let mut y = Box::new(0);
    let z = y; // y is now logically uninitialized because Box isn't Copy
    y = Box::new(1); // reinitialize y
}
```

Otherwise it's like `y` is a brand new variable.

13.2 Drop Flags

The examples in the previous section introduce an interesting problem for Rust. We have seen that it's possible to conditionally initialize, deinitialize, and reinitialize locations of memory totally safely. For Copy types, this isn't particularly notable since they're just a random pile of bits. However types with destructors are a different story: Rust needs to know whether to call a destructor whenever a variable is assigned to, or a variable goes out of scope. How can it do this with conditional initialization?

Note that this is not a problem that all assignments need worry about. In particular, assigning through a dereference unconditionally drops, and assigning in a `let` unconditionally doesn't drop:

```
let mut x = Box::new(0); // let makes a fresh variable, so never need to drop
let y = &mut x;
*y = Box::new(1); // Deref assumes the referent is initialized, so always drops
```

This is only a problem when overwriting a previously initialized variable or one of its subfields.

It turns out that Rust actually tracks whether a type should be dropped or not *at runtime*. As a variable becomes initialized and uninitialized, a *drop flag* for that

variable is toggled. When a variable might need to be dropped, this flag is evaluated to determine if it should be dropped.

Of course, it is often the case that a value's initialization state can be statically known at every point in the program. If this is the case, then the compiler can theoretically generate more efficient code! For instance, straight- line code has such *static drop semantics*:

```
let mut x = Box::new(0); // x was uninit; just overwrite.
let mut y = x;           // y was uninit; just overwrite and make x uninit.
x = Box::new(0);         // x was uninit; just overwrite.
y = x;                   // y was init; Drop y, overwrite it, and make x uninit!
                         // y goes out of scope; y was init; Drop y!
                         // x goes out of scope; x was uninit; do nothing.
```

Similarly, branched code where all branches have the same behavior with respect to initialization has static drop semantics:

```
# let condition = true;
let mut x = Box::new(0);    // x was uninit; just overwrite.
if condition {
    drop(x)                 // x gets moved out; make x uninit.
} else {
    println!(”{}”, x);
    drop(x)                 // x gets moved out; make x uninit.
}
x = Box::new(0);            // x was uninit; just overwrite.
                            // x goes out of scope; x was init; Drop x!
```

However code like this *requires* runtime information to correctly Drop:

```
# let condition = true;
let x;
if condition {
    x = Box::new(0);        // x was uninit; just overwrite.
    println!(”{}”, x);
}
                            // x goes out of scope; x might be uninit;
                            // check the flag!
```

Of course, in this case it's trivial to retrieve static drop semantics:

```
# let condition = true;
if condition {
    let x = Box::new(0);
    println!(”{}”, x);
}
```

As of Rust 1.0, the drop flags are actually not-so-secretly stashed in a hidden field of any type that implements Drop. Rust sets the drop flag by overwriting the entire

value with a particular bit pattern. This is pretty obviously Not The Fastest and causes a bunch of trouble with optimizing code. It's legacy from a time when you could do much more complex conditional initialization.

As such work is currently under way to move the flags out onto the stack frame where they more reasonably belong. Unfortunately, this work will take some time as it requires fairly substantial changes to the compiler.

Regardless, Rust programs don't need to worry about uninitialized values on the stack for correctness. Although they might care for performance. Thankfully, Rust makes it easy to take control here! Uninitialized values are there, and you can work with them in Safe Rust, but you're never in danger.

13.3 Unchecked

One interesting exception to this rule is working with arrays. Safe Rust doesn't permit you to partially initialize an array. When you initialize an array, you can either set every value to the same thing with `let x = [val; N]`, or you can specify each member individually with `let x = [val1, val2, val3]`. Unfortunately this is pretty rigid, especially if you need to initialize your array in a more incremental or dynamic way.

Unsafe Rust gives us a powerful tool to handle this problem: `mem::uninitialized`. This function pretends to return a value when really it does nothing at all. Using it, we can convince Rust that we have initialized a variable, allowing us to do trickier things with conditional and incremental initialization.

Unfortunately, this opens us up to all kinds of problems. Assignment has a different meaning to Rust based on whether it believes that a variable is initialized or not. If it's believed uninitialized, then Rust will semantically just memcopy the bits over the uninitialized ones, and do nothing else. However if Rust believes a value to be initialized, it will try to `Drop` the old value! Since we've tricked Rust into believing that the value is initialized, we can no longer safely use normal assignment.

This is also a problem if you're working with a raw system allocator, which returns a pointer to uninitialized memory.

To handle this, we must use the `ptr` module. In particular, it provides three functions that allow us to assign bytes to a location in memory without dropping the old value: `write`, `copy`, and `copy_nonoverlapping`.

- `ptr::write(ptr, val)` takes a `val` and moves it into the address pointed to by `ptr`.
- `ptr::copy(src, dest, count)` copies the bits that `count` T's would occupy from src to dest. (this is equivalent to memmove – note that the argument order is reversed!)
- `ptr::copy_nonoverlapping(src, dest, count)` does what `copy` does, but a little faster on the assumption that the two ranges of memory don't overlap. (this is equivalent to memcpy – note that the argument order is reversed!)

It should go without saying that these functions, if misused, will cause serious havoc or just straight up Undefined Behavior. The only things that these functions *themselves* require is that the locations you want to read and write are allocated. However the

ways writing arbitrary bits to arbitrary locations of memory can break things are basically uncountable!

Putting this all together, we get the following:

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

// size of the array is hard-coded but easy to change. This means we can't
// use [a, b, c] syntax to initialize the array, though!
const SIZE: usize = 10;

let mut x: [Box<u32>; SIZE];

unsafe {
    // convince Rust that x is Totally Initialized
    x = mem::uninitialized();
    for i in 0..SIZE {
        // very carefully overwrite each index without reading it
        // NOTE: exception safety is not a concern; Box can't panic
        ptr::write(&mut x[i], Box::new(i as u32));
    }
}

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

It's worth noting that you don't need to worry about `ptr::write`-style shenanigans with types which don't implement `Drop` or contain `Drop` types, because Rust knows not to try to drop them. Similarly you should be able to assign to fields of partially initialized structs directly if those fields don't contain any `Drop` types.

However when working with uninitialized memory you need to be ever-vigilant for Rust trying to drop values you make like this before they're fully initialized. Every control path through that variable's scope must initialize the value before it ends, if it has a destructor. *This includes code panicking*[1].

And that's about it for working with uninitialized memory! Basically nothing anywhere expects to be handed uninitialized memory, so if you're going to pass it around at all, be sure to be *really* careful.

[1] `unwinding.html`

Chapter 14

Ownership Based Resource Management

OBRM (AKA RAII: Resource Acquisition Is Initialization) is something you'll interact with a lot in Rust. Especially if you use the standard library.

Roughly speaking the pattern is as follows: to acquire a resource, you create an object that manages it. To release the resource, you simply destroy the object, and it cleans up the resource for you. The most common "resource" this pattern manages is simply *memory*. `Box`, `Rc`, and basically everything in `std::collections` is a convenience to enable correctly managing memory. This is particularly important in Rust because we have no pervasive GC to rely on for memory management. Which is the point, really: Rust is about control. However we are not limited to just memory. Pretty much every other system resource like a thread, file, or socket is exposed through this kind of API.

14.1 Constructors

There is exactly one way to create an instance of a user-defined type: name it, and initialize all its fields at once:

```
struct Foo {
    a: u8,
    b: u32,
    c: bool,
}

enum Bar {
    X(u32),
    Y(bool),
}

struct Unit;
```

```
let foo = Foo { a: 0, b: 1, c: false };
let bar = Bar::X(0);
let empty = Unit;
```

That's it. Every other way you make an instance of a type is just calling a totally vanilla function that does some stuff and eventually bottoms out to The One True Constructor.

Unlike C++, Rust does not come with a slew of built-in kinds of constructor. There are no Copy, Default, Assignment, Move, or whatever constructors. The reasons for this are varied, but it largely boils down to Rust's philosophy of *being explicit*.

Move constructors are meaningless in Rust because we don't enable types to "care" about their location in memory. Every type must be ready for it to be blindly memcopied to somewhere else in memory. This means pure on-the-stack-but- still-movable intrusive linked lists are simply not happening in Rust (safely).

Assignment and copy constructors similarly don't exist because move semantics are the only semantics in Rust. At most `x = y` just moves the bits of y into the x variable. Rust does provide two facilities for providing C++'s copy- oriented semantics: `Copy` and `Clone`. Clone is our moral equivalent of a copy constructor, but it's never implicitly invoked. You have to explicitly call `clone` on an element you want to be cloned. Copy is a special case of Clone where the implementation is just "copy the bits". Copy types *are* implicitly cloned whenever they're moved, but because of the definition of Copy this just means not treating the old copy as uninitialized – a no-op.

While Rust provides a `Default` trait for specifying the moral equivalent of a default constructor, it's incredibly rare for this trait to be used. This is because variables aren't implicitly initialized[1]. Default is basically only useful for generic programming. In concrete contexts, a type will provide a static `new` method for any kind of "default" constructor. This has no relation to `new` in other languages and has no special meaning. It's just a naming convention.

TODO: talk about "placement new"?

14.2 Destructors

What the language *does* provide is full-blown automatic destructors through the `Drop` trait, which provides the following method:

```
fn drop(&mut self);
```

This method gives the type time to somehow finish what it was doing.

After `drop` is run, Rust will recursively try to drop all of the fields of `self`.

This is a convenience feature so that you don't have to write "destructor boilerplate" to drop children. If a struct has no special logic for being dropped other than dropping its children, then it means `Drop` doesn't need to be implemented at all!

There is no stable way to prevent this behavior in Rust 1.0.

[1] `uninitialized.html`

Note that taking `&mut self` means that even if you could suppress recursive Drop, Rust will prevent you from e.g. moving fields out of self. For most types, this is totally fine.

For instance, a custom implementation of `Box` might write `Drop` like this:

```
#![feature(alloc, heap_api, drop_in_place, unique)]

extern crate alloc;

use std::ptr::{drop_in_place, Unique};
use std::mem;

use alloc::heap;

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(*self.ptr);
            heap::deallocate((*self.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
        }
    }
}
# fn main() {}
```

and this works fine because when Rust goes to drop the `ptr` field it just sees a Unique[2] that has no actual `Drop` implementation. Similarly nothing can use-after-free the `ptr` because when drop exits, it becomes inaccessible.

However this wouldn't work:

```
#![feature(alloc, heap_api, drop_in_place, unique)]

extern crate alloc;

use std::ptr::{drop_in_place, Unique};
use std::mem;

use alloc::heap;

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
```

[2] `phantom-data.html`

```
            drop_in_place(*self.ptr);
            heap::deallocate((*self.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
        }
    }
}

struct SuperBox<T> { my_box: Box<T> }

impl<T> Drop for SuperBox<T> {
    fn drop(&mut self) {
        unsafe {
            // Hyper-optimized: deallocate the box's contents for it
            // without `drop`ing the contents
            heap::deallocate((*self.my_box.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
        }
    }
}
# fn main() {}
```

After we deallocate the `box`'s ptr in SuperBox's destructor, Rust will happily proceed to tell the box to Drop itself and everything will blow up with use-after-frees and double-frees.

Note that the recursive drop behavior applies to all structs and enums regardless of whether they implement Drop. Therefore something like

```
struct Boxy<T> {
    data1: Box<T>,
    data2: Box<T>,
    info: u32,
}
```

will have its data1 and data2's fields destructors whenever it "would" be dropped, even though it itself doesn't implement Drop. We say that such a type *needs Drop,* even though it is not itself Drop.

Similarly,

```
enum Link {
    Next(Box<Link>),
    None,
}
```

will have its inner Box field dropped if and only if an instance stores the Next variant.

In general this works really nicely because you don't need to worry about adding/removing drops when you refactor your data layout. Still there's certainly many valid usecases for needing to do trickier things with destructors.

The classic safe solution to overriding recursive drop and allowing moving out of Self during `drop` is to use an Option:

```
#![feature(alloc, heap_api, drop_in_place, unique)]

extern crate alloc;

use std::ptr::{drop_in_place, Unique};
use std::mem;

use alloc::heap;

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(*self.ptr);
            heap::deallocate((*self.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
        }
    }
}

struct SuperBox<T> { my_box: Option<Box<T>> }

impl<T> Drop for SuperBox<T> {
    fn drop(&mut self) {
        unsafe {
            // Hyper-optimized: deallocate the box's contents for it
            // without `drop`ing the contents. Need to set the `box`
            // field as `None` to prevent Rust from trying to Drop it.
            let my_box = self.my_box.take().unwrap();
            heap::deallocate((*my_box.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
            mem::forget(my_box);
        }
    }
}
# fn main() {}
```

However this has fairly odd semantics: you're saying that a field that *should* always be Some *may* be None, just because that happens in the destructor. Of course this conversely makes a lot of sense: you can call arbitrary methods on self during the destructor, and this should prevent you from ever doing so after deinitializing the field. Not that it will prevent you from producing any other arbitrarily invalid state in there.

On balance this is an ok choice. Certainly what you should reach for by default. However, in the future we expect there to be a first-class way to announce that a field shouldn't be automatically dropped.

14.3 Leaking

Ownership-based resource management is intended to simplify composition. You acquire resources when you create the object, and you release the resources when it gets destroyed. Since destruction is handled for you, it means you can't forget to release the resources, and it happens as soon as possible! Surely this is perfect and all of our problems are solved.

Everything is terrible and we have new and exotic problems to try to solve.

Many people like to believe that Rust eliminates resource leaks. In practice, this is basically true. You would be surprised to see a Safe Rust program leak resources in an uncontrolled way.

However from a theoretical perspective this is absolutely not the case, no matter how you look at it. In the strictest sense, "leaking" is so abstract as to be unpreventable. It's quite trivial to initialize a collection at the start of a program, fill it with tons of objects with destructors, and then enter an infinite event loop that never refers to it. The collection will sit around uselessly, holding on to its precious resources until the program terminates (at which point all those resources would have been reclaimed by the OS anyway).

We may consider a more restricted form of leak: failing to drop a value that is unreachable. Rust also doesn't prevent this. In fact Rust *has a function for doing this*: `mem::forget`. This function consumes the value it is passed *and then doesn't run its destructor*.

In the past `mem::forget` was marked as unsafe as a sort of lint against using it, since failing to call a destructor is generally not a well-behaved thing to do (though useful for some special unsafe code). However this was generally determined to be an untenable stance to take: there are many ways to fail to call a destructor in safe code. The most famous example is creating a cycle of reference-counted pointers using interior mutability.

It is reasonable for safe code to assume that destructor leaks do not happen, as any program that leaks destructors is probably wrong. However *unsafe* code cannot rely on destructors to be run in order to be safe. For most types this doesn't matter: if you leak the destructor then the type is by definition inaccessible, so it doesn't matter, right? For instance, if you leak a `Box<u8>` then you waste some memory but that's hardly going to violate memory-safety.

However where we must be careful with destructor leaks are *proxy* types. These are types which manage access to a distinct object, but don't actually own it. Proxy objects are quite rare. Proxy objects you'll need to care about are even rarer. However we'll focus on three interesting examples in the standard library:

- `vec::Drain`
- `Rc`
- `thread::scoped::JoinGuard`

Drain

`drain` is a collections API that moves data out of the container without consuming the container. This enables us to reuse the allocation of a `Vec` after claiming ownership over all of its contents. It produces an iterator (Drain) that returns the contents of the Vec by-value.

Now, consider Drain in the middle of iteration: some values have been moved out, and others haven't. This means that part of the Vec is now full of logically uninitialized data! We could backshift all the elements in the Vec every time we remove a value, but this would have pretty catastrophic performance consequences.

Instead, we would like Drain to fix the Vec's backing storage when it is dropped. It should run itself to completion, backshift any elements that weren't removed (drain supports subranges), and then fix Vec's `len`. It's even unwinding-safe! Easy!

Now consider the following:

```
let mut vec = vec![Box::new(0); 4];

{
    // start draining, vec can no longer be accessed
    let mut drainer = vec.drain(..);

    // pull out two elements and immediately drop them
    drainer.next();
    drainer.next();

    // get rid of drainer, but don't call its destructor
    mem::forget(drainer);
}

// Oops, vec[0] was dropped, we're reading a pointer into free'd memory!
println!("{}", vec[0]);
```

This is pretty clearly Not Good. Unfortunately, we're kind of stuck between a rock and a hard place: maintaining consistent state at every step has an enormous cost (and would negate any benefits of the API). Failing to maintain consistent state gives us Undefined Behavior in safe code (making the API unsound).

So what can we do? Well, we can pick a trivially consistent state: set the Vec's len to be 0 when we start the iteration, and fix it up if necessary in the destructor. That way, if everything executes like normal we get the desired behavior with minimal overhead. But if someone has the *audacity* to mem::forget us in the middle of the iteration, all that does is *leak even more* (and possibly leave the Vec in an unexpected but otherwise consistent state). Since we've accepted that mem::forget is safe, this is definitely safe. We call leaks causing more leaks a *leak amplification*.

Rc

Rc is an interesting case because at first glance it doesn't appear to be a proxy value at all. After all, it manages the data it points to, and dropping all the Rcs for a value will

drop that value. Leaking an Rc doesn't seem like it would be particularly dangerous. It will leave the refcount permanently incremented and prevent the data from being freed or dropped, but that seems just like Box, right?

Nope.

Let's consider a simplified implementation of Rc:

```
struct Rc<T> {
    ptr: *mut RcBox<T>,
}

struct RcBox<T> {
    data: T,
    ref_count: usize,
}

impl<T> Rc<T> {
    fn new(data: T) -> Self {
        unsafe {
            // Wouldn't it be nice if heap::allocate worked like this?
            let ptr = heap::allocate::<RcBox<T>>();
            ptr::write(ptr, RcBox {
                data: data,
                ref_count: 1,
            });
            Rc { ptr: ptr }
        }
    }

    fn clone(&self) -> Self {
        unsafe {
            (*self.ptr).ref_count += 1;
        }
        Rc { ptr: self.ptr }
    }
}

impl<T> Drop for Rc<T> {
    fn drop(&mut self) {
        unsafe {
            (*self.ptr).ref_count -= 1;
            if (*self.ptr).ref_count == 0 {
                // drop the data and then free it
                ptr::read(self.ptr);
                heap::deallocate(self.ptr);
            }
        }
    }
}
```

This code contains an implicit and subtle assumption: `ref_count` can fit in a `usize`, because there can't be more than `usize::MAX` Rcs in memory. However this itself assumes that the `ref_count` accurately reflects the number of Rcs in memory, which we know is false with `mem::forget`. Using `mem::forget` we can overflow the `ref_count`, and then get it down to 0 with outstanding Rcs. Then we can happily use-after-free the inner data. Bad Bad Not Good.

This can be solved by just checking the `ref_count` and doing *something*. The standard library's stance is to just abort, because your program has become horribly degenerate. Also *oh my gosh* it's such a ridiculous corner case.

thread::scoped::JoinGuard

The thread::scoped API intends to allow threads to be spawned that reference data on their parent's stack without any synchronization over that data by ensuring the parent joins the thread before any of the shared data goes out of scope.

```
pub fn scoped<'a, F>(f: F) -> JoinGuard<'a>
    where F: FnOnce() + Send + 'a
```

Here `f` is some closure for the other thread to execute. Saying that `F: Send +'a` is saying that it closes over data that lives for `'a`, and it either owns that data or the data was Sync (implying `&data` is Send).

Because JoinGuard has a lifetime, it keeps all the data it closes over borrowed in the parent thread. This means the JoinGuard can't outlive the data that the other thread is working on. When the JoinGuard *does* get dropped it blocks the parent thread, ensuring the child terminates before any of the closed-over data goes out of scope in the parent.

Usage looked like:

```
let mut data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
{
    let guards = vec![];
    for x in &mut data {
        // Move the mutable reference into the closure, and execute
        // it on a different thread. The closure has a lifetime bound
        // by the lifetime of the mutable reference `x` we store in it.
        // The guard that is returned is in turn assigned the lifetime
        // of the closure, so it also mutably borrows `data` as `x` did.
        // This means we cannot access `data` until the guard goes away.
        let guard = thread::scoped(move || {
            *x *= 2;
        });
        // store the thread's guard for later
        guards.push(guard);
    }
    // All guards are dropped here, forcing the threads to join
    // (this thread blocks here until the others terminate).
    // Once the threads join, the borrow expires and the data becomes
```

```
    // accessible again in this thread.
}
// data is definitely mutated here.
```

In principle, this totally works! Rust's ownership system perfectly ensures it! ...except it relies on a destructor being called to be safe.

```
let mut data = Box::new(0);
{
    let guard = thread::scoped(|| {
        // This is at best a data race. At worst, it's also a use-after-free.
        *data += 1;
    });
    // Because the guard is forgotten, expiring the loan without blocking this
    // thread.
    mem::forget(guard);
}
// So the Box is dropped here while the scoped thread may or may not be trying
// to access it.
```

Dang. Here the destructor running was pretty fundamental to the API, and it had to be scrapped in favor of a completely different design.

Chapter 15

Unwinding

Rust has a *tiered* error-handling scheme:

- If something might reasonably be absent, Option is used.
- If something goes wrong and can reasonably be handled, Result is used.
- If something goes wrong and cannot reasonably be handled, the thread panics.
- If something catastrophic happens, the program aborts.

Option and Result are overwhelmingly preferred in most situations, especially since they can be promoted into a panic or abort at the API user's discretion. Panics cause the thread to halt normal execution and unwind its stack, calling destructors as if every function instantly returned.

As of 1.0, Rust is of two minds when it comes to panics. In the long-long-ago, Rust was much more like Erlang. Like Erlang, Rust had lightweight tasks, and tasks were intended to kill themselves with a panic when they reached an untenable state. Unlike an exception in Java or C++, a panic could not be caught at any time. Panics could only be caught by the owner of the task, at which point they had to be handled or *that* task would itself panic.

Unwinding was important to this story because if a task's destructors weren't called, it would cause memory and other system resources to leak. Since tasks were expected to die during normal execution, this would make Rust very poor for long-running systems!

As the Rust we know today came to be, this style of programming grew out of fashion in the push for less-and-less abstraction. Light-weight tasks were killed in the name of heavy-weight OS threads. Still, on stable Rust as of 1.0 panics can only be caught by the parent thread. This means catching a panic requires spinning up an entire OS thread! This unfortunately stands in conflict to Rust's philosophy of zero-cost abstractions.

There is an unstable API called `catch_panic` that enables catching a panic without spawning a thread. Still, we would encourage you to only do this sparingly. In particular, Rust's current unwinding implementation is heavily optimized for the "doesn't unwind" case. If a program doesn't unwind, there should be no runtime cost for the program being *ready* to unwind. As a consequence, actually unwinding will be

more expensive than in e.g. Java. Don't build your programs to unwind under normal circumstances. Ideally, you should only panic for programming errors or *extreme* problems.

Rust's unwinding strategy is not specified to be fundamentally compatible with any other language's unwinding. As such, unwinding into Rust from another language, or unwinding into another language from Rust is Undefined Behavior. You must *absolutely* catch any panics at the FFI boundary! What you do at that point is up to you, but *something* must be done. If you fail to do this, at best, your application will crash and burn. At worst, your application *won't* crash and burn, and will proceed with completely clobbered state.

15.1 Exception Safety

Although programs should use unwinding sparingly, there's a lot of code that *can* panic. If you unwrap a None, index out of bounds, or divide by 0, your program will panic. On debug builds, every arithmetic operation can panic if it overflows. Unless you are very careful and tightly control what code runs, pretty much everything can unwind, and you need to be ready for it.

Being ready for unwinding is often referred to as *exception safety* in the broader programming world. In Rust, there are two levels of exception safety that one may concern themselves with:

- In unsafe code, we *must* be exception safe to the point of not violating memory safety. We'll call this *minimal* exception safety.
- In safe code, it is *good* to be exception safe to the point of your program doing the right thing. We'll call this *maximal* exception safety.

As is the case in many places in Rust, Unsafe code must be ready to deal with bad Safe code when it comes to unwinding. Code that transiently creates unsound states must be careful that a panic does not cause that state to be used. Generally this means ensuring that only non-panicking code is run while these states exist, or making a guard that cleans up the state in the case of a panic. This does not necessarily mean that the state a panic witnesses is a fully coherent state. We need only guarantee that it's a *safe* state.

Most Unsafe code is leaf-like, and therefore fairly easy to make exception-safe. It controls all the code that runs, and most of that code can't panic. However it is not uncommon for Unsafe code to work with arrays of temporarily uninitialized data while repeatedly invoking caller-provided code. Such code needs to be careful and consider exception safety.

Vec::push_all

`Vec::push_all` is a temporary hack to get extending a Vec by a slice reliably efficient without specialization. Here's a simple implementation:

```
impl<T: Clone> Vec<T> {
    fn push_all(&mut self, to_push: &[T]) {
        self.reserve(to_push.len());
        unsafe {
            // can't overflow because we just reserved this
            self.set_len(self.len() + to_push.len());

            for (i, x) in to_push.iter().enumerate() {
                self.ptr().offset(i as isize).write(x.clone());
            }
        }
    }
}
```

We bypass `push` in order to avoid redundant capacity and `len` checks on the Vec that we definitely know has capacity. The logic is totally correct, except there's a subtle problem with our code: it's not exception-safe! `set_len`, `offset`, and `write` are all fine; `clone` is the panic bomb we over-looked.

Clone is completely out of our control, and is totally free to panic. If it does, our function will exit early with the length of the Vec set too large. If the Vec is looked at or dropped, uninitialized memory will be read!

The fix in this case is fairly simple. If we want to guarantee that the values we *did* clone are dropped, we can set the `len` every loop iteration. If we just want to guarantee that uninitialized memory can't be observed, we can set the `len` after the loop.

BinaryHeap::sift_up

Bubbling an element up a heap is a bit more complicated than extending a Vec. The pseudocode is as follows:

```
bubble_up(heap, index):
    while index != 0 && heap[index] < heap[parent(index)]:
        heap.swap(index, parent(index))
        index = parent(index)
```

A literal transcription of this code to Rust is totally fine, but has an annoying performance characteristic: the `self` element is swapped over and over again uselessly. We would rather have the following:

```
bubble_up(heap, index):
    let elem = heap[index]
    while index != 0 && element < heap[parent(index)]:
        heap[index] = heap[parent(index)]
        index = parent(index)
    heap[index] = elem
```

This code ensures that each element is copied as little as possible (it is in fact necessary that elem be copied twice in general). However it now exposes some exception safety

trouble! At all times, there exists two copies of one value. If we panic in this function something will be double-dropped. Unfortunately, we also don't have full control of the code: that comparison is user-defined!

Unlike Vec, the fix isn't as easy here. One option is to break the user-defined code and the unsafe code into two separate phases:

```
bubble_up(heap, index):
    let end_index = index;
    while end_index != 0 && heap[end_index] < heap[parent(end_index)]:
        end_index = parent(end_index)

    let elem = heap[index]
    while index != end_index:
        heap[index] = heap[parent(index)]
        index = parent(index)
    heap[index] = elem
```

If the user-defined code blows up, that's no problem anymore, because we haven't actually touched the state of the heap yet. Once we do start messing with the heap, we're working with only data and functions that we trust, so there's no concern of panics.

Perhaps you're not happy with this design. Surely it's cheating! And we have to do the complex heap traversal *twice*! Alright, let's bite the bullet. Let's intermix untrusted and unsafe code *for reals*.

If Rust had `try` and `finally` like in Java, we could do the following:

```
bubble_up(heap, index):
    let elem = heap[index]
    try:
        while index != 0 && element < heap[parent(index)]:
            heap[index] = heap[parent(index)]
            index = parent(index)
    finally:
        heap[index] = elem
```

The basic idea is simple: if the comparison panics, we just toss the loose element in the logically uninitialized index and bail out. Anyone who observes the heap will see a potentially *inconsistent* heap, but at least it won't cause any double-drops! If the algorithm terminates normally, then this operation happens to coincide precisely with the how we finish up regardless.

Sadly, Rust has no such construct, so we're going to need to roll our own! The way to do this is to store the algorithm's state in a separate struct with a destructor for the "finally" logic. Whether we panic or not, that destructor will run and clean up after us.

```
struct Hole<'a, T: 'a> {
    data: &'a mut [T],
    /// `elt` is always `Some` from new until drop.
```

```
    elt: Option<T>,
    pos: usize,
}

impl<'a, T> Hole<'a, T> {
    fn new(data: &'a mut [T], pos: usize) -> Self {
        unsafe {
            let elt = ptr::read(&data[pos]);
            Hole {
                data: data,
                elt: Some(elt),
                pos: pos,
            }
        }
    }

    fn pos(&self) -> usize { self.pos }

    fn removed(&self) -> &T { self.elt.as_ref().unwrap() }

    unsafe fn get(&self, index: usize) -> &T { &self.data[index] }

    unsafe fn move_to(&mut self, index: usize) {
        let index_ptr: *const _ = &self.data[index];
        let hole_ptr = &mut self.data[self.pos];
        ptr::copy_nonoverlapping(index_ptr, hole_ptr, 1);
        self.pos = index;
    }
}

impl<'a, T> Drop for Hole<'a, T> {
    fn drop(&mut self) {
        // fill the hole again
        unsafe {
            let pos = self.pos;
            ptr::write(&mut self.data[pos], self.elt.take().unwrap());
        }
    }
}

impl<T: Ord> BinaryHeap<T> {
    fn sift_up(&mut self, pos: usize) {
        unsafe {
            // Take out the value at `pos` and create a hole.
            let mut hole = Hole::new(&mut self.data, pos);

            while hole.pos() != 0 {
                let parent = parent(hole.pos());
                if hole.removed() <= hole.get(parent) { break }
```

```
                hole.move_to(parent);
            }
            // Hole will be unconditionally filled here; panic or not!
        }
    }
}
```

15.2 Poisoning

Although all unsafe code *must* ensure it has minimal exception safety, not all types ensure *maximal* exception safety. Even if the type does, your code may ascribe additional meaning to it. For instance, an integer is certainly exception-safe, but has no semantics on its own. It's possible that code that panics could fail to correctly update the integer, producing an inconsistent program state.

This is *usually* fine, because anything that witnesses an exception is about to get destroyed. For instance, if you send a Vec to another thread and that thread panics, it doesn't matter if the Vec is in a weird state. It will be dropped and go away forever. However some types are especially good at smuggling values across the panic boundary.

These types may choose to explicitly *poison* themselves if they witness a panic. Poisoning doesn't entail anything in particular. Generally it just means preventing normal usage from proceeding. The most notable example of this is the standard library's Mutex type. A Mutex will poison itself if one of its MutexGuards (the thing it returns when a lock is obtained) is dropped during a panic. Any future attempts to lock the Mutex will return an `Err` or panic.

Mutex poisons not for true safety in the sense that Rust normally cares about. It poisons as a safety-guard against blindly using the data that comes out of a Mutex that has witnessed a panic while locked. The data in such a Mutex was likely in the middle of being modified, and as such may be in an inconsistent or incomplete state. It is important to note that one cannot violate memory safety with such a type if it is correctly written. After all, it must be minimally exception-safe!

However if the Mutex contained, say, a BinaryHeap that does not actually have the heap property, it's unlikely that any code that uses it will do what the author intended. As such, the program should not proceed normally. Still, if you're double-plus-sure that you can do *something* with the value, the Mutex exposes a method to get the lock anyway. It *is* safe, after all. Just maybe nonsense.

Chapter 16

Concurrency

Rust as a language doesn't *really* have an opinion on how to do concurrency or parallelism. The standard library exposes OS threads and blocking sys-calls because everyone has those, and they're uniform enough that you can provide an abstraction over them in a relatively uncontroversial way. Message passing, green threads, and async APIs are all diverse enough that any abstraction over them tends to involve trade-offs that we weren't willing to commit to for 1.0.

However the way Rust models concurrency makes it relatively easy to design your own concurrency paradigm as a library and have everyone else's code Just Work with yours. Just require the right lifetimes and Send and Sync where appropriate and you're off to the races. Or rather, off to the... not... having... races.

16.1 Races

Safe Rust guarantees an absence of data races, which are defined as:

- two or more threads concurrently accessing a location of memory
- one of them is a write
- one of them is unsynchronized

A data race has Undefined Behavior, and is therefore impossible to perform in Safe Rust. Data races are *mostly* prevented through rust's ownership system: it's impossible to alias a mutable reference, so it's impossible to perform a data race. Interior mutability makes this more complicated, which is largely why we have the Send and Sync traits (see below).

However Rust does not prevent general race conditions.

This is pretty fundamentally impossible, and probably honestly undesirable. Your hardware is racy, your OS is racy, the other programs on your computer are racy, and the world this all runs in is racy. Any system that could genuinely claim to prevent *all* race conditions would be pretty awful to use, if not just incorrect.

So it's perfectly "fine" for a Safe Rust program to get deadlocked or do something incredibly stupid with incorrect synchronization. Obviously such a program isn't

very good, but Rust can only hold your hand so far. Still, a race condition can’t violate memory safety in a Rust program on its own. Only in conjunction with some other unsafe code can a race condition actually violate memory safety. For instance:

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];
// Arc so that the memory the AtomicUsize is stored in still exists for
// the other thread to increment, even if we completely finish executing
// before it. Rust won’t compile the program without it, because of the
// lifetime requirements of thread::spawn!
let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

// ‘move‘ captures other_idx by-value, moving it into this thread
thread::spawn(move || {
    // It’s ok to mutate idx because this value
    // is an atomic, so it can’t cause a Data Race.
    other_idx.fetch_add(10, Ordering::SeqCst);
});

// Index with the value loaded from the atomic. This is safe because we
// read the atomic memory only once, and then pass a copy of that value
// to the Vec’s indexing implementation. This indexing will be correctly
// bounds checked, and there’s no chance of the value getting changed
// in the middle. However our program may panic if the thread we spawned
// managed to increment before this ran. A race condition because correct
// program execution (panicking is rarely correct) depends on order of
// thread execution.
println!(”{}”, data[idx.load(Ordering::SeqCst)]);
```

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];

let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

// ‘move‘ captures other_idx by-value, moving it into this thread
thread::spawn(move || {
    // It’s ok to mutate idx because this value
    // is an atomic, so it can’t cause a Data Race.
    other_idx.fetch_add(10, Ordering::SeqCst);
});
```

```
if idx.load(Ordering::SeqCst) < data.len() {
    unsafe {
        // Incorrectly loading the idx after we did the bounds check.
        // It could have changed. This is a race condition, *and dangerous*
        // because we decided to do 'get_unchecked', which is 'unsafe'.
        println!("{}", data.get_unchecked(idx.load(Ordering::SeqCst)));
    }
}
```

16.2 Send and Sync

Not everything obeys inherited mutability, though. Some types allow you to multiply alias a location in memory while mutating it. Unless these types use synchronization to manage this access, they are absolutely not thread safe. Rust captures this through the `Send` and `Sync` traits.

- A type is Send if it is safe to send it to another thread.
- A type is Sync if it is safe to share between threads (`&T` is Send).

Send and Sync are fundamental to Rust's concurrency story. As such, a substantial amount of special tooling exists to make them work right. First and foremost, they're unsafe traits[1]. This means that they are unsafe to implement, and other unsafe code can assume that they are correctly implemented. Since they're *marker traits* (they have no associated items like methods), correctly implemented simply means that they have the intrinsic properties an implementor should have. Incorrectly implementing Send or Sync can cause Undefined Behavior.

Send and Sync are also automatically derived traits. This means that, unlike every other trait, if a type is composed entirely of Send or Sync types, then it is Send or Sync. Almost all primitives are Send and Sync, and as a consequence pretty much all types you'll ever interact with are Send and Sync.

Major exceptions include:

- raw pointers are neither Send nor Sync (because they have no safety guards).
- `UnsafeCell` isn't Sync (and therefore `Cell` and `RefCell` aren't).
- `Rc` isn't Send or Sync (because the refcount is shared and unsynchronized).

`Rc` and `UnsafeCell` are very fundamentally not thread-safe: they enable unsynchronized shared mutable state. However raw pointers are, strictly speaking, marked as thread-unsafe as more of a *lint*. Doing anything useful with a raw pointer requires dereferencing it, which is already unsafe. In that sense, one could argue that it would be "fine" for them to be marked as thread safe.

However it's important that they aren't thread safe to prevent types that contain them from being automatically marked as thread safe. These types have non-trivial untracked ownership, and it's unlikely that their author was necessarily thinking hard about thread safety. In the case of Rc, we have a nice example of a type that contains a `*mut` that is definitely not thread safe.

Types that aren't automatically derived can simply implement them if desired:

[1] `safe-unsafe-meaning.html`

```
struct MyBox(*mut u8);

unsafe impl Send for MyBox {}
unsafe impl Sync for MyBox {}
```

In the *incredibly rare* case that a type is inappropriately automatically derived to be Send or Sync, then one can also unimplement Send and Sync:

```
#![feature(optin_builtin_traits)]

// I have some magic semantics for some synchronization primitive!
struct SpecialThreadToken(u8);

impl !Send for SpecialThreadToken {}
impl !Sync for SpecialThreadToken {}
```

Note that *in and of itself* it is impossible to incorrectly derive Send and Sync. Only types that are ascribed special meaning by other unsafe code can possible cause trouble by being incorrectly Send or Sync.

Most uses of raw pointers should be encapsulated behind a sufficient abstraction that Send and Sync can be derived. For instance all of Rust's standard collections are Send and Sync (when they contain Send and Sync types) in spite of their pervasive use of raw pointers to manage allocations and complex ownership. Similarly, most iterators into these collections are Send and Sync because they largely behave like an `&` or `&mut` into the collection.

TODO: better explain what can or can't be Send or Sync. Sufficient to appeal only to data races?

16.3 Atomics

Rust pretty blatantly just inherits C11's memory model for atomics. This is not due to this model being particularly excellent or easy to understand. Indeed, this model is quite complex and known to have several flaws[2]. Rather, it is a pragmatic concession to the fact that *everyone* is pretty bad at modeling atomics. At very least, we can benefit from existing tooling and research around C.

Trying to fully explain the model in this book is fairly hopeless. It's defined in terms of madness-inducing causality graphs that require a full book to properly understand in a practical way. If you want all the nitty-gritty details, you should check out C's specification (Section 7.17)[3]. Still, we'll try to cover the basics and some of the problems Rust developers face.

The C11 memory model is fundamentally about trying to bridge the gap between the semantics we want, the optimizations compilers want, and the inconsistent chaos our hardware wants. *We* would like to just write programs and have them do exactly what we said but, you know, fast. Wouldn't that be great?

[2] http://plv.mpi-sws.org/c11comp/popl15.pdf

[3] http://www.open-std.org/jtc1/sc22/wg14/www/standards.html#9899

16.3.1 Compiler Reordering

Compilers fundamentally want to be able to do all sorts of crazy transformations to reduce data dependencies and eliminate dead code. In particular, they may radically change the actual order of events, or make events never occur! If we write something like

```
x = 1;
y = 3;
x = 2;
```

The compiler may conclude that it would be best if your program did

```
x = 2;
y = 3;
```

This has inverted the order of events and completely eliminated one event. From a single-threaded perspective this is completely unobservable: after all the statements have executed we are in exactly the same state. But if our program is multi-threaded, we may have been relying on `x` to actually be assigned to 1 before `y` was assigned. We would like the compiler to be able to make these kinds of optimizations, because they can seriously improve performance. On the other hand, we'd also like to be able to depend on our program *doing the thing we said.*

16.3.2 Hardware Reordering

On the other hand, even if the compiler totally understood what we wanted and respected our wishes, our hardware might instead get us in trouble. Trouble comes from CPUs in the form of memory hierarchies. There is indeed a global shared memory space somewhere in your hardware, but from the perspective of each CPU core it is *so very far away* and *so very slow.* Each CPU would rather work with its local cache of the data and only go through all the anguish of talking to shared memory only when it doesn't actually have that memory in cache.

After all, that's the whole point of the cache, right? If every read from the cache had to run back to shared memory to double check that it hadn't changed, what would the point be? The end result is that the hardware doesn't guarantee that events that occur in the same order on *one* thread, occur in the same order on *another* thread. To guarantee this, we must issue special instructions to the CPU telling it to be a bit less smart.

For instance, say we convince the compiler to emit this logic:

```
initial state: x = 0, y = 1

THREAD 1        THREAD2
y = 3;          if x == 1 {
x = 1;              y *= 2;
                }
```

Ideally this program has 2 possible final states:

- `y = 3`: (thread 2 did the check before thread 1 completed)
- `y = 6`: (thread 2 did the check after thread 1 completed)

However there's a third potential state that the hardware enables:

- `y = 2`: (thread 2 saw `x = 1`, but not `y = 3`, and then overwrote `y = 3`)

It's worth noting that different kinds of CPU provide different guarantees. It is common to separate hardware into two categories: strongly-ordered and weakly- ordered. Most notably x86/64 provides strong ordering guarantees, while ARM provides weak ordering guarantees. This has two consequences for concurrent programming:

- Asking for stronger guarantees on strongly-ordered hardware may be cheap or even free because they already provide strong guarantees unconditionally. Weaker guarantees may only yield performance wins on weakly-ordered hardware.
- Asking for guarantees that are too weak on strongly-ordered hardware is more likely to *happen* to work, even though your program is strictly incorrect. If possible, concurrent algorithms should be tested on weakly-ordered hardware.

16.3.3 Data Accesses

The C11 memory model attempts to bridge the gap by allowing us to talk about the *causality* of our program. Generally, this is by establishing a *happens before* relationship between parts of the program and the threads that are running them. This gives the hardware and compiler room to optimize the program more aggressively where a strict happens-before relationship isn't established, but forces them to be more careful where one is established. The way we communicate these relationships are through *data accesses* and *atomic accesses*.

Data accesses are the bread-and-butter of the programming world. They are fundamentally unsynchronized and compilers are free to aggressively optimize them. In particular, data accesses are free to be reordered by the compiler on the assumption that the program is single-threaded. The hardware is also free to propagate the changes made in data accesses to other threads as lazily and inconsistently as it wants. Most critically, data accesses are how data races happen. Data accesses are very friendly to the hardware and compiler, but as we've seen they offer *awful* semantics to try to write synchronized code with. Actually, that's too weak.

It is literally impossible to write correct synchronized code using only data accesses.

Atomic accesses are how we tell the hardware and compiler that our program is multi-threaded. Each atomic access can be marked with an *ordering* that specifies what kind of relationship it establishes with other accesses. In practice, this boils down to telling the compiler and hardware certain things they *can't* do. For the compiler, this largely revolves around re-ordering of instructions. For the hardware, this largely revolves around how writes are propagated to other threads. The set of orderings Rust exposes are:

- Sequentially Consistent (SeqCst)
- Release
- Acquire
- Relaxed

(Note: We explicitly do not expose the C11 *consume* ordering)

TODO: negative reasoning vs positive reasoning? TODO: "can't forget to synchronize"

16.3.4 Sequentially Consistent

Sequentially Consistent is the most powerful of all, implying the restrictions of all other orderings. Intuitively, a sequentially consistent operation cannot be reordered: all accesses on one thread that happen before and after a SeqCst access stay before and after it. A data-race-free program that uses only sequentially consistent atomics and data accesses has the very nice property that there is a single global execution of the program's instructions that all threads agree on. This execution is also particularly nice to reason about: it's just an interleaving of each thread's individual executions. This does not hold if you start using the weaker atomic orderings.

The relative developer-friendliness of sequential consistency doesn't come for free. Even on strongly-ordered platforms sequential consistency involves emitting memory fences.

In practice, sequential consistency is rarely necessary for program correctness. However sequential consistency is definitely the right choice if you're not confident about the other memory orders. Having your program run a bit slower than it needs to is certainly better than it running incorrectly! It's also mechanically trivial to downgrade atomic operations to have a weaker consistency later on. Just change `SeqCst` to `Relaxed` and you're done! Of course, proving that this transformation is *correct* is a whole other matter.

16.3.5 Acquire-Release

Acquire and Release are largely intended to be paired. Their names hint at their use case: they're perfectly suited for acquiring and releasing locks, and ensuring that critical sections don't overlap.

Intuitively, an acquire access ensures that every access after it stays after it. However operations that occur before an acquire are free to be reordered to occur after it. Similarly, a release access ensures that every access before it stays before it. However operations that occur after a release are free to be reordered to occur before it.

When thread A releases a location in memory and then thread B subsequently acquires *the same* location in memory, causality is established. Every write that happened before A's release will be observed by B after its release. However no causality is established with any other threads. Similarly, no causality is established if A and B access *different* locations in memory.

Basic use of release-acquire is therefore simple: you acquire a location of memory to begin the critical section, and then release that location to end it. For instance, a simple spinlock might look like:

```
use std::sync::Arc;
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

fn main() {
    let lock = Arc::new(AtomicBool::new(false)); // value answers "am I locked?"

    // ... distribute lock to threads somehow ...

    // Try to acquire the lock by setting it to true
    while lock.compare_and_swap(false, true, Ordering::Acquire) { }
    // broke out of the loop, so we successfully acquired the lock!

    // ... scary data accesses ...

    // ok we're done, release the lock
    lock.store(false, Ordering::Release);
}
```

On strongly-ordered platforms most accesses have release or acquire semantics, making release and acquire often totally free. This is not the case on weakly-ordered platforms.

16.3.6 Relaxed

Relaxed accesses are the absolute weakest. They can be freely re-ordered and provide no happens-before relationship. Still, relaxed operations are still atomic. That is, they don't count as data accesses and any read-modify-write operations done to them occur atomically. Relaxed operations are appropriate for things that you definitely want to happen, but don't particularly otherwise care about. For instance, incrementing a counter can be safely done by multiple threads using a relaxed `fetch_add` if you're not using the counter to synchronize any other accesses.

There's rarely a benefit in making an operation relaxed on strongly-ordered platforms, since they usually provide release-acquire semantics anyway. However relaxed operations can be cheaper on weakly-ordered platforms.

Chapter 17

Implementing Vec

To bring everything together, we're going to write `std::Vec` from scratch. Because all the best tools for writing unsafe code are unstable, this project will only work on nightly (as of Rust 1.2.0). With the exception of the allocator API, much of the unstable code we'll use is expected to be stabilized in a similar form as it is today.

However we will generally try to avoid unstable code where possible. In particular we won't use any intrinsics that could make a code a little bit nicer or efficient because intrinsics are permanently unstable. Although many intrinsics *do* become stabilized elsewhere (`std::ptr` and `str::mem` consist of many intrinsics).

Ultimately this means our implementation may not take advantage of all possible optimizations, though it will be by no means *naive*. We will definitely get into the weeds over nitty-gritty details, even when the problem doesn't *really* merit it.

You wanted advanced. We're gonna go advanced.

17.1 Layout

First off, we need to come up with the struct layout. A Vec has three parts: a pointer to the allocation, the size of the allocation, and the number of elements that have been initialized.

Naively, this means we just want this design:

```
pub struct Vec<T> {
    ptr: *mut T,
    cap: usize,
    len: usize,
}
# fn main() {}
```

And indeed this would compile. Unfortunately, it would be incorrect. First, the compiler will give us too strict variance. So a `&Vec<&'static str>` couldn't be used where an `&Vec<&'a str>` was expected. More importantly, it will give incorrect ownership information to the drop checker, as it will conservatively assume we don't own any

values of type `T`. See the chapter on ownership and lifetimes[1] for all the details on variance and drop check.

As we saw in the ownership chapter, we should use `Unique<T>` in place of `*mut T` when we have a raw pointer to an allocation we own. Unique is unstable, so we'd like to not use it if possible, though.

As a recap, Unique is a wrapper around a raw pointer that declares that:

- We are variant over `T`
- We may own a value of type `T` (for drop check)
- We are Send/Sync if `T` is Send/Sync
- We deref to `*mut T` (so it largely acts like a `*mut` in our code)
- Our pointer is never null (so `Option<Vec<T>>` is null-pointer-optimized)

We can implement all of the above requirements except for the last one in stable Rust:

```
use std::marker::PhantomData;
use std::ops::Deref;
use std::mem;

struct Unique<T> {
    ptr: *const T,              // *const for variance
    _marker: PhantomData<T>,    // For the drop checker
}

// Deriving Send and Sync is safe because we are the Unique owners
// of this data. It's like Unique<T> is "just" T.
unsafe impl<T: Send> Send for Unique<T> {}
unsafe impl<T: Sync> Sync for Unique<T> {}

impl<T> Unique<T> {
    pub fn new(ptr: *mut T) -> Self {
        Unique { ptr: ptr, _marker: PhantomData }
    }
}

impl<T> Deref for Unique<T> {
    type Target = *mut T;
    fn deref(&self) -> &*mut T {
        // There's no way to cast the *const to a *mut
        // while also taking a reference. So we just
        // transmute it since it's all "just pointers".
        unsafe { mem::transmute(&self.ptr) }
    }
}
# fn main() {}
```

[1] `ownership.html`

Unfortunately the mechanism for stating that your value is non-zero is unstable and unlikely to be stabilized soon. As such we're just going to take the hit and use std's Unique:

```
#![feature(unique)]

use std::ptr::{Unique, self};

pub struct Vec<T> {
    ptr: Unique<T>,
    cap: usize,
    len: usize,
}

# fn main() {}
```

If you don't care about the null-pointer optimization, then you can use the stable code. However we will be designing the rest of the code around enabling the optimization. In particular, `Unique::new` is unsafe to call, because putting `null` inside of it is Undefined Behavior. Our stable Unique doesn't need `new` to be unsafe because it doesn't make any interesting guarantees about its contents.

17.2 Allocating

Using Unique throws a wrench in an important feature of Vec (and indeed all of the std collections): an empty Vec doesn't actually allocate at all. So if we can't allocate, but also can't put a null pointer in `ptr`, what do we do in `Vec::new`? Well, we just put some other garbage in there!

This is perfectly fine because we already have `cap == 0` as our sentinel for no allocation. We don't even need to handle it specially in almost any code because we usually need to check if `cap > len` or `len > 0` anyway. The traditional Rust value to put here is `0x01`. The standard library actually exposes this as `alloc::heap::EMPTY`. There are quite a few places where we'll want to use `heap::EMPTY` because there's no real allocation to talk about but `null` would make the compiler do bad things.

All of the `heap` API is totally unstable under the `heap_api` feature, though. We could trivially define `heap::EMPTY` ourselves, but we'll want the rest of the `heap` API anyway, so let's just get that dependency over with.

So:

```
#![feature(alloc, heap_api)]

use std::mem;

use alloc::heap::EMPTY;

impl<T> Vec<T> {
    fn new() -> Self {
```

```
        assert!(mem::size_of::<T>() != 0, "We're not ready to handle ZSTs");
        unsafe {
            // need to cast EMPTY to the actual ptr type we want, let
            // inference handle it.
            Vec { ptr: Unique::new(heap::EMPTY as *mut _), len: 0, cap: 0 }
        }
    }
}
```

I slipped in that assert there because zero-sized types will require some special handling throughout our code, and I want to defer the issue for now. Without this assert, some of our early drafts will do some Very Bad Things.

Next we need to figure out what to actually do when we *do* want space. For that, we'll need to use the rest of the heap APIs. These basically allow us to talk directly to Rust's allocator (jemalloc by default).

We'll also need a way to handle out-of-memory (OOM) conditions. The standard library calls the `abort` intrinsic, which just calls an illegal instruction to crash the whole program. The reason we abort and don't panic is because unwinding can cause allocations to happen, and that seems like a bad thing to do when your allocator just came back with "hey I don't have any more memory".

Of course, this is a bit silly since most platforms don't actually run out of memory in a conventional way. Your operating system will probably kill the application by another means if you legitimately start using up all the memory. The most likely way we'll trigger OOM is by just asking for ludicrous quantities of memory at once (e.g. half the theoretical address space). As such it's *probably* fine to panic and nothing bad will happen. Still, we're trying to be like the standard library as much as possible, so we'll just kill the whole program.

We said we don't want to use intrinsics, so doing exactly what `std` does is out. Instead, we'll call `std::process::exit` with some random number.

```
fn oom() {
    ::std::process::exit(-9999);
}
```

Okay, now we can write growing. Roughly, we want to have this logic:

```
if cap == 0:
    allocate()
    cap = 1
else:
    reallocate()
    cap *= 2
```

But Rust's only supported allocator API is so low level that we'll need to do a fair bit of extra work. We also need to guard against some special conditions that can occur with really large allocations or empty allocations.

In particular, `ptr::offset` will cause us a lot of trouble, because it has the semantics of LLVM's GEP inbounds instruction. If you're fortunate enough to not have dealt with

this instruction, here's the basic story with GEP: alias analysis, alias analysis, alias analysis. It's super important to an optimizing compiler to be able to reason about data dependencies and aliasing.

As a simple example, consider the following fragment of code:

```
# let x = &mut 0;
# let y = &mut 0;
*x *= 7;
*y *= 3;
```

If the compiler can prove that `x` and `y` point to different locations in memory, the two operations can in theory be executed in parallel (by e.g. loading them into different registers and working on them independently). However the compiler can't do this in general because if x and y point to the same location in memory, the operations need to be done to the same value, and they can't just be merged afterwards.

When you use GEP inbounds, you are specifically telling LLVM that the offsets you're about to do are within the bounds of a single "allocated" entity. The ultimate payoff being that LLVM can assume that if two pointers are known to point to two disjoint objects, all the offsets of those pointers are *also* known to not alias (because you won't just end up in some random place in memory). LLVM is heavily optimized to work with GEP offsets, and inbounds offsets are the best of all, so it's important that we use them as much as possible.

So that's what GEP's about, how can it cause us trouble?

The first problem is that we index into arrays with unsigned integers, but GEP (and as a consequence `ptr::offset`) takes a signed integer. This means that half of the seemingly valid indices into an array will overflow GEP and actually go in the wrong direction! As such we must limit all allocations to `isize::MAX` elements. This actually means we only need to worry about byte-sized objects, because e.g. `> isize::MAX u16`s will truly exhaust all of the system's memory. However in order to avoid subtle corner cases where someone reinterprets some array of `< isize::MAX` objects as bytes, std limits all allocations to `isize::MAX` bytes.

On all 64-bit targets that Rust currently supports we're artificially limited to significantly less than all 64 bits of the address space (modern x64 platforms only expose 48-bit addressing), so we can rely on just running out of memory first. However on 32-bit targets, particularly those with extensions to use more of the address space (PAE x86 or x32), it's theoretically possible to successfully allocate more than `isize::MAX` bytes of memory.

However since this is a tutorial, we're not going to be particularly optimal here, and just unconditionally check, rather than use clever platform-specific `cfg`s.

The other corner-case we need to worry about is empty allocations. There will be two kinds of empty allocations we need to worry about: `cap = 0` for all T, and `cap > 0` for zero-sized types.

These cases are tricky because they come down to what LLVM means by "allocated". LLVM's notion of an allocation is significantly more abstract than how we usually use it. Because LLVM needs to work with different languages' semantics and custom allocators, it can't really intimately understand allocation. Instead, the main idea behind allocation is "doesn't overlap with other stuff". That is, heap allocations, stack

allocations, and globals don't randomly overlap. Yep, it's about alias analysis. As such, Rust can technically play a bit fast an loose with the notion of an allocation as long as it's *consistent*.

Getting back to the empty allocation case, there are a couple of places where we want to offset by 0 as a consequence of generic code. The question is then: is it consistent to do so? For zero-sized types, we have concluded that it is indeed consistent to do a GEP inbounds offset by an arbitrary number of elements. This is a runtime no-op because every element takes up no space, and it's fine to pretend that there's infinite zero-sized types allocated at `0x01`. No allocator will ever allocate that address, because they won't allocate `0x00` and they generally allocate to some minimal alignment higher than a byte. Also generally the whole first page of memory is protected from being allocated anyway (a whole 4k, on many platforms).

However what about for positive-sized types? That one's a bit trickier. In principle, you can argue that offsetting by 0 gives LLVM no information: either there's an element before the address or after it, but it can't know which. However we've chosen to conservatively assume that it may do bad things. As such we will guard against this case explicitly.

Phew

Ok with all the nonsense out of the way, let's actually allocate some memory:

```
fn grow(&mut self) {
    // this is all pretty delicate, so let's say it's all unsafe
    unsafe {
        // current API requires us to specify size and alignment manually.
        let align = mem::align_of::<T>();
        let elem_size = mem::size_of::<T>();

        let (new_cap, ptr) = if self.cap == 0 {
            let ptr = heap::allocate(elem_size, align);
            (1, ptr)
        } else {
            // as an invariant, we can assume that `self.cap < isize::MAX`,
            // so this doesn't need to be checked.
            let new_cap = self.cap * 2;
            // Similarly this can't overflow due to previously allocating this
            let old_num_bytes = self.cap * elem_size;

            // check that the new allocation doesn't exceed `isize::MAX` at all
            // regardless of the actual size of the capacity. This combines the
            // `new_cap <= isize::MAX` and `new_num_bytes <= usize::MAX` checks
            // we need to make. We lose the ability to allocate e.g. 2/3rds of
            // the address space with a single Vec of i16's on 32-bit though.
            // Alas, poor Yorick -- I knew him, Horatio.
            assert!(old_num_bytes <= (::std::isize::MAX as usize) / 2,
                    "capacity overflow");

            let new_num_bytes = old_num_bytes * 2;
            let ptr = heap::reallocate(*self.ptr as *mut _,
```

```
                                        old_num_bytes,
                                        new_num_bytes,
                                        align);
            (new_cap, ptr)
        };

        // If allocate or reallocate fail, we'll get `null` back
        if ptr.is_null() { oom(); }

        self.ptr = Unique::new(ptr as *mut _);
        self.cap = new_cap;
    }
}
```

Nothing particularly tricky here. Just computing sizes and alignments and doing some careful multiplication checks.

17.3 Push and Pop

Alright. We can initialize. We can allocate. Let's actually implement some functionality! Let's start with `push`. All it needs to do is check if we're full to grow, unconditionally write to the next index, and then increment our length.

To do the write we have to be careful not to evaluate the memory we want to write to. At worst, it's truly uninitialized memory from the allocator. At best it's the bits of some old value we popped off. Either way, we can't just index to the memory and dereference it, because that will evaluate the memory as a valid instance of T. Worse, `foo[idx] = x` will try to call `drop` on the old value of `foo[idx]`!

The correct way to do this is with `ptr::write`, which just blindly overwrites the target address with the bits of the value we provide. No evaluation involved.

For `push`, if the old len (before push was called) is 0, then we want to write to the 0th index. So we should offset by the old len.

```
pub fn push(&mut self, elem: T) {
    if self.len == self.cap { self.grow(); }

    unsafe {
        ptr::write(self.ptr.offset(self.len as isize), elem);
    }

    // Can't fail, we'll OOM first.
    self.len += 1;
}
```

Easy! How about `pop`? Although this time the index we want to access is initialized, Rust won't just let us dereference the location of memory to move the value out, because that would leave the memory uninitialized! For this we need `ptr::read`, which just copies out the bits from the target address and interprets it as a value of

type T. This will leave the memory at this address logically uninitialized, even though there is in fact a perfectly good instance of T there.

For pop, if the old len is 1, we want to read out of the 0th index. So we should offset by the new len.

```
pub fn pop(&mut self) -> Option<T> {
    if self.len == 0 {
        None
    } else {
        self.len -= 1;
        unsafe {
            Some(ptr::read(self.ptr.offset(self.len as isize)))
        }
    }
}
```

17.4 Deallocating

Next we should implement Drop so that we don't massively leak tons of resources. The easiest way is to just call pop until it yields None, and then deallocate our buffer. Note that calling pop is unneeded if `T: !Drop`. In theory we can ask Rust if `T needs_drop` and omit the calls to pop. However in practice LLVM is *really* good at removing simple side-effect free code like this, so I wouldn't bother unless you notice it's not being stripped (in this case it is).

We must not call `heap::deallocate` when `self.cap == 0`, as in this case we haven't actually allocated any memory.

```
impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            while let Some(_) = self.pop() { }

            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();
            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.ptr, num_bytes, align);
            }
        }
    }
}
```

17.5 Deref

Alright! We've got a decent minimal stack implemented. We can push, we can pop, and we can clean up after ourselves. However there's a whole mess of functionality

we'd reasonably want. In particular, we have a proper array, but none of the slice functionality. That's actually pretty easy to solve: we can implement `Deref<Target=[T]>`. This will magically make our Vec coerce to, and behave like, a slice in all sorts of conditions.

All we need is `slice::from_raw_parts`. It will correctly handle empty slices for us. Later once we set up zero-sized type support it will also Just Work for those too.

```
use std::ops::Deref;

impl<T> Deref for Vec<T> {
    type Target = [T];
    fn deref(&self) -> &[T] {
        unsafe {
            ::std::slice::from_raw_parts(*self.ptr, self.len)
        }
    }
}
```

And let's do DerefMut too:

```
use std::ops::DerefMut;

impl<T> DerefMut for Vec<T> {
    fn deref_mut(&mut self) -> &mut [T] {
        unsafe {
            ::std::slice::from_raw_parts_mut(*self.ptr, self.len)
        }
    }
}
```

Now we have `len`, `first`, `last`, indexing, slicing, sorting, `iter`, `iter_mut`, and all other sorts of bells and whistles provided by slice. Sweet!

17.6 Insert and Remove

Something *not* provided by slice is `insert` and `remove`, so let's do those next.

Insert needs to shift all the elements at the target index to the right by one. To do this we need to use `ptr::copy`, which is our version of C's `memmove`. This copies some chunk of memory from one location to another, correctly handling the case where the source and destination overlap (which will definitely happen here).

If we insert at index `i`, we want to shift the `[i .. len]` to `[i+1 .. len+1]` using the old len.

```
pub fn insert(&mut self, index: usize, elem: T) {
    // Note: '<=' because it's valid to insert after everything
    // which would be equivalent to push.
    assert!(index <= self.len, "index out of bounds");
```

```
    if self.cap == self.len { self.grow(); }

    unsafe {
        if index < self.len {
            // ptr::copy(src, dest, len): "copy from source to dest len elems"
            ptr::copy(self.ptr.offset(index as isize),
                      self.ptr.offset(index as isize + 1),
                      len - index);
        }
        ptr::write(self.ptr.offset(index as isize), elem);
        self.len += 1;
    }
}
```

Remove behaves in the opposite manner. We need to shift all the elements from `[i+1 .. len + 1]` to `[i .. len]` using the *new* len.

```
pub fn remove(&mut self, index: usize) -> T {
    // Note: '<' because it's *not* valid to remove after everything
    assert!(index < self.len, "index out of bounds");
    unsafe {
        self.len -= 1;
        let result = ptr::read(self.ptr.offset(index as isize));
        ptr::copy(self.ptr.offset(index as isize + 1),
                  self.ptr.offset(index as isize),
                  len - index);
        result
    }
}
```

17.7 IntoIter

Let's move on to writing iterators. `iter` and `iter_mut` have already been written for us thanks to The Magic of Deref. However there's two interesting iterators that Vec provides that slices can't: `into_iter` and `drain`.

IntoIter consumes the Vec by-value, and can consequently yield its elements by-value. In order to enable this, IntoIter needs to take control of Vec's allocation.

IntoIter needs to be DoubleEnded as well, to enable reading from both ends. Reading from the back could just be implemented as calling `pop`, but reading from the front is harder. We could call `remove(0)` but that would be insanely expensive. Instead we're going to just use ptr::read to copy values out of either end of the Vec without mutating the buffer at all.

To do this we're going to use a very common C idiom for array iteration. We'll make two pointers; one that points to the start of the array, and one that points to one-element past the end. When we want an element from one end, we'll read out the value pointed to at that end and move the pointer over by one. When the two pointers are equal, we know we're done.

Note that the order of read and offset are reversed for `next` and `next_back` For `next_back` the pointer is always after the element it wants to read next, while for `next` the pointer is always at the element it wants to read next. To see why this is, consider the case where every element but one has been yielded.

The array looks like this:

```
          S  E
[X, X, X, O, X, X, X]
```

If E pointed directly at the element it wanted to yield next, it would be indistinguishable from the case where there are no more elements to yield.

Although we don't actually care about it during iteration, we also need to hold onto the Vec's allocation information in order to free it once IntoIter is dropped.

So we're going to use the following struct:

```
struct IntoIter<T> {
    buf: Unique<T>,
    cap: usize,
    start: *const T,
    end: *const T,
}
```

And this is what we end up with for initialization:

```
impl<T> Vec<T> {
    fn into_iter(self) -> IntoIter<T> {
        // Can't destructure Vec since it's Drop
        let ptr = self.ptr;
        let cap = self.cap;
        let len = self.len;

        // Make sure not to drop Vec since that will free the buffer
        mem::forget(self);

        unsafe {
            IntoIter {
                buf: ptr,
                cap: cap,
                start: *ptr,
                end: if cap == 0 {
                    // can't offset off this pointer, it's not allocated!
                    *ptr
                } else {
                    ptr.offset(len as isize)
                }
            }
        }
    }
}
```

Here's iterating forward:

```
impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                let result = ptr::read(self.start);
                self.start = self.start.offset(1);
                Some(result)
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let len = (self.end as usize - self.start as usize)
                  / mem::size_of::<T>();
        (len, Some(len))
    }
}
```

And here's iterating backwards.

```
impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                self.end = self.end.offset(-1);
                Some(ptr::read(self.end))
            }
        }
    }
}
```

Because IntoIter takes ownership of its allocation, it needs to implement Drop to free it. However it also wants to implement Drop to drop any elements it contains that weren't yielded.

```
impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            // drop any remaining elements
            for _ in &mut *self {}

            let align = mem::align_of::<T>();
```

```
            let elem_size = mem::size_of::<T>();
            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.buf as *mut _, num_bytes, align);
            }
        }
    }
}
```

17.8 RawVec

We've actually reached an interesting situation here: we've duplicated the logic for specifying a buffer and freeing its memory in Vec and IntoIter. Now that we've implemented it and identified *actual* logic duplication, this is a good time to perform some logic compression.

We're going to abstract out the `(ptr, cap)` pair and give them the logic for allocating, growing, and freeing:

```
struct RawVec<T> {
    ptr: Unique<T>,
    cap: usize,
}

impl<T> RawVec<T> {
    fn new() -> Self {
        assert!(mem::size_of::<T>() != 0, "TODO: implement ZST support");
        unsafe {
            RawVec { ptr: Unique::new(heap::EMPTY as *mut T), cap: 0 }
        }
    }

    // unchanged from Vec
    fn grow(&mut self) {
        unsafe {
            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();

            let (new_cap, ptr) = if self.cap == 0 {
                let ptr = heap::allocate(elem_size, align);
                (1, ptr)
            } else {
                let new_cap = 2 * self.cap;
                let ptr = heap::reallocate(*self.ptr as *mut _,
                                           self.cap * elem_size,
                                           new_cap * elem_size,
                                           align);
                (new_cap, ptr)
```

```
            };

            // If allocate or reallocate fail, we'll get `null` back
            if ptr.is_null() { oom() }

            self.ptr = Unique::new(ptr as *mut _);
            self.cap = new_cap;
        }
    }
}


impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();
            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.ptr as *mut _, num_bytes, align);
            }
        }
    }
}
```

And change Vec as follows:

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}

impl<T> Vec<T> {
    fn ptr(&self) -> *mut T { *self.buf.ptr }

    fn cap(&self) -> usize { self.buf.cap }

    pub fn new() -> Self {
        Vec { buf: RawVec::new(), len: 0 }
    }

    // push/pop/insert/remove largely unchanged:
    // * `self.ptr -> self.ptr()`
    // * `self.cap -> self.cap()`
    // * `self.grow -> self.buf.grow()`
}

impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
```

```
        while let Some(_) = self.pop() {}
        // deallocation is handled by RawVec
    }
}
```

And finally we can really simplify IntoIter:

```
struct IntoIter<T> {
    _buf: RawVec<T>, // we don't actually care about this. Just need it to live.
    start: *const T,
    end: *const T,
}

// next and next_back literally unchanged since they never referred to the buf

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        // only need to ensure all our elements are read;
        // buffer will clean itself up afterwards.
        for _ in &mut *self {}
    }
}

impl<T> Vec<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        unsafe {
            // need to use ptr::read to unsafely move the buf out since it's
            // not Copy, and Vec implements Drop (so we can't destructure it).
            let buf = ptr::read(&self.buf);
            let len = self.len;
            mem::forget(self);

            IntoIter {
                start: *buf.ptr,
                end: buf.ptr.offset(len as isize),
                _buf: buf,
            }
        }
    }
}
```

Much better.

17.9 Drain

Let's move on to Drain. Drain is largely the same as IntoIter, except that instead of consuming the Vec, it borrows the Vec and leaves its allocation untouched. For now we'll only implement the "basic" full-range version.

```
use std::marker::PhantomData;

struct Drain<'a, T: 'a> {
    // Need to bound the lifetime here, so we do it with `&'a mut Vec<T>`
    // because that's semantically what we contain. We're "just" calling
    // `pop()` and `remove(0)`.
    vec: PhantomData<&'a mut Vec<T>>
    start: *const T,
    end: *const T,
}

impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
```

– wait, this is seeming familiar. Let's do some more compression. Both IntoIter and Drain have the exact same structure, let's just factor it out.

```
struct RawValIter<T> {
    start: *const T,
    end: *const T,
}

impl<T> RawValIter<T> {
    // unsafe to construct because it has no associated lifetimes.
    // This is necessary to store a RawValIter in the same struct as
    // its actual allocation. OK since it's a private implementation
    // detail.
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if slice.len() == 0 {
                // if `len = 0`, then this is not actually allocated memory.
                // Need to avoid offsetting because that will give wrong
                // information to LLVM via GEP.
                slice.as_ptr()
            } else {
                slice.as_ptr().offset(slice.len() as isize)
            }
        }
    }
}

// Iterator and DoubleEndedIterator impls identical to IntoIter.
```

And IntoIter becomes the following:

```
pub struct IntoIter<T> {
    _buf: RawVec<T>, // we don't actually care about this. Just need it to live.
    iter: RawValIter<T>,
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

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

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        for _ in &mut self.iter {}
    }
}

impl<T> Vec<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            let buf = ptr::read(&self.buf);
            mem::forget(self);

            IntoIter {
                iter: iter,
                _buf: buf,
            }
        }
    }
}
```

Note that I've left a few quirks in this design to make upgrading Drain to work with arbitrary subranges a bit easier. In particular we *could* have RawValIter drain itself on drop, but that won't work right for a more complex Drain. We also take a slice to simplify Drain initialization.

Alright, now Drain is really easy:

```
use std::marker::PhantomData;

pub struct Drain<'a, T: 'a> {
    vec: PhantomData<&'a mut Vec<T>>,
    iter: RawValIter<T>,
}
```

```
impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

impl<'a, T> DoubleEndedIterator for Drain<'a, T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<'a, T> Drop for Drain<'a, T> {
    fn drop(&mut self) {
        for _ in &mut self.iter {}
    }
}

impl<T> Vec<T> {
    pub fn drain(&mut self) -> Drain<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            // this is a mem::forget safety thing. If Drain is forgotten, we just
            // leak the whole Vec's contents. Also we need to do this *eventually*
            // anyway, so why not do it now?
            self.len = 0;

            Drain {
                iter: iter,
                vec: PhantomData,
            }
        }
    }
}
```

For more details on the `mem::forget` problem, see the section on leaks[2].

17.10 Handling Zero-Sized Types

It's time. We're going to fight the specter that is zero-sized types. Safe Rust *never* needs to care about this, but Vec is very intensive on raw pointers and raw allocations, which are exactly the two things that care about zero-sized types. We need to be careful of two things:

- The raw allocator API has undefined behavior if you pass in 0 for an allocation size.

[2] `leaking.html`

- raw pointer offsets are no-ops for zero-sized types, which will break our C-style pointer iterator.

Thankfully we abstracted out pointer-iterators and allocating handling into RawValIter and RawVec respectively. How mysteriously convenient.

Allocating Zero-Sized Types

So if the allocator API doesn't support zero-sized allocations, what on earth do we store as our allocation? Why, `heap::EMPTY` of course! Almost every operation with a ZST is a no-op since ZSTs have exactly one value, and therefore no state needs to be considered to store or load them. This actually extends to `ptr::read` and `ptr::write`: they won't actually look at the pointer at all. As such we never need to change the pointer.

Note however that our previous reliance on running out of memory before overflow is no longer valid with zero-sized types. We must explicitly guard against capacity overflow for zero-sized types.

Due to our current architecture, all this means is writing 3 guards, one in each method of RawVec.

```
impl<T> RawVec<T> {
    fn new() -> Self {
        unsafe {
            // !0 is usize::MAX. This branch should be stripped at compile time.
            let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

            // heap::EMPTY doubles as "unallocated" and "zero-sized allocation"
            RawVec { ptr: Unique::new(heap::EMPTY as *mut T), cap: cap }
        }
    }

    fn grow(&mut self) {
        unsafe {
            let elem_size = mem::size_of::<T>();

            // since we set the capacity to usize::MAX when elem_size is
            // 0, getting to here necessarily means the Vec is overfull.
            assert!(elem_size != 0, "capacity overflow");

            let align = mem::align_of::<T>();

            let (new_cap, ptr) = if self.cap == 0 {
                let ptr = heap::allocate(elem_size, align);
                (1, ptr)
            } else {
                let new_cap = 2 * self.cap;
                let ptr = heap::reallocate(*self.ptr as *mut _,
                                           self.cap * elem_size,
```

```
                                            new_cap * elem_size,
                                            align);
                (new_cap, ptr)
            };

            // If allocate or reallocate fail, we'll get `null` back
            if ptr.is_null() { oom() }

            self.ptr = Unique::new(ptr as *mut _);
            self.cap = new_cap;
        }
    }
}

impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        let elem_size = mem::size_of::<T>();

        // don't free zero-sized allocations, as they were never allocated.
        if self.cap != 0 && elem_size != 0 {
            let align = mem::align_of::<T>();

            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.ptr as *mut _, num_bytes, align);
            }
        }
    }
}
```

That's it. We support pushing and popping zero-sized types now. Our iterators (that aren't provided by slice Deref) are still busted, though.

Iterating Zero-Sized Types

Zero-sized offsets are no-ops. This means that our current design will always initialize `start` and `end` as the same value, and our iterators will yield nothing. The current solution to this is to cast the pointers to integers, increment, and then cast them back:

```
impl<T> RawValIter<T> {
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if mem::size_of::<T>() == 0 {
                ((slice.as_ptr() as usize) + slice.len()) as *const _
            } else if slice.len() == 0 {
                slice.as_ptr()
            } else {
```

```
                slice.as_ptr().offset(slice.len() as isize)
            }
        }
    }
}
```

Now we have a different bug. Instead of our iterators not running at all, our iterators now run *forever*. We need to do the same trick in our iterator impls. Also, our size_hint computation code will divide by 0 for ZSTs. Since we'll basically be treating the two pointers as if they point to bytes, we'll just map size 0 to divide by 1.

```
impl<T> Iterator for RawValIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                let result = ptr::read(self.start);
                self.start = if mem::size_of::<T>() == 0 {
                    (self.start as usize + 1) as *const _
                } else {
                    self.start.offset(1);
                }
                Some(result)
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let elem_size = mem::size_of::<T>();
        let len = (self.end as usize - self.start as usize)
                  / if elem_size == 0 { 1 } else { elem_size };
        (len, Some(len))
    }
}

impl<T> DoubleEndedIterator for RawValIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                self.end = if mem::size_of::<T>() == 0 {
                    (self.end as usize - 1) as *const _
                } else {
                    self.end.offset(-1);
                }
                Some(ptr::read(self.end))
```

```
            }
        }
    }
}
```

And that's it. Iteration works!

17.11 Final Code

```
#![feature(unique)]
#![feature(alloc, heap_api)]

extern crate alloc;

use std::ptr::{Unique, self};
use std::mem;
use std::ops::{Deref, DerefMut};
use std::marker::PhantomData;

use alloc::heap;

struct RawVec<T> {
    ptr: Unique<T>,
    cap: usize,
}

impl<T> RawVec<T> {
    fn new() -> Self {
        unsafe {
            // !0 is usize::MAX. This branch should be stripped at compile time.
            let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

            // heap::EMPTY doubles as "unallocated" and "zero-sized allocation"
            RawVec { ptr: Unique::new(heap::EMPTY as *mut T), cap: cap }
        }
    }

    fn grow(&mut self) {
        unsafe {
            let elem_size = mem::size_of::<T>();

            // since we set the capacity to usize::MAX when elem_size is
            // 0, getting to here necessarily means the Vec is overfull.
            assert!(elem_size != 0, "capacity overflow");

            let align = mem::align_of::<T>();

            let (new_cap, ptr) = if self.cap == 0 {
```

```
                let ptr = heap::allocate(elem_size, align);
                (1, ptr)
            } else {
                let new_cap = 2 * self.cap;
                let ptr = heap::reallocate(*self.ptr as *mut _,
                                           self.cap * elem_size,
                                           new_cap * elem_size,
                                           align);
                (new_cap, ptr)
            };

            // If allocate or reallocate fail, we'll get `null` back
            if ptr.is_null() { oom() }

            self.ptr = Unique::new(ptr as *mut _);
            self.cap = new_cap;
        }
    }
}

impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        let elem_size = mem::size_of::<T>();
        if self.cap != 0 && elem_size != 0 {
            let align = mem::align_of::<T>();

            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.ptr as *mut _, num_bytes, align);
            }
        }
    }
}
```

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}

impl<T> Vec<T> {
    fn ptr(&self) -> *mut T { *self.buf.ptr }

    fn cap(&self) -> usize { self.buf.cap }

    pub fn new() -> Self {
```

```
    Vec { buf: RawVec::new(), len: 0 }
}
pub fn push(&mut self, elem: T) {
    if self.len == self.cap() { self.buf.grow(); }

    unsafe {
        ptr::write(self.ptr().offset(self.len as isize), elem);
    }

    // Can't fail, we'll OOM first.
    self.len += 1;
}

pub fn pop(&mut self) -> Option<T> {
    if self.len == 0 {
        None
    } else {
        self.len -= 1;
        unsafe {
            Some(ptr::read(self.ptr().offset(self.len as isize)))
        }
    }
}

pub fn insert(&mut self, index: usize, elem: T) {
    assert!(index <= self.len, "index out of bounds");
    if self.cap() == self.len { self.buf.grow(); }

    unsafe {
        if index < self.len {
            ptr::copy(self.ptr().offset(index as isize),
                      self.ptr().offset(index as isize + 1),
                      self.len - index);
        }
        ptr::write(self.ptr().offset(index as isize), elem);
        self.len += 1;
    }
}

pub fn remove(&mut self, index: usize) -> T {
    assert!(index < self.len, "index out of bounds");
    unsafe {
        self.len -= 1;
        let result = ptr::read(self.ptr().offset(index as isize));
        ptr::copy(self.ptr().offset(index as isize + 1),
                  self.ptr().offset(index as isize),
                  self.len - index);
        result
    }
```

```
    }

    pub fn into_iter(self) -> IntoIter<T> {
        unsafe {
            let iter = RawValIter::new(&self);
            let buf = ptr::read(&self.buf);
            mem::forget(self);

            IntoIter {
                iter: iter,
                _buf: buf,
            }
        }
    }

    pub fn drain(&mut self) -> Drain<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            // this is a mem::forget safety thing. If Drain is forgotten, we just
            // leak the whole Vec's contents. Also we need to do this *eventually*
            // anyway, so why not do it now?
            self.len = 0;

            Drain {
                iter: iter,
                vec: PhantomData,
            }
        }
    }
}

impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() {}
        // allocation is handled by RawVec
    }
}

impl<T> Deref for Vec<T> {
    type Target = [T];
    fn deref(&self) -> &[T] {
        unsafe {
            ::std::slice::from_raw_parts(self.ptr(), self.len)
        }
    }
}

impl<T> DerefMut for Vec<T> {
```

```
    fn deref_mut(&mut self) -> &mut [T] {
        unsafe {
            ::std::slice::from_raw_parts_mut(self.ptr(), self.len)
        }
    }
}
```

```
struct RawValIter<T> {
    start: *const T,
    end: *const T,
}

impl<T> RawValIter<T> {
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if mem::size_of::<T>() == 0 {
                ((slice.as_ptr() as usize) + slice.len()) as *const _
            } else if slice.len() == 0 {
                slice.as_ptr()
            } else {
                slice.as_ptr().offset(slice.len() as isize)
            }
        }
    }
}

impl<T> Iterator for RawValIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                let result = ptr::read(self.start);
                self.start = self.start.offset(1);
                Some(result)
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let elem_size = mem::size_of::<T>();
        let len = (self.end as usize - self.start as usize)
                  / if elem_size == 0 { 1 } else { elem_size };
```

```
        (len, Some(len))
    }
}

impl<T> DoubleEndedIterator for RawValIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                self.end = self.end.offset(-1);
                Some(ptr::read(self.end))
            }
        }
    }
}
```

```
pub struct IntoIter<T> {
    _buf: RawVec<T>, // we don't actually care about this. Just need it to live.
    iter: RawValIter<T>,
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

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

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        for _ in &mut *self {}
    }
}
```

```
pub struct Drain<'a, T: 'a> {
    vec: PhantomData<&'a mut Vec<T>>,
    iter: RawValIter<T>,
}
```

```
impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next_back() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

impl<'a, T> DoubleEndedIterator for Drain<'a, T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<'a, T> Drop for Drain<'a, T> {
    fn drop(&mut self) {
        // pre-drain the iter
        for _ in &mut self.iter {}
    }
}

/// Abort the process, we're out of memory!
///
/// In practice this is probably dead code on most OSes
fn oom() {
    ::std::process::exit(-9999);
}

# fn main() {}
```

Chapter 18

Implementing Arc and Mutex

Knowing the theory is all fine and good, but the *best* way to understand something is to use it. To better understand atomics and interior mutability, we'll be implementing versions of the standard library's Arc and Mutex types.

TODO: ALL OF THIS OMG

Part III

Language Reference

Chapter 19

Grammar

19.1 Introduction

This document is the primary reference for the Rust programming language grammar. It provides only one kind of material:

- Chapters that formally define the language grammar.

This document does not serve as an introduction to the language. Background familiarity with the language is assumed. A separate guide[1] is available to help acquire such background.

This document also does not serve as a reference to the standard[2] library included in the language distribution. Those libraries are documented separately by extracting documentation attributes from their source code. Many of the features that one might expect to be language features are library features in Rust, so what you're looking for may be there, not here.

19.2 Notation

Rust's grammar is defined over Unicode codepoints, each conventionally denoted `U+XXXX`, for 4 or more hexadecimal digits `X`. *Most* of Rust's grammar is confined to the ASCII range of Unicode, and is described in this document by a dialect of Extended Backus-Naur Form (EBNF), specifically a dialect of EBNF supported by common automated LL(k) parsing tools such as `llgen`, rather than the dialect given in ISO 14977. The dialect can be defined self-referentially as follows:

```
grammar : rule + ;
rule    : nonterminal ':' productionrule ';' ;
productionrule : production [ '|' production ] * ;
production : term * ;
```

[1] `guide.html`
[2] `std/index.html`

```
term : element repeats ;
element : LITERAL | IDENTIFIER | '[' productionrule ']' ;
repeats : [ '*' | '+' ] NUMBER ? | NUMBER ? | '?' ;
```

Where:

- Whitespace in the grammar is ignored.
- Square brackets are used to group rules.
- `LITERAL` is a single printable ASCII character, or an escaped hexadecimal ASCII code of the form `\xQQ`, in single quotes, denoting the corresponding Unicode codepoint `U+00QQ`.
- `IDENTIFIER` is a nonempty string of ASCII letters and underscores.
- The `repeat` forms apply to the adjacent `element`, and are as follows:
- `?` means zero or one repetition
- `*` means zero or more repetitions
- `+` means one or more repetitions
- NUMBER trailing a repeat symbol gives a maximum repetition count
- NUMBER on its own gives an exact repetition count

This EBNF dialect should hopefully be familiar to many readers.

19.2.1 Unicode productions

A few productions in Rust's grammar permit Unicode codepoints outside the ASCII range. We define these productions in terms of character properties specified in the Unicode standard, rather than in terms of ASCII-range codepoints. The section Special Unicode Productions lists these productions.

19.2.2 String table productions

Some rules in the grammar — notably unary operators, binary operators, and keywords — are given in a simplified form: as a listing of a table of unquoted, printable whitespace-separated strings. These cases form a subset of the rules regarding the token rule, and are assumed to be the result of a lexical-analysis phase feeding the parser, driven by a DFA, operating over the disjunction of all such string table entries.

When such a string enclosed in double-quotes (") occurs inside the grammar, it is an implicit reference to a single member of such a string table production. See tokens for more information.

19.3 Lexical structure

19.3.1 Input format

Rust input is interpreted as a sequence of Unicode codepoints encoded in UTF-8. Most Rust grammar rules are defined in terms of printable ASCII-range codepoints,

but a small number are defined in terms of Unicode properties or explicit codepoint lists. [3]

19.3.2 Special Unicode Productions

The following productions in the Rust grammar are defined in terms of Unicode properties: `ident`, `non_null`, `non_eol`, `non_single_quote` and `non_double_quote`.

Identifiers

The `ident` production is any nonempty Unicode[4] string of the following form:

- The first character has property `XID_start`
- The remaining characters have property `XID_continue`

that does *not* occur in the set of keywords.

> **Note**: `XID_start` and `XID_continue` as character properties cover the character ranges used to form the more familiar C and Java language-family identifiers.

Delimiter-restricted productions

Some productions are defined by exclusion of particular Unicode characters:

- `non_null` is any single Unicode character aside from `U+0000` (null)
- `non_eol` is `non_null` restricted to exclude `U+000A` (`'\n'`)
- `non_single_quote` is `non_null` restricted to exclude `U+0027` (`'`)
- `non_double_quote` is `non_null` restricted to exclude `U+0022` (`"`)

19.3.3 Comments

```
comment : block_comment | line_comment ;
block_comment : "/*" block_comment_body * "*/" ;
block_comment_body : [block_comment | character] * ;
line_comment : "//" non_eol * ;
```

FIXME: add doc grammar?

19.3.4 Whitespace

```
whitespace_char : '\x20' | '\x09' | '\x0a' | '\x0d' ;
whitespace : [ whitespace_char | comment ] + ;
```

[3] Substitute definitions for the special Unicode productions are provided to the grammar verifier, restricted to ASCII range, when verifying the grammar in this document.

[4] Non-ASCII characters in identifiers are currently feature gated. This is expected to improve soon.

19.3.5 Tokens

```
simple_token : keyword | unop | binop ;
token : simple_token | ident | literal | symbol | whitespace token ;
```

Keywords

abstract	alignof	as	become	box
break	const	continue	crate	do
else	enum	extern	false	final
fn	for	if	impl	in
let	loop	macro	match	mod
move	mut	offsetof	override	priv
proc	pub	pure	ref	return
Self	self	sizeof	static	struct
super	trait	true	type	typeof
unsafe	unsized	use	virtual	where
while	yield			

Each of these keywords has special meaning in its grammar, and all of them are excluded from the `ident` rule.

Literals

```
lit_suffix : ident;
literal : [ string_lit | char_lit | byte_string_lit | byte_lit | num_lit | bool_lit ] lit_suff
```

The optional `lit_suffix` production is only used for certain numeric literals, but is reserved for future extension. That is, the above gives the lexical grammar, but a Rust parser will reject everything but the 12 special cases mentioned in Number literals[5] in the reference.

Character and string literals

```
char_lit : '\x27' char_body '\x27' ;
string_lit : '"' string_body * '"' | 'r' raw_string ;

char_body : non_single_quote
          | '\x5c' [ '\x27' | common_escape | unicode_escape ] ;

string_body : non_double_quote
            | '\x5c' [ '\x22' | common_escape | unicode_escape ] ;
raw_string : '"' raw_string_body '"' | '#' raw_string '#' ;

common_escape : '\x5c'
```

[5] `reference.html#number-literals`

```
              | 'n' | 'r' | 't' | '0'
              | 'x' hex_digit 2
unicode_escape : 'u' '{' hex_digit+ 6 '}';

hex_digit : 'a' | 'b' | 'c' | 'd' | 'e' | 'f'
          | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
          | dec_digit ;
oct_digit : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' ;
dec_digit : '0' | nonzero_dec ;
nonzero_dec: '1' | '2' | '3' | '4'
           | '5' | '6' | '7' | '8' | '9' ;
```

Byte and byte string literals

```
byte_lit : "b\x27" byte_body '\x27' ;
byte_string_lit : "b\x22" string_body * '\x22' | "br" raw_byte_string ;

byte_body : ascii_non_single_quote
          | '\x5c' [ '\x27' | common_escape ] ;

byte_string_body : ascii_non_double_quote
            | '\x5c' [ '\x22' | common_escape ] ;
raw_byte_string : '"' raw_byte_string_body '"' | '#' raw_byte_string '#' ;
```

Number literals

```
num_lit : nonzero_dec [ dec_digit | '_' ] * float_suffix ?
        | '0' [       [ dec_digit | '_' ] * float_suffix ?
              | 'b'   [ '1' | '0' | '_' ] +
              | 'o'   [ oct_digit | '_' ] +
              | 'x'   [ hex_digit | '_' ] +  ] ;

float_suffix : [ exponent | '.' dec_lit exponent ? ] ? ;

exponent : ['E' | 'e'] ['-' | '+' ] ? dec_lit ;
dec_lit : [ dec_digit | '_' ] + ;
```

Boolean literals

```
bool_lit : [ "true" | "false" ] ;
```

The two values of the boolean type are written `true` and `false`.

Symbols

```
symbol : "::" | "->"
       | '#' | '[' | ']' | '(' | ')' | '{' | '}'
       | ',' | ';' ;
```

Symbols are a general class of printable tokens that play structural roles in a variety of grammar productions. They are cataloged here for completeness as the set of remaining miscellaneous printable tokens that do not otherwise appear as unary operators, binary operators, or keywords.

19.3.6 Paths

```
expr_path : [ "::" ] ident [ "::" expr_path_tail ] + ;
expr_path_tail : '<' type_expr [ ',' type_expr ] + '>'
               | expr_path ;

type_path : ident [ type_path_tail ] + ;
type_path_tail : '<' type_expr [ ',' type_expr ] + '>'
               | "::" type_path ;
```

19.4 Syntax extensions

19.4.1 Macros

```
expr_macro_rules : "macro_rules" '!' ident '(' macro_rule * ')' ';'
                 | "macro_rules" '!' ident '{' macro_rule * '}' ;
macro_rule : '(' matcher * ')' "=>" '(' transcriber * ')' ';' ;
matcher : '(' matcher * ')' | '[' matcher * ']'
        | '{' matcher * '}' | '$' ident ':' ident
        | '$' '(' matcher * ')' sep_token? [ '*' | '+' ]
        | non_special_token ;
transcriber : '(' transcriber * ')' | '[' transcriber * ']'
            | '{' transcriber * '}' | '$' ident
            | '$' '(' transcriber * ')' sep_token? [ '*' | '+' ]
            | non_special_token ;
```

19.5 Crates and source files

FIXME: grammar? What production covers #[crate_id = “foo”] ?

19.6 Items and attributes

FIXME: grammar?

19.6.1 Items

```
item : vis ? mod_item | fn_item | type_item | struct_item | enum_item
     | const_item | static_item | trait_item | impl_item | extern_block_item ;
```

Type Parameters

FIXME: grammar?

Modules

```
mod_item : ”mod” ident ( ’;’ | ’{’ mod ’}’ );
mod : [ view_item | item ] * ;
```

View items

```
view_item : extern_crate_decl | use_decl ’;’ ;
```

Extern crate declarations

```
extern_crate_decl : ”extern” ”crate” crate_name
crate_name: ident | ( ident ”as” ident )
```

Use declarations

```
use_decl : vis ? ”use” [ path ”as” ident
                       | path_glob ] ;

path_glob : ident [ ”::” [ path_glob
                         | ’*’ ] ] ?
          | ’{’ path_item [ ’,’ path_item ] * ’}’ ;

path_item : ident | ”self” ;
```

Functions

FIXME: grammar?

Generic functions **FIXME:** grammar?

Unsafety **FIXME:** grammar?

Unsafe functions **FIXME:** grammar?

Unsafe blocks **FIXME:** grammar?

Diverging functions **FIXME:** grammar?

Type definitions

FIXME: grammar?

Structures

FIXME: grammar?

Enumerations

FIXME: grammar?

Constant items

```
const_item : ”const” ident ’:’ type ’=’ expr ’;’ ;
```

Static items

```
static_item : ”static” ident ’:’ type ’=’ expr ’;’ ;
```

Mutable statics

FIXME: grammar?

Traits

FIXME: grammar?

Implementations

FIXME: grammar?

External blocks

```
extern_block_item : ”extern” ’{’ extern_block ’}’ ;
extern_block : [ foreign_fn ] * ;
```

19.6.2 Visibility and Privacy

```
vis : ”pub” ;
```

Re-exporting and Visibility

See Use declarations.

19.6.3 Attributes

```
attribute : '#' '!' ? '[' meta_item ']' ;
meta_item : ident [ '=' literal
                  | '(' meta_seq ')' ] ? ;
meta_seq : meta_item [ ',' meta_seq ] ? ;
```

19.7 Statements and expressions

19.7.1 Statements

```
stmt : decl_stmt | expr_stmt | ';' ;
```

Declaration statements

```
decl_stmt : item | let_decl ;
```

Item declarations See Items.

Variable declarations

```
let_decl : "let" pat [':' type ] ? [ init ] ? ';' ;
init : [ '=' ] expr ;
```

Expression statements

```
expr_stmt : expr ';' ;
```

19.7.2 Expressions

```
expr : literal | path | tuple_expr | unit_expr | struct_expr
     | block_expr | method_call_expr | field_expr | array_expr
     | idx_expr | range_expr | unop_expr | binop_expr
     | paren_expr | call_expr | lambda_expr | while_expr
     | loop_expr | break_expr | continue_expr | for_expr
     | if_expr | match_expr | if_let_expr | while_let_expr
     | return_expr ;
```

Lvalues, rvalues and temporaries **FIXME:** grammar?

Moved and copied types **FIXME:** Do we want to capture this in the grammar as different productions?

Literal expressions

See Literals.

Path expressions

See Paths.

Tuple expressions

```
tuple_expr : '(' [ expr [ ',' expr ] * | expr ',' ] ? ')' ;
```

Unit expressions

```
unit_expr : "()" ;
```

Structure expressions

```
struct_expr : expr_path '{' ident ':' expr
                      [ ',' ident ':' expr ] *
                      [ ".." expr ] '}' |
              expr_path '(' expr
                      [ ',' expr ] * ')' |
              expr_path ;
```

Block expressions

```
block_expr : '{' [ stmt ';' | item ] *
                 [ expr ] '}' ;
```

Method-call expressions

```
method_call_expr : expr '.' ident paren_expr_list ;
```

Field expressions

```
field_expr : expr '.' ident ;
```

Array expressions

```
array_expr : '[' "mut" ? array_elems? ']' ;

array_elems : [expr [',' expr]*] | [expr ';' expr] ;
```

Index expressions

```
idx_expr : expr '[' expr ']' ;
```

Range expressions

```
range_expr : expr ".." expr |
             expr ".." |
             ".." expr |
             ".." ;
```

Unary operator expressions

```
unop_expr : unop expr ;
unop : '-' | '*' | '!' ;
```

Binary operator expressions

```
binop_expr : expr binop expr | type_cast_expr
           | assignment_expr | compound_assignment_expr ;
binop : arith_op | bitwise_op | lazy_bool_op | comp_op
```

Arithmetic operators

```
arith_op : '+' | '-' | '*' | '/' | '%' ;
```

Bitwise operators

```
bitwise_op : '&' | '|' | '^' | "<<" | ">>" ;
```

Lazy boolean operators

```
lazy_bool_op : "&&" | "||" ;
```

Comparison operators

```
comp_op : "==" | "!=" | '<' | '>' | "<=" | ">=" ;
```

Type cast expressions

```
type_cast_expr : value "as" type ;
```

Assignment expressions

```
assignment_expr : expr '=' expr ;
```

Compound assignment expressions

```
compound_assignment_expr : expr [ arith_op | bitwise_op ] '=' expr ;
```

Grouped expressions

```
paren_expr : '(' expr ')' ;
```

Call expressions

```
expr_list : [ expr [ ',' expr ]* ] ? ;
paren_expr_list : '(' expr_list ')' ;
call_expr : expr paren_expr_list ;
```

Lambda expressions

```
ident_list : [ ident [ ',' ident ]* ] ? ;
lambda_expr : '|' ident_list '|' expr ;
```

While loops

```
while_expr : [ lifetime ':' ] ? "while" no_struct_literal_expr '{' block '}' ;
```

Infinite loops

```
loop_expr : [ lifetime ':' ] ? "loop" '{' block '}';
```

Break expressions

```
break_expr : "break" [ lifetime ] ?;
```

Continue expressions

```
continue_expr : "continue" [ lifetime ] ?;
```

For expressions

```
for_expr : [ lifetime ':' ] ? "for" pat "in" no_struct_literal_expr '{' block '}' ;
```

If expressions

```
if_expr : "if" no_struct_literal_expr '{' block '}'
          else_tail ? ;

else_tail : "else" [ if_expr | if_let_expr
                   | '{' block '}' ] ;
```

Match expressions

```
match_expr : ”match” no_struct_literal_expr ’{’ match_arm * ’}’ ;

match_arm : attribute * match_pat ”=>” [ expr ”,” | ’{’ block ’}’ ] ;

match_pat : pat [ ’|’ pat ] * [ ”if” expr ] ? ;
```

If let expressions

```
if_let_expr : ”if” ”let” pat ’=’ expr ’{’ block ’}’
               else_tail ? ;
```

While let loops

```
while_let_expr : [ lifetime ’:’ ] ? ”while” ”let” pat ’=’ expr ’{’ block ’}’ ;
```

Return expressions

```
return_expr : ”return” expr ? ;
```

19.8 Type system

FIXME: is this entire chapter relevant here? Or should it all have been covered by some production already?

19.8.1 Types

Primitive types

FIXME: grammar?

Machine types **FIXME:** grammar?

Machine-dependent integer types **FIXME:** grammar?

Textual types

FIXME: grammar?

Tuple types

FIXME: grammar?

Array, and Slice types

FIXME: grammar?

Structure types

FIXME: grammar?

Enumerated types

FIXME: grammar?

Pointer types

FIXME: grammar?

Function types

FIXME: grammar?

Closure types

```
closure_type := [ 'unsafe' ] [ '<' lifetime-list '>' ] '|' arg-list '|'
                [ ':' bound-list ] [ '->' type ]
lifetime-list := lifetime | lifetime ',' lifetime-list
arg-list := ident ':' type | ident ':' type ',' arg-list
bound-list := bound | bound '+' bound-list
bound := path | lifetime
```

Object types

FIXME: grammar?

Type parameters

FIXME: grammar?

Self types

FIXME: grammar?

19.8.2 Type kinds

FIXME: this is probably not relevant to the grammar...

19.9 Memory and concurrency models

FIXME: is this entire chapter relevant here? Or should it all have been covered by some production already?

19.9.1 Memory model

Memory allocation and lifetime

Memory ownership

Variables

Boxes

19.9.2 Threads

Communication between threads

Thread lifecycle

Chapter 20

Reference

20.1 Introduction

This document is the primary reference for the Rust programming language. It provides three kinds of material:

- Chapters that informally describe each language construct and their use.
- Chapters that informally describe the memory model, concurrency model, runtime services, linkage model and debugging facilities.
- Appendix chapters providing rationale and references to languages that influenced the design.

This document does not serve as an introduction to the language. Background familiarity with the language is assumed. A separate book[1] is available to help acquire such background familiarity.

This document also does not serve as a reference to the standard[2] library included in the language distribution. Those libraries are documented separately by extracting documentation attributes from their source code. Many of the features that one might expect to be language features are library features in Rust, so what you're looking for may be there, not here.

You may also be interested in the grammar[3].

20.2 Notation

20.2.1 Unicode productions

A few productions in Rust's grammar permit Unicode code points outside the ASCII range. We define these productions in terms of character properties specified in the

[1] book/index.html
[2] std/index.html
[3] grammar.html

Unicode standard, rather than in terms of ASCII-range code points. The grammar has a Special Unicode Productions[4] section that lists these productions.

20.2.2 String table productions

Some rules in the grammar — notably unary operators, binary operators, and keywords[5] — are given in a simplified form: as a listing of a table of unquoted, printable whitespace-separated strings. These cases form a subset of the rules regarding the token rule, and are assumed to be the result of a lexical-analysis phase feeding the parser, driven by a DFA, operating over the disjunction of all such string table entries.

When such a string enclosed in double-quotes (") occurs inside the grammar, it is an implicit reference to a single member of such a string table production. See tokens for more information.

20.3 Lexical structure

20.3.1 Input format

Rust input is interpreted as a sequence of Unicode code points encoded in UTF-8. Most Rust grammar rules are defined in terms of printable ASCII-range code points, but a small number are defined in terms of Unicode properties or explicit code point lists. [6]

20.3.2 Identifiers

An identifier is any nonempty Unicode[7] string of the following form:

Either

- The first character has property `XID_start`
- The remaining characters have property `XID_continue`

Or

- The first character is _
- The identifier is more than one character, _ alone is not an identifier
- The remaining characters have property `XID_continue`

that does *not* occur in the set of keywords[8].

> **Note**: `XID_start` and `XID_continue` as character properties cover the character ranges used to form the more familiar C and Java language-family identifiers.

[4] `grammar.html#special-unicode-productions`

[5] `grammar.html#keywords`

[6] Substitute definitions for the special Unicode productions are provided to the grammar verifier, restricted to ASCII range, when verifying the grammar in this document.

[7] Non-ASCII characters in identifiers are currently feature gated. This is expected to improve soon.

[8] `grammar.html#keywords`

20.3.3 Comments

Comments in Rust code follow the general C++ style of line (`//`) and block (`/* ... */`) comment forms. Nested block comments are supported.

Line comments beginning with exactly *three* slashes (`///`), and block comments (`/** ... */`), are interpreted as a special syntax for `doc` attributes. That is, they are equivalent to writing `#[doc="..."]` around the body of the comment, i.e., `/// Foo` turns into `#[doc="Foo"]`.

Line comments beginning with `//!` and block comments `/*! ... !*/` are doc comments that apply to the parent of the comment, rather than the item that follows. That is, they are equivalent to writing `#![doc="..."]` around the body of the comment. `//!` comments are usually used to document modules that occupy a source file.

Non-doc comments are interpreted as a form of whitespace.

20.3.4 Whitespace

Whitespace is any non-empty string containing only the following characters:

- `U+0020` (space, `' '`)
- `U+0009` (tab, `'\t'`)
- `U+000A` (LF, `'\n'`)
- `U+000D` (CR, `'\r'`)

Rust is a "free-form" language, meaning that all forms of whitespace serve only to separate *tokens* in the grammar, and have no semantic significance.

A Rust program has identical meaning if each whitespace element is replaced with any other legal whitespace element, such as a single space character.

20.3.5 Tokens

Tokens are primitive productions in the grammar defined by regular (non-recursive) languages. "Simple" tokens are given in string table production form, and occur in the rest of the grammar as double-quoted strings. Other tokens have exact rules given.

Literals

A literal is an expression consisting of a single token, rather than a sequence of tokens, that immediately and directly denotes the value it evaluates to, rather than referring to it by name or some other evaluation rule. A literal is a form of constant expression, so is evaluated (primarily) at compile time.

Examples

	Example	# sets	Characters	Escapes
Character	`'H'`	`N/A`	All Unicode	Quote & Byte & Unicode
String	`"hello"`	`N/A`	All Unicode	Quote & Byte & Unicode
Raw	`r#"hello"#`	`0...`	All Unicode	`N/A`
Byte	`b'H'`	`N/A`	All ASCII	Quote & Byte
Byte string	`b"hello"`	`N/A`	All ASCII	Quote & Byte
Raw byte string	`br#"hello"#`	`0...`	All ASCII	`N/A`

Characters and strings

	Name
`\x7F`	8-bit character code (exactly 2 digits)
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash
`\0`	Null

Byte escapes

	Name
`\u{7FFF}`	24-bit Unicode character code (up to 6 digits)

Unicode escapes

	Name
`\'`	Single quote
`\"`	Double quote

Quote escapes

Number literals*	Example	Exponentiation	Suffixes
Decimal integer	`98_222`	`N/A`	Integer suffixes
Hex integer	`0xff`	`N/A`	Integer suffixes
Octal integer	`0o77`	`N/A`	Integer suffixes
Binary integer	`0b1111_0000`	`N/A`	Integer suffixes
Floating-point	`123.0E+77`	`Optional`	Floating-point suffixes

Numbers * All number literals allow _ as a visual separator: `1_234.0E+18f64`

Integer	Floating-point
`u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, `isize`, `usize`	`f32`, `f64`

Suffixes

Character and string literals

Character literals A *character literal* is a single Unicode character enclosed within two `U+0027` (single-quote) characters, with the exception of `U+0027` itself, which must be *escaped* by a preceding `U+005C` character (`\`).

String literals A *string literal* is a sequence of any Unicode characters enclosed within two `U+0022` (double-quote) characters, with the exception of `U+0022` itself, which must be *escaped* by a preceding `U+005C` character (`\`).

Line-break characters are allowed in string literals. Normally they represent themselves (i.e. no translation), but as a special exception, when a `U+005C` character (`\`) occurs immediately before the newline, the `U+005C` character, the newline, and all whitespace at the beginning of the next line are ignored. Thus `a` and `b` are equal:

```
let a = "foobar";
let b = "foo\
         bar";

assert_eq!(a,b);
```

Character escapes Some additional *escapes* are available in either character or non-raw string literals. An escape starts with a `U+005C` (`\`) and continues with one of the following forms:

- An *8-bit code point escape* starts with `U+0078` (`x`) and is followed by exactly two *hex digits*. It denotes the Unicode code point equal to the provided hex value.
- A *24-bit code point escape* starts with `U+0075` (`u`) and is followed by up to six *hex digits* surrounded by braces `U+007B` (`{`) and `U+007D` (`}`). It denotes the Unicode code point equal to the provided hex value.
- A *whitespace escape* is one of the characters `U+006E` (`n`), `U+0072` (`r`), or `U+0074` (`t`), denoting the Unicode values `U+000A` (LF), `U+000D` (CR) or `U+0009` (HT) respectively.
- The *backslash escape* is the character `U+005C` (`\`) which must be escaped in order to denote *itself*.

Raw string literals Raw string literals do not process any escapes. They start with the character U+0072 (r), followed by zero or more of the character U+0023 (#) and a U+0022 (double-quote) character. The *raw string body* can contain any sequence of Unicode characters and is terminated only by another U+0022 (double-quote) character, followed by the same number of U+0023 (#) characters that preceded the opening U+0022 (double-quote) character.

All Unicode characters contained in the raw string body represent themselves, the characters U+0022 (double-quote) (except when followed by at least as many U+0023 (#) characters as were used to start the raw string literal) or U+005C (\) do not have any special meaning.

Examples for string literals:

```
”foo”; r”foo”;                      // foo
”\”foo\””; r#””foo””#;               // ”foo”

”foo #\”# bar”;
r##”foo #”# bar”##;                 // foo #”# bar

”\x52”; ”R”; r”R”;                  // R
”\\x52”; r”\x52”;                   // \x52
```

Byte and byte string literals

Byte literals A *byte literal* is a single ASCII character (in the U+0000 to U+007F range) or a single *escape* preceded by the characters U+0062 (b) and U+0027 (single-quote), and followed by the character U+0027. If the character U+0027 is present within the literal, it must be *escaped* by a preceding U+005C (\) character. It is equivalent to a u8 unsigned 8-bit integer *number literal.*

Byte string literals A non-raw *byte string literal* is a sequence of ASCII characters and *escapes,* preceded by the characters U+0062 (b) and U+0022 (double-quote), and followed by the character U+0022. If the character U+0022 is present within the literal, it must be *escaped* by a preceding U+005C (\) character. Alternatively, a byte string literal can be a *raw byte string literal,* defined below. A byte string literal of length n is equivalent to a `&’static [u8; n]` borrowed fixed-sized array of unsigned 8-bit integers.

Some additional *escapes* are available in either byte or non-raw byte string literals. An escape starts with a U+005C (\) and continues with one of the following forms:

- A *byte escape* escape starts with U+0078 (x) and is followed by exactly two *hex digits.* It denotes the byte equal to the provided hex value.
- A *whitespace escape* is one of the characters U+006E (n), U+0072 (r), or U+0074 (t), denoting the bytes values 0x0A (ASCII LF), 0x0D (ASCII CR) or 0x09 (ASCII HT) respectively.
- The *backslash escape* is the character U+005C (\) which must be escaped in order to denote its ASCII encoding 0x5C.

Raw byte string literals Raw byte string literals do not process any escapes. They start with the character `U+0062` (`b`), followed by `U+0072` (`r`), followed by zero or more of the character `U+0023` (`#`), and a `U+0022` (double-quote) character. The *raw string body* can contain any sequence of ASCII characters and is terminated only by another `U+0022` (double-quote) character, followed by the same number of `U+0023` (`#`) characters that preceded the opening `U+0022` (double-quote) character. A raw byte string literal can not contain any non-ASCII byte.

All characters contained in the raw string body represent their ASCII encoding, the characters `U+0022` (double-quote) (except when followed by at least as many `U+0023` (`#`) characters as were used to start the raw string literal) or `U+005C` (`\`) do not have any special meaning.

Examples for byte string literals:

```
b"foo"; br"foo";                     // foo
b"\"foo\""; br#""foo""#;             // "foo"

b"foo #\"# bar";
br##"foo #"# bar"##;                 // foo #"# bar

b"\x52"; b"R"; br"R";                // R
b"\\x52"; br"\x52";                  // \x52
```

Number literals A *number literal* is either an *integer literal* or a *floating-point literal*. The grammar for recognizing the two kinds of literals is mixed.

Integer literals An *integer literal* has one of four forms:

- A *decimal literal* starts with a *decimal digit* and continues with any mixture of *decimal digits* and *underscores*.
- A *hex literal* starts with the character sequence `U+0030` `U+0078` (`0x`) and continues as any mixture of hex digits and underscores.
- An *octal literal* starts with the character sequence `U+0030` `U+006F` (`0o`) and continues as any mixture of octal digits and underscores.
- A *binary literal* starts with the character sequence `U+0030` `U+0062` (`0b`) and continues as any mixture of binary digits and underscores.

Like any literal, an integer literal may be followed (immediately, without any spaces) by an *integer suffix*, which forcibly sets the type of the literal. The integer suffix must be the name of one of the integral types: `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, `isize`, or `usize`.

The type of an *unsuffixed* integer literal is determined by type inference:

- If an integer type can be *uniquely* determined from the surrounding program context, the unsuffixed integer literal has that type.
- If the program context under-constrains the type, it defaults to the signed 32-bit integer `i32`.

- If the program context over-constrains the type, it is considered a static type error.

Examples of integer literals of various forms:

```
123i32;                         // type i32
123u32;                         // type u32
123_u32;                        // type u32
0xff_u8;                        // type u8
0o70_i16;                       // type i16
0b1111_1111_1001_0000_i32;      // type i32
0usize;                         // type usize
```

Floating-point literals A *floating-point literal* has one of two forms:

- A *decimal literal* followed by a period character `U+002E` (`.`). This is optionally followed by another decimal literal, with an optional *exponent*.
- A single *decimal literal* followed by an *exponent*.

Like integer literals, a floating-point literal may be followed by a suffix, so long as the pre-suffix part does not end with `U+002E` (`.`). The suffix forcibly sets the type of the literal. There are two valid *floating-point suffixes*, `f32` and `f64` (the 32-bit and 64-bit floating point types), which explicitly determine the type of the literal.

The type of an *unsuffixed* floating-point literal is determined by type inference:

- If a floating-point type can be *uniquely* determined from the surrounding program context, the unsuffixed floating-point literal has that type.
- If the program context under-constrains the type, it defaults to `f64`.
- If the program context over-constrains the type, it is considered a static type error.

Examples of floating-point literals of various forms:

```
123.0f64;        // type f64
0.1f64;          // type f64
0.1f32;          // type f32
12E+99_f64;      // type f64
let x: f64 = 2.; // type f64
```

This last example is different because it is not possible to use the suffix syntax with a floating point literal ending in a period. `2.f64` would attempt to call a method named `f64` on `2`.

The representation semantics of floating-point numbers are described in "Machine Types".

Boolean literals The two values of the boolean type are written `true` and `false`.

Symbols

Symbols are a general class of printable tokens that play structural roles in a variety of grammar productions. They are a set of remaining miscellaneous printable tokens that do not otherwise appear as unary operators, binary operators, or keywords[9]. They are catalogued in the Symbols section[10] of the Grammar document.

20.3.6 Paths

A *path* is a sequence of one or more path components *logically* separated by a namespace qualifier (`::`). If a path consists of only one component, it may refer to either an item or a variable in a local control scope. If a path has multiple components, it refers to an item.

Every item has a *canonical path* within its crate, but the path naming an item is only meaningful within a given crate. There is no global namespace across crates; an item's canonical path merely identifies it within the crate.

Two examples of simple paths consisting of only identifier components:

```
x;
x::y::z;
```

Path components are usually identifiers, but they may also include angle-bracket-enclosed lists of type arguments. In expression context, the type argument list is given after a `::` namespace qualifier in order to disambiguate it from a relational expression involving the less-than symbol (`<`). In type expression context, the final namespace qualifier is omitted.

Two examples of paths with type arguments:

```
# struct HashMap<K, V>(K,V);
# fn f() {
# fn id<T>(t: T) -> T { t }
type T = HashMap<i32,String>; // Type arguments used in a type expression
let x  = id::<i32>(10);       // Type arguments used in a call expression
# }
```

Paths can be denoted with various leading qualifiers to change the meaning of how it is resolved:

- Paths starting with `::` are considered to be global paths where the components of the path start being resolved from the crate root. Each identifier in the path must resolve to an item.

```
mod a {
    pub fn foo() {}
}
```

[9] `grammar.html#keywords`
[10] `grammar.html#symbols`

```
mod b {
    pub fn foo() {
        ::a::foo(); // call a's foo function
    }
}
# fn main() {}
```

- Paths starting with the keyword `super` begin resolution relative to the parent module. Each further identifier must resolve to an item.

```
mod a {
    pub fn foo() {}
}
mod b {
    pub fn foo() {
        super::a::foo(); // call a's foo function
    }
}
# fn main() {}
```

- Paths starting with the keyword `self` begin resolution relative to the current module. Each further identifier must resolve to an item.

```
fn foo() {}
fn bar() {
    self::foo();
}
# fn main() {}
```

20.4 Syntax extensions

A number of minor features of Rust are not central enough to have their own syntax, and yet are not implementable as functions. Instead, they are given names, and invoked through a consistent syntax: `some_extension!(...)`.

Users of `rustc` can define new syntax extensions in two ways:

- Compiler plugins[11] can include arbitrary Rust code that manipulates syntax trees at compile time. Note that the interface for compiler plugins is considered highly unstable.
- Macros[12] define new syntax in a higher-level, declarative way.

20.4.1 Macros

`macro_rules` allows users to define syntax extension in a declarative way. We call such extensions "macros by example" or simply "macros" — to be distinguished from the "procedural macros" defined in compiler plugins[13].

[11] `book/compiler-plugins.html`
[12] `book/macros.html`
[13] `book/compiler-plugins.html`

Currently, macros can expand to expressions, statements, items, or patterns.

(A `sep_token` is any token other than `*` and `+`. A `non_special_token` is any token other than a delimiter or `$`.)

The macro expander looks up macro invocations by name, and tries each macro rule in turn. It transcribes the first successful match. Matching and transcription are closely related to each other, and we will describe them together.

Macro By Example

The macro expander matches and transcribes every token that does not begin with a `$` literally, including delimiters. For parsing reasons, delimiters must be balanced, but they are otherwise not special.

In the matcher, `$` *name* `:` *designator* matches the nonterminal in the Rust syntax named by *designator*. Valid designators are:

- `item`: an item
- `block`: a block
- `stmt`: a statement
- `pat`: a pattern
- `expr`: an expression
- `ty`: a type
- `ident`: an identifier
- `path`: a path
- `tt`: either side of the `=>` in macro rules
- `meta`: the contents of an attribute

In the transcriber, the designator is already known, and so only the name of a matched nonterminal comes after the dollar sign.

In both the matcher and transcriber, the Kleene star-like operator indicates repetition. The Kleene star operator consists of `$` and parentheses, optionally followed by a separator token, followed by `*` or `+`. `*` means zero or more repetitions, `+` means at least one repetition. The parentheses are not matched or transcribed. On the matcher side, a name is bound to *all* of the names it matches, in a structure that mimics the structure of the repetition encountered on a successful match. The job of the transcriber is to sort that structure out.

The rules for transcription of these repetitions are called "Macro By Example". Essentially, one "layer" of repetition is discharged at a time, and all of them must be discharged by the time a name is transcribed. Therefore, `( $( $i:ident ),* ) => ( $i )` is an invalid macro, but `( $( $i:ident ),* ) => ( $( $i:ident ),* )` is acceptable (if trivial).

When Macro By Example encounters a repetition, it examines all of the `$` *name* s that occur in its body. At the "current layer", they all must repeat the same number of times, so `( $( $i:ident ),* ; $( $j:ident ),* ) => ( $( ($i,$j) ),* )` is valid if given the argument `(a,b,c ; d,e,f)`, but not `(a,b,c ; d,e)`. The repetition walks through the choices at that layer in lockstep, so the former input transcribes to `(a,d), (b,e), (c,f)`.

Nested repetitions are allowed.

Parsing limitations

The parser used by the macro system is reasonably powerful, but the parsing of Rust syntax is restricted in two ways:

1. Macro definitions are required to include suitable separators after parsing expressions and other bits of the Rust grammar. This implies that a macro definition like `$i:expr [ , ]` is not legal, because `[` could be part of an expression. A macro definition like `$i:expr ,` or `$i:expr ;` would be legal, however, because `,` and `;` are legal separators. See RFC 550[14] for more information.
2. The parser must have eliminated all ambiguity by the time it reaches a `$` *name* `:` *designator*. This requirement most often affects name-designator pairs when they occur at the beginning of, or immediately after, a `$(...)*`; requiring a distinctive token in front can solve the problem.

20.5 Crates and source files

Although Rust, like any other language, can be implemented by an interpreter as well as a compiler, the only existing implementation is a compiler, and the language has always been designed to be compiled. For these reasons, this section assumes a compiler.

Rust's semantics obey a *phase distinction* between compile-time and run-time.[15] Semantic rules that have a *static interpretation* govern the success or failure of compilation, while semantic rules that have a *dynamic interpretation* govern the behavior of the program at run-time.

The compilation model centers on artifacts called *crates*. Each compilation processes a single crate in source form, and if successful, produces a single crate in binary form: either an executable or some sort of library.[16]

A *crate* is a unit of compilation and linking, as well as versioning, distribution and runtime loading. A crate contains a *tree* of nested module scopes. The top level of this tree is a module that is anonymous (from the point of view of paths within the module) and any item within a crate has a canonical module path denoting its location within the crate's module tree.

The Rust compiler is always invoked with a single source file as input, and always produces a single output crate. The processing of that source file may result in other source files being loaded as modules. Source files have the extension `.rs`.

A Rust source file describes a module, the name and location of which — in the module tree of the current crate — are defined from outside the source file: either by an explicit `mod_item` in a referencing source file, or by the name of the crate itself. Every source file is a module, but not every module needs its own source file: module definitions can be nested within one file.

[14] `https://github.com/rust-lang/rfcs/blob/master/text/0550-macro-future-proofing.md`

[15] This distinction would also exist in an interpreter. Static checks like syntactic analysis, type checking, and lints should happen before the program is executed regardless of when it is executed.

[16] A crate is somewhat analogous to an *assembly* in the ECMA-335 CLI model, a *library* in the SML/NJ Compilation Manager, a *unit* in the Owens and Flatt module system, or a *configuration* in Mesa.

Each source file contains a sequence of zero or more `item` definitions, and may optionally begin with any number of attributes that apply to the containing module, most of which influence the behavior of the compiler. The anonymous crate module can have additional attributes that apply to the crate as a whole.

```
// Specify the crate name.
#![crate_name = "projx"]

// Specify the type of output artifact.
#![crate_type = "lib"]

// Turn on a warning.
// This can be done in any module, not just the anonymous crate module.
#![warn(non_camel_case_types)]
```

A crate that contains a `main` function can be compiled to an executable. If a `main` function is present, its return type must be `()` ("unit") and it must take no arguments.

20.6 Items and attributes

Crates contain items, each of which may have some number of attributes attached to it.

20.6.1 Items

An *item* is a component of a crate. Items are organized within a crate by a nested set of modules. Every crate has a single "outermost" anonymous module; all further items within the crate have paths within the module tree of the crate.

Items are entirely determined at compile-time, generally remain fixed during execution, and may reside in read-only memory.

There are several kinds of item:

- `extern crate` declarations
- `use` declarations
- modules
- functions
- type definitions[17]
- structs
- enumerations
- constant items
- static items
- traits
- implementations

[17] `grammar.html#type-definitions`

Some items form an implicit scope for the declaration of sub-items. In other words, within a function or module, declarations of items can (in many cases) be mixed with the statements, control blocks, and similar artifacts that otherwise compose the item body. The meaning of these scoped items is the same as if the item was declared outside the scope — it is still a static item — except that the item's *path name* within the module namespace is qualified by the name of the enclosing item, or is private to the enclosing item (in the case of functions). The grammar specifies the exact locations in which sub-item declarations may appear.

Type Parameters

All items except modules, constants and statics may be *parameterized* by type. Type parameters are given as a comma-separated list of identifiers enclosed in angle brackets (`<...>`), after the name of the item and before its definition. The type parameters of an item are considered "part of the name", not part of the type of the item. A referencing path must (in principle) provide type arguments as a list of comma-separated types enclosed within angle brackets, in order to refer to the type-parameterized item. In practice, the type-inference system can usually infer such argument types from context. There are no general type-parametric types, only type-parametric items. That is, Rust has no notion of type abstraction: there are no higher-ranked (or "forall") types abstracted over other types, though higher-ranked types do exist for lifetimes.

Modules

A module is a container for zero or more items.

A *module item* is a module, surrounded in braces, named, and prefixed with the keyword `mod`. A module item introduces a new, named module into the tree of modules making up a crate. Modules can nest arbitrarily.

An example of a module:

```
mod math {
    type Complex = (f64, f64);
    fn sin(f: f64) -> f64 {
        /* ... */
# panic!();
    }
    fn cos(f: f64) -> f64 {
        /* ... */
# panic!();
    }
    fn tan(f: f64) -> f64 {
        /* ... */
# panic!();
    }
}
```

Modules and types share the same namespace. Declaring a named type with the same name as a module in scope is forbidden: that is, a type definition, trait, struct,

enumeration, or type parameter can't shadow the name of a module in scope, or vice versa.

A module without a body is loaded from an external file, by default with the same name as the module, plus the `.rs` extension. When a nested submodule is loaded from an external file, it is loaded from a subdirectory path that mirrors the module hierarchy.

```
// Load the ‘vec‘ module from ‘vec.rs‘
mod vec;

mod thread {
    // Load the ‘local_data‘ module from ‘thread/local_data.rs‘
    // or ‘thread/local_data/mod.rs‘.
    mod local_data;
}
```

The directories and files used for loading external file modules can be influenced with the `path` attribute.

```
#[path = ”thread_files”]
mod thread {
    // Load the ‘local_data‘ module from ‘thread_files/tls.rs‘
    #[path = ”tls.rs”]
    mod local_data;
}
```

Extern crate declarations An `extern crate` *declaration* specifies a dependency on an external crate. The external crate is then bound into the declaring scope as the `ident` provided in the `extern_crate_decl`.

The external crate is resolved to a specific `soname` at compile time, and a runtime linkage requirement to that `soname` is passed to the linker for loading at runtime. The `soname` is resolved at compile time by scanning the compiler's library path and matching the optional `crateid` provided against the `crateid` attributes that were declared on the external crate when it was compiled. If no `crateid` is provided, a default `name` attribute is assumed, equal to the `ident` given in the `extern_crate_decl`.

Three examples of `extern crate` declarations:

```
extern crate pcre;

extern crate std; // equivalent to: extern crate std as std;

extern crate std as ruststd; // linking to 'std' under another name
```

Use declarations A *use declaration* creates one or more local name bindings synonymous with some other path. Usually a `use` declaration is used to shorten the path required to refer to a module item. These declarations may appear at the top of modules and blocks[18].

[18] `grammar.html#block-expressions`

> **Note**: Unlike in many languages, `use` declarations in Rust do *not* declare linkage dependency with external crates. Rather, `extern crate` declarations declare linkage dependencies.

Use declarations support a number of convenient shortcuts:

- Rebinding the target name as a new local name, using the syntax `use p::q::r as x;`
- Simultaneously binding a list of paths differing only in their final element, using the glob-like brace syntax `use a::b::{c,d,e,f};`
- Binding all paths matching a given prefix, using the asterisk wildcard syntax `use a::b::*;`
- Simultaneously binding a list of paths differing only in their final element and their immediate parent module, using the `self` keyword, such as `use a::b::{self, c, d};`

An example of `use` declarations:

```
use std::option::Option::{Some, None};
use std::collections::hash_map::{self, HashMap};

fn foo<T>(_: T){}
fn bar(map1: HashMap<String, usize>, map2: hash_map::HashMap<String, usize>){}

fn main() {
    // Equivalent to 'foo(vec![std::option::Option::Some(1.0f64),
    // std::option::Option::None]);'
    foo(vec![Some(1.0f64), None]);

    // Both ‘hash_map‘ and ‘HashMap‘ are in scope.
    let map1 = HashMap::new();
    let map2 = hash_map::HashMap::new();
    bar(map1, map2);
}
```

Like items, `use` declarations are private to the containing module, by default. Also like items, a `use` declaration can be public, if qualified by the `pub` keyword. Such a `use` declaration serves to *re-export* a name. A public `use` declaration can therefore *redirect* some public name to a different target definition: even a definition with a private canonical path, inside a different module. If a sequence of such redirections form a cycle or cannot be resolved unambiguously, they represent a compile-time error.

An example of re-exporting:

```
# fn main() { }
mod quux {
    pub use quux::foo::{bar, baz};
```

```
    pub mod foo {
        pub fn bar() { }
        pub fn baz() { }
    }
}
```

In this example, the module `quux` re-exports two public names defined in `foo`.

Also note that the paths contained in `use` items are relative to the crate root. So, in the previous example, the `use` refers to `quux::foo::{bar, baz}`, and not simply to `foo::{bar, baz}`. This also means that top-level module declarations should be at the crate root if direct usage of the declared modules within `use` items is desired. It is also possible to use `self` and `super` at the beginning of a `use` item to refer to the current and direct parent modules respectively. All rules regarding accessing declared modules in `use` declarations apply to both module declarations and `extern crate` declarations.

An example of what will and will not work for `use` items:

```
# #![allow(unused_imports)]
use foo::baz::foobaz;    // good: foo is at the root of the crate

mod foo {

    mod example {
        pub mod iter {}
    }

    use foo::example::iter; // good: foo is at crate root
//  use example::iter;      // bad:  example is not at the crate root
    use self::baz::foobaz;  // good: self refers to module 'foo'
    use foo::bar::foobar;   // good: foo is at crate root

    pub mod bar {
        pub fn foobar() { }
    }

    pub mod baz {
        use super::bar::foobar; // good: super refers to module 'foo'
        pub fn foobaz() { }
    }
}

fn main() {}
```

Functions

A *function item* defines a sequence of statements and a final expression, along with a name and a set of parameters. Other than a name, all these are optional. Functions are declared with the keyword `fn`. Functions may declare a set of *input variables* as

parameters, through which the caller passes arguments into the function, and the *output type* of the value the function will return to its caller on completion.

A function may also be copied into a first-class *value*, in which case the value has the corresponding *function type*, and can be used otherwise exactly as a function item (with a minor additional cost of calling the function indirectly).

Every control path in a function logically ends with a `return` expression or a diverging expression. If the outermost block of a function has a value-producing expression in its final-expression position, that expression is interpreted as an implicit `return` expression applied to the final-expression.

An example of a function:

```
fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

As with `let` bindings, function arguments are irrefutable patterns, so any pattern that is valid in a let binding is also valid as an argument.

```
fn first((value, _): (i32, i32)) -> i32 { value }
```

Generic functions A *generic function* allows one or more *parameterized types* to appear in its signature. Each type parameter must be explicitly declared, in an angle-bracket-enclosed, comma-separated list following the function name.

```
// foo is generic over A and B

fn foo<A, B>(x: A, y: B) {
```

Inside the function signature and body, the name of the type parameter can be used as a type name. Trait bounds can be specified for type parameters to allow methods with that trait to be called on values of that type. This is specified using the `where` syntax:

```
fn foo<T>(x: T) where T: Debug {
```

When a generic function is referenced, its type is instantiated based on the context of the reference. For example, calling the `foo` function here:

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

fn foo<T>(x: &[T]) where T: Debug {
    // details elided
    # ()
}

foo(&[1, 2]);
```

will instantiate type parameter `T` with `i32`.

The type parameters can also be explicitly supplied in a trailing path component after the function name. This might be necessary if there is not sufficient context to determine the type parameters. For example, `mem::size_of::<u32>() == 4`.

Diverging functions A special kind of function can be declared with a `!` character where the output type would normally be. For example:

```
fn my_err(s: &str) -> ! {
    println!("{}", s);
    panic!();
}
```

We call such functions "diverging" because they never return a value to the caller. Every control path in a diverging function must end with a `panic!()` or a call to another diverging function on every control path. The `!` annotation does *not* denote a type.

It might be necessary to declare a diverging function because as mentioned previously, the typechecker checks that every control path in a function ends with a `return` or diverging expression. So, if `my_err` were declared without the `!` annotation, the following code would not typecheck:

```
# fn my_err(s: &str) -> ! { panic!() }

fn f(i: i32) -> i32 {
    if i == 42 {
        return 42;
    }
    else {
        my_err("Bad number!");
    }
}
```

This will not compile without the `!` annotation on `my_err`, since the `else` branch of the conditional in `f` does not return an `i32`, as required by the signature of `f`. Adding the `!` annotation to `my_err` informs the typechecker that, should control ever enter `my_err`, no further type judgments about `f` need to hold, since control will never resume in any context that relies on those judgments. Thus the return type on `f` only needs to reflect the `if` branch of the conditional.

Extern functions Extern functions are part of Rust's foreign function interface, providing the opposite functionality to external blocks. Whereas external blocks allow Rust code to call foreign code, extern functions with bodies defined in Rust code *can be called by foreign code*. They are defined in the same way as any other Rust function, except that they have the `extern` modifier.

```
// Declares an extern fn, the ABI defaults to "C"
extern fn new_i32() -> i32 { 0 }

// Declares an extern fn with "stdcall" ABI
extern "stdcall" fn new_i32_stdcall() -> i32 { 0 }
```

Unlike normal functions, extern fns have type `extern "ABI" fn()`. This is the same type as the functions declared in an extern block.

```
# extern fn new_i32() -> i32 { 0 }
let fptr: extern "C" fn() -> i32 = new_i32;
```

Extern functions may be called directly from Rust code as Rust uses large, contiguous stack segments like C.

Type aliases

A *type alias* defines a new name for an existing type. Type aliases are declared with the keyword `type`. Every value has a single, specific type, but may implement several different traits, or be compatible with several different type constraints.

For example, the following defines the type `Point` as a synonym for the type `(u8, u8)`, the type of pairs of unsigned 8 bit integers:

```
type Point = (u8, u8);
let p: Point = (41, 68);
```

Structs

A *struct* is a nominal struct type defined with the keyword `struct`.

An example of a `struct` item and its use:

```
struct Point {x: i32, y: i32}
let p = Point {x: 10, y: 11};
let px: i32 = p.x;
```

A *tuple struct* is a nominal tuple type, also defined with the keyword `struct`. For example:

```
struct Point(i32, i32);
let p = Point(10, 11);
let px: i32 = match p { Point(x, _) => x };
```

A *unit-like struct* is a struct without any fields, defined by leaving off the list of fields entirely. Such a struct implicitly defines a constant of its type with the same name. For example:

```
# #![feature(braced_empty_structs)]
struct Cookie;
let c = [Cookie, Cookie {}, Cookie, Cookie {}];
```

is equivalent to

```
# #![feature(braced_empty_structs)]
struct Cookie {}
const Cookie: Cookie = Cookie {};
let c = [Cookie, Cookie {}, Cookie, Cookie {}];
```

The precise memory layout of a struct is not specified. One can specify a particular layout using the `repr` attribute.

Enumerations

An *enumeration* is a simultaneous definition of a nominal enumerated type as well as a set of *constructors*, that can be used to create or pattern-match values of the corresponding enumerated type.

Enumerations are declared with the keyword `enum`.

An example of an `enum` item and its use:

```
enum Animal {
    Dog,
    Cat,
}

let mut a: Animal = Animal::Dog;
a = Animal::Cat;
```

Enumeration constructors can have either named or unnamed fields:

```
enum Animal {
    Dog (String, f64),
    Cat { name: String, weight: f64 }
}

let mut a: Animal = Animal::Dog("Cocoa".to_string(), 37.2);
a = Animal::Cat { name: "Spotty".to_string(), weight: 2.7 };
```

In this example, `Cat` is a *struct-like enum variant*, whereas `Dog` is simply called an enum variant.

Enums have a discriminant. You can assign them explicitly:

```
enum Foo {
    Bar = 123,
}
```

If a discriminant isn't assigned, they start at zero, and add one for each variant, in order.

You can cast an enum to get this value:

```
# enum Foo { Bar = 123 }
let x = Foo::Bar as u32; // x is now 123u32
```

This only works as long as none of the variants have data attached. If it were `Bar(i32)`, this is disallowed.

Constant items

A *constant item* is a named *constant value* which is not associated with a specific memory location in the program. Constants are essentially inlined wherever they are used, meaning that they are copied directly into the relevant context when used. References to the same constant are not necessarily guaranteed to refer to the same memory address.

Constant values must not have destructors, and otherwise permit most forms of data. Constants may refer to the address of other constants, in which case the address will have the `static` lifetime. The compiler is, however, still at liberty to translate the constant many times, so the address referred to may not be stable.

Constants must be explicitly typed. The type may be `bool`, `char`, a number, or a type derived from those primitive types. The derived types are references with the `static` lifetime, fixed-size arrays, tuples, enum variants, and structs.

```
const BIT1: u32 = 1 << 0;
const BIT2: u32 = 1 << 1;

const BITS: [u32; 2] = [BIT1, BIT2];
const STRING: &'static str = "bitstring";

struct BitsNStrings<'a> {
    mybits: [u32; 2],
    mystring: &'a str
}

const BITS_N_STRINGS: BitsNStrings<'static> = BitsNStrings {
    mybits: BITS,
    mystring: STRING
};
```

Static items

A *static item* is similar to a *constant,* except that it represents a precise memory location in the program. A static is never "inlined" at the usage site, and all references to it refer to the same memory location. Static items have the `static` lifetime, which outlives all other lifetimes in a Rust program. Static items may be placed in read-only memory if they do not contain any interior mutability.

Statics may contain interior mutability through the `UnsafeCell` language item. All access to a static is safe, but there are a number of restrictions on statics:

- Statics may not contain any destructors.
- The types of static values must ascribe to `Sync` to allow thread-safe access.
- Statics may not refer to other statics by value, only by reference.
- Constants cannot refer to statics.

Constants should in general be preferred over statics, unless large amounts of data are being stored, or single-address and mutability properties are required.

Mutable statics If a static item is declared with the `mut` keyword, then it is allowed to be modified by the program. One of Rust's goals is to make concurrency bugs hard to run into, and this is obviously a very large source of race conditions or other bugs. For this reason, an `unsafe` block is required when either reading or writing a mutable static variable. Care should be taken to ensure that modifications to a mutable static are safe with respect to other threads running in the same process.

Mutable statics are still very useful, however. They can be used with C libraries and can also be bound from C libraries (in an `extern` block).

```
# fn atomic_add(_: &mut u32, _: u32) -> u32 { 2 }

static mut LEVELS: u32 = 0;

// This violates the idea of no shared state, and this doesn't internally
// protect against races, so this function is 'unsafe'
unsafe fn bump_levels_unsafe1() -> u32 {
    let ret = LEVELS;
    LEVELS += 1;
    return ret;
}

// Assuming that we have an atomic_add function which returns the old value,
// this function is "safe" but the meaning of the return value may not be what
// callers expect, so it's still marked as 'unsafe'
unsafe fn bump_levels_unsafe2() -> u32 {
    return atomic_add(&mut LEVELS, 1);
}
```

Mutable statics have the same restrictions as normal statics, except that the type of the value is not required to ascribe to `Sync`.

Traits

A *trait* describes an abstract interface that types can implement. This interface consists of associated items, which come in three varieties:

- functions
- constants
- types

Associated functions whose first parameter is named `self` are called methods and may be invoked using `.` notation (e.g., `x.foo()`).

All traits define an implicit type parameter `Self` that refers to "the type that is implementing this interface". Traits may also contain additional type parameters. These type parameters (including `Self`) may be constrained by other traits and so forth as usual.

Trait bounds on `Self` are considered "supertraits". These are required to be acyclic. Supertraits are somewhat different from other constraints in that they affect what methods are available in the vtable when the trait is used as a trait object.

Traits are implemented for specific types through separate implementations.

Consider the following trait:

```
# type Surface = i32;
# type BoundingBox = i32;
trait Shape {
    fn draw(&self, Surface);
    fn bounding_box(&self) -> BoundingBox;
}
```

This defines a trait with two methods. All values that have implementations of this trait in scope can have their `draw` and `bounding_box` methods called, using `value.bounding_box()` syntax.

Traits can include default implementations of methods, as in:

```
trait Foo {
    fn bar(&self);
    fn baz(&self) { println!(”We called baz.”); }
}
```

Here the `baz` method has a default implementation, so types that implement `Foo` need only implement `bar`. It is also possible for implementing types to override a method that has a default implementation.

Type parameters can be specified for a trait to make it generic. These appear after the trait name, using the same syntax used in generic functions.

```
trait Seq<T> {
    fn len(&self) -> u32;
    fn elt_at(&self, n: u32) -> T;
    fn iter<F>(&self, F) where F: Fn(T);
}
```

It is also possible to define associated types for a trait. Consider the following example of a `Container` trait. Notice how the type is available for use in the method signatures:

```
trait Container {
    type E;
    fn empty() -> Self;
    fn insert(&mut self, Self::E);
}
```

In order for a type to implement this trait, it must not only provide implementations for every method, but it must specify the type `E`. Here’s an implementation of `Container` for the standard library type `Vec`:

```
# trait Container {
#     type E;
#     fn empty() -> Self;
```

```
#     fn insert(&mut self, Self::E);
# }
impl<T> Container for Vec<T> {
    type E = T;
    fn empty() -> Vec<T> { Vec::new() }
    fn insert(&mut self, x: T) { self.push(x); }
}
```

Generic functions may use traits as *bounds* on their type parameters. This will have two effects:

- Only types that have the trait may instantiate the parameter.
- Within the generic function, the methods of the trait can be called on values that have the parameter's type.

For example:

```
# type Surface = i32;
# trait Shape { fn draw(&self, Surface); }
fn draw_twice<T: Shape>(surface: Surface, sh: T) {
    sh.draw(surface);
    sh.draw(surface);
}
```

Traits also define a trait object with the same name as the trait. Values of this type are created by coercing from a pointer of some specific type to a pointer of trait type. For example, `&T` could be coerced to `&Shape` if `T: Shape` holds (and similarly for `Box<T>`). This coercion can either be implicit or explicit. Here is an example of an explicit coercion:

```
trait Shape { }
impl Shape for i32 { }
let mycircle = 0i32;
let myshape: Box<Shape> = Box::new(mycircle) as Box<Shape>;
```

The resulting value is a box containing the value that was cast, along with information that identifies the methods of the implementation that was used. Values with a trait type can have methods called on them, for any method in the trait, and can be used to instantiate type parameters that are bounded by the trait.

Trait methods may be static, which means that they lack a `self` argument. This means that they can only be called with function call syntax (`f(x)`) and not method call syntax (`obj.f()`). The way to refer to the name of a static method is to qualify it with the trait name, treating the trait name like a module. For example:

```
trait Num {
    fn from_i32(n: i32) -> Self;
}
impl Num for f64 {
    fn from_i32(n: i32) -> f64 { n as f64 }
}
let x: f64 = Num::from_i32(42);
```

Traits may inherit from other traits. Consider the following example:

```
trait Shape { fn area(&self) -> f64; }
trait Circle : Shape { fn radius(&self) -> f64; }
```

The syntax `Circle : Shape` means that types that implement `Circle` must also have an implementation for `Shape`. Multiple supertraits are separated by `+`, `trait Circle : Shape + PartialEq { }`. In an implementation of `Circle` for a given type `T`, methods can refer to `Shape` methods, since the typechecker checks that any type with an implementation of `Circle` also has an implementation of `Shape`:

```
struct Foo;

trait Shape { fn area(&self) -> f64; }
trait Circle : Shape { fn radius(&self) -> f64; }
impl Shape for Foo {
    fn area(&self) -> f64 {
        0.0
    }
}
impl Circle for Foo {
    fn radius(&self) -> f64 {
        println!(”calling area: {}”, self.area());

        0.0
    }
}

let c = Foo;
c.radius();
```

In type-parameterized functions, methods of the supertrait may be called on values of subtrait-bound type parameters. Referring to the previous example of `trait Circle : Shape`:

```
# trait Shape { fn area(&self) -> f64; }
# trait Circle : Shape { fn radius(&self) -> f64; }
fn radius_times_area<T: Circle>(c: T) -> f64 {
    // ‘c‘ is both a Circle and a Shape
    c.radius() * c.area()
}
```

Likewise, supertrait methods may also be called on trait objects.

```
# trait Shape { fn area(&self) -> f64; }
# trait Circle : Shape { fn radius(&self) -> f64; }
# impl Shape for i32 { fn area(&self) -> f64 { 0.0 } }
# impl Circle for i32 { fn radius(&self) -> f64 { 0.0 } }
# let mycircle = 0i32;
let mycircle = Box::new(mycircle) as Box<Circle>;
let nonsense = mycircle.radius() * mycircle.area();
```

Implementations

An *implementation* is an item that implements a trait for a specific type. Implementations are defined with the keyword `impl`.

```
# #[derive(Copy, Clone)]
# struct Point {x: f64, y: f64};
# type Surface = i32;
# struct BoundingBox {x: f64, y: f64, width: f64, height: f64};
# trait Shape { fn draw(&self, Surface); fn bounding_box(&self) -> BoundingBox; }
# fn do_draw_circle(s: Surface, c: Circle) { }
struct Circle {
    radius: f64,
    center: Point,
}

impl Copy for Circle {}

impl Clone for Circle {
    fn clone(&self) -> Circle { *self }
}

impl Shape for Circle {
    fn draw(&self, s: Surface) { do_draw_circle(s, *self); }
    fn bounding_box(&self) -> BoundingBox {
        let r = self.radius;
        BoundingBox {
            x: self.center.x - r,
            y: self.center.y - r,
            width: 2.0 * r,
            height: 2.0 * r,
        }
    }
}
```

It is possible to define an implementation without referring to a trait. The methods in such an implementation can only be used as direct calls on the values of the type that the implementation targets. In such an implementation, the trait type and `for` after `impl` are omitted. Such implementations are limited to nominal types (enums, structs, trait objects), and the implementation must appear in the same crate as the `self` type:

```
struct Point {x: i32, y: i32}

impl Point {
    fn log(&self) {
        println!(”Point is at ({}, {})”, self.x, self.y);
    }
}
```

```
let my_point = Point {x: 10, y:11};
my_point.log();
```

When a trait *is* specified in an `impl`, all methods declared as part of the trait must be implemented, with matching types and type parameter counts.

An implementation can take type parameters, which can be different from the type parameters taken by the trait it implements. Implementation parameters are written after the `impl` keyword.

```
# trait Seq<T> { fn dummy(&self, _: T) { } }
impl<T> Seq<T> for Vec<T> {
    /* ... */
}
impl Seq<bool> for u32 {
    /* Treat the integer as a sequence of bits */
}
```

External blocks

External blocks form the basis for Rust's foreign function interface. Declarations in an external block describe symbols in external, non-Rust libraries.

Functions within external blocks are declared in the same way as other Rust functions, with the exception that they may not have a body and are instead terminated by a semicolon.

Functions within external blocks may be called by Rust code, just like functions defined in Rust. The Rust compiler automatically translates between the Rust ABI and the foreign ABI.

A number of attributes control the behavior of external blocks.

By default external blocks assume that the library they are calling uses the standard C "cdecl" ABI. Other ABIs may be specified using an `abi` string, as shown here:

```
// Interface to the Windows API
extern "stdcall" { }
```

The `link` attribute allows the name of the library to be specified. When specified the compiler will attempt to link against the native library of the specified name.

```
#[link(name = "crypto")]
extern { }
```

The type of a function declared in an extern block is `extern "abi" fn(A1, ..., An) -> R`, where `A1...An` are the declared types of its arguments and `R` is the declared return type.

It is valid to add the `link` attribute on an empty extern block. You can use this to satisfy the linking requirements of extern blocks elsewhere in your code (including upstream crates) instead of adding the attribute to each extern block.

20.6.2 Visibility and Privacy

These two terms are often used interchangeably, and what they are attempting to convey is the answer to the question "Can this item be used at this location?"

Rust's name resolution operates on a global hierarchy of namespaces. Each level in the hierarchy can be thought of as some item. The items are one of those mentioned above, but also include external crates. Declaring or defining a new module can be thought of as inserting a new tree into the hierarchy at the location of the definition.

To control whether interfaces can be used across modules, Rust checks each use of an item to see whether it should be allowed or not. This is where privacy warnings are generated, or otherwise "you used a private item of another module and weren't allowed to."

By default, everything in Rust is *private*, with one exception. Enum variants in a `pub` enum are also public by default. When an item is declared as `pub`, it can be thought of as being accessible to the outside world. For example:

```
# fn main() {}
// Declare a private struct
struct Foo;

// Declare a public struct with a private field
pub struct Bar {
    field: i32
}

// Declare a public enum with two public variants
pub enum State {
    PubliclyAccessibleState,
    PubliclyAccessibleState2,
}
```

With the notion of an item being either public or private, Rust allows item accesses in two cases:

1. If an item is public, then it can be used externally through any of its public ancestors.
2. If an item is private, it may be accessed by the current module and its descendants.

These two cases are surprisingly powerful for creating module hierarchies exposing public APIs while hiding internal implementation details. To help explain, here's a few use cases and what they would entail:

- A library developer needs to expose functionality to crates which link against their library. As a consequence of the first case, this means that anything which is usable externally must be `pub` from the root down to the destination item. Any private item in the chain will disallow external accesses.

- A crate needs a global available "helper module" to itself, but it doesn't want to expose the helper module as a public API. To accomplish this, the root of the crate's hierarchy would have a private module which then internally has a "public API". Because the entire crate is a descendant of the root, then the entire local crate can access this private module through the second case.
- When writing unit tests for a module, it's often a common idiom to have an immediate child of the module to-be-tested named `mod test`. This module could access any items of the parent module through the second case, meaning that internal implementation details could also be seamlessly tested from the child module.

In the second case, it mentions that a private item "can be accessed" by the current module and its descendants, but the exact meaning of accessing an item depends on what the item is. Accessing a module, for example, would mean looking inside of it (to import more items). On the other hand, accessing a function would mean that it is invoked. Additionally, path expressions and import statements are considered to access an item in the sense that the import/expression is only valid if the destination is in the current visibility scope.

Here's an example of a program which exemplifies the three cases outlined above:

```
// This module is private, meaning that no external crate can access this
// module. Because it is private at the root of this current crate, however, any
// module in the crate may access any publicly visible item in this module.
mod crate_helper_module {

    // This function can be used by anything in the current crate
    pub fn crate_helper() {}

    // This function *cannot* be used by anything else in the crate. It is not
    // publicly visible outside of the `crate_helper_module`, so only this
    // current module and its descendants may access it.
    fn implementation_detail() {}
}

// This function is "public to the root" meaning that it's available to external
// crates linking against this one.
pub fn public_api() {}

// Similarly to 'public_api', this module is public so external crates may look
// inside of it.
pub mod submodule {
    use crate_helper_module;

    pub fn my_method() {
        // Any item in the local crate may invoke the helper module's public
        // interface through a combination of the two rules above.
        crate_helper_module::crate_helper();
    }
```

```
    // This function is hidden to any module which is not a descendant of
    // 'submodule'
    fn my_implementation() {}

    #[cfg(test)]
    mod test {

        #[test]
        fn test_my_implementation() {
            // Because this module is a descendant of 'submodule', it's allowed
            // to access private items inside of 'submodule' without a privacy
            // violation.
            super::my_implementation();
        }
    }
}

# fn main() {}
```

For a rust program to pass the privacy checking pass, all paths must be valid accesses given the two rules above. This includes all use statements, expressions, types, etc.

Re-exporting and Visibility

Rust allows publicly re-exporting items through a `pub use` directive. Because this is a public directive, this allows the item to be used in the current module through the rules above. It essentially allows public access into the re-exported item. For example, this program is valid:

```
pub use self::implementation::api;

mod implementation {
    pub mod api {
        pub fn f() {}
    }
}

# fn main() {}
```

This means that any external crate referencing `implementation::api::f` would receive a privacy violation, while the path `api::f` would be allowed.

When re-exporting a private item, it can be thought of as allowing the "privacy chain" being short-circuited through the reexport instead of passing through the namespace hierarchy as it normally would.

20.6.3 Attributes

Any item declaration may have an *attribute* applied to it. Attributes in Rust are modeled on Attributes in ECMA-335, with the syntax coming from ECMA-334 (C#). An attribute is a general, free-form metadatum that is interpreted according to name, convention, and language and compiler version. Attributes may appear as any of:

- A single identifier, the attribute name
- An identifier followed by the equals sign '=' and a literal, providing a key/value pair
- An identifier followed by a parenthesized list of sub-attribute arguments

Attributes with a bang ("!") after the hash ("#") apply to the item that the attribute is declared within. Attributes that do not have a bang after the hash apply to the item that follows the attribute.

An example of attributes:

```
// General metadata applied to the enclosing module or crate.
#![crate_type = "lib"]

// A function marked as a unit test
#[test]
fn test_foo() {
    /* ... */
}

// A conditionally-compiled module
#[cfg(target_os="linux")]
mod bar {
    /* ... */
}

// A lint attribute used to suppress a warning/error
#[allow(non_camel_case_types)]
type int8_t = i8;
```

> **Note:** At some point in the future, the compiler will distinguish between language-reserved and user-available attributes. Until then, there is effectively no difference between an attribute handled by a loadable syntax extension and the compiler.

Crate-only attributes

- `crate_name` - specify the crate's crate name.
- `crate_type` - see linkage.
- `feature` - see compiler features.
- `no_builtins` - disable optimizing certain code patterns to invocations of library functions that are assumed to exist

- `no_main` - disable emitting the `main` symbol. Useful when some other object being linked to defines `main`.
- `no_start` - disable linking to the `native` crate, which specifies the "start" language item.
- `no_std` - disable linking to the `std` crate.
- `plugin` - load a list of named crates as compiler plugins, e.g. `#![plugin(foo, bar)]`. Optional arguments for each plugin, i.e. `#![plugin(foo(... args ...))]`, are provided to the plugin's registrar function. The `plugin` feature gate is required to use this attribute.
- `recursion_limit` - Sets the maximum depth for potentially infinitely-recursive compile-time operations like auto-dereference or macro expansion. The default is `#![recursion_limit="64"]`.

Module-only attributes

- `no_implicit_prelude` - disable injecting `use std::prelude::*` in this module.
- `path` - specifies the file to load the module from. `#[path="foo.rs"] mod bar;` is equivalent to `mod bar { /* contents of foo.rs */ }`. The path is taken relative to the directory that the current module is in.

Function-only attributes

- `main` - indicates that this function should be passed to the entry point, rather than the function in the crate root named `main`.
- `plugin_registrar` - mark this function as the registration point for compiler plugins[19], such as loadable syntax extensions.
- `start` - indicates that this function should be used as the entry point, overriding the "start" language item. See the "start" language item for more details.
- `test` - indicates that this function is a test function, to only be compiled in case of `--test`.
- `should_panic` - indicates that this test function should panic, inverting the success condition.
- `cold` - The function is unlikely to be executed, so optimize it (and calls to it) differently.

Static-only attributes

- `thread_local` - on a `static mut`, this signals that the value of this static may change depending on the current thread. The exact consequences of this are implementation-defined.

FFI attributes

On an `extern` block, the following attributes are interpreted:

[19] `book/compiler-plugins.html`

- `link_args` - specify arguments to the linker, rather than just the library name and type. This is feature gated and the exact behavior is implementation-defined (due to variety of linker invocation syntax).
- `link` - indicate that a native library should be linked to for the declarations in this block to be linked correctly. `link` supports an optional `kind` key with three possible values: `dylib`, `static`, and `framework`. See external blocks for more about external blocks. Two examples: `#[link(name = "readline")]` and `#[link(name = "CoreFoundation", kind = "framework")]`.
- `linked_from` - indicates what native library this block of FFI items is coming from. This attribute is of the form `#[linked_from = "foo"]` where `foo` is the name of a library in either `#[link]` or a `-l` flag. This attribute is currently required to export symbols from a Rust dynamic library on Windows, and it is feature gated behind the `linked_from` feature.

On declarations inside an `extern` block, the following attributes are interpreted:

- `link_name` - the name of the symbol that this function or static should be imported as.
- `linkage` - on a static, this specifies the linkage type[20].

On `enum`s:

- `repr` - on C-like enums, this sets the underlying type used for representation. Takes one argument, which is the primitive type this enum should be represented for, or `C`, which specifies that it should be the default `enum` size of the C ABI for that platform. Note that enum representation in C is undefined, and this may be incorrect when the C code is compiled with certain flags.

On `struct`s:

- `repr` - specifies the representation to use for this struct. Takes a list of options. The currently accepted ones are `C` and `packed`, which may be combined. `C` will use a C ABI compatible struct layout, and `packed` will remove any padding between fields (note that this is very fragile and may break platforms which require aligned access).

Macro-related attributes

- `macro_use` on a `mod` — macros defined in this module will be visible in the module's parent, after this module has been included.

- `macro_use` on an `extern crate` — load macros from this crate. An optional list of names `#[macro_use(foo, bar)]` restricts the import to just those macros named. The `extern crate` must appear at the crate root, not inside `mod`, which ensures proper function of the `$crate` macro variable[21].

- `macro_reexport` on an `extern crate` — re-export the named macros.

[20] `http://llvm.org/docs/LangRef.html#linkage-types`
[21] `book/macros.html#the-variable-crate`

- `macro_export` - export a macro for cross-crate usage.
- `no_link` on an `extern crate` — even if we load this crate for macros, don't link it into the output.

See the macros section of the book[22] for more information on macro scope.

Miscellaneous attributes

- `export_name` - on statics and functions, this determines the name of the exported symbol.
- `link_section` - on statics and functions, this specifies the section of the object file that this item's contents will be placed into.
- `no_mangle` - on any item, do not apply the standard name mangling. Set the symbol for this item to its identifier.
- `simd` - on certain tuple structs, derive the arithmetic operators, which lower to the target's SIMD instructions, if any; the `simd` feature gate is necessary to use this attribute.
- `unsafe_destructor_blind_to_params` - on `Drop::drop` method, asserts that the destructor code (and all potential specializations of that code) will never attempt to read from nor write to any references with lifetimes that come in via generic parameters. This is a constraint we cannot currently express via the type system, and therefore we rely on the programmer to assert that it holds. Adding this to a Drop impl causes the associated destructor to be considered "uninteresting" by the Drop-Check rule, and thus it can help sidestep data ordering constraints that would otherwise be introduced by the Drop-Check rule. Such sidestepping of the constraints, if done incorrectly, can lead to undefined behavior (in the form of reading or writing to data outside of its dynamic extent), and thus this attribute has the word "unsafe" in its name. To use this, the `unsafe_destructor_blind_to_params` feature gate must be enabled.
- `unsafe_no_drop_flag` - on structs, remove the flag that prevents destructors from being run twice. Destructors might be run multiple times on the same object with this attribute. To use this, the `unsafe_no_drop_flag` feature gate must be enabled.
- `doc` - Doc comments such as `/// foo` are equivalent to `#[doc = "foo"]`.
- `rustc_on_unimplemented` - Write a custom note to be shown along with the error when the trait is found to be unimplemented on a type. You may use format arguments like `{T}`, `{A}` to correspond to the types at the point of use corresponding to the type parameters of the trait of the same name. `{Self}` will be replaced with the type that is supposed to implement the trait but doesn't. To use this, the `on_unimplemented` feature gate must be enabled.

Conditional compilation

Sometimes one wants to have different compiler outputs from the same code, depending on build target, such as targeted operating system, or to enable release builds.

[22] `book/macros.html#scoping-and-macro-importexport`

There are two kinds of configuration options, one that is either defined or not (`#[cfg(foo)]`), and the other that contains a string that can be checked against (`#[cfg(bar = "baz")]`). Currently, only compiler-defined configuration options can have the latter form.

```
// The function is only included in the build when compiling for OSX
#[cfg(target_os = "macos")]
fn macos_only() {
  // ...
}

// This function is only included when either foo or bar is defined
#[cfg(any(foo, bar))]
fn needs_foo_or_bar() {
  // ...
}

// This function is only included when compiling for a unixish OS with a 32-bit
// architecture
#[cfg(all(unix, target_pointer_width = "32"))]
fn on_32bit_unix() {
  // ...
}

// This function is only included when foo is not defined
#[cfg(not(foo))]
fn needs_not_foo() {
  // ...
}
```

This illustrates some conditional compilation can be achieved using the `#[cfg(...)]` attribute. `any`, `all` and `not` can be used to assemble arbitrarily complex configurations through nesting.

The following configurations must be defined by the implementation:

- `debug_assertions` - Enabled by default when compiling without optimizations. This can be used to enable extra debugging code in development but not in production. For example, it controls the behavior of the standard library's `debug_assert!` macro.
- `target_arch = "..."` - Target CPU architecture, such as "`x86`", "`x86_64`" "`mips`", "`powerpc`", "`arm`", or "`aarch64`".
- `target_endian = "..."` - Endianness of the target CPU, either "`little`" or "`big`".
- `target_env = ".."` - An option provided by the compiler by default describing the runtime environment of the target platform. Some examples of this are `musl` for builds targeting the MUSL libc implementation, `msvc` for Windows builds targeting MSVC, and `gnu` frequently the rest of the time. This option may also be blank on some platforms.
- `target_family = "..."` - Operating system family of the target, e. g. "`unix`" or "`windows`". The value of this configuration option is defined as a configuration itself, like `unix` or `windows`.

- `target_os = "..."` - Operating system of the target, examples include "`windows`", "`macos`", "`ios`", "`linux`", "`android`", "`freebsd`", "`dragonfly`", "`bitrig`", "`openbsd`" or "`netbsd`".
- `target_pointer_width = "..."` - Target pointer width in bits. This is set to "`32`" for targets with 32-bit pointers, and likewise set to "`64`" for 64-bit pointers.
- `target_vendor = "..."` - Vendor of the target, for example `apple`, `pc`, or simply "`unknown`".
- `test` - Enabled when compiling the test harness (using the `--test` flag).
- `unix` - See `target_family`.
- `windows` - See `target_family`.

You can also set another attribute based on a `cfg` variable with `cfg_attr`:

```
#[cfg_attr(a, b)]
```

Will be the same as `#[b]` if `a` is set by `cfg`, and nothing otherwise.

Lint check attributes

A lint check names a potentially undesirable coding pattern, such as unreachable code or omitted documentation, for the static entity to which the attribute applies.

For any lint check `C`:

- `allow(C)` overrides the check for `C` so that violations will go unreported,
- `deny(C)` signals an error after encountering a violation of `C`,
- `forbid(C)` is the same as `deny(C)`, but also forbids changing the lint level afterwards,
- `warn(C)` warns about violations of `C` but continues compilation.

The lint checks supported by the compiler can be found via `rustc -W help`, along with their default settings. Compiler plugins[23] can provide additional lint checks.

```
mod m1 {
    // Missing documentation is ignored here
    #[allow(missing_docs)]
    pub fn undocumented_one() -> i32 { 1 }

    // Missing documentation signals a warning here
    #[warn(missing_docs)]
    pub fn undocumented_too() -> i32 { 2 }

    // Missing documentation signals an error here
    #[deny(missing_docs)]
    pub fn undocumented_end() -> i32 { 3 }
}
```

This example shows how one can use `allow` and `warn` to toggle a particular check on and off:

[23] `book/compiler-plugins.html#lint-plugins`

```
#[warn(missing_docs)]
mod m2{
    #[allow(missing_docs)]
    mod nested {
        // Missing documentation is ignored here
        pub fn undocumented_one() -> i32 { 1 }

        // Missing documentation signals a warning here,
        // despite the allow above.
        #[warn(missing_docs)]
        pub fn undocumented_two() -> i32 { 2 }
    }

    // Missing documentation signals a warning here
    pub fn undocumented_too() -> i32 { 3 }
}
```

This example shows how one can use `forbid` to disallow uses of `allow` for that lint check:

```
#[forbid(missing_docs)]
mod m3 {
    // Attempting to toggle warning signals an error here
    #[allow(missing_docs)]
    /// Returns 2.
    pub fn undocumented_too() -> i32 { 2 }
}
```

Language items

Some primitive Rust operations are defined in Rust code, rather than being implemented directly in C or assembly language. The definitions of these operations have to be easy for the compiler to find. The `lang` attribute makes it possible to declare these operations. For example, the `str` module in the Rust standard library defines the string equality function:

```
#[lang = "str_eq"]
pub fn eq_slice(a: &str, b: &str) -> bool {
    // details elided
}
```

The name `str_eq` has a special meaning to the Rust compiler, and the presence of this definition means that it will use this definition when generating calls to the string equality function.

The set of language items is currently considered unstable. A complete list of the built-in language items will be added in the future.

Inline attributes

The inline attribute suggests that the compiler should place a copy of the function or static in the caller, rather than generating code to call the function or access the static where it is defined.

The compiler automatically inlines functions based on internal heuristics. Incorrectly inlining functions can actually make the program slower, so it should be used with care.

`#[inline]` and `#[inline(always)]` always cause the function to be serialized into the crate metadata to allow cross-crate inlining.

There are three different types of inline attributes:

- `#[inline]` hints the compiler to perform an inline expansion.
- `#[inline(always)]` asks the compiler to always perform an inline expansion.
- `#[inline(never)]` asks the compiler to never perform an inline expansion.

`derive`

The `derive` attribute allows certain traits to be automatically implemented for data structures. For example, the following will create an `impl` for the `PartialEq` and `Clone` traits for `Foo`, the type parameter `T` will be given the `PartialEq` or `Clone` constraints for the appropriate `impl`:

```
#[derive(PartialEq, Clone)]
struct Foo<T> {
    a: i32,
    b: T
}
```

The generated `impl` for `PartialEq` is equivalent to

```
# struct Foo<T> { a: i32, b: T }
impl<T: PartialEq> PartialEq for Foo<T> {
    fn eq(&self, other: &Foo<T>) -> bool {
        self.a == other.a && self.b == other.b
    }

    fn ne(&self, other: &Foo<T>) -> bool {
        self.a != other.a || self.b != other.b
    }
}
```

Compiler Features

Certain aspects of Rust may be implemented in the compiler, but they're not necessarily ready for every-day use. These features are often of "prototype quality" or "almost production ready", but may not be stable enough to be considered a full-fledged language feature.

For this reason, Rust recognizes a special crate-level attribute of the form:

```
#![feature(feature1, feature2, feature3)]
```

This directive informs the compiler that the feature list: `feature1`, `feature2`, and `feature3` should all be enabled. This is only recognized at a crate-level, not at a module-level. Without this directive, all features are considered off, and using the features will result in a compiler error.

The currently implemented features of the reference compiler are:

- `advanced_slice_patterns` - See the match expressions section for discussion; the exact semantics of slice patterns are subject to change, so some types are still unstable.
- `slice_patterns` - OK, actually, slice patterns are just scary and completely unstable.
- `asm` - The `asm!` macro provides a means for inline assembly. This is often useful, but the exact syntax for this feature along with its semantics are likely to change, so this macro usage must be opted into.
- `associated_consts` - Allows constants to be defined in `impl` and `trait` blocks, so that they can be associated with a type or trait in a similar manner to methods and associated types.
- `box_patterns` - Allows `box` patterns, the exact semantics of which is subject to change.
- `box_syntax` - Allows use of `box` expressions, the exact semantics of which is subject to change.
- `cfg_target_vendor` - Allows conditional compilation using the `target_vendor` matcher which is subject to change.
- `concat_idents` - Allows use of the `concat_idents` macro, which is in many ways insufficient for concatenating identifiers, and may be removed entirely for something more wholesome.
- `custom_attribute` - Allows the usage of attributes unknown to the compiler so that new attributes can be added in a backwards compatible manner (RFC 572).
- `custom_derive` - Allows the use of `#[derive(Foo,Bar)]` as sugar for `#[derive_-Foo] #[derive_Bar]`, which can be user-defined syntax extensions.
- `intrinsics` - Allows use of the "rust-intrinsics" ABI. Compiler intrinsics are inherently unstable and no promise about them is made.
- `lang_items` - Allows use of the `#[lang]` attribute. Like `intrinsics`, lang items are inherently unstable and no promise about them is made.
- `link_args` - This attribute is used to specify custom flags to the linker, but usage is strongly discouraged. The compiler's usage of the system linker is not guaranteed to continue in the future, and if the system linker is not used then specifying custom flags doesn't have much meaning.

- `link_llvm_intrinsics` – Allows linking to LLVM intrinsics via `#[link_name="llvm.*"]`.
- `linkage` - Allows use of the `linkage` attribute, which is not portable.
- `log_syntax` - Allows use of the `log_syntax` macro attribute, which is a nasty hack that will certainly be removed.
- `main` - Allows use of the `#[main]` attribute, which changes the entry point into a Rust program. This capability is subject to change.
- `macro_reexport` - Allows macros to be re-exported from one crate after being imported from another. This feature was originally designed with the sole use case of the Rust standard library in mind, and is subject to change.
- `non_ascii_idents` - The compiler supports the use of non-ascii identifiers, but the implementation is a little rough around the edges, so this can be seen as an experimental feature for now until the specification of identifiers is fully fleshed out.
- `no_std` - Allows the `#![no_std]` crate attribute, which disables the implicit `extern crate std`. This typically requires use of the unstable APIs behind the libstd "facade", such as libcore and libcollections. It may also cause problems when using syntax extensions, including `#[derive]`.
- `on_unimplemented` - Allows the `#[rustc_on_unimplemented]` attribute, which allows trait definitions to add specialized notes to error messages when an implementation was expected but not found.
- `optin_builtin_traits` - Allows the definition of default and negative trait implementations. Experimental.
- `plugin` - Usage of compiler plugins[24] for custom lints or syntax extensions. These depend on compiler internals and are subject to change.
- `plugin_registrar` - Indicates that a crate provides compiler plugins[25].
- `quote` - Allows use of the `quote_*!` family of macros, which are implemented very poorly and will likely change significantly with a proper implementation.
- `rustc_attrs` - Gates internal `#[rustc_*]` attributes which may be for internal use only or have meaning added to them in the future.
- `rustc_diagnostic_macros`- A mysterious feature, used in the implementation of rustc, not meant for mortals.
- `simd` - Allows use of the `#[simd]` attribute, which is overly simple and not the SIMD interface we want to expose in the long term.
- `simd_ffi` - Allows use of SIMD vectors in signatures for foreign functions. The SIMD interface is subject to change.

[24] `book/compiler-plugins.html`
[25] `book/compiler-plugins.html`

- `staged_api` - Allows usage of stability markers and `#![staged_api]` in a crate. Stability markers are also attributes: `#[stable]`, `#[unstable]`, and `#[rustc_deprecated]` are the three levels.
- `start` - Allows use of the `#[start]` attribute, which changes the entry point into a Rust program. This capability, especially the signature for the annotated function, is subject to change.
- `thread_local` - The usage of the `#[thread_local]` attribute is experimental and should be seen as unstable. This attribute is used to declare a `static` as being unique per-thread leveraging LLVM's implementation which works in concert with the kernel loader and dynamic linker. This is not necessarily available on all platforms, and usage of it is discouraged.
- `trace_macros` - Allows use of the `trace_macros` macro, which is a nasty hack that will certainly be removed.
- `unboxed_closures` - Rust's new closure design, which is currently a work in progress feature with many known bugs.
- `unsafe_no_drop_flag` - Allows use of the `#[unsafe_no_drop_flag]` attribute, which removes hidden flag added to a type that implements the `Drop` trait. The design for the `Drop` flag is subject to change, and this feature may be removed in the future.
- `unmarked_api` - Allows use of items within a `#![staged_api]` crate which have not been marked with a stability marker. Such items should not be allowed by the compiler to exist, so if you need this there probably is a compiler bug.
- `visible_private_types` - Allows public APIs to expose otherwise private types, e.g. as the return type of a public function. This capability may be removed in the future.
- `allow_internal_unstable` - Allows `macro_rules!` macros to be tagged with the `#[allow_internal_unstable]` attribute, designed to allow `std` macros to call `#[unstable]`/feature-gated functionality internally without imposing on callers (i.e. making them behave like function calls in terms of encapsulation).
- – `default_type_parameter_fallback` - Allows type parameter defaults to influence type inference.
- – `braced_empty_structs` - Allows use of empty structs and enum variants with braces.

If a feature is promoted to a language feature, then all existing programs will start to receive compilation warnings about `#![feature]` directives which enabled the new feature (because the directive is no longer necessary). However, if a feature is decided to be removed from the language, errors will be issued (if there isn't a parser error first). The directive in this case is no longer necessary, and it's likely that existing code will break if the feature isn't removed.

If an unknown feature is found in a directive, it results in a compiler error. An unknown feature is one which has never been recognized by the compiler.

20.7 Statements and expressions

Rust is *primarily* an expression language. This means that most forms of value-producing or effect-causing evaluation are directed by the uniform syntax category of *expressions*. Each kind of expression can typically *nest* within each other kind of expression, and rules for evaluation of expressions involve specifying both the value produced by the expression and the order in which its sub-expressions are themselves evaluated.

In contrast, statements in Rust serve *mostly* to contain and explicitly sequence expression evaluation.

20.7.1 Statements

A *statement* is a component of a block, which is in turn a component of an outer expression or function.

Rust has two kinds of statement: declaration statements and expression statements.

Declaration statements

A *declaration statement* is one that introduces one or more *names* into the enclosing statement block. The declared names may denote new variables or new items.

Item declarations An *item declaration statement* has a syntactic form identical to an item declaration within a module. Declaring an item — a function, enumeration, struct, type, static, trait, implementation or module — locally within a statement block is simply a way of restricting its scope to a narrow region containing all of its uses; it is otherwise identical in meaning to declaring the item outside the statement block.

> **Note**: there is no implicit capture of the function's dynamic environment when declaring a function-local item.

`let` **statements** A `let` *statement* introduces a new set of variables, given by a pattern. The pattern may be followed by a type annotation, and/or an initializer expression. When no type annotation is given, the compiler will infer the type, or signal an error if insufficient type information is available for definite inference. Any variables introduced by a variable declaration are visible from the point of declaration until the end of the enclosing block scope.

Expression statements

An *expression statement* is one that evaluates an expression and ignores its result. The type of an expression statement `e;` is always `()`, regardless of the type of `e`. As a rule, an expression statement's purpose is to trigger the effects of evaluating its expression.

20.7.2 Expressions

An expression may have two roles: it always produces a *value*, and it may have *effects* (otherwise known as "side effects"). An expression *evaluates to* a value, and has effects during *evaluation*. Many expressions contain sub-expressions (operands). The meaning of each kind of expression dictates several things:

- Whether or not to evaluate the sub-expressions when evaluating the expression
- The order in which to evaluate the sub-expressions
- How to combine the sub-expressions' values to obtain the value of the expression

In this way, the structure of expressions dictates the structure of execution. Blocks are just another kind of expression, so blocks, statements, expressions, and blocks again can recursively nest inside each other to an arbitrary depth.

Lvalues, rvalues and temporaries Expressions are divided into two main categories: *lvalues* and *rvalues*. Likewise within each expression, sub-expressions may occur in *lvalue context* or *rvalue context*. The evaluation of an expression depends both on its own category and the context it occurs within.

An lvalue is an expression that represents a memory location. These expressions are paths (which refer to local variables, function and method arguments, or static variables), dereferences (`*expr`), indexing expressions (`expr[expr]`), and field references (`expr.f`). All other expressions are rvalues.

The left operand of an assignment or compound-assignment expression is an lvalue context, as is the single operand of a unary borrow. The discriminant or subject of a match expression may be an lvalue context, if ref bindings are made, but is otherwise an rvalue context. All other expression contexts are rvalue contexts.

When an lvalue is evaluated in an *lvalue context*, it denotes a memory location; when evaluated in an *rvalue context*, it denotes the value held *in* that memory location.

Temporary lifetimes When an rvalue is used in an lvalue context, a temporary unnamed lvalue is created and used instead. The lifetime of temporary values is typically the innermost enclosing statement; the tail expression of a block is considered part of the statement that encloses the block.

When a temporary rvalue is being created that is assigned into a `let` declaration, however, the temporary is created with the lifetime of the enclosing block instead, as using the enclosing statement (the `let` declaration) would be a guaranteed error (since a pointer to the temporary would be stored into a variable, but the temporary would be freed before the variable could be used). The compiler uses simple syntactic rules to decide which values are being assigned into a `let` binding, and therefore deserve a longer temporary lifetime.

Here are some examples:

- `let x = foo(&temp())`. The expression `temp()` is an rvalue. As it is being borrowed, a temporary is created which will be freed after the innermost enclosing statement (the `let` declaration, in this case).

- `let x = temp().foo()`. This is the same as the previous example, except that the value of `temp()` is being borrowed via autoref on a method-call. Here we are assuming that `foo()` is an `&self` method defined in some trait, say `Foo`. In other words, the expression `temp().foo()` is equivalent to `Foo::foo(&temp())`.
- `let x = &temp()`. Here, the same temporary is being assigned into `x`, rather than being passed as a parameter, and hence the temporary's lifetime is considered to be the enclosing block.
- `let x = SomeStruct { foo: &temp() }`. As in the previous case, the temporary is assigned into a struct which is then assigned into a binding, and hence it is given the lifetime of the enclosing block.
- `let x = [ &temp() ]`. As in the previous case, the temporary is assigned into an array which is then assigned into a binding, and hence it is given the lifetime of the enclosing block.
- `let ref x = temp()`. In this case, the temporary is created using a ref binding, but the result is the same: the lifetime is extended to the enclosing block.

Moved and copied types When a local variable is used as an rvalue, the variable will be copied if its type implements `Copy`. All others are moved.

Literal expressions

A *literal expression* consists of one of the literal forms described earlier. It directly describes a number, character, string, boolean value, or the unit value.

```
();        // unit type
"hello";   // string type
'5';       // character type
5;         // integer type
```

Path expressions

A path used as an expression context denotes either a local variable or an item. Path expressions are lvalues.

Tuple expressions

Tuples are written by enclosing zero or more comma-separated expressions in parentheses. They are used to create tuple-typed values.

```
(0.0, 4.5);
("a", 4usize, true);
```

You can disambiguate a single-element tuple from a value in parentheses with a comma:

```
(0,); // single-element tuple
(0); // zero in parentheses
```

Struct expressions

There are several forms of struct expressions. A *struct expression* consists of the path of a struct item, followed by a brace-enclosed list of one or more comma-separated name-value pairs, providing the field values of a new instance of the struct. A field name can be any identifier, and is separated from its value expression by a colon. The location denoted by a struct field is mutable if and only if the enclosing struct is mutable.

A *tuple struct expression* consists of the path of a struct item, followed by a parenthesized list of one or more comma-separated expressions (in other words, the path of a struct item followed by a tuple expression). The struct item must be a tuple struct item.

A *unit-like struct expression* consists only of the path of a struct item.

The following are examples of struct expressions:

```
# struct Point { x: f64, y: f64 }
# struct TuplePoint(f64, f64);
# mod game { pub struct User<'a> { pub name: &'a str, pub age: u32, pub score: usize } }
# struct Cookie; fn some_fn<T>(t: T) {}
Point {x: 10.0, y: 20.0};
TuplePoint(10.0, 20.0);
let u = game::User {name: "Joe", age: 35, score: 100_000};
some_fn::<Cookie>(Cookie);
```

A struct expression forms a new value of the named struct type. Note that for a given *unit-like* struct type, this will always be the same value.

A struct expression can terminate with the syntax `..` followed by an expression to denote a functional update. The expression following `..` (the base) must have the same struct type as the new struct type being formed. The entire expression denotes the result of constructing a new struct (with the same type as the base expression) with the given values for the fields that were explicitly specified and the values in the base expression for all other fields.

```
# struct Point3d { x: i32, y: i32, z: i32 }
let base = Point3d {x: 1, y: 2, z: 3};
Point3d {y: 0, z: 10, .. base};
```

Block expressions

A *block expression* is similar to a module in terms of the declarations that are possible. Each block conceptually introduces a new namespace scope. Use items can bring new names into scopes and declared items are in scope for only the block itself.

A block will execute each statement sequentially, and then execute the expression (if given). If the block ends in a statement, its value is `()`:

```
let x: () = { println!("Hello."); };
```

If it ends in an expression, its value and type are that of the expression:

```
let x: i32 = { println!("Hello."); 5 };

assert_eq!(5, x);
```

Method-call expressions

A *method call* consists of an expression followed by a single dot, an identifier, and a parenthesized expression-list. Method calls are resolved to methods on specific traits, either statically dispatching to a method if the exact `self`-type of the left-hand-side is known, or dynamically dispatching if the left-hand-side expression is an indirect trait object.

Field expressions

A *field expression* consists of an expression followed by a single dot and an identifier, when not immediately followed by a parenthesized expression-list (the latter is a method call expression). A field expression denotes a field of a struct.

```
mystruct.myfield;
foo().x;
(Struct {a: 10, b: 20}).a;
```

A field access is an lvalue referring to the value of that field. When the type providing the field inherits mutability, it can be assigned to.

Also, if the type of the expression to the left of the dot is a pointer, it is automatically dereferenced as many times as necessary to make the field access possible. In cases of ambiguity, we prefer fewer autoderefs to more.

Array expressions

An array *expression* is written by enclosing zero or more comma-separated expressions of uniform type in square brackets.

In the `[expr ';' expr]` form, the expression after the `';'` must be a constant expression that can be evaluated at compile time, such as a literal or a static item.

```
[1, 2, 3, 4];
["a", "b", "c", "d"];
[0; 128];              // array with 128 zeros
[0u8, 0u8, 0u8, 0u8];
```

Index expressions

Array-typed expressions can be indexed by writing a square-bracket-enclosed expression (the index) after them. When the array is mutable, the resulting lvalue can be assigned to.

Indices are zero-based, and may be of any integral type. Vector access is bounds-checked at compile-time for constant arrays being accessed with a constant index

value. Otherwise a check will be performed at run-time that will put the thread in a *panicked state* if it fails.

```
([1, 2, 3, 4])[0];

let x = (["a", "b"])[10]; // compiler error: const index-expr is out of bounds

let n = 10;
let y = (["a", "b"])[n]; // panics

let arr = ["a", "b"];
arr[10]; // panics
```

Also, if the type of the expression to the left of the brackets is a pointer, it is automatically dereferenced as many times as necessary to make the indexing possible. In cases of ambiguity, we prefer fewer autoderefs to more.

Range expressions

The `..` operator will construct an object of one of the `std::ops::Range` variants.

```
1..2;   // std::ops::Range
3..;    // std::ops::RangeFrom
..4;    // std::ops::RangeTo
..;     // std::ops::RangeFull
```

The following expressions are equivalent.

```
let x = std::ops::Range {start: 0, end: 10};
let y = 0..10;

assert_eq!(x, y);
```

Unary operator expressions

Rust defines the following unary operators. They are all written as prefix operators, before the expression they apply to.

- `-` Negation. May only be applied to numeric types.
- `*` Dereference. When applied to a pointer it denotes the pointed-to location. For pointers to mutable locations, the resulting lvalue can be assigned to. On non-pointer types, it calls the `deref` method of the `std::ops::Deref` trait, or the `deref_mut` method of the `std::ops::DerefMut` trait (if implemented by the type and required for an outer expression that will or could mutate the dereference), and produces the result of dereferencing the `&` or `&mut` borrowed pointer returned from the overload method.

- `!` Logical negation. On the boolean type, this flips between `true` and `false`. On integer types, this inverts the individual bits in the two's complement representation of the value.

- `&` **and** `&mut` Borrowing. When applied to an lvalue, these operators produce a reference (pointer) to the lvalue. The lvalue is also placed into a borrowed state for the duration of the reference. For a shared borrow (`&`), this implies that the lvalue may not be mutated, but it may be read or shared again. For a mutable borrow (`&mut`), the lvalue may not be accessed in any way until the borrow expires. If the `&` or `&mut` operators are applied to an rvalue, a temporary value is created; the lifetime of this temporary value is defined by syntactic rules.

Binary operator expressions

Binary operators expressions are given in terms of operator precedence.

Arithmetic operators Binary arithmetic expressions are syntactic sugar for calls to built-in traits, defined in the `std::ops` module of the `std` library. This means that arithmetic operators can be overridden for user-defined types. The default meaning of the operators on standard types is given here.

- `+` Addition and array/string concatenation. Calls the `add` method on the `std::ops::Add` trait.

- `-` Subtraction. Calls the `sub` method on the `std::ops::Sub` trait.

- `*` Multiplication. Calls the `mul` method on the `std::ops::Mul` trait.

- `/` Quotient. Calls the `div` method on the `std::ops::Div` trait.

- `%` Remainder. Calls the `rem` method on the `std::ops::Rem` trait.

Bitwise operators Like the arithmetic operators, bitwise operators are syntactic sugar for calls to methods of built-in traits. This means that bitwise operators can be overridden for user-defined types. The default meaning of the operators on standard types is given here. Bitwise `&`, `|` and `^` applied to boolean arguments are equivalent to logical `&&`, `||` and `!=` evaluated in non-lazy fashion.

- `&` Bitwise AND. Calls the `bitand` method of the `std::ops::BitAnd` trait.

- `|` Bitwise inclusive OR. Calls the `bitor` method of the `std::ops::BitOr` trait.

- `^` Bitwise exclusive OR. Calls the `bitxor` method of the `std::ops::BitXor` trait.

- `«` Left shift. Calls the `shl` method of the `std::ops::Shl` trait.

- `»` Right shift (arithmetic). Calls the `shr` method of the `std::ops::Shr` trait.

Lazy boolean operators The operators `||` and `&&` may be applied to operands of boolean type. The `||` operator denotes logical 'or', and the `&&` operator denotes logical 'and'. They differ from `|` and `&` in that the right-hand operand is only evaluated when the left-hand operand does not already determine the result of the expression. That is, `||` only evaluates its right-hand operand when the left-hand operand evaluates to `false`, and `&&` only when it evaluates to `true`.

Comparison operators Comparison operators are, like the arithmetic operators, and bitwise operators, syntactic sugar for calls to built-in traits. This means that comparison operators can be overridden for user-defined types. The default meaning of the operators on standard types is given here.

- `==` Equal to. Calls the `eq` method on the `std::cmp::PartialEq` trait.
- `!=` Unequal to. Calls the `ne` method on the `std::cmp::PartialEq` trait.
- `<` Less than. Calls the `lt` method on the `std::cmp::PartialOrd` trait.
- `>` Greater than. Calls the `gt` method on the `std::cmp::PartialOrd` trait.
- `<=` Less than or equal. Calls the `le` method on the `std::cmp::PartialOrd` trait.
- `>=` Greater than or equal. Calls the `ge` method on the `std::cmp::PartialOrd` trait.

Type cast expressions A type cast expression is denoted with the binary operator `as`.

Executing an `as` expression casts the value on the left-hand side to the type on the right-hand side.

An example of an `as` expression:

```
# fn sum(values: &[f64]) -> f64 { 0.0 }
# fn len(values: &[f64]) -> i32 { 0 }

fn average(values: &[f64]) -> f64 {
    let sum: f64 = sum(values);
    let size: f64 = len(values) as f64;
    sum / size
}
```

Some of the conversions which can be done through the `as` operator can also be done implicitly at various points in the program, such as argument passing and assignment to a `let` binding with an explicit type. Implicit conversions are limited to "harmless" conversions that do not lose information and which have minimal or no risk of surprising side-effects on the dynamic execution semantics.

Assignment expressions An *assignment expression* consists of an lvalue expression followed by an equals sign (=) and an rvalue expression.

Evaluating an assignment expression either copies or moves its right-hand operand to its left-hand operand.

```
# let mut x = 0;
# let y = 0;
x = y;
```

Compound assignment expressions The +, −, *, /, %, &, |, ^, «, and » operators may be composed with the = operator. The expression `lval OP= val` is equivalent to `lval = lval OP val`. For example, `x = x + 1` may be written as `x += 1`.

Any such expression always has the `unit` type.

Operator precedence The precedence of Rust binary operators is ordered as follows, going from strong to weak:

```
as
* / %
+ -
<< >>
&
^
|
== != < > <= >=
&&
||
= ..
```

Operators at the same precedence level are evaluated left-to-right. Unary operators have the same precedence level and are stronger than any of the binary operators.

Grouped expressions

An expression enclosed in parentheses evaluates to the result of the enclosed expression. Parentheses can be used to explicitly specify evaluation order within an expression.

An example of a parenthesized expression:

```
let x: i32 = (2 + 3) * 4;
```

Call expressions

A *call expression* invokes a function, providing zero or more input variables and an optional location to move the function's output into. If the function eventually returns, then the expression completes.

Some examples of call expressions:

```
# fn add(x: i32, y: i32) -> i32 { 0 }

let x: i32 = add(1i32, 2i32);
let pi: Result<f32, _> = "3.14".parse();
```

Lambda expressions

A *lambda expression* (sometimes called an "anonymous function expression") defines a function and denotes it as a value, in a single expression. A lambda expression is a pipe-symbol-delimited (`|`) list of identifiers followed by an expression.

A lambda expression denotes a function that maps a list of parameters (`ident_list`) onto the expression that follows the `ident_list`. The identifiers in the `ident_list` are the parameters to the function. These parameters' types need not be specified, as the compiler infers them from context.

Lambda expressions are most useful when passing functions as arguments to other functions, as an abbreviation for defining and capturing a separate function.

Significantly, lambda expressions *capture their environment*, which regular function definitions do not. The exact type of capture depends on the function type inferred for the lambda expression. In the simplest and least-expensive form (analogous to a `|| { }` expression), the lambda expression captures its environment by reference, effectively borrowing pointers to all outer variables mentioned inside the function. Alternately, the compiler may infer that a lambda expression should copy or move values (depending on their type) from the environment into the lambda expression's captured environment.

In this example, we define a function `ten_times` that takes a higher-order function argument, and we then call it with a lambda expression as an argument:

```
fn ten_times<F>(f: F) where F: Fn(i32) {
    for index in 0..10 {
        f(index);
    }
}

ten_times(|j| println!("hello, {}", j));
```

Infinite loops

A `loop` expression denotes an infinite loop.

A `loop` expression may optionally have a *label*. The label is written as a lifetime preceding the loop expression, as in `'foo: loop{ }`. If a label is present, then labeled `break` and `continue` expressions nested within this loop may exit out of this loop or return control to its head. See break expressions and continue expressions.

`break` expressions

A `break` expression has an optional *label*. If the label is absent, then executing a `break` expression immediately terminates the innermost loop enclosing it. It is only

permitted in the body of a loop. If the label is present, then `break 'foo` terminates the loop with label `'foo`, which need not be the innermost label enclosing the `break` expression, but must enclose it.

`continue` expressions

A `continue` expression has an optional *label*. If the label is absent, then executing a `continue` expression immediately terminates the current iteration of the innermost loop enclosing it, returning control to the loop *head*. In the case of a `while` loop, the head is the conditional expression controlling the loop. In the case of a `for` loop, the head is the call-expression controlling the loop. If the label is present, then `continue 'foo` returns control to the head of the loop with label `'foo`, which need not be the innermost label enclosing the `break` expression, but must enclose it.

A `continue` expression is only permitted in the body of a loop.

`while` loops

A `while` loop begins by evaluating the boolean loop conditional expression. If the loop conditional expression evaluates to `true`, the loop body block executes and control returns to the loop conditional expression. If the loop conditional expression evaluates to `false`, the `while` expression completes.

An example:

```
let mut i = 0;

while i < 10 {
    println!("hello");
    i = i + 1;
}
```

Like `loop` expressions, `while` loops can be controlled with `break` or `continue`, and may optionally have a *label*. See infinite loops, break expressions, and continue expressions for more information.

`for` expressions

A `for` expression is a syntactic construct for looping over elements provided by an implementation of `std::iter::IntoIterator`.

An example of a `for` loop over the contents of an array:

```
# type Foo = i32;
# fn bar(f: &Foo) { }
# let a = 0;
# let b = 0;
# let c = 0;

let v: &[Foo] = &[a, b, c];
```

```
for e in v {
    bar(e);
}
```

An example of a for loop over a series of integers:

```
# fn bar(b:usize) { }
for i in 0..256 {
    bar(i);
}
```

Like `loop` expressions, `for` loops can be controlled with `break` or `continue`, and may optionally have a *label*. See infinite loops, break expressions, and continue expressions for more information.

`if` expressions

An `if` expression is a conditional branch in program control. The form of an `if` expression is a condition expression, followed by a consequent block, any number of `else if` conditions and blocks, and an optional trailing `else` block. The condition expressions must have type `bool`. If a condition expression evaluates to `true`, the consequent block is executed and any subsequent `else if` or `else` block is skipped. If a condition expression evaluates to `false`, the consequent block is skipped and any subsequent `else if` condition is evaluated. If all `if` and `else if` conditions evaluate to `false` then any `else` block is executed.

`match` expressions

A `match` expression branches on a *pattern*. The exact form of matching that occurs depends on the pattern. Patterns consist of some combination of literals, destructured arrays or enum constructors, structs and tuples, variable binding specifications, wildcards (`..`), and placeholders (`_`). A `match` expression has a *head expression*, which is the value to compare to the patterns. The type of the patterns must equal the type of the head expression.

In a pattern whose head expression has an `enum` type, a placeholder (`_`) stands for a *single* data field, whereas a wildcard `..` stands for *all* the fields of a particular variant.

A `match` behaves differently depending on whether or not the head expression is an lvalue or an rvalue. If the head expression is an rvalue, it is first evaluated into a temporary location, and the resulting value is sequentially compared to the patterns in the arms until a match is found. The first arm with a matching pattern is chosen as the branch target of the `match`, any variables bound by the pattern are assigned to local variables in the arm's block, and control enters the block.

When the head expression is an lvalue, the match does not allocate a temporary location (however, a by-value binding may copy or move from the lvalue). When possible, it is preferable to match on lvalues, as the lifetime of these matches inherits the lifetime of the lvalue, rather than being restricted to the inside of the match.

An example of a `match` expression:

```
let x = 1;

match x {
    1 => println!(”one”),
    2 => println!(”two”),
    3 => println!(”three”),
    4 => println!(”four”),
    5 => println!(”five”),
    _ => println!(”something else”),
}
```

Patterns that bind variables default to binding to a copy or move of the matched value (depending on the matched value’s type). This can be changed to bind to a reference by using the `ref` keyword, or to a mutable reference using `ref mut`.

Subpatterns can also be bound to variables by the use of the syntax `variable @ subpattern`. For example:

```
let x = 1;

match x {
    e @ 1 ... 5 => println!(”got a range element {}”, e),
    _ => println!(”anything”),
}
```

Patterns can also dereference pointers by using the `&`, `&mut` and `box` symbols, as appropriate. For example, these two matches on `x: &i32` are equivalent:

```
# let x = &3;
let y = match *x { 0 => ”zero”, _ => ”some” };
let z = match x { &0 => ”zero”, _ => ”some” };

assert_eq!(y, z);
```

Multiple match patterns may be joined with the `|` operator. A range of values may be specified with `...`. For example:

```
# let x = 2;

let message = match x {
    0 | 1  => ”not many”,
    2 ... 9 => ”a few”,
    _       => ”lots”
};
```

Range patterns only work on scalar types (like integers and characters; not like arrays and structs, which have sub-components). A range pattern may not be a sub-range of another range pattern inside the same `match`.

Finally, match patterns can accept *pattern guards* to further refine the criteria for matching a case. Pattern guards appear after the pattern and consist of a bool-typed

expression following the `if` keyword. A pattern guard may refer to the variables bound within the pattern they follow.

```
# let maybe_digit = Some(0);
# fn process_digit(i: i32) { }
# fn process_other(i: i32) { }

let message = match maybe_digit {
    Some(x) if x < 10 => process_digit(x),
    Some(x) => process_other(x),
    None => panic!()
};
```

`if let` expressions

An `if let` expression is semantically identical to an `if` expression but in place of a condition expression it expects a `let` statement with a refutable pattern. If the value of the expression on the right hand side of the `let` statement matches the pattern, the corresponding block will execute, otherwise flow proceeds to the first `else` block that follows.

```
let dish = (”Ham”, ”Eggs”);

// this body will be skipped because the pattern is refuted
if let (”Bacon”, b) = dish {
    println!(”Bacon is served with {}”, b);
}

// this body will execute
if let (”Ham”, b) = dish {
    println!(”Ham is served with {}”, b);
}
```

`while let` loops

A `while let` loop is semantically identical to a `while` loop but in place of a condition expression it expects `let` statement with a refutable pattern. If the value of the expression on the right hand side of the `let` statement matches the pattern, the loop body block executes and control returns to the pattern matching statement. Otherwise, the while expression completes.

`return` expressions

Return expressions are denoted with the keyword `return`. Evaluating a `return` expression moves its argument into the designated output location for the current function call, destroys the current function activation frame, and transfers control to the caller frame.

An example of a `return` expression:

```
fn max(a: i32, b: i32) -> i32 {
    if a > b {
        return a;
    }
    return b;
}
```

20.8 Type system

20.8.1 Types

Every variable, item and value in a Rust program has a type. The *type* of a *value* defines the interpretation of the memory holding it.

Built-in types and type-constructors are tightly integrated into the language, in non-trivial ways that are not possible to emulate in user-defined types. User-defined types have limited capabilities.

Primitive types

The primitive types are the following:

- The boolean type `bool` with values `true` and `false`.
- The machine types (integer and floating-point).
- The machine-dependent integer types.

Machine types The machine types are the following:

- The unsigned word types `u8`, `u16`, `u32` and `u64`, with values drawn from the integer intervals [0, 2^8 - 1], [0, 2^16 - 1], [0, 2^32 - 1] and [0, 2^64 - 1] respectively.
- The signed two's complement word types `i8`, `i16`, `i32` and `i64`, with values drawn from the integer intervals [-(2^(7)), 2^7 - 1], [-(2^(15)), 2^15 - 1], [-(2^(31)), 2^31 - 1], [-(2^(63)), 2^63 - 1] respectively.
- The IEEE 754-2008 `binary32` and `binary64` floating-point types: `f32` and `f64`, respectively.

Machine-dependent integer types The `usize` type is an unsigned integer type with the same number of bits as the platform's pointer type. It can represent every memory address in the process.

The `isize` type is a signed integer type with the same number of bits as the platform's pointer type. The theoretical upper bound on object and array size is the maximum `isize` value. This ensures that `isize` can be used to calculate differences between pointers into an object or array and can address every byte within an object along with one byte past the end.

Textual types

The types `char` and `str` hold textual data.

A value of type `char` is a Unicode scalar value[26] (i.e. a code point that is not a surrogate), represented as a 32-bit unsigned word in the 0x0000 to 0xD7FF or 0xE000 to 0x10FFFF range. A `[char]` array is effectively an UCS-4 / UTF-32 string.

A value of type `str` is a Unicode string, represented as an array of 8-bit unsigned bytes holding a sequence of UTF-8 code points. Since `str` is of unknown size, it is not a *first-class* type, but can only be instantiated through a pointer type, such as `&str`.

Tuple types

A tuple *type* is a heterogeneous product of other types, called the *elements* of the tuple. It has no nominal name and is instead structurally typed.

Tuple types and values are denoted by listing the types or values of their elements, respectively, in a parenthesized, comma-separated list.

Because tuple elements don't have a name, they can only be accessed by pattern-matching or by using `N` directly as a field to access the `N`th element.

An example of a tuple type and its use:

```
type Pair<'a> = (i32, &'a str);
let p: Pair<'static> = (10, "ten");
let (a, b) = p;

assert_eq!(a, 10);
assert_eq!(b, "ten");
assert_eq!(p.0, 10);
assert_eq!(p.1, "ten");
```

For historical reasons and convenience, the tuple type with no elements (`()`) is often called 'unit' or 'the unit type'.

Array, and Slice types

Rust has two different types for a list of items:

- `[T; N]`, an 'array'
- `&[T]`, a 'slice'

An array has a fixed size, and can be allocated on either the stack or the heap.

A slice is a 'view' into an array. It doesn't own the data it points to, it borrows it.

Examples:

[26] `http://www.unicode.org/glossary/#unicode_scalar_value`

```
// A stack-allocated array
let array: [i32; 3] = [1, 2, 3];

// A heap-allocated array
let vector: Vec<i32> = vec![1, 2, 3];

// A slice into an array
let slice: &[i32] = &vector[..];
```

As you can see, the `vec!` macro allows you to create a `Vec<T>` easily. The `vec!` macro is also part of the standard library, rather than the language.

All in-bounds elements of arrays and slices are always initialized, and access to an array or slice is always bounds-checked.

Struct types

A `struct` *type* is a heterogeneous product of other types, called the *fields* of the type.[27]

New instances of a `struct` can be constructed with a struct expression.

The memory layout of a `struct` is undefined by default to allow for compiler optimizations like field reordering, but it can be fixed with the `#[repr(...)]` attribute. In either case, fields may be given in any order in a corresponding struct *expression*; the resulting `struct` value will always have the same memory layout.

The fields of a `struct` may be qualified by visibility modifiers, to allow access to data in a struct outside a module.

A *tuple struct* type is just like a struct type, except that the fields are anonymous.

A *unit-like struct* type is like a struct type, except that it has no fields. The one value constructed by the associated struct expression is the only value that inhabits such a type.

Enumerated types

An *enumerated type* is a nominal, heterogeneous disjoint union type, denoted by the name of an `enum` item. [28]

An `enum` item declares both the type and a number of *variant constructors*, each of which is independently named and takes an optional tuple of arguments.

New instances of an `enum` can be constructed by calling one of the variant constructors, in a call expression.

Any `enum` value consumes as much memory as the largest variant constructor for its corresponding `enum` type.

Enum types cannot be denoted *structurally* as types, but must be denoted by named reference to an `enum` item.

[27] `struct` types are analogous to `struct` types in C, the *record* types of the ML family, or the *struct* types of the Lisp family.

[28] The `enum` type is analogous to a `data` constructor declaration in ML, or a *pick ADT* in Limbo.

Recursive types

Nominal types — enumerations and structs — may be recursive. That is, each `enum` constructor or `struct` field may refer, directly or indirectly, to the enclosing `enum` or `struct` type itself. Such recursion has restrictions:

- Recursive types must include a nominal type in the recursion (not mere type definitions[29], or other structural types such as arrays or tuples).
- A recursive `enum` item must have at least one non-recursive constructor (in order to give the recursion a basis case).
- The size of a recursive type must be finite; in other words the recursive fields of the type must be pointer types.
- Recursive type definitions can cross module boundaries, but not module *visibility* boundaries, or crate boundaries (in order to simplify the module system and type checker).

An example of a *recursive* type and its use:

```
enum List<T> {
    Nil,
    Cons(T, Box<List<T>>)
}

let a: List<i32> = List::Cons(7, Box::new(List::Cons(13, Box::new(List::Nil))));
```

Pointer types

All pointers in Rust are explicit first-class values. They can be copied, stored into data structs, and returned from functions. There are two varieties of pointer in Rust:

- **References (`&`)** These point to memory *owned by some other value.* A reference type is written `&type`, or `&'a type` when you need to specify an explicit lifetime. Copying a reference is a "shallow" operation: it involves only copying the pointer itself. Releasing a reference has no effect on the value it points to, but a reference of a temporary value will keep it alive during the scope of the reference itself.
- **Raw pointers (`*`)** Raw pointers are pointers without safety or liveness guarantees. Raw pointers are written as `*const T` or `*mut T`, for example `*const i32` means a raw pointer to a 32-bit integer. Copying or dropping a raw pointer has no effect on the lifecycle of any other value. Dereferencing a raw pointer or converting it to any other pointer type is an `unsafe` operation. Raw pointers are generally discouraged in Rust code; they exist to support interoperability with foreign code, and writing performance-critical or low-level functions.

The standard library contains additional 'smart pointer' types beyond references and raw pointers.

[29] `grammar.html#type-definitions`

Function types

The function type constructor `fn` forms new function types. A function type consists of a possibly-empty set of function-type modifiers (such as `unsafe` or `extern`), a sequence of input types and an output type.

An example of a `fn` type:

```
fn add(x: i32, y: i32) -> i32 {
    return x + y;
}

let mut x = add(5,7);

type Binop = fn(i32, i32) -> i32;
let bo: Binop = add;
x = bo(5,7);
```

Function types for specific items Internal to the compiler, there are also function types that are specific to a particular function item. In the following snippet, for example, the internal types of the functions `foo` and `bar` are different, despite the fact that they have the same signature:

```
fn foo() { }
fn bar() { }
```

The types of `foo` and `bar` can both be implicitly coerced to the fn pointer type `fn()`. There is currently no syntax for unique fn types, though the compiler will emit a type like `fn() {foo}` in error messages to indicate "the unique fn type for the function `foo`".

Closure types

A lambda expression produces a closure value with a unique, anonymous type that cannot be written out.

Depending on the requirements of the closure, its type implements one or more of the closure traits:

- `FnOnce` The closure can be called once. A closure called as `FnOnce` can move out values from its environment.
- `FnMut` The closure can be called multiple times as mutable. A closure called as `FnMut` can mutate values from its environment. `FnMut` inherits from `FnOnce` (i.e. anything implementing `FnMut` also implements `FnOnce`).
- `Fn` The closure can be called multiple times through a shared reference. A closure called as `Fn` can neither move out from nor mutate values from its environment. `Fn` inherits from `FnMut`, which itself inherits from `FnOnce`.

Trait objects

In Rust, a type like `&SomeTrait` or `Box<SomeTrait>` is called a *trait object.* Each instance of a trait object includes:

- a pointer to an instance of a type `T` that implements `SomeTrait`
- a *virtual method table,* often just called a *vtable,* which contains, for each method of `SomeTrait` that `T` implements, a pointer to `T`'s implementation (i.e. a function pointer).

The purpose of trait objects is to permit "late binding" of methods. A call to a method on a trait object is only resolved to a vtable entry at compile time. The actual implementation for each vtable entry can vary on an object-by-object basis.

Note that for a trait object to be instantiated, the trait must be *object-safe.* Object safety rules are defined in RFC 255[30].

Given a pointer-typed expression `E` of type `&T` or `Box<T>`, where `T` implements trait `R`, casting `E` to the corresponding pointer type `&R` or `Box<R>` results in a value of the *trait object* `R`. This result is represented as a pair of pointers: the vtable pointer for the `T` implementation of `R`, and the pointer value of `E`.

An example of a trait object:

```
trait Printable {
    fn stringify(&self) -> String;
}

impl Printable for i32 {
    fn stringify(&self) -> String { self.to_string() }
}

fn print(a: Box<Printable>) {
    println!("{}", a.stringify());
}

fn main() {
    print(Box::new(10) as Box<Printable>);
}
```

In this example, the trait `Printable` occurs as a trait object in both the type signature of `print`, and the cast expression in `main`.

Type parameters

Within the body of an item that has type parameter declarations, the names of its type parameters are types:

[30] `https://github.com/rust-lang/rfcs/blob/master/text/0255-object-safety.md`

```
fn to_vec<A: Clone>(xs: &[A]) -> Vec<A> {
    if xs.is_empty() {
        return vec![];
    }
    let first: A = xs[0].clone();
    let mut rest: Vec<A> = to_vec(&xs[1..]);
    rest.insert(0, first);
    rest
}
```

Here, `first` has type `A`, referring to `to_vec`'s `A` type parameter; and `rest` has type `Vec<A>`, a vector with element type `A`.

Self types

The special type `Self` has a meaning within traits and impls. In a trait definition, it refers to an implicit type parameter representing the "implementing" type. In an impl, it is an alias for the implementing type. For example, in:

```
trait Printable {
    fn make_string(&self) -> String;
}

impl Printable for String {
    fn make_string(&self) -> String {
        (*self).clone()
    }
}
```

The notation `&self` is a shorthand for `self: &Self`. In this case, in the impl, `Self` refers to the value of type `String` that is the receiver for a call to the method `make_string`.

20.8.2 Subtyping

Subtyping is implicit and can occur at any stage in type checking or inference. Subtyping in Rust is very restricted and occurs only due to variance with respect to lifetimes and between types with higher ranked lifetimes. If we were to erase lifetimes from types, then the only subtyping would be due to type equality.

Consider the following example: string literals always have `'static` lifetime. Nevertheless, we can assign `s` to `t`:

```
fn bar<'a>() {
    let s: &'static str = "hi";
    let t: &'a str = s;
}
```

Since `'static` "lives longer" than `'a`, `&'static str` is a subtype of `&'a str`.

20.8.3 Type coercions

Coercions are defined in RFC401[31]. A coercion is implicit and has no syntax.

Coercion sites

A coercion can only occur at certain coercion sites in a program; these are typically places where the desired type is explicit or can be derived by propagation from explicit types (without type inference). Possible coercion sites are:

- `let` statements where an explicit type is given.

For example, `128` is coerced to have type `i8` in the following:

```
rust    let _: i8 = 128;
```

- `static` and `const` statements (similar to `let` statements).
- Arguments for function calls

The value being coerced is the actual parameter, and it is coerced to the type of the formal parameter.
For example, `128` is coerced to have type `i8` in the following:
"‘rust fn bar(_: i8) { }
fn main() { bar(128); } "‘

- Instantiations of struct or variant fields

For example, `128` is coerced to have type `i8` in the following:
"‘rust struct Foo { x: i8 }
fn main() { Foo { x: 128 }; } "‘

- Function results, either the final line of a block if it is not semicolon-terminated or any expression in a `return` statement

For example, `128` is coerced to have type `i8` in the following:

```
rust   fn foo() -> i8 {       128   }
```

If the expression in one of these coercion sites is a coercion-propagating expression, then the relevant sub-expressions in that expression are also coercion sites. Propagation recurses from these new coercion sites. Propagating expressions and their relevant sub-expressions are:

- Array literals, where the array has type `[U; n]`. Each sub-expression in the array literal is a coercion site for coercion to type `U`.
- Array literals with repeating syntax, where the array has type `[U; n]`. The repeated sub-expression is a coercion site for coercion to type `U`.

[31] `https://github.com/rust-lang/rfcs/blob/master/text/0401-coercions.md`

- Tuples, where a tuple is a coercion site to type `(U_0, U_1, ..., U_n)`. Each sub-expression is a coercion site to the respective type, e.g. the zeroth sub-expression is a coercion site to type `U_0`.
- Parenthesized sub-expressions (`(e)`): if the expression has type `U`, then the sub-expression is a coercion site to `U`.
- Blocks: if a block has type `U`, then the last expression in the block (if it is not semicolon-terminated) is a coercion site to `U`. This includes blocks which are part of control flow statements, such as `if`/`else`, if the block has a known type.

Coercion types

Coercion is allowed between the following types:

- `T` to `U` if `T` is a subtype of `U` (*reflexive case*)
- `T_1` to `T_3` where `T_1` coerces to `T_2` and `T_2` coerces to `T_3` (*transitive case*)

  Note that this is not fully supported yet
- `&mut T` to `&T`
- `*mut T` to `*const T`
- `&T` to `*const T`
- `&mut T` to `*mut T`
- `&T` to `&U` if `T` implements `Deref<Target = U>`. For example:

“‘rust use std::ops::Deref;
struct CharContainer { value: char }
impl Deref for CharContainer { type Target = char;

```
  fn deref<'a>(&'a self) -> &'a char {
      &self.value
  }
```

}
fn foo(arg: &char) {}
fn main() { let x = &mut CharContainer { value: ‘y’ }; foo(x); //&mut CharContainer is coerced to &char. } “‘

- `&mut T` to `&mut U` if `T` implements `DerefMut<Target = U>`.
- TyCtor(`T`) to TyCtor(coerce_inner(`T`)), where TyCtor(`T`) is one of
  - `&T`
  - `&mut T`
  - `*const T`

- `*mut T`
- `Box<T>`

and where

- coerce_inner(`[T, ..n]`) = `[T]`
- coerce_inner(`T`) = `U` where `T` is a concrete type which implements the trait `U`.

In the future, coerce_inner will be recursively extended to tuples and structs. In addition, coercions from sub-traits to super-traits will be added. See RFC401[32] for more details.

20.9 Special traits

Several traits define special evaluation behavior.

20.9.1 The `Copy` trait

The `Copy` trait changes the semantics of a type implementing it. Values whose type implements `Copy` are copied rather than moved upon assignment.

20.9.2 The `Sized` trait

The `Sized` trait indicates that the size of this type is known at compile-time.

20.9.3 The `Drop` trait

The `Drop` trait provides a destructor, to be run whenever a value of this type is to be destroyed.

20.9.4 The `Deref` trait

The `Deref<Target = U>` trait allows a type to implicitly implement all the methods of the type `U`. When attempting to resolve a method call, the compiler will search the top-level type for the implementation of the called method. If no such method is found, `.deref()` is called and the compiler continues to search for the method implementation in the returned type `U`.

[32] `https://github.com/rust-lang/rfcs/blob/master/text/0401-coercions.md`

20.10 Memory model

A Rust program's memory consists of a static set of *items* and a *heap*. Immutable portions of the heap may be safely shared between threads, mutable portions may not be safely shared, but several mechanisms for effectively-safe sharing of mutable values, built on unsafe code but enforcing a safe locking discipline, exist in the standard library.

Allocations in the stack consist of *variables*, and allocations in the heap consist of *boxes*.

Memory allocation and lifetime

The *items* of a program are those functions, modules and types that have their value calculated at compile-time and stored uniquely in the memory image of the rust process. Items are neither dynamically allocated nor freed.

The *heap* is a general term that describes boxes. The lifetime of an allocation in the heap depends on the lifetime of the box values pointing to it. Since box values may themselves be passed in and out of frames, or stored in the heap, heap allocations may outlive the frame they are allocated within.

Memory ownership

When a stack frame is exited, its local allocations are all released, and its references to boxes are dropped.

Variables

A *variable* is a component of a stack frame, either a named function parameter, an anonymous temporary, or a named local variable.

A *local variable* (or *stack-local* allocation) holds a value directly, allocated within the stack's memory. The value is a part of the stack frame.

Local variables are immutable unless declared otherwise like: `let mut x = ...`.

Function parameters are immutable unless declared with `mut`. The `mut` keyword applies only to the following parameter (so `|mut x, y|` and `fn f(mut x: Box<i32>, y: Box<i32>)` declare one mutable variable `x` and one immutable variable `y`).

Methods that take either `self` or `Box<Self>` can optionally place them in a mutable variable by prefixing them with `mut` (similar to regular arguments):

```
trait Changer {
    fn change(mut self) -> Self;
    fn modify(mut self: Box<Self>) -> Box<Self>;
}
```

Local variables are not initialized when allocated; the entire frame worth of local variables are allocated at once, on frame-entry, in an uninitialized state. Subsequent statements within a function may or may not initialize the local variables. Local variables can be used only after they have been initialized; this is enforced by the compiler.

20.11 Linkage

The Rust compiler supports various methods to link crates together both statically and dynamically. This section will explore the various methods to link Rust crates together, and more information about native libraries can be found in the FFI section of the book[33].

In one session of compilation, the compiler can generate multiple artifacts through the usage of either command line flags or the `crate_type` attribute. If one or more command line flags are specified, all `crate_type` attributes will be ignored in favor of only building the artifacts specified by command line.

- `--crate-type=bin`, `#[crate_type = "bin"]` - A runnable executable will be produced. This requires that there is a `main` function in the crate which will be run when the program begins executing. This will link in all Rust and native dependencies, producing a distributable binary.

- `--crate-type=lib`, `#[crate_type = "lib"]` - A Rust library will be produced. This is an ambiguous concept as to what exactly is produced because a library can manifest itself in several forms. The purpose of this generic `lib` option is to generate the "compiler recommended" style of library. The output library will always be usable by rustc, but the actual type of library may change from time-to-time. The remaining output types are all different flavors of libraries, and the `lib` type can be seen as an alias for one of them (but the actual one is compiler-defined).

- `--crate-type=dylib`, `#[crate_type = "dylib"]` - A dynamic Rust library will be produced. This is different from the `lib` output type in that this forces dynamic library generation. The resulting dynamic library can be used as a dependency for other libraries and/or executables. This output type will create `*.so` files on linux, `*.dylib` files on osx, and `*.dll` files on windows.

- `--crate-type=staticlib`, `#[crate_type = "staticlib"]` - A static system library will be produced. This is different from other library outputs in that the Rust compiler will never attempt to link to `staticlib` outputs. The purpose of this output type is to create a static library containing all of the local crate's code along with all upstream dependencies. The static library is actually a `*.a` archive on linux and osx and a `*.lib` file on windows. This format is recommended for use in situations such as linking Rust code into an existing non-Rust application because it will not have dynamic dependencies on other Rust code.

- `--crate-type=rlib`, `#[crate_type = "rlib"]` - A "Rust library" file will be produced. This is used as an intermediate artifact and can be thought of as a "static Rust library". These `rlib` files, unlike `staticlib` files, are interpreted by the Rust compiler in future linkage. This essentially means that `rustc` will look for metadata in `rlib` files like it looks for metadata in dynamic libraries. This form of output is used to produce statically linked executables as well as `staticlib` outputs.

[33] `book/ffi.html`

Note that these outputs are stackable in the sense that if multiple are specified, then the compiler will produce each form of output at once without having to recompile. However, this only applies for outputs specified by the same method. If only `crate_type` attributes are specified, then they will all be built, but if one or more `--crate-type` command line flags are specified, then only those outputs will be built.

With all these different kinds of outputs, if crate A depends on crate B, then the compiler could find B in various different forms throughout the system. The only forms looked for by the compiler, however, are the `rlib` format and the dynamic library format. With these two options for a dependent library, the compiler must at some point make a choice between these two formats. With this in mind, the compiler follows these rules when determining what format of dependencies will be used:

1. If a static library is being produced, all upstream dependencies are required to be available in `rlib` formats. This requirement stems from the reason that a dynamic library cannot be converted into a static format.

Note that it is impossible to link in native dynamic dependencies to a static library, and in this case warnings will be printed about all unlinked native dynamic dependencies.

2. If an `rlib` file is being produced, then there are no restrictions on what format the upstream dependencies are available in. It is simply required that all upstream dependencies be available for reading metadata from.

The reason for this is that `rlib` files do not contain any of their upstream dependencies. It wouldn't be very efficient for all `rlib` files to contain a copy of `libstd.rlib`!

3. If an executable is being produced and the `-C prefer-dynamic` flag is not specified, then dependencies are first attempted to be found in the `rlib` format. If some dependencies are not available in an rlib format, then dynamic linking is attempted (see below).

4. If a dynamic library or an executable that is being dynamically linked is being produced, then the compiler will attempt to reconcile the available dependencies in either the rlib or dylib format to create a final product.

A major goal of the compiler is to ensure that a library never appears more than once in any artifact. For example, if dynamic libraries B and C were each statically linked to library A, then a crate could not link to B and C together because there would be two copies of A. The compiler allows mixing the rlib and dylib formats, but this restriction must be satisfied.

The compiler currently implements no method of hinting what format a library should be linked with. When dynamically linking, the compiler will attempt to maximize dynamic dependencies while still allowing some dependencies to be linked in via an rlib.

For most situations, having all libraries available as a dylib is recommended if dynamically linking. For other situations, the compiler will emit a warning if it is unable to determine which formats to link each library with.

In general, `--crate-type=bin` or `--crate-type=lib` should be sufficient for all compilation needs, and the other options are just available if more fine-grained control is desired over the output format of a Rust crate.

20.12 Unsafety

Unsafe operations are those that potentially violate the memory-safety guarantees of Rust's static semantics.

The following language level features cannot be used in the safe subset of Rust:

- Dereferencing a raw pointer.
- Reading or writing a mutable static variable.
- Calling an unsafe function (including an intrinsic or foreign function).

20.12.1 Unsafe functions

Unsafe functions are functions that are not safe in all contexts and/or for all possible inputs. Such a function must be prefixed with the keyword `unsafe` and can only be called from an `unsafe` block or another `unsafe` function.

20.12.2 Unsafe blocks

A block of code can be prefixed with the `unsafe` keyword, to permit calling `unsafe` functions or dereferencing raw pointers within a safe function.

When a programmer has sufficient conviction that a sequence of potentially unsafe operations is actually safe, they can encapsulate that sequence (taken as a whole) within an `unsafe` block. The compiler will consider uses of such code safe, in the surrounding context.

Unsafe blocks are used to wrap foreign libraries, make direct use of hardware or implement features not directly present in the language. For example, Rust provides the language features necessary to implement memory-safe concurrency in the language but the implementation of threads and message passing is in the standard library.

Rust's type system is a conservative approximation of the dynamic safety requirements, so in some cases there is a performance cost to using safe code. For example, a doubly-linked list is not a tree structure and can only be represented with reference-counted pointers in safe code. By using `unsafe` blocks to represent the reverse links as raw pointers, it can be implemented with only boxes.

20.12.3 Behavior considered undefined

The following is a list of behavior which is forbidden in all Rust code, including within `unsafe` blocks and `unsafe` functions. Type checking provides the guarantee that these issues are never caused by safe code.

- Data races
- Dereferencing a null/dangling raw pointer
- Reads of undef[34] (uninitialized) memory

[34] `http://llvm.org/docs/LangRef.html#undefined-values`

- Breaking the pointer aliasing rules[35] with raw pointers (a subset of the rules used by C)
- `&mut` and `&` follow LLVM's scoped noalias[36] model, except if the `&T` contains an `UnsafeCell<U>`. Unsafe code must not violate these aliasing guarantees.
- Mutating non-mutable data (that is, data reached through a shared reference or data owned by a `let` binding), unless that data is contained within an `UnsafeCell<U>`.
- Invoking undefined behavior via compiler intrinsics:
- Indexing outside of the bounds of an object with `std::ptr::offset` (`offset` intrinsic), with the exception of one byte past the end which is permitted.
- Using `std::ptr::copy_nonoverlapping_memory` (`memcpy32`/`memcpy64` intrinsics) on overlapping buffers
- Invalid values in primitive types, even in private fields/locals:
- Dangling/null references or boxes
- A value other than `false` (0) or `true` (1) in a `bool`
- A discriminant in an `enum` not included in the type definition
- A value in a `char` which is a surrogate or above `char::MAX`
- Non-UTF-8 byte sequences in a `str`
- Unwinding into Rust from foreign code or unwinding from Rust into foreign code. Rust's failure system is not compatible with exception handling in other languages. Unwinding must be caught and handled at FFI boundaries.

20.12.4 Behavior not considered unsafe

This is a list of behavior not considered *unsafe* in Rust terms, but that may be undesired.

- Deadlocks
- Leaks of memory and other resources
- Exiting without calling destructors
- Integer overflow
- Overflow is considered "unexpected" behavior and is always user-error, unless the `wrapping` primitives are used. In non-optimized builds, the compiler will insert debug checks that panic on overflow, but in optimized builds overflow instead results in wrapped values. See RFC 560[37] for the rationale and more details.

20.13 Appendix: Influences

Rust is not a particularly original language, with design elements coming from a wide range of sources. Some of these are listed below (including elements that have since been removed):

- SML, OCaml: algebraic data types, pattern matching, type inference, semicolon statement separation

[35] `http://llvm.org/docs/LangRef.html#pointer-aliasing-rules`
[36] `http://llvm.org/docs/LangRef.html#noalias`
[37] `https://github.com/rust-lang/rfcs/blob/master/text/0560-integer-overflow.md`

- C++: references, RAII, smart pointers, move semantics, monomorphization, memory model
- ML Kit, Cyclone: region based memory management
- Haskell (GHC): typeclasses, type families
- Newsqueak, Alef, Limbo: channels, concurrency
- Erlang: message passing, thread failure, ~~linked thread failure~~, ~~lightweight concurrency~~
- Swift: optional bindings
- Scheme: hygienic macros
- C#: attributes
- Ruby: ~~block syntax~~
- NIL, Hermes: ~~typestate~~
- Unicode Annex #31[38]: identifier and pattern syntax

[38] http://www.unicode.org/reports/tr31/

Part IV

Standard Library Reference

Part V

Tools Reference