Rust Crash Course

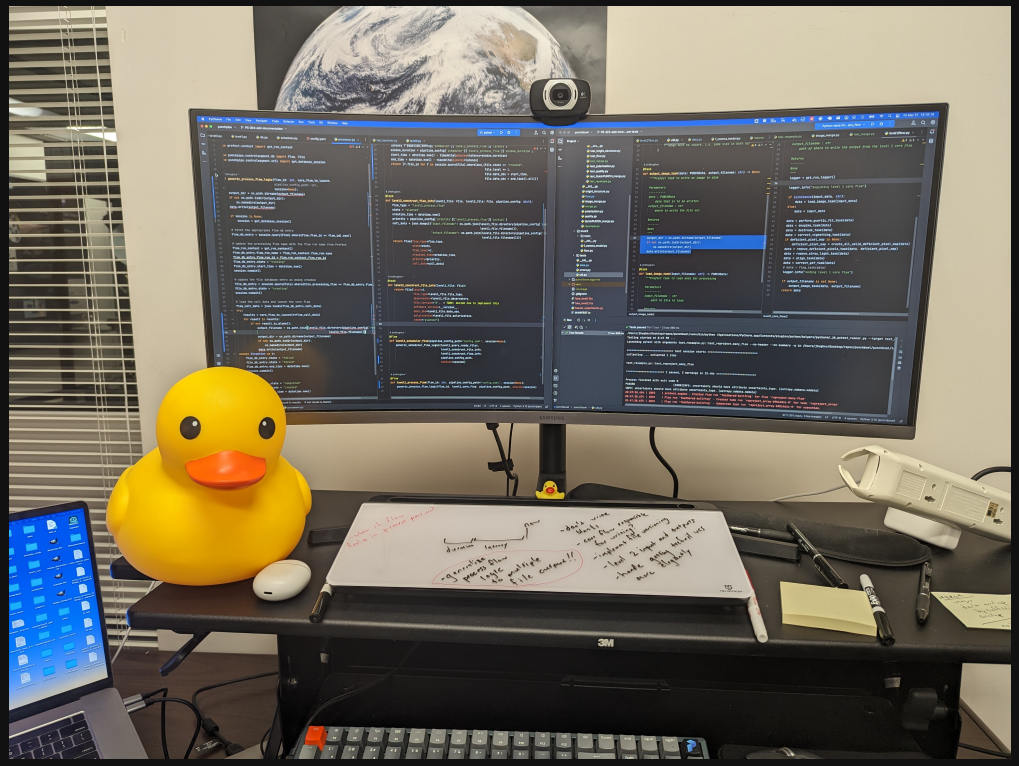
Marcus Hughes

SwRI SST 27 April 2023

PDF available at

https://publish.obsidian.md/arbor/attachments/crash_course_in_rust_slides.pdf

Ducks help with debugging!



Say hi to your new duck friend!

My Rust journey has just begun

- Re-implementing the core of TomograPy in Rust using Rayon for parallel execution
- Made a simple tower defense game in Rust's Bevy game engine

I'm still a Rust noob.

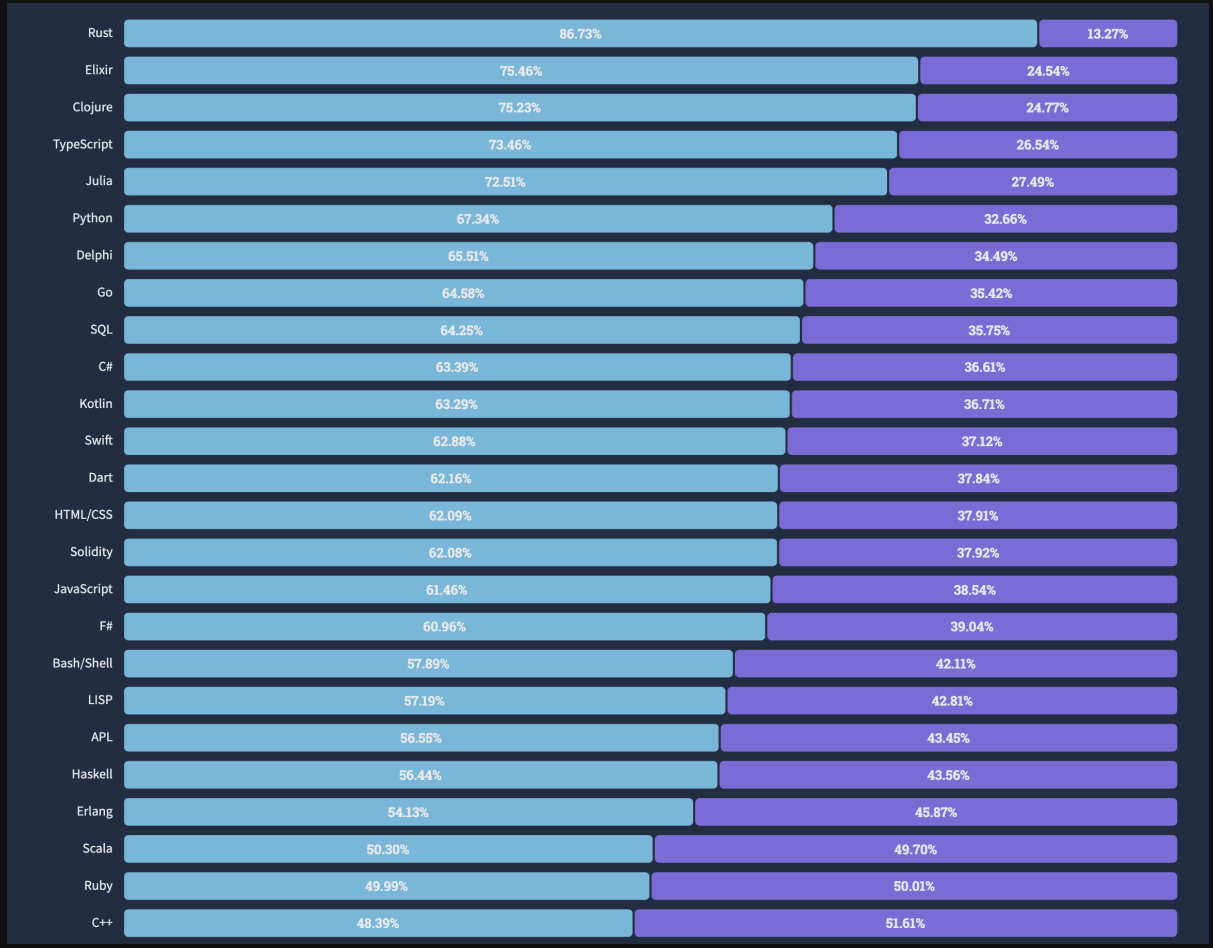
Why learn Rust?

- Rust is a memory safe language.
- Rust is fast.
- Rust is concurrent.
- Rust is expressive.
- Rust is a modern language.
- Rust can accelerate Python using PyO3 and Maturin.

Why does Rust have the reputation of being a difficult language to learn?

- Ownership and borrowing
- Lifetimes
- Generics

Why is Rust among the most loved langauges?



Stack Overflow 2022 Survey

Why is Rust among the most loved langauges?

- Memory safety!
- Instructive compiler
- Beautiful error handling
- An elegant and powerful type system

Free learning resources

- The Rust Book
- Rust By Example
- Rustlings exercises
- No Boilerplate YouTube
- Code to the Moon YouTube

Buckle up!

We're going to move quickly.

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

```rust
fn main() {
    let age = 27;
    println!("Hello I am {}", age);
}
```

```rust
fn main() {
    let x: i32 = -1;
    if x == 42 {
        println!("You have found the answer.");
    } else {
        println!("Oh no! Keep thinking.");
    }
}
```

What if we forgot to initialize?

Rust helps!

```
error[E0381]: used binding `x` isn't initialized
 --> exercises/variables/variables2.rs:6:8
  |
5 |     let x: i32;
  |         - binding declared here but left uninitialized
6 |     if x == 10 {
  |        ^ `x` used here but it isn't initialized
  |
help: consider assigning a value
  |
5 |     let x: i32 = 0;
  |                +++

error: aborting due to previous error
```

```rust
fn main() {
    let mut x = 3;
    println!("Number {}", x);

    x = 5;
    println!("Number {}", x);
}
```

Challenge!

Only add symbols

```
fn main() {
    let number = "T-H-R-E-E"; // don't change this line
    println!("Spell a Number : {}", number);

    number = 3; // don't rename this variable
    println!("Number plus two is : {}", number + 2);

}
```

Solution: scoping

```
fn main() {
    let number = "T-H-R-E-E"; // don't change this line
    println!("Spell a Number : {}", number);
    {
        let number = 3; // don't rename this variable
        println!("Number plus two is : {}", number + 2);
    }
}
```

Functions

```
fn add_two_and_print() {
    let number = 3;
    println!("Number plus two is : {}", number + 2);
}

fn main() {
    let number = "T-H-R-E-E";
    println!("Spell a Number : {}", number);
    add_two_and_print();
}
```

Loopy function with a parameter

```
fn main() {
    call_me(3);
}


fn call_me(num: u32) {
    for i in 0..num {
        println!("Ring! Call number {}", i + 1);
    }
}
```

Returning from a function

```
fn main() {
    let number = 51;
    println!("Even? {}", is_even(number));
}

fn is_even(num: i32) -> bool {
    num % 2 == 0
}
```

Challenge: write `favorite_number`

```
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn marcus_favorite() {
        assert_eq!(favorite_number("marcus"), 42)
    }
    #[test]
    fn enrico_favorite() {
        assert_eq!(favorite_number("enrico"), 1)
    }
    #[test]
    fn other_favorite() {
        assert_eq!(favorite_number("anyone else"), 7)
    }
}
```

Solution

```
pub fn favorite_number(name: &str) -> i32 {
    if name == "marcus" {
        42
    } else if name == "enrico" {
        1
    } else {
        7
    }
}
```

Another way!

```
pub fn favorite_number(name: &str) -> i32 {
    match name {
        "marcus" => 42,
        "enrico" => 1,
        _ => 7
    }
}
```

Welcome to structs!

```
struct Point {
    x: i32,
    y: i32,
}

fn print_point(point: Point) {
    println!("({}, {})", point.x, point.y);
}

fn main() {
    let p = Point {x: 3, y: 6};
    print_point(p);
}
```

Structs can have associated functions

```
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn add(self, other: Point) -> Point {
        Point {x: self.x + other.x, y: self.y + other.y}
    }
}

fn main() {
    let p1 = Point {x: 3, y: 6};
    let p2 = Point {x: -3, y: -6};
    let origin = p1.add(p2);
}
```

Enumerated types allow you to define a variable type that has set possible values.

```
enum Direction {
    North,
    South,
    East,
    West,
}

fn main() {
    let direction = Direction::North;

    match direction {
        Direction::North => println!("Going north"),
        Direction::South => println!("Going south"),
        Direction::East => println!("Going east"),
        Direction::West => println!("Going west"),
    }
```

If I forget a case, Rust will complain.

Rust's enums are more complete than other languages.

They can contain more information and have methods.

```rust
#[derive(Debug)]
enum Message {
    Move{x: i32, y: i32},
    Echo(String),
    ChangeColor(i32, i32, i32),
    Quit
}

impl Message {
    fn call(&self) {
        println!("{:?}", self);
    }
}
```

```rust
fn main() {
    let messages = [
        Message::Move { x: 10, y: 30 },
        Message::Echo(String::from("hello world")),
        Message::ChangeColor(200, 255, 255),
        Message::Quit,
    ];

    for message in &messages {
        message.call();
    }
}
```

This code fails!

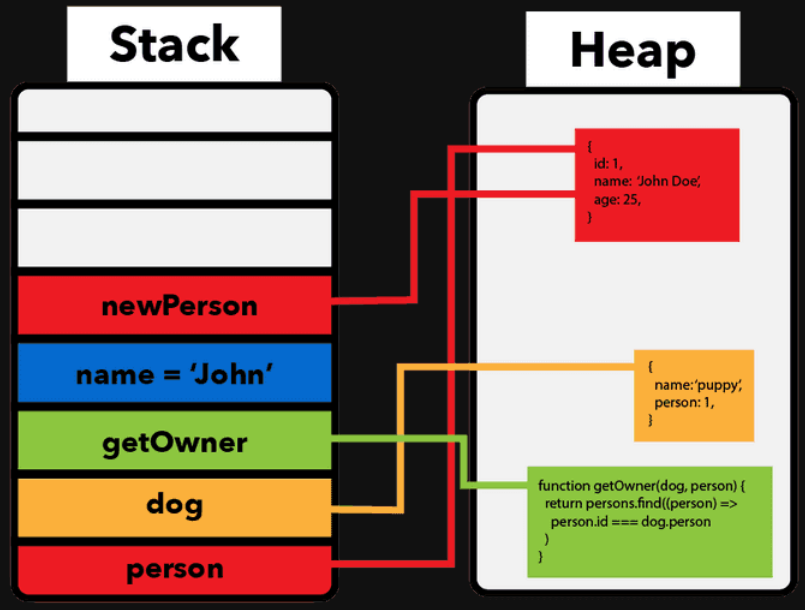
```
struct Point {
    x: i32,
    y: i32,
}

fn print_point(point: Point) {
    println!("({}, {})", point.x, point.y);
}

fn main() {
    let p = Point {x: 3, y: 6};
    print_point(p);
    print_point(p); // Only added this line
}
```

```
error[E0382]: use of moved value: `p`
  --> src/main.rs:13:14
   |
11 |     let p = Point {x: 3, y: 6};
   |         - move occurs because `p` has type `Point`, which does not implement the `Copy` trait
12 |     print_point(p);
   |                 - value moved here
13 |     print_point(p); // I added this line only
   |                 ^ value used here after move
   |
note: consider changing this parameter type in function `print_point` to borrow instead if owning the value isn't necessary
  --> src/main.rs:6:23
   |
6  | fn print_point(point: Point) {
   |    -----------       ^^^^^ this parameter takes ownership of the value
   |    |
   |    in this function

For more information about this error, try `rustc --explain E0382`.
error: could not compile `rust_examples` due to previous error
```

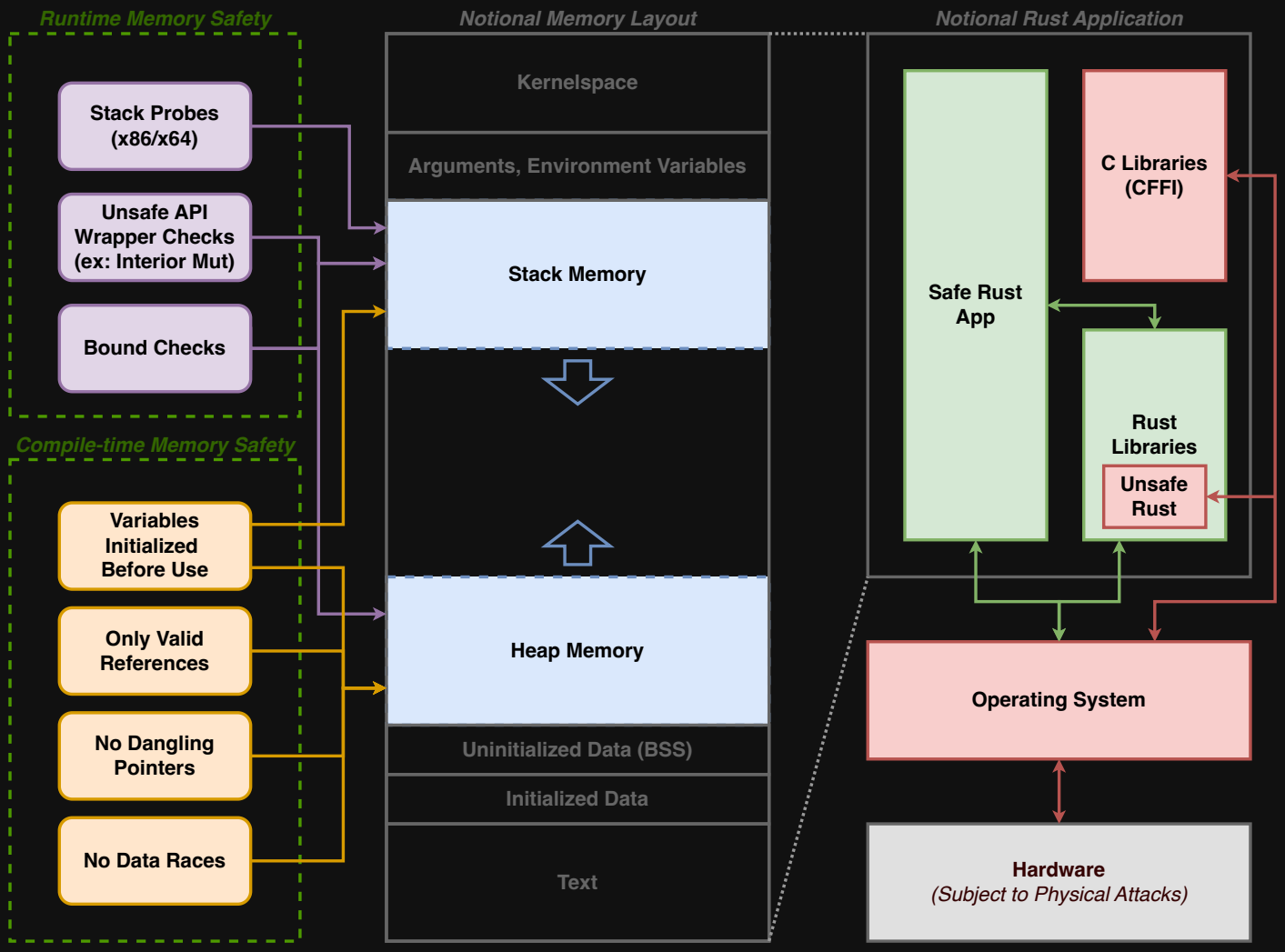
Ownership 101

- "Ownership is a set of rules that govern how a Rust program manages memory."
- It'll take some getting used to so don't panic. You have your duck friend (who is an expert)!

```
const person = {
  id: 1,
  name: 'John',
  age: 25,
}

const dog = {
  name: 'puppy',
  personId: 1,
}

function getOwner(dog, persons) {
  return persons.find((person) =>
     person.id === dog.person
  )
}

const name = 'John';

const newPerson = person;
```

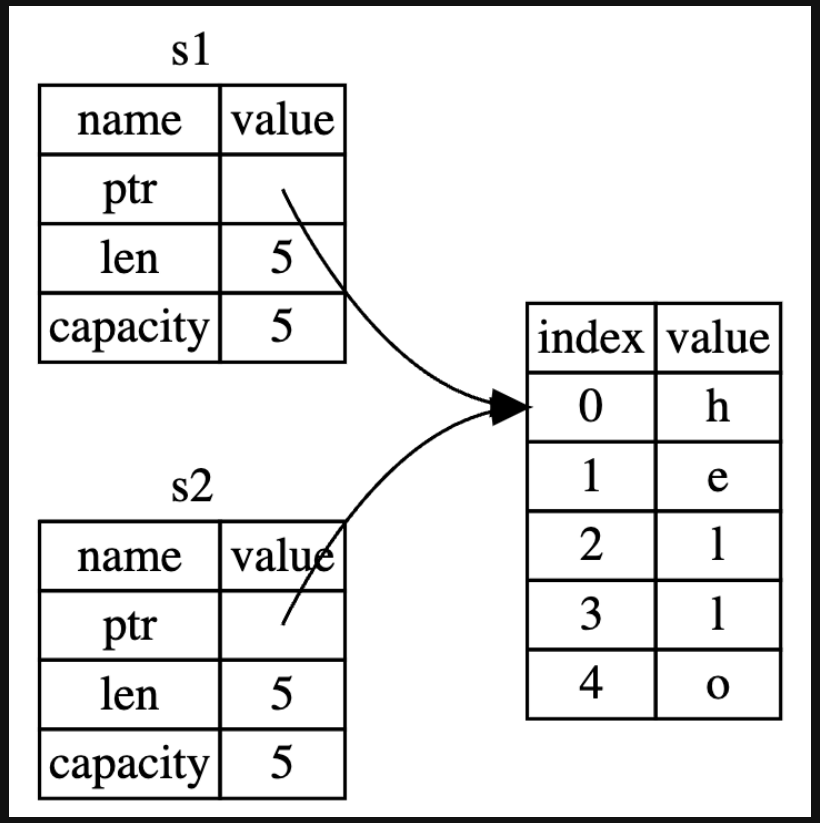


https://felixgerschau.com/javascript-memory-management/



https://highassurance.rs/chp4/safe_rust_PLACEHOLDER.html

```
let s1 = String::from("hello");
let s2 = s1;
```



Left is stack, Right is Heap

Here are some examples of when a Rust variable gets borrowed:

- When a variable is passed to a function as a reference.
- When a variable is used as a key in a hash map.
- When a variable is used as an element in a vector.
- When a variable is used as a field in a struct.

Generally,

when a variable is used to access its data,

it is being borrowed.

References are one solution

`&` denotes a reference in Rust

For example:

```
let x = 1;
let y = &x; // y is an immutable reference to x
```

References can be mutable

`&mut` is how you indicate that.

```
let x = 1;
let y = &mut x; // y is a mutable reference to x
```

Solution to our problem

```
struct Point {
    x: i32,
    y: i32,
}

fn print_point(point: &Point) {
    println!("({}, {})", point.x, point.y);
}

fn main() {
    let p = Point {x: 3, y: 6};
    print_point(&p);
    print_point(&p);
}
```

Be careful with mutability

```
fn main() {
    let x = 1;
    let y = &mut x;  // y is a mutable reference to x
    let z = &mut x;  // error! y and z both own x
}
```

You can only have one **mutable** reference for a variable at a time.

You can have as many **immutable** references for a variable as you want!

Let's talk generics!

```
fn main() {
    let mut prices: Vec<f32> = Vec::new();
    prices.push(32.99);
}
```

```rust
// This function runs for any partially ordered type!
fn max<T: std::cmp::PartialOrd>(a: T, b: T) -> T {
    if a > b {
        a
    } else {
        b
    }
}

fn main() {
    let x = 5;
    let y = 10;
    println!("The max is {}", max(x, y));
}
```

What is `std::cmp::PartialOrd`?

It's a "trait" or a contract for how a type behaves.

```
trait Printable {
    fn print(&self);
}

impl Printable for String {
    fn print(&self) {
        println!("{}", self);
    }
}

fn main() {
    let s = String::from("Hello, world!");
    s.print();
}
```

This code fails! Why?

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is '{}'", result);
}
```

It needs to know how long the reference persists.

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:33
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
1 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++

For more information about this error, try `rustc --explain E0106`.
```

Solution

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is '{}'", result);
}
```

This code fails! How do you fix it?

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}

fn main() {
    let string1 = String::from("xyz");
    let result;
    {
        let string2 = String::from("long string is long");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is '{}'", result);
}
```

Look at the error

```
error[E0597]: `string2` does not live long enough
  --> src/main.rs:14:44
   |
14 |         result = longest(string1.as_str(), string2.as_str());
   |                                            ^^^^^^^^^^^^^^^^ borrowed value does not live long enough
15 |     }
   |     - `string2` dropped here while still borrowed
16 |     println!("The longest string is '{}'", result);
   |                                            ------ borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `rust_examples` due to previous error
```

Solution

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}

fn main() {
    let string1 = String::from("xyz");
    let result;
    let string2 = String::from("long string is long");
    result = longest(string1.as_str(), string2.as_str());
    println!("The longest string is '{}'", result);
}
```

Iterators are a key Rust concept.

```rust
fn main() {
    let my_numbers = vec![1, 2, 3, 4, 6];
    let is_even: Vec<bool> = my_numbers.iter()
                                       .map(|x| x % 2 == 0)
                                       .collect();
}
```

```rust
fn main() {
    let words = vec!["we", "the", "people"];

    let reformatted: Vec<String> = words.iter()
                                        .map(|&s| capitalize_first(s))
                                        .collect();

    for word in reformatted {
        println!("{}", word);
    }
}
```

```rust
pub fn capitalize_first(input: &str) -> String {
    let mut c = input.chars();
    match c.next() {
        None => String::new(),
        Some(first) =>
            first.to_string().to_uppercase() + c.as_str(),
    }
}
```

Strings are a little weird in Rust

There are two types of strings in Rust:

- **&str** is a reference to a string slice. A string slice is a view into a string, and it does not own the underlying data.
- **String** is a heap-allocated string. A String owns the underlying data, and it can grow and shrink as needed.

Congratulations!

You've taken your first steps in mastering Rust.