Welcome to Comprehensive Rust

This is a free Rust course developed by the Android team at Google. The course covers the full spectrum of Rust, from basic syntax to advanced topics like generics and error handling.

The latest version of the course can be found at https://google.github.io/comprehensive-rust/. If you are reading somewhere else, please check there for updates.

The course is available in other languages. Select your preferred language in the top right corner of the page or check the Translations page for a list of all available translations.

The course is also available as a PDF.

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.

We call the first four course days Rust Fundamentals.

Building on 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.
- Chromium: a half-day course on using Rust in Chromium-based browsers. This includes interoperability with C++ and how to include third-party crates in Chromium.
- 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

- Learning how to develop macros: please see the Rust Book and Rust by Example instead.

Assumptions

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

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

▼ *Speaker Notes*

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

Running the Course

> This page is for the course instructor.

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

We typically run classes from 9:00 am to 4:00 pm, with a 1 hour lunch break in the middle. This leaves 3 hours for the morning class and 3 hours for the afternoon class. Both sessions contain multiple breaks and time for students to work on exercises.

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 four 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

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!

▼ *Speaker Notes*

Instructor Preparation

- **Go through all the material:** Before teaching the course, make sure you have gone through all the slides and exercises yourself. This will help you anticipate questions and potential difficulties.
- **Prepare for live coding:** The course involves a lot of live coding. Practice the examples and exercises beforehand to ensure you can type them out smoothly during the class. Have the solutions ready in case you get stuck.
- **Familiarize yourself with `mdbook`:** The course is presented using `mdbook`. Knowing how to navigate, search, and use its features will make the presentation smoother.
- **Slice size helper:** Press `Ctrl` + `Alt` + `B` to toggle a visual guide showing the amount of space available when presenting. Expect any content outside of the red box to be hidden initially. Use this as a guide when editing slides. You can also enable it via this link.

Creating a Good Learning Environment

- **Encourage questions:** Reiterate that there are no “stupid” questions. A welcoming atmosphere for questions is crucial for learning.
- **Manage time effectively:** Keep an eye on the schedule, but be flexible. It’s more important that students understand the concepts than sticking rigidly to the timeline.
- **Facilitate group work:** During exercises, encourage students to work together. This can help them learn from each other and feel less stuck.

Course Structure

This page is for the course instructor.

Rust Fundamentals

The first four days make up Rust Fundamentals. The days are fast-paced and we cover a lot of ground!

Course schedule:

- Day 1 Morning (2 hours and 10 minutes, including breaks)

Segment	Duration
Welcome	5 minutes
Hello, World	15 minutes
Types and Values	40 minutes
Control Flow Basics	45 minutes

- Day 1 Afternoon (2 hours and 45 minutes, including breaks)

Segment	Duration
Tuples and Arrays	35 minutes
References	55 minutes
User-Defined Types	1 hour

- Day 2 Morning (2 hours and 50 minutes, including breaks)

Segment	Duration
Welcome	3 minutes
Pattern Matching	50 minutes
Methods and Traits	45 minutes
Generics	50 minutes

- Day 2 Afternoon (2 hours and 50 minutes, including breaks)

Segment	Duration
Standard Library Types	1 hour
Standard Library Traits	1 hour

- Day 3 Morning (2 hours and 20 minutes, including breaks)

Segment	Duration
Welcome	3 minutes
Memory Management	1 hour
Smart Pointers	55 minutes

- Day 3 Afternoon (2 hours and 10 minutes, including breaks)

Segment	Duration
Borrowing	55 minutes
Lifetimes	1 hour and 5 minutes

- Day 4 Morning (2 hours and 50 minutes, including breaks)

Segment	Duration
Welcome	3 minutes
Iterators	55 minutes
Modules	45 minutes
Testing	45 minutes

- Day 4 Afternoon (2 hours and 20 minutes, including breaks)

Segment	Duration
Error Handling	55 minutes
Unsafe Rust	1 hour and 15 minutes

Deep Dives

In addition to the 4-day class on Rust Fundamentals, we cover some more specialized topics:

Rust in Android

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.

Rust in Chromium

The Rust in Chromium deep dive is a half-day course on using Rust as part of the Chromium browser. It includes using Rust in Chromium's `gn` build system, bringing in third-party libraries ("crates") and C++ interoperability.

You will need to be able to build Chromium — a debug, component build is recommended for speed but any build will work. Ensure that you can run the Chromium browser that you've built.

Bare-Metal Rust

The Bare-Metal Rust deep dive is 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 in Rust

The Concurrency in Rust 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
cargo run
```

Course schedule:

Segment	Duration
Threads	30 minutes
Channels	20 minutes
Send and Sync	15 minutes
Shared State	30 minutes
Exercises	1 hour and 10 minutes

- Afternoon (3 hours and 30 minutes, including breaks)

Segment	Duration
Async Basics	40 minutes
Channels and Control Flow	20 minutes
Pitfalls	55 minutes
Exercises	1 hour and 10 minutes

Idiomatic Rust

The Idiomatic Rust deep dive is a 2-day class on Rust idioms and patterns.

You should be familiar with the material in Rust Fundamentals before starting this course.

Course schedule:

- Morning (5 hours and 5 minutes, including breaks)

Segment	Duration
Leveraging the Type System	5 hours and 5 minutes

Unsafe (Work in Progress)

The Unsafe deep dive is a two-day class on the *unsafe* Rust language. It covers the fundamentals of Rust's safety guarantees, the motivation for `unsafe`, review process for `unsafe` code, FFI basics, and building data structures that the borrow checker would normally reject.

Course schedule:

- Day 1 Morning (1 hour, including breaks)

Segment	Duration

Segment	Duration
Motivations	20 minutes
Foundations	25 minutes

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.

▼ *Speaker Notes*

- Mention that these shortcuts are standard for `mdbook` and can be useful when navigating any `mdbook`-generated site.
- You can demonstrate each shortcut live to the students.
- The `s` key for search is particularly useful for quickly finding topics that have been discussed earlier.
- `Ctrl` + `Enter` will be super important for you since you'll do a lot of live coding.

Translations

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

- Brazilian Portuguese by @rastringer, @hugojacob, @joaovicmendes, and @henrif75.
- Chinese (Simplified) by @suetfei, @wnghl, @anlunx, @kongy, @noahdragon, @superwhd, @SketchK, and @nodmp.
- Chinese (Traditional) by @hueich, @victorhsieh, @mingyc, @kuanhungchen, and @johnathan79717.
- Farsi by @DannyRavi, @javad-jafari, @Alix1383, @moaminsharifi , @hamidrezakp and @mehrad77.
- Japanese by @CoinEZ-JPN, @momotaro1105, @HidenoriKobayashi and @kantasv.
- Korean by @keispace, @jiyongp, @jooyunghan, and @namhyung.
- Spanish by @deavid.
- Ukrainian by @git-user-cpp, @yaremam and @reta.

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:

- Arabic by @younies
- Bengali by @raselmandol.
- French by @KookaS, @vcaen and @AdrienBaudemont.
- German by @Throvn and @ronaldfw.
- Italian by @henrythebuilder and @detro.

The full list of translations with their current status is also available either as of their last update or synced to the latest version of the course.

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

▼ *Speaker Notes*

- This is a good opportunity to thank the volunteers who have contributed to the translations.
- If there are students in the class who speak any of the listed languages, you can

- Highlight that the project is open source and contributions are welcome, not just for translations but for the course content itself.

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

Please follow the instructions on https://rustup.rs/.

This will give you the Cargo build tool (`cargo`) and the Rust compiler (`rustc`). You will also get `rustup` , a command line utility that you can use to install different compiler versions.

After installing Rust, you should configure your editor or IDE to work with Rust. Most editors do this by talking to rust-analyzer, which provides auto-completion and jump-to-definition functionality for VS Code, Emacs, Vim/Neovim, and many others. There is also a different IDE available called RustRover.

▼ *Speaker Notes*

- On Debian/Ubuntu, you can install `rustup` via `apt` :

  ```
  sudo apt install rustup
  ```

- On macOS, you can use Homebrew to install Rust, but this may provide an outdated version. Therefore, it is recommended to install Rust from the official site.

The Rust Ecosystem

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

- `rustc` : the Rust compiler that turns `.rs` files into binaries and other intermediate formats.
- `cargo` : the Rust dependency manager and build tool. Cargo knows how to download dependencies, usually 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 are 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.

▼ *Speaker Notes*

Key points:

- Rust has a rapid release schedule with a new release coming out every six weeks. New releases maintain backwards compatibility with old releases — plus they enable new functionality.
- There are three release channels: “stable”, “beta”, and “nightly”.
- New features are being tested on “nightly”, “beta” is what becomes “stable” every six weeks.
- Dependencies can also be resolved from alternative registries, git, folders, and more.
- Rust also has editions: the current edition is Rust 2024. Previous editions were Rust 2015, Rust 2018 and Rust 2021.
    - The editions are allowed to make backwards incompatible changes to the language.
    - To prevent breaking code, editions are opt-in: you select the edition for your crate via the `Cargo.toml` file.
    - To avoid splitting the ecosystem, Rust compilers can mix code written for different editions.

- It might be worth alluding that Cargo itself is an extremely powerful and comprehensive tool. It is capable of many advanced features including but not limited to:
  - Project/package structure
  - workspaces
  - Dev Dependencies and Runtime Dependency management/caching
  - build scripting
  - global installation
  - It is also extensible with sub command plugins as well (such as cargo clippy).
- Read more from the official Cargo Book

Code Samples in This Training

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

Installing Cargo is still encouraged: it will make it easier for you to do the exercises. On the last day, we will do a larger exercise that 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.

▼ *Speaker Notes*

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

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

Running Code Locally with Cargo

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

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

You can use any later version too since Rust maintains backwards compatibility.

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 boilerplate code in `src/main.rs` with your own code. For example, using the example on the previous page, make `src/main.rs` look like

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

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

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

6. Use `cargo check` to quickly check your project for errors, use `cargo build` to compile it without running it. You will find the output in `target/debug/` for a normal debug build. Use `cargo build --release` to produce an optimized release build in `target/release/`.

7. You can add dependencies for your project by editing `Cargo.toml`. When you run `cargo` commands, it will automatically download and compile missing dependencies for you.

▼ *Speaker Notes*

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

Welcome to Day 1

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

- Basic Rust syntax: variables, scalar and compound types, enums, structs, references, functions, and methods.
- Types and type inference.
- Control flow constructs: loops, conditionals, and so on.
- User-defined types: structs and enums.

Schedule

Including 10 minute breaks, this session should take about 2 hours and 10 minutes. It contains:

Segment	Duration
Welcome	5 minutes
Hello, World	15 minutes
Types and Values	40 minutes
Control Flow Basics	45 minutes

▼ *Speaker Notes*

This slide should take about 5 minutes.

Please remind the students that:

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

The idea for the first day is to show the "basic" things in Rust that should have immediate parallels in other languages. The more advanced parts of Rust come on the subsequent days.

exercise solution after the break. The times listed here are a suggestion in order to keep the course on schedule. Feel free to be flexible and adjust as necessary!

Hello, World

This segment should take about 15 minutes. It contains:

Slide	Duration
What is Rust?	10 minutes
Benefits of Rust	3 minutes
Playground	2 minutes

What is Rust?

Rust is a new programming language that 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.

▼ *Speaker Notes*

This slide should take about 10 minutes.

Rust fits in the same area as C++:

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

Benefits of Rust

Some unique selling points of Rust:

- *Compile time memory safety* - whole classes of memory bugs are prevented at compile time
  - No uninitialized variables.
  - No double-frees.
  - No use-after-free.
  - No `NULL` pointers.
  - No forgotten locked mutexes.
  - No data races between threads.
  - No iterator invalidation.
- *No undefined runtime behavior* - what a Rust statement does is never left unspecified
  - Array access is bounds checked.
  - Integer overflow is defined (panic or wrap-around).
- *Modern language features* - as expressive and ergonomic as higher-level languages
  - Enums and pattern matching.
  - Generics.
  - No overhead FFI.
  - Zero-cost abstractions.
  - Great compiler errors.
  - Built-in dependency manager.
  - Built-in support for testing.
  - Excellent Language Server Protocol support.

▼ *Speaker Notes*

This slide should take about 3 minutes.

Do not spend much time here. All of these points will be covered in more depth later.

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

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

and predictable performance like C and C++ (no garbage collector) as well as access to low-level hardware (should you need it).

Playground

The Rust Playground provides an easy way to run short Rust programs, and is the basis for the examples and exercises in this course. Try running the “hello-world” program it starts with. It comes with a few handy features:

- Under “Tools”, use the `rustfmt` option to format your code in the “standard” way.
- Rust has two main “profiles” for generating code: Debug (extra runtime checks, less optimization) and Release (fewer runtime checks, lots of optimization). These are accessible under “Debug” at the top.
- If you’re interested, use “ASM” under “…” to see the generated assembly code.

▼ *Speaker Notes*

This slide should take about 2 minutes.

As students head into the break, encourage them to open up the playground and experiment a little. Encourage them to keep the tab open and try things out during the rest of the course. This is particularly helpful for advanced students who want to know more about Rust’s optimizations or generated assembly.

Types and Values

This segment should take about 40 minutes. It contains:

Slide	Duration
Hello, World	5 minutes
Variables	5 minutes
Values	5 minutes
Arithmetic	3 minutes
Type Inference	3 minutes
Exercise: Fibonacci	15 minutes

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`.
- The `main` function is the entry point of the program.
- Blocks are delimited by curly braces like in C and C++.
- Statements end with `;`.
- `println` is a macro, indicated by the `!` in the invocation.
- Rust strings are UTF-8 encoded and can contain any Unicode character.

▼ *Speaker Notes*

This slide should take about 5 minutes.

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

Key points:

- Rust is very much like other languages in the C/C++/Java tradition. It is imperative and it doesn't try to reinvent things unless absolutely necessary.
- Rust is modern with full support for Unicode.
- Rust uses macros for situations where you want to have a variable number of arguments (no function overloading).
- `println!` is a macro because it needs to handle an arbitrary number of arguments based on the format string, which can't be done with a regular function. Otherwise it can be treated like a regular function.
- Rust is multi-paradigm. For example, it has powerful object-oriented programming features, and, while it is not a functional language, it includes a range of functional concepts.

Variables

Rust provides type safety via static typing. Variable bindings are made with `let`:

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

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Uncomment the `x = 20` to demonstrate that variables are immutable by default. Add the `mut` keyword to allow changes.
- Warnings are enabled for this slide, such as for unused variables or unnecessary `mut`. These are omitted in most slides to avoid distracting warnings. Try removing the mutation but leaving the `mut` keyword in place.
- The `i32` here is the type of the variable. This must be known at compile time, but type inference (covered later) allows the programmer to omit it in many cases.

Values

Here are some basic built-in types, and the syntax for literal values of each type.

	Types	Literals
Signed integers	`i8`, `i16`, `i32`, `i64`, `i128`, `isize`	`-10`, `0`, `1_000`, `123_i64`
Unsigned integers	`u8`, `u16`, `u32`, `u64`, `u128`, `usize`	`0`, `123`, `10_u16`
Floating point numbers	`f32`, `f64`	`3.14`, `-10.0e20`, `2_f32`
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.

▼ *Speaker Notes*

This slide should take about 5 minutes.

There are a few syntaxes that are not shown above:

- All underscores in numbers can be left out, they are for legibility only. So `1_000` can be written as `1000` (or `10_00`), and `123_i64` can be written as `123i64`.

Arithmetic

```
fn interproduct(a: i32, b: i32, c: i32) -> i32 {
    return a * b + b * c + c * a;
}

fn main() {
    println!("result: {}", interproduct(120, 100, 248));
}
```

▼ *Speaker Notes*

This slide should take about 3 minutes.

This is the first time we've seen a function other than `main`, but the meaning should be clear: it takes three integers, and returns an integer. Functions will be covered in more detail later.

Arithmetic is very similar to other languages, with similar precedence.

What about integer overflow? In C and C++ overflow of *signed* integers is actually undefined, and might do unknown things at runtime. In Rust, it's defined.

Change the `i32`'s to `i16` to see an integer overflow, which panics (checked) in a debug build and wraps in a release build. There are other options, such as overflowing, saturating, and carrying. These are accessed with method syntax, e.g., `(a * b).saturating_add(b * c).saturating_add(c * a)`.

In fact, the compiler will detect overflow of constant expressions, which is why the example requires a separate function.

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);
}
```

▼ *Speaker Notes*

This slide should take about 3 minutes.

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

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

When nothing constrains the type of an integer literal, Rust defaults to `i32`. This sometimes appears as `{integer}` in error messages. Similarly, floating-point literals default to `f64`.

```
fn main() {
    let x = 3.14;
    let y = 20;
    assert_eq!(x, y);
    // ERROR: no implementation for `{float} == {integer}`
}
```

Exercise: Fibonacci

The Fibonacci sequence begins with `[0, 1]`. For `n > 1`, the next number is the sum of the previous two.

Write a function `fib(n)` that calculates the nth Fibonacci number. When will this function panic?

```
fn fib(n: u32) -> u32 {
    if n < 2 {
        // The base case.
        return todo!("Implement this");
    } else {
        // The recursive case.
        return todo!("Implement this");
    }
}

fn main() {
    let n = 20;
    println!("fib({n}) = {}", fib(n));
}
```

▼ *Speaker Notes*

This slide and its sub-slides should take about 15 minutes.

- This exercise is a classic introduction to recursion.
- Encourage students to think about the base cases and the recursive step.
- The question “When will this function panic?” is a hint to think about integer overflow. The Fibonacci sequence grows quickly!
- Students might come up with an iterative solution as well, which is a great opportunity to discuss the trade-offs between recursion and iteration (e.g., performance, stack overflow for deep recursion).

Solution

```
fn fib(n: u32) -> u32 {
    if n < 2 {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}

fn main() {
    let n = 20;
    println!("fib({n}) = {}", fib(n));
}
```

▼ *Speaker Notes*

- Walk through the solution step-by-step.
- Explain the recursive calls and how they lead to the final result.
- Discuss the integer overflow issue. With `u32`, the function will panic for `n` around 47. You can demonstrate this by changing the input to `main`.
- Show an iterative solution as an alternative and compare its performance and memory usage with the recursive one. An iterative solution will be much more efficient.

More to Explore

For a more advanced discussion, you can introduce memoization or dynamic programming to optimize the recursive Fibonacci calculation, although this is beyond the scope of the current topic.

Control Flow Basics

This segment should take about 45 minutes. It contains:

Slide	Duration
Blocks and Scopes	5 minutes
if Expressions	4 minutes
match Expressions	5 minutes
Loops	5 minutes
break and continue	4 minutes
Functions	3 minutes
Macros	2 minutes
Exercise: Collatz Sequence	15 minutes

▼ *Speaker Notes*

- We will now cover the many kinds of flow control found in Rust.
- Most of this will be very familiar to what you have seen in other programming languages.

Blocks and Scopes

- A block in Rust contains a sequence of expressions, enclosed by braces `{}`.
- The final expression of a block determines the value and type of the whole block.

```
fn main() {
    let z = 13;
    let x = {
        let y = 10;
        dbg!(y);
        z - y
    };
    dbg!(x);
    // dbg!(y);
}
```

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

A variable's scope is limited to the enclosing block.

▼ *Speaker Notes*

This slide should take about 5 minutes.

- You can explain that dbg! is a Rust macro that prints and returns the value of a given expression for quick and dirty debugging.
- You can show how the value of the block changes by changing the last line in the block. For instance, adding/removing a semicolon or using a `return`.
- Demonstrate that attempting to access `y` outside of its scope won't compile.
- Values are effectively "deallocated" when they go out of their scope, even if their data on the stack is still there.

`if` expressions

You use `if` expressions exactly like `if` statements in other languages:

```
fn main() {
    let x = 10;
    if x == 0 {
        println!("zero!");
    } else if x < 100 {
        println!("biggish");
    } else {
        println!("huge");
    }
}
```

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 x = 10;
    let size = if x < 20 { "small" } else { "large" };
    println!("number size: {}", size);
}
```

▼ *Speaker Notes*

This slide should take about 4 minutes.

Because `if` is an expression and must have a particular type, both of its branch blocks must have the same type. Show what happens if you add `;` after `"small"` in the second example.

An `if` expression should be used in the same way as the other expressions. For example, when it is used in a `let` statement, the statement must be terminated with a `;` as well. Remove the `;` before `println!` to see the compiler error.

`match` Expressions

`match` can be used to check a value against one or more options:

```
fn main() {
    let val = 1;
    match val {
        1 => println!("one"),
        10 => println!("ten"),
        100 => println!("one hundred"),
        _ => {
            println!("something else");
        }
    }
}
```

Like `if` expressions, `match` can also return a value;

```
fn main() {
    let flag = true;
    let val = match flag {
        true => 1,
        false => 0,
    };
    println!("The value of {flag} is {val}");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `match` arms are evaluated from top to bottom, and the first one that matches has its corresponding body executed.
- There is no fall-through between cases the way that `switch` works in other languages.
- The body of a `match` arm can be a single expression or a block. Technically this is the same thing, since blocks are also expressions, but students may not fully understand that symmetry at this point.
- `match` expressions need to be exhaustive, meaning they either need to cover all possible values or they need to have a default case such as `_`. Exhaustiveness is easiest to demonstrate with enums, but enums haven't been introduced yet. Instead we demonstrate matching on a `bool`, which is the simplest primitive type.
- This slide introduces `match` without talking about pattern matching, giving students a chance to get familiar with the syntax without front-loading too much information.

More to Explore

- To further motivate the usage of `match`, you can compare the examples to their equivalents written with `if`. In the second case, matching on a `bool`, an `if {} else {}` block is pretty similar. But in the first example that checks multiple cases, a `match` expression can be more concise than `if {} else if {} else if {} else`.
- `match` also supports match guards, which allow you to add an arbitrary logical condition that will get evaluated to determine if the match arm should be taken. However talking about match guards requires explaining about pattern matching, which we're trying to avoid on this slide.

Loops

There are three looping keywords in Rust: `while`, `loop`, and `for`:

`while`

The `while` keyword works much like in other languages, executing the loop body as long as the condition is true.

```
fn main() {
    let mut x = 200;
    while x >= 10 {
        x = x / 2;
    }
    dbg!(x);
}
```

for

The `for` loop iterates over ranges of values or the items in a collection:

```
fn main() {
    for x in 1..5 {
        dbg!(x);
    }

    for elem in [2, 4, 8, 16, 32] {
        dbg!(elem);
    }
}
```

▼ *Speaker Notes*

- Under the hood `for` loops use a concept called "iterators" to handle iterating over different kinds of ranges/collections. Iterators will be discussed in more detail later.
- Note that the first `for` loop only iterates to `4`. Show the `1..=5` syntax for an inclusive range.

loop

The `loop` statement just loops forever, until a `break`.

```
fn main() {
    let mut i = 0;
    loop {
        i += 1;
        dbg!(i);
        if i > 100 {
            break;
        }
    }
}
```

▼ *Speaker Notes*

- The `loop` statement works like a `while true` loop. Use it for things like servers that will serve connections forever.

`break` and `continue`

If you want to immediately start the next iteration use `continue`.

If you want to exit any kind of loop early, use `break`. With `loop`, this can take an optional expression that becomes the value of the `loop` expression.

```
fn main() {
    let mut i = 0;
    loop {
        i += 1;
        if i > 5 {
            break;
        }
        if i % 2 == 0 {
            continue;
        }
        dbg!(i);
    }
}
```

▼ *Speaker Notes*

This slide and its sub-slides should take about 4 minutes.

Note that `loop` is the only looping construct that can return a non-trivial value. This is because it's guaranteed to only return at a `break` statement (unlike `while` and `for` loops, which can also return when the condition fails).

Labels

Both `continue` and `break` can optionally take a label argument that is used to break out of nested loops:

```
fn main() {
    let s = [[5, 6, 7], [8, 9, 10], [21, 15, 32]];
    let mut elements_searched = 0;
    let target_value = 10;
    'outer: for i in 0..=2 {
        for j in 0..=2 {
            elements_searched += 1;
            if s[i][j] == target_value {
                break 'outer;
            }
        }
    }
    dbg!(elements_searched);
}
```

▼ *Speaker Notes*

- Labeled break also works on arbitrary blocks, e.g.

```
'label: {
    break 'label;
    println!("This line gets skipped");
}
```

Functions

```
fn gcd(a: u32, b: u32) -> u32 {
    if b > 0 { gcd(b, a % b) } else { a }
}

fn main() {
    dbg!(gcd(143, 52));
}
```

▼ *Speaker Notes*

This slide should take about 3 minutes.

- Declaration parameters are followed by a type (the reverse of some programming languages), then a return type.
- The last expression in a function body (or any block) becomes the return value. Simply omit the `;` at the end of the expression. The `return` keyword can be used for early return, but the "bare value" form is idiomatic at the end of a function (refactor `gcd` to use a `return`).
- Some functions have no return value, and return the 'unit type', `()`. The compiler will infer this if the return type is omitted.
- Overloading is not supported – each function has a single implementation.
  - Always takes a fixed number of parameters. Default arguments are not supported. Macros can be used to support variadic functions.
  - Always takes a single set of parameter types. These types can be generic, which will be covered later.

Macros

Macros are expanded into Rust code during compilation, and can take a variable number of arguments. They are distinguished by a `!` at the end. The Rust standard library includes an assortment of useful macros.

- `println!(format, ..)` prints a line to standard output, applying formatting described in `std::fmt`.
- `format!(format, ..)` works just like `println!` but returns the result as a string.
- `dbg!(expression)` logs the value of the expression and returns it.
- `todo!()` marks a bit of code as not-yet-implemented. If executed, it will panic.

```
fn factorial(n: u32) -> u32 {
    let mut product = 1;
    for i in 1..=n {
        product *= dbg!(i);
    }
    product
}

fn fizzbuzz(n: u32) -> u32 {
    todo!()
}

fn main() {
    let n = 4;
    println!("{n}! = {}", factorial(n));
}
```

▼ *Speaker Notes*

This slide should take about 2 minutes.

The takeaway from this section is that these common conveniences exist, and how to use them. Why they are defined as macros, and what they expand to, is not especially critical.

The course does not cover defining macros, but a later section will describe use of derive macros.

More To Explore

There are a number of other useful macros provided by the standard library. Some other examples you can share with students if they want to know more:

- `eprintln!` allows you to print to stderr.

Exercise: Collatz Sequence

The Collatz Sequence is defined as follows, for an arbitrary n_1 greater than zero:

- If n_i is 1, then the sequence terminates at n_i.
- If n_i is even, then $n_{i+1} = n_i / 2$.
- If n_i is odd, then $n_{i+1} = 3 * n_i + 1$.

For example, beginning with $n_1 = 3$:

- 3 is odd, so $n_2 = 3 * 3 + 1 = 10$;
- 10 is even, so $n_3 = 10 / 2 = 5$;
- 5 is odd, so $n_4 = 3 * 5 + 1 = 16$;
- 16 is even, so $n_5 = 16 / 2 = 8$;
- 8 is even, so $n_6 = 8 / 2 = 4$;
- 4 is even, so $n_7 = 4 / 2 = 2$;
- 2 is even, so $n_8 = 1$; and
- the sequence terminates.

Write a function to calculate the length of the Collatz sequence for a given initial `n`.

```
/// Determine the length of the collatz sequence beginning at `n`.
fn collatz_length(mut n: i32) -> u32 {
  todo!("Implement this")
}

fn main() {
    println!("Length: {}", collatz_length(11)); // should be 15
}
```

Solution

```
/// Determine the length of the collatz sequence beginning at `n`.
fn collatz_length(mut n: i32) -> u32 {
    let mut len = 1;
    while n > 1 {
        n = if n % 2 == 0 { n / 2 } else { 3 * n + 1 };
        len += 1;
    }
    len
}

fn main() {
    println!("Length: {}", collatz_length(11)); // should be 15
}
```

▼ *Speaker Notes*

- Note that the argument `n` is marked as `mut`, allowing you to change the value of `n` in the function. Like variables, function arguments are immutable by default and you must add `mut` if you want to modify their value. This does not affect how the function is called or how the argument is passed in.

Welcome Back

Including 10 minute breaks, this session should take about 2 hours and 45 minutes. It contains:

Segment	Duration
Tuples and Arrays	35 minutes
References	55 minutes
User-Defined Types	1 hour

Tuples and Arrays

This segment should take about 35 minutes. It contains:

Slide	Duration
Arrays	5 minutes
Tuples	5 minutes
Array Iteration	3 minutes
Patterns and Destructuring	5 minutes
Exercise: Nested Arrays	15 minutes

▼ *Speaker Notes*

- We have seen how primitive types work in Rust. Now it's time for you to start building new composite types.

Arrays

```
fn main() {
    let mut a: [i8; 5] = [5, 4, 3, 2, 1];
    a[2] = 0;
    println!("a: {a:?}");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Arrays can also be initialized using the shorthand syntax, e.g. `[0; 1024]`. This can be useful when you want to initialize all elements to the same value, or if you have a large array that would be hard to initialize manually.
- A value of the array type `[T; N]` holds `N` (a compile-time constant) elements of the same type `T`. Note that the length of the array is *part of its type*, which means that `[u8; 3]` and `[u8; 4]` are considered two different types. Slices, which have a size determined at runtime, are covered later.
- Try accessing an out-of-bounds array element. The compiler is able to determine that the index is unsafe, and will not compile the code:

```
fn main() {
    let mut a: [i8; 5] = [5, 4, 3, 2, 1];
    a[6] = 0;
    println!("a: {a:?}");
}
```

- Array accesses are checked at runtime. Rust can usually optimize these checks away; meaning if the compiler can prove the access is safe, it removes the runtime check for better performance. They can be avoided using unsafe Rust. The optimization is so good that it's hard to give an example of runtime checks failing. The following code will compile but panic at runtime:

```
fn get_index() -> usize {
    6
}

fn main() {
    let mut a: [i8; 5] = [5, 4, 3, 2, 1];
    a[get_index()] = 0;
    println!("a: {a:?}");
}
```

- We can use literals to assign values to arrays.

expect if they come from a garbage-collected language, where arrays may be heap allocated by default.

- There is no way to remove elements from an array, nor add elements to an array. The length of an array is fixed at compile-time, and so its length cannot change at runtime.

Debug Printing

- The `println!` macro asks for the debug implementation with the `?` format parameter: `{}` gives the default output, `{:?}` gives the debug output. Types such as integers and strings implement the default output, but arrays only implement the debug output. This means that we must use debug output here.
- Adding `#`, eg `{a:#?}`, invokes a "pretty printing" format, which can be easier to read.

Tuples

```
1  fn main() {
2      let t: (i8, bool) = (7, true);
3      dbg!(t.0);
4      dbg!(t.1);
5  }
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Like arrays, tuples have a fixed length.
- Tuples group together values of different types into a compound type.
- Fields of a tuple can be accessed by the period and the index of the value, e.g. `t.0`, `t.1`.
- The empty tuple `()` is referred to as the "unit type" and signifies absence of a return value, akin to `void` in other languages.
- Unlike arrays, tuples cannot be used in a `for` loop. This is because a `for` loop requires all the elements to have the same type, which may not be the case for a tuple.
- There is no way to add or remove elements from a tuple. The number of elements and their types are fixed at compile time and cannot be changed at runtime.

Array Iteration

The `for` statement supports iterating over arrays (but not tuples).

```
fn main() {
    let primes = [2, 3, 5, 7, 11, 13, 17, 19];
    for prime in primes {
        for i in 2..prime {
            assert_ne!(prime % i, 0);
        }
    }
}
```

▼ *Speaker Notes*

This slide should take about 3 minutes.

This functionality uses the `IntoIterator` trait, but we haven't covered that yet.

The `assert_ne!` macro is new here. There are also `assert_eq!` and `assert!` macros. These are always checked, while debug-only variants like `debug_assert!` compile to nothing in release builds.

Patterns and Destructuring

Rust supports using pattern matching to destructure a larger value like a tuple into its constituent parts:

```
fn check_order(tuple: (i32, i32, i32)) -> bool {
    let (left, middle, right) = tuple;
    left < middle && middle < right
}

fn main() {
    let tuple = (1, 5, 3);
    println!(
        "{tuple:?}: {}",
        if check_order(tuple) { "ordered" } else { "unordered" }
    );
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- The patterns used here are “irrefutable”, meaning that the compiler can statically verify that the value on the right of `=` has the same structure as the pattern.
- A variable name is an irrefutable pattern that always matches any value, hence why we can also use `let` to declare a single variable.
- Rust also supports using patterns in conditionals, allowing for equality comparison and destructuring to happen at the same time. This form of pattern matching will be discussed in more detail later.
- Edit the examples above to show the compiler error when the pattern doesn’t match the value being matched on.

Exercise: Nested Arrays

Arrays can contain other arrays:

```
let array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
```

What is the type of this variable?

Use an array such as the above to write a function `transpose` that transposes a matrix (turns rows into columns):

Copy the code below to https://play.rust-lang.org/ and implement the function. This function only operates on 3x3 matrices.

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

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

    println!("Original:");
    for row in &matrix {
        println!("{:?}", row);
    }

    let transposed = transpose(matrix);

    println!("\nTransposed:");
    for row in &transposed {
        println!("{:?}", row);
    }
}
```

Solution

```rust
fn transpose(matrix: [[i32; 3]; 3]) -> [[i32; 3]; 3] {
    let mut result = [[0; 3]; 3];
    for i in 0..3 {
        for j in 0..3 {
            result[j][i] = matrix[i][j];
        }
    }
    result
}

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

    println!("Original:");
    for row in &matrix {
        println!("{:?}", row);
    }

    let transposed = transpose(matrix);

    println!("\nTransposed:");
    for row in &transposed {
        println!("{:?}", row);
    }
}
```

References

This segment should take about 55 minutes. It contains:

Slide	Duration
Shared References	10 minutes
Exclusive References	5 minutes
Slices	10 minutes
Strings	10 minutes
Reference Validity	3 minutes
Exercise: Geometry	20 minutes

Shared References

A reference provides a way to access another value without taking ownership of the value, and is also called "borrowing". Shared references are read-only, and the referenced data cannot change.

```
fn main() {
    let a = 'A';
    let b = 'B';

    let mut r: &char = &a;
    dbg!(r);

    r = &b;
    dbg!(r);
}
```

A shared reference to a type `T` has type `&T`. A reference value is made with the `&` operator. The `*` operator "dereferences" a reference, yielding its value.

▼ *Speaker Notes*

This slide should take about 7 minutes.

- References can never be null in Rust, so null checking is not necessary.
- A reference is said to "borrow" the value it refers to, and this is a good model for students not familiar with pointers: code can use the reference to access the value, but is still "owned" by the original variable. The course will get into more detail on ownership in day 3.
- References are implemented as pointers, and a key advantage is that they can be much smaller than the thing they point to. Students familiar with C or C++ will recognize references as pointers. Later parts of the course will cover how Rust prevents the memory-safety bugs that come from using raw pointers.
- Explicit referencing with `&` is usually required. However, Rust performs automatic referencing and dereferencing when invoking methods.
- Rust will auto-dereference in some cases, in particular when invoking methods (try `r.is_ascii()`). There is no need for an `->` operator like in C++.
- In this example, `r` is mutable so that it can be reassigned (`r = &b`). Note that this re-binds `r`, so that it refers to something else. This is different from C++, where assignment to a reference changes the referenced value.

- Rust is tracking the lifetimes of all references to ensure they live long enough. Dangling references cannot occur in safe Rust.
- We will talk more about borrowing and preventing dangling references when we get to ownership.

Exclusive References

Exclusive references, also known as mutable references, allow changing the value they refer to. They have type `&mut T`.

```
fn main() {
    let mut point = (1, 2);
    let x_coord = &mut point.0;
    *x_coord = 20;
    println!("point: {point:?}");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

Key points:

- "Exclusive" means that only this reference can be used to access the value. No other references (shared or exclusive) can exist at the same time, and the referenced value cannot be accessed while the exclusive reference exists. Try making an `&point.0` or changing `point.0` while `x_coord` is alive.
- Be sure to note the difference between `let mut x_coord: &i32` and `let x_coord: &mut i32`. The first one is a shared reference that can be bound to different values, while the second is an exclusive reference to a mutable value.

Slices

A slice gives you a view into a larger collection:

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

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

- Slices borrow data from the sliced type.

▼ *Speaker Notes*

This slide should take about 7 minutes.

- We create a slice by borrowing `a` and specifying the starting and ending indexes in brackets.
- If the slice starts at index 0, Rust's range syntax allows us to drop the starting index, meaning that `&a[0..a.len()]` and `&a[..a.len()]` are identical.
- The same is true for the last index, so `&a[2..a.len()]` and `&a[2..]` are identical.
- To easily create a slice of the full array, we can therefore use `&a[..]`.
- `s` is a reference to a slice of `i32` s. Notice that the type of `s` (`&[i32]`) no longer mentions the array length. This allows us to perform computation on slices of different sizes.
- Slices always borrow from another object. In this example, `a` has to remain 'alive' (in scope) for at least as long as our slice.
- You can't "grow" a slice once it's created:
  - You can't append elements of the slice, since it doesn't own the backing buffer.
  - You can't grow a slice to point to a larger section of the backing buffer. A slice does not have information about the length of the underlying buffer and so you can't know how large the slice can be grown.
  - To get a larger slice you have to back to the original buffer and create a larger slice from there.

Strings

We can now understand the two string types in Rust:

- `&str` is a slice of UTF-8 encoded bytes, similar to `&[u8]`.
- `String` is an owned buffer of UTF-8 encoded bytes, similar to `Vec<T>`.

```
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[2..9];
    println!("s3: {s3}");
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

- `&str` introduces a string slice, which is an immutable reference to UTF-8 encoded string data stored in a block of memory. String literals (`"Hello"`), are stored in the program's binary.
- Rust's `String` type is a wrapper around a vector of bytes. As with a `Vec<T>`, it is owned.
- As with many other types `String::from()` creates a string from a string literal; `String::new()` creates a new empty string, to which string data can be added using the `push()` and `push_str()` methods.
- The `format!()` macro is a convenient way to generate an owned string from dynamic values. It accepts the same format specification as `println!()`.
- You can borrow `&str` slices from `String` via `&` and optionally range selection. If you select a byte range that is not aligned to character boundaries, the expression will panic. The `chars` iterator iterates over characters and is preferred over trying to get character boundaries right.
- For C++ programmers: think of `&str` as `std::string_view` from C++, but the one that always points to a valid string in memory. Rust `String` is a rough equivalent of

- Byte strings literals allow you to create a `&[u8]` value directly:

```
fn main() {
    println!("{:?}", b"abc");
    println!("{:?}", &[97, 98, 99]);
}
```

- Raw strings allow you to create a `&str` value with escapes disabled: `r"\n" == "\\n"`. You can embed double-quotes by using an equal amount of `#` on either side of the quotes:

```
fn main() {
    println!(r#"<a href="link.html">link</a>"#);
    println!("<a href=\"link.html\">link</a>");
}
```

Reference Validity

Rust enforces a number of rules for references that make them always safe to use. One rule is that references can never be `null`, making them safe to use without `null` checks. The other rule we'll look at for now is that references can't *outlive* the data they point to.

```
fn main() {
    let x_ref = {
        let x = 10;
        &x
    };
    dbg!(x_ref);
}
```

▼ *Speaker Notes*

This slide should take about 3 minutes.

- This slide gets students thinking about references as not simply being pointers, since Rust has different rules for references than other languages.
- We'll look at the rest of Rust's borrowing rules on day 3 when we talk about Rust's ownership system.

More to Explore

- Rust's equivalent of nullability is the `Option` type, which can be used to make any type "nullable" (not just references/pointers). We haven't yet introduced enums or pattern matching, though, so try not to go into too much detail about this here.

Exercise: Geometry

We will create a few utility functions for 3-dimensional geometry, representing a point as `[f64;3]`. It is up to you to determine the function signatures.

```
// Calculate the magnitude of a vector by summing the squares of its coordir
// and taking the square root. Use the `sqrt()` method to calculate the squa
// root, like `v.sqrt()`.


fn magnitude(...) -> f64 {
    todo!()
}

// Normalize a vector by calculating its magnitude and dividing all of its
// coordinates by that magnitude.


fn normalize(...) {
    todo!()
}

// Use the following `main` to test your work.

fn main() {
    println!("Magnitude of a unit vector: {}", magnitude(&[0.0, 1.0, 0.0]));

    let mut v = [1.0, 2.0, 9.0];
    println!("Magnitude of {v:?}: {}", magnitude(&v));
    normalize(&mut v);
    println!("Magnitude of {v:?} after normalization: {}", magnitude(&v));
}
```

Solution

```rust
/// Calculate the magnitude of the given vector.
fn magnitude(vector: &[f64; 3]) -> f64 {
    let mut mag_squared = 0.0;
    for coord in vector {
        mag_squared += coord * coord;
    }
    mag_squared.sqrt()
}

/// Change the magnitude of the vector to 1.0 without changing its directior
fn normalize(vector: &mut [f64; 3]) {
    let mag = magnitude(vector);
    for item in vector {
        *item /= mag;
    }
}

fn main() {
    println!("Magnitude of a unit vector: {}", magnitude(&[0.0, 1.0, 0.0]));

    let mut v = [1.0, 2.0, 9.0];
    println!("Magnitude of {v:?}: {}", magnitude(&v));
    normalize(&mut v);
    println!("Magnitude of {v:?} after normalization: {}", magnitude(&v));
}
```

▼ *Speaker Notes*

- Note that in `normalize` we were able to do `*item /= mag` to modify each element. This is because we're iterating using a mutable reference to an array, which causes the `for` loop to give mutable references to each element.
- It is also possible to take slice references here, e.g., `fn magnitude(vector: &[f64]) -> f64`. This makes the function more general, at the cost of a runtime length check.

User-Defined Types

This segment should take about 1 hour. It contains:

Slide	Duration
Named Structs	10 minutes
Tuple Structs	10 minutes
Enums	5 minutes
Type Aliases	2 minutes
Const	10 minutes
Static	5 minutes
Exercise: Elevator Events	15 minutes

Named Structs

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

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

fn describe(person: &Person) {
    println!("{} is {} years old", person.name, person.age);
}

fn main() {
    let mut peter = Person {
        name: String::from("Peter"),
        age: 27,
    };
    describe(&peter);

    peter.age = 28;
    describe(&peter);

    let name = String::from("Avery");
    let age = 39;
    let avery = Person { name, age };
    describe(&avery);
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

Key Points:

- Structs work like in C or C++.
  - Like in C++, and unlike in C, no typedef is needed to define a type.
  - Unlike in C++, there is no inheritance between structs.
- This may be a good time to let people know there are different types of structs.
  - Zero-sized structs (e.g. `struct Foo;`) might be used when implementing a trait on some type but don't have any data that you want to store in the value itself.
  - The next slide will introduce Tuple structs, used when the field names are not important.
- If you already have variables with the right names, then you can create the struct using a shorthand.
- Struct fields do not support default values. Default values are specified by implementing the `Default` trait which we will cover later.

More to Explore

- You can also demonstrate the struct update syntax here:

```
let jackie = Person { name: String::from("Jackie"), ..avery };
```

- It allows us to copy the majority of the fields from the old struct without having to explicitly type it all out. It must always be the last element.
- It is mainly used in combination with the `Default` trait. We will talk about struct update syntax in more detail on the slide on the `Default` trait, so we don't need to talk about it here unless students ask about it.

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);
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

- Newtypes are a great way to encode additional information about the value in a primitive type, for example:
  - The number is measured in some units: `Newtons` in the example above.
  - The value passed some validation when it was created, so you no longer have to validate it again at every use: `PhoneNumber(String)` or `OddNumber(u32)`.
- The newtype pattern is covered extensively in the "Idiomatic Rust" module.
- Demonstrate how to add a `f64` value to a `Newtons` type by accessing the single field in the newtype.
  - Rust generally avoids implicit conversions, like automatic unwrapping or using booleans as integers.
    - Operator overloading is discussed on Day 2 (Standard Library Traits).
- When a tuple struct has zero fields, the `()` can be omitted. The result is a zero-sized type (ZST), of which there is only one value (the name of the type).
  - This is common for types that implement some behavior but have no data

- The example is a subtle reference to the Mars Climate Orbiter failure.

Enums

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

```
#[derive(Debug)]
enum Direction {
    Left,
    Right,
}

#[derive(Debug)]
enum PlayerMove {
    Pass,                          // Simple variant
    Run(Direction),                // Tuple variant
    Teleport { x: u32, y: u32 }, // Struct variant
}

fn main() {
    let dir = Direction::Left;
    let player_move: PlayerMove = PlayerMove::Run(dir);
    println!("On this turn: {player_move:?}");
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

Key Points:

- Enumerations allow you to collect a set of values under one type.
- `Direction` is a type with variants. There are two values of `Direction`: `Direction::Left` and `Direction::Right`.
- `PlayerMove` is a type with three variants. In addition to the payloads, Rust will store a discriminant so that it knows at runtime which variant is in a `PlayerMove` value.
- This might be a good time to compare structs and enums:
  - In both, you can have a simple version without fields (unit struct) or one with different types of fields (variant payloads).
  - You could even implement the different variants of an enum with separate structs but then they wouldn't be the same type as they would if they were all defined in an enum.
- Rust uses minimal space to store the discriminant.
  - If necessary, it stores an integer of the smallest required size
  - If the allowed variant values do not cover all bit patterns, it will use invalid bit patterns to encode the discriminant (the "niche optimization"). For example, `Option<&u8>` stores either a pointer to an integer or `NULL` for the `None` variant.

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

fn main() {
    println!("A: {}", Bar::A as u32);
    println!("B: {}", Bar::B as u32);
    println!("C: {}", Bar::C as u32);
}
```

Without `repr`, the discriminant type takes 2 bytes, because 10001 fits 2 bytes.

More to Explore

Rust has several optimizations it can employ to make enums take up less space.

- Null pointer optimization: For some types, Rust guarantees that `size_of::<T>()` equals `size_of::<Option<T>>()`.

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

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

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

fn main() {
    unsafe {
        println!("bool:");
        dbg_bits!(false, u8);
        dbg_bits!(true, u8);

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

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

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

Type Aliases

A type alias creates a name for another type. The two types can be used interchangeably.

```
enum CarryableConcreteItem {
    Left,
    Right,
}

type Item = CarryableConcreteItem;

// Aliases are more useful with long, complex types:
use std::cell::RefCell;
use std::sync::{Arc, RwLock};
type PlayerInventory = RwLock<Vec<Arc<RefCell<Item>>>>;
```

▼ *Speaker Notes*

This slide should take about 2 minutes.

- A newtype is often a better alternative since it creates a distinct type. Prefer `struct InventoryCount(usize)` to `type InventoryCount = usize`.
- C programmers will recognize this as similar to a `typedef`.

const

Constants are evaluated at compile time and their values are inlined wherever they are used:

```
const DIGEST_SIZE: usize = 3;
const FILL_VALUE: u8 = calculate_fill_value();

const fn calculate_fill_value() -> u8 {
    if DIGEST_SIZE < 10 { 42 } else { 13 }
}

fn compute_digest(text: &str) -> [u8; DIGEST_SIZE] {
    let mut digest = [FILL_VALUE; 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:?}");
}
```

Only functions marked `const` can be called at compile time to generate `const` values. `const` functions can however be called at runtime.

▼ *Speaker Notes*

This slide should take about 10 minutes.

- Mention that `const` behaves semantically similar to C++'s `constexpr`

static

Static variables will live during the whole execution of the program, and therefore will not move:

```
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. When a globally-scoped value does not have a reason to need object identity, `const` is generally preferred.

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `static` is similar to mutable global variables in C++.
- `static` provides object identity: an address in memory and state as required by types with interior mutability such as `Mutex<T>`.

More to Explore

Because `static` variables are accessible from any thread, they must be `Sync`. Interior mutability is possible through a `Mutex`, atomic or similar.

It is common to use `OnceLock` in a static as a way to support initialization on first use. `OnceCell` is not `Sync` and thus cannot be used in this context.

Thread-local data can be created with the macro `std::thread_local`.

Exercise: Elevator Events

We will create a data structure to represent an event in an elevator control system. It is up to you to define the types and functions to construct various events. Use `#[derive(Debug)]` to allow the types to be formatted with `{:?}`.

This exercise only requires creating and populating data structures so that `main` runs without errors. The next part of the course will cover getting data out of these structures.

```rust
#![allow(dead_code)]

#[derive(Debug)]
/// An event in the elevator system that the controller must react to.
enum Event {
    // TODO: add required variants
}

/// A direction of travel.
#[derive(Debug)]
enum Direction {
    Up,
    Down,
}

/// The car has arrived on the given floor.
fn car_arrived(floor: i32) -> Event {
    todo!()
}

/// The car doors have opened.
fn car_door_opened() -> Event {
    todo!()
}

/// The car doors have closed.
fn car_door_closed() -> Event {
    todo!()
}

/// A directional button was pressed in an elevator lobby on the given floor
fn lobby_call_button_pressed(floor: i32, dir: Direction) -> Event {
    todo!()
}

/// A floor button was pressed in the elevator car.
fn car_floor_button_pressed(floor: i32) -> Event {
    todo!()
}

fn main() {
    println!(
        "A ground floor passenger has pressed the up button: {:?}",
        lobby_call_button_pressed(0, Direction::Up)
    );
    println!("The car has arrived on the ground floor: {:?}", car_arrived(0)
    println!("The car door opened: {:?}", car_door_opened());
    println!(
        "A passenger has pressed the 3rd floor button: {:?}",
        car_floor_button_pressed(3)
    );
    println!("The car door closed: {:?}", car_door_closed());
    println!("The car has arrived on the 3rd floor: {:?}", car_arrived(3));
}
```

- If students ask about `#![allow(dead_code)]` at the top of the exercise, it's necessary because the only thing we do with the `Event` type is print it out. Due to a nuance of how the compiler checks for dead code this causes it to think the code is unused. They can ignore it for the purpose of this exercise.

Solution

```rust
#![allow(dead_code)]

#[derive(Debug)]
/// An event in the elevator system that the controller must react to.
enum Event {
    /// A button was pressed.
    ButtonPressed(Button),

    /// The car has arrived at the given floor.
    CarArrived(Floor),

    /// The car's doors have opened.
    CarDoorOpened,

    /// The car's doors have closed.
    CarDoorClosed,
}

/// A floor is represented as an integer.
type Floor = i32;

/// A direction of travel.
#[derive(Debug)]
enum Direction {
    Up,
    Down,
}

/// A user-accessible button.
#[derive(Debug)]
enum Button {
    /// A button in the elevator lobby on the given floor.
    LobbyCall(Direction, Floor),

    /// A floor button within the car.
    CarFloor(Floor),
}

/// The car has arrived on the given floor.
fn car_arrived(floor: i32) -> Event {
    Event::CarArrived(floor)
}

/// The car doors have opened.
fn car_door_opened() -> Event {
    Event::CarDoorOpened
}

/// The car doors have closed.
fn car_door_closed() -> Event {
    Event::CarDoorClosed
}
```

```
}

/// A floor button was pressed in the elevator car.
fn car_floor_button_pressed(floor: i32) -> Event {
    Event::ButtonPressed(Button::CarFloor(floor))
}

fn main() {
    println!(
        "A ground floor passenger has pressed the up button: {:?}",
        lobby_call_button_pressed(0, Direction::Up)
    );
    println!("The car has arrived on the ground floor: {:?}", car_arrived(0)
    println!("The car door opened: {:?}", car_door_opened());
    println!(
        "A passenger has pressed the 3rd floor button: {:?}",
        car_floor_button_pressed(3)
    );
    println!("The car door closed: {:?}", car_door_closed());
    println!("The car has arrived on the 3rd floor: {:?}", car_arrived(3));
}
```

Welcome to Day 2

Now that we have seen a fair amount of Rust, today will focus on Rust's type system:

- Pattern matching: extracting data from structures.
- Methods: associating functions with types.
- Traits: behaviors shared by multiple types.
- Generics: parameterizing types on other types.
- Standard library types and traits: a tour of Rust's rich standard library.
- Closures: function pointers with data.

Schedule

Including 10 minute breaks, this session should take about 2 hours and 50 minutes. It contains:

Segment	Duration
Welcome	3 minutes
Pattern Matching	50 minutes
Methods and Traits	45 minutes
Generics	50 minutes

Pattern Matching

This segment should take about 50 minutes. It contains:

Slide	Duration
Irrefutable Patterns	5 minutes
Matching Values	10 minutes
Destructuring Structs	4 minutes
Destructuring Enums	4 minutes
Let Control Flow	10 minutes
Exercise: Expression Evaluation	15 minutes

Irrefutable Patterns

In day 1 we briefly saw how patterns can be used to *destructure* compound values. Let's review that and talk about a few other things patterns can express:

```
fn takes_tuple(tuple: (char, i32, bool)) {
    let a = tuple.0;
    let b = tuple.1;
    let c = tuple.2;

    // This does the same thing as above.
    let (a, b, c) = tuple;

    // Ignore the first element, only bind the second and third.
    let (_, b, c) = tuple;

    // Ignore everything but the last element.
    let (.., c) = tuple;
}

fn main() {
    takes_tuple(('a', 777, true));
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- All of the demonstrated patterns are *irrefutable*, meaning that they will always match the value on the right hand side.
- Patterns are type-specific, including irrefutable patterns. Try adding or removing an element to the tuple and look at the resulting compiler errors.
- Variable names are patterns that always match and bind the matched value into a new variable with that name.
- `_` is a pattern that always matches any value, discarding the matched value.
- `..` allows you to ignore multiple values at once.

More to Explore

- You can also demonstrate more advanced usages of `..`, such as ignoring the middle elements of a tuple.

```
fn takes_tuple(tuple: (char, i32, bool, u8)) {
    let (first, .., last) = tuple;
}
```

- All of these patterns work with arrays as well:

```
fn takes_array(array: [u8; 5]) {
    let [first, .., last] = array;
}
```

Matching Values

The `match` keyword lets you match a value against one or more *patterns*. The patterns can be simple values, similarly to `switch` in C and C++, but they can also be used to express more complex conditions:

```
#[rustfmt::skip]
fn main() {
    let input = 'x';
    match input {
        'q'                         => println!("Quitting"),
        'a' | 's' | 'w' | 'd'       => println!("Moving around"),
        '0'..='9'                   => println!("Number input"),
        key if key.is_lowercase() => println!("Lowercase: {key}"),
        _                           => println!("Something else"),
    }
}
```

A variable in the pattern (`key` in this example) will create a binding that can be used within the match arm. We will learn more about this on the next slide.

A match guard causes the arm to match only if the condition is true. If the condition is false the match will continue checking later cases.

▼ *Speaker Notes*

This slide should take about 10 minutes.

Key Points:

- You might point out how some specific characters are being used when in a pattern
  - `|` as an `or`
  - `..` matches any number of items
  - `1..=5` represents an inclusive range
  - `_` is a wild card
- Match guards as a separate syntax feature are important and necessary when we wish to concisely express more complex ideas than patterns alone would allow.
- Match guards are different from `if` expressions after the `=>`. An `if` expression is evaluated after the match arm is selected. Failing the `if` condition inside of that block won't result in other arms of the original `match` expression being considered. In the following example, the wildcard pattern `_ =>` is never even attempted.

```rust
#[rustfmt::skip]
fn main() {
    let input = 'a';
    match input {
        key if key.is_uppercase() => println!("Uppercase"),
        key => if input == 'q' { println!("Quitting") },
        _   => println!("Bug: this is never printed"),
    }
}
```

- The condition defined in the guard applies to every expression in a pattern with an `|`.
- Note that you can't use an existing variable as the condition in a match arm, as it will instead be interpreted as a variable name pattern, which creates a new variable that will shadow the existing one. For example:

```rust
let expected = 5;
match 123 {
    expected => println!("Expected value is 5, actual is {expected}"),
    _ => println!("Value was something else"),
}
```

  Here we're trying to match on the number 123, where we want the first case to check if the value is 5. The naive expectation is that the first case won't match because the value isn't 5, but instead this is interpreted as a variable pattern which always matches, meaning the first branch will always be taken. If a constant is used instead this will then work as expected.

More To Explore

- Another piece of pattern syntax you can show students is the `@` syntax which binds a part of a pattern to a variable. For example:

```rust
let opt = Some(123);
match opt {
    outer @ Some(inner) => {
        println!("outer: {outer:?}, inner: {inner}");
    }
    None => {}
}
```

  In this example `inner` has the value 123 which it pulled from the `Option` via destructuring, `outer` captures the entire `Some(inner)` expression, so it contains the full `Some(123)`. This is rarely used but can be useful in more complex

Structs

Like tuples, structs can also be destructured by matching:

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

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

▼ *Speaker Notes*

This slide should take about 4 minutes.

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

More to Explore

- Try `match &foo` and check the type of captures. The pattern syntax remains the same, but the captures become shared references. This is match ergonomics and is often useful with `match self` when implementing methods on an enum.
  - The same effect occurs with `match &mut foo`: the captures become exclusive references.
- The distinction between a capture and a constant expression can be hard to spot. Try changing the `2` in the first arm to a variable, and see that it subtly doesn't work. Change it to a `const` and see it working again.

Enums

Like tuples, enums can also be destructured by matching:

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.

▼ *Speaker Notes*

This slide should take about 4 minutes.

- The `if` / `else` expression is returning an enum that is later unpacked with a `match`.
- You can try adding a third variant to the enum definition and displaying the errors when running the code. Point out the places where your code is now inexhaustive and how the compiler tries to give you hints.
- The values in the enum variants can only be accessed after being pattern matched.
- Demonstrate what happens when the search is inexhaustive. Note the advantage the Rust compiler provides by confirming when all cases are handled.
- Demonstrate the syntax for a struct-style variant by adding one to the enum definition and the `match`. Point out how this is syntactically similar to matching on a struct.

Let Control Flow

Rust has a few control flow constructs that differ from other languages. They are used for pattern matching:

- `if let` expressions
- `while let` expressions
- `let else` expressions

`if let` Expressions

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

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

fn sleep_for(secs: f32) {
    let result = Duration::try_from_secs_f32(secs);

    if let Ok(duration) = result {
        std::thread::sleep(duration);
        println!("slept for {duration:?}");
    }
}

fn main() {
    sleep_for(-10.0);
    sleep_for(0.8);
}
```

▼ *Speaker Notes*

- Unlike `match`, `if let` does not have to cover all branches. This can make it more concise than `match`.
- A common usage is handling `Some` values when working with `Option`.
- Unlike `match`, `if let` does not support guard clauses for pattern matching.
- With an `else` clause, this can be used as an expression.

`while let` Statements

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

```
fn main() {
    let mut name = String::from("Comprehensive Rust 🦀");
    while let Some(c) = name.pop() {
        dbg!(c);
    }
    // (There are more efficient ways to reverse a string!)
}
```

Here `String::pop` returns `Some(c)` until the string is empty, after which it will return `None`. The `while let` lets us keep iterating through all items.

▼ *Speaker Notes*

- Point out that the `while let` loop will keep going as long as the value matches the pattern.
- You could rewrite the `while let` loop as an infinite loop with an if statement that breaks when there is no value to unwrap for `name.pop()`. The `while let` provides syntactic sugar for the above scenario.
- This form cannot be used as an expression, because it may have no value if the condition is false.

`let else` Statements

For the common case of matching a pattern and returning from the function, use `let else`. The "else" case must diverge (`return`, `break`, or panic - anything but falling off the end of the block).

```
fn hex_or_die_trying(maybe_string: Option<String>) -> Result<u32, String> {
    let s = if let Some(s) = maybe_string {
        s
    } else {
        return Err(String::from("got None"));
    };

    let first_byte_char = if let Some(first) = s.chars().next() {
        first
    } else {
        return Err(String::from("got empty string"));
    };

    let digit = if let Some(digit) = first_byte_char.to_digit(16) {
        digit
    } else {
        return Err(String::from("not a hex digit"));
    };

    Ok(digit)
}

fn main() {
    println!("result: {:?}", hex_or_die_trying(Some(String::from("foo"))));
}
```

▼ *Speaker Notes*

The rewritten version is:

```
fn hex_or_die_trying(maybe_string: Option<String>) -> Result<u32, String> {
    let Some(s) = maybe_string else {
        return Err(String::from("got None"));
    };

    let Some(first_byte_char) = s.chars().next() else {
        return Err(String::from("got empty string"));
    };

    let Some(digit) = first_byte_char.to_digit(16) else {
        return Err(String::from("not a hex digit"));
    };

    Ok(digit)
```

More to Explore

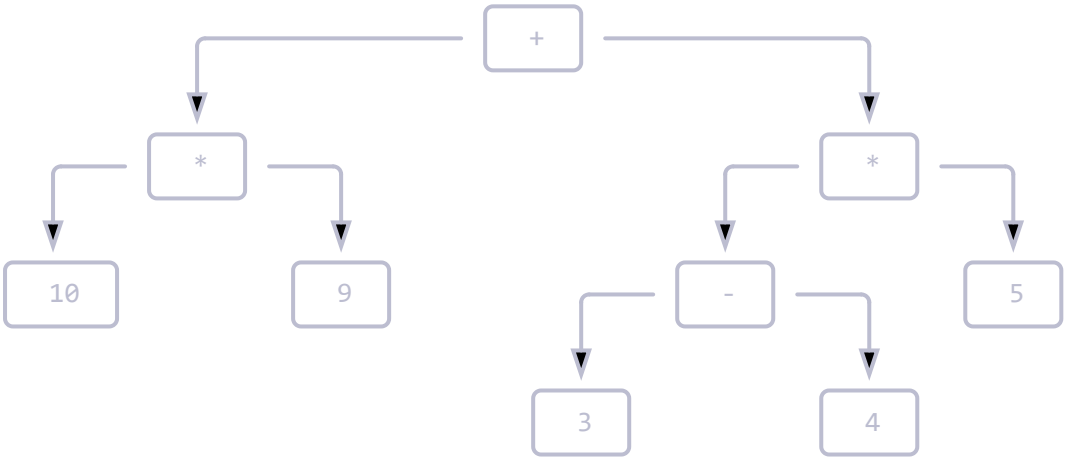
- This early return-based control flow is common in Rust error handling code, where you try to get a value out of a `Result`, returning an error if the `Result` was `Err`.
- If students ask, you can also demonstrate how real error handling code would be written with `?`.

Exercise: Expression Evaluation

Let's write a simple recursive evaluator for arithmetic expressions.

An example of a small arithmetic expression could be `10 + 20`, which evaluates to `30`. We can represent the expression as a tree:

A bigger and more complex expression would be `(10 * 9) + ((3 - 4) * 5)`, which evaluates to `85`. We represent this as a much bigger tree:



In code, we will represent the tree with two types:

```
/// An operation to perform on two subexpressions.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// An expression, in tree form.
#[derive(Debug)]
enum Expression {
    /// An operation on two subexpressions.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

```

The `Box` type here is a smart pointer, and will be covered in detail later in the course. An expression can be "boxed" with `Box::new` as seen in the tests. To evaluate a boxed expression, use the deref operator (`*`) to "unbox" it: `eval(*boxed_expr)` .

Copy and paste the code into the Rust playground, and begin implementing `eval` . The final product should pass the tests. It may be helpful to use `todo!()` and get the tests to pass one-by-one. You can also skip a test temporarily with `#[ignore]` :

```
#[test]
#[ignore]
fn test_value() { .. }
```

```rust
/// An operation to perform on two subexpressions.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// An expression, in tree form.
#[derive(Debug)]
enum Expression {
    /// An operation on two subexpressions.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// A literal value
    Value(i64),
}

fn eval(e: Expression) -> i64 {
    todo!()
}

#[test]
fn test_value() {
    assert_eq!(eval(Expression::Value(19)), 19);
}

#[test]
fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
        }),
        30
    );
}

#[test]
fn test_recursion() {
    let term1 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Value(10)),
        right: Box::new(Expression::Value(9)),
    };
    let term2 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(3)),
            right: Box::new(Expression::Value(4)),
        }),
        right: Box::new(Expression::Value(5)),
    };
```

```
            left: Box::new(term1),
            right: Box::new(term2),
        }),
        85
    );
}

#[test]
fn test_zeros() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Mul,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
}

#[test]
fn test_div() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Div,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(2)),
        }),
        5
    )
}
```

Solution

```rust
/// An operation to perform on two subexpressions.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// An expression, in tree form.
#[derive(Debug)]
enum Expression {
    /// An operation on two subexpressions.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// A literal value
    Value(i64),
}

fn eval(e: Expression) -> i64 {
    match e {
        Expression::Op { op, left, right } => {
            let left = eval(*left);
            let right = eval(*right);
            match op {
                Operation::Add => left + right,
                Operation::Sub => left - right,
                Operation::Mul => left * right,
                Operation::Div => left / right,
            }
        }
        Expression::Value(v) => v,
    }
}

#[test]
fn test_value() {
    assert_eq!(eval(Expression::Value(19)), 19);
}

#[test]
fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
        }),
        30
    );
}
```

```
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(9)),
        };
        let term2 = Expression::Op {
            op: Operation::Mul,
            left: Box::new(Expression::Op {
                op: Operation::Sub,
                left: Box::new(Expression::Value(3)),
                right: Box::new(Expression::Value(4)),
            }),
            right: Box::new(Expression::Value(5)),
        };
        assert_eq!(
            eval(Expression::Op {
                op: Operation::Add,
                left: Box::new(term1),
                right: Box::new(term2),
            }),
            85
        );
    }

    #[test]
    fn test_zeros() {
        assert_eq!(
            eval(Expression::Op {
                op: Operation::Add,
                left: Box::new(Expression::Value(0)),
                right: Box::new(Expression::Value(0))
            }),
            0
        );
        assert_eq!(
            eval(Expression::Op {
                op: Operation::Mul,
                left: Box::new(Expression::Value(0)),
                right: Box::new(Expression::Value(0))
            }),
            0
        );
        assert_eq!(
            eval(Expression::Op {
                op: Operation::Sub,
                left: Box::new(Expression::Value(0)),
                right: Box::new(Expression::Value(0))
            }),
            0
        );
    }

    #[test]
    fn test_div() {
        assert_eq!(
            eval(Expression::Op {
                op: Operation::Div,
```

```
116         }
117     }
```

Methods and Traits

This segment should take about 45 minutes. It contains:

Slide	Duration
Methods	10 minutes
Traits	15 minutes
Deriving	3 minutes
Exercise: Generic Logger	15 minutes

Methods

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

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

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

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

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

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

fn main() {
    let mut race = CarRace::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);
}
```

The `self` arguments specify the "receiver" - the object the method acts on. There are several common receivers for a method:

- `&self` : borrows the object from the caller using a shared and immutable reference.

- `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 that are called `new` by convention.

▼ *Speaker Notes*

This slide should take about 8 minutes.

Key Points:

- It can be helpful to introduce methods by comparing them to functions.
  - Methods are called on an instance of a type (such as a struct or enum), the first parameter represents the instance as `self` .
  - Developers may choose to use methods to take advantage of method receiver syntax and to help keep them more organized. By using methods we can keep all the implementation code in one predictable place.
  - Note that methods can also be called like associated functions by explicitly passing the receiver in, e.g. `CarRace::add_lap(&mut race, 20)` .
- Point out the use of the keyword `self` , a method receiver.
  - Show that it is an abbreviated term for `self: Self` and perhaps show how the struct name could also be used.
  - Explain that `Self` is a type alias for the type the `impl` block is in and can be used elsewhere in the block.
  - Note how `self` is used like other structs and dot notation can be used to refer to individual fields.
  - This might be a good time to demonstrate how the `&self` differs from `self` by trying to run `finish` twice.
  - Beyond variants on `self` , there are also special wrapper types allowed to be receiver types, such as `Box<Self>` .

Traits

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

```
trait Pet {
    /// Return a sentence from this pet.
    fn talk(&self) -> String;

    /// Print a string to the terminal greeting this pet.
    fn greet(&self);
}
```

▼ *Speaker Notes*

This slide and its sub-slides should take about 15 minutes.

- A trait defines a number of methods that types must have in order to implement the trait.
- In the "Generics" segment, next, we will see how to build functionality that is generic over all types implementing a trait.

Implementing Traits

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

    fn greet(&self) {
        println!("Oh you're a cutie! What's your name? {}", self.talk());
    }
}

struct Dog {
    name: String,
    age: i8,
}

impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Woof, my name is {}!", self.name)
    }
}

fn main() {
    let fido = Dog { name: String::from("Fido"), age: 5 };
    dbg!(fido.talk());
    fido.greet();
}
```

▼ *Speaker Notes*

- To implement `Trait` for `Type`, you use an `impl Trait for Type { .. }` block.
- Unlike Go interfaces, just having matching methods is not enough: a `Cat` type with a `talk()` method would not automatically satisfy `Pet` unless it is in an `impl Pet` block.
- Traits may provide default implementations of some methods. Default implementations can rely on all the methods of the trait. In this case, `greet` is provided, and relies on `talk`.
- Multiple `impl` blocks are allowed for a given type. This includes both inherent `impl` blocks and trait `impl` blocks. Likewise multiple traits can be implemented for a given type (and often types implement many traits!). `impl` blocks can even be spread across multiple modules/files.

Supertraits

A trait can require that types implementing it also implement other traits, called *supertraits*. Here, any type implementing `Pet` must implement `Animal`.

```
trait Animal {
    fn leg_count(&self) -> u32;
}

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

struct Dog(String);

impl Animal for Dog {
    fn leg_count(&self) -> u32 {
        4
    }
}

impl Pet for Dog {
    fn name(&self) -> String {
        self.0.clone()
    }
}

fn main() {
    let puppy = Dog(String::from("Rex"));
    println!("{} has {} legs", puppy.name(), puppy.leg_count());
}
```

▼ *Speaker Notes*

This is sometimes called “trait inheritance” but students should not expect this to behave like OO inheritance. It just specifies an additional requirement on implementations of a trait.

Associated Types

Associated types are placeholder types that are supplied by the trait implementation.

```
#[derive(Debug)]
struct Meters(i32);
#[derive(Debug)]
struct MetersSquared(i32);

trait Multiply {
    type Output;
    fn multiply(&self, other: &Self) -> Self::Output;
}

impl Multiply for Meters {
    type Output = MetersSquared;
    fn multiply(&self, other: &Self) -> Self::Output {
        MetersSquared(self.0 * other.0)
    }
}

fn main() {
    println!("{:?}", Meters(10).multiply(&Meters(20)));
}
```

▼ *Speaker Notes*

- Associated types are sometimes also called “output types”. The key observation is that the implementer, not the caller, chooses this type.
- Many standard library traits have associated types, including arithmetic operators and `Iterator`.

Deriving

Supported traits can be automatically implemented for your custom types, as follows:

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

fn main() {
    let p1 = Player::default(); // Default trait adds `default` constructor.
    let mut p2 = p1.clone(); // Clone trait adds `clone` method.
    p2.name = String::from("EldurScrollz");
    // Debug trait adds support for printing with `{:?}`.
    println!("{p1:?} vs. {p2:?}");
}
```

▼ *Speaker Notes*

This slide should take about 3 minutes.

- Derivation is implemented with macros, and many crates provide useful derive macros to add useful functionality. For example, `serde` can derive serialization support for a struct using `#[derive(Serialize)]`.
- Derivation is usually provided for traits that have a common boilerplate implementation that is correct for most cases. For example, demonstrate how a manual `Clone` impl can be repetitive compared to deriving the trait:

```
impl Clone for Player {
    fn clone(&self) -> Self {
        Player {
            name: self.name.clone(),
            strength: self.strength.clone(),
            hit_points: self.hit_points.clone(),
        }
    }
}
```

  Not all of the `.clone()`s in the above are necessary in this case, but this demonstrates the generally boilerplate-y pattern that manual impls would follow, which should help make the use of `derive` clear to students.

Exercise: Logger Trait

Let's design a simple logging utility, using a trait `Logger` with a `log` method. Code that might log its progress can then take an `&impl Logger`. In testing, this might put messages in the test logfile, while in a production build it would send messages to a log server.

However, the `StderrLogger` given below logs all messages, regardless of verbosity. Your task is to write a `VerbosityFilter` type that will ignore messages above a maximum verbosity.

This is a common pattern: a struct wrapping a trait implementation and implementing that same trait, adding behavior in the process. In the "Generics" segment, we will see how to make the wrapper generic over the wrapped type.

```
trait Logger {
    /// Log a message at the given verbosity level.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

/// Only log messages up to the given verbosity level.
struct VerbosityFilter {
    max_verbosity: u8,
    inner: StderrLogger,
}

// TODO: Implement the `Logger` trait for `VerbosityFilter`.

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}
```

Solution

```rust
trait Logger {
    /// Log a message at the given verbosity level.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

/// Only log messages up to the given verbosity level.
struct VerbosityFilter {
    max_verbosity: u8,
    inner: StderrLogger,
}

impl Logger for VerbosityFilter {
    fn log(&self, verbosity: u8, message: &str) {
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}
```

Generics

This segment should take about 50 minutes. It contains:

Slide	Duration
Generic Functions	5 minutes
Trait Bounds	10 minutes
Generic Data Types	10 minutes
Generic Traits	5 minutes
impl Trait	5 minutes
dyn Trait	5 minutes
Exercise: Generic min	10 minutes

Generic Functions

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

```
fn pick<T>(cond: bool, left: T, right: T) -> T {
    if cond { left } else { right }
}

fn main() {
    println!("picked a number: {:?}", pick(true, 222, 333));
    println!("picked a string: {:?}", pick(false, 'L', 'R'));
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- It can be helpful to show the monomorphized versions of `pick`, either before talking about the generic `pick` in order to show how generics can reduce code duplication, or after talking about generics to show how monomorphization works.

```
fn pick_i32(cond: bool, left: i32, right: i32) -> i32 {
    if cond { left } else { right }
}

fn pick_char(cond: bool, left: char, right: char) -> char {
    if cond { left } else { right }
}
```

- Rust infers a type for T based on the types of the arguments and return value.
- In this example we only use the primitive types `i32` and `char` for `T`, but we can use any type here, including user-defined types:

```
struct Foo {
    val: u8,
}

pick(false, Foo { val: 7 }, Foo { val: 99 });
```

- This is similar to C++ templates, but Rust partially compiles the generic function immediately, so that function must be valid for all types matching the constraints. For

- Generic code is turned into non-generic code based on the call sites. This is a zero-cost abstraction: you get exactly the same result as if you had hand-coded the data structures without the abstraction.

Trait 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`:

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

struct NotCloneable;

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

▼ *Speaker Notes*

This slide should take about 8 minutes.

- Try making a `NotCloneable` and passing it to `duplicate`.
- When multiple traits are necessary, use `+` to join them.
- Show a `where` clause, students will encounter it when reading code.

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

  - It declutters the function signature if you have many parameters.
  - It has additional features making it more powerful.
    - If someone asks, the extra feature is that the type on the left of ":" can be arbitrary, like `Option<T>`.
- Note that Rust does not (yet) support specialization. For example, given the original `duplicate`, it is invalid to add a specialized `duplicate(a: u32)`.

Generic Data Types

You can use generics to abstract over the concrete field type. Returning to the exercise for the previous segment:

```
pub trait Logger {
    /// Log a message at the given verbosity level.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

/// Only log messages up to the given verbosity level.
struct VerbosityFilter<L> {
    max_verbosity: u8,
    inner: L,
}

impl<L: Logger> Logger for VerbosityFilter<L> {
    fn log(&self, verbosity: u8, message: &str) {
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

- *Q:* Why is `L` specified twice in `impl<L: Logger> .. VerbosityFilter<L>` ? Isn't that redundant?
  - This is because it is a generic implementation section for generic type. They are independently generic.
  - It means these methods are defined for any `L` .
  - It is possible to write `impl VerbosityFilter<StderrLogger> { .. }` .
    - `VerbosityFilter` is still generic and you can use `VerbosityFilter<f64>` ,

- Note that we don't put a trait bound on the `VerbosityFilter` type itself. You can put bounds there as well, but generally in Rust we only put the trait bounds on the impl blocks.

Generic Traits

Traits can also be generic, just like types and functions. A trait's parameters get concrete types when it is used. For example the `From<T>` trait is used to define type conversions:

```
pub trait From<T>: Sized {
    fn from(value: T) -> Self;
}
```

```
#[derive(Debug)]
struct Foo(String);

impl From<u32> for Foo {
    fn from(from: u32) -> Foo {
        Foo(format!("Converted from integer: {from}"))
    }
}

impl From<bool> for Foo {
    fn from(from: bool) -> Foo {
        Foo(format!("Converted from bool: {from}"))
    }
}

fn main() {
    let from_int = Foo::from(123);
    let from_bool = Foo::from(true);
    dbg!(from_int);
    dbg!(from_bool);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- The `From` trait will be covered later in the course, but its definition in the `std` docs is simple, and copied here for reference.
- Implementations of the trait do not need to cover all possible type parameters. Here, `Foo::from("hello")` would not compile because there is no `From<&str>` implementation for `Foo`.
- Generic traits take types as "input", while associated types are a kind of "output" type. A trait can have multiple implementations for different input types.
- In fact, Rust requires that at most one implementation of a trait match for any type T. Unlike some other languages, Rust has no heuristic for choosing the "most specific" match. There is work on adding this support, called specialization.

impl Trait

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

```
// 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
}

fn pair_of(x: u32) -> impl std::fmt::Debug {
    (x + 1, x - 1)
}

fn main() {
    let many = add_42_millions(42_i8);
    dbg!(many);
    let many_more = add_42_millions(10_000_000);
    dbg!(many_more);
    let debuggable = pair_of(27);
    dbg!(debuggable);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

`impl Trait` allows you to work with types that you cannot name. The meaning of `impl Trait` is a bit different in the different positions.

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

  Inference is hard in return position. A function returning `impl Foo` picks the concrete type it returns, without writing it out in the source. A function returning a generic type like `collect<B>() -> B` can return any type satisfying `B`, and the caller may need to choose one, such as with `let x: Vec<_> = foo.collect()` or with the turbofish, `foo.collect::<Vec<_>>()`.

What is the type of `debuggable`? Try `let debuggable: () = ..` to see what the error message shows.

dyn Trait

In addition to using traits for static dispatch via generics, Rust also supports using them for type-erased, dynamic dispatch via trait objects:

```
struct Dog {
    name: String,
    age: i8,
}
struct Cat {
    lives: i8,
}

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

impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Woof, my name is {}!", self.name)
    }
}

impl Pet for Cat {
    fn talk(&self) -> String {
        String::from("Miau!")
    }
}

// Uses generics and static dispatch.
fn generic(pet: &impl Pet) {
    println!("Hello, who are you? {}", pet.talk());
}

// Uses type-erasure and dynamic dispatch.
fn dynamic(pet: &dyn Pet) {
    println!("Hello, who are you? {}", pet.talk());
}

fn main() {
    let cat = Cat { lives: 9 };
    let dog = Dog { name: String::from("Fido"), age: 5 };

    generic(&cat);
    generic(&dog);

    dynamic(&cat);
    dynamic(&dog);
}
```

▼ *Speaker Notes*

- Generics, including `impl Trait`, use monomorphization to create a specialized instance of the function for each different type that the generic is instantiated with. This means that calling a trait method from within a generic function still uses static dispatch, as the compiler has full type information and can resolve that type's trait implementation to use.

- When using `dyn Trait`, it instead uses dynamic dispatch through a virtual method table (vtable). This means that there's a single version of `fn dynamic` that is used regardless of what type of `Pet` is passed in.

- When using `dyn Trait`, the trait object needs to be behind some kind of indirection. In this case it's a reference, though smart pointer types like `Box` can also be used (this will be demonstrated on day 3).

- At runtime, a `&dyn Pet` is represented as a "fat pointer", i.e. a pair of two pointers: One pointer points to the concrete object that implements `Pet`, and the other points to the vtable for the trait implementation for that type. When calling the `talk` method on `&dyn Pet` the compiler looks up the function pointer for `talk` in the vtable and then invokes the function, passing the pointer to the `Dog` or `Cat` into that function. The compiler doesn't need to know the concrete type of the `Pet` in order to do this.

- A `dyn Trait` is considered to be "type-erased", because we no longer have compile-time knowledge of what the concrete type is.

Exercise: Generic `min`

In this short exercise, you will implement a generic `min` function that determines the minimum of two values, using the `Ord` trait.

```
use std::cmp::Ordering;

// TODO: implement the `min` function used in the tests.

#[test]
fn integers() {
    assert_eq!(min(0, 10), 0);
    assert_eq!(min(500, 123), 123);
}

#[test]
fn chars() {
    assert_eq!(min('a', 'z'), 'a');
    assert_eq!(min('7', '1'), '1');
}

#[test]
fn strings() {
    assert_eq!(min("hello", "goodbye"), "goodbye");
    assert_eq!(min("bat", "armadillo"), "armadillo");
}
```

▼ *Speaker Notes*

This slide and its sub-slides should take about 10 minutes.

- Show students the `Ord` trait and `Ordering` enum.

Solution

```rust
use std::cmp::Ordering;

fn min<T: Ord>(l: T, r: T) -> T {
    match l.cmp(&r) {
        Ordering::Less | Ordering::Equal => l,
        Ordering::Greater => r,
    }
}

#[test]
fn integers() {
    assert_eq!(min(0, 10), 0);
    assert_eq!(min(500, 123), 123);
}

#[test]
fn chars() {
    assert_eq!(min('a', 'z'), 'a');
    assert_eq!(min('7', '1'), '1');
}

#[test]
fn strings() {
    assert_eq!(min("hello", "goodbye"), "goodbye");
    assert_eq!(min("bat", "armadillo"), "armadillo");
}
```

Welcome Back

Including 10 minute breaks, this session should take about 2 hours and 50 minutes. It contains:

Segment	Duration
Closures	30 minutes
Standard Library Types	1 hour
Standard Library Traits	1 hour

Closures

This segment should take about 30 minutes. It contains:

Slide	Duration
Closure Syntax	3 minutes
Capturing	5 minutes
Closure Traits	10 minutes
Exercise: Log Filter	10 minutes

Closure Syntax

Closures are created with vertical bars: `|..| ...`

```
fn main() {
    // Argument and return type can be inferred for lightweight syntax:
    let double_it = |n| n * 2;
    dbg!(double_it(50));

    // Or we can specify types and bracket the body to be fully explicit:
    let add_1f32 = |x: f32| -> f32 { x + 1.0 };
    dbg!(add_1f32(50.));
}
```

▼ *Speaker Notes*

This slide should take about 3 minutes.

- The arguments go between the `|..|`. The body can be surrounded by `{ .. }`, but if it is a single expression these can be omitted.
- Argument types are optional, and are inferred if not given. The return type is also optional, but can only be written if using `{ .. }` around the body.
- The examples can both be written as mere nested functions instead – they do not capture any variables from their lexical environment. We will see captures next.

More to Explore

- The ability to store functions in variables doesn't just apply to closures, regular functions can be put in variables and then invoked the same way that closures can: Example in the playground.
    - The linked example also demonstrates that closures that don't capture anything can also coerce to a regular function pointer.

Capturing

A closure can capture variables from the environment where it was defined.

```
fn main() {
    let max_value = 5;
    let clamp = |v| {
        if v > max_value { max_value } else { v }
    };

    dbg!(clamp(1));
    dbg!(clamp(3));
    dbg!(clamp(5));
    dbg!(clamp(7));
    dbg!(clamp(10));
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- By default, a closure captures values by reference. Here `max_value` is captured by `clamp`, but still available to `main` for printing. Try making `max_value` mutable, changing it, and printing the clamped values again. Why doesn't this work?
- If a closure mutates values, it will capture them by mutable reference. Try adding `max_value += 1` to `clamp`.
- You can force a closure to move values instead of referencing them with the `move` keyword. This can help with lifetimes, for example if the closure must outlive the captured values (more on lifetimes later).

  This looks like `move |v| ...`. Try adding this keyword and see if `main` can still access `max_value` after defining `clamp`.
- By default, closures will capture each variable from an outer scope by the least demanding form of access they can (by shared reference if possible, then exclusive reference, then by move). The `move` keyword forces capture by value.

Closure traits

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

The special types `fn(..) -> T` refer to function pointers - either the address of a function, or a closure that captures nothing.

```
fn apply_and_log(
    func: impl FnOnce(&'static str) -> String,
    func_name: &'static str,
    input: &'static str,
) {
    println!("Calling {func_name}({input}): {}", func(input))
}

fn main() {
    let suffix = "-itis";
    let add_suffix = |x| format!("{x}{suffix}");
    apply_and_log(&add_suffix, "add_suffix", "senior");
    apply_and_log(&add_suffix, "add_suffix", "appendix");

    let mut v = Vec::new();
    let mut accumulate = |x| {
        v.push(x);
        v.join("/")
    };
    apply_and_log(&mut accumulate, "accumulate", "red");
    apply_and_log(&mut accumulate, "accumulate", "green");
    apply_and_log(&mut accumulate, "accumulate", "blue");

    let take_and_reverse = |prefix| {
        let mut acc = String::from(prefix);
        acc.push_str(&v.into_iter().rev().collect::<Vec<_>>().join("/"));
        acc
    };
    apply_and_log(take_and_reverse, "take_and_reverse", "reversed: ");
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

An `Fn` (e.g. `add_suffix`) neither consumes nor mutates captured values. It can be called needing only a shared reference to the closure, which means the closure can be executed repeatedly and even concurrently.

An `FnMut` (e.g. `accumulate`) might mutate captured values. The closure object is accessed via exclusive reference, so it can be called repeatedly but not concurrently.

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

When you define a function that takes a closure, you should take `FnOnce` if you can (i.e. you call it once), or `FnMut` else, and last `Fn`. This allows the most flexibility for the caller.

In contrast, when you have a closure, the most flexible you can have is `Fn` (which can be passed to a consumer of any of the three closure traits), then `FnMut`, and lastly `FnOnce`.

The compiler also infers `Copy` (e.g. for `add_suffix`) and `Clone` (e.g. `take_and_reverse`), depending on what the closure captures. Function pointers (references to `fn` items) implement `Copy` and `Fn`.

Exercise: Log Filter

Building on the generic logger from this morning, implement a `Filter` that uses a closure to filter log messages, sending those that pass the filtering predicate to an inner logger.

```
pub trait Logger {
    /// Log a message at the given verbosity level.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

// TODO: Define and implement `Filter`.

fn main() {
    let logger = Filter::new(StderrLogger, |_verbosity, msg| msg.contains("y
    logger.log(5, "FYI");
    logger.log(1, "yikes, something went wrong");
    logger.log(2, "uhoh");
}
```

Solution

```
pub trait Logger {
    /// Log a message at the given verbosity level.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

/// Only log messages matching a filtering predicate.
struct Filter<L, P> {
    inner: L,
    predicate: P,
}

impl<L, P> Filter<L, P>
where
    L: Logger,
    P: Fn(u8, &str) -> bool,
{
    fn new(inner: L, predicate: P) -> Self {
        Self { inner, predicate }
    }
}
impl<L, P> Logger for Filter<L, P>
where
    L: Logger,
    P: Fn(u8, &str) -> bool,
{
    fn log(&self, verbosity: u8, message: &str) {
        if (self.predicate)(verbosity, message) {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
    let logger = Filter::new(StderrLogger, |_verbosity, msg| msg.contains("y
    logger.log(5, "FYI");
    logger.log(1, "yikes, something went wrong");
    logger.log(2, "uhoh");
}
```

▼ *Speaker Notes*

removing it and showing how the compiler now needs type annotations for the closure passed to `new`.

Standard Library Types

This segment should take about 1 hour. It contains:

Slide	Duration
Standard Library	3 minutes
Documentation	5 minutes
Option	10 minutes
Result	5 minutes
String	5 minutes
Vec	5 minutes
HashMap	5 minutes
Exercise: Counter	20 minutes

▼ *Speaker Notes*

For each of the slides in this section, spend some time reviewing the documentation pages, highlighting some of the more common methods.

Standard Library

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

In fact, Rust contains several layers of the Standard Library: `core`, `alloc` and `std`.

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

Documentation

Rust comes with extensive documentation. For example:

- All of the details about loops.
- Primitive types like `u8`.
- Standard library types like `Option` or `BinaryHeap`.

Use `rustup doc --std` or https://std.rs to view the documentation.

In fact, you can document your own code:

```
/// 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;
    }
    lhs % rhs == 0
}
```

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.

To document an item from inside the item (such as inside a module), use `//!` or `/*! .. */`, called "inner doc comments":

```
//! This module contains functionality relating to divisibility of integers.
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Show students the generated docs for the `rand` crate at https://docs.rs/rand.

Option

We have already seen some use of `Option<T>`. It stores either a value of type `T` or nothing. For example, `String::find` returns an `Option<usize>`.

```
fn main() {
    let name = "Löwe 老虎 Léopard Gepardi";
    let mut position: Option<usize> = name.find('é');
    dbg!(position);
    assert_eq!(position.unwrap(), 14);
    position = name.find('Z');
    dbg!(position);
    assert_eq!(position.expect("Character not found"), 0);
}
```

▼ *Speaker Notes*

This slide should take about 10 minutes.

- `Option` is widely used, not just in the standard library.
- `unwrap` will return the value in an `Option`, or panic. `expect` is similar but takes an error message.
    - You can panic on None, but you can't "accidentally" forget to check for None.
    - It's common to `unwrap` / `expect` all over the place when hacking something together, but production code typically handles `None` in a nicer fashion.
- The "niche optimization" means that `Option<T>` often has the same size in memory as `T`, if there is some representation that is not a valid value of T. For example, a reference cannot be NULL, so `Option<&T>` automatically uses NULL to represent the `None` variant, and thus can be stored in the same memory as `&T`.

Result

`Result` is similar to `Option`, but indicates the success or failure of an operation, each with a different enum variant. It is generic: `Result<T, E>` where `T` is used in the `Ok` variant and `E` appears in the `Err` variant.

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

fn main() {
    let file: Result<File, std::io::Error> = File::open("diary.txt");
    match file {
        Ok(mut file) => {
            let mut contents = String::new();
            if let Ok(bytes) = file.read_to_string(&mut contents) {
                println!("Dear diary: {contents} ({bytes} bytes)");
            } else {
                println!("Could not read file content");
            }
        }
        Err(err) => {
            println!("The diary could not be opened: {err}");
        }
    }
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- As with `Option`, the successful value sits inside of `Result`, forcing the developer to explicitly extract it. This encourages error checking. In the case where an error should never happen, `unwrap()` or `expect()` can be called, and this is a signal of the developer intent too.
- `Result` documentation is a recommended read. Not during the course, but it is worth mentioning. It contains a lot of convenience methods and functions that help functional-style programming.
- `Result` is the standard type to implement error handling as we will see on Day 4.

String

`String` is a growable UTF-8 encoded string:

```
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().cour
}
```

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

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `String::new` returns a new empty string, use `String::with_capacity` when you know how much data you want to push to the string.
- `String::len` returns the size of the `String` in bytes (which can be different from its length in characters).
- `String::chars` returns an iterator over the actual characters. Note that a `char` can be different from what a human will consider a "character" due to grapheme clusters.
- When people refer to strings they could either be talking about `&str` or `String`.
- When a type implements `Deref<Target = T>`, the compiler will let you transparently call methods from `T`.
  - We haven't discussed the `Deref` trait yet, so at this point this mostly explains the structure of the sidebar in the documentation.
  - `String` implements `Deref<Target = str>` which transparently gives it access to `str`'s methods.
  - Write and compare `let s3 = s1.deref();` and `let s3 = &*s1;`.
- `String` is implemented as a wrapper around a vector of bytes, many of the operations you see supported on vectors are also supported on `String`, but with some extra guarantees.
- Compare the different ways to index a `String`:
  - To a character by using `s3.chars().nth(i).unwrap()` where `i` is in-bound,

- Many types can be converted to a string with the `to_string` method. This trait is automatically implemented for all types that implement `Display`, so anything that can be formatted can also be converted to a string.

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`.

▼ *Speaker Notes*

This slide should take about 5 minutes.

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

HashMap

Standard hash map with protection against HashDoS attacks:

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

fn main() {
    let mut page_counts = HashMap::new();
    page_counts.insert("Adventures of Huckleberry Finn", 207);
    page_counts.insert("Grimms' Fairy Tales", 751);
    page_counts.insert("Pride and Prejudice", 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).or_insert(0);
        *page_count += 1;
    }

    dbg!(page_counts);
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `HashMap` is not defined in the prelude and needs to be brought into scope.
- Try the following lines of code. The first line will see if a book is in the hashmap and if not return an alternative value. The second line will insert the alternative value in the hashmap if the book is not found.

```rust
let pc1 = page_counts
    .get("Harry Potter and the Sorcerer's Stone")
    .unwrap_or(&336);
let pc2 = page_counts
```

- Unlike `vec!`, there is unfortunately no standard `hashmap!` macro.
  - Although, since Rust 1.56, HashMap implements `From<[(K, V); N]>`, which allows us to easily initialize a hash map from a literal array:

```
let page_counts = HashMap::from([
  ("Harry Potter and the Sorcerer's Stone".to_string(), 336),
  ("The Hunger Games".to_string(), 374),
]);
```

- Alternatively HashMap can be built from any `Iterator` that yields key-value tuples.
- This type has several "method-specific" return types, such as `std::collections::hash_map::Keys`. These types often appear in searches of the Rust docs. Show students the docs for this type, and the helpful link back to the `keys` method.

Exercise: Counter

In this exercise you will take a very simple data structure and make it generic. It uses a `std::collections::HashMap` to keep track of what values have been seen and how many times each one has appeared.

The initial version of `Counter` is hardcoded to only work for `u32` values. Make the struct and its methods generic over the type of value being tracked, that way `Counter` can track any type of value.

If you finish early, try using the `entry` method to halve the number of hash lookups required to implement the `count` method.

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

/// Counter counts the number of times each value of type T has been seen.
struct Counter {
    values: HashMap<u32, u64>,
}

impl Counter {
    /// Create a new Counter.
    fn new() -> Self {
        Counter {
            values: HashMap::new(),
        }
    }

    /// Count an occurrence of the given value.
    fn count(&mut self, value: u32) {
        if self.values.contains_key(&value) {
            *self.values.get_mut(&value).unwrap() += 1;
        } else {
            self.values.insert(value, 1);
        }
    }

    /// Return the number of times the given value has been seen.
    fn times_seen(&self, value: u32) -> u64 {
        self.values.get(&value).copied().unwrap_or_default()
    }
}

fn main() {
    let mut ctr = Counter::new();
    ctr.count(13);
    ctr.count(14);
    ctr.count(16);
    ctr.count(14);
    ctr.count(14);
    ctr.count(11);

    for i in 10..20 {
        println!("saw {} values equal to {}", ctr.times_seen(i), i);
    }

    let mut strctr = Counter::new();
    strctr.count("apple");
    strctr.count("orange");
    strctr.count("apple");
    println!("got {} apples", strctr.times_seen("apple"));
}
```

Solution

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

/// Counter counts the number of times each value of type T has been seen.
struct Counter<T> {
    values: HashMap<T, u64>,
}

impl<T: Eq + Hash> Counter<T> {
    /// Create a new Counter.
    fn new() -> Self {
        Counter { values: HashMap::new() }
    }

    /// Count an occurrence of the given value.
    fn count(&mut self, value: T) {
        *self.values.entry(value).or_default() += 1;
    }

    /// Return the number of times the given value has been seen.
    fn times_seen(&self, value: T) -> u64 {
        self.values.get(&value).copied().unwrap_or_default()
    }
}

fn main() {
    let mut ctr = Counter::new();
    ctr.count(13);
    ctr.count(14);
    ctr.count(16);
    ctr.count(14);
    ctr.count(14);
    ctr.count(11);

    for i in 10..20 {
        println!("saw {} values equal to {}", ctr.times_seen(i), i);
    }

    let mut strctr = Counter::new();
    strctr.count("apple");
    strctr.count("orange");
    strctr.count("apple");
    println!("got {} apples", strctr.times_seen("apple"));
}
```

Standard Library Traits

This segment should take about 1 hour. It contains:

Slide	Duration
Comparisons	5 minutes
Operators	5 minutes
From and Into	5 minutes
Casting	5 minutes
Read and Write	5 minutes
Default, struct update syntax	5 minutes
Exercise: ROT13	30 minutes

▼ *Speaker Notes*

As with the standard library types, spend time reviewing the documentation for each trait.

This section is long. Take a break midway through.