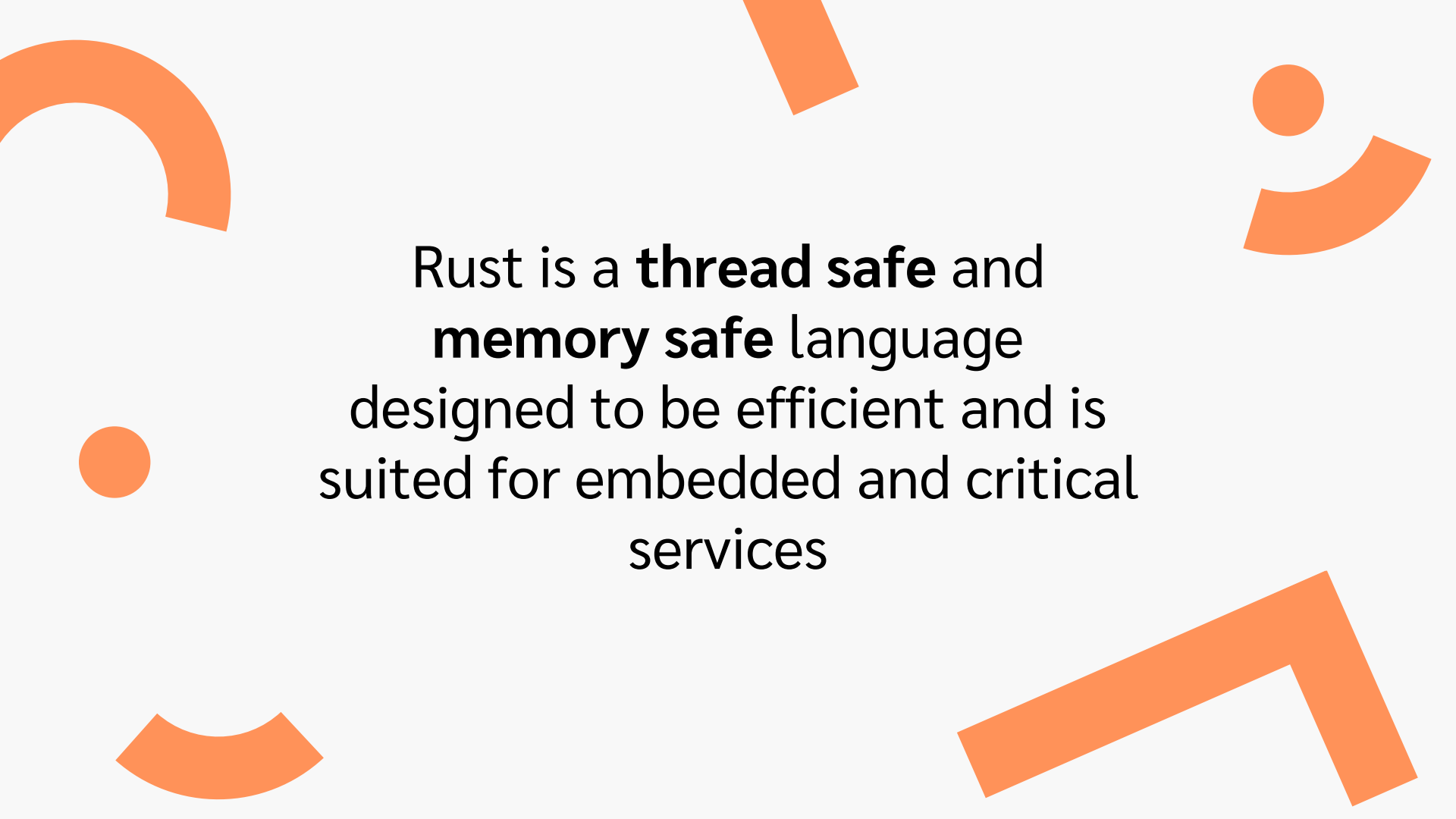


BENJAMIN MARASCO

Rust Language Review



The background features several abstract orange geometric shapes: a large arc in the top-left, a small circle in the middle-left, a small arc in the bottom-left, a small circle and a larger arc in the top-right, and a large L-shaped block in the bottom-right.

Rust is a **thread safe** and
memory safe language
designed to be efficient and is
suited for embedded and critical
services

BENJAMIN MARASCO

01

Rust Features

And supported programming paradigms

Names, Binding, and Scope

Names

Variables are named using the **let** keyword, for example:

```
let x = 5;
```

Binding

Variables by default are **immutable**, so you can't reassign them after declaration. To make it mutable you use the **mut** keyword:

```
let mut x = 5;
```

Scope

Scope in rust is complex. Every value has an **owner**, and there can only be **one owner** at a time. If the owner leaves scope, it is dropped:

```
let s1 = String::from("hello");  
let s2 = s1; //s1 ceases to exist
```

Datatypes in Rust

Rust provides a set of scalar and vector datatypes, as well as user-defined types through **structs** and **enums**

Integers	Signed(i) and Unsigned(u) integers such as; i8 , u8 , i16 , u16 , ...
Floating point	Ether f32 or f64 with the default being f64 when type is interpreted
Boolean	Your typical true or false
Character	Not your typical char , it stores Unicode Scalar Values which are more than just ASCII
Tuple	A collection of multiple types, with a fixed size
Array	A collection of the same type, with a fixed length
Vector	A collection that can grow or shrink, which accepts generic types

The background features several large, bright green geometric shapes: a curved line in the top left, a thick diagonal line crossing the middle, a circle in the top right, and a thick line forming a large 'V' shape at the bottom right.

Rust provides object oriented paradigms

Rust does not explicitly refer to **structs** or **enums** as objects but provides ways to define methods which act on encapsulated data within those types.

It does **not** provide **inheritance**.

By default, **everything is private**, however you can use the **pub** keyword to change that.

```
struct Stack<T> {  
    data: Vec<T>,  
}  
  
impl<T> Stack<T> {  
    pub fn new() -> Stack<T> {  
        Stack { data: Vec::new() }  
    }  
  
    pub fn push(&mut self, item: T) {  
        self.data.push(item);  
    }  
  
    pub fn pop(&mut self) -> Option<T> {  
        self.data.pop()  
    }  
  
    pub fn peek(&self) -> Result<&T, &str>  
    {  
        if(self.data.is_empty()) {  
            Err("Stack is empty")  
        } else {  
            Ok(self.data.last())  
        }  
    }  
  
    pub fn empty(&self) -> bool {  
        self.data.is_empty()  
    }  
}
```

Simple Stack

Here's an example of the simple stack from the textbook implemented in rust using a **struct** and **impl** block

Rust provides concurrency

Rust provides **threads** for running concurrent tasks, threads can be waited on by calling **join**.

Variables can be **moved** into threads when they are spawned, removing them from the calling scope. When moved into a thread, **only that thread** can access the variable.

Rust also provides **channels**, where multiple producing threads can send to a single consuming thread.

Mutexes are also available to share data across threads as well.


```

use std::sync::{Arc, Mutex};
use std::thread;

const BUFFER_SIZE: usize = 5;
const NUM_ITEMS: usize = 10;

fn producer(producer_id: usize, buffer: Arc<Mutex<Vec<i32>>>) {
    for i in 0..NUM_ITEMS {
        let item = i as i32;

        // Acquire the lock to access the buffer
        let mut buffer = buffer.lock().unwrap();

        // Wait until there is space available in the buffer
        while buffer.len() >= BUFFER_SIZE {
            buffer = buffer.wait().unwrap();
        }

        // Produce an item and add it to the buffer
        buffer.push(item);
        println!("Producer {} produced item: {}", producer_id, item);

        // Notify the consumer that an item is available
        buffer.notify_all();
    }
}

fn consumer(consumer_id: usize, buffer: Arc<Mutex<Vec<i32>>>) {
    for _ in 0..NUM_ITEMS {
        // Acquire the lock to access the buffer
        let mut buffer = buffer.lock().unwrap();

        // Wait until there is an item available in the buffer
        while buffer.is_empty() {
            buffer = buffer.wait().unwrap();
        }

        // Consume an item from the buffer
        let item = buffer.pop().unwrap();
        println!("Consumer {} consumed item: {}", consumer_id, item);

        // Notify the producer that space is available in the buffer
        buffer.notify_all();
    }
}

```

```

fn main() {
    // Create a shared queue buffer with a mutex
    let buffer = Arc::new(Mutex::new(Vec::new()));

    // Create producer threads
    let producers: Vec<_> = (0..3).map(|id| {
        let buffer = buffer.clone();
        thread::spawn(move || producer(id, buffer))
    }).collect();

    // Create consumer threads
    let consumers: Vec<_> = (0..2).map(|id| {
        let buffer = buffer.clone();
        thread::spawn(move || consumer(id, buffer))
    }).collect();

    // Wait for all threads to finish
    for producer in producers {
        producer.join().unwrap();
    }
    for consumer in consumers {
        consumer.join().unwrap();
    }
}

```

Concurrent Example


Using a **mutex** to create a shared queue, we have multiple producers and consumers writing to the queue

Rust does exceptions differently

The rust language groups exceptions into two types: **recoverable** and **unrecoverable**.

Unrecoverable errors cause the program to terminate, such as accessing memory outside of an array bounds.

Recoverable errors are conveyed through the **Result<T, E>** datatype. Logic branches from the **Result** type to either handle an error or continue with success.



```
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

Handling Exceptions

Here's a sample of how rust uses **match** to identify if the **Result<T,E>** datatype returned an error **E** or type **T**

The background features several large, bright green geometric shapes: a curved line in the top left, a thick diagonal line crossing the bottom right, a curved line in the top right, a small circle in the middle right, and a larger circle in the bottom left.

Rust supports functional programming

Rust has a few functional programming paradigm features, mainly **closures** and **iterators**.

Closures are anonymous functions that can be passed as parameters or saved to variables.

Iterators can perform actions on sequences of items, such as applying an operation to all items, summing values, or simply iterating through values.

```
fn f(x: i32) -> i32 {  
    x + 2  
}  
  
fn g(x: i32) -> i32 {  
    x * 3  
}  
  
fn b(x: i32, f: fn(i32) -> i32, g: fn(i32) -> i32) -> i32 {  
    f(g(x))  
}  
  
fn main() {  
    let compositionResult = b(2, f, g);  
    println!("Composite: {}", compositionResult);  
  
    let applyToAllResult = [2, 3, 4].iter().map(|x| x*x).collect();  
    println!("ApplyToAll: {:?}", applyToAllResult)  
}
```

Functional Example

Closures are used to implement the function $b=f(g(x))$, and **iterators** for $b(x)=x*x \alpha(b(2,3,4))$

BENJAMIN MARASCO

02

Rust Project

Junction, a simple TCP load balancer
written in Rust

Objectives for the Load balancer

Distribute TCP connections using a round-robin method
written in Rust

- Must be concurrent
- Should accept a variable number of destinations
- Use functional programming elements

Constraints and Features

Constraints

- Can only distributed TCP traffic
- Can only round-robin distribute traffic
- Network traffic moves from kernel to user space

Features


- Can route HTTP traffic!
- Round-robins traffic to a variable number of destinations



How it was developed

This was developed using the **rust** language and utilized elements of **functional programming**, **object-orientation**, **concurrency**, and **exception handling**

To test the load balancer, multiple flask servers were run and connected to the load balancer. Multiple web clients then connected and were routed to the different servers.

The background features several abstract orange geometric shapes: a large arc in the top-left, a small circle in the middle-left, a small arc in the bottom-left, a small circle in the top-right, a small arc in the middle-right, and a large L-shaped block in the bottom-right.

Load balancers are **critical** to
distributed cloud infrastructure,
and a **memory-safe** load
balancer provides **reliable**
service



Thank you!

*Information on the rust language provided in this presentation is obtained from the official documentation here:
<https://doc.rust-lang.org/book/>*

This presentation format was provided by slidesgo