Welcome to Comprehensive Rust

This is a three 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.

The first three days show you the fundamentals of Rust. Following this, you're invited to dive into one or more specialized topics:

- Android: a half-day course on using Rust for Android platform development (AOSP). This includes interoperability with C, C++, and Java.
- Bare-metal: a whole-day class on using Rust for bare-metal (embedded) development. Both microcontrollers and application processors are covered.
- Concurrency: a whole-day class on concurrency in Rust. We cover both classical concurrency (preemptively scheduling using threads and mutexes) and async/await concurrency (cooperative multitasking using futures).

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:

- Learning 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*

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.

Before you run the course, you will want to:

1. Make yourself familiar with the course material. We've included speaker notes to help highlight the key points (please help us by contributing more speaker notes!). When presenting, 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 takes at least three full days, we recommend that you schedule the 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 15-25 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. Make sure the room has *desks* for yourself and for the students: you will all need to be able to sit and work with your laptops. In particular, you will be doing a lot of live-coding as an instructor, so a lectern won't be very helpful for you.

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. We typically spend 30-45 minutes on exercises in the morning and in the afternoon (including time to review the solutions). 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.

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.

Deep Dives

In addition to the 3-day class on Rust Fundamentals, we cover some more specialized topics:

Android

The Android Deep Dive is a half-day course on using Rust for Android platform development. This includes interoperability with C, C++, and Java.

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.

Bare-Metal

The Bare-Metal Deep Dive: a full day class on using Rust for bare-metal (embedded) development. Both microcontrollers and application processors are covered.

For the microcontroller part, you will need to buy the BBC micro:bit v2 development board ahead of time. Everybody will need to install a number of packages as described on the welcome page.

Concurrency

The Concurrency Deep Dive is a full day class on classical as well as `async` / `await` concurrency.

You will need a fresh crate set up and the dependencies downloaded and ready to go. You can then copy/paste the examples into `src/main.rs` to experiment with them:

```
cargo init concurrency
cd concurrency
cargo add tokio --features full
```

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.

Translations

The course has been translated into other languages by a set of wonderful volunteers:

- Brazilian Portuguese by @rastringer and @hugojacob.
- Korean by @keispace, @jiyongp and @jooyunghan.

Use the language picker in the top-right corner to switch between languages.

Incomplete Translations

There is a large number of in-progress translations. We link to the most recently updated translations:

- Bengali by @raselmandol.
- French by @KookaS and @vcaen.
- German by @Throvn and @ronaldfw.
- Japanese by @CoinEZ-JPN and @momotaro1105.

If you want to help with this effort, please see our instructions for how to get going. Translations are coordinated on the issue tracker.

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

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

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 1.69.0 (84c898d65 2023-04-16)
% cargo --version
cargo 1.69.0 (6e9a83356 2023-04-12)
```

With this in place, 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`

▶ Details

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

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

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

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

Why Rust?

Some unique selling points of Rust:

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

▶ Details

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

Runtime Guarantees

No undefined behavior at runtime:

- Array access is bounds checked.
- Integer overflow is defined (panic or wrap-around).

▶ Details

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

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"`, `"two\nlines"`
Unicode scalar values	`char`	`'a'`, `'α'`, `'∞'`
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 bits wide,
- `bool` is 8 bits wide.

▶ Details

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

References

Like C++, Rust has references:

```
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

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 mut 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

`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

Functions

A Rust version of the famous FizzBuzz interview question:

```rust
fn main() {
    print_fizzbuzz_to(20);
}

fn is_divisible(n: u32, divisor: u32) -> bool {
    if divisor == 0 {
        return false;
    }
    n % divisor == 0
}

fn fizzbuzz(n: u32) -> String {
    let fizz = if is_divisible(n, 3) { "fizz" } else { "" };
    let buzz = if is_divisible(n, 5) { "buzz" } else { "" };
    if fizz.is_empty() && buzz.is_empty() {
        return format!("{n}");
    }
    format!("{fizz}{buzz}")
}

fn print_fizzbuzz_to(n: u32) {
    for i in 1..=n {
        println!("{}", fizzbuzz(i));
    }
}
```

▶ Details

Rustdoc

All language items in Rust can be documented using special `///` syntax.

```
/// Determine whether the first argument is divisible by the second argument.
///
/// If the second argument is zero, the result is false.
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
}
```

The contents are treated as Markdown. All published Rust library crates are automatically documented at `docs.rs` using the rustdoc tool. It is idiomatic to document all public items in an API using this pattern.

▶ Details

Methods

Methods are functions associated with a type. The `self` argument of a method is an instance of the type it is associated with:

```
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.

▶ Details

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

Day 1: Morning Exercises

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

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

▶ Details

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):

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

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

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

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

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 to 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

Scopes and Shadowing

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

```
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

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.

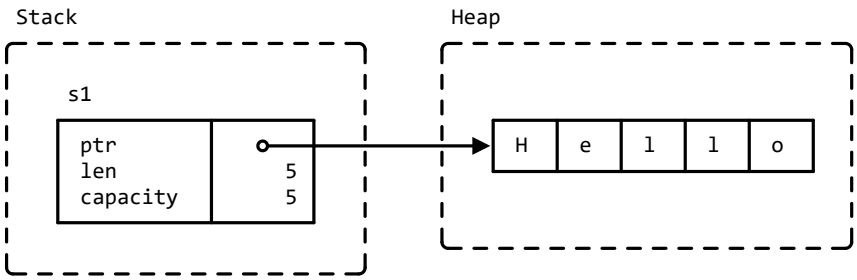
The Stack vs The Heap

- 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

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 = 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:

```
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.

Rust achieves this by modeling *ownership* explicitly.

▶ Details

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.

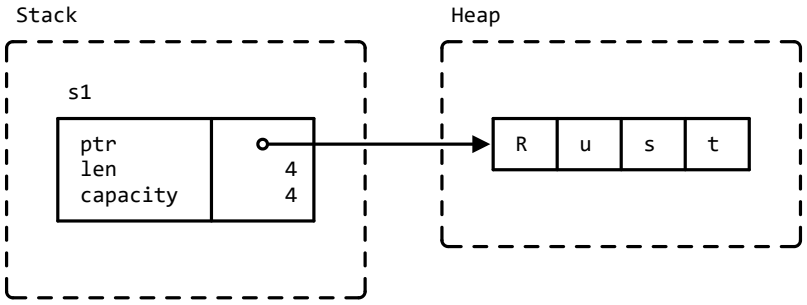
▶ Details

Moved Strings in Rust

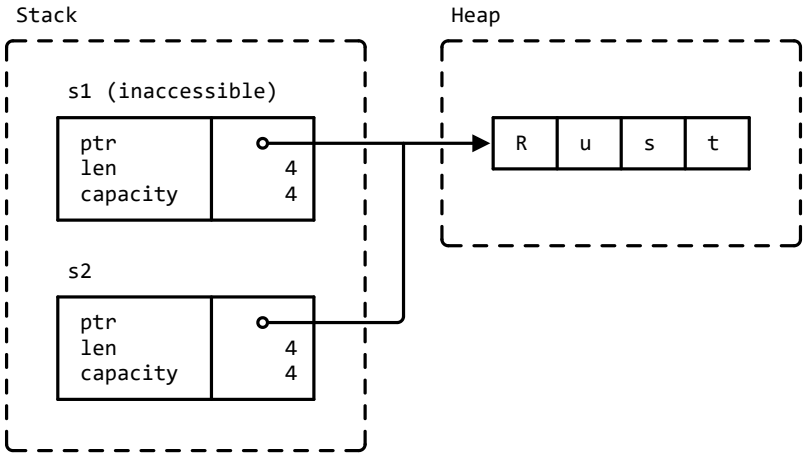
```
fn main() {
    let s1: String = String::from("Rust");
    let s2: String = s1;
}
```

- 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`:



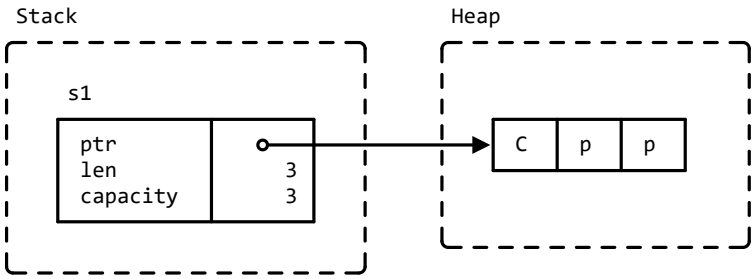
Double Frees in Modern C++

Modern C++ solves this differently:

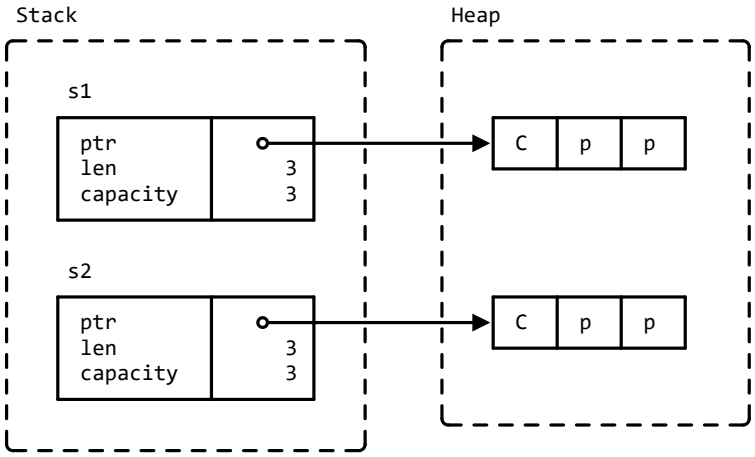
```
std::string s1 = "Cpp";
std::string s2 = s1;  // Duplicate the data in s1.
```

- 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

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

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

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

Lifetimes

A borrowed value has a *lifetime*:

- The lifetime can be implicit: `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 for function arguments and return values must be fully specified, but Rust allows lifetimes to be elided in most cases with a few simple rules.

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

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

Day 1: Afternoon Exercises

We will look at two things:

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

▶ Details

Storing Books

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);
    let midpoint = vec.len() / 2;
    println!("middle value: {}", vec[midpoint]);
    for item in &vec {
        println!("item: {item}");
    }
}
```

Use this to model a library's book collection. Copy the code below to https://play.rust-lang.org/ and update the types to make it compile:

```
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,
        }
    }
}

// Implement the methods below. Update the `self` parameter to
// indicate the method's required level of ownership over the object:
//
// - `&self` for shared read-only access,
// - `&mut self` for unique and mutable access,
// - `self` for unique access by value.
impl Library {
    fn new() -> Library {
        todo!("Initialize and return a `Library` value")
    }

    //fn len(self) -> usize {
    //    todo!("Return the length of `self.books`")
    //}

    //fn is_empty(self) -> bool {
    //    todo!("Return `true` if `self.books` is empty")
    //}

    //fn add_book(self, book: Book) {
    //    todo!("Add a new book to `self.books`")
    //}

    //fn print_books(self) {
    //    todo!("Iterate over `self.books` and each book's title and year")
    //}

    //fn oldest_book(self) -> Option<&Book> {
    //    todo!("Return a reference to the oldest book (if any)")
    //}
}

// 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!("The library is empty: 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));
    //
    //println!("The library is no longer empty: library.is_empty() -> {}",
library.is_empty());
```

```
    //library.print_books();
    //
    //match library.oldest_book() {
    //    Some(book) => println!("The oldest book is {}", book.title),
    //    None => println!("The library is empty!"),
    //}
    //
    //println!("The library has {} books", library.len());
    //library.print_books();
}
```

▶ Details

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

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 PoundsOfForce(f64);
struct Newtons(f64);

fn compute_thruster_force() -> PoundsOfForce {
    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

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

Enums

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

```
fn generate_random_number() -> i32 {
    // Implementation based on https://xkcd.com/221/
    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

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:

```
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

Enum Sizes

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

```
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,
}

fn main() {
    dbg_size!(Foo);
}
```

- See the Rust Reference.

▶ Details

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

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. Complete ownership does not automatically mean mutability.
- `mut self`: same as above, but the method can mutate the object.
- 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

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 s
        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

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

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

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, x = {i:?}"),
        Foo { y, .. }        => println!("y = {y}, other fields were ignored"),
    }
}
```

▶ Details

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

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:

```rust
#[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

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

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:

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

pub 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);
```

```
        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

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 contains a sequence of expressions. Each block has a value and a type, which are those of 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}");
}
```

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

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));
}
```

▶ Details

`if` expressions

You use `if` expressions exactly like `if` statements 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 `if` as an expression. The last expression of each block becomes the value of the `if` expression:

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

▶ Details

`if let` expressions

The `if let` expression lets you execute different code depending on whether a value matches a pattern:

```
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

`while` loops

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` loops

Like with `if let`, 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

for loops

The `for` loop is closely related to the `while let` loop. 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

`loop` expressions

Finally, there is a `loop` keyword which creates an endless loop.

Here you must either `break` or `return` to stop the loop:

```
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}");
}
```

▶ Details

`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.

▶ Details

`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:

```
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

`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

String

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

```
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(" C H ");
    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

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());

    // Canonical macro to initialize a vector with elements.
    let mut v3 = vec![0, 0, 1, 2, 3, 4];

    // Retain only the even elements.
    v3.retain(|x| x % 2 == 0);
    println!("{v3:?}");

    // Remove consecutive duplicates.
    v3.dedup();
    println!("{v3:?}");
}
```

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

▶ Details

HashMap

Standard hash map with protection against HashDoS attacks:

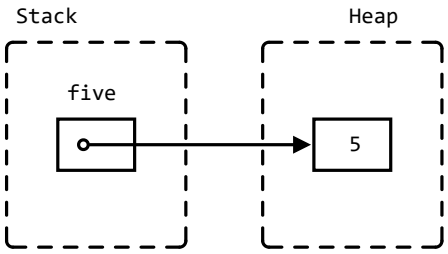
```
use std::collections::HashMap;

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 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.")
        }
    }

    // Use the .entry() method to insert a value if nothing is found.
    for book in ["Pride and Prejudice", "Alice's Adventure in Wonderland"] {
        let page_count: &mut i32 = page_counts.entry(book.to_string()).or_insert(0);
        *page_count += 1;
    }

    println!("{page_counts:#?}");
}
```

▶ Details

Box

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

```
1 fn main() {
2     let five = Box::new(5);
3     println!("five: {}", *five);
4 }
```



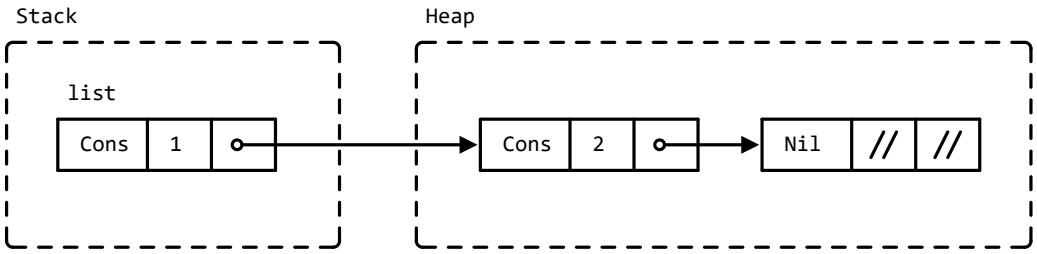
`Box<T>` implements `Deref<Target = T>`, which means that you can call methods from `T` directly on a `Box<T>`.

▶ Details

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:?}");
}
```

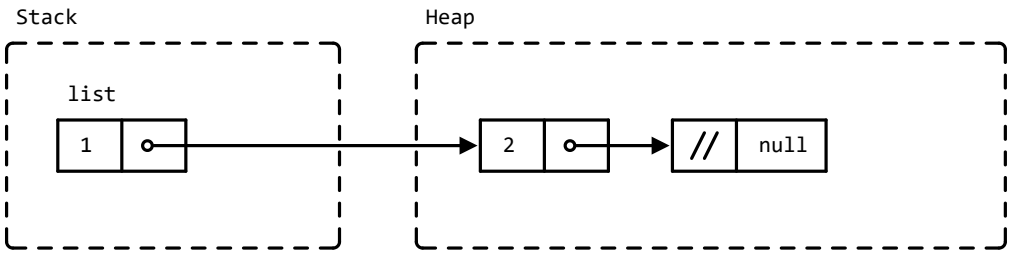


▶ Details

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` 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 = Rc::clone(&a);

    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.
- You can *downgrade* a shared pointer into a `Weak` pointer to create cycles that will get dropped.

▶ Details

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();
}
```

▶ Details

Visibility

Modules are a privacy boundary:

- Module items are private by default (hides implementation details).
- Parent and sibling items are always visible.
- In other words, if an item is visible in module `foo`, it's visible in all the descendants of `foo`.

```
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();
}
```

▶ Details

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.

A module can bring symbols from another module into scope with `use`. You will typically see something like this at the top of each module:

```
use std::collections::HashSet;
use std::mem::transmute;
```

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)

Modules defined in files can be documented, too, using “inner doc comments”. These document the item that contains them – in this case, a module.

```
//! This module implements the garden, including a highly performant germination
//! implementation.

// Re-export types from this module.
pub use seeds::SeedPacket;
pub use garden::Garden;

/// Sow the given seed packets.
pub fn sow(seeds: Vec<SeedPacket>) { todo!() }

/// Harvest the produce in the garden that is ready.
pub fn harvest(garden: &mut Garden) { todo!() }
```

▶ Details

Day 2: Afternoon Exercises

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

▶ Details

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:

```rust
// 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:

```
// 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.

Generics

Rust support generics, which lets you abstract algorithms or data structures (such as sorting or a binary tree) over the types used or stored.

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:?}");
}
```

▶ Details

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

Monomorphization

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

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

behaves as if you wrote

```
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.

Traits

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

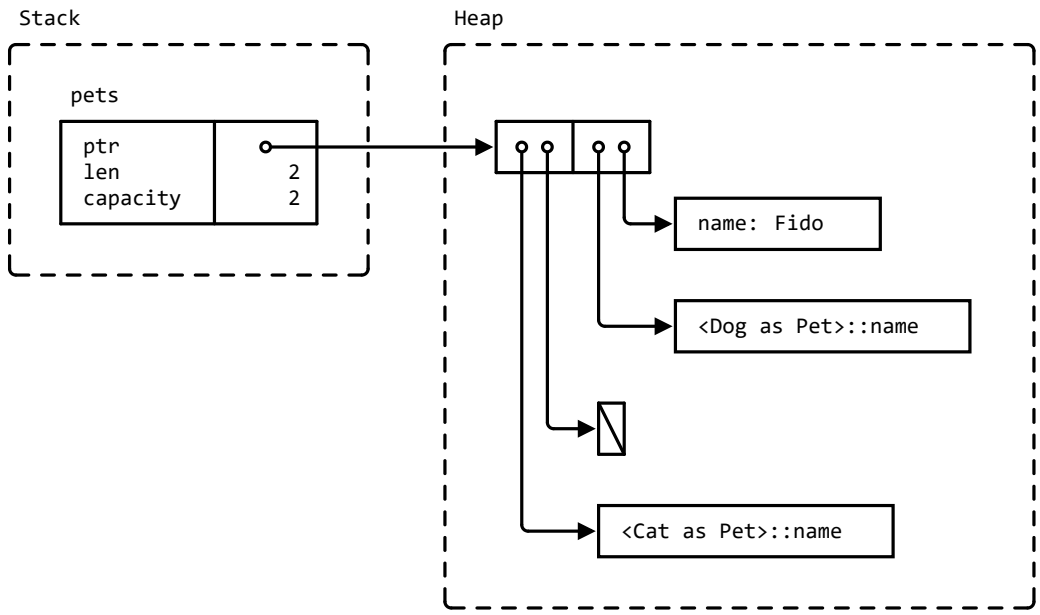
```
trait Pet {
    fn name(&self) -> String;
}

struct Dog {
    name: String,
}

struct Cat;

impl Pet for Dog {
    fn name(&self) -> String {
        self.name.clone()
    }
}

impl Pet for Cat {
    fn name(&self) -> String {
        String::from("The cat") // No name, cats won't respond to it anyway.
    }
}

fn greet<P: Pet>(pet: &P) {
    println!("Who's a cutie? {} is!", pet.name());
}

fn main() {
    let fido = Dog { name: "Fido".into() };
    greet(&fido);

    let captain_floof = Cat;
    greet(&captain_floof);
}
```

Trait Objects

Trait objects allow for values of different types, for instance in a collection:

```
trait Pet {
    fn name(&self) -> String;
}

struct Dog {
    name: String,
}

struct Cat;

impl Pet for Dog {
    fn name(&self) -> String {
        self.name.clone()
    }
}

impl Pet for Cat {
    fn name(&self) -> String {
        String::from("The cat") // No name, cats won't respond to it anyway.
    }
}

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

Memory layout after allocating `pets`:



▶ Details

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));
}
```

▶ Details

Trait Bounds

When working with generics, you often want to require the types to implement some trait, so that you can call this trait's methods.

You can do this with `T: Trait` or `impl Trait`:

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

// Syntactic sugar for:
//   fn add_42_millions<T: Into<i32>>(x: T) -> i32 {
fn add_42_millions(x: impl Into<i32>) -> i32 {
    x.into() + 42_000_000
}

// struct NotClonable;

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

    let many = add_42_millions(42_i8);
    println!("{many}");
    let many_more = add_42_millions(10_000_000);
    println!("{many_more}");
}
```

▶ Details

`impl Trait`

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

```
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` allows you to work with types which you cannot name.

▶ Details

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

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

`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

`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(())
}
```

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

The `Default` Trait

`Default` trait produces a default value for a type.

```
#[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

`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

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

    let mut v = Vec::new();
    let mut accumulate = |x: i32| {
        v.push(x);
        v.iter().sum::<i32>()
    };
    println!("accumulate: {}", apply_with_log(&mut accumulate, 4));
    println!("accumulate: {}", apply_with_log(&mut accumulate, 5));

    let multiply_sum = |x| x * v.into_iter().sum::<i32>();
    println!("multiply_sum: {}", apply_with_log(multiply_sum, 3));
}
```

▶ Details

Day 3: Morning Exercises

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

▶ Details

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:

```
// 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);
    }

    fn inner_width(&self) -> usize {
        std::cmp::max(
            self.title.chars().count(),
            self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
```

```
}


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");
}
```

```
+--------------------------------+
|       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;

fn main() {
    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;
use std::io::Read;

fn main() {
    let file = fs::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

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 handling code:

```
use std::{fs, io};
use std::io::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(err) => return Err(err),
    };

    let mut username = String::new();
    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(err) => Err(err),
    }
}

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

▶ Details

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

```rust
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 {filen
        }
    }
}

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

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::new();
    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

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;
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::new();
    fs::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

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)
        .with_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

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/`:

```
use my_library::init;

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

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

Useful crates for writing tests

Rust comes with only basic support for writing tests.

Here are some additional crates which we recommend for writing tests:

- googletest: Comprehensive test assertion library in the tradition of GoogleTest for C++.
- proptest: Property-based testing for Rust.
- rstest: Support for fixtures and parameterised tests.

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

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 = r1 as *const i32;

    // Safe because r1 and r2 were obtained from references and so are
    // guaranteed to be non-null and properly aligned, the objects underlying
    // the references from which they were obtained are 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

Mutable Static Variables

It is safe to read an immutable static variable:

```
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:

```
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

Unions

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

```
#[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

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!("emoji: {}", emojis.get_unchecked(0..4));
        println!("emoji: {}", emojis.get_unchecked(4..7));
        println!("emoji: {}", emojis.get_unchecked(7..11));
    }

    println!("char count: {}", count_chars(unsafe { emojis.get_unchecked(0..7) }));

    // Not upholding the UTF-8 encoding requirement breaks memory safety!
    // println!("emoji: {}", unsafe { emojis.get_unchecked(0..3) });
    // println!("char count: {}", count_chars(unsafe { emojis.get_unchecked(0..3) }
}

fn count_chars(s: &str) -> usize {
    s.chars().map(|_| 1).sum()
}
```

Writing Unsafe Functions

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

```
1  /// Swaps the values pointed to by the given pointers.
2  ///
3  /// # Safety
4  ///
5  /// The pointers must be valid and properly aligned.
6  unsafe fn swap(a: *mut u8, b: *mut u8) {
7      let temp = *a;
8      *a = *b;
9      *b = temp;
10 }
11
12 fn main() {
13     let mut a = 42;
14     let mut b = 66;
15
16     // Safe because ...
17     unsafe {
18         swap(&mut a, &mut b);
19     }
20
21     println!("a = {}, b = {}", a, b);
22 }
```

▶ Details

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

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(sel
        }
    }
}

// Safe because u32 has a defined representation and no padding.
unsafe impl AsBytes for u32 {}
```

▶ Details

Day 3: Afternoon Exercises

Let us build a safe wrapper for reading directory content!

For this exercise, we suggest using a local dev environment instead of the Playground. This will allow you to run your binary on your own machine.

To get started, follow the running locally instructions.

▶ Details

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. There you find a number of string types which you need for the exercise:

Types	Encoding	Use
`str` and `String`	UTF-8	Text processing in Rust
`CStr` and `CString`	NUL-terminated	Communicating with C functions
`OsStr` and `OsString`	OS-specific	Communicating with the OS

You will convert between all these types:

- `&str` to `CString`: you need to allocate space for a trailing `\0` character,
- `CString` to `*const i8`: you need a pointer to call C functions,
- `*const i8` to `&CStr`: you need something which can find the trailing `\0` character,
- `&CStr` to `&[u8]`: a slice of bytes is the universal interface for "some unknow data",
- `&[u8]` to `&OsStr`: `&OsStr` is a step towards `OsString`, use `OsStrExt` to create it,
- `&OsStr` to `OsString`: you need to clone the data in `&OsStr` to be able to return it and call `readdir` again.

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:

```
// 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};
    #[cfg(not(target_os = "macos"))]
    use std::os::raw::{c_long, c_ulong, c_ushort, c_uchar};

    // 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 according to the Linux man page for readdir(3), where ino_t and
    // off_t are resolved according to the definitions in
    // /usr/include/x86_64-linux-gnu/{sys/types.h, bits/typesizes.h}.
    #[cfg(not(target_os = "macos"))]
    #[repr(C)]
    pub struct dirent {
        pub d_ino: c_ulong,
        pub d_off: c_long,
        pub d_reclen: c_ushort,
        pub d_type: c_uchar,
        pub d_name: [c_char; 256],
    }

    // Layout according to the macOS man page for dir(5).
    #[cfg(all(target_os = "macos"))]
    #[repr(C)]
    pub struct dirent {
        pub d_fileno: u64,
        pub d_seekoff: u64,
        pub d_reclen: u16,
        pub d_namlen: u16,
        pub d_type: u8,
        pub d_name: [c_char; 1024],
    }

    extern "C" {
        pub fn opendir(s: *const c_char) -> *mut DIR;

        #[cfg(not(all(target_os = "macos", target_arch = "x86_64")))]
        pub fn readdir(s: *mut DIR) -> *const dirent;

        // See https://github.com/rust-lang/libc/issues/414 and the section on
        // _DARWIN_FEATURE_64_BIT_INODE in the macOS man page for stat(2).
        //
        // "Platforms that existed before these updates were available" refers
        // to macOS (as opposed to iOS / wearOS / etc.) on Intel and PowerPC.
        #[cfg(all(target_os = "macos", target_arch = "x86_64"))]
        #[link_name = "readdir$INODE64"]
        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,
```

```
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 Rust in 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.

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.

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::IB
irthdayService;
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::Bn
BirthdayService;
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::IB
irthdayService;
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:

```
rust_binary {
    name: "print_birthday_card",
    srcs: ["main.rs"],
    rustlibs: ["libbirthday_bindgen"],
}
```

interoperability/bindgen/main.rs:

```
//! 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:

```
$ 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",
}
```

```
$ 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

```
#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

```
#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:

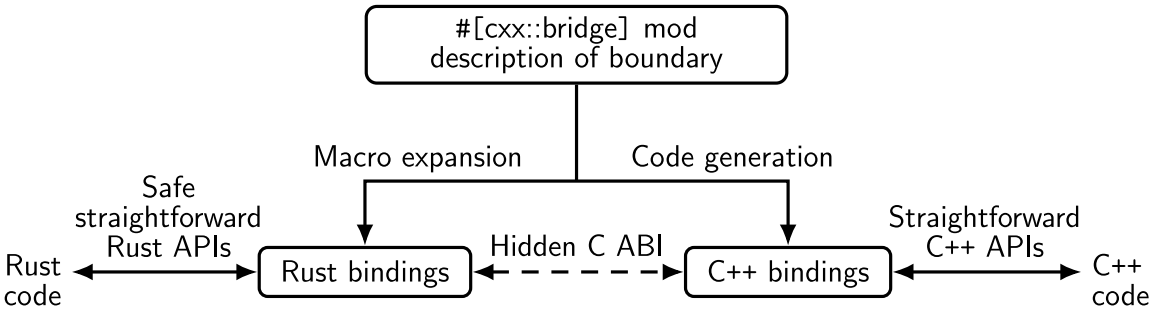
```
$ m analyze_numbers
$ adb push "$ANDROID_PRODUCT_OUT/system/bin/analyze_numbers /data/local/tmp"
$ adb shell /data/local/tmp/analyze_numbers
```

▶ Details

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

This is a group exercise: We will look at one of the projects you work with and try to integrate some Rust into it. Some suggestions:

- Call your AIDL service with a client written in Rust.
- Move a function from your project to Rust and call it.

▶ Details

Welcome to Bare Metal Rust

This is a standalone one-day course about bare-metal Rust, aimed at people who are familiar with the basics of Rust (perhaps from completing the Comprehensive Rust course), and ideally also have some experience with bare-metal programming in some other language such as C.

Today we will talk about 'bare-metal' Rust: running Rust code without an OS underneath us. This will be divided into several parts:

- What is `no_std` Rust?
- Writing firmware for microcontrollers.
- Writing bootloader / kernel code for application processors.
- Some useful crates for bare-metal Rust development.

For the microcontroller part of the course we will use the BBC micro:bit v2 as an example. It's a development board based on the Nordic nRF51822 microcontroller with some LEDs and buttons, an I2C-connected accelerometer and compass, and an on-board SWD debugger.

To get started, install some tools we'll need later. On gLinux or Debian:

```
sudo apt install gcc-aarch64-linux-gnu gdb-multiarch libudev-dev picocom pkg-config
qemu-system-arm
rustup update
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils cargo-embed
```

And give users in the `plugdev` group access to the micro:bit programmer:

```
echo 'SUBSYSTEM=="usb", ATTR{idVendor}=="0d28", MODE="0664", GROUP="plugdev"' |\
  sudo tee /etc/udev/rules.d/50-microbit.rules
sudo udevadm control --reload-rules
```

On MacOS:

```
xcode-select --install
brew install gdb picocom qemu
brew install --cask gcc-aarch64-embedded
rustup update
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils cargo-embed
```

`no_std`

`core`	`alloc`	`std`
• Slices, `&str`, `CStr` • `NonZeroU8` ... • `Option`, `Result` • `Display`, `Debug`, `write!` ... • `Iterator` • `panic!`, `assert_eq!` ... • `NonNull` and all the usual pointer-related functions • `Future` and `async` / `await` • `fence`, `AtomicBool`, `AtomicPtr`, `AtomicU32` ... • `Duration`	• `Box`, `Cow`, `Arc`, `Rc` • `Vec`, `BinaryHeap`, `BtreeMap`, `LinkedList`, `VecDeque` • `String`, `CString`, `format!`	• `Error` • `HashMap` • `Mutex`, `Condvar`, `Barrier`, `Once`, `RwLock`, `mpsc` • `File` and the rest of `fs` • `println!`, `Read`, `Write`, `Stdin`, `Stdout` and the rest of `io` • `Path`, `OsString` • `net` • `Command`, `Child`, `ExitCode` • `spawn`, `sleep` and the rest of `thread` • `SystemTime`, `Instant`

▶ Details

A minimal `no_std` program

```
#![no_main]
#![no_std]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_panic: &PanicInfo) -> ! {
    loop {}
}
```

▶ Details

alloc

To use `alloc` you must implement a global (heap) allocator.

```
#![no_main]
#![no_std]

extern crate alloc;
extern crate panic_halt as _;

use alloc::string::ToString;
use alloc::vec::Vec;
use buddy_system_allocator::LockedHeap;

#[global_allocator]
static HEAP_ALLOCATOR: LockedHeap<32> = LockedHeap::<32>::new();

static mut HEAP: [u8; 65536] = [0; 65536];

pub fn entry() {
    // Safe because `HEAP` is only used here and `entry` is only called once.
    unsafe {
        // Give the allocator some memory to allocate.
        HEAP_ALLOCATOR
            .lock()
            .init(HEAP.as_mut_ptr() as usize, HEAP.len());
    }

    // Now we can do things that require heap allocation.
    let mut v = Vec::new();
    v.push("A string".to_string());
}
```

▶ Details

Microcontrollers

The `cortex_m_rt` crate provides (among other things) a reset handler for Cortex M microcontrollers.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

mod interrupts;

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    loop {}
}
```

Next we'll look at how to access peripherals, with increasing levels of abstraction.

▶ Details

Raw MMIO

Most microcontrollers access peripherals via memory-mapped IO. Let's try turning on an LED on our micro:bit:

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

mod interrupts;

use core::mem::size_of;
use cortex_m_rt::entry;

/// GPIO port 0 peripheral address
const GPIO_P0: usize = 0x5000_0000;

// GPIO peripheral offsets
const PIN_CNF: usize = 0x700;
const OUTSET: usize = 0x508;
const OUTCLR: usize = 0x50c;

// PIN_CNF fields
const DIR_OUTPUT: u32 = 0x1;
const INPUT_DISCONNECT: u32 = 0x1 << 1;
const PULL_DISABLED: u32 = 0x0 << 2;
const DRIVE_S0S1: u32 = 0x0 << 8;
const SENSE_DISABLED: u32 = 0x0 << 16;

#[entry]
fn main() -> ! {
    // Configure GPIO 0 pins 21 and 28 as push-pull outputs.
    let pin_cnf_21 = (GPIO_P0 + PIN_CNF + 21 * size_of::<u32>()) as *mut u32;
    let pin_cnf_28 = (GPIO_P0 + PIN_CNF + 28 * size_of::<u32>()) as *mut u32;
    // Safe because the pointers are to valid peripheral control registers, and
    // no aliases exist.
    unsafe {
        pin_cnf_21.write_volatile(
            DIR_OUTPUT | INPUT_DISCONNECT | PULL_DISABLED | DRIVE_S0S1 | SENSE_DISA
        );
        pin_cnf_28.write_volatile(
            DIR_OUTPUT | INPUT_DISCONNECT | PULL_DISABLED | DRIVE_S0S1 | SENSE_DISA
        );
    }

    // Set pin 28 low and pin 21 high to turn the LED on.
    let gpio0_outset = (GPIO_P0 + OUTSET) as *mut u32;
    let gpio0_outclr = (GPIO_P0 + OUTCLR) as *mut u32;
    // Safe because the pointers are to valid peripheral control registers, and
    // no aliases exist.
    unsafe {
        gpio0_outclr.write_volatile(1 << 28);
        gpio0_outset.write_volatile(1 << 21);
    }

    loop {}
}
```

▶ Details

Peripheral Access Crates

`svd2rust` generates mostly-safe Rust wrappers for memory-mapped peripherals from CMSIS-SVD files.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use nrf52833_pac::Peripherals;

#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();
    let gpio0 = p.P0;

    // Configure GPIO 0 pins 21 and 28 as push-pull outputs.
    gpio0.pin_cnf[21].write(|w| {
        w.dir().output();
        w.input().disconnect();
        w.pull().disabled();
        w.drive().s0s1();
        w.sense().disabled();
        w
    });
    gpio0.pin_cnf[28].write(|w| {
        w.dir().output();
        w.input().disconnect();
        w.pull().disabled();
        w.drive().s0s1();
        w.sense().disabled();
        w
    });

    // Set pin 28 low and pin 21 high to turn the LED on.
    gpio0.outclr.write(|w| w.pin28().clear());
    gpio0.outset.write(|w| w.pin21().set());

    loop {}
}
```

▶ Details

HAL crates

HAL crates for many microcontrollers provide wrappers around various peripherals. These generally implement traits from `embedded-hal`.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use nrf52833_hal::gpio::{p0, Level};
use nrf52833_hal::pac::Peripherals;
use nrf52833_hal::prelude::*;

#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();

    // Create HAL wrapper for GPIO port 0.
    let gpio0 = p0::Parts::new(p.P0);

    // Configure GPIO 0 pins 21 and 28 as push-pull outputs.
    let mut col1 = gpio0.p0_28.into_push_pull_output(Level::High);
    let mut row1 = gpio0.p0_21.into_push_pull_output(Level::Low);

    // Set pin 28 low and pin 21 high to turn the LED on.
    col1.set_low().unwrap();
    row1.set_high().unwrap();

    loop {}
}
```

▶ Details

Board support crates

Board support crates provide a further level of wrapping for a specific board for convenience.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use microbit::hal::prelude::*;
use microbit::Board;

#[entry]
fn main() -> ! {
    let mut board = Board::take().unwrap();

    board.display_pins.col1.set_low().unwrap();
    board.display_pins.row1.set_high().unwrap();

    loop {}
}
```

▶ Details

The type state pattern

```rust
#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();
    let gpio0 = p0::Parts::new(p.P0);

    let pin: P0_01<Disconnected> = gpio0.p0_01;

    // let gpio0_01_again = gpio0.p0_01; // Error, moved.
    let pin_input: P0_01<Input<Floating>> = pin.into_floating_input();
    if pin_input.is_high().unwrap() {
        // ...
    }
    let mut pin_output: P0_01<Output<OpenDrain>> = pin_input
        .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Level::Low);
    pin_output.set_high().unwrap();
    // pin_input.is_high(); // Error, moved.

    let _pin2: P0_02<Output<OpenDrain>> = gpio0
        .p0_02
        .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Level::Low);
    let _pin3: P0_03<Output<PushPull>> = gpio0.p0_03.into_push_pull_output(Level::L

    loop {}
}
```

▶ Details

`embedded-hal`

The `embedded-hal` crate provides a number of traits covering common microcontroller peripherals.

- GPIO
- ADC
- I2C, SPI, UART, CAN
- RNG
- Timers
- Watchdogs

Other crates then implement drivers in terms of these traits, e.g. an accelerometer driver might need an I2C or SPI bus implementation.

▶ Details

`probe-rs`, `cargo-embed`

probe-rs is a handy toolset for embedded debugging, like OpenOCD but better integrated.

- SWD and JTAG via CMSIS-DAP, ST-Link and J-Link probes
- GDB stub and Microsoft DAP server
- Cargo integration

`cargo-embed` is a cargo subcommand to build and flash binaries, log RTT output and connect GDB. It's configured by an `Embed.toml` file in your project directory.

▶ Details

Debugging

Embed.toml:

```
[default.general]
chip = "nrf52833_xxAA"

[debug.gdb]
enabled = true
```

In one terminal under `src/bare-metal/microcontrollers/examples/`:

```
cargo embed --bin board_support debug
```

In another terminal in the same directory:

```
gdb-multiarch target/thumbv7em-none-eabihf/debug/board_support --eval-command="target
remote :1337"
```

▶ Details

Other projects

- RTIC
  - “Real-Time Interrupt-driven Concurrency”
  - Shared resource management, message passing, task scheduling, timer queue
- Embassy
  - `async` executors with priorities, timers, networking, USB
- TockOS
  - Security-focused RTOS with preemptive scheduling and Memory Protection Unit support
- Hubris
  - Microkernel RTOS from Oxide Computer Company with memory protection, unprivileged drivers, IPC
- Bindings for FreeRTOS
- Some platforms have `std` implementations, e.g. esp-idf.

▶ Details

Exercises

We will read the direction from an I2C compass, and log the readings to a serial port.

▶ Details

Compass

We will read the direction from an I2C compass, and log the readings to a serial port. If you have time, try displaying it on the LEDs somehow too, or use the buttons somehow.

Hints:

- Check the documentation for the `lsm303agr` and `microbit-v2` crates, as well as the micro:bit hardware.
- The LSM303AGR Inertial Measurement Unit is connected to the internal I2C bus.
- TWI is another name for I2C, so the I2C master peripheral is called TWIM.
- The LSM303AGR driver needs something implementing the `embedded_hal::blocking::i2c::WriteRead` trait. The `microbit::hal::Twim` struct implements this.
- You have a `microbit::Board` struct with fields for the various pins and peripherals.
- You can also look at the nRF52833 datasheet if you want, but it shouldn't be necessary for this exercise.

Download the exercise template and look in the `compass` directory for the following files.

`src/main.rs`:

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use core::fmt::Write;
use cortex_m_rt::entry;
use microbit::{hal::uarte::{Baudrate, Parity, Uarte}, Board};

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();

    // Configure serial port.
    let mut serial = Uarte::new(
        board.UARTE0,
        board.uart.into(),
        Parity::EXCLUDED,
        Baudrate::BAUD115200,
    );

    // Set up the I2C controller and Inertial Measurement Unit.
    // TODO

    writeln!(serial, "Ready.").unwrap();

    loop {
        // Read compass data and log it to the serial port.
        // TODO
    }
}
```

`Cargo.toml` (you shouldn't need to change this):

```
[workspace]

[package]
name = "compass"
version = "0.1.0"
edition = "2021"
publish = false

[dependencies]
cortex-m-rt = "0.7.3"
embedded-hal = "0.2.6"
lsm303agr = "0.2.2"
microbit-v2 = "0.13.0"
panic-halt = "0.2.0"
```

`Embed.toml` (you shouldn't need to change this):

```
[default.general]
chip = "nrf52833_xxAA"

[debug.gdb]
enabled = true

[debug.reset]
halt_afterwards = true
```

`.cargo/config.toml` (you shouldn't need to change this):

```
[build]
target = "thumbv7em-none-eabihf" # Cortex-M4F

[target.'cfg(all(target_arch = "arm", target_os = "none"))']
rustflags = ["-C", "link-arg=-Tlink.x"]
```

See the serial output on Linux with:

```
picocom --baud 115200 --imap lfcrlf /dev/ttyACM0
```

Or on Mac OS something like (the device name may be slightly different):

```
picocom --baud 115200 --imap lfcrlf /dev/tty.usbmodem14502
```

Use Ctrl+A Ctrl+Q to quit picocom.

Application processors

So far we've talked about microcontrollers, such as the Arm Cortex-M series. Now let's try writing something for Cortex-A. For simplicity we'll just work with QEMU's aarch64 'virt' board.

▶ Details

Getting Ready to Rust

Before we can start running Rust code, we need to do some initialisation.

```
.section .init.entry, "ax"
.global entry
entry:
    /*
     * Load and apply the memory management configuration, ready to enable MMU and
     * caches.
     */
    adrp x30, idmap
    msr ttbr0_el1, x30

    mov_i x30, .Lmairval
    msr mair_el1, x30

    mov_i x30, .Ltcrval
    /* Copy the supported PA range into TCR_EL1.IPS. */
    mrs x29, id_aa64mmfr0_el1
    bfi x30, x29, #32, #4

    msr tcr_el1, x30

    mov_i x30, .Lsctlrval

    /*
     * Ensure everything before this point has completed, then invalidate any
     * potentially stale local TLB entries before they start being used.
     */
    isb
    tlbi vmalle1
    ic iallu
    dsb nsh
    isb

    /*
     * Configure sctlr_el1 to enable MMU and cache and don't proceed until this
     * has completed.
     */
    msr sctlr_el1, x30
    isb

    /* Disable trapping floating point access in EL1. */
    mrs x30, cpacr_el1
    orr x30, x30, #(0x3 << 20)
    msr cpacr_el1, x30
    isb

    /* Zero out the bss section. */
    adr_l x29, bss_begin
    adr_l x30, bss_end
0:  cmp x29, x30
    b.hs 1f
    stp xzr, xzr, [x29], #16
    b 0b

1:  /* Prepare the stack. */
    adr_l x30, boot_stack_end
    mov sp, x30

    /* Set up exception vector. */
    adr x30, vector_table_el1
    msr vbar_el1, x30

    /* Call into Rust code. */
    bl main

    /* Loop forever waiting for interrupts. */
```

```
2:  wfi
    b 2b
```

▶ Details

Inline assembly

Sometimes we need to use assembly to do things that aren't possible with Rust code. For example, to make an HVC to tell the firmware to power off the system:

```
#![no_main]
#![no_std]

use core::arch::asm;
use core::panic::PanicInfo;

mod exceptions;

const PSCI_SYSTEM_OFF: u32 = 0x84000008;

#[no_mangle]
extern "C" fn main(_x0: u64, _x1: u64, _x2: u64, _x3: u64) {
    // Safe because this only uses the declared registers and doesn't do
    // anything with memory.
    unsafe {
        asm!("hvc #0",
            inout("w0") PSCI_SYSTEM_OFF => _,
            inout("w1") 0 => _,
            inout("w2") 0 => _,
            inout("w3") 0 => _,
            inout("w4") 0 => _,
            inout("w5") 0 => _,
            inout("w6") 0 => _,
            inout("w7") 0 => _,
            options(nomem, nostack)
        );
    }

    loop {}
}
```

(If you actually want to do this, use the `smccc` crate which has wrappers for all these functions.)

▶ Details

Volatile memory access for MMIO

- Use `pointer::read_volatile` and `pointer::write_volatile`.
- Never hold a reference.
- `addr_of!` lets you get fields of structs without creating an intermediate reference.

▶ Details

Let's write a UART driver

The QEMU 'virt' machine has a PL011 UART, so let's write a driver for that.

```
const FLAG_REGISTER_OFFSET: usize = 0x18;
const FR_BUSY: u8 = 1 << 3;
const FR_TXFF: u8 = 1 << 5;

/// Minimal driver for a PL011 UART.
#[derive(Debug)]
pub struct Uart {
    base_address: *mut u8,
}

impl Uart {
    /// Constructs a new instance of the UART driver for a PL011 device at the
    /// given base address.
    ///
    /// # Safety
    ///
    /// The given base address must point to the 8 MMIO control registers of a
    /// PL011 device, which must be mapped into the address space of the process
    /// as device memory and not have any other aliases.
    pub unsafe fn new(base_address: *mut u8) -> Self {
        Self { base_address }
    }

    /// Writes a single byte to the UART.
    pub fn write_byte(&self, byte: u8) {
        // Wait until there is room in the TX buffer.
        while self.read_flag_register() & FR_TXFF != 0 {}

        // Safe because we know that the base address points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe {
            // Write to the TX buffer.
            self.base_address.write_volatile(byte);
        }

        // Wait until the UART is no longer busy.
        while self.read_flag_register() & FR_BUSY != 0 {}
    }

    fn read_flag_register(&self) -> u8 {
        // Safe because we know that the base address points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe { self.base_address.add(FLAG_REGISTER_OFFSET).read_volatile() }
    }
}
```

▶ Details