PROGRAMMING CONCEPTS AND LANGUAGES, 2024W

RUST Programming Language: Part I

University of Vienna

CONTENTS

The RUST Programming Language

Basics

- Hello World
- Functions, values, variables, pointers, references, and variable bindings

Rust Ownership System

- Allocation (Stack and Heap)
- Binding Scopes
- Ownership, Borrowing, Lifetimes

Literature

- https://doc.rust-lang.org, https://github.com/rust-lang/rustlings/, https://doc.rust-lang.org/rust-by-example/
- https://play.rust-lang.org/

RUST PROGRAMMING LANGUAGE

Originated in 2006, sponsored by Mozilla in 2009

- First stable release (v1.0) in 2015
- Mozilla → used in "Servo" (concurrent HTML engine) and parts of Firefox
- Increasing popularity: https://survey.stackoverflow.co/2024/technology#admired-and-desired

Open Source, systems programming language

- **Safe** and **efficient** use of available (underlying) resources
- Emphasis on control over the performance and resource consumption of programs and libraries
- Like C and C++, while still being memory safe by default, thus eliminating entire classes of common bugs
- Rich ecosystem of third-party tools and libraries
- Built-in tools for building, testing, documenting, and sharing code

Rich abstraction features

- Allow developers to encode many of the invariants of their program into code
- The code is then checked by the compiler instead of relying on convention or documentation

Rust in production: https://www.rust-lang.org/production

RUST :: HIGHLIGHTS

Zero-cost abstractions

- Minimal to no performance costs when using high-level abstractions, like iterations, interfaces, and functional programming
- The abstractions perform as well as if you wrote the underlying code by hand
  - Abstractions like data structures, control structures, generics, etc..
  - Many compile-time optimizations
  - Main cost: learning curve

Type safety

- Static typing → Rust must know the types of all variables at compile time
- Via explicit annotations, or by letting compiler infer the data type from the context
- The compiler assures that no operation will be applied to a variable of a wrong type

Memory safety

- Rust pointers (known as references) always refer to valid memory

RUST :: HIGHLIGHTS

Data race free

- Rust's *borrow checker* guarantees thread-safety by ensuring that multiple parts of a program can't mutate the same value at the same time.

Runtime Efficiency

- The language also has no garbage collector to manage memory efficiently
- In this way, Rust is most similar to languages like C and C++
- Rust can target embedded and "bare metal" programming, making it suitable to write an operating system kernel or device drivers
- Full control over the memory layout

RUST :: SPECIFIC FEATURES

Mutability

Rust Ownership System

- Set of rules that govern memory management
- Provides memory safety, and prevents data races
- Rules about **ownership** of values
- Rules of **borrowing**, i.e., accessing data without taking ownership
- Handling **lifetime** aspects of borrowing and ensuring that references are valid as long as needed

Basics

RUST :: HELLO WORLD

Function name

fn keyword

println! macro

```
fn main() {

    println!("Hello World from Rust");

}
```

Functions (named blocks of code) **are declared with `fn` keyword**

- Main function → starting point of every Rust program

Print to standard output

- `println!` and `print!` are macros
- `println!` is same as the `print!` but adds a newline at the end

Try it out on: https://play.rust-lang.org/

RUST :: FUNCTIONS

A function is a block of code that does a specific task

- The **function body** is defined inside curly brackets `{}`, input parameters are listed inside the parentheses `()`

```
fn my_fn(a: i32, b: i32) {
 // …
}
```

Every Rust program must have one function named `main`, which is the first code to run

- We can call other functions from within the main function, or from within other functions
- Rust functions returns exactly one value, declared with arrow (`->`)

```
fn my_fn(a: i32, b: i32) -> i32 {
   let a = 42;
   // ...
   a // returns a
}
```

- Code statements end with a semicolon `;`
- The return value can be specified with the `return` keyword or by omitting the semicolon

Read more: https://doc.rust-lang.org/book/ch03-03-how-functions-work.html

RUST :: VARIABLES

In Rust, a *variable* is declared with the keyword `let`

- In other words, we introduce a ***variable binding***

Immutable by default

- After a value is bound to a name, you can't change that value

Scope of a variable is defined by the block of code in which is declared

Shadowing is allowed

- A variable can be re-declared in the same scope with the same name

Read more: https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html, https://doc.rust-lang.org/beta/reference/variables.html

RUST :: BINDINGS

Keyword `let` is used to introduce a binding (declare a variable)

```
let x = 42;
```

Left-hand side is a pattern (more about this later)

$$\texttt{let } \underbrace{\texttt{(x, y)}}_{\textit{pattern}} \texttt{ = } \underbrace{\texttt{(5, 6)}}_{\textit{value}}\texttt{;}$$

Bindings can be type-annotated

```
let x: i32 = 42;
```

- Rust compiler can **often** infer the type from the context

Must be initialized to be used

```
let x: i32 = 42;
println!("The value of x is: {}", x);
```

Read more: https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html, https://doc.rust-lang.org/beta/reference/variables.html

RUST :: TERMINOLOGY

Place

- A location (in memory) that can hold a *value*

Variable

- A component of a stack frame (more about this later)
- A named function parameter, an anonymous temporary, or a named local variable*
- Immutable by default → cannot be changed after we set the initial *value*

Pointer

- A value that holds the address of a region of memory
- A pointer points to a *place*
- The same pointer can be stored in more than one variable
- `let x_ptr = &x;`

Reference

- A pointer that is assumed to be aligned, not null, and pointing to memory containing a valid value
- Represents a *borrow* of some owned value (more about borrow later)
- Accomplished with ”&” symbol or the `ref` keyword

*https://doc.rust-lang.org/beta/reference/variables.html

RUST :: HELLO WORLD WITH VARIABLES

```
fn main() {
   let x = 42; // same as let x: i32 = 42;
   println!("The answer is: ", x);
}
```

Output: The answer is: 42

Variable x has to be initialized if used

- If it is uninitialized but used then compiler gives an error

```
fn main() {
   let x: i32; // uninitialized
   println!("The answer is: ", x);
}
```

```
error: use of possibly uninitialized variable: `x`
println!("The answer is: {}", x);
                              ^
```

RUST :: HELLO WORLD WITH VARIABLES

```
fn main() {
   let x = 42;  // same as let x: i32 = 42;
   let _y: i32; // uninitialized but not used
   println!("The answer is: ", x);
}
```

Output: The answer is: 42

Variable x has to be initialized if used

- If it is uninitialized but used then compiler gives an error
- If it is uninitialized but not used then compiler gives a warning
  - You can tell compiler to ignore unused variable by prepending it with an underscore ("_y”)

Try it out on: https://play.rust-lang.org/

BINDINGS :: SCOPE

Range within a program for which an item is valid

- Global scope → accessible throughout the entire program
- Local scope → accessible only within a particular function or block of code

Variable bindings live in the *block* they were defined in

- A block is a collection of statements enclosed by "**{**" and "**}**" (e.g., a function definition is also a block)
- When a variable goes out of scope it is *dropped*

outer scope

Inner scope

```
fn main() {
    let x: i32 = 42;
    {
        let y: i32 = 43; // y valid from this point on
        println!("The value of x is {} and value of y is {}", x, y);
    } // this scope is over, and y is no longer valid and y is dropped

    println!("The value of x is {} and value of y is {}", x, y); // Does not compile!
}
```

BASIC TYPES :: OVERVIEW

Booleans (`bool`)

```
let x: bool = true;
```

Numeric

default in red

- Signed-integers (`i8, i16, i32, i64`)
- Unsigned-integers (`u8, u16, u32, u64`)
- Floating-point types (`f32, f64`)
- Architecture-dependent integer types
  - Variable-sized type
  - `usize` – unsigned int of the same number of bits as the platform's pointer type
    - Guaranteed to be big enough to address any pointer or any offset in a data structure (location in memory)
  - `isize` – signed int of the same number of bits as the platform's pointer type
    - The theoretical upper bound on object and array size is the maximum `isize` value
  - E.g., 64-bit architecture → 64-bit (8 bytes) sizes for `isize` and `usize`

Textual

- Char type (`char`)
- String types (more on this later)

BASIC TYPES :: EXAMPLES

```
fn main() {
   let x: i32 = 42;
   let mut y: u32 = 43;
   y = x; // Does not compile!

   let z = 45; // Type of i32 (default)

   let u: u16 = 42_u8 as u16; // Convert an integer type to another integer type
   println!("u8 max is {}", u8::MAX); // Will print 255

   let u: u8 = 256_u8; // Does not compile! (error: literal out of range for `u8`)
   let f1 = 1_000.000_1; // f64 default floating type
   let f2: f32 = 0.12;
   let f3: f64 = 0.01_f64;

   assert!(0.1+0.2==0.3); // panic at runtime

   let ch1: char = 'a’;
   let ch2: char = '🙃'; // UTF-8 support
   println!("ch is {}, ch2 is {}", ch1, ch2);

}
```

Note that even integer types must match

RUST :: MUTABILITY BASICS

Mutability → *the ability to change*

(Variable) **Bindings are immutable by default**

- For safety reasons (compiler will let you know if you changed something that you did not intend to change)
- Can be made mutable with the "`mut`" keyword

```
let x = 42; // x is immutable
x = 43; // Does not compile!
```

Compiler output:

```
error: re-assignment of immutable
variable `x` x = 43; ^~~~~~~
```

Mutability must be handled explicitly

- When a binding is *mutable*, it means you're allowed to change what the binding points to

```
let mut x = 42; // mut x: i32 → x is now mutable!
x = 43; // now valid!
```

The binding changed from one `i32` to another!

RUST :: BINDINGS :: MUTABILITY :: REFERENCES

References can also be mutable

```
let mut x = 42; // x is mutable

let y = &mut x;     // immutable binding to a mutable reference

let mut z = &mut x; // mutable binding to a mutable reference
```

- Here `y` is an immutable binding to a mutable reference
- `*y` can be used to bind x to something else, e.g., `*y = 43;`
- `z` is mutable, so you can also change what `z` is referencing, e.g., `*z = 43;`

Rust Ownership System

RUST :: OWNERSHIP

Set of rules that govern memory management enforced at compile time

- Prevents memory safety issues such as
  - Dangling pointers
  - Trying to free memory that has already been freed (double-free)
  - Memory leaks (not freeing memory that should been freed)

Three rules of ownership in rust:

1. Each value in Rust has an owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the value will be *dropped*

The ***owner*** of a value is the variable or data structure that holds it and is responsible for allocating and freeing the memory used to store that data

Read more: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

RUST :: ALLOCATION, STACK AND HEAP

Stack Memory

- Last-in, first-out (stack stores values in order it gets them, and removes them in the opposite order)
- The last item *pushed* to the stack will be the first thing *popped* from the stack
- All data must have known, fixed-size (e.g., integers, floats, chars, booleans)
- Faster than heap → location for new data is always at the top of the stack
- Types of unknown size are allocated to the heap, and a pointer to that value is pushed to the stack
- Impact on the developer's mental model

Heap Memory

- Data of unknown size (e.g., string, vector, etc.)
- Allocation on heap returns a pointer to that data
  - The memory allocator needs to find a place in memory that is big enough
- Both allocation and access slower than stack
  - Accessing data is slower because a pointer needs to be followed to get to the data

Read more: https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/the-stack-and-the-heap.html

RUST :: STACK EXAMPLE

```rust
fn main() {
    let x = 42;
}
```

*Actual address
in memory is not 0

addr	name	value
0	x	42

stack frame (simplified)

Local variables and function parameters have to be allocated when a funcion has been called

- (also some other data that we ignore for the purpose of this example)
- This is called a *Stack Frame*

Here we have only one variable binding

- `let x = 42;`
- `x` is `i32` type (default), `i32` is fixed-size in memory and is allocated on the stack
- When `main()` is over, its stack frame is deallocated

RUST :: STACK EXAMPLE

```
fn foo() {
  let a = 21;
  let b = 4;
}

fn main() {
  let x = 42;
  foo();
}
```

	addr	name	value
foo()'s stack frame	2	b	4
	1	a	21
main()'s stack frame	0	x	42

When `foo()` is called a new stack frame is allocated

- All local variables are, again, fixed-size and are allocated on the stack
- Since address 0 is used for the `main()`'s stack frame, 1 and 2 are used for `foo()`'s stack frame
- When `foo()`'s is over its frame is deallocated, and afterwards the `main()`'s stack frame goes away

RUST :: HEAP EXAMPLE

```
fn main() {
   let x = 42;                          // i32 on the stack
   let y = Box::new(5);                 // i32 on the heap
   let str1 = String::from("Marvin"); // (next slide)
}
```

addr	name	value
2	str1	???
1	y	???
0	x	42

In rust you allocate on the heap with Box<T>

- <T> represents the use of a generic type T
- We use a generic type declaration when we don't yet know the actual data type
- Actual value of y is a structure with a pointer to the heap
- The value of y could outlive the lifetime of the function
  - However, here it does not - when it goes out of scope a Drop is called (more about this later)

Note that you can check the actual address, like this:

- `println!("The memory of y is {:p}", y);` (Possible output: The memory of y is 0x7f8931705f20)
- Note that y is of &i32 type;

RUST :: STRING TYPE

Dynamically-sized type

- String size can change at runtime
- Stored on the stack with a pointer to the heap
- The value of String is stored on the heap

```
let str1 = String::from("Marvin");

println!("The name is {}", str1);
```

str1 (stack)

name	value
ptr	→
len	6
capacity	6

heap

index	value
0	M
1	a
2	r
3	v
4	i
5	n

`str1` → **ptr to data that is stored on the heap**

- Size of str1 is 3*`std::mem::size_of::<usize>()` → 3*8 = 24 bytes

`len` → **data size in bytes**

`capacity` → **total amount of memory allocated**

RUST :: OWNERSHIP :: COPY OR MOVE

Copy

- Scalar values with fixed sizes
- That lives on the stack
- Copy is cheap

Move

- Dynamically sized data
- That lives on the heap
- Copy would be too expensive

Example

```
let x: i32 = 42;
let y: i32 = x; // Copy or move?

println!("The values of x and y are {}, {}", x, y);
```

- The value of x is copied into y, and both variables are usable

RUST :: OWNERSHIP :: STRING TYPE

```
1: fn main() {
2:   let str1 = String::from("Marvin");
3:   let str2 = str1; // Copy or move?
4:   println!("The values of str1 and str2 are {}, {}", str1, str2); // Does not compile!
5: }
```

Does this work?

- It produces a compilation error!
- `str1` is dropped at line 3
  - str1 cannot be used anymore
  - Different from e.g., i32 due to a copy
- `str2` is the new owner of the value
  - "str1 was moved into str2"

str1 (stack)

name	value
ptr	
len	6
capacity	6

str2 (stack)

name	value
ptr	
len	6
capacity	6

heap

index	value
0	M
1	a
2	r
3	v
4	i
5	n

RUST :: OWNERSHIP :: COPY OR MOVE

```
fn main() {
  let str1 = String::from("Marvin");
  let str2 = str1; // Copy or move?
  println!("The values of str1 and str2 are {}, {}", str1, str2); // Does not compile!
}
```

Output:

```
5 |     println!("The values of str1 and str2 are {}, {}", str1, str2); // Does not compile!
  |                                                        ^^^^ value borrowed here after move
```

`str1` is just a pointer, which will get copied into `str2`

- Data on the heap will not be copied!
- `str1` will be dropped after assigning it to `str2` to avoid dangling pointers
  - Otherwise, the second rule of ownership in Rust would be violated, as there can only be ONE owner at the time
- You can explicitly do a deep copy with `str1.clone()` (→ next slide)

RUST :: OWNERSHIP :: COPY OR MOVE

We can explicitly do a deep copy with `str1.clone()`

```rust
let str1 = String::from("Marvin");
let str2 = str1.clone();
println!("The values of str1 and str2 are {}, {}", str1, str2); // Compiles fine
```

str1 (stack)

name	value
ptr	
len	6
capacity	6

heap

index	value
0	M
1	a
2	r
3	v
4	i
5	n

str2 (stack)

name	value
ptr	
len	6
capacity	6

heap

index	value
0	M
1	a
2	r
3	v
4	i
5	n

RUST :: OWNERSHIP :: EXAMPLE

```
fn foo() -> String {
   let s = String::from("Marvin");
   s

}

fn main() {
   let str1 = foo();
   println!("{}", str1);
}
```

Does this work?

RUST :: OWNERSHIP :: EXAMPLE

```
fn foo() -> String {
   let s = String::from("Marvin"); // s comes into scope
   s                               // foo() will move its return value into
                                   // the function that calls it
}

fn main() {
   let str1 = foo();               // foo moves its return value into str1
   println!("{}", str1);
}
```

Does this work?

- It does
- Note: the ownership of s is moved from foo() into str1 in the main() function

RUST :: OWNERSHIP :: EXAMPLE

```
fn foo(input_str: String) {              // 3. input_str comes into scope
   println!("{}", input_str);            // 4. string is printed to standard output
}                                        // 5. input_str goes out of scope

fn main() {
   let str1 = String::from("Marvin"); // 1. str1 comes into scope
   foo(str1);                            // 2. str1 moved into foo, no longer valid in this scope
}
```

Does this work?

RUST :: OWNERSHIP :: EXAMPLE

```
fn foo(input_str: String) {              // 3. input_str comes into scope
   println!("{}", input_str);            // 4. string is printed to standard output
}                                        // 5. input_str goes out of scope

fn main() {
   let str1 = String::from("Marvin"); // 1. str1 comes into scope
   foo(str1);                            // 2. str1 moved into foo, no longer valid in this scope
}
```

Does this work?

- It does
- Note: the ownership of `str1` moved into `foo()`

RUST :: OWNERSHIP :: EXAMPLE

```
fn foo(input_str: String) {              // input_str comes into scope
   println!("{}", input_str);            // string is printed to standard output
}                                        // input_str goes out of scope

fn main() {
   let str1 = String::from("Marvin"); // str1 comes into scope
   foo(str1);                            // str1 moved into foo, no longer valid in this scope

   println!("{}", str1);
}
```

Does this work?

RUST :: OWNERSHIP :: EXAMPLE

```
fn foo(input_str: String) {              // input_str comes into scope
   println!("{}", input_str);            // string is printed to standard output
}                                        // input_str goes out of scope

fn main() {
   let str1 = String::from("Marvin"); // str1 comes into scope
   foo(str1);                            // str1 moved into foo, no longer valid in this scope

   println!("{}", str1);                 // Does not compile!
}
```

Does this work?

- It does not
- Note: the ownership `str1` moved into `foo()` – Why is this important?
- You could solve it by making a deep copy (`str1.clone()`)
- Or you could maybe borrow ownership to function foo?

RUST :: OWNERSHIP :: BORROWING

Borrowing

- Way of temporarily accessing data without taking ownership of it
- Accomplished with references: when borrowing you are taking a **reference** (pointer) to the data, not the data itself
- A binding that borrows something does not deallocate the resource when it goes out of scope

Two kinds: Immutable and Mutable

- Immutable by default, but can be made mutable with the `mut` keyword

Rules

1. Any borrow must last for a scope no greater than that of the owner
2. You may have one or the other of these two kinds of borrows, but not both at the same time:
   - **One or more references (`&T`) to a resource**
   - **Exactly one mutable reference (`&mut T`)**

This leads to *data race* free programs

- What are data races?
- There is a 'data race' when two or more pointers access the same memory location at the same time, where at least one of them is writing, and the operations are not synchronized

Read more: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html, https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/references-and-borrowing.html

RUST :: OWNERSHIP :: BORROWING

```
fn foo(z: &i32) {
   let z = 21;
}

fn main() {
   let x = 42;
   let y = &x;

   foo(y);
}
```

	addr	name	value
foo()'s stack frame	3	z	21
	2	z	→ 0
main()'s stack frame	1	y	→ 0
	0	x	42

Does this work?

- It does
- Note: the ownership x is borrowed by foo(), at the end of the foo()'s scope, the ownership is returned

RUST :: OWNERSHIP :: BORROWING :: EXAMPLES

```
fn main() {
   let str1 = String::from("Marvin");
   let r1 = &str1;
   let r2 = &str1;

   println!("{}, {}, {}", str1, r1, r2);
}
```

Does this work?

```
fn main() {
   let str1 = String::from("Marvin");
   let r1 = &str1;
   let r2 = &mut str1;

   println!("{}, {}, {}", str1, r1, r2);
}
```

Does this work?

Rules:

1. **One or more references (`&T`) to a resource**
2. **Exactly one mutable reference (`&mut T`)**

RUST :: OWNERSHIP :: BORROWING :: EXAMPLES

```
fn main() {
    let str1 = String::from("Marvin");
    let r1 = &str1;
    let r2 = &str1;

    println!("{}, {}, {}", str1, r1, r2);
}
```

Does this work?

- Yes, it does!

```
fn main() {
    let str1 = String::from("Marvin");
    let r1 = &str1;
    let r2 = &mut str1;

    println!("{}, {}, {}", str1, r1, r2);
}
```

Does this work?

- No, it does not!
- → violates the rules

Rules:

1. **One or more references (`&T`) to a resource**
2. **Exactly one mutable reference (`&mut T`)**

RUST :: OWNERSHIP :: BORROWING :: EXAMPLES

```
fn main() {
  let mut str1 = String::from("Marvin"); // str1 comes into scope
  {
     let r1 = &str1;  // ownership borrowed to r1
                      // you can use immutable reference here, but not other way
  }                   // r1 goes out of scope, ownership returned
  let r2 = &mut str1;
}
```

```
fn main() {
  let str1 = String::from("Marvin"); // str1 comes into scope
  {
     let r1 = &mut str1;  // Does not compile!
  }
}
```

RUST :: OWNERSHIP :: BORROWING :: EXAMPLES

```
fn main() {
  let mut str1 = String::from("Marvin"); // str1 comes into scope
  {
     let r1 = &str1;  // ownership borrowed to r1
                      // you can use immutable reference here, but not other way
  }                   // r1 goes out of scope, ownership returned
  let r2 = &mut str1;
}
```

Does this work?

- It does, since `r1` goes out of scope, and we have only 1 mutable reference to the same data at a time

```
fn main() {
  let str1 = String::from("Marvin"); // str1 comes into scope
  {
     let r1 = &mut str1;  // Does not compile!
  }
}
```

This does not work → `str1` is not mutable

RUST :: OWNERSHIP :: BORROWING :: EXAMPLE

```rust
fn foo() -> &String {
    let s = String::from("Marvin");
    &s
}

fn main() {
    let str1 = foo();
    println!("{}", str);
}
```

```rust
fn main() {
    let y: &i32;
    {
        let x = 5;
        y = &x;
    }

    println!("{}", y);
}
```

Does this work?

Output:

```
1 | fn foo() -> &String {
  |             ^ expected named lifetime parameter
  = help: this function's return type contains
a borrowed value, but there is no value for it to be
borrowed from
help: consider using the `'static` lifetime
```

```
4 |         let x = 5;
  |             - binding `x` declared here
5 |         y = &x;
  |             ^^ borrowed value does not live long enough
6 |     }
  |     - `x` dropped here while still borrowed
7 |
8 |     println!("{}", y);
  |                    - borrow later used here
```

RUST :: OWNERSHIP :: BORROWING :: EXAMPLE

```rust
fn foo() -> &String {
    let s = String::from("Marvin");
    &s
}

fn main() {
    let str1 = foo();
    println!("{}", str);
}
```

```rust
fn main() {
    let y: &i32;
    {
        let x = 5;
        y = &x;
    }

    println!("{}", y);
}
```

Does not compile! → violattion of the second rule, i.e., references must be valid

Output:

```
1 | fn foo() -> &String {
  |             ^ expected named lifetime parameter
  = help: this function's return type contains
a borrowed value, but there is no value for it to be
borrowed from
help: consider using the `'static` lifetime
```

```
4 |         let x = 5;
  |             - binding `x` declared here
5 |         y = &x;
  |             ^^ borrowed value does not live long enough
6 |     }
  |     - `x` dropped here while still borrowed
7 |
8 |     println!("{}", y);
  |                    - borrow later used here
```

RUST :: OWNERSHIP :: BORROWING :: EXAMPLES

```
fn foo(input_str: &String) {          // input_str comes into scope
   println!("{}", input_str);         // string is printed to standard output
}                                     // input_str goes out of scope

fn main() {
   let str1 = String::from("Marvin"); // str1 comes into scope
   foo(&str1);                        // str1 borrowed

   println!("{}", str1);              // valid!
}
```

Does this work?

RUST :: OWNERSHIP :: BORROWING :: EXAMPLES

```
fn foo(input_str: &String) {          // input_str comes into scope
   println!("{}", input_str);         // string is printed to standard output
}                                     // input_str goes out of scope

fn main() {
   let str1 = String::from("Marvin"); // str1 comes into scope
   foo(&str1);                        // str1 borrowed

   println!("{}", str1);              // valid!
}
```

Does this work?

- It does
- Note: the ownership `str1` is borrowed by `foo()`, at the end of the `foo()`'s scope, the ownership is returned

RUST :: LIFETIMES

Every reference has a *lifetime* associated with it (the scope for which a reference is valid)

- Used by compiler to ensure that all borrows are valid
- Most of the time implicit and inferred, but can be explicitly annotated (if compiler cannot infer it)
- Different from the scope

Example

```
fn main() {
   let r;                // Introduce reference: r
   {
      let x = 42;        // Introduce scoped value: x
      r = &x;            // Store reference of x in r
   }                     // x goes out of scope and is dropped
   println!("{}", r); // r still refers to x
}
```

Will produce an error:

^^ borrowed value does not live long enough

RUST :: LIFETIMES :: BORROW CHECKER

Compares scopes to determine if all borrows are valid

- Key part of the Rust’s ownership system
- Tracks lifetimes of references and ensures that they do not violate the ownership rules

Ensures that

- A value is not accessed after it has been moved or freed from memory
- A reference to a value must never outlive the value itself

Explicit lifetime annotations can be provided to borrow checker

- Most of the time not needed
- Example

```
// One input reference with lifetime 'a which must live at least as long as the function
fn foo<'a>(x: &'a i32) {
   println!("x is {}", x);
}
```

Source, read more: https://doc.rust-lang.org/rust-by-example/scope/lifetime/fn.html

RUST :: GETTING STARTED :: RUSTUP & CARGO

Install Rust

- https://rustup.rs
- Once installed you should have 3 new commands available: **`rustc`**, **`rustdoc`**, **`cargo`**

Cargo

- Compilation and package manager
- Used as a tool to create a new project, build and run Rust programs and manage external libraries

`rustc`

- Rust compiler, typically used via Cargo

`rustdoc`

- Rust documentation tool that generates documentation from comments, also typically used via Cargo

RUST :: MODULE SYSTEM

Crates

- A Rust crate is a compilation unit - the smallest piece of code the Rust compiler can run
- A crate contains a hierarchy of Rust modules with an implicit, unnamed top-level module.
- The code in a crate is compiled together to create a binary executable or a library
- Only crates are compiled as reusable units.

Modules

- Organize your program by managing the scope of the individual code items inside a crate
- Related code items or items that are used together can be grouped into the same module
- Recursive code definitions can span other modules.

Paths

- You can use paths to name items in your code and/or hide implementation details
- You can specify the parts of your code that are accessible publicly versus parts that are private

RUST CRATES AND LIBRARIES :: STANDARD LIBRARY

std → the Rust standard library

std::collections

- Definitions for collection types, such as HashMap.

std::env

- Functions for working with your environment.

std::fmt

- Functionality to control output format.

std::fs

- Functions for working with the file system.

std::io

- Definitions and functionality for working with input/output.

std::path

- Definitions and functions that support working with file system path data.

RUST CRATES AND LIBRARIES :: 3RD PARTY LIBS

chrono

- A third-party crate to handle date and time data

regex

- A third-party crate to work with regular expressions

serde

- A third-party crate of serialization and deserialization operations for Rust data structures

structopt

- A third-party crate for easily parsing command-line arguments.

RESOURCES, REFERENCES AND LINKS

1. **Resources at rust-lang.org**
    1. The Book: https://doc.rust-lang.org/book/
    2. Rust by example: https://doc.rust-lang.org/rust-by-example/
    3. Course: https://github.com/rust-lang/rustlings/
2. **The Rust Reference: https://doc.rust-lang.org/beta/reference/**
3. **https://learn.microsoft.com/en-us/training/paths/rust-first-steps/**
4. **https://dhghomon.github.io/easy_rust/Chapter_12.html**
5. **https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/primitive-types.html**
6. **https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/unsized-types.html**