Rust Bootcamp 2021

Day 1

Day 1 Activities

- ✓ Installing Rust
- ✓ Cargo: build system & package manager
- ✓ Hello World!
- ✓ Procedural programming basics

Initial Resources

Our repository:
https://github.com/icecube/rust-bootcamp-2021

The Rust Book:
https://doc.rust-lang.org/book/

Rust by Example:
https://doc.rust-lang.org/stable/rust-by-example/

Installation

The first step is to install Rust. On Linux or Mac:

```
$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

https://doc.rust-lang.org/book/ch01-01-installation.html

Verifying Installation

Rust should now load automatically in new shells. To test this, run

```
$ rustc --version
```

You should be on at least 1.51 for this bootcamp.

https://doc.rust-lang.org/book/ch01-01-installation.html

Updating Rust Versions

Updating is easy:

```
$ rustup update
```

https://doc.rust-lang.org/book/ch01-01-installation.html

Cargo: Rust's build system & package manager

Cargo does *all the things* for you:

- Create a new project
- Manage dependencies
- Build project
- Build docs
- Run tests
- Build a release-optimized binary

https://doc.rust-lang.org/book/ch01-03-hello-cargo.html

Cargo: Rust's build system & package manager

Cargo does *all the things* for you:

- Create a new project
- Manage dependencies (day 6)
- Build project
- Build docs (day 3)
- Run tests (day 3)
- Build a release-optimized binary

https://doc.rust-lang.org/book/ch01-03-hello-cargo.html

Cargo: create a new project

Let's look at the basics:

```
$ cargo new hello_world
$ cd hello_world
```

Here we create a new project called "hello_world". By default it creates a new git repository, unless you are already in one.

https://doc.rust-lang.org/book/ch01-03-hello-cargo.html

Cargo: create a new project

The file *Cargo.toml* contains the project configuration.

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[dependencies]
```

https://doc.rust-lang.org/book/ch01-03-hello-cargo.html

Cargo: create a new project

The file *src/main.rs* is the default entrypoint for the binary, and defaults to

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

Looks like it wrote Hello World for us!

https://doc.rust-lang.org/book/ch01-03-hello-cargo.html

Cargo: build

The *build* command compiles the project and any dependencies.

```
$ cargo build
   Compiling hello_world v0.1.0 (file:///projects/hello_world)
    Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

Note that this is an unoptimized debug build, which is mainly useful for local testing.

https://doc.rust-lang.org/book/ch01-03-hello-cargo.html

Cargo: run

The built binary can be run manually, or with the *run* command.

```
$ ./target/debug/hello_world
Hello, world!

$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/hello_world`
Hello, world!
```

https://doc.rust-lang.org/book/ch01-03-hello-cargo.html

Cargo: build release

The *build* command has a *release* flag as well.

```
$ cargo build --release
```

This will be an optimized version, and is located in *target/release*.

https://doc.rust-lang.org/book/ch01-03-hello-cargo.html

Cargo: check

The *check* command is useful for checking the correctness of your work.

```
$ cargo check
   Checking hello_cargo v0.1.0 (file:///projects/hello_world)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

It does not generate a binary, so is generally much faster than the *build* command.

https://doc.rust-lang.org/book/ch01-03-hello-cargo.html

Hello World

Now, back to the source code.

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

Here we see some of the basic structure of Rust. Functions use the keyword *fn*. Every binary must have a main function, though libraries do not.

Printing in Rust is handled by the *println* builtin macro, and macros have an exclamation mark at the end of their name.

https://doc.rust-lang.org/book/ch01-02-hello-world.html#anatomy-of-a-rust-program

Procedural Programming

Let’s go through some basics. Rust has a several reserved keywords: https://doc.rust-lang.org/book/appendix-01-keywords.html

We’ll cover some today, but most are similar to other languages. If you want to use the same name as a keyword for something, you must use a raw identifier, prefixing the keyword with r#

```
fn r#move() {
    println!("We’re moving now");
}
fn main() {
    r#move();
}
```

Variables

Variables in Rust are by default immutable (think const in C++).

```
fn main() {
    let x = 5;
    x = 6; // compiler error here - `x` is immutable
}
```

https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html

Variables

You can make mutable, or editable, variables with the *mut* keyword.

```
fn main() {
    let mut x = 5;
    x = 6;
}
```

https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html

Constants

A constant in Rust is similar to a static const variable in C++, in that its value is fixed at compile time and cannot change.

```
const ANSWER: u8 = 42;
fn main() {
    println!(“The answer is {}”, ANSWER);
}
```

Note that a constant must also have a type defined immediately, instead of the compiler inferring the type.

https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html

Data Types

Rust is a statically typed language, but the compiler can usually infer what type you mean. Sometimes though, you'll need to add a type annotation to help it.

```
fn main() {
    let guess: u32 = "42".parse().expect("Not a number!");
}
```

Here, we're telling the parse function we'd like an unsigned int.

https://doc.rust-lang.org/book/ch03-02-data-types.html

Data Types

There are the standard unsigned and signed integers, floats, booleans, characters*, and string literals—as well as compound types like tuples and arrays.

Other data types are provided by the standard library, such as vector, string, and hash map. More about these on day 3.

* characters and strings are unicode in Rust, unless explicitly specified

https://doc.rust-lang.org/book/ch03-02-data-types.html

Functions

In Rust, a function's parameter and return types must be specified.

```
fn plus_one(x: i32) -> i32 {
    x + 1
}
```

The *return* keyword can be used, but another convention is to return the last expression in the function. This is done by removing the last semicolon.

If you keep the semicolon, the last expression is empty, denoted as ().

https://doc.rust-lang.org/book/ch03-03-how-functions-work.html

Control Flow - if/else

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("number less than 5");
    } else if number % 2 == 0 {
        println!("number divisible by 2");
    } else {
        println!("unknown number");
    }
}
```

https://doc.rust-lang.org/book/ch03-05-control-flow.html

Control Flow - if/else inline

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {}", number);
}
```

https://doc.rust-lang.org/book/ch03-05-control-flow.html

Control Flow - loops

```
fn main() {
    loop {
        println!("again!");
    }
}
```

A basic infinite loop.

https://doc.rust-lang.org/book/ch03-05-control-flow.html#repetition-with-loops

Control Flow - loops

```
fn main() {
    let mut counter = 0;
    let result = loop {
        counter += 1;
        if counter == 10 {
            break counter * 2;
        }
    };
    println!("The result is {}", result);
}
```

https://doc.rust-lang.org/book/ch03-05-control-flow.html#repetition-with-loops

Control Flow - while loop

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);

        index += 1;
    }
}
```

https://doc.rust-lang.org/book/ch03-05-control-flow.html#repetition-with-loops

Control Flow - for loop

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a.iter() {
        println!("the value is: {}", element);
    }
}
```

https://doc.rust-lang.org/book/ch03-05-control-flow.html#repetition-with-loops

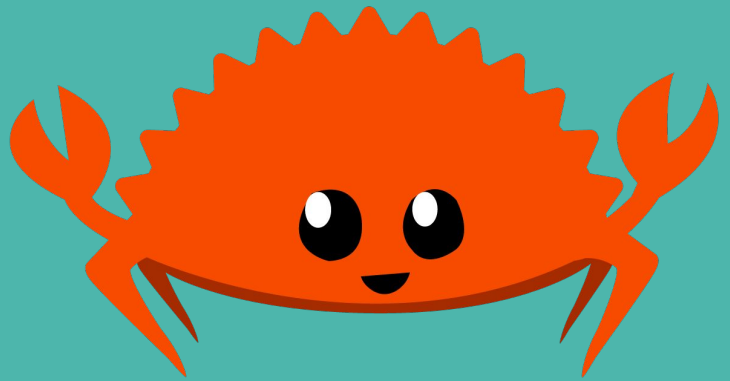
Control Flow - basic counter

```
fn main() {
    for number in (1..4).rev() {
        println!("{}!", number);
    }
    println!("LIFTOFF!!!");
}
```

The .. is a range expression, and we reverse it for this loop.

https://doc.rust-lang.org/book/ch03-05-control-flow.html#repetition-with-loops

End of Lecture



Exercises

Write a program to calculate the first N fibonacci numbers. Bonus points for writing both the loop and function implementations.

Write a program to find statistics for an array of numbers - min / max / mean / mode / ...

Write a program to determine if a number is prime or not.