Intro to Rust

Benjamin Owad, David Rudo, and Connor Tsui

Why Rust?

TODO A few slides about this...

Rust seems awesome!

But how do I actually use Rust?

Hello World!

To create an executable, we need a `main` function:

```
// main.rs

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

To compile `main.rs`, run this in the terminal:

```
$ rustc main.rs
```

Cargo

Rust has its own build system and package manager called **Cargo**.

- Cargo is built into Rust, so no `make` files or third-party build systems
- Manages packages similar to `pip` for `python` or `npm` for `node.js`

Creating a new project with `cargo new`

To create a new cargo project called `hello_cargo`, run this in the terminal:

```
$ cargo new hello_cargo
$ cd hello_cargo
```

- You will find a few important things
  - `src/main.rs`
  - `Cargo.toml`
  - `.git` repository and `.gitignore`
- We will come back to those last two in the future

Building your new project with `cargo build`

To build your project:

```
$ cargo build
   Compiling hello_cargo v0.1.0 (<path>/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 1.00s
```

- This creates an executable file at `target/debug/hello_cargo`
- To build for release, run `cargo build --release`
- What if we want to actually run this executable?
  - Can run `./target/debug/hello_cargo --`, but this is verbose

Running your new project with `cargo run`

To run your project:

```
$ cargo run
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target/debug/hello_cargo`
Hello, world!
```

- Note that if you compiled with `cargo build` right before, you wouldn't see the `Compiling hello_cargo ...` message
- To run in release, run `cargo run --release`

Check if your project is okay with `cargo check`

To check if your code compiles:

```
$ cargo check
   Checking hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
```

- Must faster than `cargo build` since it doesn't build the executable
- Useful when programming to check if your code still compiles!

Cargo recap

- We can create a project using `cargo new`
- We can build a project using `cargo build`
- We can build and run a project in one step using `cargo run`
- We can check a project for errors using `cargo check`
- Cargo stores our executable in the `target/debug` directory

Variables

Variables are values bound to a name. We define variables with the `let` keyword.

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
}
```

Immutability

All variables in Rust are *immutable* by default.

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

- What happens when we try to compile this?

Immutability

When we try to compile, we get this error message:

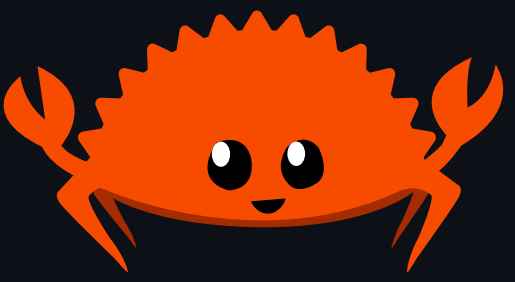
```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     println!("The value of x is: {}", x);
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable
```

- Let's follow the compiler's advice!

Mutability

To declare a variable as mutable, we use the `mut` keyword.

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```



When we run the program now, we now get this:

```
$ cargo run
    <-- snip -->
The value of x is: 5
The value of x is: 6
```

Const variables

Like immutable variables, *constants* are values bound to a name

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

- Constants cannot be `mut`
- Constants must have an explicit type

Scopes and Shadowing

You can create nested scopes within functions.

```
fn main() {
    let x = 5;

    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {x}");
    }

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

- Let's dissect this!

```
let x = 5;
let x = x + 1;
{
    let x = x * 2;
    println!("The value of x in the inner scope is: {x}");
}
println!("The value of x is: {x}");
```

- `x` is bound to `5` first
- A new variable `x` is created and bound to `x + 1 = 6`
- An inner scope is created with the opening curly brace `{`
- The third `let` statement shadows `x`
- The shadowed `x` is set to `x * 2 = 12`
- The inner scope ends with the closing curly brace `}`
- `x` returns to being 6 again

```
let x = 5;
let x = x + 1;
{
    let x = x * 2;
    println!("The value of x in the inner scope is: {x}");
}
println!("The value of x is: {x}");
```

Let's run this now:

```
$ cargo run
    <-- snip -->
The value of x in the inner scope is: 12
The value of x is: 6
```

Shadowing vs Mutability

- We get a compile time error if we try to modify a non-`mut` variable
- Using `let` multiple times allows a few transformations on a value but keep it immutable
- Shadowing effectively creates a new variable, so it can change type

Shadowing vs Mutability

Shadowing:

```
let spaces = "   ";
let spaces = spaces.len();
```

Mutability:

```
let mut spaces = "   ";
spaces = spaces.len();
```

- The second one does not compile!

Data Types

Like most langauges, there are two main types of Types.

- Scalar Types
  - Integers
  - Floating-Points
  - Boolean
  - Character
- Compound Types
  - Tuples
  - Arrays

Integers

Rust has similar integer types you would expect to see in C.

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

Floating-Points

Rust has both a 32-bit and 64-bit floating-point type.

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

Numeric Operations

```
fn main() {
    // addition
    let sum = 5 + 10;

    // subtraction
    let difference = 95.5 - 4.3;

    // multiplication
    let product = 4 * 30;

    // division
    let quotient = 56.7 / 32.2;
    let truncated = -5 / 3; // Results in -1

    // remainder
    let remainder = 43 % 5;
}
```

Booleans

A *boolean* in Rust has two values `true` and `false` (as in most other languages). Booleans are always 1 byte in size.

```
fn main() {
    let t = true;

    let f: bool = false; // with explicit type annotation
}
```

Characters

Rust has a UTF-8 character type `char`. Use `char` with single quotes. Due to `char` being UTF-8, a `char` is at least 1 byte and up to 4 bytes in length.

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
    let heart_eyed_cat = '😻';
}
```

- We will talk more about this and UTF-8 in the future

Tuples

A *tuple* is a way of grouping together a number of values with a variety of types.

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

Tuples

You can destructure tuples like so:

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

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

Tuples

You can also access specific elements in the tuples:

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

Arrays

To store a collection of multiple values, we use *arrays*.

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let months = ["January", "February", "March", "April", "May", "June", "July",
                  "August", "September", "October", "November", "December"];
}
```

- Unlike tuples, all elements must be the same type.
- The number of elements is always fixed (stack allocated)
    - If you want a collection that grows and shrinks, use a vector (lecture 4)

Arrays

We define an array's type be specifying the type of the elements and the length of the array.

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

We can also initialize the array such that every element has the same value.

```
let a = [3; 5];
// Same as writing
// let a = [3, 3, 3, 3, 3];
```

Arrays

To access an array element, we use square brackets.

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

- Rust will check if the index is in bounds at runtime
  - This is *not* done in C/C++

Functions

Like all languages, Rust has functions.

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

fn another_function() {
    println!("Another function.");
}
```

```
$ cargo run
    <-- snip -->
Hello, world!
Another function.
```

Functions

All parameters / arguments to functions must be given an explicit type.

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {}{}", value, unit_label);
}
```

```
$ cargo run
    <-- snip -->
The measurement is: 5h
```

Statements and Expressions

All functions are a series of statements optionally ending in an expression

- **Statements** are instructions that do some action and don't return a value
  - `let y = 6;` is a statement and does not return a value
  - You cannot write `x = y = 6`
- **Expressions** evaluate to a resultant value
  - `2 + 2` is an expression
  - Calling a function is an expression
  - A scope is an expression
    - This implies scopes return values!
- If you add a semicolon to an expression, it turns into a statement

Statements and Expressions

Observe the following code:

```
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

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

- Notice that there is no semicolon after `x + 1`
- Scopes return the value of their last expression
- Since functions are scopes, they can also return values!

Return values

Functions can return values back to the callers.

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

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

- Functions must have a specific return value or none at all
  - No return type is equivalent to the unit type `()`
- Notice again that there is no semicolon after `x + 1`

Suppose we did add a semicolon:

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

We get this error:

```
error[E0308]: mismatched types
 --> src/main.rs:7:24
  |
7 | fn plus_one(x: i32) -> i32 {
  |    --------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
8 |     x + 1;
  |          - help: remove this semicolon to return this value
```

`if` Expressions

We can define runtime control flow with `if`:

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

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

`if` Expressions

`if` expressions must take a boolean.

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

    if number {
        println!("number was three");
    }
}
```


```
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |     if number {
  |        ^^^^^^ expected `bool`, found integer
```

`else if` Branching

You can handle multiple conditions with `else if`

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

`if`s are Expressions!

Since `if` expressions are expressions, we can bind the result of an if expression to a variable.

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

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

- `if` expressions must always return the same type in all branches

Loops

There are 3 kinds of loops in Rust:

- `loop`
- `while`
- `for`

`loop` loops

`loop` will keep looping until you tell it to stop with `break`.

```
fn main() {
    let mut counter = 0;

    loop {
        counter += 1;

        if counter == 10 {
            break;
        }
    }

    println!("The counter is {counter}");
}
```

- `break` and `continue` apply to the innnermost loop where they are called

`loop`s are Expressions

Like everything else, you can return a value from a `loop`.

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");
}
```

Do we talk about loop lables

`while` loops

Just like other languages, we have `while` loops that stop after some condition.

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

    while number != 0 {
        println!("{number}!");

        number -= 1;
    }

    println!("LIFTOFF!!!");
}
```

`for` loops

We can also loop through collections with a `for` loop.

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

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

`for` loops and ranges

To loop over a range, use the `..` syntax to create a range.

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

Recap

- Variables
- Scalar and Compound Data Types
- Functions
- Control Flow

Course logistics

TODO

Installing Rust

TODO