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 within 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

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

- Learning how to develop macros: please see 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 class and offer a solution, e.g., by showing people where to find the relevant information in the standard library.

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

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

Course Structure

> This page is for the course instructor.

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 45 minutes, including breaks)

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

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

Segment	Duration
Standard Library Types	1 hour
Closures	30 minutes
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

Segment	Duration
Borrowing	55 minutes
Lifetimes	50 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

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

You will need an AOSP checkout. Make a checkout of the course repository on the same machine and move the `src/android/` directory into the root of your AOSP checkout. This will ensure that the Android build system sees the `Android.bp` files in `src/android/`.

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

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.

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:

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

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 (25 minutes, including breaks)

Segment	Duration
Leveraging the Type System	25 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:

Segment	Duration
Setup	2 minutes
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.

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.

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 to 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 which 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.
    - Mention that it is quite rare to ever use the compiler directly not through `cargo` (most users never do).
    - It might be worth alluding that Cargo itself is an extremely powerful and comprehensive tool. It is capable of many advanced features including but not limited to:
        - Project/package structure
        - workspaces
        - Dev Dependencies and Runtime Dependency management/caching
        - build scripting
        - global installation
        - It is also extensible with sub command plugins as well (such as cargo clippy).
    - Read more from the official Cargo Book

Code Samples in This Training

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

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

The code blocks in this course are fully interactive:

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

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

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

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

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

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

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

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.

If you're teaching this in a classroom, this is a good place to go over the schedule. Note that there is an exercise at the end of each segment, followed by a break. Plan to cover the 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 which had its 1.0 release in 2015:

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

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

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:

```
1 fn main() {
2     println!("Hello 🌍!");
3 }
```

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 `;`.
- Rust has hygienic macros, `println!` is an example of this.
- 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 things like Unicode.
- Rust uses macros for situations where you want to have a variable number of arguments (no function overloading).
- Macros being 'hygienic' means they don't accidentally capture identifiers from the scope they are used in. Rust macros are actually only partially hygienic.
- 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 which 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 n'th Fibonacci number is calculated recursively as the sum of the n-1'th and n-2'th Fibonacci numbers.

Write a function `fib(n)` that calculates the n'th 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));
}
```

Solution

```rust
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));
}
```

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 `{}`. Each block has a value and a type, which are those of the last expression of the 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. We'll be talking about pattern matching in more detail tomorrow, so try not to go into too much detail here.

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 which 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 which 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 which 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:

- `assert!` and related macros can be used to add assertions to your code. These are used heavily in writing tests.
- `unreachable!` is used to mark a branch of control flow that should never be hit.
- `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

```rust
/// 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
}
```

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.
- 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.
- Arrays are not heap-allocated. They are regular values with a fixed size known at compile time, meaning they go on the stack. This can be different from what students expect if they come from a garbage collected language, where arrays may be heap allocated by default.

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.

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.

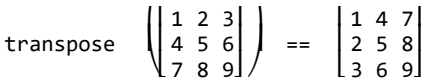
Exercise: Nested Arrays

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` which will transpose a matrix (turn 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],
    ];

    dbg!(matrix);
    let transposed = transpose(matrix);
    dbg!(transposed);
}
```

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],
    ];

    dbg!(matrix);
    let transposed = transpose(matrix);
    dbg!(transposed);
}
```

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