Introduction to the Rust programming Language

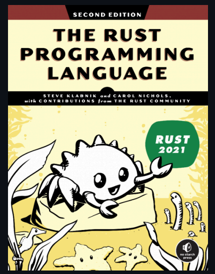
Following along The Rust Book from the official source

by: **Simon Lalonde**

For: **IFT-769** (Theoritical concepts CS)

📖 **Project overview -** Going through "The Rust Programming Language"

The Rust Programming Language by Steve Klabnik and Carol Nichols



Book overview:

- Official guide to the Rust programming language
- Covers the basics (syntax, types, functions) + toolchain
- Advanced and Rust-specific features:
  - Ownership, borrowing, lifetimes
  - Unique error handling
  - Concurrency

Klabnik, Steve, and Carol Nichols. The Rust Programming Language. 2nd ed., No Starch Press.

📚 **Theoretical concepts** - Key topics covered

1. Common Programming Concepts (*variables, types, control flow*)
2. Understanding Ownership (*memory management*)
3. Structs, Enums and Pattern Matching
4. Containers/Collections
5. Error Handling
6. Generics, Traits and Lifetimes
7. OO features (⚠️ *little covered in this presentation*)
8. Smart pointers and Concurrency (🛑 *Not covered in this presentation*)

Rust Overview

Written by **Graydon Hoare in 2006**, Rust is a systems programming language focused on safety, speed, and concurrency. Backed by **Mozilla** and now the Rust Foundation.

Currently known projects

- Now inside some of the Linux kernel
- Used in components of Firefox browser

Predicted use cases

- Replacement for C/C++ in systems programming and embedded systems
- WebAssembly, ML/AI, and other performance-critical applications

Rust Programming Language. 2024. Official Website

Lin Clark. The whole web at maximum FPS: How WebRender gets rid of jank. 2017. Mozilla Blog

Clive Thompson. How Rust went from a side project to the world's most-loved programming language. 2023. TechReview

📝 Pros and Cons of Rust

PROS:

- **Memory safety:** No null pointers, dangling pointers, or buffer overflows
- **Error handling**: With the `Result` and `Option` types
- **Concurrency**: Safe and efficient with the ownership system
- **Performance**: Comparable to C/C++ with zero-cost abstractions
- **Ecosystem**: Growing with a strong community and package manager (**Cargo**)
- **Helpful compiler**: Provides detailed error messages and warnings

CONS:

- **Learning curve**: Ownership, borrowing, and lifetimes can be challenging
- **Tooling and prevalence**: Not as mature as other languages (C/C++, Python, etc.)
- **Syntax**: Can be verbose and complex compared to other languages

⚙️ Installation and setup

<u>Installation</u>:

1. Install Rust using `rustup` (Rust toolchain installer)

<u>Included toolchain</u>:

- `rustc`: Rust compiler
- `rustup`: Rust toolchain manager
- `rustfmt`: Rust code formatter
- `cargo`: Rust package manager and build tool

Package and library management

- **Crates** are Rust packages that can be shared and reused
- Managed with **Cargo**, the Rust package manager

Development environment - Toolchain overview

Env setup and features:

- Easy install: `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`
- Rustup for managing toolchains: `rustup update`
- Included formatter: `rustfmt --check src/main.rs` (dry-run mode)
- Cargo for building and managing projects: `cargo new project_name`
- Quality of life with `rust-analyzer`: LSP, build/debug IDE support etc.

Development environment - Cargo features

Useful Cargo commands when building a project:

- `cargo build` or `cargo run` to compile and run the project. Use `--release` flag for compilation with optimizations inside `target/release/`
- `cargo check`: Check the project for errors without building
- `cargo doc`: Generate documentation for the project
- `cargo clean`: Remove build artifacts
- `cargo update`: Update dependencies
- `cargo fmt`: Format the code according to the Rust style guidelines
- `cargo test`: Run tests in the project

🏗️ Basic programming features

Overview of common programming concepts in Rust

./projects/guessing_game

📝 Variables and mutability

Variables are immutable by default

```rust
let x = 5;              // immutable variable
x = 10;                 //!ERROR!

let mut y = 10;         // mutable variable
y = 15;                 // Will compile (overwrite with same type)
let y = "hello";        // Will compile (shadowing)
```

Constants are always immutable within the scope

```rust
const MAX_POINTS: u32 = 100_000;
```

🧮 Statically typed + type inference

`rust-analyzer` IDE Frontend provides type hints and suggestions

```
let secret_num = rand::thread_rng().gen_range(1..101); // Will infer i32 type
```

Explicit type annotations can or must be used

```
let mut num: String = String::new(); // Can be annotated or inferred
num = "42".to_string();

let guess = guess.trim().parse().expect("Please enter a number");  // Wont Compile
let guess: u32 = guess.trim().parse().expect("Please enter a number"); // Will compile
```

🧮 Data types - Scalars

Data type	Size	Specifity
int	8-128 bits	signed/unsigned
float	32/64 bits	simple/double precision
char	4 bytes	unicode
bool	1 byte	true/false

🧮 Data types - Compound

Data type	Size	Elements	Example	Access
tuple	fixed	mixed types	`(1, "hello", 3.14)`	`tuple.0`
array	fixed	same type	`[1, 2, 3, 4, 5]`	`array[0]`
vec	dynamic	same type	`vec![1, 2, 3, 4, 5]`	`vec[0]`

Access safety with runtime bounds checking. If using `array[10]` will panic at runtime instead of *undefined behavior like in C/C++*

⚙️ Functions - `main`

Functions are defined with the `fn` keyword. All programs start with a `main` function

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

    say_hello_back();
}

fn say_hello_back() {
    println!("Hello back!");
}
```

⚙️ **Functions** - Parameters and return

Function signatures and use:

- Parameters must have type annotations
- Return type must be specified with `->`
- Functions can return multiple values with tuples

```rust
fn main() {
    let num_sum = add(5, 10);
    println!("The sum is: {}", num_sum);
}

fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

📜 Statements

- `let` is a statement, and `x + y` is an expression.
- Compared to C/C++, var assignment is an expression in Rust and **does not** return a value
- Statements must end with a semicolon `;`

```
let x = 5;            // statement
let y = z = 10;       // ERROR! z = 10 does not return a value
```

💡 Expressions

- Expressions **evaluate** to a value (*func calls, operations, blocks*)
- No `;` at the end of expressions
- **Blocks** `{}` are expressions and can be used to create new scopes + return values

```
fn main() {
    let x = 5; // whole line is statement, 5 is expression
    let y = {
        let x = 3;
        x + 1
    }; // an expression
    println!("The value of y is: {}", y); // Prints 4!
}
```

🧰 Control Flow - Conditionals

if/else: (Only takes boolean expressions)

```rust
// Classic if/else if/else
let mut condition = false;
if number < 5 {
    println!("Too small!");
} else if number > 5 {
    println!("Too big!");
} else {
    println!("Just right!");
    condition = true;
}

// Assignement with if/else
let result = if condition { 5 } else { 6 };
```

🧰 **Control Flow** - Loops overview

3 types of loops in Rust: `loop`, `while` and `for`

- `loop`: Infinite loop until `break` or return
- `while`: Loop while condition is true
- `for`: Loop over an iterator

```rust
// Conditional loop
let mut counter = 0;
while counter < 10 {
    println!("counter = {counter}");
    counter += 1;
}
```

🧰 Control Flow - Loop labels

Loop labels can be used to distinguish nested loops (*break* and *continue*). See ./projects/loop_labels

```
fn main() {
    let mut count = 0;
    'counting_up: loop { // Label the outer loop
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;   // Break the outer loop
            }
            remaining -= 1;
        }
        count += 1;
    }
    println!("End count = {count}");
}
```

🧰 **Control Flow** - Collection with `for`

No need for manual indexing, `for` loops iterate over collections

```rust
let collection = [10, 20, 30, 40, 50];
for element in collection {
    println!("The value is: {element}");
};
```

🔎 **Ranges**, use the `..` operator

```rust
for number in 1..4 {
    println!("The value is: {number}");
}
```

🛠️ **Practical project #0** - Guessing game

Great way to introduce to the development environment and basic concepts of Rust:

- Common programming concepts (types, funcs, control flow)
- Use of another crate (rand) inside the project
- I/O, String manipulation, error handling
- Compiler warnings and error messages
- `rust-analyzer` compiler FE for IDE support

Klabnik, Steve, and Carol Nichols. The Rust Programming Language. 2nd ed., No Starch Press.

🚀 Demo Time!

Simple guessing game CLI app 🎲 (Basics and dev environment features)

🔎 **Demo reminders** - P#0 (Guessing game)

- `Result` type with `.expect()` for error handling
- `cargo doc --open` to generate and view documentation
- `cargo fmt` to format the code
- Type annotations and `let` for variable declaration

🔑 **Ownership** - Overview

Ownership is a key feature of Rust regarding the management of stack (*static, compile-time known, LIFO*) and heap memory (*allocated at runtime, dynamic, FIFO*).

It ensures memory safety without garbage collection.

The 3 rules of ownership:

1. Each value in Rust has a variable that's its owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the value will be dropped

🔑 **Ownership** - String Type vs. literals

```rust
let s1: &str = "hello"; // string literal, immutable

{
    // s1 is still valid
    let mut s2 = String::from("hello"); // allocated on the heap
    s2.push_str(", world!"); // Mutable

} // calls drop(), s2 goes out of scope its memory is freed
```

- String literals hardcoded into binary. Immutable and fast.
- `String` type is allocated on the heap and is mutable. Memory freed when out of scope. Similar to smart pointers in C++.

🔑 **Ownership** - Move

```rust
// MOVE
let s1 = String::from("hello");
let s2 = s1; // s1 is moved to s2
println!("{s1}"); // ERROR! s1 is no longer valid

// DEEP COPY
let s3 = s2.clone(); // deep copy
println!("{s2}"); // s2 is still valid
```

No *double free* or *dangling pointers* with the **move** operation (first 3 lines of code).

🔑 **Ownership** - Copy

Types that implement the `Copy` trait are copied instead of moved. Stack-only data types (i.e. integers, booleans, char etc.) for speed and efficiency.

```
let x = 5;
let y = x; // x is copied to y

println!("{x}"); // x is still valid. Same as x.clone() but no needed
```

🔑 Taking ownership - Functions

```rust
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
    println!("{s} world!");         // ERROR! s is no longer valid

    let x = 5;                      // x comes into scope
    let y = makes_copy(x);          // x would move into the function, but i32 is Copy
    println!("{x}");                // OK! x us still valid
}

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{some_string}");
} // `some_string` goes out of scope, `drop` is called and memory is freed

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{some_integer}");
} // `some_integer` goes out of scope, nothing happens.
```

🔑 **Transfer Ownership** - Function return and scope

A bit tedious, but ownership can be transferred back to the calling function with the return value.

```rust
fn main() {
    let s1 = gives_ownership();         // `gives_ownership` moves its return val into s1

    let s2 = String::from("hello");     // s2 comes into scope

    let s3 = takes_and_gives_back(s2);  // s2 is moved into `takes_and_gives_back` becomes invalid
                                        // `takes_and_gives_back` returns a new String that into s3
}

fn gives_ownership() -> String {             // `gives_ownership` move return val into the
                                             // function that calls it

    let some_string = String::from("yours"); // some_string comes into scope

    some_string                              // some_string is returned moves out of calling func
}

// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into scope

    a_string  // a_string is returned and moves out to the calling function
}
```

🤝 **References and Borrowing** - Overview

Kind of like passing by reference in C/C++ but with some key differences:

- **References** are immutable by default
- **Borrowing** allows multiple references to the same data
- **Mutable references** are exclusive and have strict rules

References are created with the `&` symbol, and borrowing is done with `&mut` for mutable references (see next slide).

🤝 References and Borrowing - Simple borrowing example

```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{s1}' is {len}."); // s1 still valid because it is borrowed
}

fn calculate_length(s: &String) -> usize { // s is a reference to a String
    s.len()
} // s goes out of scope, but does not have ownership of what it refers to
```

🤝 Mutable references - General case

Borrowed references are not mutable by default. To allow mutation, use `&mut`

```rust
// let s = String::from("hello"); // WOULD NOT COMPILE!
let mut s = String::from("hello");
change(&mut s);

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

🤝 **Mutable references** - Data races safety

Compile time checks for mutable refs

⚠️ **NO** multiple mutable references to the same data

```
let mut s = String::from("hello");
let r1 = &mut s;
let r2 = &mut s;    // ERROR! r1 is still active
println!("{}, {}", r1, r2);
```

⚠️ **NO** mutable references while immutable references are active

```
let mut s = String::from("hello");
let r1 = &s;         // OK
let r2 = &s;         // OK
let r3 = &mut s;     // ERROR! r1 and r2 are still active
println!("{}, {}, {}", r1, r2, r3);
```

🤝 **Mutable references** - Data races safety (2/2)

🔎 Use of scopes to limit mutable references

```rust
let mut s = String::from("hello");
{
    let r1 = &mut s;
} // r1 goes out of scope, allowing a new mutable reference
let r2 = &mut s; // OK!
```

🔎 Reference's scope ends after the last usage of the reference.

```rust
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    println!("{r1} and {r2}");
    // variables r1 and r2 will not be used after this point

    let r3 = &mut s; // no problem because r1/r2 are no longer valid
    println!("{r3}");
```

⚠️ **Reference caution** - Fixing a state management problem

❗ Tedious or even problematic when working on a reference

```rust
let mut s = String::from("hello world");
let word_index = first_word(&s); // word_index will get the value 5

s.clear(); // empties the String, making it equal to ""
// `word_index` still has the value 5 here, but no more string tied because s is invalid
println!("the first word is: {s[..word_index]}"); // ERROR! s is empty
```

```rust
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }
    s.len()
}
// Imagine implementing second_word() and managing state...
```

String Slice Type - A kind of reference

Slices are references to a contiguous sequence of elements in a collection. They are a reference to a part of a string or array.

```rust
let s = String::from("hello world");
let hello = &s[0..5];   // same as &s[..5]. Excludes the last index
let world = &s[6..11];  // same as &s[6..]. Includes the first index
```

🔪 String Slice - Refactoring `first_word()`

```rust
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i]; // return a slice(ref) of the string
        }
    }
    &s[..]  // or return the slice of whole string
}
```

```rust
// Compiler assures that the slice is valid as long as the string is valid
fn main() {
    let mut s = String::from("hello world");
    let word = first_word(&s); // immutable borrow (return type is &str)

    s.clear(); // error! mutable borrow while immutable borrow is active
    println!("the first word is: {word}");
}
```

🔪 **Other Slice types** - Array example `first_word()`

Similar to strings, slices can be used with arrays

```rust
fn main() {
    let a = [1, 2, 3, 4, 5];
    let slice = &a[1..3]; // slice is of type &[i32]
    assert_eq!([2, 3], slice);
}
```

Useful for passing parts of arrays to functions without copying the data.

🏗️ **Structs** - Overview

Structs data structure encapsulate fields of specific types and methods (just like in C++/OO language).

- If declared mutable, the whole struct is mutable.
- dot notation for named field access
- Methods are defined within the `impl` block

```
let mut user1 = User {
    username: String::from("user1"),
    phone: 1234567890
    active: true,
};

user1.active = false;
```

🏗️ **Structs** - Shorthands

```
fn build_user(username: String, phone: u32) -> User {  // Returns a User struct
    User {
        username, // shorthand for username: username
        phone,    // shorthand for phone: phone
        active: true
    }
}

// Struct update syntax
let user2 = User {
    phone: 9876543210
    ..user1 // copy the rest of the fields from user1
}
```

🏗️ Tuple Structs

Tuple structs are similar, but don't have named fields. Useful for naming tuples and creating new types.

```rust
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    // black and origin are different types
    let red = Color(255, 0, 0);
    let origin = Point(0, 0, 0);
}
// Cannot pass a Point even though they have the same fields' types
fn make_paler(color: Color) -> Color {
    Color(color.0 / 2, color.1 / 2, color.2 / 2)
}
```

⚙️ **Methods** - Basic implementation

Methods are defined within the `impl` block.

```rust
struct SumArgs {
  n1: i32,
  n2: i32,
}
impl SumArgs {
  fn add_numbers(&self) -> i32 { // self is alias for Self (instead of args: &SumArgs)
    self.n1 + self.n2
  }
}
fn main() {
  let args = SumArgs { n1: 2, n2: 3 };
  let sum = args.add_numbers(); // Or SumArgs::add_numbers(&args)
  println!("{} + {} = {}", args.n1, args.n2, sum);
}
```

Gian Lorenzetto. Rust - Structs, Functions and Methods. 2021. Medium Post

⚙️ **Methods** - Mutability

Use `&self` for read-only and `&mut self` for methods that modify the struct.

```rust
struct Rectangle {
    width: u32,
    height: u32
}
impl Rectangle {
    fn area(&self) -> u32 {  // takes ownership of self (read-only)
        self.width * self.height
    }
    fn half_rect(&mut self) {  // borrows mutably
        self.width /= 2;
        self.height /= 2;
    }
    fn width(&self) -> bool {  // Getters in Rust
        self.width > 0
    }
}
let mut rect = Rectangle { width: 10, height: 20 };
println!("rect's width is valid: {} because width={}", rect.width(), rect.width);
```

⚙️ **Methods** - Automatic referencing/dereferencing

Unlike in C/C++, Rust automatically references and dereferences when calling methods (No `->` operator or `(*object).something()`)

```
p1.distance(&p2); // Both are the same, version1 is more readable
(&p1).distance(&p2);
```

With `object.something()`, Rust automatically adds in `&`, `&mut`, or `*` to match signature of the method.

🔎 It depends wether method is **reading (`&self`), writing (`&mut self`), or consuming (`self`)**

⚙️ **Methods** - Associated function

When a function is associated with a struct, it doesn't take `self` as a parameter.

- Often used for constructor
- Called with the `::` syntax

```rust
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
let square = Rectangle::square(10);
```

📋 **Enums** - Overview

- `Enums` are a way to define a type by enumerating its possible variants
- Each variant can have different data associated with it (*i.e.* `struct`, `String` ...)
- Namespaced under identifier, accessed with `Enum::variant` syntax
- Default constructor is `Enum::variant(data)`

```rust
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}
// Construct instances of each variant
let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

📋 **Enums** - Advantages over `struct`

Use of `impl` blocks for **common methods** that applies to **all variants**

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
} // Could be same as 4 different structs `struct Quit`, `struct Move{...}`

impl Message {
    fn send(&self) {
        // self ref to the variant instance
        println!("Sending message {:?}...", self);
    }
}
let m = Message::Write(String::from("Hello, world!"));
m.send();
```

❓ Option Enum - NULL free!

Rust has no `null` value, but uses the `Option` enum to represent the presence or absence of a value from standard library.

```
enum Option<T> { // Generic type T
    Some(T),     // Some value of type T
    None,
}

let x: i8 = 5;
let y: Option<i8> = Some(5); // Some value
let z: Option<i8> = None;  // No value

let sum = x + y;  // Won't compile because i8 + Option<i8> are different types
                  // and sum not implemented
```

With `Option`, the compiler forces you to handle the case where the value is `None`.

⚔️ **Match Expression** - Overview

`match` is a control flow operator that compares a value against a series of patterns and then executes code based on which pattern matches.

```rust
#[derive(Debug)] // to inspect the state inside match expr
enum UsState {Alabama, Alaska, //...}
enum Coin {Penny, Nickel, Dime, Quarter(UsState)}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {state:?}!");
            25
        } // Passing a Coin::Quarter(UsState::Alaska) will print "State
    }     // quarter from Alaska!" and return 25
}
```

⚔️ **Match Expression** - Matching with `Option<T>`

More powerful than `switch` in C/C++ because it can match on any type.

```rust
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);       // returns Some(6)
let none = plus_one(None);      // returns None
```

⚔️ **Match Expression** - Exhaustive matching and catch-all

Evaluation is in order. We can use `other` or `_` to catch all other cases.

```rust
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    other => move_player(other),     // if no param needed,
}                                    // use _ => paramless_func()

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn move_player(num_spaces: u8) {}
```

Powerful (type checking, Option, enums) and concise (no `if-else` chains).

🔄 Concise Control Flow

Syntax sugar for single match arms with `if` guards or single catch-all arm.

```rust
let mut count = 0;
// match version
match coin {
    Coin::Quarter(state) => println!("State quarter from {state:?}!"),
    _ => count += 1,
}
// if let version
if let Coin::Quarter(state) = coin {
    println!("State quarter from {state:?}!");
} else {
    count += 1;
}
```

📂 **Common Collections** - Overview

- Collections are data structures that can store multiple values
- Heap allocated
- Unknown size at compile time, but can grow or shrink at runtime

Discussed here:

1. **Vectors** - Dynamic array
2. **Strings** - UTF-8 encoded
3. **Hash Maps** - Key/Value pairs

📂 **Collections** - Vectors init and access

Vectors are dynamic arrays. Generic type of `Vec<T>`

```rust
// Initialization
let v: Vec<u8> = Vec::new();    // type required
let mut my_vec = vec![1, 2, 3];     // type inferred with `vec!` macro
my_vec.push(4);                     // Add an element
```

Accessing elements and bounds checking. Both yield a reference.

```rust
let third: &i32 = &my_vec[2];    // Panics! if out of bounds
println!("The third element is {third}");

let third: Option<&i32> = my_vec.get(2);    // Returns None if out of bounds
match third {
    Some(third) => println!("The third element is {third}"),
    None => println!("There is no third element."),
}
```

📂 **Collections** - Vectors' iteration and types

Iterate in read-only or mutable mode with `for` loop

```
for i in &my_vec {     // Readonly
    println!("{i}");
}
```

Store only similar types within same vec, but can use `enum` for different types

```
{
    enum CliArg {
        Int(i32),
        Text(String),
    }
    let mut arguments = vec![
        CliArg::Int(5),
        CliArg::Text(String::from("my_database_name")),
    ];
    arguments.push(CliArg::Text(String::from("my_table_name")));   // mutable with mixed types
} // <---- Out of scope, `arguments`'s memory is freed
```

Collections - Strings Overview

String `str` std vs `String` type:

- `str` is immutable, usually used as a slice/reference that can be borrowed
- `String` is mutable, heap-allocated, growable, and owned
- `String` is a wrapper around a `Vec<u8>`

```
let mut s = String::new();      // Empty string
let data = "initial contents";  // str slice

// `to_string` is available on any type that implements the `Display` trait
let s = data.to_string();                   // Convert &str to String
let s = String::from("initial contents");   // Same as above
s.push_str(" and more");                        // Append a string slice, does not take ownership
println!("{s}");                            // Prints "initial contents and more"
```

📂 **Collections** - String basic ops: Concat

Concat with `+` or `format!` macro

`+` Operator is actually a call to add which takes a reference to a `String` and returns a new `String`. So one of the strings will be moved and the other will be borrowed.

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // s1 has been moved here and can no longer be used

let s4 = String::from("tic");
let s5 = String::from("toc");
let s6 = format!("{s4}-{s5}"); // format! does not take ownership
```

📂 **Collections** - String basic ops: Indexing

- Cannot index directly with `[]` because of UTF-8 encoding**
- Use `&str` slices to index per byte (careful 💀!)
- Use `chars()` method to iterate over Unicode characters

```rust
let hello = "Здравствуйте";
let s = &hello[0..4];   // Corresponds to first 4 bytes so `Зд` in UTF-8

// To iterate per character regardless of size
for c in hello.chars() {
    println!("{c}");
    // prints `З`, `д`, `р`, `а`, `в`, `с`, `т`, `в`, `у`, `й`, `т`, `е`
}
```

📂 Collections - Hash Maps

- `HashMap<K, V>` is a collection of key-value pairs/dictionary/hash table with unique keys
- Homogenous typing for keys and values

```rust
use std::collections::HashMap;
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");  // later borrowed by get as `&str`
let score = scores.get(&team_name).copied().unwrap_or(0);
// get() returns an Option<&V> so we use copied() to get the value
// unwrap_or() returns the value or a default if None

// Iterate over key-value pairs
for (key, value) in &scores {
    println!("{key}: {value}");
}
```

📂 **Collections** - Hash Maps Ownership and updates

- Inserts take ownership of the key and value (see ./projects/hashmaps_test). Can use references but must be valid for the lifetime of the map.
- Overwrites the value if the key already exists. Use `entry` to insert instead.

```
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.entry(String::from("Yellow")).or_insert(50);  // `entry` returns Entry enum
scores.entry(String::from("Blue")).or_insert(50);  // `or_insert` returns a mutable reference
println!("{scores:?}");    // {"Blue": 10, "Yellow": 50}
```

Mutable references by `entry.or_insert` to update value

```
let text = "hello world wonderful world";
let mut map = HashMap::new();                                              // {}

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;   // Dereference count to update the value
}
println!("{map:?}");    // {}
```

⚠️ **Error Handling** - (Un)recoverable Errors overview 🚨

Two types of errors in Rust:

1. **Recoverable errors** are handled by `Result<T, E>` enum
2. **Unrecoverable errors** are handled by `panic!` macro

Let's us handle errors in a way that can be made explicit in the function signature.

🚨 **Error Handling** - Unrecoverable Errors and `panic!`

Full backtrace on demand when a program encounters an error that it cannot handle.

```rust
fn main() {
    let my_vec: Vec<i32> = vec![1, 2, 3];
    println!("{my_vec[99]}"); // Panics! Index out of bounds
}
```

When running with `RUST_BACKTRACE=1 cargo run`, the error message will include a backtrace. See next slide for the output:

```
~/out_of_bounds_panic main !1 ?1 ❯ RUST_BACKTRACE=1 cargo run
    'Finished' `dev` profile [unoptimized + debuginfo] target(s) in 0.01s
     'Running' `target/debug/out_of_bounds_panic`
my_deque[0]: a
thread 'main' panicked at src/main.rs:8:42:
Out of bounds access
stack backtrace:
   0: rust_begin_unwind
             at /rustc/129f3b9964af4d4a709d1383930ade12dfe7c081/library/std/src/panicking.rs:652:5
   1: core::panicking::panic_fmt
             at /rustc/129f3b9964af4d4a709d1383930ade12dfe7c081/library/core/src/panicking.rs:72:14
   2: core::panicking::panic_display
             at /rustc/129f3b9964af4d4a709d1383930ade12dfe7c081/library/core/src/panicking.rs:263:5
   3: core::option::expect_failed
             at /rustc/129f3b9964af4d4a709d1383930ade12dfe7c081/library/core/src/option.rs:1994:5
   4: core::option::Option::expect
             at /rustc/129f3b9964af4d4a709d1383930ade12dfe7c081/library/core/src/option.rs:895:21
   5: <alloc::collections::vec_deque::VecDeque as core::ops::index::Index<usize>>::index
             at /rustc/129f3b9964af4d4a709d1383930ade12dfe7c081/library/alloc/src/collections/vec_deque/mod.rs:2784:25
   6: out_of_bounds_panic::main
             at ./src/main.rs:8:42
   7: core::ops::function::FnOnce::call_once
             at /rustc/129f3b9964af4d4a709d1383930ade12dfe7c081/library/core/src/ops/function.rs:250:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

⚠️ **Error Handling** - Recoverable Errors and `Result`

- `Result<T, E>` is an enum with two variants: `Ok(T)` and `Err(E)` avail from prelude
- Use match to handle the `Result` enum (both `Ok` and `Err` cases)

```rust
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file_result = File::open("hello.txt");          // Returns Result<File, Error>

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {                       // ErrorKind enum
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,                                     // Returns the file handle when created
                Err(e) => panic!("Problem creating the file: {e:?}"),
            },
            other_error => {                                      // Choose to panic for other possible ErrorKind
                panic!("Problem opening the file: {other_error:?}");
            }
        },
    };
}
```

⚠️ **Error Handling** - Unwrap and expect shortcuts

Almost similar behaviors as `match` expression but more concise:

- `unwrap()` returns the value if `Ok` or panics with the error message
- `expect()` is similar but allows you to specify the error message

```rust
// Panics with the error message if missing file
let f = File::open("hello.txt").unwrap();

// Same as above but with custom message
let f = File::open("hello.txt").expect("Failed to open hello.txt");
```

⚠️ Error Handling - Propagating errors

Return the error to the calling function using type `Result<T, E>` in the function signature.

```
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {     // Take note of Result enum
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();
    // Possible to call `.read_to_string()` directly on the Result enum
    // even when Err. This is Propagating the error
    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    } // Returns the username or the error
}
```

⚠️ **Error Handling** - Propagating shortcut with `?` operator

Use of `?` operator to propagate errors and reduce boilerplate code. It is a shortcut for `match` and `return` the error.

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username = String::new();

    File::open("hello.txt")?.read_to_string(&mut username)?;

    Ok(username)
}
```

At each `?`, the error is returned to the calling function if it is of `Err` type. Otherwise, the value is unwrapped and returned.

⚠️ To panic! or not to panic!? 🚨

When writing library/API code ->better to return `Result` (let caller handle error)...

...But we can use `panic!` when testing, when the state is invalid, calling external code etc.

Take advantage of Rust's strong type system to catch errors at compile time (more concise code) and use `Result` enum to handle errors at runtime.

⚠️ OOP-styled safe design example 🚨

```rust
pub struct Guess {
    value: i32,
}

impl Guess {   // Interface for the struct with constructor and getter
    // Constructor panic if value is out of bounds
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
             panic!("Guess value must be between 1 and 100, got {value}.");
        }
        Guess { value }
    }
    // borrows self to get the value
    pub fn value(&self) -> i32 {
        self.value   // value is a private, cannot be modified
    }
}
```

Calling `Guess` constructor will panic if out-of-bounds (should be in API docs).

`value` can never return an invalid value of an invalid type.

🧩 **Generics** - Data type overview

- **Generics** allow you to define *functions*, *structs*, *enums*, and *methods* that work with any data type.
- Like C++ templates, but more type constraints (shared trait) and safety (borrow checker)

Example of a function that would :

- Take a reference to a slice of any type
- Return a reference to the an object of the same type

```
fn smallest<T>(list: &[T]) -> &T {...} //
```

Generics... Must implement traits

Traits are similar to interfaces in other languages. They define a set of methods that a type must implement.

A function cannot use a generic type `T` unless it knows that `T` implements a specific trait.

```rust
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];
    for item in list {
        if item > largest {   // Error! T does not implement the `PartialOrd` trait
            largest = item;
        }
    }
    largest
}

fn main() {    // Won't compile!!!
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest(&number_list);
    println!("The largest number is {result}");

    let char_list = vec!['y', 'm', 'a', 'q'];
    let result = largest(&char_list);
    println!("The largest char is {result}");
}
```

🧩 **Generics** - Struct example

Use a single generic type `T` for both fields of the struct. Each field cannot mix types.

```
struct BadPoint<T> {
    x: T,
    y: T,
}
let wont_work = BadPoint { x: 5, y: 4.0 };

struct GoodPoint<T, U> {   // `U` is just another generic type
    x: T,
    y: U,
}
let will_work = GoodPoint { x: 5, y: 4.0 };
```

🧩 **Generics** - Enum example

stdlib provides `Option` and `Result` enums that use generics. They can expression the presence or absence of a value or the success or failure of an operation.

`Result` even uses multiple generics for its `Ok` and `Err` variants.

```
enum Option<T> {
    Some(T),     // Holds some value of type T
    None,        // Does not hold a value
}
enum Result<T, E> {
    Ok(T),       // Holds a value of type T
    Err(E),      // Holds a value of Type E (error)
}
```

Generics - Method definitions

Methods written within an `impl` that declares the generic type will be defined on any instance of the type regardless of concrete type substituted.

```
struct Point<T> {
    x: T,
    y: T,
}
impl<T> Point<T> {    // Can be used by any Point struct
    fn x(&self) -> &T {
        &self.x
    }
}
impl Point<f32> {   // Specific to Point<f32> structs
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };    // Cannot use `.distance_from_origin()` because `Point<i32>`
    println!("p.x = {}", p.x());
    let p2 = Point { x: 5.0, y: 10.0 };
    println!("Distance from origin: {p2.distance_from_origin()}");
}
```

🛠️ **Traits** - Shared behavior

Similar to interfaces in other lang like *Java, C#*.

Group methods signatures into a set of behaviors tied to a type

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

Keywords:

- `pub` makes the trait public. Can be accessed outside of this crate
- `fn` defines a method signature
- `trait` keyword defines the trait.

Public `Summary` trait has a single method `summarize` that returns a `String`. Not implemented yet.

🛠️ **Traits** - Implementing a trait on a Type

As mentionned, a trait is a set of behaviors that a type can implement.

Keywords (continued):

- `impl` keyword to implement the trait on a type
- `for` keyword to specify the type

Here is how to implement the `Summary` trait on a `NewsArticle` and `Tweet` structs:

We are building an `aggregator` crate in `src/lib.rs`:

```rust
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

We must bring that trait into scope to use it. We can do this by adding a `use` statement at the top of the file.

```rust
use aggregator::{Summary, Tweet};

fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
```

🛠️ **Traits** - Trait as parameters

Define functions that accept multiple types using traits.

```rust
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}

pub fn notify<T: Summary>(item1: &T, item2: &T) {} // Same as above, but more verbose
                                                   // and allows multiple types
```

- `item` can be any type that implements `Summary`.
- Allows calling `summarize` method on `item`.
- Ensures type safety: only types implementing `Summary` are accepted.

This means we can call `notify` on both `NewsArticle` and `Tweet` instances, but not on other types.

🛠️ **Traits** - Return types with implemented traits

Return types can also be traits. This is useful when returning different types that implement the same trait. **WARNING: Needs to return only one type.**

```rust
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from(
                "Penguins win the Stanley Cup Championship!",
            ),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                 hockey team in the NHL.",
            ),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from(
                "of course, as you probably already know, people",
            ),
            reply: false,
            retweet: false,
        }
    }  // Won't COMPILE! Possible workaround with Box<dyn Summary>
}
```

🛠️ **Traits** - Powerful conditionally `impl` method

Using the `+` operator to specify multiple traits. Here we can only use cmp_display on types that implement both `PartialOrd` (comparison) and `Display`.

```rust
use std::fmt::Display;
struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

🛟 **Lifetimes** - Overview

Rust compiler requires annotation of lifetimes (how long references are valid) to ensure runtime safety.

Main goal is to prevent dangling references 💀.

```rust
fn main() {
    let r;   // r is not null, its not even init

    {
        let x = 5;
        r = &x;
    }

    println!("r: {r}");     // Error! x is out of scope (does not live long enough)
                            // and r is a dangling reference
}
```

🛟 **Lifetimes** - Borrow checker mechanism

// WONT COMPILE!

```rust
fn main() {
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |fire
                          //          |
    println!("r: {r}");   //          |
}                         // ---------+
```

// OK!

```rust
fn main() {
    let x = 5;            // ---------+-- 'b
                          //          |
    let r = &x;           // --+-- 'a  |
                          //   |      |
    println!("r: {r}");   //   |      |
}                         // --+      |
                          //          |
                          // ---------+
```

BC compares a' and b' lifetimesd to ensure that the reference in r is valid when it is used. Basically is 'a > 'b?

🛟 **Lifetimes** - Function signature, not so simple

Maintain the reference's validity by specifying lifetimes in the function signature, otherwise the compiler cannot determine the lifetime of the reference inside the function. See /projects/lifetime_func_sig_err

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
// Used as follows
let string1 = String::from("long string is long");
let string2 = "shortest";
println!("The longest string is {longest(&string1, string2)}");    // Won't Compile!
```

🛟 **Lifetimes** - Annotation Syntax

- Single quotes and usually are short with ids like `'a`, `'b`...
- Meant to tell rust how long references are valid in relation to each other.

<u>Standalone example just to show syntax:</u>

```
&i32         // a reference
&'a i32      // a reference with an explicit lifetime
&'a mut i32 // a mutable reference with an explicit lifetime
```

🛟 **Lifetimes** - Function signatures

Use lifetime annotation like generic types to specify that the references must have the same lifetime.

-> In other words, the **lifetime of the reference returned** by the function is the **same as the smaller of the lifetimes of the references passed in**.

```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
// No changes to lifietimes themselves, only making borrow checker
// reject invalid references to lifetimes constraints
```

And to showcase the use of the function on next slide...

```rust
// OK!
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {result}");
    } // string2 goes out of scope here, but string1 is still valid
}
```

See /projects/lifetime_func_sig_err

```rust
// WON'T COMPILE!
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }  // `longest()` return lifetime is smaller of lifetimes passed-in
       // therefore borrow checker will reject as string2 does not live long enough
    println!("The longest string is {result}");
}
```

🛟 **Lifetimes** - In `struct` definitions

Safety for the struct instance cannot outlive the reference it holds.

```rust
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().unwrap();
    let i = ImportantExcerpt { part: first_sentence };
}
```

Let's say first sentence is invalidated by the novel being dropped (out of scope, freed), then the field `part` in `ImportantExcerpt` would be a dangling reference. Thus, the **borrow checker** will reject the code.

🛟 **Lifetimes** - Defaults and rules

To simplify, Rust has **lifetime elision rules** that allow the compiler to infer lifetimes in many cases. Here are the elision rules:

1. Each parameter that is a reference gets its own lifetime parameter: `fn foo<'a>(x: &'a i32)` and `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`.
2. If exactly 1 input lifetime param, gets assigned to all output lifetime params: `fn foo<'a>(x: &'a i32) -> &'a i32`.
3. If multiple input lifetime params, but one is `&self` or `&mut self`, the lifetime of `self` is assigned to all output lifetime params.

`'static` lifetime is a special lifetime that lasts for the entire duration of the program (i.e. string litterals)

```rust
let s: &'static str = "I have a static lifetime.";
```

🛠️ **Practical project #1** - Write an I/O CLI program

Project for a `grep` clone CLI app covers:

1. Code organization (crates, modules)
2. Use of containers and strings
3. Error handling
4. Using traits and lifetimes
5. Testing and documentation

Klabnik, Steve, and Carol Nichols. The Rust Programming Language. 2nd ed., No Starch Press.

🚀 Demo Time!

🧑🏻‍💻 I/O CLI program `grep` clone: minigrep

🚀 Conclusion

Only scratched the surface of Rust's features as this material covers half of the book! Remaining topics to be covered:

- Closures, iterators, smart pointers, advanced pattern matching and concurrency (language features)
- Testing, documentation, and cargo (tooling)

Still, we have seen how Rust's strong type system, ownership, and borrowing and why it is considered a strong candidate for replacing C/C++ in systems programming, backends, and high-performance applications.

Regardless of the learning curve, because of its performance and memory safety, **Rust is being more adopted in both academia and industry.**

Klabnik, Steve, and Carol Nichols. The Rust Programming Language. 2nd ed., No Starch Press.

Rust for Linux. 2024. https://rust-for-linux.com/industry-and-academia-support