Introducing Rust

A language empowering everyone to build reliable and efficient software.

By Micah Williams, Chris He, and Ariana Mondiri

Rust's Origins

Graydon Hoare created Rust in 2006 as a personal project while employed at Mozilla

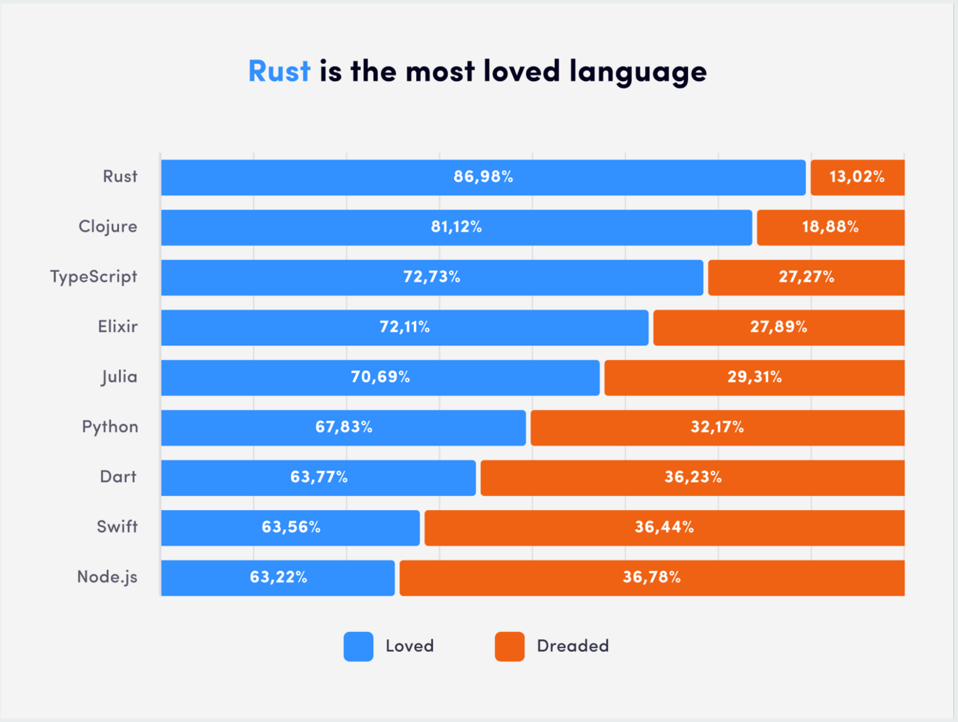
Was picked up and officially sponsored by Mozilla in 2009

First official announcement was by Mozilla in 2010 as part of a larger project

Rust 1.0, the first stable version, was released in 2015

Completely open-source with a passionate and rapidly growing community

Then And Now



“The Rust Foundation” created in 2021 support and steward Rust’s development w/ support by major companies such as Google, AWS, Mozilla, Huawei, and Microsoft

Development continues to be community-driven

Added to the Linux Kernel in December, 2022

Ranked on Stack Overflow Developer survey as the most loved language for 8 years in a row.(Source)

Was one of the fastest growing language on Github in 2022, seeing over 50% growth (source)

Real World Applications

Dropbox

Rust used in optimizing several parts of their cloud file storage.

“Rust has been a force multiplier for our team, and betting on Rust was one of the best decisions we made.

-Sujay Jayakar, Dropbox Sync Tech Lead

Facebook

Rewriting portions of the source code control system in Rust.

“Facebook has embraced Rust since 2016 and utilizes it in all aspects of development, from source control to compilers,”

-Joel Marcey, Open Source Ecosystem Lead at

Discord

Rewrote multiple systems from Go to Rust to increase performance

“Discord is using Rust in many places across its software stack. We use it for the game SDK, video capturing and encoding for Go Live, Elixir NIFs..”

-Jesse Howarth, Discord Software Engineer

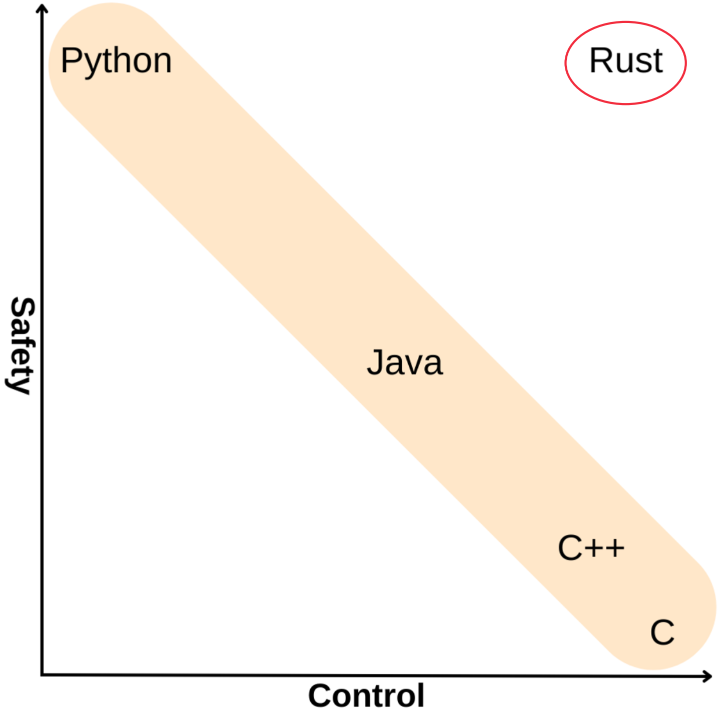
Primary Goal

Above all else, Rust was created to balance **low-level control** with **ensured memory safety**

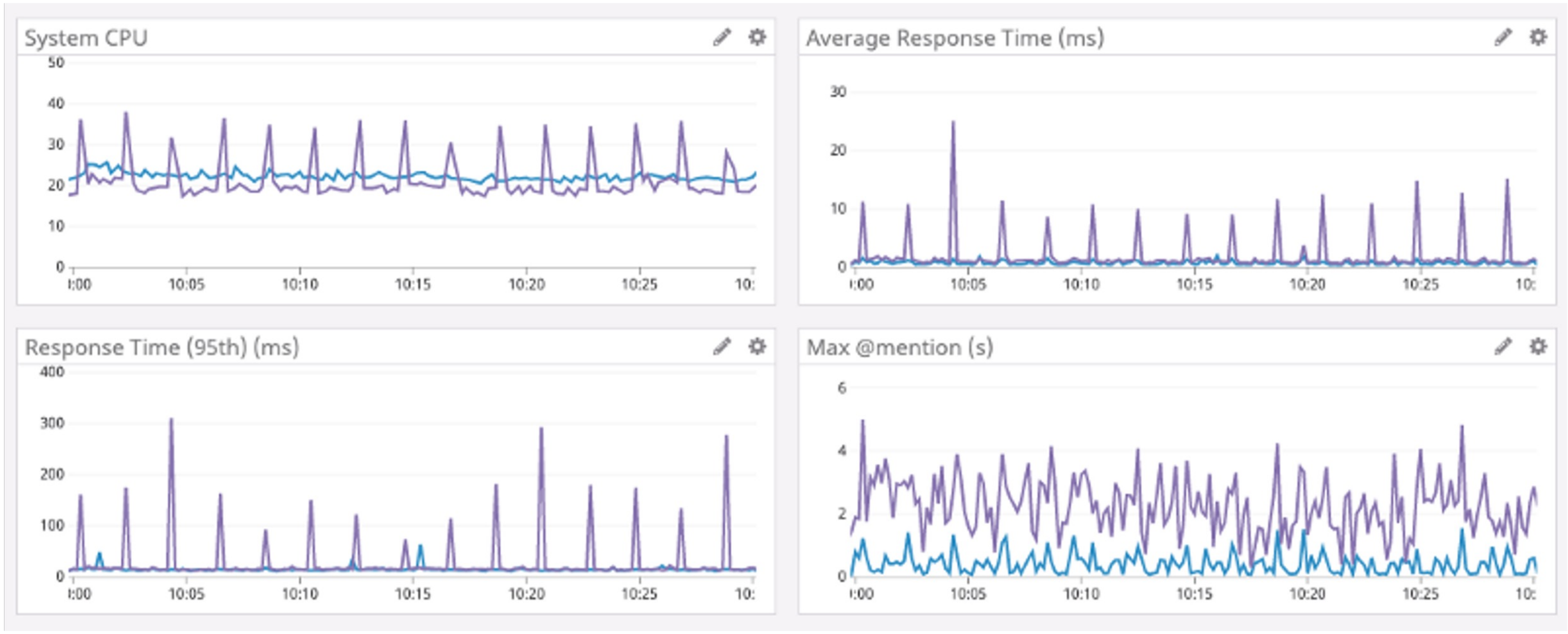
C/C++ grants a high level of control over each variable and it's lifetime with manual memory allocation.

Python is significantly safer because it automates memory management using a garbage collector.

Rust seeks to combine the best of both.



Discord Swapped from Go to Rust



Purple is Go, blue is Rust

Rust’s Way

The Challenge

Maintain the low-level control that manual memory allocation affords, while also having safety guarantees that are comparable to a garbage collector.

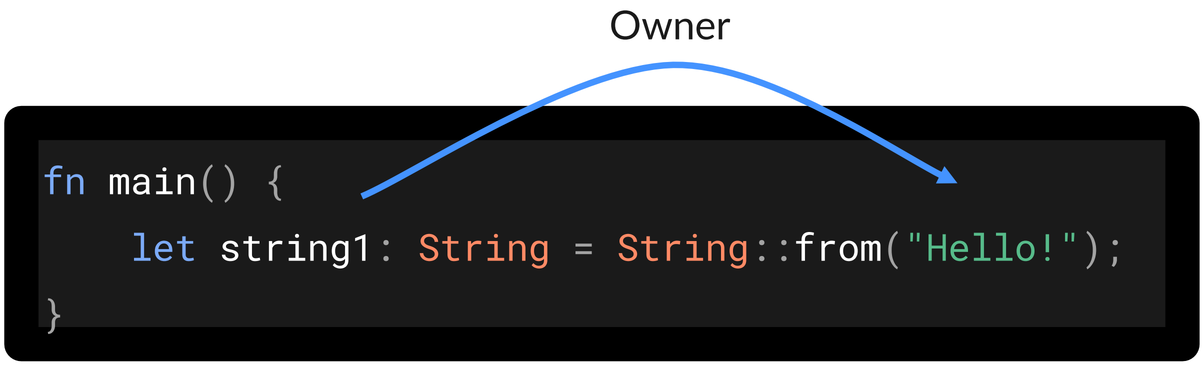
Rust’s Solution

Created a compiled, statically and strongly typed language with its own unique memory management system called Ownership that balances low-level control and safety

- Three Rules
- “If it compiles, it works”
- Concurrency

Ownership

1. Each value in Rust has an owner.



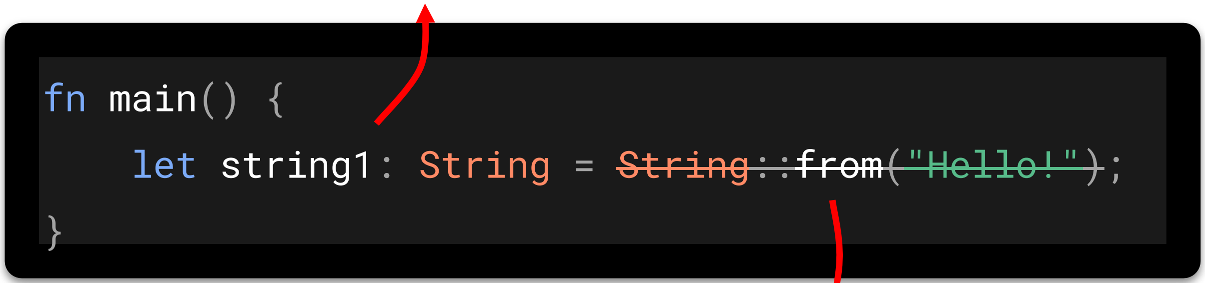
```
fn main() {
    let string1: String = String::from("Hello!");
}
```

Ownership

1. Each value in Rust has an owner.

2. When owner goes out of scope, the value is dropped.

Owner leaves scope



```rust
fn main() {
    let string1: String = String::from("Hello!");
}
```

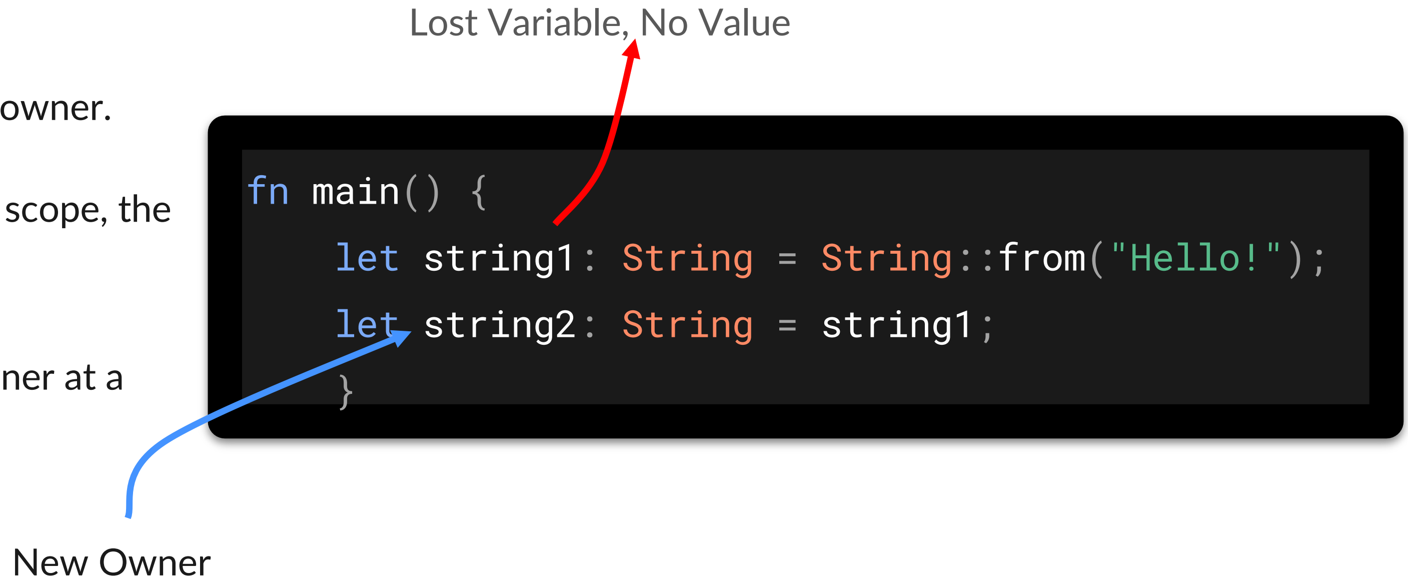
Memory Deallocated

Ownership

1. Each value in Rust has an owner.

2. When owner goes out of scope, the value is dropped.

3. There can only be one owner at a time.



Lost Variable, No Value

```
fn main() {
    let string1: String = String::from("Hello!");
    let string2: String = string1;
    }
```

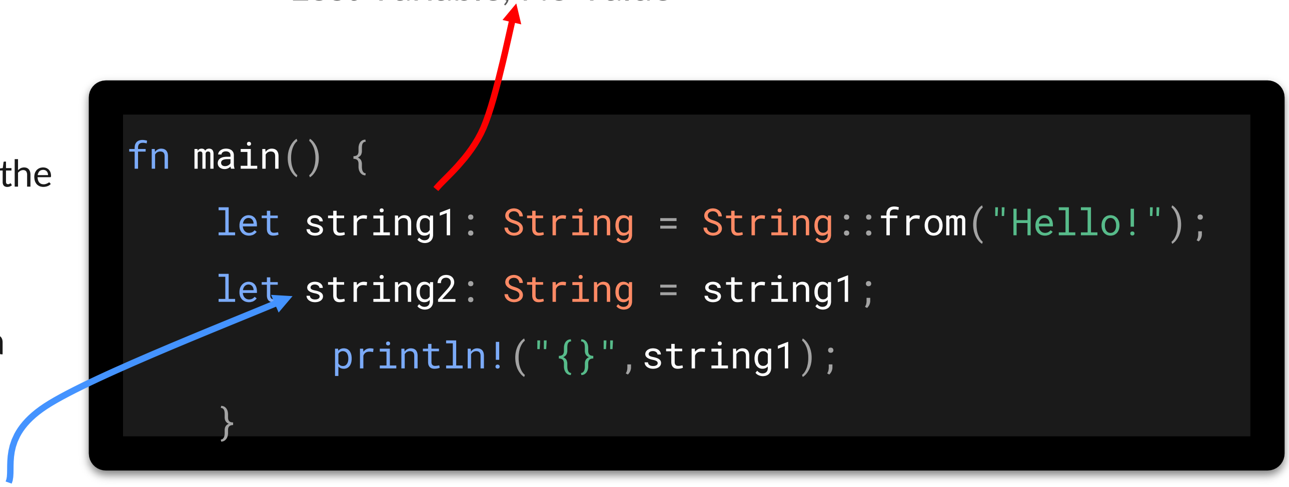
New Owner

Ownership

1. Each value in Rust has an owner.

2. When owner goes out of scope, the value is dropped.

3. There can only be one owner at a time.



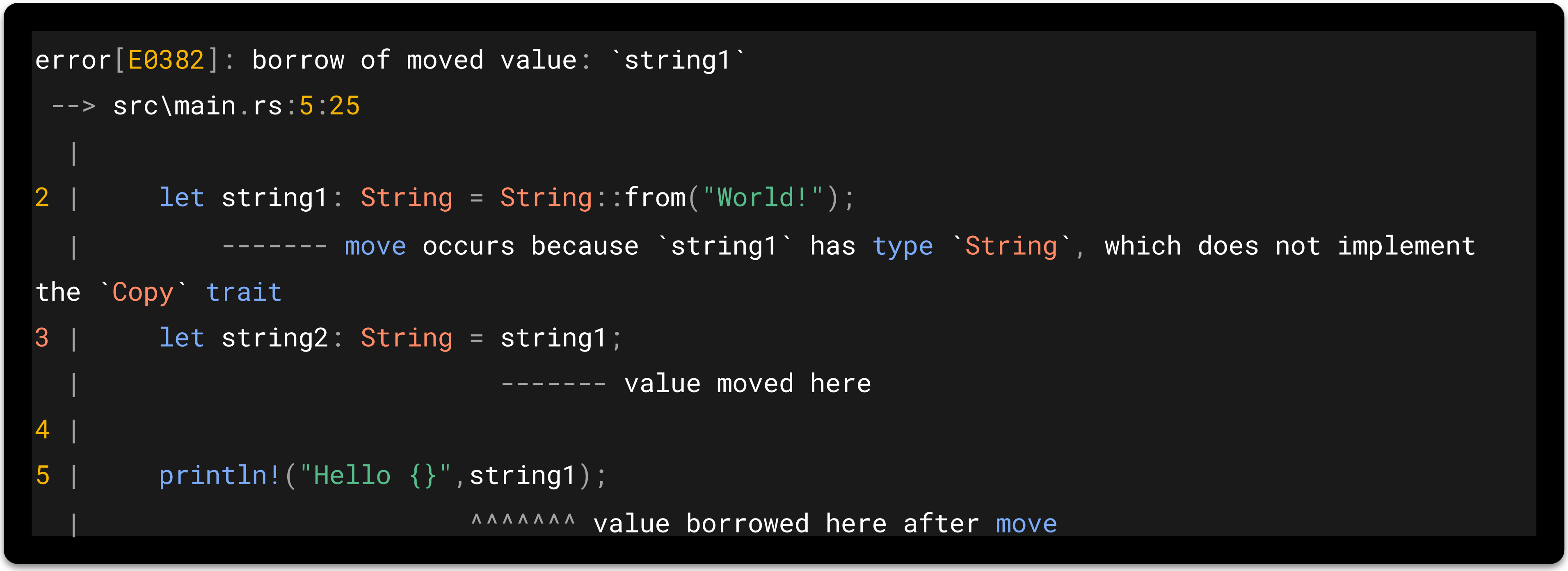
```
fn main() {
    let string1: String = String::from("Hello!");
    let string2: String = string1;
        println!("{}",string1);
    }
```

Lost Variable, No Value

New Owner

Fails to
compile

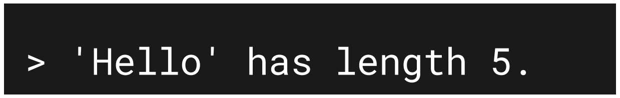
Ownership Error Message



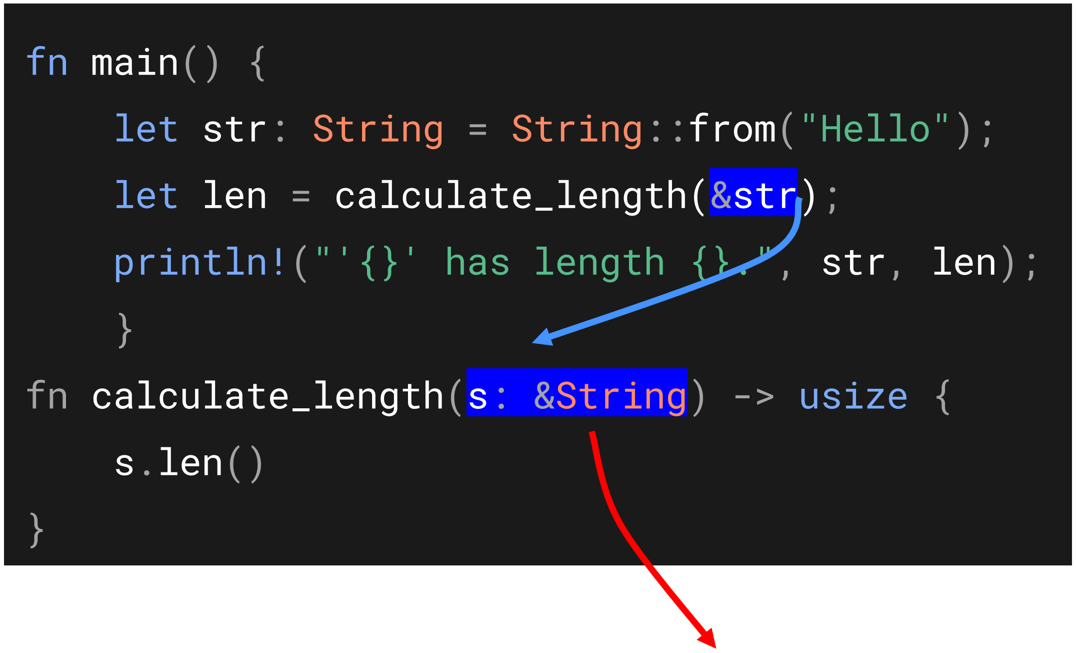
```
error[E0382]: borrow of moved value: `string1`
 --> src\main.rs:5:25
  |
2 |     let string1: String = String::from("World!");
  |         ------- move occurs because `string1` has type `String`, which does not implement
the `Copy` trait
3 |     let string2: String = string1;
  |                           ------- value moved here
4 |
5 |     println!("Hello {}",string1);
  |                         ^^^^^^^ value borrowed here after move
```

Ownership & References

- Can create references to data using & (ampersand) to pass into functions.
  - This is known as borrowing
- References by default are immutable
- At any given time there can be:
- an infinite number of immutable references OR one mutable reference being borrowed



```
> 'Hello' has length 5.
```



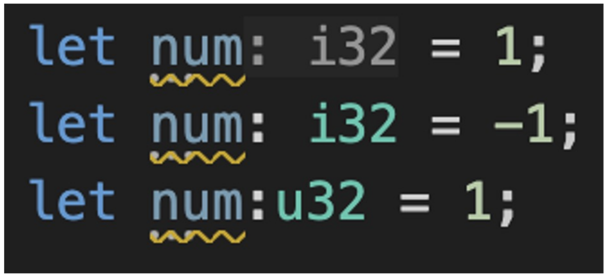
```
fn main() {
    let str: String = String::from("Hello");
    let len = calculate_length(&str);
    println!("'{}' has length {}.", str, len);
    }
fn calculate_length(s: &String) -> usize {
    s.len()
}
```

This function is borrowing str

SCALAR DATA TYPES

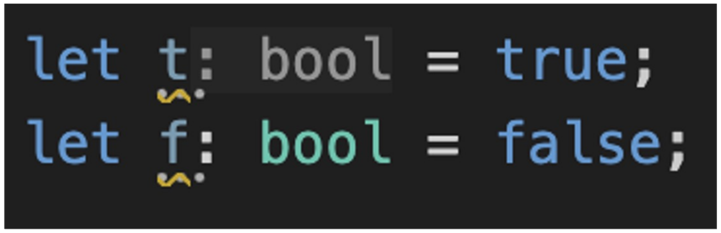
Rust is statically typed and favor safety and easy concurrency with immutable variables by default

Integer



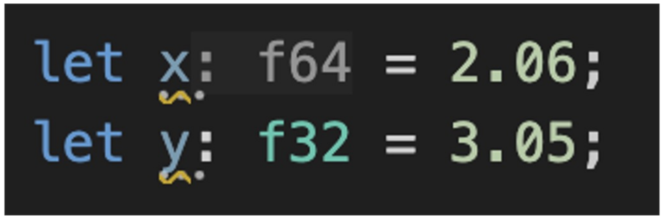
```
let num: i32 = 1;
let num: i32 = -1;
let num:u32 = 1;
```

Boolean



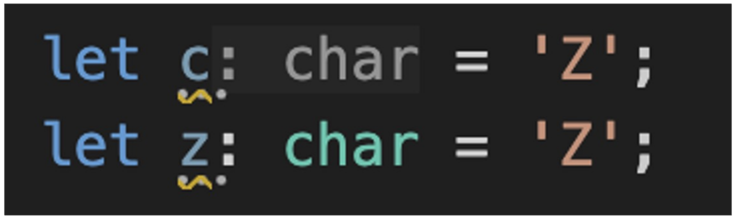
```
let t: bool = true;
let f: bool = false;
```

Floating-Point



```
let x: f64 = 2.06;
let y: f32 = 3.05;
```

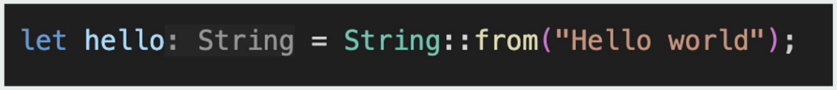
Character



```
let c: char = 'Z';
let z: char = 'Z';
```

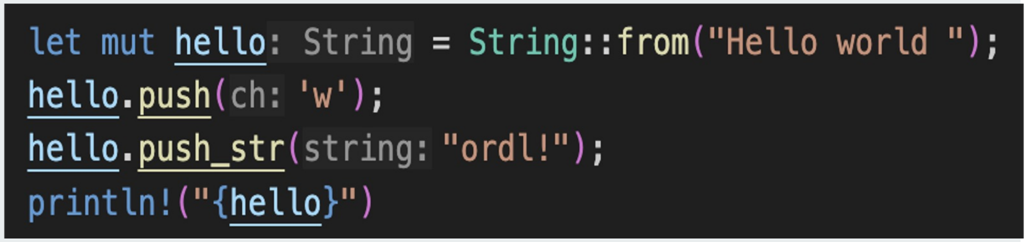
Strings

Dynamically allocated on the heap. Unknown size at compile time.



```
let hello: String = String::from("Hello world");
```

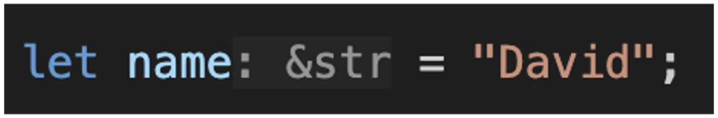
Strings are **mutable** and operations can be performed on them.



```
let mut hello: String = String::from("Hello world ");
hello.push(ch: 'w');
hello.push_str(string: "ordl!");
println!("{hello}")
```

>>> Hello world world!

String slices (&str)



```
let name: &str = "David";
```

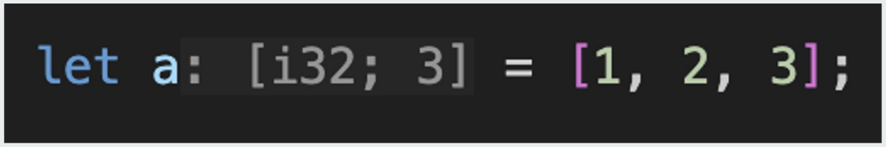
- Slice Type: Let you reference a contiguous sequence of elements in a sequence rather than the whole collection. Does not take ownership
- They have two components: pointers and length.
- String slices are reference to a sequence of UTF-8 bytes. Size is known at runtime

Since they are references string slices are **immutable.**

Arrays and tuples

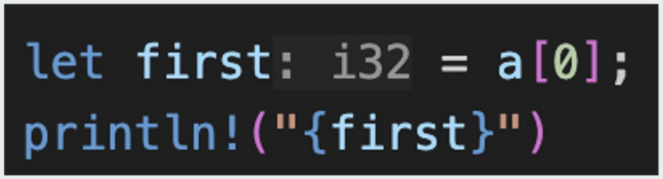
Compound types can group multiple values into one type. Rust has two majors primitive compound types: tuples and arrays

The array Type



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

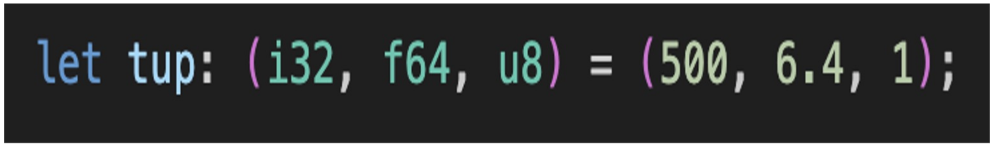
- Must have the same type
- Fixed length
- Stored on the stack



```rust
let first: i32 = a[0];
println!("{first}")
```

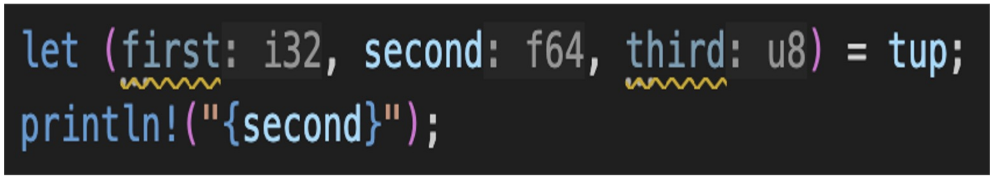
>>> 1

The tuple Type



```rust
let tup: (i32, f64, u8) = (500, 6.4, 1);
```

- Fixed length
- Multiple type allowed
- Stored on the stack

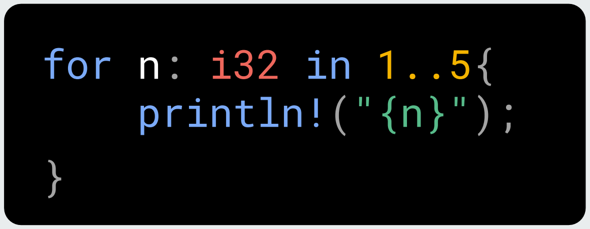


```rust
let (first: i32, second: f64, third: u8) = tup;
println!("{second}");
```

>>> 6.4

Control flow

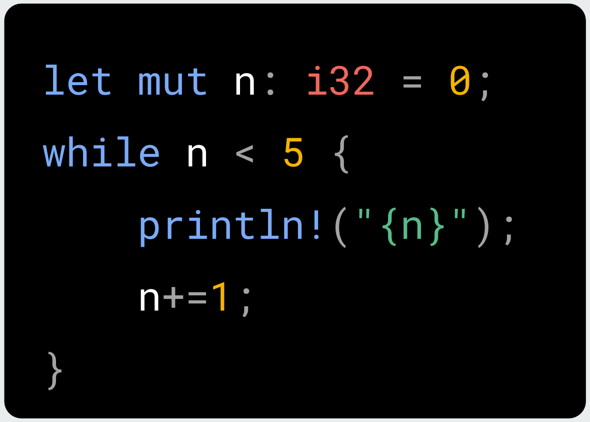
For loop



```rust
for n: i32 in 1..5{
    println!("{n}");
}
```

```
>>> 1 2 3 4
```

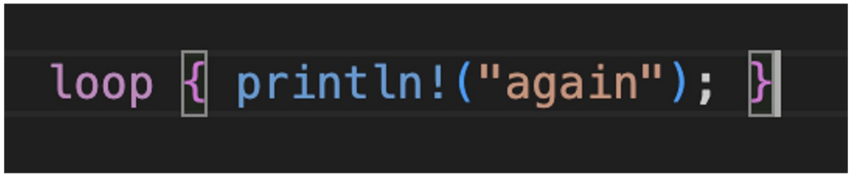
While loop



```rust
let mut n: i32 = 0;
while n < 5 {
    println!("{n}");
    n+=1;
}
```

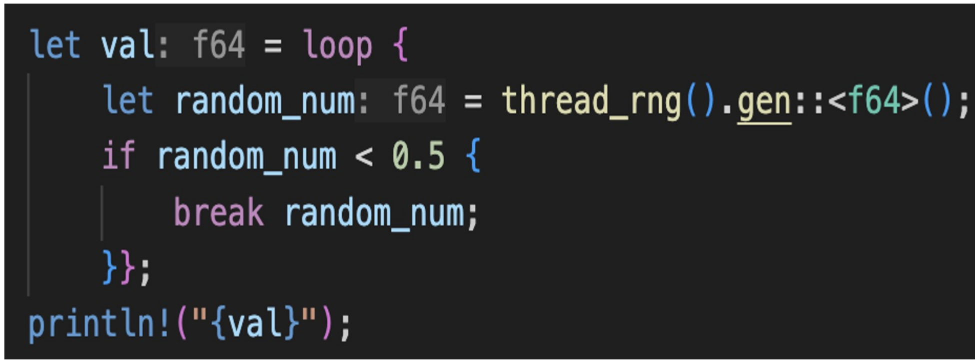
```
>>> 0 1 2 3 4
```

Infinite loop



```rust
loop { println!("again"); }
```

```
>> again!
>> again!
>> again!
....
```

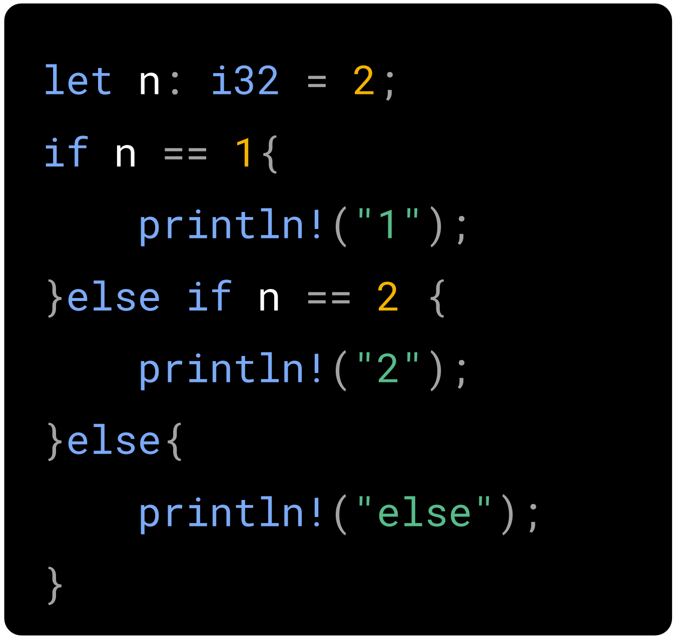


```rust
let val: f64 = loop {
    let random_num: f64 = thread_rng().gen::<f64>();
    if random_num < 0.5 {
        break random_num;
    }};
println!("{val}");
```

>>> some floating-point < 0.5

CONTROL STATEMENTS

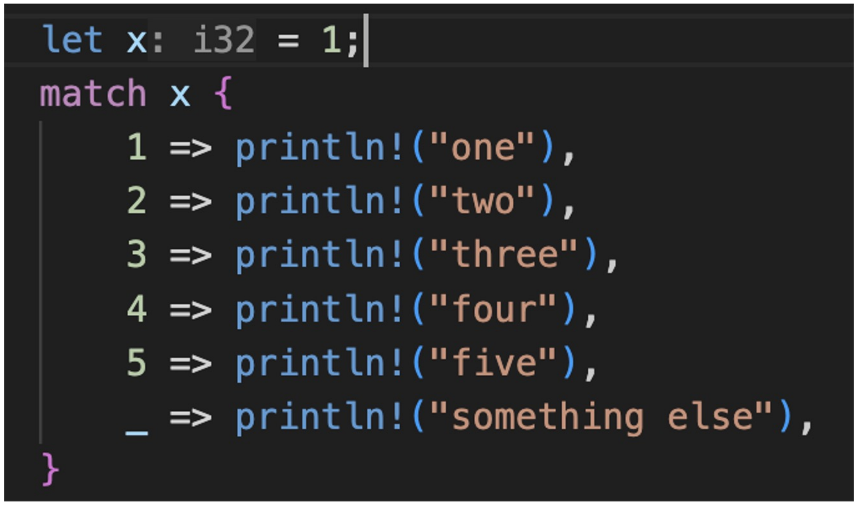
If statement



```
let n: i32 = 2;
if n == 1{
    println!("1");
}else if n == 2 {
    println!("2");
}else{
    println!("else");
}
```

>>> 2

Match



```
let x: i32 = 1;
match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}
```

>>> one

Abstraction

Structs, Traits and Generics

Zero-Cost Abstractions

“You don't pay for what you don't use”

Prioritized in Rust’s development

High level abstractions that have no runtime overhead

All abstractions evaluated in compile-time

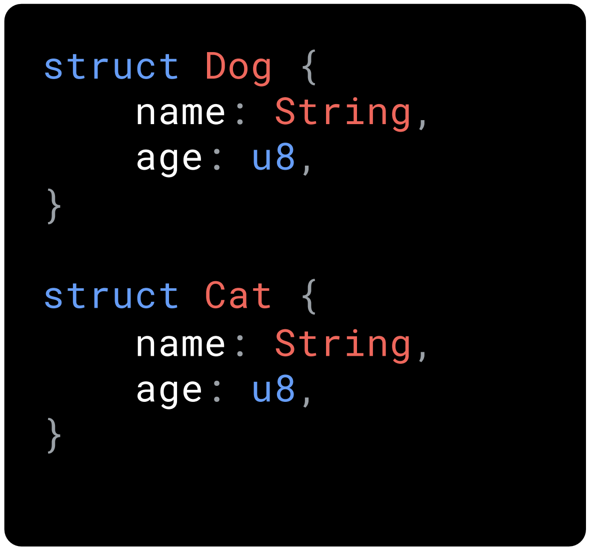
Structs

Implemented with **struct**

Structs define custom types that group related data and behavior

Data is contained within the struct in fields with specified types

Behavior is outlined outside of the struct in external methods or traits, which are implemented and defined by Structs externally using **impl**



```
struct Dog {
    name: String,
    age: u8,
}

struct Cat {
    name: String,
    age: u8,
}
```

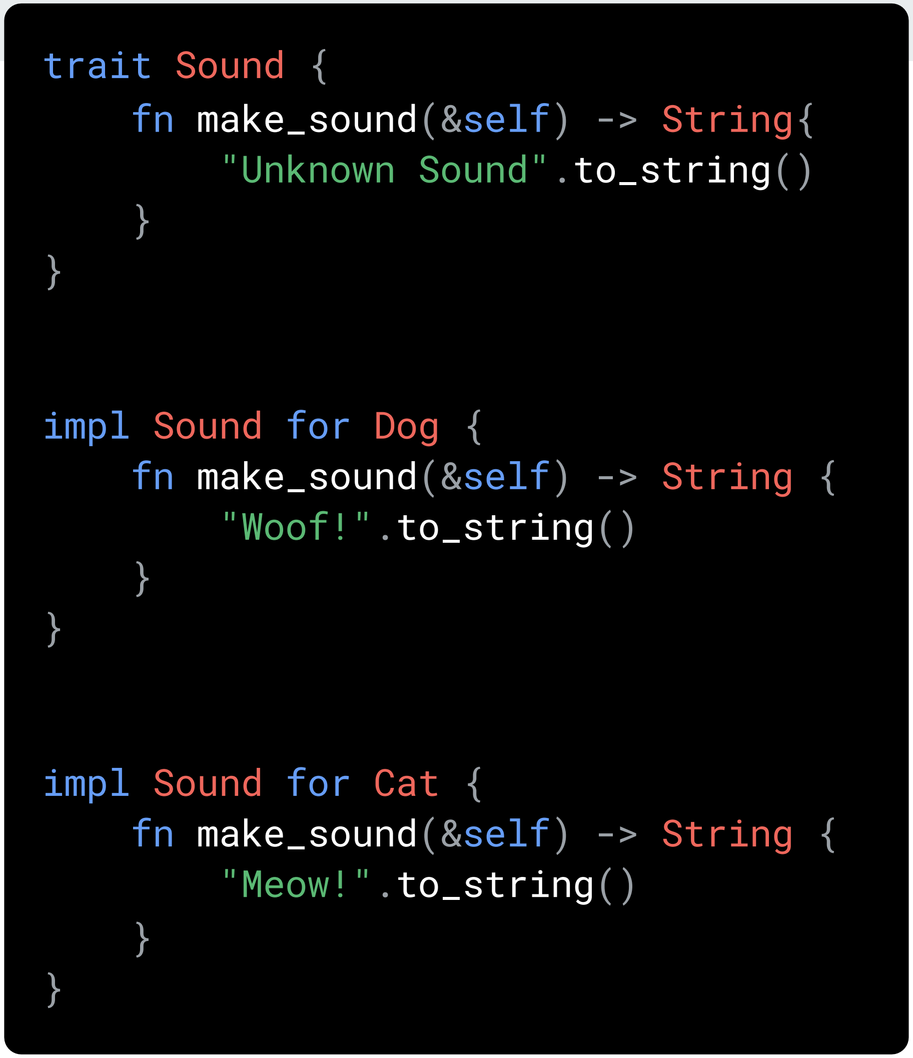
Traits

Implemented using trait

Collection of method signatures that define shared behavior across types

Similar to interfaces in Java, but with additional functionalities

Unlike interfaces, traits can have a default definition for a function



```rust
trait Sound {
    fn make_sound(&self) -> String{
        "Unknown Sound".to_string()
    }
}


impl Sound for Dog {
    fn make_sound(&self) -> String {
        "Woof!".to_string()
    }
}


impl Sound for Cat {
    fn make_sound(&self) -> String {
        "Meow!".to_string()
    }
}
```

Generics

Generic types are specified in angle brackets `<T>`

Used in the creation of generic functions, structs, and methods

Creates extremely robust and reusable code that is applicable to multiple different types

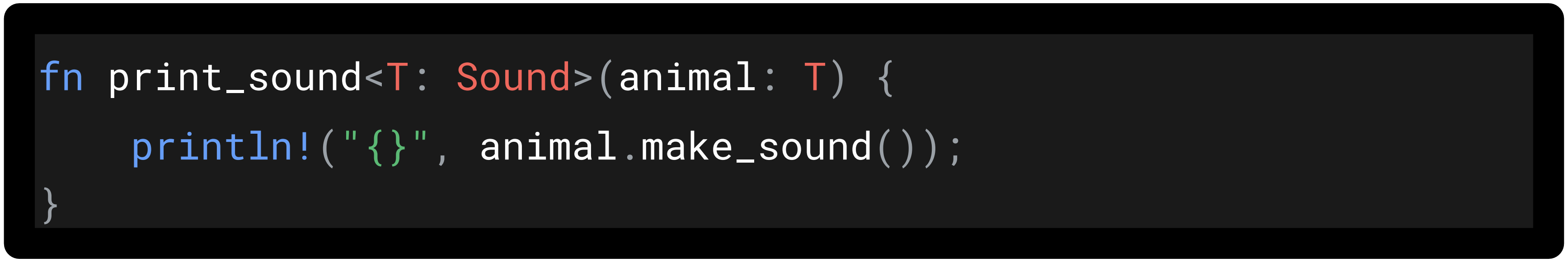
Generic type parameters are specified in angle brackets <T>

```rust
fn example<T>(sample: T) {
    println!("Received a value of a generic type.");
}
```

Traits Bound & Generic Parameters

“Trait bounds” constrain generics functions to types implementing specific traits.

Similarly, trait bounds can be used on generic structs in a similar way.



```rust
fn print_sound<T: Sound>(animal: T) {
    println!("{}", animal.make_sound());
}
```

Example Put Together

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=3f28d2c2ef4b84cc12980641722bbdb6

References

https://doc.rust-lang.org/book/ch03-02-data-types.html

https://www.rust-lang.org/production/users

https://www.rust-lang.org/

https://medium.com/@dexwritescode/comparison-between-java-go-and-rust-fdb21bd5fb7c

https://discord.com/blog/why-discord-is-switching-from-go-to-rust