URL

https://github.com/jlgerber/rustclass

What is Rust?

Rust advertises itself thusly:

```
Rust is blazingly fast and memory-efficient: with no runtime or garbage
collector, it can power performance-critical services, run on embedded
devices, and easily integrate with other languages.

Rust’s rich type system and ownership model guarantee memory-safety and
thread-safety — enabling you to eliminate many classes of bugs at compile-
time.

Rust has great documentation, a friendly compiler with useful error
messages, and top-notch tooling — an integrated package manager and build
tool, smart multi-editor support with auto-completion and type
inspections, an auto-formatter, and more.
```

Rust is a modern, systems level language whose peers include c and c++.

It is compiled, strongly typed, with low level access to the underlying hardware, but with certain safety guarantees that c and c++ cannot make. In a nutshell, Rust combines the control of c or c++ with the expressively and type richness of a Haskell or Ocaml. It is multi-paradigm, or maybe its own paradigm, being neither purely functional, nor classically OO.

So what do we have to work with in Rust?

A full complement of built in data types

ints of various sizes, signed and unsigned

- 8 bit i8 & u8
- 16 bit - i16 & u16
- 32 bit - i32 & u32
- 64 bit - i64 & u64
- 128 bit - i128 & u128
- isize & usize (per architecture)
- bools - true or false
- floating point types
- utf8 Strings and slices

Structs

These provide much of the same utility as classes in c++. Like in C++, Structs can have methods...

Algebraic Data types

Product Types

- tuples
- lists
- arrays
- hashmaps
- sets
- deques

Sum Types

- enum (but on steroids.... or cocaine) Enums can also have methods... (mind blown?)

Iterators

And iterator adapters. A full complement of functional programming goodness. (map, flatmap, fold, etc)

Closures

Because of course Rust has closures.

Traits

Super charged interfaces.

These are basically as close to typeclasses from Haskell this side of, well, Haskell. They are interfaces that can have default implementations, and that structs can implement, either manually, or, for some, via compiler directed derivations.

Generics

These come in two flavors - monomorphised, statically dispatched - dynamically dispatched - Generics may be constrained by traits - somewhat like c++'s upcoming concepts lite.

Lifetimes

One of Rusts unique features. In order for Rust to provide its sweeping guarantees, it needs a way to track and reason about how long a borrow lives. Rust provides lifetime variables to allow authors to help influence its understanding of lifetime. any variable starting with a tick is a lifetime variable. You will typically see single letter lifetime variables, like `'a`, although that isn't necessary.

Macros

These are not the "dangerous" c variety. Rather, these are passed through the compiler and maintain type information.

Deep testing integration

Inline tests which only get compiled on request.

By the way, all code in documentation also gets validated. No more writing docs with out of date examples.

Great Tooling

Rust comes with modern tooling, including a "build" system (cargo), a formatter (rustfmt), linting (), and documentation generation. It also comes in a couple flavors, including stable, beta, and nightly.

Decent Size Standard Library

A good standard library, and an even better ecosystem of crates (packages)

Great ecosystem of Crates

No matter what the need, Rust probably has you covered.

Installing Rust

So lets get started by installing rust.

Go to `rustup.rs` in a browser. Rustup is a tool to install and update rust. The website should detect your os and give you appropriate instructions.

Editor

Ok, lets start to get a feel for rust. I am using VS Code by the way. It has pretty good support for rust, including syntax highlighting, and debugging. But you can use whatever you like. Chances are you already have it installed. If you do, go to the extensions area and search for "rust".

Install `Rust` and `rust-analyzer`

Hello World

Lets create the project. Run the following in a shell to create a new one:

```
cargo new hello_world
```

cd into `hello_world` and take a look at what was created.

```
src/
    main.rs
Cargo.toml
.git/
.gitignore
```

Cargo.toml

This is your project manifest. It is where you will list dependencies for cargo to fetch, as well as set your version, and provide a number of other configuration options that we will get into later.

src/main.rs

This is your source code. lets crack it open and take a look.

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

Ok. Well, not too big eh? Lets run it, shall we?

```
cargo run
```

This will compile and run the project.

There are a number of other things which we can do. We can compile our project.

```
cargo build
```

As advertised, this builds an executable. But where is it? Take a look in the root of your project. You should see a directory called `target`.

If you snoop in there, you should find `hello_world`, under a `debug` directory followed by a platform directory (which will be different depending on the os that you run build from).

A couple things to note. First, by default, build compiles a `debug` build. This is a non-optimized build with symbols compiled in, perfect for debugging, but not so great for profiling.

To get an optimized build, run the following

```
cargo build --release
```

Go ahead and do this, and look in the target directory.

The Basics

Let's go back to our main.rs file and start building on it.

```
fn main() {
    let x = 1;
    println!("Hello World, {} time", x);
}
```

Variable Declaration and Assignment

Rust uses the `let` keyword for variable declaration. Under normal circumstances, Rust does not allow a variable to be uninitialized, so one usually both declares and assigns.

```
let x = 1;
```

So I said that Rust is strongly typed, but where is the type for x? Rust inherits a lot from functional languages, including a relatively advanced type system (it uses Hindley-Milner). If it can, Rust will infer the type based on static analysis of the source code.

However, we can annotate types, and we will often do so. We add type information *after* the variable name, unlike c or c++.

```
let x: i32 = 1;
```

Where do Variables Live?

By default, variable are put on the stack. Heap allocation is done explicitly, either by you, or by a library author on your behalf (The String type for example, allocates data on the heap)

There are various ways of putting data on the heap. Rust provides analogs to a lot of C++'s modern container wrapped pointers. For example, c++'s basic `std::unique_ptr` finds an analog in Rust's `std::boxed::Box`. To use a box, you do the following:

```
let departed = Box::new(String::from("Fred Willard"));
println!("we recently lost {}", departed);
```

Notice that I don't have to do anything special to dereference the boxed string. In c++, with `unique_ptr`, I would have to call the `get` method to access the data, for example. This bit of magic is handled by the `Deref trait`, which Box implements.

And note, just like in C++, there is no actual reason to box a String, as it already boxes its data.

Rust has a large number of heap allocating containers:

- Box
- Cow (copy on write)

- Rc (non-atomic reference counted)
- Arc (atomic reference counted)

We will look at them more later.

Move Semantics by Default

Rust uses move semantics for types which don't implement the `Copy trait`.

```
// create an owned string
let name = String::from("Mr. Meeseeks");

println!("my name is {}", &name);

// give it away
let hisname = name;
println!("His name is {}", &hisname);

// ERROR
println!("I said my name is {}", &name);
```

Mutation

In c or c++ variables are mutable by default and must be declared const to make them immutable. Rust turns that on its head. Variables are immutable by default, and must be declared otherwise. The following does *not* work:

```
// create an owned string
let name = String::from("Mr. Meeseeks");
println!("my name is {}", &name);
// ERROR: you cannot mutate name
name = String::from("Mr. Meeseeks2");
println!("Now my name is {}", name);
```

To make it work, we need to declare that name is mutable:

```
// create an owned string
let mut name = String::from("Mr. Meeseeks");
println!("my name is {}", &name);

name = String::from("Mr. Meeseeks2");

println!("Now my name is {}", name);
```

References

In addition to moving the contents of a variable into another variable, you can reference one variable from another using the very familiar (to c/c++) ampersand operator.

```
let x = String::from("this is it");
let y = &x;
println!("{}", y);
```

This is straightforward enough, although now we have to touch briefly on the most dreaded part of Rust - the "borrow checker". You see Rust has some pretty draconian rules designed to keep you safe. The first is this:

```
you can either have any number of immutable references to a variable, or a
single mutable reference, within a given scope.
```

Let that sink in

```
let mut foo = stringify!(what a deal);
let  bar = &mut foo;
*bar = stringify!(is it though?);
println!("{}", bar);
// this will error if you uncomment it
//println!("{}", foo);
```

Flow Control

if/else

Very similar to other languages except the condition doesn't need to be surrounded by parens:

```
let n = 5;

if n < 0 {
    print!("{} is negative", n);
} else if n > 0 {
    print!("{} is positive", n);
} else {
    print!("{} is zero", n);
}
```

While

again, like other languages. Once again, parens are not needed for the condition

```
let mut n = 1;

// Loop while `n` is less than 101
while n < 101 {
    if n % 15 == 0 {
        println!("fizzbuzz");
    } else if n % 3 == 0 {
        println!("fizz");
    } else if n % 5 == 0 {
        println!("buzz");
    } else {
        println!("{}", n);
    }

    // Increment counter
    n += 1;
}
```

for and range

Rust uses `..` to mean a half open range, and `..=` for a closed range

```
fn main() {
    // `n` will take the values: 1, 2, ..., 100 in each iteration
    for n in 1..101 {
        if n % 15 == 0 {
            println!("fizzbuzz");
        } else if n % 3 == 0 {
            println!("fizz");
        } else if n % 5 == 0 {
            println!("buzz");
        } else {
            println!("{}", n);
        }
    }
}
```

```
fn main() {
    // `n` will take the values: 1, 2, ..., 100 in each iteration
    for n in 1..=100 {
        if n % 15 == 0 {
            println!("fizzbuzz");
        } else if n % 3 == 0 {
            println!("fizz");
        } else if n % 5 == 0 {
            println!("buzz");
        } else {
            println!("{}", n);
            }
        }
    }
```

match

Rust provides pattern matching via the match keyword, which can be used like a C switch. However, it is more powerful, leveraging pattern matching. Which we will get into next week.

```
let number = 13;
    // TODO ^ Try different values for `number`

println!("Tell me about {}", number);
match number {
    // Match a single value
    1 => println!("One!"),
    // Match several values
    2 | 3 | 5 | 7 | 11 => println!("This is a prime"),
    // Match an inclusive range
    13..=19 => println!("A teen"),
    // Handle the rest of cases
    _ => println!("Ain't special"),
}

let boolean = true;
// Match is an expression too
let binary = match boolean {
    // The arms of a match must cover all the possible values
    false => 0,
    true => 1,
    // TODO ^ Try commenting out one of these arms
};

println!("{} -> {}", boolean, binary);
```

If/let & while/let

Related to destructuring. We will delve into these next week.

Functions

Functions start with the `fn` keyword, followed by parens containing zero or more arguments , then followed by an optional slim arrow identifying the return value. For example:

```
fn my_fn(name: &str) -> String {
    format!("Mr. {}", name)
}
```

Rust is an expression oriented language. By default, all scopes will implicitly return the result of their last expression. Semi-colons end the current expression and start a new expression. An empty expression evaluates to the unit `()` value.

In the above function, the format! macro is invoked without a semicolon, meaning that the function will implicitly return the format! function's results.

Structs

Lets get into structs a bit eh? First, lets see if we can create a LevelSpec struct. We will spend some time improving this, but for now:

```
struct LevelSpec {
    show: string,
    seq: string,
    shot: string
}
```

And here is how you create an instance of it

```
let mylevel = LevelSpec {
    show: String::from("DEV01"),
    seq: String::from("RD"),
    shot: String::from("0001")
}
```

What is one of the first things you probably want to know how to do? Right - Print it.

```
// this will fail
println!("{}", mylevel);
```

Ok, we need some help here. And guess what? Macros to the rescue. In addition to macro by example, there is another type of macro - the procedural macro. We will use a procedural macro to add the ability to print LevelSpec instances by `deriving` a trait implementation. There are two traits that are relevant - `std::fmt::Debug`, and `std::fmt::Display`. We are going to derive `Debug`. We start with our original definition and add a bit on top.

```
#[derive(Debug)]
struct LevelSpec {
    show: string,
    seq: string,
    shot: string
}
```

While we are at it, lets add a constructor function. Constructor functions aren't a formal part of the language; they are just associated functions that act like constructors in other languages.

To bind a function to a `struct`, we use the `impl` keyword.

```
impl LevelSpec {
    fn new(show: String, seq: String, shot: String) -> Self {
        Self {
            show, seq, shot
        }
    }
}
```

There are some new concepts to take in here. First, we can use `Self` to refer our type, although we don't have to.

Second, when a variable name matches the function parameter name, we do not have to explicitly call it out.

Anyway, now that we have defined a constructor function and derived debug, lets get to it.

```
let mylevel = LevelSpec::new(
    String::from("DEV01"),
    String::from("RD"),
    String::from("0001")
);

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

Notice the formatting twist? `{:?}` instructs the compiler to use the `Debug` trait instead of the `Display` trait. Throwing in a `#` makes Rust pretty print the Debug.

That is still a bit of typing eh? Let's try and clean this up a bit by sticking our toe into generics.

In addition to String, rust has &strs. These are references to string slices.

```
impl<T> LevelSpec {
    fn new(show: T, seq: T, shot: T) -> Self
    where
        T: Into<String>
    {
        Self {
            show: show.into(),
            seq: seq.into(),
            shot: shot.into()
        }
    }
}
```

So what is going on here? We define a generic var `T` on the `impl` block, and then say that show, seq, and shot are of type T. And oh by the way, T must implement the `Into<String>` trait. It just so happens both String and str implement said trait...

```
let mylevel = LevelSpec::new("DEV01", "RD", "0001");
```

Coming Up

- match statement, pattern matching, destructuring oh my
- module system
- enums in detail (aka product types)
- more on types and generics
- standard library
- multi-threading
- async
- common crates
    - serde for serialization deserialization
    - serde_json
    - structopt for command line arguments
    - nom for parsers
    - regex if you want (but you will probably use a parser/combinator instead)
    - interfacing with
        - postgres
        - rest / http
        - grpc

Resources

- The Rust Website
- crates.io - crate repo
- The Book
- This Week In Rust