The Rust Programming Language

Presented by: Ioannis Kostaras

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

FUTURE AGENDA

- Standard Library
- Closures
- Multi-threading
- Packages and Modules
- Error Handling
- Unit Testing

AGENDA

- ➢Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

For six years in a row, Rust has been voted the most loved programming language by Stack Overflow.

CHARACTERISTICS

- System's language
- Secure
- Strongly/statically typed
- Supports both functional and imperative paradigms
- Safe (memory safety without GC)
  - no need to manage memory (it is handled internally)
  - no Null Pointers
- Concurrent, for multicore systems
- Syntax similar to C++
- Open source software that is freely available to anyone and publicly shared

(FUTURE) DESIGN GOALS

- Reliable
  - if it compiles, it works!
- Performant
  - idiomatic code runs efficiently
- Supportive
  - the language, tools, and community are here to help
- Productive
  - a little effort => a lot of work
- Transparent
  - you can predict and control low-level details
- Versatile
  - you can do anything

HISTORY

Created by Mozilla in 2006 by

- Brendan Eich (https://brendaneich.com/)
- Dave Herman (https://medium.com/@davidherman)
- **Graydon Hoare** (https://gist.github.com/graydon)

Images: Wikipedia and the Internet

HISTORY

- Since: 2006 (project rust fungi)
- 15 May 2015: Stable Version 1.0
- August 2020, Mozilla laid off 250 stuff among them the Rust team
- 8 February 2021: Rust Foundation
  - AWS, Huawei, Google, Microsoft, and Mozilla
- 6 April 2021 Android supports Rust

- Current Version: **1.68.0 released 06/03/2023**

- Online:
  - https://play.rust-lang.org/
  - https://replit.com/

GRAPH

Tag
c++
rust

% of Stack Overflow questions that month

6.00%
5.50%
5.00%
4.50%
4.00%
3.50%
3.00%
2.50%
2.00%
1.50%
1.00%
0.50%
0.00%

2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021

Year

C → C++

JavaScript → TypeScript

Objective-C → Swift

Java → Kotlin?

C++ → Rust?

AGENDA

- Introduction and History
- ➢Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

PLAYGROUND

Rust Playground

https://play.rust-lang.org

RUN ▶ ··· DEBUG STABLE ··· SHARE TOOLS CONFIG

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

Execution

Close

Standard Error

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 1.41s
     Running `target/debug/playground`
```

Standard Output

```
Hello, world!
```

https://doc.rust-lang.org/stable/rust-by-example/

https://rust-by-example-ext.com

INSTALLATION

- Installation binaries can be downloaded for
  - Windows (install C++ Build Tools)
  - MacOSX
  - Linux

```
$ curl https://sh.rustup.rs -sSf | sh
$ rustup component add rust-docs
// MacOSX
$ brew install rustup-init
$ rustup-init
```

IDE SUPPORT

Source: https://areweideyet.com/

	Syntax highlighting (.rs)	Syntax highlighting (.toml)	Snippets	Code Completion	Linting	Code Formatting	Go-to Definition	Debugging	Documentation Tooltips
Atom	✓	✓	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]		✓[1]
Emacs	✓[1]	✓	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]		✓[1]
Sublime	✓	✓[1]	✓	✓[1]	✓	✓[1]	✓[1]		
Vim/Neovim	✓	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]		✓[1]
VS Code	✓	✓[1]	✓	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]
Show more editors ⇩									
Eclipse	✓[1]		✓[1]	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]
IntelliJ-based IDEs	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]
Visual Studio	✓			✓[1]			✓[1]	✓[1]	
GNOME Builder	✓		✓	✓	✓	✓[1]	✓		
Show more IDEs ⇩									

✓ = supported out-of-the-box, ✓[1] = supported via plugin

AGENDA

- Introduction and History
- Installation
  - ➢ Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

TOOLS

- `rustc` : Rust compiler
- `rustup` : command line utility to install and update Rust
- `cargo` :
  - Build system
  - Package manager
  - Test runner
  - Docs generator

Packages in Rust are referred to as crates and can be publicly found at https://crates.io

TOOLS

- rustfmt
  - `cargo fmt`
  - `.rustfmt.toml`
- clippy
  - `cargo clippy`
- doc
  - `cargo doc`
  - `target/doc/packagename/index.html`
- Rustup
  - `rustup doc --std`

RUST DOC

- `/// Inner documentation comment`
- `//! Outer documentation comment`
- Markdown
  - `#, ##, ###,...`
  - ``[`link`]``
  - `[Link](https://doc.rust-lang.org/book/)`
  - `` `code` ``
  - `- bullet1`
  - `- bullet2`

AGENDA

- Introduction and History
- Installation
  - Tools
- ➢ Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

MY FIRST PROGRAM (1/4)

```
$ cargo new hellorust
     Created binary (application) `hellorust` package

hellorust/
├── .git
├── .gitignore
├── Cargo.toml
└── src
    └── main.rs
$ cd hellorust
$ cargo run
Hello, world!
$ target/debug/hellorust
Hello, world!
```

MY FIRST PROGRAM (2/4)

- Cargo.toml

```
[package]
name = "hellorust"
version = "0.1.0"
authors = ["jkost <jkost@users.noreply.github.com>"]
edition = "2021"

[dependencies]
```

MY FIRST PROGRAM (3/4)

- main.rs

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

Macro: function with variable number of arguments
Not to be confused with C macros

MY FIRST PROGRAM (4/4)

- main.rs

Old way!

```
// main function
fn main() {
   println!("Hello, world!");
}
---------
$ rustc main.rs
$ ./main
Hello, world!
```

MY SECOND PROGRAM

```
fn main() {
	let name = "Ioannis";
	println!("Hello {}", name);
}
---------
Hello Ioannis
```

A *binding* binds a variable to a value. Binding's data types in Rust are implicitly inferred but can be explicitly declared using the separator operator (:).

MY THIRD PROGRAM

```
fn main() {
    let name = "Ioannis";
    println!("Hello {}", name);
    name = "Katerina";
    println!("Hello {}", name);
}
---------
error[E0384]: cannot assign twice to immutable variable
`name` --> src/main.rs:4:5 |
2 | let name = "Ioannis";
  | ----
  | |
  | first assignment to `name`
  | help: consider making this binding mutable: `mut
name`
3 | println!("Hello {}", name); 4 | name = "Katerina";
  | ^^^^^^^^^^^^^^^^^^ cannot assign twice to immutable
variable
```

Bindings are immutable by default

MY THIRD PROGRAM (CONT.)

```
fn main() {
    let mut name = "Ioannis";
    println!("Hello {}", name);
    name = "Katerina";
    println!("Hello {}", name);
}
---------
Hello Ioannis
Hello Katerina
```

VARIABLE DECLARATION

- `let` (by default all variables are immutable)
- `const`
- `static`

```
let my_variable = 0;
const PI: f32 = 3.14;
static MY_STRING: String = "RUST";
```

VARIABLE DECLARATION CONVENTIONS

Object	Case
Variables	snake_case
Functions	snake_case
Files	snake_case
Constants	SCREAMING_SNAKE_CASE
Statics	SCREAMING_SNAKE_CASE
Types	PascalCase
Traits	PascalCase
Enums	PascalCase

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - ➢ Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

DATA TYPES

- Rust contains 25 primitive data types:
  - 12 **integer** primitive data types
  - 2 **floating point** primitive data types
  - 1 **logical** primitive data type
  - 1 **character** primitive data type
  - 1 primitive data type for **string slices**
  - **array** is a primitive data type in Rust
  - **tuple**, a finite heterogeneous sequence of data
  - 2 **pointer** data types, 1 raw unsafe pointer to data and 1 function pointer
  - 1 **reference** data type
  - **unit** data type
  - **never** data type

DATA TYPES

- Integer:
  `i8, i16, i32, i64, i128, isize,`
  `u8, u16, u32, u64, u128, usize`
  - `u` means *unsigned* data while `i` means *signed* data.
  - `u32` represents an unsigned 32-bit integer, while `i64` represents a signed 64-bit integer.
  - `isize` and `usize` are types that can vary in size.

- Literals:
  - Decimal: `1000`
  - Hex: `0xdeadbeef`
  - Octal: `0o77543`

  Binary: `0b11100`

  Byte (u8): `b'A'`

DATA TYPES (CONT.)

- Floating-point (IEEE-754): `f32, f64`

```
let interest: f32 = 3.;
```

- Boolean: `true` or `false`
- Character (UCS-4/UTF-32): `'c'`
- String: `"This is a string"`
- Array: `a = [1,2,3], a:[f32;2]` `a[0]`
- Slice: `&a[1...2]`
- Tuple: `t = (1, 2.0, 3)` `t.0`

DATA TYPE CONVERSION

```
let var1: f32 = 3.14;
// convert f32 to i32
let var2: i32 = var1 as i32;
println!("{} {}", var1, var2);
---
3.14 3
```

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - ➢ Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

CONTROL FLOW

- Conditionals:
  - `if x {…}`
  - `match x {…}`

- Loops:
  - `loop { ... }`
  - `while x { ... }`
  - `for x in 1..100 { ... }`

Infinite loop

IF, LET-IF

```
let mark = if grade >= 5 {
   "Pass!"
} else {
   "Fail!"
};
```

MATCH

```
fn main() {
  let mark = 16;
  match mark {
    19..=20 => println!("Excellent!"),
    17 | 18 => println!("Very Good."),
    15 | 16 => println!("Good."),
    m@10..=14 => println!("Average: {}", m),
    _ => println!("Rejected.")
  }
}
```

MATCH (CONT.)

- Matches in Rust are *exhaustive*, which means that the code must cover all potential scenarios in order to be valid.
- If we forget to write the `None` case, the Rust compiler will report "`pattern 'None' not covered`" as an error.

LOOPS

```
loop {
  …
  break;
}
---
while
condition{
 // if true
}
```

```
for var in condition {
  …
}
---
for num in 1..4 {
  print!("num={},", num);
}
---
num=1,num=2,num=3,
```

Range expression

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - ➢ Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

ARRAYS

- Fixed size
- Contain values of *only one type*.

The compiler needs to know the size

```
let mut array: [type; length] = [default; length];
let array: [type; length] = [val1, val2, val3, …];
------------------------------------------
let mut arr: [i32; 4] = [1; 4];
arr[1] = 10;
arr[2] = 20;
println!("{} {} {} {}", arr[0], arr[1], arr[2], arr[3]);
println!("{:?}", arr);
----
1 10 20 1
[1 10 20 1]
```

Debug print

SLICE

- Slice is a data type that isn't owned by anyone.
- A slice is derived from an existing variable rather than being constructed from scratch.
- Instead of referencing the entire collection, a slice refers to a contiguous memory allocation.
- Think of them as *views* into an underlying array of values.
- Slices *borrow* their data from their arrays.
- It allows you to access an array in a safe and efficient manner without having to replicate it.
- Slices only *behave* like arrays.
- The size of a slice is only known at run-time

ARRAY SLICE

```
let slice = &array[ start..end-1 ];
-------------------------------------
let arr:[f32; 4] = [1.0, 2.0, 3.0, 4.0];
let slice = &arr[1..3];
println!("{} {} ", slice[0], slice[1]);
-----
2.0 3.0
```

ARRAY SLICE EXAMPLE

```
fn sum(values: &[i32]) -> i32 {
    let mut res = 0;
    for i in 0..values.len() {
        res += values[i]
    }
    res
}

fn main() {
    let arr = [10,20,30,40];
    let res = sum(&arr);
    println!("sum = {}", res);
}
---
sum = 100
```

`arr` is borrowed

Rust Array ⇔ C array
Rust slice ⇔ C pointer

TUPLE

- A collection of different data types

```
let tuple = (val1, val2, val3…);
```

```
let t = ("ETA", 8, 'h');
print!("{}: {}{}", t.0, t.1, t.2);
---
ETA: 8h
```

TUPLES - EXTRACT VALUES

```
let tup = (1,"hello".to_string());
let (num,s) = tup; // tup moved
println!("{:?}", tup);
------
error[E0382]: borrow of partially moved value: `tup`
 --> src/main.rs:4:18
  |
3 | let (num,s) = tup; // tup moved
  |          - value partially moved here
4 | println!("{:?}", tup);
  |                  ^^^ value borrowed here after partial move
```

TUPLES - EXTRACT VALUES (CONT.)

```
let tup = (1,"hello".to_string());
let (num,ref s) = tup; // borrowing is OK
println!("{:?}", tup);
------
(1, "hello")
```

TUPLES - EXTRACT VALUES (CONT.)

```
#[derive(Debug)]
struct Point {
    x: f32,
    y: f32
} // structs implement Copy trait
fn main() {
    let p = Point{x:0.0,y:0.0};
    let Point{x,y} = p;
    println!("{:?}", p);
}
-----
Point { x: 0.0, y: 0.0 }
```

TUPLES AND MATCHING

```
let t: (i32,String) = (10, "ETA".to_string());

let text = match t {
    (0, s) => format!("zero {}", s),
    (10, ref s) if s == "ETA" => format!("{} = 10 minutes", s),
    (n, _) if n > 10 => format!("Too late: {} minutes", n),
    _ => format!("no match")
};

println!("{}", text);
-----
ETA = 10 minutes
```

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - ➢ Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

STRINGS & STRING SLICES

- There are two kinds of strings in Rust:
  - Owned (`String`) and borrowed (`&str`)
- `String`s are like `Vec`s, allocated dynamically and resizable
- Strings in Rust are UTF-8
- String literals are slices (`&str`)
  - `String` ⇔ `Vec<u8>`
  - `&str` ⇔ `*u[8]`
- String slices are immutable and of fixed size
- Strings are *not* arrays of chars!
- The compiler can convert `String`s to `&str` but not vice versa
  - use `to_string()`

STRINGS

```
let mut s = String::new();
```

```
fn main() {
   let s: String = "Bye";
   println!("{}", s);
}
error[E0308]: mismatched types
 --> src/main.rs:2:20
  |
2 |     let s: String = "Bye";
  |            ------   ^^^^^
  |            |        |
  |            |        expected struct `String`, found `&str`
  |            |        help: try using a conversion method: `"Bye".to_string()`
  |            expected due to this
```

A `String` *slice*, not a `String`

or `String::from("Bye")` or `let s:&str = "bye";`

STRING SLICE

- String literals are considered string slices since they are stored in binary.
- `&str` is an immutable reference, and string literals are immutable.

```
let slice = &string[ start..end-1 ];
---------------------------------------
let str=String::from("Hello world");
let hello_slice1 = &str[0..5];
let hello_slice2 = &str[0..=4];
let hello_slice = &str[..];
println!("{}, {}, {} ", hello_slice1, hello_slice2,
hello_slice);
-----
Hello, Hello, Hello world
```

STRINGS AND &STR

Stack

String

Name	Value
length	12
capacity	14
ptr	

&str

Name	Value
length	2
ptr	

Heap

1	2	7	.	0	.	0	.	1	:	8	0		

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - ➢ Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

FUNCTIONS

```
fn f_to_c(fahreneit: f64) -> f64 {
    // return (f - 32) / 1.8;
    (f - 32) / 1.8
}
```

PASS ARGUMENTS BY VALUE

```
fn pow(x: i32) -> i32 {
      x*x  // ⇔ return x*x;
}
fn main() {
   let n: i32 = 4;
   println!("{}^2 = {}", n, pow(n));
}
----
4^2 = 16
```

Take ownership

PASS ARGUMENTS BY REFERENCE

```
fn pow(x: &i32) -> i32 {
    return x * x;
}
fn main() {
    let n: i32 = 4;
    println!("{}^2 = {}", n, pow(&n));
}
----
4^2 = 16
```

borrow

Passing by reference is important when we have a large object and don't wish to copy it.

MUTABLE REFERENCES

```
fn pow(x: &mut i32) {
    *x = *x * *x;
}
fn main() {
    let mut n: i32 = 4;
    pow(&mut n);
    println!("n = {}", n);
}
----
4^2 = 16
```

PASS A STRING TO A FUNCTION

```
fn report(s: &str) {
    println!("str '{}'", s);
}

fn main() {
    let text = "Hello World";  // the string slice
    let s = text.to_string();  // an allocated
string
    // the borrow operator & coerces String to &str
    report(text);
    report(&s);
}
```

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- ➢ Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

STACK

- LIFO (Last-In First-Out)
- Limited in size
- Very fast
- Stores data whose size is known at compile time, e.g. integers, booleans, characters, arrays
- Stores function variables in stack frames (their scope)
  - Every function has its own stack frame
- When a function exits it's stack frame is released (allocated memory is managed for us)
- Stack overflow

STACK

```
fn main() {
    let a = 1;
    stack_only(a);
}

fn stack_only(b: i32) {
    let c = 2;
}
```

stack_only()
c = 2
b = 1

main()
a = 1

HEAP

- Stores data whose size is unknown at compile time
- The operating system returns a pointer to an empty place in the heap memory. This is referred to as "allocating on the heap."
- Not automatically managed
  - (De-)allocate memory manually
- Accessible by any function
- Heap allocations are expensive
- Heap fragmentation

HEAP

Heap

0xff123454: 4

Stack

stack_and_heap()
e = (0xff123454)
d = 3

stack_only()
c = 2
b = 1

main()
a = 1

```
fn main() {
    let a = 1;
    stack_only(a);
}

fn stack_only(b: i32) {
    let c = 2;
    stack_and_heap();
}

fn stack_and_heap() {
    let d = 3;
    let e = Box::new(4);
}
```

BOXES

- All values in Rust are stack allocated by default.
- Values can be boxed (allocated on the heap) by creating a Box<T>.
- A box is a smart pointer to a heap allocated value of type T.
- When a box goes out of scope, its destructor is called, the inner object is destroyed, and the memory on the heap is freed.
- `Box::from(`*`variable`*`)`

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - ➢ Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

VARIABLE OWNERSHIP

- Ownership is a unique feature of the Rust programming language that ensures memory safety without the use of a garbage collector or pointers.
- The term "ownership" refers to when a piece of code owns a resource. The code constructs an object that holds the resource. The object is destroyed and the resource is freed when the control reaches the conclusion of the block.
- Rust utilizes the *borrow checker* during compile time. If it compiles, it will most likely work during runtime.

VARIABLE OWNERSHIP (CONT.)

3 Rules:

1. Each value has an owner (a variable that owns it)
2. There can be only one owner at a time
3. Value gets dropped if its owner goes out of scope

OWNER

VARIABLE OWNERSHIP (CONT.)

- Every value in Rust has a variable linked with it, which is referred to as its *owner*.
- Ownership can be transferred from one variable to another.
- The "owner" of a variable can modify its owning value.
- Only one owner can be present at any given moment.
- When the owner is removed from the scope, the value connected with them is lost.

VARIABLE OWNERSHIP (CONT.)

```
let myvar = 42;
```

- `myvar` is a *variable binding*
- `myvar` *owns/is bound to* the value 42
- Every value has exactly one owner
- Owner can only change value if it's *mutable*

```
let mut myvar = 42;
```

```
let x = String::from("Hi"); // x owns "Hi"
let y = x; // Warning! The ownership of x moves to y
println!("{}", x); // Error! x is no longer available
```

VARIABLE OWNERSHIP (CONT.)

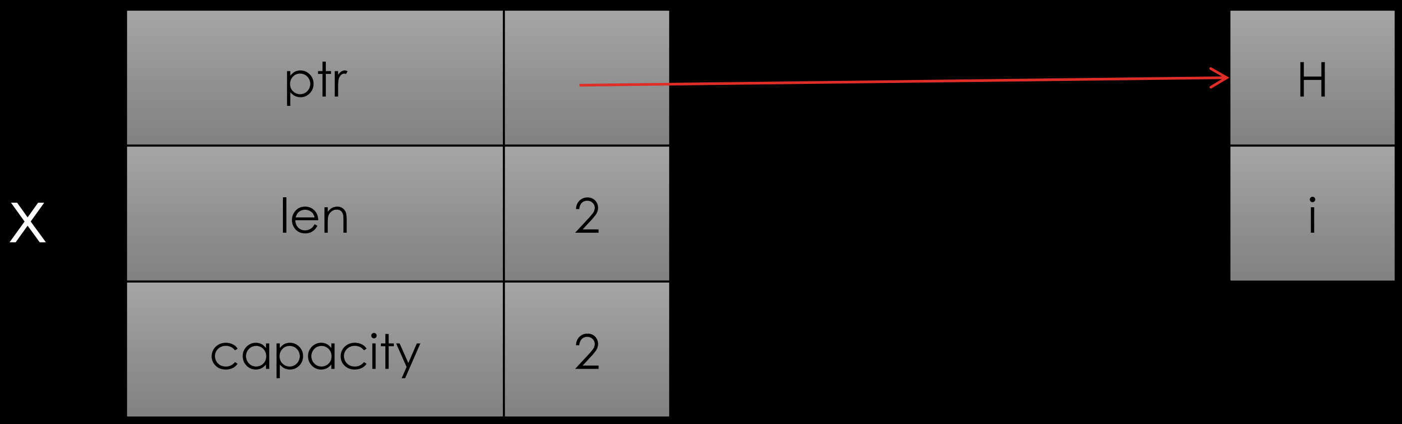
```
3 | let y = x; // Warning! The ownership of x moves to y
  |     ^ help: if this is intentional, prefix it with an underscore: `_y`

error[E0382]: borrow of moved value: `x`
 --> src/main.rs:4:16
  |
2 | let x = String::from("Hi"); // x owns "Hi"
  |     - move occurs because `x` has type `String`, which does not implement the `Copy` trait
3 | let y = x; // Warning! The ownership of x moves to y
  |         - value moved here
4 | println!("{}", x); // Error! x is no longer available
  |                ^ value borrowed here after move

error: aborting due to previous error; 1 warning emitted

For more information about this error, try `rustc --explain E0382`.
```

OWNERSHIP APPLIES TO REFERENCES ONLY

```
let x = String::from("Hi"); // x owns "Hi"
let y = x; // Warning! The ownership of x moves to y
println!("{}", x); // Error! x is no longer available
```

```
let x = 42;
let y = x;
println!("{}", x);
----
42
```

VARIABLE OWNERSHIP

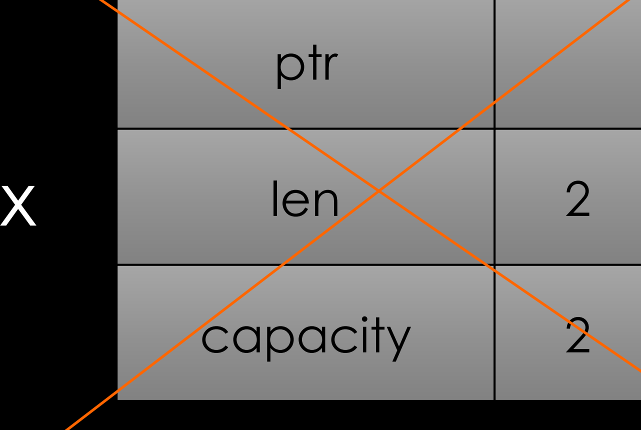


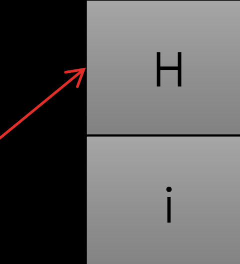
VARIABLE OWNERSHIP

Stack

Heap

.)





y

ptr	
len	2
capacity	2

COPY & CLONE TRAITS

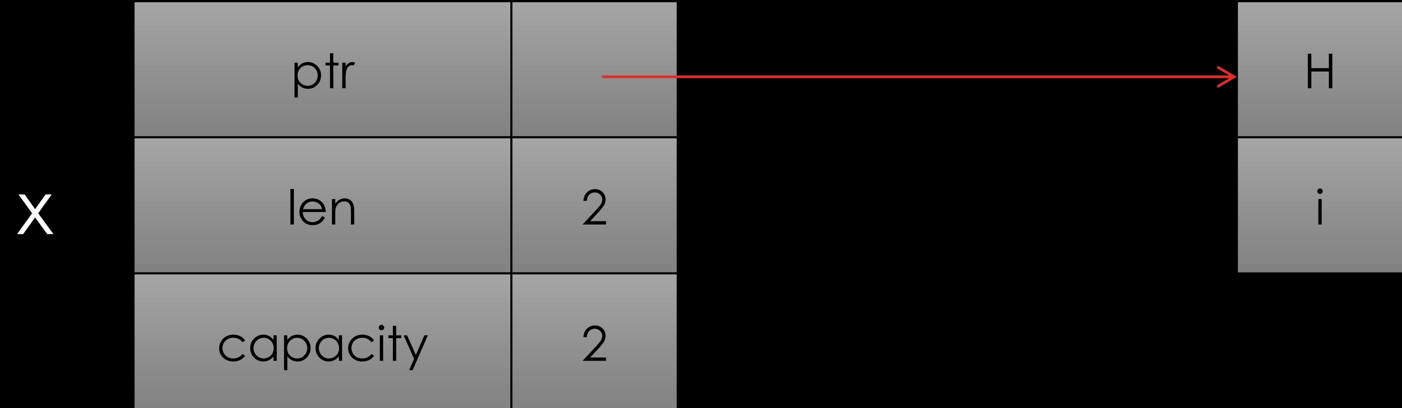
- The copy trait is a particular annotation that is applied to types that are stored on the stack.
- If the types have the copy trait, the older variable can be used even after the assignment action.
- Clone is explicit by using the `clone()` method and creates a duplicate owner to a binding.

VARIABLE OWNERSHIP (CONT.)

```
let x = String::from("Hi"); // x owns "Hi"
let y = x.clone();
println!("{}", x); // "Hi"
```

VARIABLE OWNERSHIP

Stack Heap .)



y

ptr	
len	2
capacity	2

H
i

FUNCTION OWNERSHIP

- When a variable is handed to a function, ownership is transferred to the called function's variable.
- Passing value has the same semantics as assigning a value to a variable.
- When you return values from a function, you're also transferring ownership.

FUNCTION OWNERSHIP

```
fn main() {
    let x = String::from("Hi"); // x owns "Hi"
    report(x);
    println!("{}", x); // Error! x is no longer available
}

fn report(s: String) {
     println!("{}", s);
}
```

Value borrowed here after move

Ownership is transferred to `s`.

FUNCTION OWNERSHIP ERROR MESSAGE

```
error[E0382]: borrow of moved value: `x`
 --> src/main.rs:8:19
  |
6 |     let x = String::from("Hi"); // x owns "Hi"
  |         - move occurs because `x` has type
`String`, which does not implement the `Copy` trait
7 |     report(x);
  |            - value moved here
8 |     println!("{}", x); // Error! x is no longer
available
  |                    ^ value borrowed here after
move
```

FUNCTION OWNERSHIP SOLUTION

```
fn report(s: &String) {
    println!("{}", s);
}

fn main() {
   let x = String::from("Hi"); // x owns "Hi"
   report(&x);
   println!("{}", x); // Error! x is no longer available
}
```

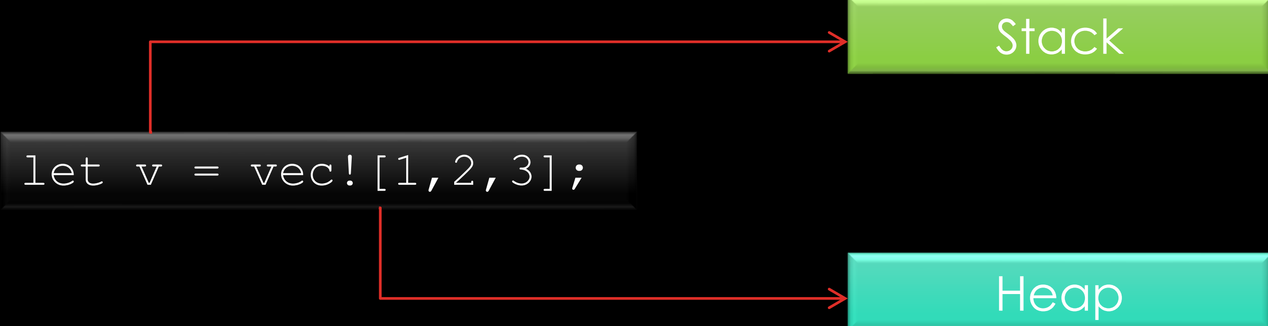
Ugly; better use `&str`
Rust converts `&String` to `&str`

FUNCTION OWNERSHIP

```
fn report(arr: &[i32]) {
    println!("arr is {:?}", arr);
}

fn main() {
    let mut v = Vec::new();
    v.push(1);
    v.push(2);
    v.push(3);

    report(&v);

    let slice = &v[1..];
    println!("slice is {:?}", slice);
}
```

borrow operator & is coercing the vector into a slice

MEMORY MANAGEMENT (CONT.)



MEMORY MANAGEMENT (CONT.)

```
let v = vec![1,2,3];
```

Stack

v

length

capacity

Heap

1

2

3

MEMORY MANAGEMENT (CONT.)

```
let v = vec![1,2,3];
let w = v;
```

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

REFERENCES

- References allow us to refer to a value without taking ownership of it
- References are immutable by default
- mutable references: `&mut`
- In the same scope we can many immutable references or one mutable reference
- If we have a mutable borrow we can't have any other borrows
- No data races!

REFERENCES (CONT.)

- A reference is an address that is supplied as an argument to a function. Similar to a pointer in C.
- All references are borrowed from some value, and all values have lifetimes
- The lifetime of a reference cannot be longer than the lifetime of that value.

REFERENCES (CONT.)

- Borrowing is the same as borrowing something and returning it after we are finished with it.
- Borrowing permits many references to a single resource while maintaining the need of a "single owner."
- Borrowing and references are mutually exclusive, meaning that when a reference is relinquished, the borrowing ceases as well.
- There are two sorts of references:
  - Mutable: are relocated
  - Immutable: are copied

REFERENCES (CONT.)

- After a variable is referenced by other variables, the ownership of its value remains and will not be lost.

```
&variable
parameter: &type
```

```
fn main() {
  let s=String::from("Hi"); // s owns "Hi";
  let n=length(&s); // s still owns the value
  println!("Value: {}",s);
  println!("Length: {}",n);
}

fn length(str:&String) -> usize {
  str.len() // get the length of the string
}
```

When the variables are supplied to the function as a reference rather than actual values, we don't need to return the values to reclaim

REFERENCES (CONT.)

```
fn main() {
 let a=1;
 double_it(&a); println!("Value: {}",a);
}
fn double_it(x:&i32) {
 *x *= 2
}
```

```
error[E0594]: cannot assign to `*x`, which is behind a `&` reference
 --> src/main.rs:6:2
  |
5 | fn double_it(x:&i32) {
  |                ---- help: consider changing this to be a mutable reference: `&mut i32`
6 |  *x *= 2;
  |  ^^^^^^^ `x` is a `&` reference, so the data it refers to cannot be written

For more information about this error, try `rustc --explain E0594`.
```

REFERENCES (CONT.)

```
fn main() {
 let a=1;
 double_it(&a); println!("Value: {}",a);
}
fn double_it(x:&mut i32) {
 *x *= 2
}
```

Mutable reference

```
error[E0308]: mismatched types
 --> src/main.rs:3:12
  |
3 |  double_it(&a);
  |            ^^ types differ in mutability
  |
  = note: expected mutable reference `&mut i32`
                     found reference `&{integer}`

For more information about this error, try `rustc --explain E0308`.
```

REFERENCES (CONT.)

```
fn main() {
 let a=1;
 double_it(&mut a); println!("Value: {}",a);
}
fn double_it(x:&mut i32) {
 *x *= 2;
}
```

```
error[E0596]: cannot borrow `a` as mutable, as it is not
declared as mutable
 --> src/main.rs:3:12
  |
2 |   let a = 1;
  |       - help: consider changing this to be mutable: `mut
a`
3 |   double_it(&mut a);
  |            ^^^^^^ cannot borrow as mutable

For more information about this error, try `rustc --explain
E0596`.
```

REFERENCES (CONT.)

```
fn main() {
 let mut a=1;
 double_it(&mut a); println!("Value: {}",a);
}
fn double_it(x:&mut i32) {
 *x *= 2;
}
```

```
Value: 2
```

RESTRICTIONS OF MUTABLE REFERENCES

- In a given scope, we can only have one mutable reference

```
let mut s=String::from("Hi");
let s1 = &mut s;
let s2 = &mut s;
-----
error[E0499]: cannot borrow `s` as mutable more than once at a
time
```

RESTRICTIONS OF MUTABLE REFERENCES

- If we have an immutable reference, then we can't have a mutable reference.

```
let mut s=String::from("Hi");
let s1 = &s; let s2 = &mut s;
-----
error[E0499]: cannot borrow `s` as mutable because it is also
borrowed as immutable
```

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- Summary

SCOPES

```
{
    let a = 5;
    let b = "hi";
    {
        let c = "hi".to_string();
        // a,b and c are visible
    }
    // a,b are visible, c is dropped
    for i in 0..a {
        let b = &b[i..];
        // original b is shadowed.
    }
    // b is dropped, i is not visible
    // a, b are visible
}
```

SCOPES

```
let s = "hello".to_string();
let mut rs = &s;
{
    let tmp = "hello world".to_string();
    rs = &tmp;
}
println!("ref {}", rs);
------
error: `tmp` does not live long enough
  --> ref.rs:8:5
   |
7  |         rs = &tmp;
   |               --- borrow occurs here
8  |     }
   |     ^ `tmp` dropped here while still borrowed
9  |     println!("ref {}", rs);
10 | }
   | - borrowed value needs to live until here
```

LIFETIMES

- Rust is a *block-scoped* language.
- Variables only exist for the duration of their block
- Variable binding disappears when it goes out of scope, it releases the resource, and loses ownership.
- No borrow may outlive its value's owner.

```
fn main() {
  let s=String::from("Hi"); // s owns "Hi"; "Hi" is bound to s
  let n=length(s); // Warning! s will lose the ownership after
used
  println!("Value: {}",s); // s is no longer available
  println!("Length: {}",n);
}

fn length(str:String) -> usize { // str takes onwership of "Hi"
  str.len() // get the length of the string
}
```

LIFETIMES

```
error[E0382]: borrow of moved value: `s`
 --> src/main.rs:4:41
  |
2 |   let s=String::from("Hi"); // s owns "Hi"
  |         - move occurs because `s` has type `String`, which does not implement the `Copy` trait
3 |   let n=length(s); // Warning! s will lose the ownership after used
  |                - value moved here
4 |   println!("Value: {}",s); // s is no longer available
  |                        ^ value borrowed here after move

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
```

LIFETIMES

```
fn main() {
  let c='a'; // c owns 'a'; 'a' is bound to c
  prnt(c);
  println!("Value: {}",c); // char has the copy trait
}

fn prnt(ch:char) { // ch takes onwership of 'a'
  println!("Value: {}",ch);
}
-------
Value: a
Value: a
```

LIFETIMES

- To be able to statically check all the references in our code, the Rust compiler makes use of lifetime specifiers, i.e. special annotations to our references
- Lifetime example: `<'buf>`

```
s: &'buf str
```

LIFETIMES

- Lifetimes allow the Rust compiler to guarantee memory safety
- Lifetime parameters don't allow us to choose for how long a value lives
  - they communicate to the compiler that some references are related to the same memory and are expected to share the same lifetime

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- ➢ Object Orientation
  - Structs, Enums and Traits
- Summary

CUSTOM TYPES

- Structs & Enums
- Traits
  - describe a type's abilities
  - glue the data types together

OO IN RUST

- OO Properties:
  - Encapsulation √
  - Abstraction √
  - Inheritance (√)
  - Polymorphism √

- *Encapsulation is supported with Structs and modules*
- *Only Interface Inheritance is supported via Traits (no implementation inheritance)*
  - *Only Trait inheritance*
- *Polymorphism via trait objects*
  - *Bounded parametric polymorphism: generics + trait bounds*
- *Monomorphism via generics*

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - ➢ Structs, Enums and Traits
- Summary

STRUCT

- a user-defined data type
- Rust has three struct types: a classic C struct, a tuple struct, and a unit struct.
- Struct members are called *fields*
- *No struct inheritance!*

```
// struct creation
struct StructName {
member1: type,
member2: type,
…
}
```



```
// struct initialization
let object = StructName {
member1: value1,
member2: value2,
…
}
```

STRUCT EXAMPLE

```
// struct creation
struct Book {
  isbn: String,
  pages: u16,
}
fn main() {
     let book = Book {
          isbn: "123456789".to_string(),
          pages:123};
     print!("ISBN: {}, pages: {}",
            book.isbn, book.pages);
}
----
ISBN: 123456789, pages: 123, …
```

STRUCT METHODS

- `self` represents the instance on which the function is called
- it is always the first parameter of such methods

```
impl Struct {
    fn method_name(&self) -> type {
        self.member // access the member variable
    }
}
```

STRUCT METHODS AND ASSOCIATED FUNCTIONS

- Structs consist of 2 parts:
  - Definition which defines the data (fields)
  - Implementation with `impl` block which defines the functionality
- A Struct can have two types of functionality:
  - Methods
    - require `self` as the first parameter (equivalent to `this`)
    - `self` represents the instance on which the function is called
  - Associated functions are associated with the struct type
    - don’t need an instance of the struct (like static methods)

STRUCT METHODS

```
struct Book {
  isbn: String,
  pages: u32,
}
impl Book {
    fn new(isbn: &str, pages: u32) -> Book {
        Book {
            isbn: isbn.to_string(),
            pages: pages,  // pages
        }
    }
}
fn main() {
     let book = Book::new("123456789", 123);
     print!("ISBN: {}, pages: {}, …",
             book.isbn, book.pages);
}
```

Self

Self

Associated function

STRUCT METHODS

```
struct Book {
  isbn: String,
  pages: u32,
  available: bool
}
impl Book {
    fn new(isbn: &str, pages: u32, avail: bool) -> Book{
        Book {
            isbn: isbn.to_string(),
            pages: pages,
            available: avail
        }
    }
    fn available(&self) -> bool {  &self.available }
}
fn main() {
      let book = Book::new("123456789", 123, true);
      if book.available() { //… }
}
```

STRUCT METHODS

```
struct Book {
  isbn: String,
  pages: u16,
  available: bool
}
impl Book {
    //...
    fn set_isbn(&mut self, isbn: &str) {
        self.isbn = isbn.to_string();
    }
    fn copy(&self) -> Self {
       Self::new(&self.isbn,&self.pages,&self.available)
    }
}
```

STRUCT METHODS SUMMARY

- no `self` argument: you can associate functions with structs, like the `new` "constructor".
- `&self` argument: can use the values of the struct, but not change them
- `&mut self` argument: can modify the values
- `self` argument: will consume the value, which will move.

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - ➢ Structs, Enums and Traits
- Summary

ENUM

- A set of fixed values
- Can contain methods
- Similar to C unions

```
enum Enum_Name{
  value1,
  value2,
…
}
Enum_Name::value1
```

ENUM EXAMPLE

```
enum Choice {
  RedPill, BluePill
}
let choice = Choice::RedPill;
match choice {
  Choice::BluePill => println!("Keep on
sleeping..."),
  Choice::RedPill => println!("Welcome to
Matrix!")
}
```

ANOTHER ENUM EXAMPLE

```
enum Day {
  MON, TUE, WED, THU, FRI, SAT, SUN
}
fn work_or_not(day:Day) -> bool {
   match day {
      Day::MON | Day::TUE | Day::WED | Day::THU | Day::FRI => true,
      Day::SAT | Day::SUN => false
   }
}
fn main() {
  println!("Work on Monday? {}", work_or_not(Day::MON));
  println!("Work on Sunday? {}", work_or_not(Day::SUN));
}
```

ENUM METHODS

```
impl Enum {
   fn method_name(&self) -> type {
      self.member // access the member variable
   }
}
```

ENUM METHOD EXAMPLE

```
enum Day {
  MON, TUE, WED, THU, FRI, SAT, SUN
}
impl Day {
    fn work_or_not(self) -> bool {
        match self {
            Day::MON | Day::TUE | Day::WED | Day::THU | Day::FRI
=> true,
            Day::SAT | Day::SUN => false
        }
    }
}
fn main() {
  println!("Work on Monday? {}", Day::work_or_not(Day::MON));
  println!("Work on Sunday? {}", Day::work_or_not(Day::SUN));
```

ENUMS (CONT.)

- Rust's enums are similar to algebraic data types in functional languages, such as F#, OCaml, and Haskell.
- Enums can
  - have methods defined on them
  - implement traits
- No ordering/ordinal (use `PartialOrd` trait)
- Enum values can have default values
- Enum values can be of different types
- Enum values cannot be compared (use `PartialEq` trait)

ENUMS (CONT.)

```
enum Guard {
      Battalion (String),
      Move { x: i32, y: i32 },
      Color (u16, u16, u16)
}

Guard::Move { x: 10, y: 30 }
Guard::Battalion(String::from("Red"))
Guard::Color(200, 255, 255),
```

OPTION<T>

```
enum Option<T> {
      Some(T),
      None,
}

let var = slice.get(5);
match var {
   Some(x) => { println!("Value is {}", x); },
   None(x) => { println!("No Value"); },
   _       => { println!("Who cares!"); }
}
var.is_some();
var.is_none();
var.unwrap();
```

IF LET

```
let arguments: Vec<String> = env::args().collect();
let who = match arguments.get(1) {
    Some(someone) => someone,
    None => "World"
};
println!("Hello, {who}!");
-----------------------------------------------------------
let arguments: Vec<String> = env::args().collect();
let who = if let Some(someone) = arguments.get(1) {
    someone
} else {
    "World"
};
println!("Hello, {who}!");
```

RESULT<T, E>

```
enum Result<T, E> {
     Ok(T),
     Err(E),
}

use std::fs::File;
fn main() {
     let result = File::open("passwd");
     if result.is_ok() {
          let f = result.unwrap();
     }
     let e = result.expect("error message");
}
```

RESULT<T, E>

```
enum Result<T, E> {
     Ok(T),
     Err(E),
}

use std::fs::File;
fn main() {
     let result = File::open("passwd");
     match result {
          Ok(f)  => { /* do stuff */ },
          Err(e) => { /* do stuff */ },
     }
    // let result = File::open("passwd")?;
}
```

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - ➢ Structs, Enums and Traits
- Summary

TRAITS

- describe a type's abilities
- glue the data types together

TRAITS

First letter should be capital

```
trait Trait_Name {  // similar to interface
    type: member;
    fn trait_method(&self);
}

impl Trait_Name for Struct_Name {
   fn trait_method(&self){self.member}
}
```

TRAITS

```
trait Borrowed {
      fn is_borrowed(&self) -> bool;
}

impl Borrowed for Book {
     fn is_borrowed(&self) -> bool {
            !self.is_available()
      }
}
```

TRAITS

```
trait Borrowed {
	fn is_borrowed(&self) -> bool;
}

fn borrowed<T: Borrowed>(item: T) {
  println!("{}", item.is_borrowed());
}
```

TRAITS

- Trait functions can have default implementations
- Implementors can override these default implementations

```
trait Borrowed {
     fn is_borrowed(&self) -> bool {
       false
    }
}

fn borrowed<T: Borrowed>(item: T) {
  println!("{}", item.is_borrowed());
}
```

TRAIT INHERITANCE

```
trait Borrowed {
    fn is_borrowed(&self) -> bool;
}
trait Returned {
    fn is_returned(&self) -> bool;
}

trait Available: Borrowed+Returned{}
```

No dynamic casting, e.g. `Borrowed` to `Available`

COPY TRAIT

```
#[derive(Copy)]
struct Book {
  isbn: String,
  pages: u32,
  available: bool
}
```

ERROR TRAIT

```
std::error::Error
pub trait Error: Debug + Display {
...
}
```

Debug formatter {:?}

Format string

Trait inheritance

DERIVABLE TRAITS

```
#[derive(Copy)]
#[derive(Debug)]
println!("{:?}", x); // Debug
println!("{:#?}", x); // Pretty debug

#[derive(Clone)]
#[derive(Default)]
```

DEREF TRAIT

- the trait for the 'dereference' operator `*`
- `String` implements `Deref<Target=str>` and so all the methods defined on `&str` are automatically available for `String` as well
- Same for `Box<Something>` and `Something`

STATIC & DYNAMIC DISPATCH

```
pub fn send(&self, stream: &mut dyn Write) {}

pub fn send(&self, stream: &mut impl Write) {}
```

OO SUMMARY

- class ⇔ data and traits
- structs and enums are dumb,
  - although you can define methods and do data hiding
- a limited form of subtyping is possible on data using the `Deref` trait
- traits don't have any data
  - but can be implemented for any type (not just structs)
- traits can inherit from other traits
- traits can have provided methods, allowing interface code re-use
- traits give you both
  - virtual methods (polymorphism)
  - generic constraints (monomorphism)

AGENDA

- Introduction and History
- Installation
  - Tools
- Basics
  - Data types
  - Variables, Control flow and loops
  - Arrays, Tuples
  - Strings and Slices
  - Functions
- Memory Management
  - Ownership, References, Lifetimes
- Object Orientation
  - Structs, Enums and Traits
- ➢ Summary

POINTS TO REMEMBER

- Type inference
  - Rust is a strongly typed static language
  - Rust likes to *infer* types, but you *can't* change the inferred type later; e.g. Rust won't automatically convert between `u32` and `u64`
- Mutable References
  - There can be only one mutable reference at a time
  - can't have immutable references while there's a mutable reference out
  - the borrow checker is not always as smart as it could be

POINTS TO REMEMBER (CONT.)

- References & Lifetimes
  - Rust cannot allow a situation where a reference outlives the value
  - Otherwise we would have a 'dangling reference' where it refers to a dead value a.k.a. a segmentation fault
  - An explicit lifetime is needed when a struct or a function borrows a reference, unless *lifetime elision* can be applied
  - For both structs and functions, the lifetime needs to be declared in <> like a type parameter, e.g. `<'a>`

POINTS TO REMEMBER (CONT.)

- Strings and literals
  - `String` is an *owned* string, allocated on the heap
  - a *string literal* (e.g. "hello") is of type `&str` ("string slice") and might be either put into the executable ("static") as is or borrowed from a `String`
  - `String` and `&String` are different types
  - `s1 + s2`

```
   ▲     ▲
String  &str
```

BOOKS

- Klabnic S. & Nichols C. (2021), *The Rust Programming Language,* 2nd Ed., No Starch Press.
- Abhishek K. (2022), *Rust Crash Course,* BPB.
- Alves C. (2021), *Rust Programming Language,* 3rd Ed.
- Anderson B. (2023), *Rust for Network Programming and Automation,* GitforGits.
- Bhattacharjee J. (2020), *Practical Machine Learning with Rust,* Apress.
- Bos M. (2023), *Rust Atomics and Locs,* O'Reilly.
- Blandy J. (2015), *Why Rust,* O'Reilly.
- Blandy J. et al. (2021), *Programming Rust*, 2nd Ed., O'Reilly.
- Eshwarla P. (2020), *Practical System Programming for Rust Developers*, Packt.

BOOKS (CONT.)

- Flitton M. (2023), *Rust Web Programming,* Packt.
- Gjengset J. (2022), *Rust for Rustaceans,* No Starch Press.
- Khan M. (2023), *Rust for C++ Programmers,* BPBOnline.
- Kolodin D. (2019), *Hands-on Microservices with Rust,* Packt.
- Lyu S. (2020), *Practical Rust Projects,* Apress.
- Lyu S. (2021), *Practical Rust Web Projects,* Apress.
- Matzinger C. (2019a), *Rust Programming Cookbook,* Packt.
- Matzinger C. (2019b), *Hands-on Data Structures and Algorithms with Rust,* Packt.
- Matzinger C. (2022), *Learn Rust Programming,* Packt.

BOOKS (CONT.)

- McNamara T. S. (2021), *Rust in Action,* Manning.
- Mesier R. (2021), *Beginning Rust Programming,* Wiley.
- Milanesi C. (2018), *Beginning Rust,* Apress.
- Rufus S. (2021), *Rust Programming,* 3rd Ed., NLN.
- Rustucean Team (2021), *Practical Rust 1.x Cookbook,* GitforGits.
- Rustucean Team (2023), *Rust in Practice,* GitforGits.
- Snoyman M. & Snoyman M. (), *Begin Rust,* BR.
- Wolverson H. (2021), *Hands-on-Rust,* The Pragmatic Programmer.
- Wolverson H. (2022), *Rust Brain Teasers,* The Pragmatic Programmer.
- Xu J. (2021), *Practical GPU Graphics with wgpu and Rust,* UniCAD.

LINKS (CONT.)

- Klabnic S. & Nichols C. (2021), *The Rust Programming Language*, 2nd Ed., online.
- The Rust Reference
- Rust by Example
- A Gentle Introduction to Rust
- Rustlings
- Advent of Code challenges in Rust
- Rust for Java Developers, blog
- Design patterns in Rust
- Rust design patterns

Questions