Quick Trip Down A Rusty Road

Slides and questions are available at:

https://github.com/ahtoms/crush.git

About this talk

- Quick history and rationale of the language
- Rustup and cargo
- The memory model, ownership, immutable borrowing, mutable borrowing
- Loading and writing to files
- Traits and some trait patterns, documentation from the traits perspective
- Writing generic functions and structs
- Towards the end we will get set up with some programming exercises which you can also access from github
- ***Maybe get to*** coding patterns and documentation.

I cannot talk about everything, what you can follow up on!

- Dynamic guarantees (Rc, Cell, RefCell, Arc)
- Procedural Macros (Procedural macros are a wild beast)
- Writing automated tests (Can point you in the right direction though!)
- Lifetime Subtyping and annotations
- More prominent functional language features
- Anything else not outlined in the previous slide
- Functional patterns
- Function pointers

So what is this rust thing?

- It is a systems programming language (compiles to machine code + other targets like **WASM**)
- Addresses many memory safety issues and enforces a different memory model
- Static garbage collector (not a run-time garbage collector, the compiler does the work for you)
- Is a multi paradigm language, imperative, object oriented and functional
- Rust doesn’t find all memory issues but it removes a lot of common memory issues

What do you get from Rust?

- Greater sense of quality assurance from the compiler (The feeling of successfully compiling an application and working *correctly**)
- If has a steep learning curve but once you have a solid understanding, it can be very quick to develop applications with
- Easily audited language, any place where you have written unsafe block, you are able to quickly perform a **grep** and inspect
- Very portable ecosystem, easy build tool, toolchains can be easily accessed and setup through the **rustup** tool.

Time to get set up!

To makes things progress quicker (hopefully everyone brought their laptop to install!), please run the following, if you don’t trust (also good, never trust random links!) you can visit **rustup.rs**

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

Time to get set up!

To makes things progress quicker (hopefully everyone brought their laptop to install!), please run the following, if you don’t trust (also good, never trust random links!) you can visit **rustup.rs**

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

Allows us to have a version management system to deal with multiple targets and releases of the Rust toolchain

There is also the rust playground

You can visit the following link https://play.rust-lang.org/

This will provide you with a simple editor for writing rust programs and share the snippets with others.

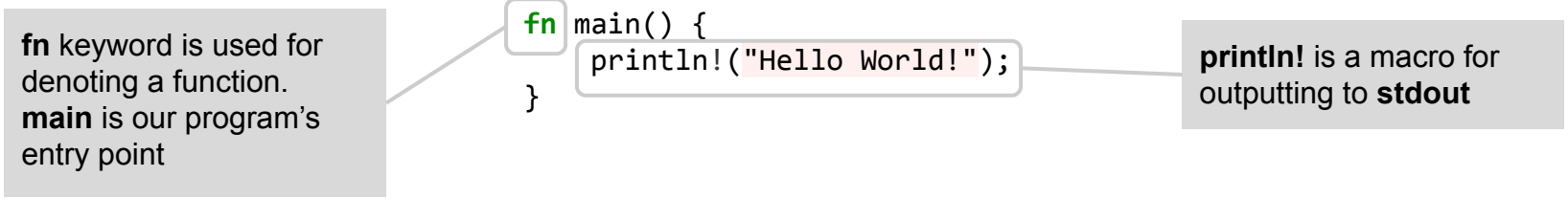
Hello World! Rust Style

Classic program where we will write a **Hello World** program and play with a few different examples.

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

Hello World! Rust Style

Classic program where we will write a **Hello World** program and play with a few different examples.



Hello World! Rust Style

Classic program where we will write a **Hello World** program and play with a few different examples.

fn keyword is used for denoting a function. **main** is our program's entry point

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

println! is a macro for outputting to **stdout**

Easy! We are seeing patterns from other languages form here

Going through the basics

The rust ecosystem incorporates a simple build system (Cargo) and we have language constructs that allow the following:

- Variable bindings
- Types
- Control Flow (if, match, for, while, loop)
- Comments

Cargo

Cargo is the build tool for Rust. It allows the following:

- Creating binary applications and libraries
- Specifying dependencies (commonly retrieved from https://crates.io)
- Provides a default template and allows you to structure your rust code
- Provides testing functionality, allow the programmer to create unit tests on their **crate**
- Specifying different builds, providing simple declaration in a **.toml** file

Once set up

Once **Rustup** has been set up on your system, you are able to utilise the **cargo** command.

```
> cargo new hello_world
> ls
hello_world
```

Simply, creates a new folder with **Cargo.toml**, **src** folder and **main.rs** within the src folder.

Building

We can simply move into the folder and build the current template. We are able to invoke the build command.

```
> cd hello_world
> cargo build
   Compiling hello_world v0.1.0 (/home/Rust/hello_world)
     Finished dev [unoptimized + debuginfo] target(s) in 0.25s
```

By default, this will compile the program with a **debug** build (slower but provides additional checking while testing).

Running

Like the build command we can immediately compile and run the application. By default there is a **hello world** print statement within the **main.rs** file.

```
> cd hello_world
> cargo run
   Compiling hello_world v0.1.0 (/home/Rust/hello_world)
     Running dev [unoptimized + debuginfo] target(s) in 0.25s
Hello, world!
```

Variables

The syntax for variables is derived from **OCaml**, declaring a variable using the **let** keyword marks the variable as **immutable** by default.

We are able to specify **mutability** on a variable by specifying the **mut** keyword after the **let** keyword.

```
let x = String::new();
let mut y = String::new();
```

We are able to enforce or provide assurance of a type for the compiler by adding a type annotation to a variable.

```
let x: String = String::new();
```

Types

We have the following list of primitive types within rust.

We are able to enforce or provide assurance of a type for the compiler by adding a type annotation to a variable.

```
let a: i8 = 127;
let b: i16 = 32321;
let c: i32 = 9001223;
let c: i64 = 900122322;
let d: u8 = 255;
let e: u16 = 65535;
let f: u32 = 9776788;
let g: u64 = 977678822;
let h: isize = 100;
let f: usize = 500;
```

```
let a: f32 = 2.1454;
let b: f64 = 6.3432224;
let c: bool = true;
let d: char = ‘D’;
let e: [bool, 2] = [true, false];
let f: &[bool, 1] = &e[0..1];
```

Loops!

Rust's loops have rather distinct applications. However, like most C-like languages, **while** simply expects a **boolean** expression. There is a language construct that allows for executing until a break.

```
while boolean_expression {
   // while loop body, stops executing when expression is false
}
```

```
loop {
   // Loop body, will run forever or until break
}
```

Loops!

However there are more complex loops within Rust. We are able to use a pattern within and assign the result to a binding.

```
while let Pattern = binding {
   //Allows you to operate on the binding until the expression no longer
   //Meets the type expression
}
```

Similar to python, the for loop is commonly used with an iterator (or an iterator pattern resolved at compilation)

```
for binding in iterator {
   // Loop body, will run forever or until break
}
```

The memory model

The memory model can be broken down into 3 different parts.

- Ownership
- Immutable Borrow (or just Borrow)
- Mutable Borrow

The memory model is unlike many other languages and is a point of contention for a lot of programmers coming from languages which are C-derivatives.

Ownership

The most obvious one is Ownership of the object. This is usually shown in functions as parameters without & prefixed to the type. When a function is taking ownership of an object it is responsible for it's lifetime at that point. If it returns the object in the end, the caller will take over ownership of that object, if it does not, the object will be **de-allocated** at that point.

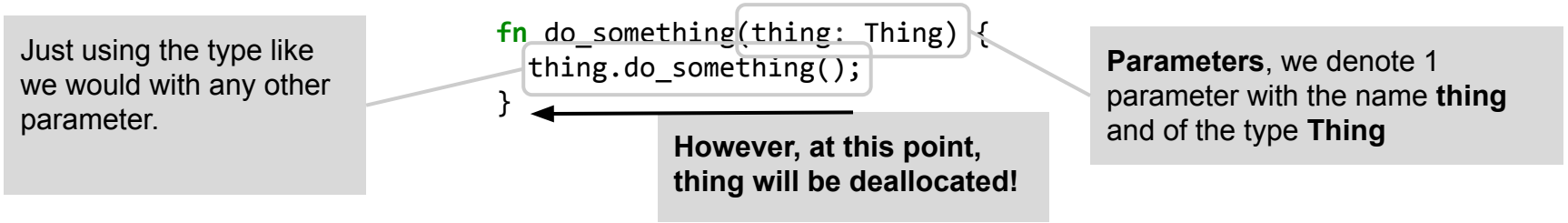
Analogy:

main(): Okay **g()**, Here's a **Car**, you can do whatever you want!

g(): Sure thing! I'll drive it first and then put it in a ditch!

Ownership

So, in the following function denotes a list of parameters, binding name with Type associated.



But it isn't just function scope itself, it is also when we apply an extra scope layer, if statement, loop or even match statement that we have to consider the life time of the object.

Borrows

The next is Immutable Borrowing of the object. This creates a view of an object, allow the object to be read but we cannot **mutate** the object. Typically there are properties of objects (or the object itself) that can

Analogy:

main(): Okay **g()** and **p()**, You may look at the Car, but you cannot drive it away since you **don’t own it!**

g(): Okay! I’ll like to inquire about the price and speed of the Car!

Borrows

So, In the previous function we denoted the type by itself, but now we are going to introduce an operator that indicates that it is a reference (borrow).

```
fn main() {
  let x = String::new()
  borrow_example(&x);
  borrow_example(&x);
}

fn borrow_example(object_ref: &String) {
  let o = object_ref;
}
```

String type but with **&,** we have specified it as a reference to a String type

We declared a variable, x which has ownership of a **String**. When we pass it to **borrow_example** we are passing the reference, it will not be consumed by the function.

Borrows

So, In the previous function we denoted the type by itself, but now we are going to introduce an operator that indicates that it is a reference (borrow).

```
fn main() {
  let x = String::new()
  borrow_example(x);
  borrow_example(x);
}

fn borrow_example(object_ref: String) {
  let o = object_ref;
}
```

Indicating that the function takes ownership of the object, disallows us from passing a reference (incorrect type specified)

This will fail, as the first time we called borrow_example, the function will have consumed the value.

Mutable Borrows

Importantly is **Mutable Borrowing** this is where you are able to borrow the object and change it. But there is a strict rule with this. Only one function is allowed to mutably borrow.

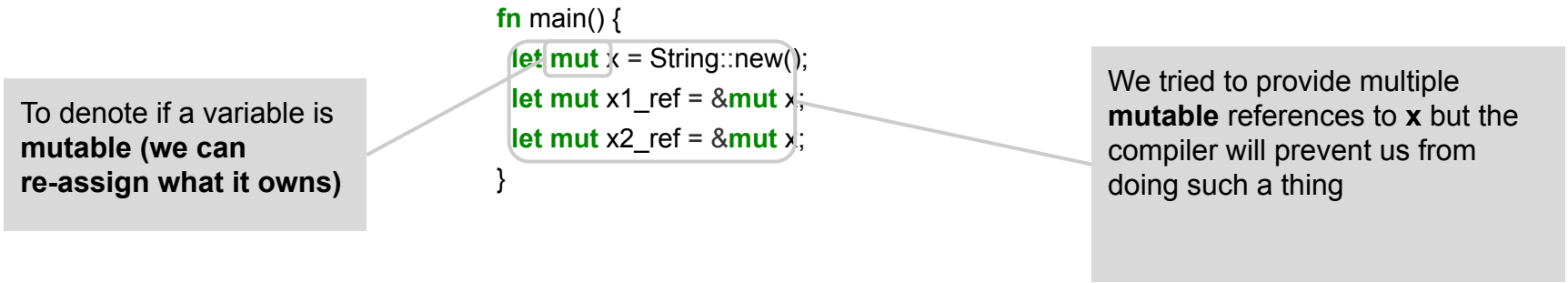
Analogy:

main(): Okay **g()**, You may borrow the car to drive, but you must return the car afterwards with full tank of petrol.

g(): Okay! I'll remember to fill it up once I am done!

Mutable Borrows

Given a **mutable** binding, we can then provide a mutable reference to your to the String.



Okay, let’s step it up!

We want to make our program simply greet someone, but we will assume we have a name assigned already.

```
fn greet(str: String) {
    println!("Hello, {}!", str);
}

fn main() {
    let name: String = String::from("Jeff");
    greet(name);
}
```

Okay, let’s step it up!

We want to make our program simply greet someone, but we will assume we have a name assigned already.

New function called **greet**, **takes ownership** of a **String** type under the name **str**.

```
fn greet(str: String) {
    println!("Hello, {}!", str);
}

fn main() {
    let name: String = String::from("Jeff");
    greet(name);
}
```

Okay, let’s step it up!

We want to make our program simply greet someone, but we will assume we have a name assigned already.

New function called **greet**, **takes ownership** of a **String** type under the name **str**.

```
fn greet(str: String) {
    println!("Hello, {}!", str);
}

fn main() {
    let name: String = String::from("Jeff");
    greet(name);
}
```

We use a format specifier in our **println!** macro and specify **str** as an argument

Okay, let's step it up!

We want to make our program simply greet someone, but we will assume we have a name assigned already.

New function called **greet**, **takes ownership** of a **String** type under the name **str**.

```
fn greet(str: String) {
    println!("Hello, {}!", str);
}

fn main() {
    let name: String = String::from("Jeff");
    greet(name);
}
```

We use a format specifier in our **println!** macro and specify **str** as an argument

We have created a variable and specified the type as **String**. This has been assigned and **name** owns the String **"Jeff".**

Okay, let's step it up!

We want to make our program simply greet someone, but we will assume we have a name assigned already.

New function called **greet**, **takes ownership** of a **String** type under the name **str**.

```
fn greet(str: String) {
    println!("Hello, {}!", str);
}

fn main() {
    let name: String = String::from("Jeff");
    greet(name);
}
```

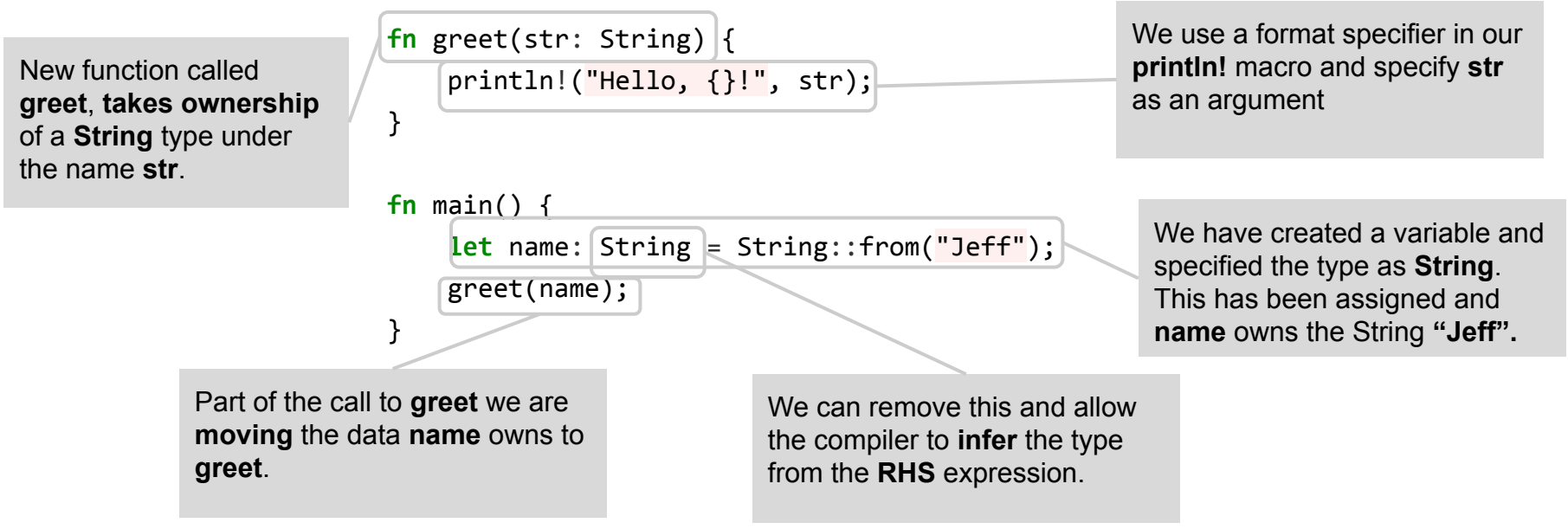
We use a format specifier in our **println!** macro and specify **str** as an argument

We have created a variable and specified the type as **String**. This has been assigned and **name** owns the String **"Jeff".**

We can remove this and allow the compiler to **infer** the type from the **RHS** expression.

Okay, let's step it up!

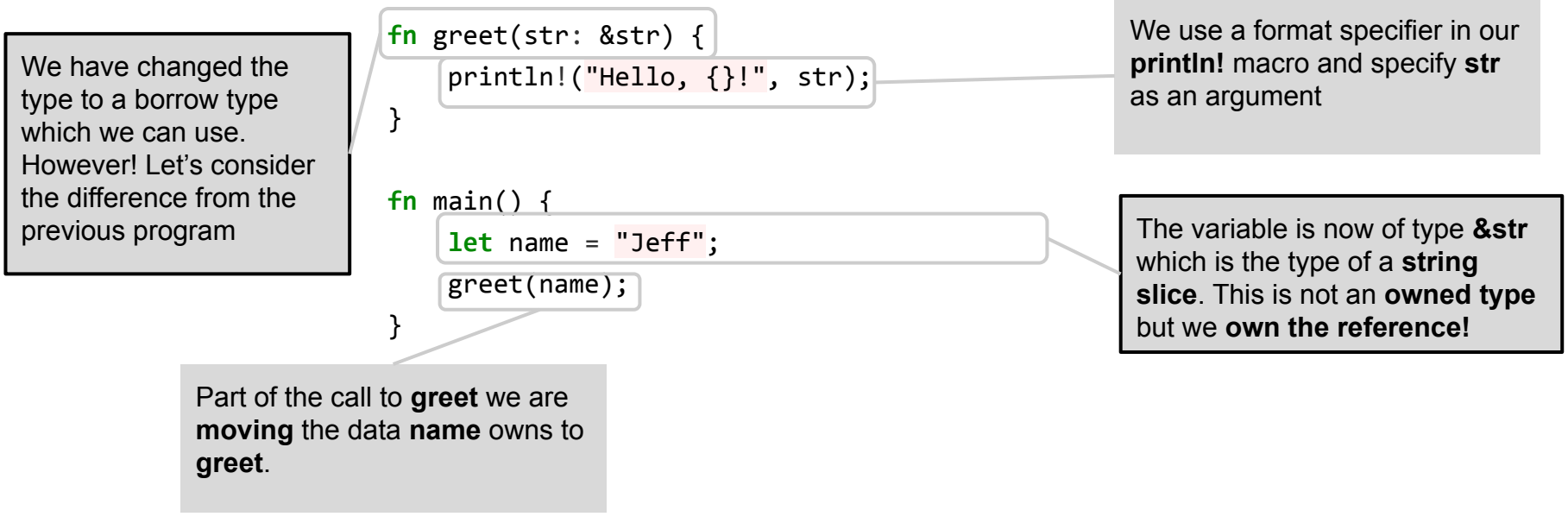
We want to make our program simply greet someone, but we will assume we have a name assigned already.



Simple example but we can start breaking down the memory model with this

Okay, let's step it up!

We want to make our program simply greet someone, but we will assume we have a name assigned already.



Okay, let’s step it up!

We want to make our program simply greet someone, but we will assume we have a name assigned already.

```
fn greet(str: String) {
    println!("Hello, {}!", str);
}

fn main() {
    let name: String = String::from("Jeff");
    greet(name);
    println!("Name is, {}!", name);
}
```

I’m going to add this line here! Let’s see what happens!

Okay, let’s step it up!

We want to make our program simply greet someone, but we will assume we have a name assigned already.

Oh! We got a compiler error!

```
fn greet(str: String) {
    println!("Hello, {}!", str);
}

fn main() {
    let name: String = String::from("Jeff");
    greet(name);
    println!("Name is, {}!", name);
}
```

I’m going to add this line here! Let’s see what happens!

But what about writing data structures?

Data structures are hard (kinda)

Writing a linked list poses a lot of challenges as you will be directly confronting the memory model of rust. A singly linked list is doable within the constraints but a doubly linked list poses a number of issues (namely, aliasing and mutability).

However we will have a discussion and aim at a stack at the end!

Save me from myself

First thing to be clear about, the compiler isn’t just simply taking your source code and translating it into machine code (or the target platform). The compiler is performing checks to ensure if adheres to:

- Memory model (lifetimes)
- Type information can be inferred or has been declared
- Bounds checks on arrays
- Type sizes with generics
- Any attempts of usage after deallocation

Why so many checks?

Because without adequately checking we could be making an unsafe or unsound assumption. The language and standard library attempts to skew the programmer to handle except cases.

For example

- Opening a file requires handling the case that a file cannot be accessed or cannot be opened (permissions)
- A function that operates on a list may not be able to return an element, the function return type can specify a **Result** or **Option** which the programmer is forced to handle (typically through a match statement).

When your program **panics**

Your program will panic when you have encountered a run-time error. This is similar to a C++ or Java **runtime exceptions** but something that you shouldn’t handle.

Idioms

- Functions that ***cannot*** guarantee an object of a type returned should utilise the **Result** or less preferred **Option** type. **Result** allows the programmer to describe an error where an **Option** does not provide any information as to why retrieval failed.

- Functions returning a **Result** or **Option** type can be associated with a **?** operator after the call. This is common syntax for immediately returning if the call encounters **Err** or **None** on **Result** and **Option** respectively.

Idioms

We will soon include the **Option** type as part of our function but given a problem where we want to search for an entry in a list and we want to return the entry, we may instinctively specify the return the Type within the list.

```
fn find(list: Vec<String>, f: &str) -> String {
    for e in list {
        if e == f {
            return e.clone();
        }
    }
    String::from("No String found!")
}


fn main() {
    let entries = vec![
        "William".to_string(),
        "Jonathan".to_string(),
        "Benny".to_string(),
        "Veronica".to_string(),
        "Chandler".to_string(),
        "Beatrice".to_string(),
        "Paul".to_string()
    ];

    let o = find(entries, "Veronica");
    println!("{}", o);
}
```

Idioms

We will soon include the **Option** type as part of our function but given a problem where we want to search for an entry in a list and we want to return the entry, we may instinctively specify the return the Type within the list.

Create a **Vector** of that holds a **String** type. We want to search for an entry within that vector

```
fn find(list: Vec<String>, f: &str) -> String {
    for e in list {
        if e == f {
            return e.clone();
        }
    }
    String::from("No String found!")
}


fn main() {
    let entries = vec![
        "William".to_string(),
        "Jonathan".to_string(),
        "Benny".to_string(),
        "Veronica".to_string(),
        "Chandler".to_string(),
        "Beatrice".to_string(),
        "Paul".to_string()
    ];

    let o = find(entries, "Veronica");
    println!("{}", o);
}
```

Remember, Rust does not have **null**

Idioms

We will soon include the **Option** type as part of our function but given a problem where we want to search for an entry in a list and we want to return the entry, we may instinctively specify the return the Type within the list.

```
fn find(list: Vec<String>, f: &str) -> String {
    for e in list {
        if e == f {
            return e.clone();
        }
    }
    String::from("No String found!")
}


fn main() {
    let entries = vec![
        "William".to_string(),
        "Jonathan".to_string(),
        "Benny".to_string(),
        "Veronica".to_string(),
        "Chandler".to_string(),
        "Beatrice".to_string(),
        "Paul".to_string()
    ];

    let o = find(entries, "Veronica");
    println!("{}", o);
}
```

We iterate through the list and treat each entry through the binding **e**.

Once found, we can return that type.

But the case where we cannot find the element seems counter intuitive.

Create a **Vector** of that holds a **String** type. We want to search for an entry within that vector

Idioms

We have now specified an **Option** return type for **find**. We can return two values from such a type.

- Some
- None

Some is able to hold a value while None represents absence of value.

```
fn find(list: Vec<String>, f: &str) -> Option<String> {
    for e in list {
        if e == f {
            return Some(e.clone());
        }
    }
    None
}

fn main() {
    let entries = vec![
        "William".to_string(), "Jonathan".to_string(),
        "Benny".to_string(), "Veronica".to_string(),
        "Chandler".to_string(), "Beatrice".to_string(),
        "Paul".to_string()
    ];
    let o = find(entries, "Veronica");
    match o {
        Some(v) => { println!("{}", v); },
        None => { println!("Cannot find element"); }
    }
}
```

Idioms

Updated return type, this will allow Some or None. The found result will be wrapped in a **Some** type.

We have now specified an **Option** return type for **find**. We can return two values from such a type.

- Some
- None

Some is able to hold a value while None represents absence of value.

```
fn find(list: Vec<String>, f: &str) -> Option<String> {
    for e in list {
        if e == f {
            return Some(e.clone());
        }
    }
    None
}

fn main() {
    let entries = vec![
        "William".to_string(), "Jonathan".to_string(),
        "Benny".to_string(), "Veronica".to_string(),
        "Chandler".to_string(), "Beatrice".to_string(),
        "Paul".to_string()
    ];
    let o = find(entries, "Veronica");
    match o {
        Some(v) => { println!("{}", v); },
        None => { println!("Cannot find element"); }
    }
}
```

Idioms

Updated return type, this will allow Some or None. The found result will be wrapped in a **Some** type.

We have now specified an **Option** return type for **find**. We can return two values from such a type.

- Some
- None

Some is able to hold a value while Nor
of value.

Used a match statement on the option type, if the type is matched to Some, we can extract value out of it.

```
fn find(list: Vec<String>, f: &str) -> Option<String> {
    for e in list {
        if e == f {
            return Some(e.clone());
        }
    }
    None
}

fn main() {
    let entries = vec![
        "William".to_string(), "Jonathan".to_string(),
        "Benny".to_string(), "Veronica".to_string(),
        "Chandler".to_string(), "Beatrice".to_string(),
        "Paul".to_string()
    ];
    let o = find(entries, "Veronica");
    match o {
        Some(v) => { println!("{}", v); },
        None => { println!("Cannot find element"); }
    }
}
```

Option Demo

Where to find documentation

Majority of documentation of the Rust ecosystem can be found at the following address https://doc.rust-lang.org/std/index.html

There is also a collection of documentation held at the following website which maintains documentation for a crates here https://docs.rs/

The documentation is easily searchable and provides details about the functions, structs, traits and implementation. The standard library provides the source code point which will allow you to inspect the implementation.

The aggregation and traits!

A powerful mechanism of the language is the type system. This allows the compiler to provide checks on type usage, ensure return types are correct and allow us to associate type information. Rust provides aggregation constructs type association constructs through the following:

- Structs
- Enums
- Tuples
- Traits

Struct and Implementations

We create our own aggregate types within Rust through the **struct** keyword. Similarly to **C** and **C++**, we are able to define properties on a struct.

```
struct Cat {
    name: String,
    age: u32,
}

fn print_cat(c: &Cat) {
    println!("Name: {}, Age: {}", c.name, c.age);
}

fn main() {
    let felix = Cat { name: String::from("Felix"), age: 10 };
    let garfield = Cat { name: String::from("Garfield"), age: 8 };
    print_cat(&felix);
    print_cat(&garfield);
}
```

Struct and Implementations

We create our own aggregate types within Rust through the **struct** keyword. Similarly to **C** and **C++**, we are able to define properties on a struct.

Able to define a **Cat** struct with the following fields **name** and **age.**

```
struct Cat {
    name: String,
    age: u32,
}
```

We borrow the cat and print out the data.

```
fn print_cat(c: &Cat) {
    println!("Name: {}, Age: {}", c.name, c.age);
}
```

```
fn main() {
    let felix = Cat { name: String::from("Felix"), age: 10 };
    let garfield = Cat { name: String::from("Garfield"), age: 8 };
    print_cat(&felix);
    print_cat(&garfield);
}
```

We can do this differently though

Struct and Implementations

We are able to implement functionality to a type (this can be **struct, trait** or **enum**). Using the **impl** keyword we are able to find functionality to a type and implement trait functionality to a struct or enum.

```
struct Cat { name: String,  age: u32, }

impl Cat {
    fn info(&self) {
        println!("Name: {}, Age: {}", self.name, self.age);
    }
}

fn main() {
    let felix = Cat { name: String::from("Felix"), age: 10 };
    let garfield = Cat { name: String::from("Garfield"), age: 8 };
    felix.info();
    garfield.info();
}
```

Struct and Implementations

We are able to implement functionality to a type (this can be **struct, trait** or **enum**). Using the **impl** keyword we are able to find functionality to a type and implement trait functionality to a struct or enum.

```
struct Cat { name: String,  age: u32, }

impl Cat {
    fn info(&self) {
        println!("Name: {}, Age: {}", self.name, self.age);
    }
}

fn main() {
    let felix = Cat { name: String::from("Felix"), age: 10 };
    let garfield = Cat { name: String::from("Garfield"), age: 8 };
    felix.info();
    garfield.info();
}
```

Take note, we are performing an immutable borrow at this point.

impl Cat allows us to specify functionality that will operate on this type.

Since the struct implements the info method for the type we can use this method like so.

Enum

Similarly to other languages, an enum type can be utilised as a simple label. However rust allows us to incorporate more complex logic, allowing us to perform match statements on the bindings.

```
enum IPAddr {
    V4{address: (u8, u8, u8, u8)},
    V6(String),
}
```

Enum

Similarly to other languages, an enum type can be utilised as a simple label. However rust allows us to incorporate more complex logic, allowing us to perform match statements on the bindings.

```
enum IPAddr {
    V4{address: (u8, u8, u8, u8)},
    V6(String),
}
```

We can simply define enum as a label, or incorporate properties, similar to structs.

Enum

Functionality can be attached to an enum type (well, any type at least) to suit our problem. In this instance where we have an IP Address, we can provide a function to convert it to a IPV4 or IPV6 version

```
impl IPAddr {
    fn from_string(addr: String) -> Option<IPAddr> {
        if addr.len() <= 15 {
            let array_parts: Vec<&str> = addr.split(".").collect();
            if array_parts.len() == 4 {
                return Some(IPAddr::V4 {
                    address : (array_parts[0].parse::<u8>().unwrap(),
                    array_parts[1].parse::<u8>().unwrap(), array_parts[2].parse::<u8>().unwrap(),
                    array_parts[3].parse::<u8>().unwrap())
                });
            }
        } else {
            return Some(IPAddr::V6(addr));
        }
        None
    }
}
```

Enum

Functionality can be attached to an enum type (well, any type at least) to suit our problem. In this instance where we have an IP Address, we can provide a function to convert it to a IPV4 or IPV6 version

Similarly to structs, we can use the impl keyword and implement a method that allows us to convert a String to the IPAddr type.

```
impl IPAddr {
    fn from_string(addr: String) -> Option<IPAddr> {
        if addr.len() <= 15 {
            let array_parts: Vec<&str> = addr.split(".").collect();
            if array_parts.len() == 4 {
                return Some(IPAddr::V4 {
                    address : (array_parts[0].parse::<u8>().unwrap(),
                    array_parts[1].parse::<u8>().unwrap(), array_parts[2].parse::<u8>().unwrap(),
                    array_parts[3].parse::<u8>().unwrap())
                });
            }
        } else {
            return Some(IPAddr::V6(addr));
        }
        None
    }
}
```

Enum

Functionality can be attached to an enum type (well, any type at least) to suit our problem. In this instance where we have an IP Address, we can provide a function to convert it to a IPV4 or IPV6 version

We can then match on this result, providing control flow based on the type we have converted it to.

```
fn main() {

    match IPAddr::from_string("127.0.0.1".to_string()) {
        Some(result) => {
            match result {
                IPAddr::V4 { address } => { println!("{}.{}.{}.{}", address.0,
                                address.1, address.2, address.3); },
                IPAddr::V6(address) => { println!("IPV6 Address"); }
            }
        },
        None => { println!("Invalid Address"); }
    }
}
```

Traits

As part of my own journey, I have found this concept to be the part where the standard library and patterns clicked for me. It allow me to observe how different types worked from the documentation and how certain implicit behaviour is defined.

Traits

We have defined a trait called **MakeNoise**, this allows us to provide type association with the struct types **Dog, Wolf** and **SmartDog.**

```
trait MakeNoise {
    fn bark(&self);
}

struct Dog {
    name: String
}

impl MakeNoise for Dog {

    fn bark(&self) {
        println!("Bark bark!");
    }

}

struct Wolf;

impl MakeNoise for Wolf {

    fn bark(&self) {
        println!("Awoooooo!");
    }

}
```

```
struct SmartDog {
    name: String,
}

impl MakeNoise for SmartDog {

    fn bark(&self) {
        println!("Woof, my name is {}",
            self.name);
    }

}
```

Traits

We have defined a trait called **MakeNoise**, this allows us to provide type association with the struct types **Dog, Wolf** and **SmartDog.**

```
trait MakeNoise {
    fn bark(&self);
}

struct Dog {
    name: String
}

impl MakeNoise for Dog {

    fn bark(&self) {
        println!("Bark bark!");
    }

}

struct Wolf;

impl MakeNoise for Wolf {

    fn bark(&self) {
        println!("Awoooooo!");
    }

}
```

```
struct SmartDog {
    name: String,
}

impl MakeNoise for SmartDog {

    fn bark(&self) {
        println!("Woof, my name is {}",
            self.name);
    }

}
```

Defined a trait that contains a method signature. This method must be defined by for a type that implements it. (typically known as an **abstract method**)

As part of impl, we can specify the **Trait** we are implementing and the **type** we are implementing it for.

Traits

Given a simple main function we can simply construct and use the bark method like any other type.

```
fn main() {
    let dog = Dog { name : String::from("Rover") };
    let wolf = Wolf;
    let lassie = SmartDog { name : String::from("Lassie") };

    dog.bark();
    wolf.bark();
    lassie.bark();
}
```

Traits

But, if we were to define a method where we define a trait type as the parameter, we can then pass all three arguments to the following method and execute bark. Since each type implements methods defined in **MakeNoise**, we can use the following method

```
fn bark(noisy: &MakeNoise) {
    noisy.bark();
}
fn main() {
    let dog = Dog { name : String::from("Rover") };
    let wolf = Wolf;
    let lassie = SmartDog { name : String::from("Lassie") };
    bark(&dog);
    bark(&wolf);
    bark(&lassie);
}
```

Traits Demo

What else to learn?

Everything specified earlier **and:**

- Crates (libraries) that are used with your application
- Custom Build Scripts, when creating a foreign function interface or packaging an application you will probably need to write a build.rs file in you cargo directory
- There is typically a binding to a C library on **https://crates.io** which is the default repository for rust libraries
- **unsafe** keyword, this is part of building safe abstractions

Further Resources

- The Rust Programming Language, https://doc.rust-lang.org/book/
- Embedded Rust, https://rust-embedded.github.io/book/
- Command Line Applications, https://rust-lang-nursery.github.io/cli-wg/
- Cargo Book, https://doc.rust-lang.org/cargo/index.html
- Rust Doc Book, https://doc.rust-lang.org/rustdoc/index.html

Common crates

- Serde, Serialization/Deserialization crate for many data formats, https://serde.rs/
- Rand, Random Number Generator, https://crates.io/crates/rand
- Actix-Web, Performant web backend framework, https://actix.rs/
- Diesel as a SQL-ORM with a cli-merge command, http://diesel.rs/
- Hyper, HTTP Client or Server, https://hyper.rs/
- Rusqlite, Rust Binding for SQLite, https://docs.rs/rusqlite/0.19.0/rusqlite/

Special Thanks

- **Xueliang Bai**, for letting me come here and conduct this crash course
- Core Rust Team, for continually improving the language
- Rust Community for their continued support of the ecosystem
- Authors of the Rust Documentation and Book, Steve Klabnik and Carol Nicols

Questions?