Welcome to Comprehensive Rust 🦀

This is a four day Rust course developed by the Android team. The course covers the full spectrum of Rust, from basic syntax to advanced topics like generics and error handling. It also includes Android-specific content on the last day.

The goal of the course is to teach you Rust. We assume you don't know anything about Rust and hope to:

- Give you a comprehensive understanding of the Rust syntax and language.
- Enable you to modify existing programs and write new programs in Rust.
- Show you common Rust idioms.

On Day 4, we will cover Android-specific things such as:

- Building Android components in Rust.
- AIDL servers and clients.
- Interoperability with C, C++, and Java.

It is important to note that this course does not cover Android **application** development in Rust, and that the Android-specific parts are specifically about writing code for Android itself, the operating system.

Non-Goals

Rust is a large language and we won't be able to cover all of it in a few days. Some non-goals of this course are:

- Learn how to use async Rust — we'll only mention async Rust when covering traditional concurrency primitives. Please see Asynchronous Programming in Rust instead for details on this topic.
- Learn how to develop macros, please see Chapter 19.5 in the Rust Book and Rust by Example instead.

Assumptions

The course assumes that you already know how to program. Rust is a statically typed language and we will sometimes make comparisons with C and C++ to better explain or contrast the Rust approach.

If you know how to program in a dynamically typed language such as Python or JavaScript, then you will be able to follow along just fine too.

▼ *Speaker Notes*

This is an example of a *speaker note*. We will use these to add additional information to the slides. This could be key points which the instructor should cover as well as answers to typical questions which come up in class.

Running the Course

> This page is for the course instructor.

Here is a bit of background information about how we've been running the course internally at Google.

To run the course, you need to:

1. Make yourself familiar with the course material. We've included speaker notes on some of the pages to help highlight the key points (please help us by contributing more speaker notes!). You should make sure to open the speaker notes in a popup (click the link with a little arrow next to "Speaker Notes"). This way you have a clean screen to present to the class.

2. Decide on the dates. Since the course is large, we recommend that you schedule the four days over two weeks. Course participants have said that they find it helpful to have a gap in the course since it helps them process all the information we give them.

3. Find a room large enough for your in-person participants. We recommend a class size of 15-20 people. That's small enough that people are comfortable asking questions — it's also small enough that one instructor will have time to answer the questions.

4. On the day of your course, show up to the room a little early to set things up. We recommend presenting directly using `mdbook serve` running on your laptop (see the installation instructions). This ensures optimal performance with no lag as you change pages. Using your laptop will also allow you to fix typos as you or the course participants spot them.

5. Let people solve the exercises by themselves or in small groups. Make sure to ask people if they're stuck or if there is anything you can help with. When you see that several people have the same problem, call it out to the class and offer a solution, e.g., by showing people where to find the relevant information in the standard library.

6. If you don't skip the Android specific parts on Day 4, you will need an AOSP checkout. Make a checkout of the course repository on the same machine and move the `src/android/` directory into the root of your AOSP checkout. This will ensure that the Android build system sees the `Android.bp` files in `src/android/`.

   Ensure that `adb sync` works with your emulator or real device and pre-build all Android examples using `src/android/build_all.sh`. Read the script to see the commands it runs and make sure they work when you run them by hand.

That is all, good luck running the course! We hope it will be as much fun for you as it has been for us!

Please provide feedback afterwards so that we can keep improving the course. We would love to hear what worked well for you and what can be made better. Your students are also very welcome to send us feedback!

Course Structure

> This page is for the course instructor.

The course is fast paced and covers a lot of ground:

- Day 1: Basic Rust, ownership and the borrow checker.
- Day 2: Compound data types, pattern matching, the standard library.
- Day 3: Traits and generics, error handling, testing, unsafe Rust.
- Day 4: Concurrency in Rust and interoperability with other languages

> **Exercise for Day 4:** Do you interface with some C/C++ code in your project which we could attempt to move to Rust? The fewer dependencies the better. Parsing code would be ideal.

Format

The course is meant to be very interactive and we recommend letting the questions drive the exploration of Rust!

Keyboard Shortcuts

There are several useful keyboard shortcuts in mdBook:

- `Arrow-Left`: Navigate to the previous page.
- `Arrow-Right`: Navigate to the next page.
- `Ctrl + Enter`: Execute the code sample that has focus.
- `s`: Activate the search bar.

Using Cargo

When you start reading about Rust, you will soon meet Cargo, the standard tool used in the Rust ecosystem to build and run Rust applications. Here we want to give a brief overview of what Cargo is and how it fits into the wider ecosystem and how it fits into this training.

Installation

Rustup (Recommended)

You can follow the instructions to install cargo and rust compiler, among other standard ecosystem tools with the rustup tool, which is maintained by the Rust Foundation.

Along with cargo and rustc, Rustup will install itself as a command line utility that you can use to install/switch toolchains, setup cross compilation, etc.

Package Managers

Debian

On Debian/Ubuntu, you can install Cargo, the Rust source and the Rust formatter with

```
$ sudo apt install cargo rust-src rustfmt
```

This will allow rust-analyzer to jump to the definitions. We suggest using VS Code to edit the code (but any LSP compatible editor works).

Some folks also like to use the JetBrains family of IDEs, which do their own analysis but have their own tradeoffs. If you prefer them, you can install the Rust Plugin. Please take note that as of January 2023 debugging only works on the CLion version of the JetBrains IDEA suite.

The Rust Ecosystem

The Rust ecosystem consists of a number of tools, of which the main ones are:

- `rustc`: the Rust compiler which turns `.rs` files into binaries and other intermediate formats.
- `cargo`: the Rust dependency manager and build tool. Cargo knows how to download dependencies hosted on https://crates.io and it will pass them to `rustc` when building your project. Cargo also comes with a built-in test runner which is used to execute unit tests.
- `rustup`: the Rust toolchain installer and updater. This tool is used to install and update `rustc` and `cargo` when new versions of Rust is released. In addition, `rustup` can also download documentation for the standard library. You can have multiple versions of Rust installed at once and `rustup` will let you switch between them as needed.

▼ Details

Key points:

- Rust has a rapid release schedule with a new release coming out every six weeks. New releases maintain backwards compatibility with old releases — plus they enable new functionality.
- There are three release channels: "stable", "beta", and "nightly".
- New features are being tested on "nightly", "beta" is what becomes "stable" every six weeks.
- Rust also has editions: the current edition is Rust 2021. Previous editions were Rust 2015 and Rust 2018.
  - The editions are allowed to make backwards incompatible changes to the language.
  - To prevent breaking code, editions are opt-in: you select the edition for your crate via the `Cargo.toml` file.
  - To avoid splitting the ecosystem, Rust compilers can mix code written for different editions.
  - Mention that it is quite rare to ever use the compiler directly not through `cargo` (most users never do).
  - It might be worth alluding that Cargo itself is an extremely powerful and comprehensive tool. It is capable of many advanced features including but not limited to:
    - Project/package structure
    - workspaces
    - Dev Dependencies and Runtime Dependency management/caching
    - build scripting
    - global installation
    - It is also extensible with sub command plugins as well (such as cargo clippy).
  - Read more from the official Cargo Book

Code Samples in This Training

For this training, we will mostly explore the Rust language through examples which can be executed through your browser. This makes the setup much easier and ensures a consistent experience for everyone.

Installing Cargo is still encouraged: it will make it easier for you to do the exercises. On the last day, we will do a larger exercise which shows you how to work with dependencies and for that you need Cargo.

The code blocks in this course are fully interactive:

```
fn main() {
    println!("Edit me!");
}
```

You can use `Ctrl + Enter` to execute the code when focus is in the text box.

▼ Details

Most code samples are editable like shown above. A few code samples are not editable for various reasons:

- The embedded playgrounds cannot execute unit tests. Copy-paste the code and open it in the real Playground to demonstrate unit tests.
- The embedded playgrounds lose their state the moment you navigate away from the page! This is the reason that the students should solve the exercises using a local Rust installation or via the Playground.

Running Code Locally with Cargo

If you want to experiment with the code on your own system, then you will need to first install Rust. Do this by following the instructions in the Rust Book. This should give you a working `rustc` and `cargo`. At the time of writing, the latest stable Rust release has these version numbers:

```
% rustc --version
rustc 1.61.0 (fe5b13d68 2022-05-18)
% cargo --version
cargo 1.61.0 (a028ae4 2022-04-29)
```

With this is in place, then follow these steps to build a Rust binary from one of the examples in this training:

1. Click the “Copy to clipboard” button on the example you want to copy.
2. Use `cargo new exercise` to create a new `exercise/` directory for your code:

   ```
   $ cargo new exercise
        Created binary (application) `exercise` package
   ```

3. Navigate into `exercise/` and use `cargo run` to build and run your binary:

   ```
   $ cd exercise
   $ cargo run
      Compiling exercise v0.1.0 (/home/mgeisler/tmp/exercise)
       Finished dev [unoptimized + debuginfo] target(s) in 0.75s
        Running `target/debug/exercise`
   Hello, world!
   ```

4. Replace the boiler-plate code in `src/main.rs` with your own code. For example, using the example on the previous page, make `src/main.rs` look like

   ```
   fn main() {
       println!("Edit me!");
   }
   ```

5. Use `cargo run` to build and run your updated binary:

   ```
   $ cargo run
      Compiling exercise v0.1.0 (/home/mgeisler/tmp/exercise)
       Finished dev [unoptimized + debuginfo] target(s) in 0.24s
        Running `target/debug/exercise`
   Edit me!
   ```

6. Use `cargo check` to quickly check your project for errors, use `cargo build` to compile it without running it. You will find the output in `target/debug/` for a normal debug build. Use `cargo build --release` to produce an optimized release build in `target/release/`.
7. You can add dependencies for your project by editing `Cargo.toml`. When you run `cargo` commands, it will automatically download and compile missing dependencies for you.

▼ Details

Try to encourage the class participants to install Cargo and use a local editor. It will make their life easier since they will have a normal development environment.

Welcome to Day 1

This is the first day of Comprehensive Rust. We will cover a lot of ground today:

- Basic Rust syntax: variables, scalar and compound types, enums, structs, references, functions, and methods.
- Memory management: stack vs heap, manual memory management, scope-based memory management, and garbage collection.
- Ownership: move semantics, copying and cloning, borrowing, and lifetimes.

▼ Details

Please remind the students that:

- They should ask questions when they get them, don't save them to the end.
- The class is meant to be interactive and discussions are very much encouraged!
  - As an instructor, you should try to keep the discussions relevant, i.e., keep the related to how Rust does things vs some other language. It can be hard to find the right balance, but err on the side of allowing discussions since they engage people much more than one-way communication.
- The questions will likely mean that we talk about things ahead of the slides.
  - This is perfectly okay! Repetition is an important part of learning. Remember that the slides are just a support and you are free to skip them as you like.

The idea for the first day is to show *just enough* of Rust to be able to speak about the famous borrow checker. The way Rust handles memory is a major feature and we should show students this right away.

If you're teaching this in a classroom, this is a good place to go over the schedule. We suggest splitting the day into two parts (following the slides):

- Morning: 9:00 to 12:00,
- Afternoon: 13:00 to 16:00.

You can of course adjust this as necessary. Please make sure to include breaks, we recommend a break every hour!

What is Rust?

Rust is a new programming language which had its 1.0 release in 2015:

- Rust is a statically compiled language in a similar role as C++
  - `rustc` uses LLVM as its backend.
- Rust supports many platforms and architectures:
  - x86, ARM, WebAssembly, ...
  - Linux, Mac, Windows, ...
- Rust is used for a wide range of devices:
  - firmware and boot loaders,
  - smart displays,
  - mobile phones,
  - desktops,
  - servers.

▼ Details

Rust fits in the same area as C++:

- High flexibility.
- High level of control.
- Can be scaled down to very constrained devices like mobile phones.
- Has no runtime or garbage collection.
- Focuses on reliability and safety without sacrificing performance.

Hello World!

Let us jump into the simplest possible Rust program, a classic Hello World program:

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

What you see:

- Functions are introduced with `fn`.
- Blocks are delimited by curly braces like in C and C++.
- The `main` function is the entry point of the program.
- Rust has hygienic macros, `println!` is an example of this.
- Rust strings are UTF-8 encoded and can contain any Unicode character.

▼ Details

This slide tries to make the students comfortable with Rust code. They will see a ton of it over the next four days so we start small with something familiar.

Key points:

- Rust is very much like other languages in the C/C++/Java tradition. It is imperative (not functional) and it doesn't try to reinvent things unless absolutely necessary.
- Rust is modern with full support for things like Unicode.
- Rust uses macros for situations where you want to have a variable number of arguments (no function overloading).

Small Example

Here is a small example program in Rust:

```
fn main() {              // Program entry point
    let mut x: i32 = 6;  // Mutable variable binding
    print!("{x}");       // Macro for printing, like printf
    while x != 1 {       // No parenthesis around expression
        if x % 2 == 0 {  // Math like in other languages
            x = x / 2;
        } else {
            x = 3 * x + 1;
        }
        print!(" -> {x}");
    }
    println!();
}
```

▼ Details

The code implements the Collatz conjecture: it is believed that the loop will always end, but this is not yet proved. Edit the code and play with different inputs.

Key points:

- Explain that all variables are statically typed. Try removing `i32` to trigger type inference. Try with `i8` instead and trigger a runtime integer overflow.
- Change `let mut x` to `let x`, discuss the compiler error.
- Show how `print!` gives a compilation error if the arguments don't match the format string.
- Show how you need to use `{}` as a placeholder if you want to print an expression which is more complex than just a single variable.
- Show the students the standard library, show them how to search for `std::fmt` which has the rules of the formatting mini-language. It's important that the students become familiar with searching in the standard library.

Why Rust?

Some unique selling points of Rust:

- Compile time memory safety.
- Lack of undefined runtime behavior.
- Modern language features.

▼ Details

Make sure to ask the class which languages they have experience with. Depending on the answer you can highlight different features of Rust:

- Experience with C or C++: Rust eliminates a whole class of *runtime errors* via the borrow checker. You get performance like in C and C++, but you don't have the memory unsafety issues. In addition, you get a modern language with constructs like pattern matching and built-in dependency management.

- Experience with Java, Go, Python, JavaScript...: You get the same memory safety as in those languages, plus a similar high-level language feeling. In addition you get fast and predictable performance like C and C++ (no garbage collector) as well as access to low-level hardware (should you need it)

Compile Time Guarantees

Static memory management at compile time:

- No uninitialized variables.
- No memory leaks (*mostly*, see notes).
- No double-frees.
- No use-after-free.
- No `NULL` pointers.
- No forgotten locked mutexes.
- No data races between threads.
- No iterator invalidation.

▼ Details

It is possible to produce memory leaks in (safe) Rust. Some examples are:

- You can for use `Box::leak` to leak a pointer. A use of this could be to get runtime-initialized and runtime-sized static variables
- You can use `std::mem::forget` to make the compiler "forget" about a value (meaning the destructor is never run).
- You can also accidentally create a reference cycle with `Rc` or `Arc`.
- In fact, some will consider infinitely populating a collection a memory leak and Rust does not protect from those.

For the purpose of this course, "No memory leaks" should be understood as "Pretty much no *accidental* memory leaks".

Runtime Guarantees

No undefined behavior at runtime:

- Array access is bounds checked.
- Integer overflow is defined.

▼ Details

Key points:

- Integer overflow is defined via a compile-time flag. The options are either a panic (a controlled crash of the program) or wrap-around semantics. By default, you get panics in debug mode (`cargo build`) and wrap-around in release mode (`cargo build --release`).

- Bounds checking cannot be disabled with a compiler flag. It can also not be disabled directly with the `unsafe` keyword. However, `unsafe` allows you to call functions such as `slice::get_unchecked` which does not do bounds checking.

Modern Features

Rust is built with all the experience gained in the last 40 years.

Language Features

- Enums and pattern matching.
- Generics.
- No overhead FFI.
- Zero-cost abstractions.

Tooling

- Great compiler errors.
- Built-in dependency manager.
- Built-in support for testing.
- Excellent Language Server Protocol support.

▼ Details

Key points:

- Zero-cost abstractions, similar to C++, means that you don't have to 'pay' for higher-level programming constructs with memory or CPU. For example, writing a loop using `for` should result in roughly the same low level instructions as using the `.iter().fold()` construct.

- It may be worth mentioning that Rust enums are 'Algebraic Data Types', also known as 'sum types', which allow the type system to express things like `Option<T>` and `Result<T, E>`.

- Remind people to read the errors — many developers have gotten used to ignore lengthy compiler output. The Rust compiler is significantly more talkative than other compilers. It will often provide you with *actionable* feedback, ready to copy-paste into your code.

- The Rust standard library is small compared to languages like Java, Python, and Go. Rust does not come with several things you might consider standard and essential:

  - a random number generator, but see rand.
  - support for SSL or TLS, but see rusttls.
  - support for JSON, but see serde_json. The reasoning behind this is that functionality in the standard library cannot go away, so it has to be very stable. For the examples above, the Rust community is still working on finding the best solution — and perhaps there isn't a single "best solution" for some of these things.

  Rust comes with a built-in package manager in the form of Cargo and this makes it trivial to download and compile third-party crates. A consequence of this is that the standard library can be smaller.

  Discovering good third-party crates can be a problem. Sites like https://lib.rs/ help with this by letting you compare health metrics for crates to find a good and trusted one.

- rust-analyzer is a well supported LSP implementation used in major IDEs and text editors.

Basic Syntax

Much of the Rust syntax will be familiar to you from C, C++ or Java:

- Blocks and scopes are delimited by curly braces.
- Line comments are started with `//`, block comments are delimited by `/* ... */`.
- Keywords like `if` and `while` work the same.
- Variable assignment is done with `=`, comparison is done with `==`.

Scalar Types

	Types	Literals
Signed integers	`i8`, `i16`, `i32`, `i64`, `i128`, `isize`	`-10`, `0`, `1_000`, `123i64`
Unsigned integers	`u8`, `u16`, `u32`, `u64`, `u128`, `usize`	`0`, `123`, `10u16`
Floating point numbers	`f32`, `f64`	`3.14`, `-10.0e20`, `2f32`
Strings	`&str`	`"foo"`, `r#"\\"#`
Unicode scalar values	`char`	`'a'`, `'α'`, `'∞'`
Byte strings	`&[u8]`	`b"abc"`, `br#" " "#`
Booleans	`bool`	`true`, `false`

The types have widths as follows:

- `iN`, `uN`, and `fN` are *N* bits wide,
- `isize` and `usize` are the width of a pointer,
- `char` is 32 bit wide,
- `bool` is 8 bit wide.

Compound Types

	Types	Literals
Arrays	`[T; N]`	`[20, 30, 40]`, `[0; 3]`
Tuples	`()`, `(T,)`, `(T1, T2)`, ...	`()`, `('x',)`, `('x', 1.2)`, ...

Array assignment and access:

```
fn main() {
    let mut a: [i8; 10] = [42; 10];
    a[5] = 0;
    println!("a: {:?}", a);
}
```

Tuple assignment and access:

```
fn main() {
    let t: (i8, bool) = (7, true);
    println!("1st index: {}", t.0);
    println!("2nd index: {}", t.1);
}
```

▼ Details

Key points:

Arrays:

- Arrays have elements of the same type, `T`, and length, `N`, which is a compile-time constant. Note that the length of the array is *part of its type*, which means that `[u8; 3]` and `[u8; 4]` are considered two different types.
- We can use literals to assign values to arrays.
- In the main function, the print statement asks for the debug implementation with the `?` format parameter: `{}` gives the default output, `{:?}` gives the debug output. We could also have used `{a}` and `{a:?}` without specifying the value after the format string.
- Adding `#`, eg `{a:#?}`, invokes a "pretty printing" format, which can be easier to read.

Tuples:

- Like arrays, tuples have a fixed length.
- Tuples group together values of different types into a compound type.
- Fields of a tuple can be accessed by the period and the index of the value, e.g. `t.0`, `t.1`.
- The empty tuple `()` is also known as the "unit type". It is both a type, and the only valid value of that type - that is to say both the type and its value are expressed as `()`. It is used to indicate, for example, that a function or expression has no return value, as we'll see in a future slide.
    - You can think of it as `void` that can be familiar to you from other programming languages.

References

Like C++, Rust has references:

```rust
fn main() {
    let mut x: i32 = 10;
    let ref_x: &mut i32 = &mut x;
    *ref_x = 20;
    println!("x: {x}");
}
```

Some notes:

- We must dereference `ref_x` when assigning to it, similar to C and C++ pointers.
- Rust will auto-dereference in some cases, in particular when invoking methods (try `ref_x.count_ones()`).
- References that are declared as `mut` can be bound to different values over their lifetime.

▼ Details

Key points:

- Be sure to note the difference between `let mut ref_x: &i32` and `let ref_x: &mut i32`. The first one represents a mutable reference which can be bound to different values, while the second represents a reference to a mutable value.

Dangling References

Rust will statically forbid dangling references:

```
fn main() {
    let ref_x: &i32;
    {
        let x: i32 = 10;
        ref_x = &x;
    }
    println!("ref_x: {ref_x}");
}
```

- A reference is said to “borrow” the value it refers to.
- Rust is tracking the lifetimes of all references to ensure they live long enough.
- We will talk more about borrowing when we get to ownership.

Slices

A slice gives you a view into a larger collection:

```
fn main() {
    let a: [i32; 6] = [10, 20, 30, 40, 50, 60];
    println!("a: {a:?}");

    let s: &[i32] = &a[2..4];
    println!("s: {s:?}");
}
```

- Slices borrow data from the sliced type.
- Question: What happens if you modify `a[3]` ?

▼ Details

- We create a slice by borrowing `a` and specifying the starting and ending indexes in brackets.
- If the slice starts at index 0, Rust's range syntax allows us to drop the starting index, meaning that `&a[0..a.len()]` and `&a[..a.len()]` are identical.
- The same is true for the last index, so `&a[2..a.len()]` and `&a[2..]` are identical.
- To easily create a slice of the full array, we can therefore use `&a[..]` .
- `s` is a reference to a slice of `i32` s. Notice that the type of `s` (`&[i32]`) no longer mentions the array length. This allows us to perform computation on slices of different sizes.
- Slices always borrow from another object. In this example, `a` has to remain 'alive' (in scope) for at least as long as our slice.
- The question about modifying `a[3]` can spark an interesting discussion, but the answer is that for memory safety reasons you cannot do it through `a` after you created a slice, but you can read the data from both `a` and `s` safely. More details will be explained in the borrow checker section.

`String` vs `str`

We can now understand the two string types in Rust:

```
fn main() {
    let s1: &str = "World";
    println!("s1: {s1}");

    let mut s2: String = String::from("Hello ");
    println!("s2: {s2}");
    s2.push_str(s1);
    println!("s2: {s2}");

    let s3: &str = &s2[6..];
    println!("s3: {s3}");
}
```

Rust terminology:

- `&str` an immutable reference to a string slice.
- `String` a mutable string buffer.

▼ Details

- `&str` introduces a string slice, which is an immutable reference to UTF-8 encoded string data stored in a block of memory. String literals (`"Hello"`), are stored in the program's binary.
- Rust's `String` type is a wrapper around a vector of bytes. As with a `Vec<T>` , it is owned.
- As with many other types `String::from()` creates a string from a string literal; `String::new()` creates a new empty string, to which string data can be added using the `push()` and `push_str()` methods.
- The `format!()` macro is a convenient way to generate an owned string from dynamic values. It accepts the same format specification as `println!()` .
- You can borrow `&str` slices from `String` via `&` and optionally range selection.
- For C++ programmers: think of `&str` as `const char*` from C++, but the one that always points to a valid string in memory. Rust `String` is a rough equivalent of `std::string` from C++ (main difference: it can only contain UTF-8 encoded bytes and will never use a small-string optimization).

Functions

A Rust version of the famous FizzBuzz interview question:

```rust
fn main() {
    fizzbuzz_to(20);   // Defined below, no forward declaration needed
}

fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    if rhs == 0 {
        return false;  // Corner case, early return
    }
    lhs % rhs == 0     // The last expression in a block is the return value
}

fn fizzbuzz(n: u32) -> () {  // No return value means returning the unit type `()`
    match (is_divisible_by(n, 3), is_divisible_by(n, 5)) {
        (true,  true)  => println!("fizzbuzz"),
        (true,  false) => println!("fizz"),
        (false, true)  => println!("buzz"),
        (false, false) => println!("{n}"),
    }
}

fn fizzbuzz_to(n: u32) {  // `-> ()` is normally omitted
    for i in 1..=n {
        fizzbuzz(i);
    }
}
```

▼ Details

- We refer in `main` to a function written below. Neither forward declarations nor headers are necessary.
- Declaration parameters are followed by a type (the reverse of some programming languages), then a return type.
- The last expression in a function body (or any block) becomes the return value. Simply omit the `;` at the end of the expression.
- Some functions have no return value, and return the 'unit type', `()`. The compiler will infer this if the `-> ()` return type is omitted.
- The range expression in the `for` loop in `fizzbuzz_to()` contains `=n`, which causes it to include the upper bound.
- The `match` expression in `fizzbuzz()` is doing a lot of work. It is expanded below to show what is happening.

  (Type annotations added for clarity, but they can be elided.)

  ```rust
  let by_3: bool = is_divisible_by(n, 3);
  let by_5: bool = is_divisible_by(n, 5);
  let by_35: (bool, bool) = (by_3, by_5);
  match by_35 {
    // ...
  ```

Methods

Rust has methods, they are simply functions that are associated with a particular type. The first argument of a method is an instance of the type it is associated with:

```rust
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn inc_width(&mut self, delta: u32) {
        self.width += delta;
    }
}

fn main() {
    let mut rect = Rectangle { width: 10, height: 5 };
    println!("old area: {}", rect.area());
    rect.inc_width(5);
    println!("new area: {}", rect.area());
}
```

- We will look much more at methods in today's exercise and in tomorrow's class.

Function Overloading

Overloading is not supported:

- Each function has a single implementation:
  - Always takes a fixed number of parameters.
  - Always takes a single set of parameter types.
- Default values are not supported:
  - All call sites have the same number of arguments.
  - Macros are sometimes used as an alternative.

However, function parameters can be generic:

```
fn pick_one<T>(a: T, b: T) -> T {
    if std::process::id() % 2 == 0 { a } else { b }
}

fn main() {
    println!("coin toss: {}", pick_one("heads", "tails"));
    println!("cash prize: {}", pick_one(500, 1000));
}
```

▼ Details

- When using generics, the standard library's `Into<T>` can provide a kind of limited polymorphism on argument types. We will see more details in a later section.

Day 1: Morning Exercises

In these exercises, we will explore two parts of Rust:

- Implicit conversions between types.
- Arrays and `for` loops.

▼ Details

A few things to consider while solving the exercises:

- Use a local Rust installation, if possible. This way you can get auto-completion in your editor. See the page about Using Cargo for details on installing Rust.
- Alternatively, use the Rust Playground.

The code snippets are not editable on purpose: the inline code snippets lose their state if you navigate away from the page.

After looking at the exercises, you can look at the solutions provided.

Implicit Conversions

Rust will not automatically apply *implicit conversions* between types (unlike C++). You can see this in a program like this:

```
fn multiply(x: i16, y: i16) -> i16 {
    x * y
}

fn main() {
    let x: i8 = 15;
    let y: i16 = 1000;

    println!("{x} * {y} = {}", multiply(x, y));
}
```

The Rust integer types all implement the `From<T>` and `Into<T>` traits to let us convert between them. The `From<T>` trait has a single `from()` method and similarly, the `Into<T>` trait has a single `into()` method. Implementing these traits is how a type expresses that it can be converted into another type.

The standard library has an implementation of `From<i8> for i16`, which means that we can convert a variable `x` of type `i8` to an `i16` by calling `i16::from(x)`. Or, simpler, with `x.into()`, because `From<i8> for i16` implementation automatically create an implementation of `Into<i16> for i8`.

The same applies for your own `From` implementations for your own types, so it is sufficient to only implement `From` to get a respective `Into` implementation automatically.

1. Execute the above program and look at the compiler error.
2. Update the code above to use `into()` to do the conversion.
3. Change the types of `x` and `y` to other things (such as `f32`, `bool`, `i128`) to see which types you can convert to which other types. Try converting small types to big types and the other way around. Check the standard library documentation to see if `From<T>` is implemented for the pairs you check.

Arrays and for Loops

We saw that an array can be declared like this:

```
let array = [10, 20, 30];
```

You can print such an array by asking for its debug representation with `{:?}`:

```
fn main() {
    let array = [10, 20, 30];
    println!("array: {array:?}");
}
```

Rust lets you iterate over things like arrays and ranges using the `for` keyword:

```
fn main() {
    let array = [10, 20, 30];
    print!("Iterating over array:");
    for n in array {
        print!(" {n}");
    }
    println!();

    print!("Iterating over range:");
    for i in 0..3 {
        print!(" {}", array[i]);
    }
    println!();
}
```

Use the above to write a function `pretty_print` which pretty-print a matrix and a function `transpose` which will transpose a matrix (turn rows into columns):

$$\text{transpose}\left(\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}\right) == \begin{bmatrix}1 & 4 & 7\\2 & 5 & 8\\3 & 6 & 9\end{bmatrix}$$

Hard-code both functions to operate on 3 × 3 matrices.

Copy the code below to https://play.rust-lang.org/ and implement the functions:

```
// TODO: remove this when you're done with your implementation.
#![allow(unused_variables, dead_code)]

fn transpose(matrix: [[i32; 3]; 3]) -> [[i32; 3]; 3] {
    unimplemented!()
}

fn pretty_print(matrix: &[[i32; 3]; 3]) {
    unimplemented!()
}

fn main() {
    let matrix = [
        [101, 102, 103], // <-- the comment makes rustfmt add a newline
        [201, 202, 203],
        [301, 302, 303],
    ];

    println!("matrix:");
    pretty_print(&matrix);

    let transposed = transpose(matrix);
    println!("transposed:");
    pretty_print(&transposed);
}
```

Bonus Question

Could you use `&[i32]` slices instead of hard-coded 3 × 3 matrices for your argument and return types? Something like `&[&[i32]]` for a two-dimensional slice-of-slices. Why or why not?

See the `ndarray` crate for a production quality implementation.

▼ Details

The solution and the answer to the bonus section are available in the Solution section.

Variables

Rust provides type safety via static typing. Variable bindings are immutable by default:

```
fn main() {
    let x: i32 = 10;
    println!("x: {x}");
    // x = 20;
    // println!("x: {x}");
}
```

▼ Details

- Due to type inference the `i32` is optional. We will gradually show the types less and less as the course progresses.
- Note that since `println!` is a macro, `x` is not moved, even using the function like syntax of `println!("x: {}", x)`

Type Inference

Rust will look at how the variable is *used* to determine the type:

```
fn takes_u32(x: u32) {
    println!("u32: {x}");
}

fn takes_i8(y: i8) {
    println!("i8: {y}");
}

fn main() {
    let x = 10;
    let y = 20;

    takes_u32(x);
    takes_i8(y);
    // takes_u32(y);
}
```

▼ Details

This slide demonstrates how the Rust compiler infers types based on constraints given by variable declarations and usages.

It is very important to emphasize that variables declared like this are not of some sort of dynamic “any type” that can hold any data. The machine code generated by such declaration is identical to the explicit declaration of a type. The compiler does the job for us and helps us to write a more concise code.

The following code tells the compiler to copy into a certain generic container without the code ever explicitly specifying the contained type, using `_` as a placeholder:

```
fn main() {
    let mut v = Vec::new();
    v.push((10, false));
    v.push((20, true));
    println!("v: {v:?}");

    let vv = v.iter().collect::<std::collections::HashSet<_>>();
    println!("vv: {vv:?}");
}
```

`collect` relies on `FromIterator`, which `HashSet` implements.

Static and Constant Variables

Global state is managed with static and constant variables.

`const`

You can declare compile-time constants:

```
const DIGEST_SIZE: usize = 3;
const ZERO: Option<u8> = Some(42);

fn compute_digest(text: &str) -> [u8; DIGEST_SIZE] {
    let mut digest = [ZERO.unwrap_or(0); DIGEST_SIZE];
    for (idx, &b) in text.as_bytes().iter().enumerate() {
        digest[idx % DIGEST_SIZE] = digest[idx % DIGEST_SIZE].wrapping_add(b);
    }
    digest
}

fn main() {
    let digest = compute_digest("Hello");
    println!("Digest: {digest:?}");
}
```

According the the Rust RFC Book these are inlined upon use.

`static`

You can also declare static variables:

```
static BANNER: &str = "Welcome to RustOS 3.14";

fn main() {
    println!("{BANNER}");
}
```

As noted in the Rust RFC Book, these are not inlined upon use and have an actual associated memory location. This is useful for unsafe and embedded code, and the variable lives through the entirety of the program execution.

We will look at mutating static data in the chapter on Unsafe Rust.

▼ Details

- Mention that `const` behaves semantically similar to C++'s `constexpr`.
- `static`, on the other hand, is much more similar to a `const` or mutable global variable in C++.
- It isn't super common that one would need a runtime evaluated constant, but it is helpful and safer than using a static.

Scopes and Shadowing

You can shadow variables, both those from outer scopes and variables from the same scope:

```rust
fn main() {
    let a = 10;
    println!("before: {a}");

    {
        let a = "hello";
        println!("inner scope: {a}");

        let a = true;
        println!("shadowed in inner scope: {a}");
    }

    println!("after: {a}");
}
```

▼ Details

- Definition: Shadowing is different from mutation, because after shadowing both variable's memory locations exist at the same time. Both are available under the same name, depending where you use it in the code.
- A shadowing variable can have a different type.
- Shadowing looks obscure at first, but is convenient for holding on to values after `.unwrap()`.
- The following code demonstrates why the compiler can't simply reuse memory locations when shadowing an immutable variable in a scope, even if the type does not change.

```rust
fn main() {
    let a = 1;
    let b = &a;
    let a = a + 1;
    println!("{a} {b}");
}
```

Memory Management

Traditionally, languages have fallen into two broad categories:

- Full control via manual memory management: C, C++, Pascal, …
- Full safety via automatic memory management at runtime: Java, Python, Go, Haskell, …

Rust offers a new mix:

> Full control *and* safety via compile time enforcement of correct memory management.

It does this with an explicit ownership concept.

First, let’s refresh how memory management works.

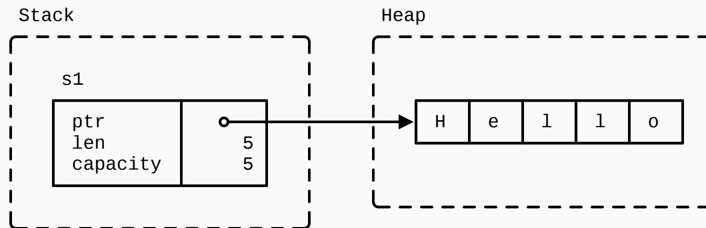
The Stack vs The Heap

- Stack: Continuous area of memory for local variables.
  - Values have fixed sizes known at compile time.
  - Extremely fast: just move a stack pointer.
  - Easy to manage: follows function calls.
  - Great memory locality.
- Heap: Storage of values outside of function calls.
  - Values have dynamic sizes determined at runtime.
  - Slightly slower than the stack: some book-keeping needed.
  - No guarantee of memory locality.

Stack Memory

Creating a `String` puts fixed-sized data on the stack and dynamically sized data on the heap:

```
fn main() {
    let s1 = String::from("Hello");
}
```



▼ Details

- Mention that a `String` is backed by a `Vec`, so it has a capacity and length and can grow if mutable via reallocation on the heap.
- If students ask about it, you can mention that the underlying memory is heap allocated using the System Allocator and custom allocators can be implemented using the Allocator API
- We can inspect the memory layout with `unsafe` code. However, you should point out that this is rightfully unsafe!

```
fn main() {
    let mut s1 = String::from("Hello");
    s1.push(' ');
    s1.push_str("world");
    // DON'T DO THIS AT HOME! For educational purposes only.
    // String provides no guarantees about its layout, so this could lead to
    // undefined behavior.
    unsafe {
        let (capacity, ptr, len): (usize, usize, usize) = std::mem::transmute(s1);
        println!("ptr = {ptr:#x}, len = {len}, capacity = {capacity}");
    }
}
```

Manual Memory Management

You allocate and deallocate heap memory yourself.

If not done with care, this can lead to crashes, bugs, security vulnerabilities, and memory leaks.

C Example

You must call `free` on every pointer you allocate with `malloc`:

```
void foo(size_t n) {
    int* int_array = (int*)malloc(n * sizeof(int));
    //
    // ... lots of code
    //
    free(int_array);
}
```

Memory is leaked if the function returns early between `malloc` and `free`: the pointer is lost and we cannot deallocate the memory.

Scope-Based Memory Management

Constructors and destructors let you hook into the lifetime of an object.

By wrapping a pointer in an object, you can free memory when the object is destroyed. The compiler guarantees that this happens, even if an exception is raised.

This is often called *resource acquisition is initialization* (RAII) and gives you smart pointers.

C++ Example

```
void say_hello(std::unique_ptr<Person> person) {
  std::cout << "Hello " << person->name << std::endl;
}
```

- The `std::unique_ptr` object is allocated on the stack, and points to memory allocated on the heap.
- At the end of `say_hello`, the `std::unique_ptr` destructor will run.
- The destructor frees the `Person` object it points to.

Special move constructors are used when passing ownership to a function:

```
std::unique_ptr<Person> person = find_person("Carla");
say_hello(std::move(person));
```

Automatic Memory Management

An alternative to manual and scope-based memory management is automatic memory management:

- The programmer never allocates or deallocates memory explicitly.
- A garbage collector finds unused memory and deallocates it for the programmer.

Java Example

The `person` object is not deallocated after `sayHello` returns:

```java
void sayHello(Person person) {
  System.out.println("Hello " + person.getName());
}
```

Memory Management in Rust

Memory management in Rust is a mix:

- Safe and correct like Java, but without a garbage collector.
- Depending on which abstraction (or combination of abstractions) you choose, can be a single unique pointer, reference counted, or atomically reference counted.
- Scope-based like C++, but the compiler enforces full adherence.
- A Rust user can choose the right abstraction for the situation, some even have no cost at runtime like C.

It achieves this by modeling *ownership* explicitly.

▼ Details

- If asked how at this point, you can mention that in Rust this is usually handled by RAII wrapper types such as Box, Vec, Rc, or Arc. These encapsulate ownership and memory allocation via various means, and prevent the potential errors in C.

- You may be asked about destructors here, the Drop trait is the Rust equivalent.

Comparison

Here is a rough comparison of the memory management techniques.

Pros of Different Memory Management Techniques

- Manual like C:
  - No runtime overhead.
- Automatic like Java:
  - Fully automatic.
  - Safe and correct.
- Scope-based like C++:
  - Partially automatic.
  - No runtime overhead.
- Compiler-enforced scope-based like Rust:
  - Enforced by compiler.
  - No runtime overhead.
  - Safe and correct.

Cons of Different Memory Management Techniques

- Manual like C:
  - Use-after-free.
  - Double-frees.
  - Memory leaks.
- Automatic like Java:
  - Garbage collection pauses.
  - Destructor delays.
- Scope-based like C++:
  - Complex, opt-in by programmer.
  - Potential for use-after-free.
- Compiler-enforced and scope-based like Rust:
  - Some upfront complexity.
  - Can reject valid programs.

Ownership

All variable bindings have a *scope* where they are valid and it is an error to use a variable outside its scope:

```
struct Point(i32, i32);

fn main() {
    {
        let p = Point(3, 4);
        println!("x: {}", p.0);
    }
    println!("y: {}", p.1);
}
```

- At the end of the scope, the variable is *dropped* and the data is freed.
- A destructor can run here to free up resources.
- We say that the variable *owns* the value.

Move Semantics

An assignment will transfer ownership between variables:

```
fn main() {
    let s1: String = String::from("Hello!");
    let s2: String = s1;
    println!("s2: {s2}");
    // println!("s1: {s1}");
}
```

- The assignment of `s1` to `s2` transfers ownership.
- The data was *moved* from `s1` and `s1` is no longer accessible.
- When `s1` goes out of scope, nothing happens: it has no ownership.
- When `s2` goes out of scope, the string data is freed.
- There is always *exactly* one variable binding which owns a value.

▼ Details

- Mention that this is the opposite of the defaults in C++, which copies by value unless you use `std::move` (and the move constructor is defined!).

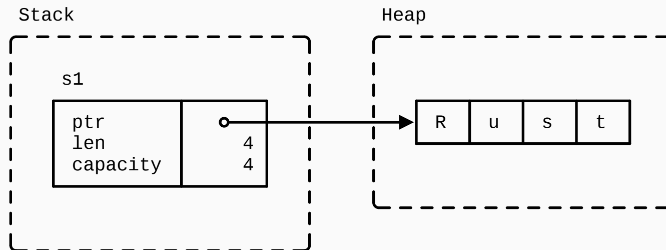
- In Rust, you clones are explicit (by using `clone`).

Moved Strings in Rust

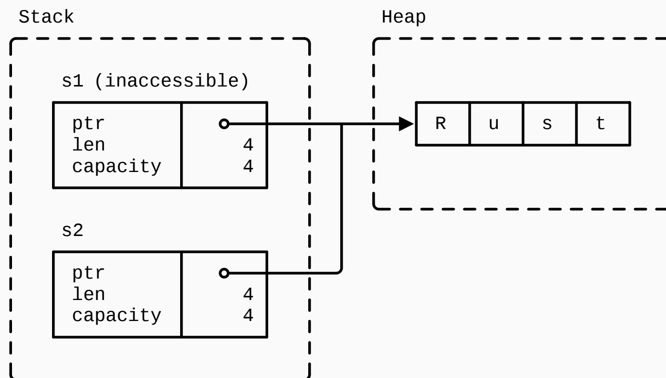
```
fn main() {
    let s1: String = String::from("Rust");
    let s2: String = s1;
}
```

- The heap data from `s1` is reused for `s2`.
- When `s1` goes out of scope, nothing happens (it has been moved from).

Before move to `s2`:



After move to `s2`:



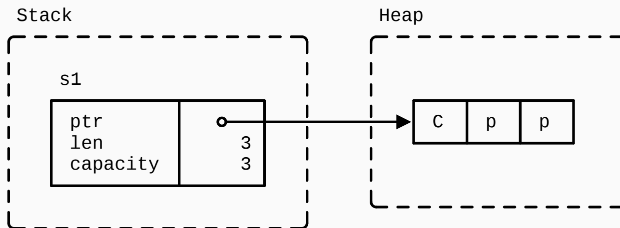
Double Frees in Modern C++

Modern C++ solves this differently:

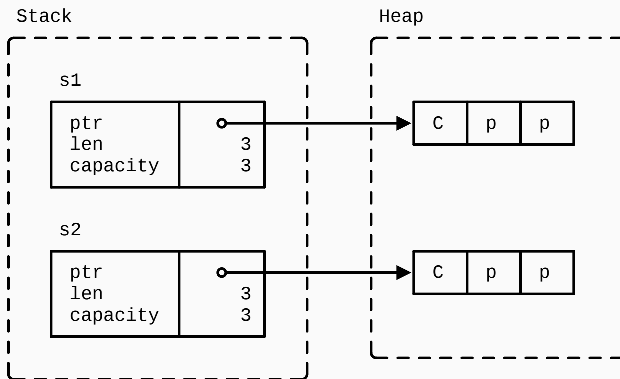
```
std::string s1 = "Cpp";
std::string s2 = s1;  // Duplicate the data in s1.
```

- The heap data from `s1` is duplicated and `s2` gets its own independent copy.
- When `s1` and `s2` go out of scope, they each free their own memory.

Before copy-assignment:



After copy-assignment:



Moves in Function Calls

When you pass a value to a function, the value is assigned to the function parameter. This transfers ownership:

```
fn say_hello(name: String) {
    println!("Hello {name}")
}

fn main() {
    let name = String::from("Alice");
    say_hello(name);
    // say_hello(name);
}
```

▼ Details

- With the first call to `say_hello`, `main` gives up ownership of `name`. Afterwards, `name` cannot be used anymore within `main`.
- The heap memory allocated for `name` will be freed at the end of the `say_hello` function.
- `main` can retain ownership if it passes `name` as a reference (`&name`) and if `say_hello` accepts a reference as a parameter.
- Alternatively, `main` can pass a clone of `name` in the first call (`name.clone()`).
- Rust makes it harder than C++ to inadvertently create copies by making move semantics the default, and by forcing programmers to make clones explicit.

Copying and Cloning

While move semantics are the default, certain types are copied by default:

```
fn main() {
    let x = 42;
    let y = x;
    println!("x: {x}");
    println!("y: {y}");
}
```

These types implement the `Copy` trait.

You can opt-in your own types to use copy semantics:

```
#[derive(Copy, Clone, Debug)]
struct Point(i32, i32);

fn main() {
    let p1 = Point(3, 4);
    let p2 = p1;
    println!("p1: {p1:?}");
    println!("p2: {p2:?}");
}
```

- After the assignment, both `p1` and `p2` own their own data.
- We can also use `p1.clone()` to explicitly copy the data.

▼ Details

Copying and cloning are not the same thing:

- Copying refers to bitwise copies of memory regions and does not work on arbitrary objects.
- Copying does not allow for custom logic (unlike copy constructors in C++).
- Cloning is a more general operation and also allows for custom behavior by implementing the `Clone` trait.
- Copying does not work on types that implement the `Drop` trait.

In the above example, try the following:

- Add a `String` field to `struct Point`. It will not compile because `String` is not a `Copy` type.
- Remove `Copy` from the `derive` attribute. The compiler error is now in the `println!` for `p1`.
- Show that it works if you clone `p1` instead.

If students ask about `derive`, it is sufficient to say that this is a way to generate code in Rust at compile time. In this case the default implementations of `Copy` and `Clone` traits are generated.

Borrowing

Instead of transferring ownership when calling a function, you can let a function *borrow* the value:

```
#[derive(Debug)]
struct Point(i32, i32);

fn add(p1: &Point, p2: &Point) -> Point {
    Point(p1.0 + p2.0, p1.1 + p2.1)
}

fn main() {
    let p1 = Point(3, 4);
    let p2 = Point(10, 20);
    let p3 = add(&p1, &p2);
    println!("{p1:?} + {p2:?} = {p3:?}");
}
```

- The `add` function *borrows* two points and returns a new point.
- The caller retains ownership of the inputs.

▼ Details

Notes on stack returns:

- Demonstrate that the return from `add` is cheap because the compiler can eliminate the copy operation. Change the above code to print stack addresses and run it on the Playground. In the "DEBUG" optimization level, the addresses should change, while the stay the same when changing to the "RELEASE" setting:

```
#[derive(Debug)]
struct Point(i32, i32);

fn add(p1: &Point, p2: &Point) -> Point {
    let p = Point(p1.0 + p2.0, p1.1 + p2.1);
    println!("&p.0: {:p}", &p.0);
    p
}

fn main() {
    let p1 = Point(3, 4);
    let p2 = Point(10, 20);
    let p3 = add(&p1, &p2);
    println!("&p3.0: {:p}", &p3.0);
    println!("{p1:?} + {p2:?} = {p3:?}");
}
```

- The Rust compiler can do return value optimization (RVO).
- In C++, copy elision has to be defined in the language specification because constructors can have side effects. In Rust, this is not an issue at all. If RVO did not happen, Rust will always performs a simple and efficient `memcpy` copy.

Shared and Unique Borrows

Rust puts constraints on the ways you can borrow values:

- You can have one or more `&T` values at any given time, *or*
- You can have exactly one `&mut T` value.

```
fn main() {
    let mut a: i32 = 10;
    let b: &i32 = &a;

    {
        let c: &mut i32 = &mut a;
        *c = 20;
    }

    println!("a: {a}");
    println!("b: {b}");
}
```

▼ Details

- The above code does not compile because `a` is borrowed as mutable (through `c`) and as immutable (through `b`) at the same time.
- Move the `println!` statement for `b` before the scope that introduces `c` to make the code compile.
- After that change, the compiler realizes that `b` is only ever used before the new mutable borrow of `a` through `c`. This is a feature of the borrow checker called “non-lexical lifetimes”.

Lifetimes

A borrowed value has a *lifetime*:

- The lifetime can be elided: `add(p1: &Point, p2: &Point) -> Point`.
- Lifetimes can also be explicit: `&'a Point`, `&'document str`.
- Read `&'a Point` as “a borrowed `Point` which is valid for at least the lifetime `a`”.
- Lifetimes are always inferred by the compiler: you cannot assign a lifetime yourself.
  - Lifetime annotations create constraints; the compiler verifies that there is a valid solution.

Lifetimes in Function Calls

In addition to borrowing its arguments, a function can return a borrowed value:

```
#[derive(Debug)]
struct Point(i32, i32);

fn left_most<'a>(p1: &'a Point, p2: &'a Point) -> &'a Point {
    if p1.0 < p2.0 { p1 } else { p2 }
}

fn main() {
    let p1: Point = Point(10, 10);
    let p2: Point = Point(20, 20);
    let p3: &Point = left_most(&p1, &p2);
    println!("left-most point: {:?}", p3);
}
```

- `'a` is a generic parameter, it is inferred by the compiler.
- Lifetimes start with `'` and `'a` is a typical default name.
- Read `&'a Point` as "a borrowed `Point` which is valid for at least the lifetime `a`".
  - The *at least* part is important when parameters are in different scopes.

▼ Details

In the above example, try the following:

- Move the declaration of `p2` and `p3` into a a new scope (`{ ... }`), resulting in the following code:

  ```
  #[derive(Debug)]
  struct Point(i32, i32);

  fn left_most<'a>(p1: &'a Point, p2: &'a Point) -> &'a Point {
      if p1.0 < p2.0 { p1 } else { p2 }
  }

  fn main() {
      let p1: Point = Point(10, 10);
      let p3: &Point;
      {
          let p2: Point = Point(20, 20);
          p3 = left_most(&p1, &p2);
      }
      println!("left-most point: {:?}", p3);
  }
  ```

  Note how this does not compile since `p3` outlives `p2`.

- Reset the workspace and change the function signature to `fn left_most<'a, 'b>(p1: &'a Point, p2: &'a Point) -> &'b Point`. This will not compile because the relationship between the lifetimes `'a` and `'b` is unclear.

- Another way to explain it:

  - Two references to two values are borrowed by a function and the function returns another reference.
  - It must have come from one of those two inputs (or from a global variable).

    - Which one is it? The compiler needs to to know, so at the call site the returned reference is not used for longer than a variable from where the reference came from.

Lifetimes in Data Structures

If a data type stores borrowed data, it must be annotated with a lifetime:

```
#[derive(Debug)]
struct Highlight<'doc>(&'doc str);

fn erase(text: String) {
    println!("Bye {text}!");
}

fn main() {
    let text = String::from("The quick brown fox jumps over the lazy dog.");
    let fox = Highlight(&text[4..19]);
    let dog = Highlight(&text[35..43]);
    // erase(text);
    println!("{fox:?}");
    println!("{dog:?}");
}
```

▼ Details

- In the above example, the annotation on `Highlight` enforces that the data underlying the contained `&str` lives at least as long as any instance of `Highlight` that uses that data.
- If `text` is consumed before the end of the lifetime of `fox` (or `dog`), the borrow checker throws an error.
- Types with borrowed data force users to hold on to the original data. This can be useful for creating lightweight views, but it generally makes them somewhat harder to use.
- When possible, make data structures own their data directly.
- Some structs with multiple references inside can have more than one lifetime annotation. This can be necessary if there is a need to describe lifetime relationships between the references themselves, in addition to the lifetime of the struct itself. Those are very advanced use cases.

Day 1: Afternoon Exercises

We will look at two things:

- A small book library,
- Iterators and ownership (hard).

▼ Details

After looking at the exercises, you can look at the solutions provided.

Designing a Library

We will learn much more about structs and the `Vec<T>` type tomorrow. For now, you just need to know part of its API:

```
fn main() {
    let mut vec = vec![10, 20];
    vec.push(30);
    println!("middle value: {}", vec[vec.len() / 2]);
    for item in vec.iter() {
        println!("item: {item}");
    }
}
```

Use this to create a library application. Copy the code below to https://play.rust-lang.org/ and update the types to make it compile:

```rust
// TODO: remove this when you're done with your implementation.
#![allow(unused_variables, dead_code)]

struct Library {
    books: Vec<Book>,
}

struct Book {
    title: String,
    year: u16,
}

impl Book {
    // This is a constructor, used below.
    fn new(title: &str, year: u16) -> Book {
        Book {
            title: String::from(title),
            year,
        }
    }
}

// This makes it possible to print Book values with {}.
impl std::fmt::Display for Book {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{} ({})", self.title, self.year)
    }
}

impl Library {
    fn new() -> Library {
        unimplemented!()
    }

    //fn len(self) -> usize {
    //    unimplemented!()
    //}

    //fn is_empty(self) -> bool {
    //    unimplemented!()
    //}

    //fn add_book(self, book: Book) {
    //    unimplemented!()
    //}

    //fn print_books(self) {
    //    unimplemented!()
    //}

    //fn oldest_book(self) -> Option<&Book> {
    //    unimplemented!()
    //}
}

// This shows the desired behavior. Uncomment the code below and
// implement the missing methods. You will need to update the
// method signatures, including the "self" parameter! You may
// also need to update the variable bindings within main.
fn main() {
    let library = Library::new();

    //println!("Our library is empty: {}", library.is_empty());
    //
    //library.add_book(Book::new("Lord of the Rings", 1954));
    //library.add_book(Book::new("Alice's Adventures in Wonderland", 1865));
    //
    //library.print_books();
    //
    //match library.oldest_book() {
    //    Some(book) => println!("My oldest book is {book}"),
```

```
    //    None => println!("My library is empty!"),
    //}
    //
    //println!("Our library has {} books", library.len());
}
```

▼ Details

Solution

Iterators and Ownership

The ownership model of Rust affects many APIs. An example of this is the `Iterator` and `IntoIterator` traits.

`Iterator`

Traits are like interfaces: they describe behavior (methods) for a type. The `Iterator` trait simply says that you can call `next` until you get `None` back:

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

You use this trait like this:

```
fn main() {
    let v: Vec<i8> = vec![10, 20, 30];
    let mut iter = v.iter();

    println!("v[0]: {:?}", iter.next());
    println!("v[1]: {:?}", iter.next());
    println!("v[2]: {:?}", iter.next());
    println!("No more items: {:?}", iter.next());
}
```

What is the type returned by the iterator? Test your answer here:

```
fn main() {
    let v: Vec<i8> = vec![10, 20, 30];
    let mut iter = v.iter();

    let v0: Option<..> = iter.next();
    println!("v0: {v0:?}");
}
```

Why is this type used?

`IntoIterator`

The `Iterator` trait tells you how to *iterate* once you have created an iterator. The related trait `IntoIterator` tells you how to create the iterator:

```
pub trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item = Self::Item>;

    fn into_iter(self) -> Self::IntoIter;
}
```

The syntax here means that every implementation of `IntoIterator` must declare two types:

- `Item`: the type we iterate over, such as `i8`,
- `IntoIter`: the `Iterator` type returned by the `into_iter` method.

Note that `IntoIter` and `Item` are linked: the iterator must have the same `Item` type, which means that it returns `Option<Item>`

Like before, what is the type returned by the iterator?

```
fn main() {
    let v: Vec<String> = vec![String::from("foo"), String::from("bar")];
    let mut iter = v.into_iter();

    let v0: Option<..> = iter.next();
    println!("v0: {v0:?}");
}
```

`for` Loops

Now that we know both `Iterator` and `IntoIterator`, we can build `for` loops. They call `into_iter()` on an expression and iterates over the resulting iterator:

```
fn main() {
    let v: Vec<String> = vec![String::from("foo"), String::from("bar")];

    for word in &v {
        println!("word: {word}");
    }

    for word in v {
        println!("word: {word}");
    }
}
```

What is the type of `word` in each loop?

Experiment with the code above and then consult the documentation for `impl IntoIterator for &Vec<T>` and `impl IntoIterator for Vec<T>` to check your answers.

Welcome to Day 2

Now that we have seen a fair amount of Rust, we will continue with:

- Structs, enums, methods.
- Pattern matching: destructuring enums, structs, and arrays.
- Control flow constructs: `if`, `if let`, `while`, `while let`, `break`, and `continue`.
- The Standard Library: `String`, `Option` and `Result`, `Vec`, `HashMap`, `Rc` and `Arc`.
- Modules: visibility, paths, and filesystem hierarchy.

Structs

Like C and C++, Rust has support for custom structs:

```
struct Person {
    name: String,
    age: u8,
}

fn main() {
    let mut peter = Person {
        name: String::from("Peter"),
        age: 27,
    };
    println!("{} is {} years old", peter.name, peter.age);

    peter.age = 28;
    println!("{} is {} years old", peter.name, peter.age);

    let jackie = Person {
        name: String::from("Jackie"),
        ..peter
    };
    println!("{} is {} years old", jackie.name, jackie.age);
}
```

▼ Details

Key Points:

- Structs work like in C or C++.
  - Like in C++, and unlike in C, no typedef is needed to define a type.
  - Unlike in C++, there is no inheritance between structs.
- Methods are defined in an `impl` block, which we will see in following slides.
- This may be a good time to let people know there are different types of structs.
  - Zero-sized structs `e.g., struct Foo;` might be used when implementing a trait on some type but don't have any data that you want to store in the value itself.
  - The next slide will introduce Tuple structs.

Tuple Structs

If the field names are unimportant, you can use a tuple struct:

```
struct Point(i32, i32);

fn main() {
    let p = Point(17, 23);
    println!("({}, {})", p.0, p.1);
}
```

This is often used for single-field wrappers (called newtypes):

```
struct PoundOfForce(f64);
struct Newtons(f64);

fn compute_thruster_force() -> PoundOfForce {
    todo!("Ask a rocket scientist at NASA")
}

fn set_thruster_force(force: Newtons) {
    // ...
}

fn main() {
    let force = compute_thruster_force();
    set_thruster_force(force);
}
```

▼ Details

Newtypes are a great way to encode additional information about the value in a primitive type, for example:

- The number is measured in some units: `Newtons` in the example above.
- The value passed some validation when it was created, so you no longer have to validate it again at every use: 'PhoneNumber(String) `or` OddNumber(u32)`.

Field Shorthand Syntax

If you already have variables with the right names, then you can create the struct using a shorthand:

```
#[derive(Debug)]
struct Person {
    name: String,
    age: u8,
}

impl Person {
    fn new(name: String, age: u8) -> Person {
        Person { name, age }
    }
}

fn main() {
    let peter = Person::new(String::from("Peter"), 27);
    println!("{peter:?}");
}
```

▼ Details

The `new` function could be written using `Self` as a type, as it is interchangeable with the struct type name

```
impl Person {
    fn new(name: String, age: u8) -> Self {
        Self { name, age }
    }
}
```

Enums

The `enum` keyword allows the creation of a type which has a few different variants:

```
fn generate_random_number() -> i32 {
    4  // Chosen by fair dice roll. Guaranteed to be random.
}

#[derive(Debug)]
enum CoinFlip {
    Heads,
    Tails,
}

fn flip_coin() -> CoinFlip {
    let random_number = generate_random_number();
    if random_number % 2 == 0 {
        return CoinFlip::Heads;
    } else {
        return CoinFlip::Tails;
    }
}

fn main() {
    println!("You got: {:?}", flip_coin());
}
```

▼ Details

Key Points:

- Enumerations allow you to collect a set of values under one type
- This page offers an enum type `CoinFlip` with two variants `Heads` and `Tail`. You might note the namespace when using variants.
- This might be a good time to compare Structs and Enums:
  - In both, you can have a simple version without fields (unit struct) or one with different types of fields (variant payloads).
  - In both, associated functions are defined within an `impl` block.
  - You could even implement the different variants of an enum with separate structs but then they wouldn't be the same type as they would if they were all defined in an enum.

Variant Payloads

You can define richer enums where the variants carry data. You can then use the `match` statement to extract the data from each variant:

```rust
enum WebEvent {
    PageLoad,                 // Variant without payload
    KeyPress(char),           // Tuple struct variant
    Click { x: i64, y: i64 }, // Full struct variant
}

#[rustfmt::skip]
fn inspect(event: WebEvent) {
    match event {
        WebEvent::PageLoad       => println!("page loaded"),
        WebEvent::KeyPress(c)    => println!("pressed '{c}'"),
        WebEvent::Click { x, y } => println!("clicked at x={x}, y={y}"),
    }
}

fn main() {
    let load = WebEvent::PageLoad;
    let press = WebEvent::KeyPress('x');
    let click = WebEvent::Click { x: 20, y: 80 };

    inspect(load);
    inspect(press);
    inspect(click);
}
```

▼ Details

- In the above example, accessing the `char` in `KeyPress`, or `x` and `y` in `Click` only works within a `match` statement.
- `match` inspects a hidden discriminant field in the `enum`.
- `WebEvent::Click { ... }` is not exactly the same as `WebEvent::Click(Click)` with a top level `struct Click { ... }`. The inlined version cannot implement traits, for example.

Enum Sizes

Rust enums are packed tightly, taking constraints due to alignment into account:

```rust
use std::mem::{align_of, size_of};

macro_rules! dbg_size {
    ($t:ty) => {
        println!("{}: size {} bytes, align: {} bytes",
                 stringify!($t), size_of::<$t>(), align_of::<$t>());
    };
}

enum Foo {
    A,
    B,
}

#[repr(u32)]
enum Bar {
    A,  // 0
    B = 10000,
    C,  // 10001
}

fn main() {
    dbg_size!(Foo);
    dbg_size!(Bar);
    dbg_size!(bool);
    dbg_size!(Option<bool>);
    dbg_size!(&i32);
    dbg_size!(Option<&i32>);
}
```

- See the Rust Reference.

▼ Details

Key Points:

- Internally Rust is using a field (discriminant) to keep track of the enum variant.
- `Bar` enum demonstrates that there is a way to control the discriminant value and type. If `repr` is removed, the discriminant type takes 2 bytes, becuase 10001 fits 2 bytes.
- As a niche optimization an enum discriminant is merged with the pointer so that `Option<&Foo>` is the same size as `&Foo`.
- `Option<bool>` is another example of tight packing.
- For some types, Rust guarantees that `size_of::<T>()` equals `size_of::<Option<T>>()`.
- Zero-sized types allow for efficient implementation of `HashSet` using `HashMap` with `()` as the value.

Example code if you want to show how the bitwise representation *may* look like in practice. It’s important to note that the compiler provides no guarantees regarding this representation, therefore this is totally unsafe.

```rust
use std::mem::transmute;

macro_rules! dbg_bits {
    ($e:expr, $bit_type:ty) => {
        println!("- {}: {:#x}", stringify!($e), transmute::<_, $bit_type>($e));
    };
}

fn main() {
    // TOTALLY UNSAFE. Rust provides no guarantees about the bitwise
    // representation of types.
    unsafe {
        println!("Bitwise representation of bool");
        dbg_bits!(false, u8);
        dbg_bits!(true, u8);

        println!("Bitwise representation of Option<bool>");
        dbg_bits!(None::<bool>, u8);
        dbg_bits!(Some(false), u8);
        dbg_bits!(Some(true), u8);

        println!("Bitwise representation of Option<Option<bool>>");
        dbg_bits!(Some(Some(false)), u8);
        dbg_bits!(Some(Some(true)), u8);
        dbg_bits!(Some(None::<bool>), u8);
        dbg_bits!(None::<Option<bool>>, u8);

        println!("Bitwise representation of Option<&i32>");
        dbg_bits!(None::<&i32>, usize);
        dbg_bits!(Some(&0i32), usize);
    }
}
```

More complex example if you want to discuss what happens when we chain more than 256 `Option`s together.

```rust
#![recursion_limit = "1000"]

use std::mem::transmute;

macro_rules! dbg_bits {
    ($e:expr, $bit_type:ty) => {
        println!("- {}: {:#x}", stringify!($e), transmute::<_, $bit_type>($e));
    };
}

// Macro to wrap a value in 2^n Some() where n is the number of "@" signs.
// Increasing the recursion limit is required to evaluate this macro.
macro_rules! many_options {
    ($value:expr) => { Some($value) };
    ($value:expr, @) => {
        Some(Some($value))
    };
    ($value:expr, @ $($more:tt)+) => {
        many_options!(many_options!($value, $($more)+), $($more)+)
    };
}

fn main() {
    // TOTALLY UNSAFE. Rust provides no guarantees about the bitwise
    // representation of types.
    unsafe {
        assert_eq!(many_options!(false), Some(false));
        assert_eq!(many_options!(false, @), Some(Some(false)));
        assert_eq!(many_options!(false, @@), Some(Some(Some(Some(false)))));

        println!("Bitwise representation of a chain of 128 Option's.");
        dbg_bits!(many_options!(false, @@@@@@@), u8);
        dbg_bits!(many_options!(true, @@@@@@@), u8);

        println!("Bitwise representation of a chain of 256 Option's.");
        dbg_bits!(many_options!(false, @@@@@@@@), u16);
        dbg_bits!(many_options!(true, @@@@@@@@), u16);

        println!("Bitwise representation of a chain of 257 Option's.");
        dbg_bits!(many_options!(Some(false), @@@@@@@@), u16);
        dbg_bits!(many_options!(Some(true), @@@@@@@@), u16);
        dbg_bits!(many_options!(None::<bool>, @@@@@@@@), u16);
    }
}
```

Methods

Rust allows you to associate functions with your new types. You do this with an `impl` block:

```
#[derive(Debug)]
struct Person {
    name: String,
    age: u8,
}

impl Person {
    fn say_hello(&self) {
        println!("Hello, my name is {}", self.name);
    }
}

fn main() {
    let peter = Person {
        name: String::from("Peter"),
        age: 27,
    };
    peter.say_hello();
}
```

▼ Details

Key Points:

- It can be helpful to introduce methods by comparing them to functions.
    - Methods are called on an instance of a type (such as a struct or enum), the first parameter represents the instance as `self`.
    - Developers may choose to use methods to take advantage of method receiver syntax and to help keep them more organized. By using methods we can keep all the implementation code in one predictable place.
- Point out the use of the keyword `self`, a method receiver.
    - Show that it is an abbreviated term for `self:&Self` and perhaps show how the struct name could also be used.
    - Explain that Self is a type alias for the type the `impl` block is in and can be used elsewhere in the block.
    - Note how self is used like other structs and dot notation can be used to refer to individual fields.
    - This might be a good time to demonstrate how the `&self` differs from `self` by modifying the code and trying to run say_hello twice.
- We describe the distinction between method receivers next.

Method Receiver

The `&self` above indicates that the method borrows the object immutably. There are other possible receivers for a method:

- `&self`: borrows the object from the caller using a shared and immutable reference. The object can be used again afterwards.
- `&mut self`: borrows the object from the caller using a unique and mutable reference. The object can be used again afterwards.
- `self`: takes ownership of the object and moves it away from the caller. The method becomes the owner of the object. The object will be dropped (deallocated) when the method returns, unless its ownership is explicitly transmitted.
- `mut self`: same as above, but while the method owns the object, it can mutate it too. Complete ownership does not automatically mean mutability.
- No receiver: this becomes a static method on the struct. Typically used to create constructors which are called `new` by convention.

Beyond variants on `self`, there are also special wrapper types allowed to be receiver types, such as `Box<Self>`.

▼ Details

Consider emphasizing on “shared and immutable” and “unique and mutable”. These constraints always come together in Rust due to borrow checker rules, and `self` is no exception. It won’t be possible to reference a struct from multiple locations and call a mutating (`&mut self`) method on it.

Example

```
#[derive(Debug)]
struct Race {
    name: String,
    laps: Vec<i32>,
}

impl Race {
    fn new(name: &str) -> Race {  // No receiver, a static method
        Race { name: String::from(name), laps: Vec::new() }
    }

    fn add_lap(&mut self, lap: i32) {  // Exclusive borrowed read-write access to self
        self.laps.push(lap);
    }

    fn print_laps(&self) {  // Shared and read-only borrowed access to self
        println!("Recorded {} laps for {}:", self.laps.len(), self.name);
        for (idx, lap) in self.laps.iter().enumerate() {
            println!("Lap {idx}: {lap} sec");
        }
    }

    fn finish(self) {  // Exclusive ownership of self
        let total = self.laps.iter().sum::<i32>();
        println!("Race {} is finished, total lap time: {}", self.name, total);
    }
}

fn main() {
    let mut race = Race::new("Monaco Grand Prix");
    race.add_lap(70);
    race.add_lap(68);
    race.print_laps();
    race.add_lap(71);
    race.print_laps();
    race.finish();
    // race.add_lap(42);
}
```

▼ Details

Key Points:

- All four methods here use a different method receiver.
  - You can point out how that changes what the function can do with the variable values and if/how it can be used again in `main`.
  - You can showcase the error that appears when trying to call `finish` twice.
- Note, that although the method receivers are different, the non-static functions are called the same way in the main body. Rust enables automatic referencing and dereferencing when calling methods. Rust automatically adds in the `&`, `*`, `muts` so that that object matches the method signature.
- You might point out that `print_laps` is using a vector that is iterated over. We describe vectors in more detail in the afternoon.

Pattern Matching

The `match` keyword let you match a value against one or more *patterns*. The comparisons are done from top to bottom and the first match wins.

The patterns can be simple values, similarly to `switch` in C and C++:

```
fn main() {
    let input = 'x';

    match input {
        'q'                   => println!("Quitting"),
        'a' | 's' | 'w' | 'd' => println!("Moving around"),
        '0'..='9'             => println!("Number input"),
        _                     => println!("Something else"),
    }
}
```

The `_` pattern is a wildcard pattern which matches any value.

▼ Details

Key Points:

- You might point out how some specific characters are being used when in a patten
  - `|` as an `or`
  - `..` can expand as much as it needs to be
  - `1..=5` represents an inclusive range
  - `_` is a wild card
- It can be useful to show how binding works, by for instance replacing a wildcard character with a variable, or removing the quotes around `q`.
- You can demonstrate matching on a reference.
- This might be a good time to bring up the concept of irrefutable patterns, as the term can show up in error messages.

Destructuring Enums

Patterns can also be used to bind variables to parts of your values. This is how you inspect the structure of your types. Let us start with a simple `enum` type:

```
enum Result {
    Ok(i32),
    Err(String),
}

fn divide_in_two(n: i32) -> Result {
    if n % 2 == 0 {
        Result::Ok(n / 2)
    } else {
        Result::Err(format!("cannot divide {n} into two equal parts"))
    }
}

fn main() {
    let n = 100;
    match divide_in_two(n) {
        Result::Ok(half) => println!("{n} divided in two is {half}"),
        Result::Err(msg) => println!("sorry, an error happened: {msg}"),
    }
}
```

Here we have used the arms to *destructure* the `Result` value. In the first arm, `half` is bound to the value inside the `Ok` variant. In the second arm, `msg` is bound to the error message.

▼ Details

Key points:

- The `if` / `else` expression is returning an enum that is later unpacked with a `match`.
- You can try adding a third variant to the enum definition and displaying the errors when running the code. Point out the places where your code is now inexhaustive and how the compiler tries to give you hints.

Destructuring Structs

You can also destructure `structs`:

```
struct Foo {
    x: (u32, u32),
    y: u32,
}

#[rustfmt::skip]
fn main() {
    let foo = Foo { x: (1, 2), y: 3 };
    match foo {
        Foo { x: (1, b), y } => println!("x.0 = 1, b = {b}, y = {y}"),
        Foo { y: 2, x: i }   => println!("y = 2, i = {i:?}"),
        Foo { y, .. }        => println!("y = {y}, other fields were ignored"),
    }
}
```

▼ Details

- Change the literal values in `foo` to match with the other patterns.
- Add a new field to `Foo` and make changes to the pattern as needed.

Destructuring Arrays

You can destructure arrays, tuples, and slices by matching on their elements:

```
#[rustfmt::skip]
fn main() {
    let triple = [0, -2, 3];
    println!("Tell me about {triple:?}");
    match triple {
        [0, y, z] => println!("First is 0, y = {y}, and z = {z}"),
        [1, ..]   => println!("First is 1 and the rest were ignored"),
        _         => println!("All elements were ignored"),
    }
}
```

▼ Details

Destructuring of slices of unknown length also works with patterns of fixed length.

```
fn main() {
    inspect(&[0, -2, 3]);
    inspect(&[0, -2, 3, 4]);
}

#[rustfmt::skip]
fn inspect(slice: &[i32]) {
    println!("Tell me about {slice:?}");
    match slice {
        &[0, y, z] => println!("First is 0, y = {y}, and z = {z}"),
        &[1, ..]   => println!("First is 1 and the rest were ignored"),
        _          => println!("All elements were ignored"),
    }
}
```

Match Guards

When matching, you can add a *guard* to a pattern. This is an arbitrary Boolean expression which will be executed if the pattern matches:

```
#[rustfmt::skip]
fn main() {
    let pair = (2, -2);
    println!("Tell me about {pair:?}");
    match pair {
        (x, y) if x == y     => println!("These are twins"),
        (x, y) if x + y == 0 => println!("Antimatter, kaboom!"),
        (x, _) if x % 2 == 1 => println!("The first one is odd"),
        _                    => println!("No correlation..."),
    }
}
```

▼ Details

Key Points:

- Match guards as a separate syntax feature are important and necessary.
- They are not the same as separate `if` expression inside of the match arm. An `if` expression inside of the branch block (after `=>`) happens after the match arm is selected. Failing the `if` condition inside of that block won't result in other arms of the original `match` expression being considered.
- You can use the variables defined in the pattern in your if expression.
- The condition defined in the guard applies to every expression in a pattern with an `|`.

Day 2: Morning Exercises

We will look at implementing methods in two contexts:

- Simple struct which tracks health statistics.
- Multiple structs and enums for a drawing library.

▼ Details

After looking at the exercises, you can look at the solutions provided.

Health Statistics

You're working on implementing a health-monitoring system. As part of that, you need to keep track of users' health statistics.

You'll start with some stubbed functions in an `impl` block as well as a `User` struct definition. Your goal is to implement the stubbed out methods on the `User` `struct` defined in the `impl` block.

Copy the code below to https://play.rust-lang.org/ and fill in the missing methods:

```
// TODO: remove this when you're done with your implementation.
#![allow(unused_variables, dead_code)]

struct User {
    name: String,
    age: u32,
    weight: f32,
}

impl User {
    pub fn new(name: String, age: u32, weight: f32) -> Self {
        unimplemented!()
    }

    pub fn name(&self) -> &str {
        unimplemented!()
    }

    pub fn age(&self) -> u32 {
        unimplemented!()
    }

    pub fn weight(&self) -> f32 {
        unimplemented!()
    }

    pub fn set_age(&mut self, new_age: u32) {
        unimplemented!()
    }

    pub fn set_weight(&mut self, new_weight: f32) {
        unimplemented!()
    }
}

fn main() {
    let bob = User::new(String::from("Bob"), 32, 155.2);
    println!("I'm {} and my age is {}", bob.name(), bob.age());
}

#[test]
fn test_weight() {
    let bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.weight(), 155.2);
}

#[test]
fn test_set_age() {
    let mut bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.age(), 32);
    bob.set_age(33);
    assert_eq!(bob.age(), 33);
}
```

Polygon Struct

We will create a `Polygon` struct which contain some points. Copy the code below to https://play.rust-lang.org/ and fill in the missing methods to make the tests pass:

```rust
// TODO: remove this when you're done with your implementation.
#![allow(unused_variables, dead_code)]

pub struct Point {
    // add fields
}

impl Point {
    // add methods
}

pub struct Polygon {
    // add fields
}

impl Polygon {
    // add methods
}

pub struct Circle {
    // add fields
}

impl Circle {
    // add methods
}

pub enum Shape {
    Polygon(Polygon),
    Circle(Circle),
}

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

    fn round_two_digits(x: f64) -> f64 {
        (x * 100.0).round() / 100.0
    }

    #[test]
    fn test_point_magnitude() {
        let p1 = Point::new(12, 13);
        assert_eq!(round_two_digits(p1.magnitude()), 17.69);
    }

    #[test]
    fn test_point_dist() {
        let p1 = Point::new(10, 10);
        let p2 = Point::new(14, 13);
        assert_eq!(round_two_digits(p1.dist(p2)), 5.00);
    }

    #[test]
    fn test_point_add() {
        let p1 = Point::new(16, 16);
        let p2 = p1 + Point::new(-4, 3);
        assert_eq!(p2, Point::new(12, 19));
    }

    #[test]
    fn test_polygon_left_most_point() {
        let p1 = Point::new(12, 13);
        let p2 = Point::new(16, 16);

        let mut poly = Polygon::new();
        poly.add_point(p1);
        poly.add_point(p2);
        assert_eq!(poly.left_most_point(), Some(p1));
    }
```

```
    #[test]
    fn test_polygon_iter() {
        let p1 = Point::new(12, 13);
        let p2 = Point::new(16, 16);

        let mut poly = Polygon::new();
        poly.add_point(p1);
        poly.add_point(p2);

        let points = poly.iter().cloned().collect::<Vec<_>>();
        assert_eq!(points, vec![Point::new(12, 13), Point::new(16, 16)]);
    }

    #[test]
    fn test_shape_perimeters() {
        let mut poly = Polygon::new();
        poly.add_point(Point::new(12, 13));
        poly.add_point(Point::new(17, 11));
        poly.add_point(Point::new(16, 16));
        let shapes = vec![
            Shape::from(poly),
            Shape::from(Circle::new(Point::new(10, 20), 5)),
        ];
        let perimeters = shapes
            .iter()
            .map(Shape::perimeter)
            .map(round_two_digits)
            .collect::<Vec<_>>();
        assert_eq!(perimeters, vec![15.48, 31.42]);
    }
}

#[allow(dead_code)]
fn main() {}
```

▼ Details

Since the method signatures are missing from the problem statements, the key part of the exercise is to specify those correctly.

Other interesting parts of the exercise:

- Derive a `Copy` trait for some structs, as in tests the methods sometimes don't borrow their arguments.
- Discover that `Add` trait must be implemented for two objects to be addable via "+".

Control Flow

As we have seen, `if` is an expression in Rust. It is used to conditionally evaluate one of two blocks, but the blocks can have a value which then becomes the value of the `if` expression. Other control flow expressions work similarly in Rust.

Blocks

A block in Rust has a value and a type: the value is the last expression of the block:

```
fn main() {
    let x = {
        let y = 10;
        println!("y: {y}");
        let z = {
            let w = {
                3 + 4
            };
            println!("w: {w}");
            y * w
        };
        println!("z: {z}");
        z - y
    };
    println!("x: {x}");
}
```

The same rule is used for functions: the value of the function body is the return value:

```
fn double(x: i32) -> i32 {
    x + x
}

fn main() {
    println!("doubled: {}", double(7));
}
```

However if the last expression ends with `;`, then the resulting value and type is `()`.

▼ Details

Key Points:

- The point of this slide is to show that blocks have a type and value in Rust.
- You can show how the value of the block changes by changing the last line in the block. For instance, adding/removing a semicolon or using a `return`.

`if` expressions

You use `if` very similarly to how you would in other languages:

```
fn main() {
    let mut x = 10;
    if x % 2 == 0 {
        x = x / 2;
    } else {
        x = 3 * x + 1;
    }
}
```

In addition, you can use it as an expression. This does the same as above:

```
fn main() {
    let mut x = 10;
    x = if x % 2 == 0 {
        x / 2
    } else {
        3 * x + 1
    };
}
```

▼ Details

Because `if` is an expression and must have a particular type, both of its branch blocks must have the same type. Consider showing what happens if you add `;` after `x / 2` in the second example.

`if let` expressions

If you want to match a value against a pattern, you can use `if let`:

```
fn main() {
    let arg = std::env::args().next();
    if let Some(value) = arg {
        println!("Program name: {value}");
    } else {
        println!("Missing name?");
    }
}
```

See pattern matching for more details on patterns in Rust.

▼ Details

- `if let` can be more concise than `match`, e.g., when only one case is interesting. In contrast, `match` requires all branches to be covered.
  - For the similar use case consider demonstrating a newly stabilized `let else` feature.
- A common usage is handling `Some` values when working with `Option`.
- Unlike `match`, `if let` does not support guard clauses for pattern matching.

`while` expressions

The `while` keyword works very similar to other languages:

```
fn main() {
    let mut x = 10;
    while x != 1 {
        x = if x % 2 == 0 {
            x / 2
        } else {
            3 * x + 1
        };
    }
    println!("Final x: {x}");
}
```

`while let` expressions

Like with `if`, there is a `while let` variant which repeatedly tests a value against a pattern:

```
fn main() {
    let v = vec![10, 20, 30];
    let mut iter = v.into_iter();

    while let Some(x) = iter.next() {
        println!("x: {x}");
    }
}
```

Here the iterator returned by `v.iter()` will return a `Option<i32>` on every call to `next()`. It returns `Some(x)` until it is done, after which it will return `None`. The `while let` lets us keep iterating through all items.

See pattern matching for more details on patterns in Rust.

▼ Details

- Point out that the `while let` loop will keep going as long as the value matches the pattern.
- You could rewrite the `while let` loop as an infinite loop with an if statement that breaks when there is no value to unwrap for `iter.next()`. The `while let` provides syntactic sugar for the above scenario.

`for` expressions

The `for` expression is closely related to the `while let` expression. It will automatically call `into_iter()` on the expression and then iterate over it:

```
fn main() {
    let v = vec![10, 20, 30];

    for x in v {
        println!("x: {x}");
    }

    for i in (0..10).step_by(2) {
        println!("i: {i}");
    }
}
```

You can use `break` and `continue` here as usual.

▼ Details

- Index iteration is not a special syntax in Rust for just that case.
- `(0..10)` is a range that implements an `Iterator` trait.
- `step_by` is a method that returns another `Iterator` that skips every other element.

`loop` expressions

Finally, there is a `loop` keyword which creates an endless loop. Here you must either `break` or `return` to stop the loop:

```rust
fn main() {
    let mut x = 10;
    loop {
        x = if x % 2 == 0 {
            x / 2
        } else {
            3 * x + 1
        };
        if x == 1 {
            break;
        }
    }
    println!("Final x: {x}");
}
```

`match` expressions

The `match` keyword is used to match a value against one or more patterns. In that sense, it works like a series of `if let` expressions:

```
fn main() {
    match std::env::args().next().as_deref() {
        Some("cat") => println!("Will do cat things"),
        Some("ls")  => println!("Will ls some files"),
        Some("mv")  => println!("Let's move some files"),
        Some("rm")  => println!("Uh, dangerous!"),
        None        => println!("Hmm, no program name?"),
        _           => println!("Unknown program name!"),
    }
}
```

Like `if let`, each match arm must have the same type. The type is the last expression of the block, if any. In the example above, the type is `()`.

See pattern matching for more details on patterns in Rust.

`break` and `continue`

If you want to exit a loop early, use `break`, if you want to immediately start the next iteration use `continue`. Both `continue` and `break` can optionally take a label argument which is used to break out of nested loops:

```rust
fn main() {
    let v = vec![10, 20, 30];
    let mut iter = v.into_iter();
    'outer: while let Some(x) = iter.next() {
        println!("x: {x}");
        let mut i = 0;
        while i < x {
            println!("x: {x}, i: {i}");
            i += 1;
            if i == 3 {
                break 'outer;
            }
        }
    }
}
```

In this case we break the outer loop after 3 iterations of the inner loop.

Standard Library

Rust comes with a standard library which helps establish a set of common types used by Rust library and programs. This way, two libraries can work together smoothly because they both use the same `String` type.

The common vocabulary types include:

- `Option` and `Result` types: used for optional values and error handling.
- `String`: the default string type used for owned data.
- `Vec`: a standard extensible vector.
- `HashMap`: a hash map type with a configurable hashing algorithm.
- `Box`: an owned pointer for heap-allocated data.
- `Rc`: a shared reference-counted pointer for heap-allocated data.

▼ Details

- In fact, Rust contains several layers of the Standard Library: `core`, `alloc` and `std`.
- `core` includes the most basic types and functions that don't depend on `libc`, allocator or even the presence of an operating system.
- `alloc` includes types which require a global heap allocator, such as `Vec`, `Box` and `Arc`.
- Embedded Rust applications often only use `core`, and sometimes `alloc`.

Option and Result

The types represent optional data:

```
fn main() {
    let numbers = vec![10, 20, 30];
    let first: Option<&i8> = numbers.first();
    println!("first: {first:?}");

    let idx: Result<usize, usize> = numbers.binary_search(&10);
    println!("idx: {idx:?}");
}
```

▼ Details

- `Option` and `Result` are widely used not just in the standard library.
- `Option<&T>` has zero space overhead compared to `&T`.
- `Result` is the standard type to implement error handling as we will see on Day 3.
- `binary_search` returns `Result<usize, usize>`.
  - If found, `Result::Ok` holds the index where the element is found.
  - Otherwise, `Result::Err` contains the index where such an element should be inserted.

String

`String` is the standard heap-allocated growable UTF-8 string buffer:

```rust
fn main() {
    let mut s1 = String::new();
    s1.push_str("Hello");
    println!("s1: len = {}, capacity = {}", s1.len(), s1.capacity());

    let mut s2 = String::with_capacity(s1.len() + 1);
    s2.push_str(&s1);
    s2.push('!');
    println!("s2: len = {}, capacity = {}", s2.len(), s2.capacity());

    let s3 = String::from("🇨🇭");
    println!("s3: len = {}, number of chars = {}", s3.len(),
             s3.chars().count());
}
```

`String` implements `Deref<Target = str>`, which means that you can call all `str` methods on a `String`.

▼ Details

- `len` returns the size of the `String` in bytes, not its length in characters.
- `chars` returns an iterator over the actual characters.
- `String` implements `Deref<Target = str>` which transparently gives it access to `str`'s methods.

Vec

`Vec` is the standard resizable heap-allocated buffer:

```
fn main() {
    let mut v1 = Vec::new();
    v1.push(42);
    println!("v1: len = {}, capacity = {}", v1.len(), v1.capacity());

    let mut v2 = Vec::with_capacity(v1.len() + 1);
    v2.extend(v1.iter());
    v2.push(9999);
    println!("v2: len = {}, capacity = {}", v2.len(), v2.capacity());
}
```

`Vec` implements `Deref<Target = [T]>`, which means that you can call slice methods on a `Vec`.

▼ Details

- `Vec` is a type of collection, along with `String` and `HashMap`. The data it contains is stored on the heap. This means the amount of data doesn't need to be known at compile time. It can grow or shrink at runtime.
- Notice how `Vec<T>` is a generic type too, but you don't have to specify `T` explicitly. As always with Rust type inference, the `T` was established during the first `push` call.
- `vec![...]` is a canonical macro to use instead of `Vec::new()` and it supports adding initial elements to the vector.
- To index the vector you use `[ ]`, but they will panic if out of bounds. Alternatively, using `get` will return an `Option`. The `pop` function will remove the last element.
- Show iterating over a vector and mutating the value: `for e in &mut v { *e += 50; }`

HashMap

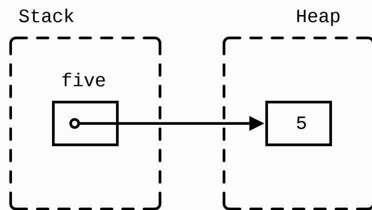
Standard hash map with protection against HashDoS attacks:

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

fn main() {
    let mut page_counts = HashMap::new();
    page_counts.insert("Adventures of Huckleberry Finn".to_string(), 207);
    page_counts.insert("Grimms' Fairy Tales".to_string(), 751);
    page_counts.insert("Pride and Prejudice".to_string(), 303);

    if !page_counts.contains_key("Les Misérables") {
        println!("We've know about {} books, but not Les Misérables.",
                 page_counts.len());
    }

    for book in ["Pride and Prejudice", "Alice's Adventure in Wonderland"] {
        match page_counts.get(book) {
            Some(count) => println!("{book}: {count} pages"),
            None => println!("{book} is unknown.")
        }
    }
}
```

Box

`Box` is an owned pointer to data on the heap:

```
fn main() {
    let five = Box::new(5);
    println!("five: {}", *five);
}
```



`Box<T>` implements `Deref<Target = T>`, which means that you can call methods from `T` directly on a `Box<T>`.

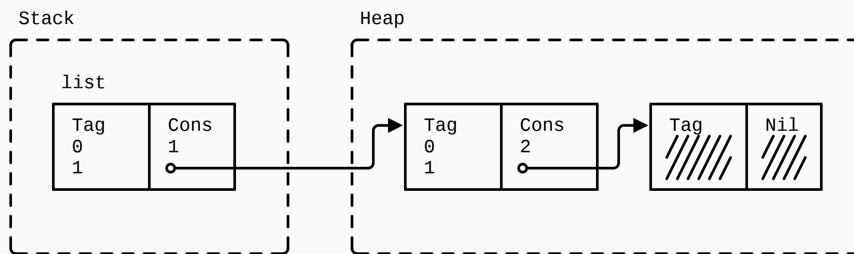
▼ Details

- `Box` is like `std::unique_ptr` in C++.
- In the above example, you can even leave out the `*` in the `println!` statement thanks to `Deref`.

Box with Recursive Data Structures

Recursive data types or data types with dynamic sizes need to use a `Box`:

```
#[derive(Debug)]
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

fn main() {
    let list: List<i32> = List::Cons(1, Box::new(List::Cons(2, Box::new(List::Nil))));
    println!("{list:?}");
}
```



▼ Details

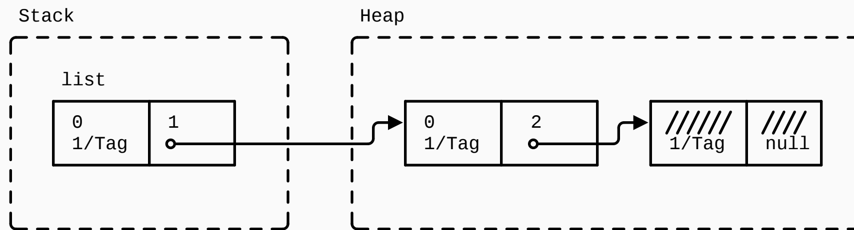
If the `Box` was not used here and we attempted to embed a `List` directly into the `List`, the compiler would not compute a fixed size of the struct in memory, it would look infinite.

`Box` solves this problem as it has the same size as a regular pointer and just points at the next element of the `List` in the heap.

Niche Optimization

```
#[derive(Debug)]
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

fn main() {
    let list: List<i32> = List::Cons(1, Box::new(List::Cons(2, Box::new(List::Nil))));
    println!("{list:?}");
}
```

A `Box` cannot be empty, so the pointer is always valid and non-`null`. This allows the compiler to optimize the memory layout:



Rc

`Rc` is a reference-counted shared pointer. Use this when you need to refer to the same data from multiple places:

```
use std::rc::Rc;

fn main() {
    let mut a = Rc::new(10);
    let mut b = a.clone();

    println!("a: {a}");
    println!("b: {b}");
}
```

If you need to mutate the data inside an `Rc`, you will need to wrap the data in a type such as `Cell` or `RefCell`. See `Arc` if you are in a multi-threaded context.

▼ Details

- Like C++'s `std::shared_ptr`.
- `clone` is cheap: creates a pointer to the same allocation and increases the reference count.
- `make_mut` actually clones the inner value if necessary ("clone-on-write") and returns a mutable reference.

Modules

We have seen how `impl` blocks let us namespace functions to a type.

Similarly, `mod` lets us namespace types and functions:

```
mod foo {
    pub fn do_something() {
        println!("In the foo module");
    }
}

mod bar {
    pub fn do_something() {
        println!("In the bar module");
    }
}

fn main() {
    foo::do_something();
    bar::do_something();
}
```

Visibility

Modules are a privacy boundary:

- Module items are private by default (hides implementation details).
- Parent and sibling items are always visible.

```rust
mod outer {
    fn private() {
        println!("outer::private");
    }

    pub fn public() {
        println!("outer::public");
    }

    mod inner {
        fn private() {
            println!("outer::inner::private");
        }

        pub fn public() {
            println!("outer::inner::public");
            super::private();
        }
    }
}

fn main() {
    outer::public();
}
```

Paths

Paths are resolved as follows:

1. As a relative path:
   - `foo` or `self::foo` refers to `foo` in the current module,
   - `super::foo` refers to `foo` in the parent module.
2. As an absolute path:
   - `crate::foo` refers to `foo` in the root of the current crate,
   - `bar::foo` refers to `foo` in the `bar` crate.

Filesystem Hierarchy

The module content can be omitted:

```
mod garden;
```

The `garden` module content is found at:

- `src/garden.rs` (modern Rust 2018 style)
- `src/garden/mod.rs` (older Rust 2015 style)

Similarly, a `garden::vegetables` module can be found at:

- `src/garden/vegetables.rs` (modern Rust 2018 style)
- `src/garden/vegetables/mod.rs` (older Rust 2015 style)

The `crate` root is in:

- `src/lib.rs` (for a library crate)
- `src/main.rs` (for a binary crate)

Day 2: Afternoon Exercises

The exercises for this afternoon will focus on strings and iterators.

▼ Details

After looking at the exercises, you can look at the solutions provided.

Luhn Algorithm

The Luhn algorithm is used to validate credit card numbers. The algorithm takes a string as input and does the following to validate the credit card number:

- Ignore all spaces. Reject number with less than two digits.
- Moving from right to left, double every second digit: for the number `1234`, we double `3` and `1`.
- After doubling a digit, sum the digits. So doubling `7` becomes `14` which becomes `5`.
- Sum all the undoubled and doubled digits.
- The credit card number is valid if the sum ends with `0`.

Copy the following code to https://play.rust-lang.org/ and implement the function:

```
// TODO: remove this when you're done with your implementation.
#![allow(unused_variables, dead_code)]

pub fn luhn(cc_number: &str) -> bool {
    unimplemented!()
}

#[test]
fn test_non_digit_cc_number() {
    assert!(!luhn("foo"));
}

#[test]
fn test_empty_cc_number() {
    assert!(!luhn(""));
    assert!(!luhn(" "));
    assert!(!luhn("  "));
    assert!(!luhn("    "));
}

#[test]
fn test_single_digit_cc_number() {
    assert!(!luhn("0"));
}

#[test]
fn test_two_digit_cc_number() {
    assert!(luhn(" 0 0 "));
}

#[test]
fn test_valid_cc_number() {
    assert!(luhn("4263 9826 4026 9299"));
    assert!(luhn("4539 3195 0343 6467"));
    assert!(luhn("7992 7398 713"));
}

#[test]
fn test_invalid_cc_number() {
    assert!(!luhn("4223 9826 4026 9299"));
    assert!(!luhn("4539 3195 0343 6476"));
    assert!(!luhn("8273 1232 7352 0569"));
}

#[allow(dead_code)]
fn main() {}
```

Strings and Iterators

In this exercise, you are implementing a routing component of a web server. The server is configured with a number of *path prefixes* which are matched against *request paths*. The path prefixes can contain a wildcard character which matches a full segment. See the unit tests below.

Copy the following code to https://play.rust-lang.org/ and make the tests pass. Try avoiding allocating a `Vec` for your intermediate results:

```rust
// TODO: remove this when you're done with your implementation.
#![allow(unused_variables, dead_code)]

pub fn prefix_matches(prefix: &str, request_path: &str) -> bool {
    unimplemented!()
}

#[test]
fn test_matches_without_wildcard() {
    assert!(prefix_matches("/v1/publishers", "/v1/publishers"));
    assert!(prefix_matches("/v1/publishers", "/v1/publishers/abc-123"));
    assert!(prefix_matches("/v1/publishers", "/v1/publishers/abc/books"));

    assert!(!prefix_matches("/v1/publishers", "/v1"));
    assert!(!prefix_matches("/v1/publishers", "/v1/publishersBooks"));
    assert!(!prefix_matches("/v1/publishers", "/v1/parent/publishers"));
}

#[test]
fn test_matches_with_wildcard() {
    assert!(prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/foo/books"
    ));
    assert!(prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/bar/books"
    ));
    assert!(prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/foo/books/book1"
    ));

    assert!(!prefix_matches("/v1/publishers/*/books", "/v1/publishers"));
    assert!(!prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/foo/booksByAuthor"
    ));
}
```

Welcome to Day 3

Today, we will cover some more advanced topics of Rust:

- Traits: deriving traits, default methods, and important standard library traits.
- Generics: generic data types, generic methods, monomorphization, and trait objects.
- Error handling: panics, `Result`, and the try operator `?`.
- Testing: unit tests, documentation tests, and integration tests.
- Unsafe Rust: raw pointers, static variables, unsafe functions, and extern functions.

Traits

Rust lets you abstract over types with traits. They're similar to interfaces:

```
trait Greet {
    fn say_hello(&self);
}

struct Dog {
    name: String,
}

struct Cat;  // No name, cats won't respond to it anyway.

impl Greet for Dog {
    fn say_hello(&self) {
        println!("Wuf, my name is {}!", self.name);
    }
}

impl Greet for Cat {
    fn say_hello(&self) {
        println!("Miau!");
    }
}

fn main() {
    let pets: Vec<Box<dyn Greet>> = vec![
        Box::new(Dog { name: String::from("Fido") }),
        Box::new(Cat),
    ];
    for pet in pets {
        pet.say_hello();
    }
}
```

▼ Details

- Traits may specify pre-implemented (default) methods and methods that users are required to implement themselves. Methods with default implementations can rely on required methods.
- Types that implement a given trait may be of different sizes. This makes it impossible to have things like `Vec<Greet>` in the example above.
- `dyn Greet` is a way to tell the compiler about a dynamically sized type that implements `Greet`.
- In the example, `pets` holds Fat Pointers to objects that implement `Greet`. The Fat Pointer consists of two components, a pointer to the actual object and a pointer to the virtual method table for the `Greet` implementation of that particular object.

Compare these outputs in the above example:

```
    println!("{} {}", std::mem::size_of::<Dog>(), std::mem::size_of::<Cat>());
    println!("{} {}", std::mem::size_of::<&Dog>(), std::mem::size_of::<&Cat>());
    println!("{}", std::mem::size_of::<&dyn Greet>());
    println!("{}", std::mem::size_of::<Box<dyn Greet>>());
```

Deriving Traits

You can let the compiler derive a number of traits:

```
#[derive(Debug, Clone, PartialEq, Eq, Default)]
struct Player {
    name: String,
    strength: u8,
    hit_points: u8,
}

fn main() {
    let p1 = Player::default();
    let p2 = p1.clone();
    println!("Is {:?}\nequal to {:?}?\nThe answer is {}!", &p1, &p2,
             if p1 == p2 { "yes" } else { "no" });
}
```

Default Methods

Traits can implement behavior in terms of other trait methods:

```
trait Equals {
    fn equal(&self, other: &Self) -> bool;
    fn not_equal(&self, other: &Self) -> bool {
        !self.equal(other)
    }
}

#[derive(Debug)]
struct Centimeter(i16);

impl Equals for Centimeter {
    fn equal(&self, other: &Centimeter) -> bool {
        self.0 == other.0
    }
}

fn main() {
    let a = Centimeter(10);
    let b = Centimeter(20);
    println!("{a:?} equals {b:?}: {}", a.equal(&b));
    println!("{a:?} not_equals {b:?}: {}", a.not_equal(&b));
}
```

Important Traits

We will now look at some of the most common traits of the Rust standard library:

- `Iterator` and `IntoIterator` used in `for` loops,
- `From` and `Into` used to convert values,
- `Read` and `Write` used for IO,
- `Add`, `Mul`, ... used for operator overloading, and
- `Drop` used for defining destructors.
- `Default` used to construct a default instance of a type.

Iterators

You can implement the `Iterator` trait on your own types:

```
struct Fibonacci {
    curr: u32,
    next: u32,
}

impl Iterator for Fibonacci {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        let new_next = self.curr + self.next;
        self.curr = self.next;
        self.next = new_next;
        Some(self.curr)
    }
}

fn main() {
    let fib = Fibonacci { curr: 0, next: 1 };
    for (i, n) in fib.enumerate().take(5) {
        println!("fib({i}): {n}");
    }
}
```

▼ Details

- `IntoIterator` is the trait that makes for loops work. It is implemented by collection types such as `Vec<T>` and references to them such as `&Vec<T>` and `&[T]`. Ranges also implement it.
- The `Iterator` trait implements many common functional programming operations over collections (e.g. `map`, `filter`, `reduce`, etc). This is the trait where you can find all the documentation about them. In Rust these functions should produce the code as efficient as equivalent imperative implementations.

FromIterator

`FromIterator` lets you build a collection from an `Iterator`.

```
fn main() {
    let primes = vec![2, 3, 5, 7];
    let prime_squares = primes
        .into_iter()
        .map(|prime| prime * prime)
        .collect::<Vec<_>>();
}
```

▼ Details

`Iterator` implements `fn collect<B>(self) -> B where B: FromIterator<Self::Item>, Self: Sized`

There are also implementations which let you do cool things like convert an `Iterator<Item = Result<V, E>>` into a `Result<Vec<V>, E>`.

From and Into

Types implement `From` and `Into` to facilitate type conversions:

```
fn main() {
    let s = String::from("hello");
    let addr = std::net::Ipv4Addr::from([127, 0, 0, 1]);
    let one = i16::from(true);
    let bigger = i32::from(123i16);
    println!("{s}, {addr}, {one}, {bigger}");
}
```

`Into` is automatically implemented when `From` is implemented:

```
fn main() {
    let s: String = "hello".into();
    let addr: std::net::Ipv4Addr = [127, 0, 0, 1].into();
    let one: i16 = true.into();
    let bigger: i32 = 123i16.into();
    println!("{s}, {addr}, {one}, {bigger}");
}
```

▼ Details

- That's why it is common to only implement `From`, as your type will get `Into` implementation too.
- When declaring a function argument input type like "anything that can be converted into a `String`", the rule is opposite, you should use `Into`. Your function will accept types that implement `From` and those that *only* implement `Into`.

Read and Write

Using `Read` and `BufRead`, you can abstract over `u8` sources:

```
use std::io::{BufRead, BufReader, Read, Result};

fn count_lines<R: Read>(reader: R) -> usize {
    let buf_reader = BufReader::new(reader);
    buf_reader.lines().count()
}

fn main() -> Result<()> {
    let slice: &[u8] = b"foo\nbar\nbaz\n";
    println!("lines in slice: {}", count_lines(slice));

    let file = std::fs::File::open(std::env::current_exe()?)?;
    println!("lines in file: {}", count_lines(file));
    Ok(())
}
```

Similarly, `Write` lets you abstract over `u8` sinks:

```
use std::io::{Result, Write};

fn log<W: Write>(writer: &mut W, msg: &str) -> Result<()> {
    writer.write_all(msg.as_bytes())?;
    writer.write_all("\n".as_bytes())
}

fn main() -> Result<()> {
    let mut buffer = Vec::new();
    log(&mut buffer, "Hello")?;
    log(&mut buffer, "World")?;
    println!("Logged: {:?}", buffer);
    Ok(())
}
```

`Add`, `Mul`, ...

Operator overloading is implemented via traits in `std::ops`:

```
#[derive(Debug, Copy, Clone)]
struct Point { x: i32, y: i32 }

impl std::ops::Add for Point {
    type Output = Self;

    fn add(self, other: Self) -> Self {
        Self {x: self.x + other.x, y: self.y + other.y}
    }
}

fn main() {
    let p1 = Point { x: 10, y: 20 };
    let p2 = Point { x: 100, y: 200 };
    println!("{:?} + {:?} = {:?}", p1, p2, p1 + p2);
}
```

▼ Details

Discussion points:

- You could implement `Add` for `&Point`. In which situations is that useful?
  - Answer: `Add:add` consumes `self`. If type `T` for which you are overloading the operator is not `Copy`, you should consider overloading the operator for `&T` as well. This avoids unnecessary cloning on the call site.
- Why is `Output` an associated type? Could it be made a type parameter?
  - Short answer: Type parameters are controlled by the caller, but associated types (like `Output`) are controlled by the implementor of a trait.

The Drop Trait

Values which implement `Drop` can specify code to run when they go out of scope:

```
struct Droppable {
    name: &'static str,
}

impl Drop for Droppable {
    fn drop(&mut self) {
        println!("Dropping {}", self.name);
    }
}

fn main() {
    let a = Droppable { name: "a" };
    {
        let b = Droppable { name: "b" };
        {
            let c = Droppable { name: "c" };
            let d = Droppable { name: "d" };
            println!("Exiting block B");
        }
        println!("Exiting block A");
    }
    drop(a);
    println!("Exiting main");
}
```

▼ Details

Discussion points:

- Why does not `Drop::drop` take `self`?
  - Short-answer: If it did, `std::mem::drop` would be called at the end of the block, resulting in another call to `Drop::drop`, and a stack overflow!
- Try replacing `drop(a)` with `a.drop()`.

The `Default` Trait

`Default` trait provides a default implementation of a trait.

```
#[derive(Debug, Default)]
struct Derived {
    x: u32,
    y: String,
    z: Implemented,
}

#[derive(Debug)]
struct Implemented(String);

impl Default for Implemented {
    fn default() -> Self {
        Self("John Smith".into())
    }
}

fn main() {
    let default_struct: Derived = Default::default();
    println!("{default_struct:#?}");

    let almost_default_struct = Derived {
        y: "Y is set!".into(),
        ..Default::default()
    };
    println!("{almost_default_struct:#?}");

    let nothing: Option<Derived> = None;
    println!("{:#?}", nothing.unwrap_or_default());
}
```

▼ Details

- It can be implemented directly or it can be derived via `#[derive(Default)]`.
- Derived implementation will produce an instance where all fields are set to their default values.
  - This means all types in the struct must implement `Default` too.
- Standard Rust types often implement `Default` with reasonable values (e.g. `0`, `""`, etc).
- The partial struct copy works nicely with default.
- Rust standard library is aware that types can implement `Default` and provides convenience methods that use it.

Generics

Rust support generics, which lets you abstract an algorithm (such as sorting) over the types used in the algorithm.

Generic Data Types

You can use generics to abstract over the concrete field type:

```
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
    println!("{integer:?} and {float:?}");
}
```

Generic Methods

You can declare a generic type on your `impl` block:

```
#[derive(Debug)]
struct Point<T>(T, T);

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.0  // + 10
    }

    // fn set_x(&mut self, x: T)
}

fn main() {
    let p = Point(5, 10);
    println!("p.x = {}", p.x());
}
```

▼ Details

- *Q:* Why `T` is specified twice in `impl<T> Point<T> {}`? Isn't that redundant?
  - This is because it is a generic implementation section for generic type. They are independently generic.
  - It means these methods are defined for any `T`.
  - It is possible to write `impl Point<u32> { .. }`.
    - `Point` is still generic and you can use `Point<f64>`, but methods in this block will only be available for `Point<u32>`.

Trait Bounds

When working with generics, you often want to limit the types. You can do this with `T: Trait` or `impl Trait`:

```rust
fn duplicate<T: Clone>(a: T) -> (T, T) {
    (a.clone(), a.clone())
}

// struct NotClonable;

fn main() {
    let foo = String::from("foo");
    let pair = duplicate(foo);
    println!("{pair:?}");
}
```

▼ Details

Show a `where` clause, students will encounter it when reading code.

```rust
fn duplicate<T>(a: T) -> (T, T)
where
    T: Clone,
{
    (a.clone(), a.clone())
}
```

- It declutters the function signature if you have many parameters.
- It has additional features making it more powerful.
    - If someone asks, the extra feature is that the type on the left of ":" can be arbitrary, like `Option<T>`.

`impl Trait`

Similar to trait bounds, an `impl Trait` syntax can be used in function arguments and return values:

```rust
use std::fmt::Display;

fn get_x(name: impl Display) -> impl Display {
    format!("Hello {name}")
}

fn main() {
    let x = get_x("foo");
    println!("{x}");
}
```

- `impl Trait` cannot be used with the `::<>` turbo fish syntax.
- `impl Trait` allows you to work with types which you cannot name.

▼ Details

The meaning of `impl Trait` is a bit different in the different positions.

- For a parameter, `impl Trait` is like an anonymous generic parameter with a trait bound.
- For a return type, it means that the return type is some concrete type that implements the trait, without naming the type. This can be useful when you don't want to expose the concrete type in a public API.

This example is great, because it uses `impl Display` twice. It helps to explain that nothing here enforces that it is *the same* `impl Display` type. If we used a single `T: Display`, it would enforce the constraint that input `T` and return `T` type are the same type. It would not work for this particular function, as the type we expect as input is likely not what `format!` returns. If we wanted to do the same via `: Display` syntax, we'd need two independent generic parameters.

Closures

Closures or lambda expressions have types which cannot be named. However, they implement special `Fn`, `FnMut`, and `FnOnce` traits:

```
fn apply_with_log(func: impl FnOnce(i32) -> i32, input: i32) -> i32 {
    println!("Calling function on {input}");
    func(input)
}

fn main() {
    let add_3 = |x| x + 3;
    let mul_5 = |x| x * 5;

    println!("add_3: {}", apply_with_log(add_3, 10));
    println!("mul_5: {}", apply_with_log(mul_5, 20));
}
```

▼ Details

If you have an `FnOnce`, you may only call it once. It might consume captured values.

An `FnMut` might mutate captured values, so you can call it multiple times but not concurrently.

An `Fn` neither consumes nor mutates captured values, or perhaps captures nothing at all, so it can be called multiple times concurrently.

`FnMut` is a subtype of `FnOnce`. `Fn` is a subtype of `FnMut` and `FnOnce`. I.e. you can use an `FnMut` wherever an `FnOnce` is called for, and you can use an `Fn` wherever an `FnMut` or `FnOnce` is called for.

`move` closures only implement `FnOnce`.

Monomorphization

Generic code is turned into non-generic code based on the call sites:

```rust
fn main() {
    let integer = Some(5);
    let float = Some(5.0);
}
```

behaves as if you wrote

```rust
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

This is a zero-cost abstraction: you get exactly the same result as if you had hand-coded the data structures without the abstraction.

Trait Objects

We've seen how a function can take arguments which implement a trait:

```
use std::fmt::Display;

fn print<T: Display>(x: T) {
    println!("Your value: {x}");
}

fn main() {
    print(123);
    print("Hello");
}
```

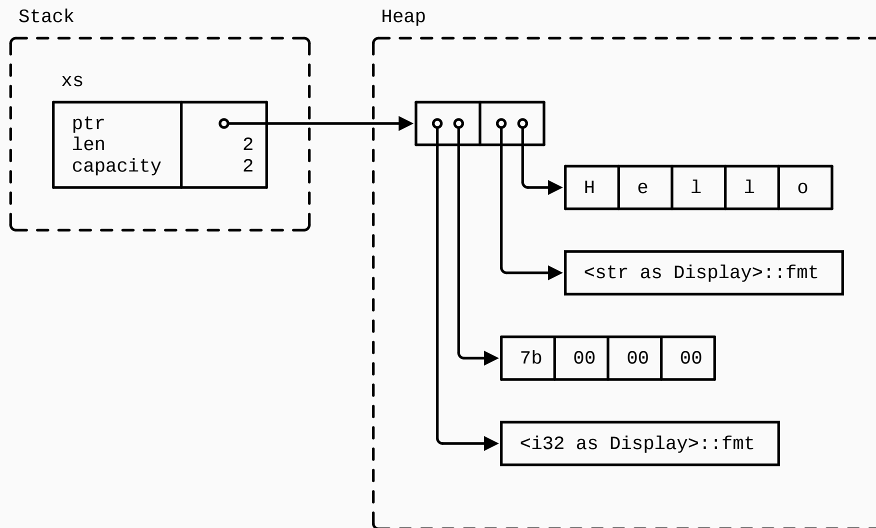
However, how can we store a collection of mixed types which implement `Display`?

```
fn main() {
    let xs = vec![123, "Hello"];
}
```

For this, we need *trait objects*:

```
use std::fmt::Display;

fn main() {
    let xs: Vec<Box<dyn Display>> = vec![Box::new(123), Box::new("Hello")];
    for x in xs {
        println!("x: {x}");
    }
}
```

Memory layout after allocating `xs`:



Similarly, you need a trait object if you want to return different values implementing a trait:

```
fn numbers(n: i32) -> Box<dyn Iterator<Item=i32>> {
    if n > 0 {
        Box::new(0..n)
    } else {
        Box::new((n..0).rev())
    }
}

fn main() {
    println!("{:?}", numbers(-5).collect::<Vec<_>>());
    println!("{:?}", numbers(5).collect::<Vec<_>>());
}
```

Day 3: Morning Exercises

We will design a classical GUI library traits and trait objects.

▼ Details

After looking at the exercises, you can look at the solutions provided.

A Simple GUI Library

Let us design a classical GUI library using our new knowledge of traits and trait objects.

We will have a number of widgets in our library:

- `Window` : has a `title` and contains other widgets.
- `Button` : has a `label` and a callback function which is invoked when the button is pressed.
- `Label` : has a `label` .

The widgets will implement a `Widget` trait, see below.

Copy the code below to https://play.rust-lang.org/, fill in the missing `draw_into` methods so that you implement the `Widget` trait:

```rust
// TODO: remove this when you're done with your implementation.
#![allow(unused_imports, unused_variables, dead_code)]

pub trait Widget {
    /// Natural width of `self`.
    fn width(&self) -> usize;

    /// Draw the widget into a buffer.
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write);

    /// Draw the widget on standard output.
    fn draw(&self) {
        let mut buffer = String::new();
        self.draw_into(&mut buffer);
        println!("{buffer}");
    }
}

pub struct Label {
    label: String,
}

impl Label {
    fn new(label: &str) -> Label {
        Label {
            label: label.to_owned(),
        }
    }
}

pub struct Button {
    label: Label,
    callback: Box<dyn FnMut()>,
}

impl Button {
    fn new(label: &str, callback: Box<dyn FnMut()>) -> Button {
        Button {
            label: Label::new(label),
            callback,
        }
    }
}

pub struct Window {
    title: String,
    widgets: Vec<Box<dyn Widget>>,
}

impl Window {
    fn new(title: &str) -> Window {
        Window {
            title: title.to_owned(),
            widgets: Vec::new(),
        }
    }

    fn add_widget(&mut self, widget: Box<dyn Widget>) {
        self.widgets.push(widget);
    }
}


impl Widget for Label {
    fn width(&self) -> usize {
        unimplemented!()
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        unimplemented!()
    }
```

```
}

impl Widget for Button {
    fn width(&self) -> usize {
        unimplemented!()
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        unimplemented!()
    }
}

impl Widget for Window {
    fn width(&self) -> usize {
        unimplemented!()
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        unimplemented!()
    }
}

fn main() {
    let mut window = Window::new("Rust GUI Demo 1.23");
    window.add_widget(Box::new(Label::new("This is a small text GUI demo.")));
    window.add_widget(Box::new(Button::new(
        "Click me!",
        Box::new(|| println!("You clicked the button!")),
    )));
    window.draw();
}
```

The output of the above program can be something simple like this:

```
========
Rust GUI Demo 1.23
========

This is a small text GUI demo.

| Click me! |
```

If you want to draw aligned text, you can use the fill/alignment formatting operators. In particular, notice how you can pad with different characters (here a `'/'`) and how you can control alignment:

```
fn main() {
    let width = 10;
    println!("left aligned:  |{:/<width$}|", "foo");
    println!("centered:      |{:/^width$}|", "foo");
    println!("right aligned: |{:/>width$}|", "foo");
}
```

Using such alignment tricks, you can for example produce output like this:

```
+--------------------------------+
|       Rust GUI Demo 1.23       |
+================================+
| This is a small text GUI demo. |
| +-----------+                  |
| | Click me! |                  |
| +-----------+                  |
+--------------------------------+
```

Error Handling

Error handling in Rust is done using explicit control flow:

- Functions that can have errors list this in their return type,
- There are no exceptions.

Panics

Rust will trigger a panic if a fatal error happens at runtime:

```
fn main() {
    let v = vec![10, 20, 30];
    println!("v[100]: {}", v[100]);
}
```

- Panics are for unrecoverable and unexpected errors.
  - Panics are symptoms of bugs in the program.
- Use non-panicking APIs (such as `Vec::get`) if crashing is not acceptable.

Catching the Stack Unwinding

By default, a panic will cause the stack to unwind. The unwinding can be caught:

```
use std::panic;

let result = panic::catch_unwind(|| {
    println!("hello!");
});
assert!(result.is_ok());

let result = panic::catch_unwind(|| {
    panic!("oh no!");
});
assert!(result.is_err());
```

- This can be useful in servers which should keep running even if a single request crashes.
- This does not work if `panic = 'abort'` is set in your `Cargo.toml`.

Structured Error Handling with `Result`

We have already seen the `Result` enum. This is used pervasively when errors are expected as part of normal operation:

```
use std::fs::File;
use std::io::Read;

fn main() {
    let file = File::open("diary.txt");
    match file {
        Ok(mut file) => {
            let mut contents = String::new();
            file.read_to_string(&mut contents);
            println!("Dear diary: {contents}");
        },
        Err(err) => {
            println!("The diary could not be opened: {err}");
        }
    }
}
```

▼ Details

- As with `Option`, the successful value sits inside of `Result`, forcing the developer to explicitly extract it. This encourages error checking. In the case where an error should never happen, `unwrap()` or `expect()` can be called, and this is a signal of the developer intent too.
- `Result` documentation is a recommended read. Not during the course, but it is worth mentioning. It contains a lot of convenience methods and functions that help functional-style programming.

Propagating Errors with `?`

The try-operator `?` is used to return errors to the caller. It lets you turn the common

```
match some_expression {
    Ok(value) => value,
    Err(err) => return Err(err),
}
```

into the much simpler

```
some_expression?
```

We can use this to simplify our error handing code:

```
use std::fs;
use std::io::{self, Read};

fn read_username(path: &str) -> Result<String, io::Error> {
    let username_file_result = fs::File::open(path);

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}

fn main() {
    //fs::write("config.dat", "alice").unwrap();
    let username = read_username("config.dat");
    println!("username or error: {username:?}");
}
```

▼ Details

Key points:

- The `username` variable can be either `Ok(string)` or `Err(error)`.
- Use the `fs::write` call to test out the different scenarios: no file, empty file, file with username.

Converting Error Types

The effective expansion of `?` is a little more complicated than previously indicated:

```
expression?
```

works the same as

```
match expression {
    Ok(value) => value,
    Err(err)  => return Err(From::from(err)),
}
```

The `From::from` call here means we attempt to convert the error type to the type returned by the function:

Converting Error Types

```
use std::error::Error;
use std::fmt::{self, Display, Formatter};
use std::fs::{self, File};
use std::io::{self, Read};

#[derive(Debug)]
enum ReadUsernameError {
    IoError(io::Error),
    EmptyUsername(String),
}

impl Error for ReadUsernameError {}

impl Display for ReadUsernameError {
    fn fmt(&self, f: &mut Formatter) -> fmt::Result {
        match self {
            Self::IoError(e) => write!(f, "IO error: {}", e),
            Self::EmptyUsername(filename) => write!(f, "Found no username in {}", filename),
        }
    }
}

impl From<io::Error> for ReadUsernameError {
    fn from(err: io::Error) -> ReadUsernameError {
        ReadUsernameError::IoError(err)
    }
}

fn read_username(path: &str) -> Result<String, ReadUsernameError> {
    let mut username = String::with_capacity(100);
    File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(ReadUsernameError::EmptyUsername(String::from(path)));
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    let username = read_username("config.dat");
    println!("username or error: {username:?}");
}
```

▼ Details

Key points:

- The `username` variable can be either `Ok(string)` or `Err(error)`.
- Use the `fs::write` call to test out the different scenarios: no file, empty file, file with username.

It is good practice for all error types to implement `std::error::Error`, which requires `Debug` and `Display`. It's generally helpful for them to implement `Clone` and `Eq` too where possible, to make life easier for tests and consumers of your library. In this case we can't easily do so, because `io::Error` doesn't implement them.

Deriving Error Enums

The thiserror crate is a popular way to create an error enum like we did on the previous page:

```
use std::{fs, io};
use std::io::Read;
use thiserror::Error;

#[derive(Debug, Error)]
enum ReadUsernameError {
    #[error("Could not read: {0}")]
    IoError(#[from] io::Error),
    #[error("Found no username in {0}")]
    EmptyUsername(String),
}

fn read_username(path: &str) -> Result<String, ReadUsernameError> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(ReadUsernameError::EmptyUsername(String::from(path)));
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("Username: {username}"),
        Err(err)     => println!("Error: {err}"),
    }
}
```

▼ Details

`thiserror`'s derive macro automatically implements `std::error::Error`, and optionally `Display` (if the `#[error(...)]` attributes are provided) and `From` (if the `#[from]` attribute is added). It also works for structs.

It doesn't affect your public API, which makes it good for libraries.

Dynamic Error Types

Sometimes we want to allow any type of error to be returned without writing our own enum covering all the different possibilities. `std::error::Error` makes this easy.

```
use std::fs::{self, File};
use std::io::Read;
use thiserror::Error;
use std::error::Error;

#[derive(Clone, Debug, Eq, Error, PartialEq)]
#[error("Found no username in {0}")]
struct EmptyUsernameError(String);

fn read_username(path: &str) -> Result<String, Box<dyn Error>> {
    let mut username = String::with_capacity(100);
    File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(EmptyUsernameError(String::from(path)).into());
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("Username: {username}"),
        Err(err)     => println!("Error: {err}"),
    }
}
```

▼ Details

This saves on code, but gives up the ability to cleanly handle different error cases differently in the program. As such it's generally not a good idea to use `Box<dyn Error>` in the public API of a library, but it can be a good option in a program where you just want to display the error message somewhere.

Adding Context to Errors

The widely used anyhow crate can help you add contextual information to your errors and allows you to have fewer custom error types:

```
use std::{fs, io};
use std::io::Read;
use anyhow::{Context, Result, bail};

fn read_username(path: &str) -> Result<String> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)
        .context(format!("Failed to open {path}"))?
        .read_to_string(&mut username)
        .context("Failed to read")?;
    if username.is_empty() {
        bail!("Found no username in {path}");
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("Username: {username}"),
        Err(err)     => println!("Error: {err:?}"),
    }
}
```

▼ Details

- `anyhow::Result<V>` is a type alias for `Result<V, anyhow::Error>`.
- `anyhow::Error` is essentially a wrapper around `Box<dyn Error>`. As such it's again generally not a good choice for the public API of a library, but is widely used in applications.
- Actual error type inside of it can be extracted for examination if necessary.
- Functionality provided by `anyhow::Result<T>` may be familiar to Go developers, as it provides similar usage patterns and ergonomics to `(T, error)` from Go.

Testing

Rust and Cargo come with a simple unit test framework:

- Unit tests are supported throughout your code.
- Integration tests are supported via the `tests/` directory.

Unit Tests

Mark unit tests with `#[test]`:

```
fn first_word(text: &str) -> &str {
    match text.find(' ') {
        Some(idx) => &text[..idx],
        None => &text,
    }
}

#[test]
fn test_empty() {
    assert_eq!(first_word(""), "");
}

#[test]
fn test_single_word() {
    assert_eq!(first_word("Hello"), "Hello");
}

#[test]
fn test_multiple_words() {
    assert_eq!(first_word("Hello World"), "Hello");
}
```

Use `cargo test` to find and run the unit tests.

Test Modules

Unit tests are often put in a nested module (run tests on the Playground):

```
fn helper(a: &str, b: &str) -> String {
    format!("{a} {b}")
}

pub fn main() {
    println!("{}", helper("Hello", "World"));
}

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

    #[test]
    fn test_helper() {
        assert_eq!(helper("foo", "bar"), "foo bar");
    }
}
```

- This lets you unit test private helpers.
- The `#[cfg(test)]` attribute is only active when you run `cargo test`.

Documentation Tests

Rust has built-in support for documentation tests:

```
/// Shortens a string to the given length.
///
/// ```
/// use playground::shorten_string;
/// assert_eq!(shorten_string("Hello World", 5), "Hello");
/// assert_eq!(shorten_string("Hello World", 20), "Hello World");
/// ```
pub fn shorten_string(s: &str, length: usize) -> &str {
    &s[..std::cmp::min(length, s.len())]
}
```

- Code blocks in `///` comments are automatically seen as Rust code.
- The code will be compiled and executed as part of `cargo test`.
- Test the above code on the Rust Playground.

Integration Tests

If you want to test your library as a client, use an integration test.

Create a `.rs` file under `tests/`:

```rust
use my_library::init;

#[test]
fn test_init() {
    assert!(init().is_ok());
}
```

These tests only have access to the public API of your crate.

Unsafe Rust

The Rust language has two parts:

- **Safe Rust:** memory safe, no undefined behavior possible.
- **Unsafe Rust:** can trigger undefined behavior if preconditions are violated.

We will be seeing mostly safe Rust in this course, but it's important to know what Unsafe Rust is.

Unsafe code is usually small and isolated, and its correctness should be carefully documented. It is usually wrapped in a safe abstraction layer.

Unsafe Rust gives you access to five new capabilities:

- Dereference raw pointers.
- Access or modify mutable static variables.
- Access `union` fields.
- Call `unsafe` functions, including `extern` functions.
- Implement `unsafe` traits.

We will briefly cover unsafe capabilities next. For full details, please see Chapter 19.1 in the Rust Book and the Rustonomicon.

▼ Details

Unsafe Rust does not mean the code is incorrect. It means that developers have turned off the compiler safety features and have to write correct code by themselves. It means the compiler no longer enforces Rust's memory-safety rules.

Dereferencing Raw Pointers

Creating pointers is safe, but dereferencing them requires `unsafe`:

```
fn main() {
    let mut num = 5;

    let r1 = &mut num as *mut i32;
    let r2 = &num as *const i32;

    // Safe because r1 and r2 were obtained from references and so are guaranteed to be non-
    // properly aligned, the objects underlying the references from which they were obtained
    // live throughout the whole unsafe block, and they are not accessed either through the
    // references or concurrently through any other pointers.
    unsafe {
        println!("r1 is: {}", *r1);
        *r1 = 10;
        println!("r2 is: {}", *r2);
    }
}
```

▼ Details

It is good practice (and required by the Android Rust style guide) to write a comment for each `unsafe` block explaining how the code inside it satisfies the safety requirements of the unsafe operations it is doing.

In the case of pointer dereferences, this means that the pointers must be *valid*, i.e.:

- The pointer must be non-null.
- The pointer must be *dereferenceable* (within the bounds of a single allocated object).
- The object must not have been deallocated.
- There must not be concurrent accesses to the same location.
- If the pointer was obtained by casting a reference, the underlying object must be live and no reference may be used to access the memory.

In most cases the pointer must also be properly aligned.

Mutable Static Variables

It is safe to read an immutable static variable:

```rust
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("HELLO_WORLD: {HELLO_WORLD}");
}
```

However, since data races can occur, it is unsafe to read and write mutable static variables:

```rust
static mut COUNTER: u32 = 0;

fn add_to_counter(inc: u32) {
    unsafe { COUNTER += inc; }  // Potential data race!
}

fn main() {
    add_to_counter(42);

    unsafe { println!("COUNTER: {COUNTER}"); }  // Potential data race!
}
```

▼ Details

Using a mutable static is generally a bad idea, but there are some cases where it might make sense in low-level `no_std` code, such as implementing a heap allocator or working with some C APIs.

Unions

Unions are like enums, but you need to track the active field yourself:

```rust
#[repr(C)]
union MyUnion {
    i: u8,
    b: bool,
}

fn main() {
    let u = MyUnion { i: 42 };
    println!("int: {}", unsafe { u.i });
    println!("bool: {}", unsafe { u.b });  // Undefined behavior!
}
```

▼ Details

Unions are very rarely needed in Rust as you can usually use an enum. They are occasionally needed for interacting with C library APIs.

If you just want to reinterpret bytes as a different type, you probably want `std::mem::transmute` or a safe wrapper such as the `zerocopy` crate.

Calling Unsafe Functions

A function or method can be marked `unsafe` if it has extra preconditions you must uphold to avoid undefined behaviour:

```
fn main() {
    let emojis = "🗻∈🌏";

    // Safe because the indices are in the correct order, within the bounds of
    // the string slice, and lie on UTF-8 sequence boundaries.
    unsafe {
        println!("{}", emojis.get_unchecked(0..4));
        println!("{}", emojis.get_unchecked(4..7));
        println!("{}", emojis.get_unchecked(7..11));
    }
}
```

Writing Unsafe Functions

You can mark your own functions as `unsafe` if they require particular conditions to avoid undefined behaviour.

```rust
/// Swaps the values pointed to by the given pointers.
///
/// # Safety
///
/// The pointers must be valid and properly aligned.
unsafe fn swap(a: *mut u8, b: *mut u8) {
    let temp = *a;
    *a = *b;
    *b = temp;
}

fn main() {
    let mut a = 42;
    let mut b = 66;

    // Safe because ...
    unsafe {
        swap(&mut a, &mut b);
    }

    println!("a = {}, b = {}", a, b);
}
```

▼ Details

We wouldn't actually use pointers for this because it can be done safely with references.

Note that unsafe code is allowed within an unsafe function without an `unsafe` block. We can prohibit this with `#[deny(unsafe_op_in_unsafe_fn)]`. Try adding it and see what happens.

Calling External Code

Functions from other languages might violate the guarantees of Rust. Calling them is thus unsafe:

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        // Undefined behavior if abs misbehaves.
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

▼ Details

This is usually only a problem for extern functions which do things with pointers which might violate Rust's memory model, but in general any C function might have undefined behaviour under any arbitrary circumstances.

The `"C"` in this example is the ABI; other ABIs are available too.

Implementing Unsafe Traits

Like with functions, you can mark a trait as `unsafe` if the implementation must guarantee particular conditions to avoid undefined behaviour.

For example, the `zerocopy` crate has an unsafe trait that looks something like this:

```
use std::mem::size_of_val;
use std::slice;

/// ...
/// # Safety
/// The type must have a defined representation and no padding.
pub unsafe trait AsBytes {
    fn as_bytes(&self) -> &[u8] {
        unsafe {
            slice::from_raw_parts(self as *const Self as *const u8, size_of_val(self))
        }
    }
}

// Safe because u32 has a defined representation and no padding.
unsafe impl AsBytes for u32 {}
```

▼ Details

There should be a `# Safety` section on the Rustdoc for the trait explaining the requirements for the trait to be safely implemented.

The actual safety section for `AsBytes` is rather longer and more complicated.

The built-in `Send` and `Sync` traits are unsafe.

Day 3: Afternoon Exercises

Let us build a safe wrapper for reading directory content!

▼ Details

After looking at the exercise, you can look at the solution provided.

Safe FFI Wrapper

Rust has great support for calling functions through a *foreign function interface* (FFI). We will use this to build a safe wrapper for the `libc` functions you would use from C to read the filenames of a directory.

You will want to consult the manual pages:

- `opendir(3)`
- `readdir(3)`
- `closedir(3)`

You will also want to browse the `std::ffi` module, particular for `CStr` and `CString` types which are used to hold NUL-terminated strings coming from C. The Nomicon also has a very useful chapter about FFI.

Copy the code below to https://play.rust-lang.org/ and fill in the missing functions and methods:

```rust
// TODO: remove this when you're done with your implementation.
#![allow(unused_imports, unused_variables, dead_code)]

mod ffi {
    use std::os::raw::{c_char, c_int, c_long, c_ulong, c_ushort};

    // Opaque type. See https://doc.rust-lang.org/nomicon/ffi.html.
    #[repr(C)]
    pub struct DIR {
        _data: [u8; 0],
        _marker: core::marker::PhantomData<(*mut u8, core::marker::PhantomPinned)>,
    }

    // Layout as per readdir(3) and definitions in /usr/include/x86_64-linux-gnu.
    #[repr(C)]
    pub struct dirent {
        pub d_ino: c_long,
        pub d_off: c_ulong,
        pub d_reclen: c_ushort,
        pub d_type: c_char,
        pub d_name: [c_char; 256],
    }

    extern "C" {
        pub fn opendir(s: *const c_char) -> *mut DIR;
        pub fn readdir(s: *mut DIR) -> *const dirent;
        pub fn closedir(s: *mut DIR) -> c_int;
    }
}

use std::ffi::{CStr, CString, OsStr, OsString};
use std::os::unix::ffi::OsStrExt;

#[derive(Debug)]
struct DirectoryIterator {
    path: CString,
    dir: *mut ffi::DIR,
}

impl DirectoryIterator {
    fn new(path: &str) -> Result<DirectoryIterator, String> {
        // Call opendir and return a Ok value if that worked,
        // otherwise return Err with a message.
        unimplemented!()
    }
}

impl Iterator for DirectoryIterator {
    type Item = OsString;
    fn next(&mut self) -> Option<OsString> {
        // Keep calling readdir until we get a NULL pointer back.
        unimplemented!()
    }
}

impl Drop for DirectoryIterator {
    fn drop(&mut self) {
        // Call closedir as needed.
        unimplemented!()
    }
}

fn main() -> Result<(), String> {
    let iter = DirectoryIterator::new(".")?;
    println!("files: {:#?}", iter.collect::<Vec<_>>());
    Ok(())
}
```

Welcome to Day 4

Today we will look at two main topics:

- Concurrency: threads, channels, shared state, `Send` and `Sync`.
- Android: building binaries and libraries, using AIDL, logging, and interoperability with C, C++, and Java.

We will attempt to call Rust from one of your own projects today. So try to find a little corner of your code base where we can move some lines of code to Rust. The fewer dependencies and "exotic" types the better. Something that parses some raw bytes would be ideal.

Fearless Concurrency

Rust has full support for concurrency using OS threads with mutexes and channels.

The Rust type system plays an important role in making many concurrency bugs compile time bugs. This is often referred to as *fearless concurrency* since you can rely on the compiler to ensure correctness at runtime.

Threads

Rust threads work similarly to threads in other languages:

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

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("Count in thread: {i}!");
            thread::sleep(Duration::from_millis(5));
        }
    });

    for i in 1..5 {
        println!("Main thread: {i}");
        thread::sleep(Duration::from_millis(5));
    }
}
```

- Threads are all daemon threads, the main thread does not wait for them.
- Thread panics are independent of each other.
  - Panics can carry a payload, which can be unpacked with `downcast_ref`.

▼ Details

Key points:

- Notice that the thread is stopped before it reaches 10 — the main thread is not waiting.
- Use `let handle = thread::spawn(...)` and later `handle.join()` to wait for the thread to finish.
- Trigger a panic in the thread, notice how this doesn't affect `main`.
- Use the `Result` return value from `handle.join()` to get access to the panic payload. This is a good time to talk about `Any`.

Scoped Threads

Normal threads cannot borrow from their environment:

```
use std::thread;

fn main() {
    let s = String::from("Hello");

    thread::spawn(|| {
        println!("Length: {}", s.len());
    });
}
```

However, you can use a scoped thread for this:

```
use std::thread;

fn main() {
    let s = String::from("Hello");

    thread::scope(|scope| {
        scope.spawn(|| {
            println!("Length: {}", s.len());
        });
    });
}
```

▼ Details

- The reason for that is that when the `thread::scope` function completes, all the threads are guaranteed to be joined, so they can return borrowed data.
- Normal Rust borrowing rules apply: you can either borrow mutably by one thread, or immutably by any number of threads.

Channels

Rust channels have two parts: a `Sender<T>` and a `Receiver<T>`. The two parts are connected via the channel, but you only see the end-points.

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    tx.send(10).unwrap();
    tx.send(20).unwrap();

    println!("Received: {:?}", rx.recv());
    println!("Received: {:?}", rx.recv());

    let tx2 = tx.clone();
    tx2.send(30).unwrap();
    println!("Received: {:?}", rx.recv());
}
```

▼ Details

- `mpsc` stands for Multi-Producer, Single-Consumer. `Sender` and `SyncSender` implement `Clone` (so you can make multiple producers) but `Receiver` does not.
- `send()` and `recv()` return `Result`. If they return `Err`, it means the counterpart `Sender` or `Receiver` is dropped and the channel is closed.

Unbounded Channels

You get an unbounded and asynchronous channel with `mpsc::channel()`:

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

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 1..10 {
            tx.send(format!("Message {i}")).unwrap();
            println!("{thread_id:?}: sent Message {i}");
        }
        println!("{thread_id:?}: done");
    });
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("Main: got {}", msg);
    }
}
```

Bounded Channels

Bounded and synchronous channels make `send` block the current thread:

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

fn main() {
    let (tx, rx) = mpsc::sync_channel(3);

    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 1..10 {
            tx.send(format!("Message {i}")).unwrap();
            println!("{thread_id:?}: sent Message {i}");
        }
        println!("{thread_id:?}: done");
    });
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("Main: got {msg}");
    }
}
```

Shared State

Rust uses the type system to enforce synchronization of shared data. This is primarily done via two types:

- `Arc<T>`, atomic reference counted `T`: handles sharing between threads and takes care to deallocate `T` when the last reference is dropped,
- `Mutex<T>`: ensures mutually exclusive access to the `T` value.

Arc

`Arc<T>` allows shared read-only access via its `clone` method:

```
use std::thread;
use std::sync::Arc;

fn main() {
    let v = Arc::new(vec![10, 20, 30]);
    let mut handles = Vec::new();
    for _ in 1..5 {
        let v = v.clone();
        handles.push(thread::spawn(move || {
            let thread_id = thread::current().id();
            println!("{thread_id:?}: {v:?}");
        }));
    }

    handles.into_iter().for_each(|h| h.join().unwrap());
    println!("v: {v:?}");
}
```

▼ Details

- `Arc` stands for "Atomic Reference Counted", a thread safe version of `Rc` that uses atomic operations.
- `Arc<T>` implements `Clone` whether or not `T` does. It implements `Send` and `Sync` iff `T` implements them both.
- `Arc::clone()` has the cost of atomic operations that get executed, but after that the use of the `T` is free.
- Beware of reference cycles, `Arc` does not use a garbage collector to detect them.
    - `std::sync::Weak` can help.

Mutex

`Mutex<T>` ensures mutual exclusion *and* allows mutable access to `T` behind a read-only interface:

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

fn main() {
    let v: Mutex<Vec<i32>> = Mutex::new(vec![10, 20, 30]);
    println!("v: {:?}", v.lock().unwrap());

    {
        let v: &Mutex<Vec<i32>> = &v;
        let mut guard = v.lock().unwrap();
        guard.push(40);
    }

    println!("v: {:?}", v.lock().unwrap());
}
```

Notice how we have a `impl<T: Send> Sync for Mutex<T>` blanket implementation.

▼ Details

- `Mutex` in Rust looks like a collection with just one element - the protected data.
  - It is not possible to forget to acquire the mutex before accessing the protected data.
- You can get an `&mut T` from an `&Mutex<T>` by taking the lock. The `MutexGuard` ensures that the `&mut T` doesn't outlive the lock being held.
- `Mutex<T>` implements both `Send` and `Sync` iff `T` implements `Send`.
- A read-write lock counterpart - `RwLock`.
- Why does `lock()` return a `Result`?
  - If the thread that held the `Mutex` panicked, the `Mutex` becomes "poisoned" to signal that the data it protected might be in an inconsistent state. Calling `lock()` on a poisoned mutex fails with a `PoisonError`. You can call `into_inner()` on the error to recover the data regardless.

Example

Let us see `Arc` and `Mutex` in action:

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

fn main() {
    let mut v = vec![10, 20, 30];
    let handle = thread::spawn(|| {
        v.push(10);
    });
    v.push(1000);

    handle.join().unwrap();
    println!("v: {v:?}");
}
```

▼ Details

Possible solution:

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

fn main() {
    let v = Arc::new(Mutex::new(vec![10, 20, 30]));

    let v2 = v.clone();
    let handle = thread::spawn(move || {
        let mut v2 = v2.lock().unwrap();
        v2.push(10);
    });

    {
        let mut v = v.lock().unwrap();
        v.push(1000);
    }

    handle.join().unwrap();

    {
        let v = v.lock().unwrap();
        println!("v: {v:?}");
    }
}
```

Notable parts:

- `v` is wrapped in both `Arc` and `Mutex`, because their concerns are orthogonal.
  - Wrapping a `Mutex` in an `Arc` is a common pattern to share mutable state between threads.
- `v: Arc<_>` needs to be cloned as `v2` before it can be moved into another thread. Note `move` was added to the lambda signature.
- Blocks are introduced to narrow the scope of the `LockGuard` as much as possible.
- We still need to acquire the `Mutex` to print our `Vec`.

`Send` and `Sync`

How does Rust know to forbid shared access across thread? The answer is in two traits:

- `Send`: a type `T` is `Send` if it is safe to move a `T` across a thread boundary.
- `Sync`: a type `T` is `Sync` if it is safe to move a `&T` across a thread boundary.

`Send` and `Sync` are unsafe traits. The compiler will automatically derive them for your types as long as they only contain `Send` and `Sync` types. You can also implement them manually when you know it is valid.

▼ Details

- One can think of these traits as markers that the type has certain thread-safety properties.
- They can be used in the generic constraints as normal traits.

Send

> A type `T` is `Send` if it is safe to move a `T` value to another thread.

The effect of moving ownership to another thread is that *destructors* will run in that thread. So the question is when you can allocate a value in one thread and deallocate it in another.

Sync

> A type `T` is `Sync` if it is safe to access a `T` value from multiple threads at the same time.

More precisely, the definition is:

> `T` is `Sync` if and only if `&T` is `Send`

▼ Details

This statement is essentially a shorthand way of saying that if a type is thread-safe for shared use, it is also thread-safe to pass references of it across threads.

This is because if a type is Sync it means that it can be shared across multiple threads without the risk of data races or other synchronization issues, so it is safe to move it to another thread. A reference to the type is also safe to move to another thread, because the data it references can be accessed from any thread safely.

Examples

`Send + Sync`

Most types you come across are `Send + Sync`:

- `i8`, `f32`, `bool`, `char`, `&str`, …
- `(T1, T2)`, `[T; N]`, `&[T]`, `struct { x: T }`, …
- `String`, `Option<T>`, `Vec<T>`, `Box<T>`, …
- `Arc<T>`: Explicitly thread-safe via atomic reference count.
- `Mutex<T>`: Explicitly thread-safe via internal locking.
- `AtomicBool`, `AtomicU8`, …: Uses special atomic instructions.

The generic types are typically `Send + Sync` when the type parameters are `Send + Sync`.

`Send + !Sync`

These types can be moved to other threads, but they're not thread-safe. Typically because of interior mutability:

- `mpsc::Sender<T>`
- `mpsc::Receiver<T>`
- `Cell<T>`
- `RefCell<T>`

`!Send + Sync`

These types are thread-safe, but they cannot be moved to another thread:

- `MutexGuard<T>`: Uses OS level primitives which must be deallocated on the thread which created them.

`!Send + !Sync`

These types are not thread-safe and cannot be moved to other threads:

- `Rc<T>`: each `Rc<T>` has a reference to an `RcBox<T>`, which contains a non-atomic reference count.
- `*const T`, `*mut T`: Rust assumes raw pointers may have special concurrency considerations.

Exercises

Let us practice our new concurrency skills with

- Dining philosophers: a classic problem in concurrency.
- Multi-threaded link checker: a larger project where you'll use Cargo to download dependencies and then check links in parallel.

▼ Details

After looking at the exercises, you can look at the solutions provided.

Dining Philosophers

The dining philosophers problem is a classic problem in concurrency:

> Five philosophers dine together at the same table. Each philosopher has their own place at the table. There is a fork between each plate. The dish served is a kind of spaghetti which has to be eaten with two forks. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat their spaghetti when they have both a left and right fork. Thus two forks will only be available when their two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, they will put down both forks.

You will need a local Cargo installation for this exercise. Copy the code below to `src/main.rs` file, fill out the blanks, and test that `cargo run` does not deadlock:

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

struct Fork;

struct Philosopher {
    name: String,
    // left_fork: ...
    // right_fork: ...
    // thoughts: ...
}

impl Philosopher {
    fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .unwrap();
    }

    fn eat(&self) {
        // Pick up forks...
        println!("{} is eating...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "Plato", "Aristotle", "Thales", "Pythagoras"];

fn main() {
    // Create forks

    // Create philosophers

    // Make them think and eat

    // Output their thoughts
}
```

Multi-threaded Link Checker

Let us use our new knowledge to create a multi-threaded link checker. It should start at a webpage and check that links on the page are valid. It should recursively check other pages on the same domain and keep doing this until all pages have been validated.

For this, you will need an HTTP client such as `reqwest`. Create a new Cargo project and `reqwest` it as a dependency with:

```
$ cargo new link-checker
$ cd link-checker
$ cargo add --features blocking,rustls-tls reqwest
```

> If `cargo add` fails with `error: no such subcommand`, then please edit the `Cargo.toml` file by hand. Add the dependencies listed below.

You will also need a way to find links. We can use `scraper` for that:

```
$ cargo add scraper
```

Finally, we'll need some way of handling errors. We use `thiserror` for that:

```
$ cargo add thiserror
```

The `cargo add` calls will update the `Cargo.toml` file to look like this:

```
[dependencies]
reqwest = { version = "0.11.12", features = ["blocking", "rustls-tls"] }
scraper = "0.13.0"
thiserror = "1.0.37"
```

You can now download the start page. Try with a small site such as `https://www.google.org/`.

Your `src/main.rs` file should look something like this:

```rust
use reqwest::blocking::{get, Response};
use reqwest::Url;
use scraper::{Html, Selector};
use thiserror::Error;

#[derive(Error, Debug)]
enum Error {
    #[error("request error: {0}")]
    ReqwestError(#[from] reqwest::Error),
}

fn extract_links(response: Response) -> Result<Vec<Url>, Error> {
    let base_url = response.url().to_owned();
    let document = response.text()?;
    let html = Html::parse_document(&document);
    let selector = Selector::parse("a").unwrap();

    let mut valid_urls = Vec::new();
    for element in html.select(&selector) {
        if let Some(href) = element.value().attr("href") {
            match base_url.join(href) {
                Ok(url) => valid_urls.push(url),
                Err(err) => {
                    println!("On {base_url}: could not parse {href:?}: {err} (ignored)",);
                }
            }
        }
    }

    Ok(valid_urls)
}

fn main() {
    let start_url = Url::parse("https://www.google.org").unwrap();
    let response = get(start_url).unwrap();
    match extract_links(response) {
        Ok(links) => println!("Links: {links:#?}"),
        Err(err) => println!("Could not extract links: {err:#}"),
    }
}
```

Run the code in `src/main.rs` with

```shell
$ cargo run
```

Tasks

- Use threads to check the links in parallel: send the URLs to be checked to a channel and let a few threads check the URLs in parallel.
- Extend this to recursively extract links from all pages on the `www.google.org` domain. Put an upper limit of 100 pages or so so that you don't end up being blocked by the site.

Android

Rust is supported for native platform development on Android. This means that you can write new operating system services in Rust, as well as extending existing services.

Setup

We will be using an Android Virtual Device to test our code. Make sure you have access to one or create a new one with:

```
$ source build/envsetup.sh
$ lunch aosp_cf_x86_64_phone-userdebug
$ acloud create
```

Please see the Android Developer Codelab for details.

Build Rules

The Android build system (Soong) supports Rust via a number of modules:

Module Type	Description
`rust_binary`	Produces a Rust binary.
`rust_library`	Produces a Rust library, and provides both `rlib` and `dylib` variants.
`rust_ffi`	Produces a Rust C library usable by `cc` modules, and provides both static and shared variants.
`rust_proc_macro`	Produces a `proc-macro` Rust library. These are analogous to compiler plugins.
`rust_test`	Produces a Rust test binary that uses the standard Rust test harness.
`rust_fuzz`	Produces a Rust fuzz binary leveraging `libfuzzer`.
`rust_protobuf`	Generates source and produces a Rust library that provides an interface for a particular protobuf.
`rust_bindgen`	Generates source and produces a Rust library containing Rust bindings to C libraries.

We will look at `rust_binary` and `rust_library` next.

Rust Binaries

Let us start with a simple application. At the root of an AOSP checkout, create the following files:

hello_rust/Android.bp:

```
rust_binary {
    name: "hello_rust",
    crate_name: "hello_rust",
    srcs: ["src/main.rs"],
}
```

hello_rust/src/main.rs:

```
//! Rust demo.

/// Prints a greeting to standard output.
fn main() {
    println!("Hello from Rust!");
}
```

You can now build, push, and run the binary:

```
$ m hello_rust
$ adb push $ANDROID_PRODUCT_OUT/system/bin/hello_rust /data/local/tmp
$ adb shell /data/local/tmp/hello_rust
Hello from Rust!
```

Rust Libraries

You use `rust_library` to create a new Rust library for Android.

Here we declare a dependency on two libraries:

- `libgreeting`, which we define below,
- `libtextwrap`, which is a crate already vendored in `external/rust/crates/`.

hello_rust/Android.bp:

```
rust_binary {
    name: "hello_rust_with_dep",
    crate_name: "hello_rust_with_dep",
    srcs: ["src/main.rs"],
    rustlibs: [
        "libgreetings",
        "libtextwrap",
    ],
    prefer_rlib: true,
}

rust_library {
    name: "libgreetings",
    crate_name: "greetings",
    srcs: ["src/lib.rs"],
}
```

hello_rust/src/main.rs:

```
//! Rust demo.

use greetings::greeting;
use textwrap::fill;

/// Prints a greeting to standard output.
fn main() {
    println!("{}", fill(&greeting("Bob"), 24));
}
```

hello_rust/src/lib.rs:

```
//! Greeting library.

/// Greet `name`.
pub fn greeting(name: &str) -> String {
    format!("Hello {name}, it is very nice to meet you!")
}
```

You build, push, and run the binary like before:

```
$ m hello_rust_with_dep
$ adb push $ANDROID_PRODUCT_OUT/system/bin/hello_rust_with_dep /data/local/tmp
$ adb shell /data/local/tmp/hello_rust_with_dep
Hello Bob, it is very
nice to meet you!
```

AIDL

The Android Interface Definition Language (AIDL) is supported in Rust:

- Rust code can call existing AIDL servers,
- You can create new AIDL servers in Rust.

AIDL Interfaces

You declare the API of your service using an AIDL interface:

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
package com.example.birthdayservice;

/** Birthday service interface. */
interface IBirthdayService {
    /** Generate a Happy Birthday message. */
    String wishHappyBirthday(String name, int years);
}
```

birthday_service/aidl/Android.bp:

```
aidl_interface {
    name: "com.example.birthdayservice",
    srcs: ["com/example/birthdayservice/*.aidl"],
    unstable: true,
    backend: {
        rust: { // Rust is not enabled by default
            enabled: true,
        },
    },
}
```

Add `vendor_available: true` if your AIDL file is used by a binary in the vendor partition.

Service Implementation

We can now implement the AIDL service:

birthday_service/src/lib.rs:

```
//! Implementation of the `IBirthdayService` AIDL interface.
use
com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService::IBirth
dayService;
use com_example_birthdayservice::binder;

/// The `IBirthdayService` implementation.
pub struct BirthdayService;

impl binder::Interface for BirthdayService {}

impl IBirthdayService for BirthdayService {
    fn wishHappyBirthday(&self, name: &str, years: i32) -> binder::Result<String> {
        Ok(format!(
            "Happy Birthday {name}, congratulations with the {years} years!"
        ))
    }
}
```

birthday_service/Android.bp:

```
rust_library {
    name: "libbirthdayservice",
    srcs: ["src/lib.rs"],
    crate_name: "birthdayservice",
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbinder_rs",
    ],
}
```

AIDL Server

Finally, we can create a server which exposes the service:

birthday_service/src/server.rs:

```
//! Birthday service.
use birthdayservice::BirthdayService;
use
com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService::BnBirt
hdayService;
use com_example_birthdayservice::binder;

const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// Entry point for birthday service.
fn main() {
    let birthday_service = BirthdayService;
    let birthday_service_binder = BnBirthdayService::new_binder(
        birthday_service,
        binder::BinderFeatures::default(),
    );
    binder::add_service(SERVICE_IDENTIFIER, birthday_service_binder.as_binder())
        .expect("Failed to register service");
    binder::ProcessState::join_thread_pool()
}
```

birthday_service/Android.bp:

```
rust_binary {
    name: "birthday_server",
    crate_name: "birthday_server",
    srcs: ["src/server.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbinder_rs",
        "libbirthdayservice",
    ],
    prefer_rlib: true,
}
```

Deploy

We can now build, push, and start the service:

```
$ m birthday_server
$ adb push $ANDROID_PRODUCT_OUT/system/bin/birthday_server /data/local/tmp
$ adb shell /data/local/tmp/birthday_server
```

In another terminal, check that the service runs:

```
$ adb shell service check birthdayservice
Service birthdayservice: found
```

You can also call the service with `service call`:

```
$ $ adb shell service call birthdayservice 1 s16 Bob i32 24
Result: Parcel(
  0x00000000: 00000000 00000036 00610048 00700070 '....6...H.a.p.p.'
  0x00000010: 00200079 00690042 00740072 00640068 'y. .B.i.r.t.h.d.'
  0x00000020: 00790061 00420020 0062006f 0020002c 'a.y. .B.o.b.,. .'
  0x00000030: 006f0063 0067006e 00610072 00750074 'c.o.n.g.r.a.t.u.'
  0x00000040: 0061006c 00690074 006e006f 00200073 'l.a.t.i.o.n.s. .'
  0x00000050: 00690077 00680074 00740020 00650068 'w.i.t.h. .t.h.e.'
  0x00000060: 00320020 00200034 00650079 00720061 ' .2.4. .y.e.a.r.'
  0x00000070: 00210073 00000000                   's.!.....        ')
```

AIDL Client

Finally, we can create a Rust client for our new service.

birthday_service/src/client.rs:

```
//! Birthday service.
use
com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService::IBirth
dayService;
use com_example_birthdayservice::binder;

const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// Connect to the BirthdayService.
pub fn connect() -> Result<binder::Strong<dyn IBirthdayService>, binder::StatusCode> {
    binder::get_interface(SERVICE_IDENTIFIER)
}

/// Call the birthday service.
fn main() -> Result<(), binder::Status> {
    let name = std::env::args()
        .nth(1)
        .unwrap_or_else(|| String::from("Bob"));
    let years = std::env::args()
        .nth(2)
        .and_then(|arg| arg.parse::<i32>().ok())
        .unwrap_or(42);

    binder::ProcessState::start_thread_pool();
    let service = connect().expect("Failed to connect to BirthdayService");
    let msg = service.wishHappyBirthday(&name, years)?;
    println!("{msg}");
    Ok(())
}
```

birthday_service/Android.bp:

```
rust_binary {
    name: "birthday_client",
    crate_name: "birthday_client",
    srcs: ["src/client.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbinder_rs",
    ],
    prefer_rlib: true,
}
```

Notice that the client does not depend on `libbirthdayservice`.

Build, push, and run the client on your device:

```
$ m birthday_client
$ adb push $ANDROID_PRODUCT_OUT/system/bin/birthday_client /data/local/tmp
$ adb shell /data/local/tmp/birthday_client Charlie 60
Happy Birthday Charlie, congratulations with the 60 years!
```

Changing API

Let us extend the API with more functionality: we want to let clients specify a list of lines for the birthday card:

```
package com.example.birthdayservice;

/** Birthday service interface. */
interface IBirthdayService {
    /** Generate a Happy Birthday message. */
    String wishHappyBirthday(String name, int years, in String[] text);
}
```

Logging

You should use the `log` crate to automatically log to `logcat` (on-device) or `stdout` (on-host):

hello_rust_logs/Android.bp:

```
rust_binary {
    name: "hello_rust_logs",
    crate_name: "hello_rust_logs",
    srcs: ["src/main.rs"],
    rustlibs: [
        "liblog_rust",
        "liblogger",
    ],
    prefer_rlib: true,
    host_supported: true,
}
```

hello_rust_logs/src/main.rs:

```
//! Rust logging demo.

use log::{debug, error, info};

/// Logs a greeting.
fn main() {
    logger::init(
        logger::Config::default()
            .with_tag_on_device("rust")
            .with_min_level(log::Level::Trace),
    );
    debug!("Starting program.");
    info!("Things are going fine.");
    error!("Something went wrong!");
}
```

Build, push, and run the binary on your device:

```
$ m hello_rust_logs
$ adb push $ANDROID_PRODUCT_OUT/system/bin/hello_rust_logs /data/local/tmp
$ adb shell /data/local/tmp/hello_rust_logs
```

The logs show up in `adb logcat`:

```
$ adb logcat -s rust
09-08 08:38:32.454  2420  2420 D rust: hello_rust_logs: Starting program.
09-08 08:38:32.454  2420  2420 I rust: hello_rust_logs: Things are going fine.
09-08 08:38:32.454  2420  2420 E rust: hello_rust_logs: Something went wrong!
```

Interoperability

Rust has excellent support for interoperability with other languages. This means that you can:

- Call Rust functions from other languages.
- Call functions written in other languages from Rust.

When you call functions in a foreign language we say that you're using a *foreign function interface*, also known as FFI.

Interoperability with C

Rust has full support for linking object files with a C calling convention. Similarly, you can export Rust functions and call them from C.

You can do it by hand if you want:

```
extern "C" {
    fn abs(x: i32) -> i32;
}

fn main() {
    let x = -42;
    let abs_x = unsafe { abs(x) };
    println!("{x}, {abs_x}");
}
```

We already saw this in the Safe FFI Wrapper exercise.

> This assumes full knowledge of the target platform. Not recommended for production.

We will look at better options next.

Using Bindgen

The bindgen tool can auto-generate bindings from a C header file.

First create a small C library:

interoperability/bindgen/libbirthday.h:

```
typedef struct card {
  const char* name;
  int years;
} card;

void print_card(const card* card);
```

interoperability/bindgen/libbirthday.c:

```
#include <stdio.h>
#include "libbirthday.h"

void print_card(const card* card) {
  printf("+--------------\n");
  printf("| Happy Birthday %s!\n", card->name);
  printf("| Congratulations with the %i years!\n", card->years);
  printf("+--------------\n");
}
```

Add this to your `Android.bp` file:

interoperability/bindgen/Android.bp:

```
cc_library {
    name: "libbirthday",
    srcs: ["libbirthday.c"],
}
```

Create a wrapper header file for the library (not strictly needed in this example):

interoperability/bindgen/libbirthday_wrapper.h:

```
#include "libbirthday.h"
```

You can now auto-generate the bindings:

interoperability/bindgen/Android.bp:

```
rust_bindgen {
    name: "libbirthday_bindgen",
    crate_name: "birthday_bindgen",
    wrapper_src: "libbirthday_wrapper.h",
    source_stem: "bindings",
    static_libs: ["libbirthday"],
}
```

Finally, we can use the bindings in our Rust program:

interoperability/bindgen/Android.bp:

```
rust_binary {
    name: "print_birthday_card",
    srcs: ["main.rs"],
    rustlibs: ["libbirthday_bindgen"],
}
```

interoperability/bindgen/main.rs:

```rust
//! Bindgen demo.

use birthday_bindgen::{card, print_card};

fn main() {
    let name = std::ffi::CString::new("Peter").unwrap();
    let card = card {
        name: name.as_ptr(),
        years: 42,
    };
    unsafe {
        print_card(&card as *const card);
    }
}
```

Build, push, and run the binary on your device:

```shell
$ m print_birthday_card
$ adb push $ANDROID_PRODUCT_OUT/system/bin/print_birthday_card /data/local/tmp
$ adb shell /data/local/tmp/print_birthday_card
```

Finally, we can run auto-generated tests to ensure the bindings work:

interoperability/bindgen/Android.bp:

```
rust_test {
    name: "libbirthday_bindgen_test",
    srcs: [":libbirthday_bindgen"],
    crate_name: "libbirthday_bindgen_test",
    test_suites: ["general-tests"],
    auto_gen_config: true,
    clippy_lints: "none", // Generated file, skip linting
    lints: "none",
}
```

```shell
$ atest libbirthday_bindgen_test
```

Calling Rust

Exporting Rust functions and types to C is easy:

interoperability/rust/libanalyze/analyze.rs

```rust
//! Rust FFI demo.
#![deny(improper_ctypes_definitions)]

use std::os::raw::c_int;

/// Analyze the numbers.
#[no_mangle]
pub extern "C" fn analyze_numbers(x: c_int, y: c_int) {
    if x < y {
        println!("x ({x}) is smallest!");
    } else {
        println!("y ({y}) is probably larger than x ({x})");
    }
}
```

interoperability/rust/libanalyze/analyze.h

```c
#ifndef ANALYSE_H
#define ANALYSE_H

extern "C" {
void analyze_numbers(int x, int y);
}

#endif
```

interoperability/rust/libanalyze/Android.bp

```
rust_ffi {
    name: "libanalyze_ffi",
    crate_name: "analyze_ffi",
    srcs: ["analyze.rs"],
    include_dirs: ["."],
}
```

We can now call this from a C binary:

interoperability/rust/analyze/main.c

```c
#include "analyze.h"

int main() {
  analyze_numbers(10, 20);
  analyze_numbers(123, 123);
  return 0;
}
```

interoperability/rust/analyze/Android.bp

```
cc_binary {
    name: "analyze_numbers",
    srcs: ["main.c"],
    static_libs: ["libanalyze_ffi"],
}
```

Build, push, and run the binary on your device:

```shell
$ m analyze_numbers
$ adb push $ANDROID_PRODUCT_OUT/system/bin/analyze_numbers /data/local/tmp
$ adb shell /data/local/tmp/analyze_numbers
```

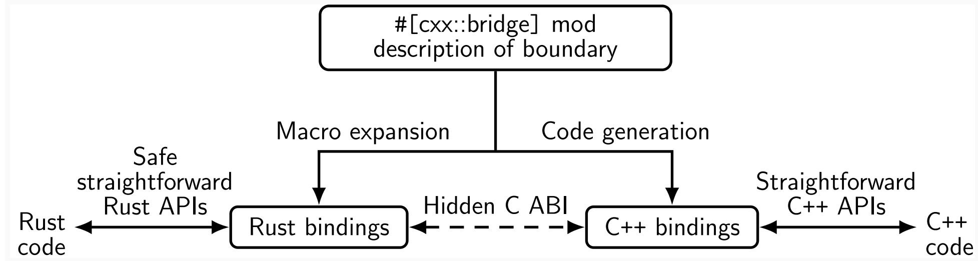
▼ Details

`#[no_mangle]` disables Rust's usual name mangling, so the exported symbol will just be the name of the function. You can also use `#[export_name = "some_name"]` to specify whatever name you want.

With C++

The CXX crate makes it possible to do safe interoperability between Rust and C++.

The overall approach looks like this:



See the CXX tutorial for an full example of using this.

Interoperability with Java

Java can load shared objects via Java Native Interface (JNI). The `jni` crate allows you to create a compatible library.

First, we create a Rust function to export to Java:

interoperability/java/src/lib.rs:

```
//! Rust <-> Java FFI demo.

use jni::objects::{JClass, JString};
use jni::sys::jstring;
use jni::JNIEnv;

/// HelloWorld::hello method implementation.
#[no_mangle]
pub extern "system" fn Java_HelloWorld_hello(
    env: JNIEnv,
    _class: JClass,
    name: JString,
) -> jstring {
    let input: String = env.get_string(name).unwrap().into();
    let greeting = format!("Hello, {input}!");
    let output = env.new_string(greeting).unwrap();
    output.into_inner()
}
```

interoperability/java/Android.bp:

```
rust_ffi_shared {
    name: "libhello_jni",
    crate_name: "hello_jni",
    srcs: ["src/lib.rs"],
    rustlibs: ["libjni"],
}
```

Finally, we can call this function from Java:

interoperability/java/HelloWorld.java:

```
class HelloWorld {
    private static native String hello(String name);

    static {
        System.loadLibrary("hello_jni");
    }

    public static void main(String[] args) {
        String output = HelloWorld.hello("Alice");
        System.out.println(output);
    }
}
```

interoperability/java/Android.bp:

```
java_binary {
    name: "helloworld_jni",
    srcs: ["HelloWorld.java"],
    main_class: "HelloWorld",
    required: ["libhello_jni"],
}
```

Finally, you can build, sync, and run the binary:

```
$ m helloworld_jni
$ adb sync  # requires adb root && adb remount
$ adb shell /system/bin/helloworld_jni
```

Exercises

For the last exercise, we will look at one of the projects you work with. Let us group up and do this together. Some suggestions:

- Call your AIDL service with a client written in Rust.
- Move a function from your project to Rust and call it.

▼ Details

No solution is provided here since this is open-ended: it relies on someone in the class having a piece of code which you can turn in to Rust on the fly.

Thanks!

Thank you for taking Comprehensive Rust 🦀! We hope you enjoyed it and that it was useful.

We've had a lot of fun putting the course together. The course is not perfect, so if you spotted any mistakes or have ideas for improvements, please get in contact with us on GitHub. We would love to hear from you.

Other Rust Resources

The Rust community has created a wealth of high-quality and free resources online.

Official Documentation

The Rust project hosts many resources. These cover Rust in general:

- The Rust Programming Language: the canonical free book about Rust. Covers the language in detail and includes a few projects for people to build.
- Rust By Example: covers the Rust syntax via a series of examples which showcase different constructs. Sometimes includes small exercises where you are asked to expand on the code in the examples.
- Rust Standard Library: full documentation of the standard library for Rust.
- The Rust Reference: an incomplete book which describes the Rust grammar and memory model.

More specialized guides hosted on the official Rust site:

- The Rustonomicon: covers unsafe Rust, including working with raw pointers and interfacing with other languages (FFI).
- Asynchronous Programming in Rust: covers the new asynchronous programming model which was introduced after the Rust Book was written.
- The Embedded Rust Book: an introduction to using Rust on embedded devices without an operating system.

Unofficial Learning Material

A small selection of other guides and tutorial for Rust:

- Learn Rust the Dangerous Way: covers Rust from the perspective of low-level C programmers.
- Rust for Embedded C Programmers: covers Rust from the perspective of developers who write firmware in C.
- Rust for professionals: covers the syntax of Rust using side-by-side comparisons with other languages such as C, C++, Java, JavaScript, and Python.
- Rust on Exercism: 100+ exercises to help you learn Rust.
- Ferrous Teaching Material: a series of small presentations covering both basic and advanced part of the Rust language. Other topics such as WebAssembly, and async/await are also covered.
- Beginner's Series to Rust and Take your first steps with Rust: two Rust guides aimed at new developers. The first is a set of 35 videos and the second is a set of 11 modules which covers Rust syntax and basic constructs.

Please see the Little Book of Rust Books for even more Rust books.

Credits

The material here builds on top of the many great sources of Rust documentation. See the page on other resources for a full list of useful resources.

The material of Comprehensive Rust is licensed under the terms of the Apache 2.0 license, please see `LICENSE` for details.

Rust by Example

Some examples and exercises have been copied and adapted from Rust by Example. Please see the `third_party/rust-by-example/` directory for details, including the license terms.

Rust on Exercism

Some exercises have been copied and adapted from Rust on Exercism. Please see the `third_party/rust-on-exercism/` directory for details, including the license terms.

CXX

The Interoperability with C++ section uses an image from CXX. Please see the `third_party/cxx/` directory for details, including the license terms.

Solutions

You will find solutions to the exercises on the following pages.

Feel free to ask questions about the solutions on GitHub. Let us know if you have a different or better solution than what is presented here.

> **Note:** Please ignore the `// ANCHOR: label` and `// ANCHOR_END: label` comments you see in the solutions. They are there to make it possible to re-use parts of the solutions as the exercises.

Day 1 Morning Exercises

Arrays and `for` Loops

(back to exercise)

```rust
// Copyright 2022 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

// ANCHOR: transpose
fn transpose(matrix: [[i32; 3]; 3]) -> [[i32; 3]; 3] {
    // ANCHOR_END: transpose
    let mut result = [[0; 3]; 3];
    for i in 0..3 {
        for j in 0..3 {
            result[j][i] = matrix[i][j];
        }
    }
    return result;
}

// ANCHOR: pretty_print
fn pretty_print(matrix: &[[i32; 3]; 3]) {
    // ANCHOR_END: pretty_print
    for row in matrix {
        println!("{row:?}");
    }
}

// ANCHOR: tests
#[test]
fn test_transpose() {
    let matrix = [
        [101, 102, 103], //
        [201, 202, 203],
        [301, 302, 303],
    ];
    let transposed = transpose(matrix);
    assert_eq!(
        transposed,
        [
            [101, 201, 301], //
            [102, 202, 302],
            [103, 203, 303],
        ]
    );
}
// ANCHOR_END: tests

// ANCHOR: main
fn main() {
    let matrix = [
        [101, 102, 103], // <-- the comment makes rustfmt add a newline
        [201, 202, 203],
        [301, 302, 303],
    ];

    println!("matrix:");
    pretty_print(&matrix);

    let transposed = transpose(matrix);
    println!("transposed:");
    pretty_print(&transposed);
}
```

Bonus question

It requires more advanced concepts. It might seem that we could use a slice-of-slices (`&[&[i32]]`) as the input type to transpose and thus make our function handle any size of matrix. However, this quickly breaks down: the return type cannot be `&[&[i32]]` since it needs to own the data you return.

You can attempt to use something like `Vec<Vec<i32>>`, but this doesn't work out-of-the-box either: it's hard to convert from `Vec<Vec<i32>>` to `&[&[i32]]` so now you cannot easily use `pretty_print` either.

Once we get to traits and generics, we'll be able to use the `std::convert::AsRef` trait to abstract over anything that can be referenced as a slice.

```
use std::convert::AsRef;
use std::fmt::Debug;

fn pretty_print<T, Line, Matrix>(matrix: Matrix)
where
    T: Debug,
    // A line references a slice of items
    Line: AsRef<[T]>,
    // A matrix references a slice of lines
    Matrix: AsRef<[Line]>
{
    for row in matrix.as_ref() {
        println!("{:?}", row.as_ref());
    }
}

fn main() {
    // &[&[i32]]
    pretty_print(&[&[1, 2, 3], &[4, 5, 6], &[7, 8, 9]]);
    // [[&str; 2]; 2]
    pretty_print([["a", "b"], ["c", "d"]]);
    // Vec<Vec<i32>>
    pretty_print(vec![vec![1, 2], vec![3, 4]]);
}
```

In addition, the type itself would not enforce that the child slices are of the same length, so such variable could contain an invalid matrix.

Day 1 Afternoon Exercises

Designing a Library

(back to exercise)

```
// Copyright 2022 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

// ANCHOR: setup
struct Library {
    books: Vec<Book>,
}

struct Book {
    title: String,
    year: u16,
}

impl Book {
    // This is a constructor, used below.
    fn new(title: &str, year: u16) -> Book {
        Book {
            title: String::from(title),
            year,
        }
    }
}

// This makes it possible to print Book values with {}.
impl std::fmt::Display for Book {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{} ({})", self.title, self.year)
    }
}
// ANCHOR_END: setup

// ANCHOR: Library_new
impl Library {
    fn new() -> Library {
        // ANCHOR_END: Library_new
        Library { books: Vec::new() }
    }

    // ANCHOR: Library_len
    //fn len(self) -> usize {
    //    unimplemented!()
    //}
    // ANCHOR_END: Library_len
    fn len(&self) -> usize {
        self.books.len()
    }

    // ANCHOR: Library_is_empty
    //fn is_empty(self) -> bool {
    //    unimplemented!()
    //}
    // ANCHOR_END: Library_is_empty
    fn is_empty(&self) -> bool {
        self.books.is_empty()
    }

    // ANCHOR: Library_add_book
    //fn add_book(self, book: Book) {
    //    unimplemented!()
    //}
```

```
    // ANCHOR_END: Library_add_book
    fn add_book(&mut self, book: Book) {
        self.books.push(book)
    }

    // ANCHOR: Library_print_books
    //fn print_books(self) {
    //    unimplemented!()
    //}
    // ANCHOR_END: Library_print_books
    fn print_books(&self) {
        for book in &self.books {
            println!("{}", book);
        }
    }

    // ANCHOR: Library_oldest_book
    //fn oldest_book(self) -> Option<&Book> {
    //    unimplemented!()
    //}
    // ANCHOR_END: Library_oldest_book
    fn oldest_book(&self) -> Option<&Book> {
        self.books.iter().min_by_key(|book| book.year)
    }
}

// ANCHOR: main
// This shows the desired behavior. Uncomment the code below and
// implement the missing methods. You will need to update the
// method signatures, including the "self" parameter! You may
// also need to update the variable bindings within main.
fn main() {
    let library = Library::new();

    //println!("Our library is empty: {}", library.is_empty());
    //
    //library.add_book(Book::new("Lord of the Rings", 1954));
    //library.add_book(Book::new("Alice's Adventures in Wonderland", 1865));
    //
    //library.print_books();
    //
    //match library.oldest_book() {
    //    Some(book) => println!("My oldest book is {book}"),
    //    None => println!("My library is empty!"),
    //}
    //
    //println!("Our library has {} books", library.len());
}
// ANCHOR_END: main

#[test]
fn test_library_len() {
    let mut library = Library::new();
    assert_eq!(library.len(), 0);
    assert!(library.is_empty());

    library.add_book(Book::new("Lord of the Rings", 1954));
    library.add_book(Book::new("Alice's Adventures in Wonderland", 1865));
    assert_eq!(library.len(), 2);
    assert!(!library.is_empty());
}

#[test]
fn test_library_is_empty() {
    let mut library = Library::new();
    assert!(library.is_empty());

    library.add_book(Book::new("Lord of the Rings", 1954));
    assert!(!library.is_empty());
}

#[test]
```

```rust
fn test_library_print_books() {
    let mut library = Library::new();
    library.add_book(Book::new("Lord of the Rings", 1954));
    library.add_book(Book::new("Alice's Adventures in Wonderland", 1865));
    // We could try and capture stdout, but let us just call the
    // method to start with.
    library.print_books();
}

#[test]
fn test_library_oldest_book() {
    let mut library = Library::new();
    assert!(library.oldest_book().is_none());

    library.add_book(Book::new("Lord of the Rings", 1954));
    assert_eq!(
        library.oldest_book().map(|b| b.title.as_str()),
        Some("Lord of the Rings")
    );

    library.add_book(Book::new("Alice's Adventures in Wonderland", 1865));
    assert_eq!(
        library.oldest_book().map(|b| b.title.as_str()),
        Some("Alice's Adventures in Wonderland")
    );
}
```

Day 2 Morning Exercises

Points and Polygons

(back to exercise)

```rust
// Copyright 2022 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

#[derive(Debug, Copy, Clone, PartialEq, Eq)]
// ANCHOR: Point
pub struct Point {
    // ANCHOR_END: Point
    x: i32,
    y: i32,
}

// ANCHOR: Point-impl
impl Point {
    // ANCHOR_END: Point-impl
    pub fn new(x: i32, y: i32) -> Point {
        Point { x, y }
    }

    pub fn magnitude(self) -> f64 {
        f64::from(self.x.pow(2) + self.y.pow(2)).sqrt()
    }

    pub fn dist(self, other: Point) -> f64 {
        (self - other).magnitude()
    }
}

impl std::ops::Add for Point {
    type Output = Self;

    fn add(self, other: Self) -> Self::Output {
        Self {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

impl std::ops::Sub for Point {
    type Output = Self;

    fn sub(self, other: Self) -> Self::Output {
        Self {
            x: self.x - other.x,
            y: self.y - other.y,
        }
    }
}

// ANCHOR: Polygon
pub struct Polygon {
    // ANCHOR_END: Polygon
    points: Vec<Point>,
}

// ANCHOR: Polygon-impl
impl Polygon {
    // ANCHOR_END: Polygon-impl
    pub fn new() -> Polygon {
        Polygon { points: Vec::new() }
```

```
    }

    pub fn add_point(&mut self, point: Point) {
        self.points.push(point);
    }

    pub fn left_most_point(&self) -> Option<Point> {
        self.points.iter().min_by_key(|p| p.x).copied()
    }

    pub fn iter(&self) -> impl Iterator<Item = &Point> {
        self.points.iter()
    }

    pub fn length(&self) -> f64 {
        if self.points.is_empty() {
            return 0.0;
        }

        let mut result = 0.0;
        let mut last_point = self.points[0];
        for point in &self.points[1..] {
            result += last_point.dist(*point);
            last_point = *point;
        }
        result += last_point.dist(self.points[0]);
        result
    }
}

// ANCHOR: Circle
pub struct Circle {
    // ANCHOR_END: Circle
    center: Point,
    radius: i32,
}

// ANCHOR: Circle-impl
impl Circle {
    // ANCHOR_END: Circle-impl
    pub fn new(center: Point, radius: i32) -> Circle {
        Circle { center, radius }
    }

    pub fn circumference(&self) -> f64 {
        2.0 * std::f64::consts::PI * f64::from(self.radius)
    }

    pub fn dist(&self, other: &Self) -> f64 {
        self.center.dist(other.center)
    }
}

// ANCHOR: Shape
pub enum Shape {
    Polygon(Polygon),
    Circle(Circle),
}
// ANCHOR_END: Shape

impl From<Polygon> for Shape {
    fn from(poly: Polygon) -> Self {
        Shape::Polygon(poly)
    }
}

impl From<Circle> for Shape {
    fn from(circle: Circle) -> Self {
        Shape::Circle(circle)
    }
}
```

```rust
impl Shape {
    pub fn perimeter(&self) -> f64 {
        match self {
            Shape::Polygon(poly) => poly.length(),
            Shape::Circle(circle) => circle.circumference(),
        }
    }
}

// ANCHOR: unit-tests
#[cfg(test)]
mod tests {
    use super::*;

    fn round_two_digits(x: f64) -> f64 {
        (x * 100.0).round() / 100.0
    }

    #[test]
    fn test_point_magnitude() {
        let p1 = Point::new(12, 13);
        assert_eq!(round_two_digits(p1.magnitude()), 17.69);
    }

    #[test]
    fn test_point_dist() {
        let p1 = Point::new(10, 10);
        let p2 = Point::new(14, 13);
        assert_eq!(round_two_digits(p1.dist(p2)), 5.00);
    }

    #[test]
    fn test_point_add() {
        let p1 = Point::new(16, 16);
        let p2 = p1 + Point::new(-4, 3);
        assert_eq!(p2, Point::new(12, 19));
    }

    #[test]
    fn test_polygon_left_most_point() {
        let p1 = Point::new(12, 13);
        let p2 = Point::new(16, 16);

        let mut poly = Polygon::new();
        poly.add_point(p1);
        poly.add_point(p2);
        assert_eq!(poly.left_most_point(), Some(p1));
    }

    #[test]
    fn test_polygon_iter() {
        let p1 = Point::new(12, 13);
        let p2 = Point::new(16, 16);

        let mut poly = Polygon::new();
        poly.add_point(p1);
        poly.add_point(p2);

        let points = poly.iter().cloned().collect::<Vec<_>>();
        assert_eq!(points, vec![Point::new(12, 13), Point::new(16, 16)]);
    }

    #[test]
    fn test_shape_perimeters() {
        let mut poly = Polygon::new();
        poly.add_point(Point::new(12, 13));
        poly.add_point(Point::new(17, 11));
        poly.add_point(Point::new(16, 16));
        let shapes = vec![
            Shape::from(poly),
            Shape::from(Circle::new(Point::new(10, 20), 5)),
        ];
```

```
        let perimeters = shapes
            .iter()
            .map(Shape::perimeter)
            .map(round_two_digits)
            .collect::<Vec<_>>();
        assert_eq!(perimeters, vec![15.48, 31.42]);
    }
}
// ANCHOR_END: unit-tests

fn main() {}
```

Day 2 Afternoon Exercises

Luhn Algorithm

(back to exercise)

```rust
// Copyright 2022 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

// ANCHOR: luhn
pub fn luhn(cc_number: &str) -> bool {
    // ANCHOR_END: luhn
    let mut digits_seen = 0;
    let mut sum = 0;
    for (i, ch) in cc_number.chars().rev().filter(|&ch| ch != ' ').enumerate() {
        match ch.to_digit(10) {
            Some(d) => {
                sum += if i % 2 == 1 {
                    let dd = d * 2;
                    dd / 10 + dd % 10
                } else {
                    d
                };
                digits_seen += 1;
            }
            None => return false,
        }
    }

    if digits_seen < 2 {
        return false;
    }

    sum % 10 == 0
}

fn main() {
    let cc_number = "1234 5678 1234 5670";
    println!(
        "Is {} a valid credit card number? {}",
        cc_number,
        if luhn(cc_number) { "yes" } else { "no" }
    );
}

// ANCHOR: unit-tests
#[test]
fn test_non_digit_cc_number() {
    assert!(!luhn("foo"));
}

#[test]
fn test_empty_cc_number() {
    assert!(!luhn(""));
    assert!(!luhn(" "));
    assert!(!luhn("  "));
    assert!(!luhn("    "));
}

#[test]
fn test_single_digit_cc_number() {
    assert!(!luhn("0"));
}

#[test]
fn test_two_digit_cc_number() {
```

```
    assert!(luhn(" 0 0 "));
}

#[test]
fn test_valid_cc_number() {
    assert!(luhn("4263 9826 4026 9299"));
    assert!(luhn("4539 3195 0343 6467"));
    assert!(luhn("7992 7398 713"));
}

#[test]
fn test_invalid_cc_number() {
    assert!(!luhn("4223 9826 4026 9299"));
    assert!(!luhn("4539 3195 0343 6476"));
    assert!(!luhn("8273 1232 7352 0569"));
}
// ANCHOR_END: unit-tests
```

Strings and Iterators

(back to exercise)

```rust
// Copyright 2022 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

// ANCHOR: prefix_matches
pub fn prefix_matches(prefix: &str, request_path: &str) -> bool {
    // ANCHOR_END: prefix_matches
    let prefixes = prefix.split('/');
    let request_paths = request_path
        .split('/')
        .map(|p| Some(p))
        .chain(std::iter::once(None));

    for (prefix, request_path) in prefixes.zip(request_paths) {
        match request_path {
            Some(request_path) => {
                if (prefix != "*") && (prefix != request_path) {
                    return false;
                }
            }
            None => return false,
        }
    }
    true
}

// ANCHOR: unit-tests
#[test]
fn test_matches_without_wildcard() {
    assert!(prefix_matches("/v1/publishers", "/v1/publishers"));
    assert!(prefix_matches("/v1/publishers", "/v1/publishers/abc-123"));
    assert!(prefix_matches("/v1/publishers", "/v1/publishers/abc/books"));

    assert!(!prefix_matches("/v1/publishers", "/v1"));
    assert!(!prefix_matches("/v1/publishers", "/v1/publishersBooks"));
    assert!(!prefix_matches("/v1/publishers", "/v1/parent/publishers"));
}

#[test]
fn test_matches_with_wildcard() {
    assert!(prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/foo/books"
    ));
    assert!(prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/bar/books"
    ));
    assert!(prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/foo/books/book1"
    ));

    assert!(!prefix_matches("/v1/publishers/*/books", "/v1/publishers"));
    assert!(!prefix_matches(
        "/v1/publishers/*/books",
        "/v1/publishers/foo/booksByAuthor"
    ));
}
// ANCHOR_END: unit-tests
```

```
fn main() {}
```

Day 3 Morning Exercise

A Simple GUI Library

(back to exercise)

```rust
// Copyright 2022 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

// ANCHOR: setup
pub trait Widget {
    /// Natural width of `self`.
    fn width(&self) -> usize;

    /// Draw the widget into a buffer.
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write);

    /// Draw the widget on standard output.
    fn draw(&self) {
        let mut buffer = String::new();
        self.draw_into(&mut buffer);
        println!("{buffer}");
    }
}

pub struct Label {
    label: String,
}

impl Label {
    fn new(label: &str) -> Label {
        Label {
            label: label.to_owned(),
        }
    }
}

pub struct Button {
    label: Label,
    callback: Box<dyn FnMut()>,
}

impl Button {
    fn new(label: &str, callback: Box<dyn FnMut()>) -> Button {
        Button {
            label: Label::new(label),
            callback,
        }
    }
}

pub struct Window {
    title: String,
    widgets: Vec<Box<dyn Widget>>,
}

impl Window {
    fn new(title: &str) -> Window {
        Window {
            title: title.to_owned(),
            widgets: Vec::new(),
        }
    }

    fn add_widget(&mut self, widget: Box<dyn Widget>) {
        self.widgets.push(widget);
```

```
    }
}

// ANCHOR_END: setup

// ANCHOR: Window-width
impl Widget for Window {
    fn width(&self) -> usize {
        // ANCHOR_END: Window-width
        std::cmp::max(
            self.title.chars().count(),
            self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
        )
    }

    // ANCHOR: Window-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Window-draw_into
        let mut inner = String::new();
        for widget in &self.widgets {
            widget.draw_into(&mut inner);
        }

        let window_width = self.width();

        // TODO: after learning about error handling, you can change
        // draw_into to return Result<(), std::fmt::Error>. Then use
        // the ?-operator here instead of .unwrap().
        writeln!(buffer, "+-{:-<window_width$}-+", "").unwrap();
        writeln!(buffer, "| {:^window_width$} |", &self.title).unwrap();
        writeln!(buffer, "+={:=<window_width$}=+", "").unwrap();
        for line in inner.lines() {
            writeln!(buffer, "| {:window_width$} |", line).unwrap();
        }
        writeln!(buffer, "+-{:-<window_width$}-+", "").unwrap();
    }
}

// ANCHOR: Button-width
impl Widget for Button {
    fn width(&self) -> usize {
        // ANCHOR_END: Button-width
        self.label.width() + 8 // add a bit of padding
    }

    // ANCHOR: Button-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Button-draw_into
        let width = self.width();
        let mut label = String::new();
        self.label.draw_into(&mut label);

        writeln!(buffer, "+{:-<width$}+", "").unwrap();
        for line in label.lines() {
            writeln!(buffer, "|{:^width$}|", &line).unwrap();
        }
        writeln!(buffer, "+{:-<width$}+", "").unwrap();
    }
}

// ANCHOR: Label-width
impl Widget for Label {
    fn width(&self) -> usize {
        // ANCHOR_END: Label-width
        self.label
            .lines()
            .map(|line| line.chars().count())
            .max()
            .unwrap_or(0)
    }

    // ANCHOR: Label-draw_into
```

```
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Label-draw_into
        writeln!(buffer, "{}", &self.label).unwrap();
    }
}

// ANCHOR: main
fn main() {
    let mut window = Window::new("Rust GUI Demo 1.23");
    window.add_widget(Box::new(Label::new("This is a small text GUI demo.")));
    window.add_widget(Box::new(Button::new(
        "Click me!",
        Box::new(|| println!("You clicked the button!")),
    )));
    window.draw();
}
// ANCHOR_END: main
```

Day 3 Afternoon Exercises

Safe FFI Wrapper

(back to exercise)

```rust
// Copyright 2022 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

// ANCHOR: ffi
mod ffi {
    use std::os::raw::{c_char, c_int, c_long, c_ulong, c_ushort};

    // Opaque type. See https://doc.rust-lang.org/nomicon/ffi.html.
    #[repr(C)]
    pub struct DIR {
        _data: [u8; 0],
        _marker: core::marker::PhantomData<(*mut u8, core::marker::PhantomPinned)>,
    }

    // Layout as per readdir(3) and definitions in /usr/include/x86_64-linux-gnu.
    #[repr(C)]
    pub struct dirent {
        pub d_ino: c_long,
        pub d_off: c_ulong,
        pub d_reclen: c_ushort,
        pub d_type: c_char,
        pub d_name: [c_char; 256],
    }

    extern "C" {
        pub fn opendir(s: *const c_char) -> *mut DIR;
        pub fn readdir(s: *mut DIR) -> *const dirent;
        pub fn closedir(s: *mut DIR) -> c_int;
    }
}

use std::ffi::{CStr, CString, OsStr, OsString};
use std::os::unix::ffi::OsStrExt;

#[derive(Debug)]
struct DirectoryIterator {
    path: CString,
    dir: *mut ffi::DIR,
}
// ANCHOR_END: ffi

// ANCHOR: DirectoryIterator
impl DirectoryIterator {
    fn new(path: &str) -> Result<DirectoryIterator, String> {
        // Call opendir and return a Ok value if that worked,
        // otherwise return Err with a message.
        // ANCHOR_END: DirectoryIterator
        let path = CString::new(path).map_err(|err| format!("Invalid path: {err}"))?;
        // SAFETY: path.as_ptr() cannot be NULL.
        let dir = unsafe { ffi::opendir(path.as_ptr()) };
        if dir.is_null() {
            Err(format!("Could not open {:?}", path))
        } else {
            Ok(DirectoryIterator { path, dir })
        }
    }
}

// ANCHOR: Iterator
impl Iterator for DirectoryIterator {
```

```
    type Item = OsString;
    fn next(&mut self) -> Option<OsString> {
        // Keep calling readdir until we get a NULL pointer back.
        // ANCHOR_END: Iterator
        // SAFETY: self.dir is never NULL.
        let dirent = unsafe { ffi::readdir(self.dir) };
        if dirent.is_null() {
            // We have reached the end of the directory.
            return None;
        }
        // SAFETY: dirent is not NULL and dirent.d_name is NUL
        // terminated.
        let d_name = unsafe { CStr::from_ptr((*dirent).d_name.as_ptr()) };
        let os_str = OsStr::from_bytes(d_name.to_bytes());
        Some(os_str.to_owned())
    }
}

// ANCHOR: Drop
impl Drop for DirectoryIterator {
    fn drop(&mut self) {
        // Call closedir as needed.
        // ANCHOR_END: Drop
        if !self.dir.is_null() {
            // SAFETY: self.dir is not NULL.
            if unsafe { ffi::closedir(self.dir) } != 0 {
                panic!("Could not close {:?}", self.path);
            }
        }
    }
}

// ANCHOR: main
fn main() -> Result<(), String> {
    let iter = DirectoryIterator::new(".")?;
    println!("files: {:#?}", iter.collect::<Vec<_>>());
    Ok(())
}
// ANCHOR_END: main
```

Day 4 Morning Exercise

Dining Philosophers

(back to exercise)

```rust
// Copyright 2022 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

// ANCHOR: Philosopher
use std::sync::mpsc;
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;

struct Fork;

struct Philosopher {
    name: String,
    // ANCHOR_END: Philosopher
    left_fork: Arc<Mutex<Fork>>,
    right_fork: Arc<Mutex<Fork>>,
    thoughts: mpsc::SyncSender<String>,
}

// ANCHOR: Philosopher-think
impl Philosopher {
    fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .unwrap();
    }
    // ANCHOR_END: Philosopher-think

    // ANCHOR: Philosopher-eat
    fn eat(&self) {
        // ANCHOR_END: Philosopher-eat
        println!("{} is trying to eat", &self.name);
        let left = self.left_fork.lock().unwrap();
        let right = self.right_fork.lock().unwrap();

        // ANCHOR: Philosopher-eat-end
        println!("{} is eating...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "Plato", "Aristotle", "Thales", "Pythagoras"];

fn main() {
    // ANCHOR_END: Philosopher-eat-end
    let (tx, rx) = mpsc::sync_channel(10);

    let forks = (0..PHILOSOPHERS.len())
        .map(|_| Arc::new(Mutex::new(Fork)))
        .collect::<Vec<_>>();

    for i in 0..forks.len() {
        let tx = tx.clone();
        let mut left_fork = forks[i].clone();
        let mut right_fork = forks[(i + 1) % forks.len()].clone();

        // To avoid a deadlock, we have to break the symmetry
        // somewhere. This will swap the forks without deinitializing
        // either of them.
```

```
        if i == forks.len() - 1 {
            std::mem::swap(&mut left_fork, &mut right_fork);
        }

        let philosopher = Philosopher {
            name: PHILOSOPHERS[i].to_string(),
            thoughts: tx,
            left_fork,
            right_fork,
        };

        thread::spawn(move || {
            for _ in 0..100 {
                philosopher.eat();
                philosopher.think();
            }
        });
    }

    drop(tx);
    for thought in rx {
        println!("{thought}");
    }
}
```