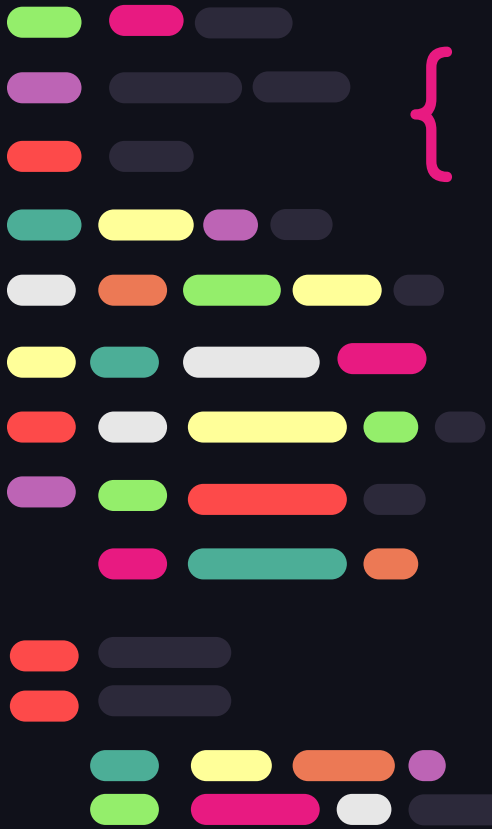
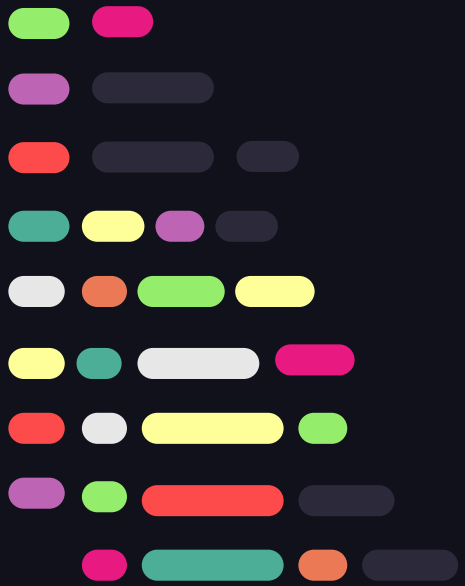


Introduction to Rust Language

<Alexandre Cotorobai - 107849>
<Bernardo Figueiredo - 108073>
<Joaquim Rosa - 109089>

<CSLP 2023/2024>

Why Rust?



01 Performance

Rust is blazingly fast and memory-efficient

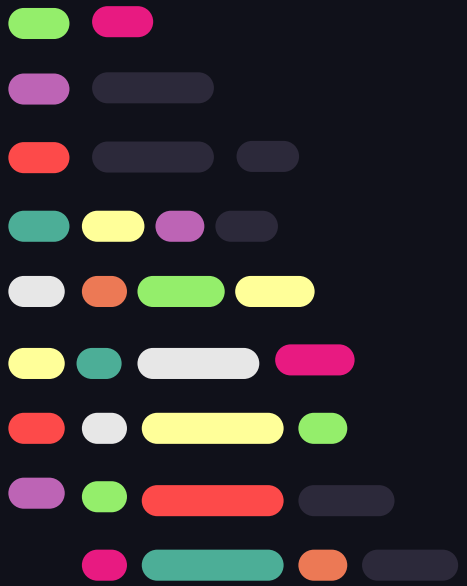
02 Reliability

Rust's rich type system and ownership model guarantee memory and thread-safety

03 Productivity

Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling

Rust Objectives



01 Performance

Optimal speed and resource efficiency through fine-grained control.

02 Memory Safety

Eliminate memory-related errors for robust and secure code.

03 Concurrency

Safe and efficient concurrent programming without data races.

Key Features & Concepts

Ownership

Only one variable can "own" data at a time in Rust, and the owner is responsible for freeing up the associated memory.

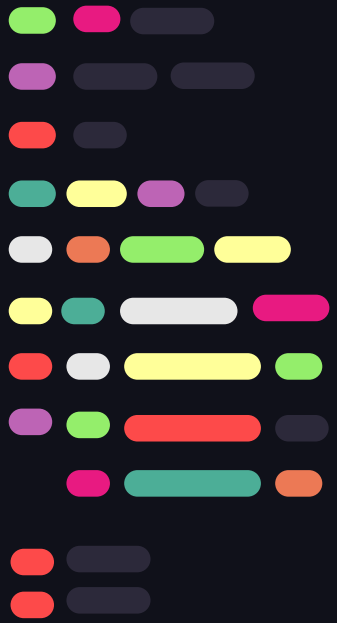
Borrowing

Rust allows variables to borrow references to owned data, enabling multiple parts of the code to interact with it simultaneously without transferring ownership.

Lifetimes

Lifetimes in Rust specify how long data remains valid, ensuring references don't outlive the data they point to, preventing potential issues.

Paradigms of Rust



Imperative

Instructions are executed sequentially, and flow control is managed through loops, conditionals, and control structures.

Functional

Includes features like higher-order functions, immutability, functional expressions and pattern destructuring.

Object Oriented

Supports object-oriented programming through types like structs and enums.

Concurrent

Rust's ownership system ensures memory safety and high-performance concurrent programming.

The Rust Ecosystem

rustup

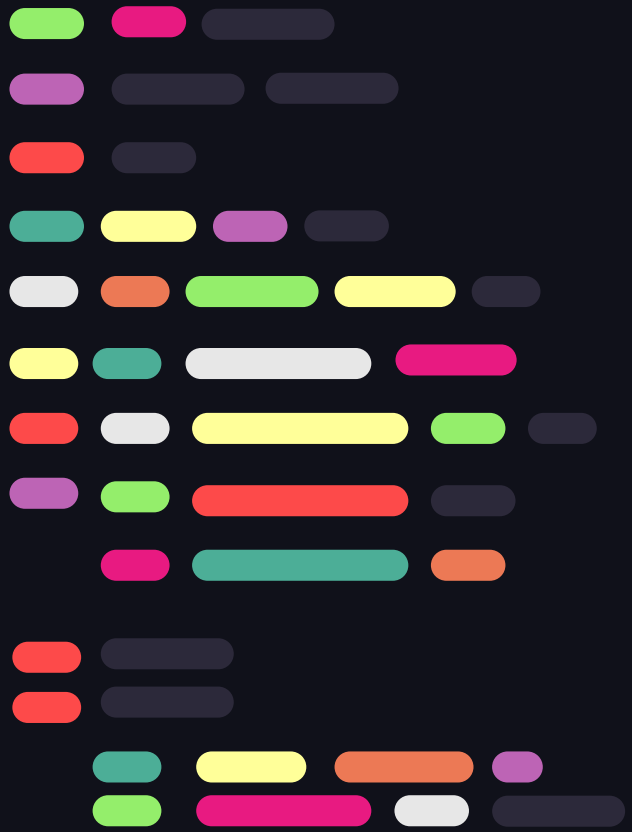
Multiple concurrent Rust toolchains can be installed and managed via rustup.

Cargo

Rust installations come with Cargo, a command line tool to manage dependencies, run tests, generate documentation, and more.

crates.io

crates.io serves as the community platform for sharing and discovering Rust libraries.



{ ..

Syntax Semantics

} ..

Variables and Mutability

Immutable/mutable variables

```
// this will not compile
let x = 5;                    // variables are immutable by default
println!("The value of x is: {x}");
x = 6;
println!("The value of x is: {x}");
```

```
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     println!("The value of x is: {x}");
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable
```

```
let mut x = 5;
println!("The value of x is: {x}");
x = 6;
println!("The value of x is: {x}");
```

Constants

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

They’re always immutable

Variables and Mutability

Shadowing

```
let x = 5;

let x = x + 1;

{
    let x = x * 2;
    println!("The value of x in the inner scope is: {x}");  // x = 12
}

println!("The value of x is: {x}"); // x = 6
```

Shadowing VS Mutability

```
let spaces = "   ";
let spaces = spaces.len();
```

Creates a new variable

```
// this will not compile
let mut spaces = "   ";
spaces = spaces.len();
```

Mutates an existing one

Data Types

Rust is a statically typed language

Scalar Types: integers, floating-point numbers, Booleans and characters.

Compound Types: tuples and arrays.

Data Types

The compiler can infer what types based on the value and how it is used.

When many types are possible, type annotation is needed

```
let guess: u32 = "42".parse().expect("Not a number!");
```

Implicit / explicit in Floating-Point Types

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

Not expliciting type

```
let guess = "42".parse().expect("Not a number!");
```


```
$ cargo build
   Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annota
error[E0282]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |         ^^^^^
  |
help: consider giving `guess` an explicit type
```

Data Types

Compound Types

The Tuple Type

```rust
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    // prints 6.4
    println!("The value of y is: {y}");

    // prints 500
    println!("The first value of tup is: {}", tup.0);

}
```

The Array Type

Example of an Array

```rust
let a = [1, 2, 3, 4, 5];
```

Specifying array type and size

```rust
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

Initializing array with 5 elements set by default to value 3

```rust
let a = [3; 5];
```

```rust
let a = [3, 3, 3, 3, 3];
```

Functions: Important Notes

Statements:instructions that perform some action and do not return a value.

Expressions:evaluate to a resultant value.

```
// This is a statement
let y = 6;
```

```
// This will not compile
let x = (let y = 6);
```

```
error: expected expression, found `let` statement
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |              ^^^
```

Function calls, macro calls and scope blocks created with curly brackets, are all expressions in Rust.

```
let y = {
    let x = 3;
    x + 1   // this is an expression
};
```

- Expressions don't end with semicolons
- Adding a semicolon will turn it into a statement

Functions

Without Return Values

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {x}");
}
```

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 1.21s
     Running `target/debug/functions`
The value of x is: 5
```

With Return Values

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

```
The value of x is: 6
```

Control Flow: if/else

Multiple conditions with if, else and else if

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

NOTE: In Rust, this doesn't work


```
let number = 3;

if number {
    println!("number was three");
}
```

```
--> src/main.rs:4:8
  |
4 |     if number {
  |        ^^^^^^ expected `bool`, found integer
```

```
let number = 3;

if number != 0 {
    println!("number was something other than zero");
}
```

Control Flow: if in a let statement

```rust
let condition = true;
let number = if condition { 5 } else { 6 };

println!("The value of number is: {number}");
```

The if and else arms must return the same type

```rust
let condition = true;

let number = if condition { 5 } else { "six" };
```

```
--> src/main.rs:4:44
  |
4 |     let number = if condition { 5 } else { "six" };
  |                                 -          ^^^^^ expected integer, found `&str`
  |                                 |
  |                                 expected because of this
```

Control Flow: loops

Repeating Code with Loop

```
fn main() {
    loop {
        println!("again!");
    }
}
```

Returning Values from Loops

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");
    // The result is 20
}
```

Control Flow: loop labels

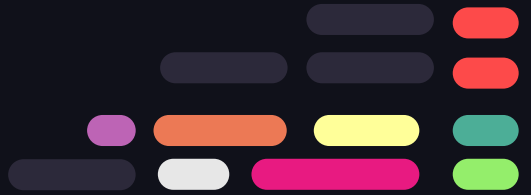
Loop Labels

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

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

Use *break* or *continue* with a loop label to specify which loop to apply

Output

```
count = 0
remaining = 10
remaining = 9
count = 1
remaining = 10
remaining = 9
count = 2
remaining = 10
End count = 2
```



Control Flow: while/for

Conditional Loops with while

```rust
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{number}!");

        number -= 1;
    }

    println!("LIFTOFF!!!");
}
```

For Loops by range

```rust
fn main() {
    for number in (1..4).rev() {
        println!("{number}!");
    }
    println!("LIFTOFF!!!");
}
```

Looping Through a Collection

```rust
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("the value is: {element}");
    }
}
```

Biggest Advantage: enhance code safety by eliminating the risk of bugs related to array access.

Ownership

Definition: set of rules that govern how a Rust program manages memory.

Ways of managing memory:

- Garbage Collector ✗
- Manually allocate/free ✗
- System of Ownership -> Rust ✓

Ownership

Ownership Rules:

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

Variable Scope

```
{                       // s is not valid here, it's not yet declared
    let s = "hello";    // s is valid from this point forward

    // do stuff with s
}                       // this scope is now over, and s is no longer valid
```

Ownership: Variables

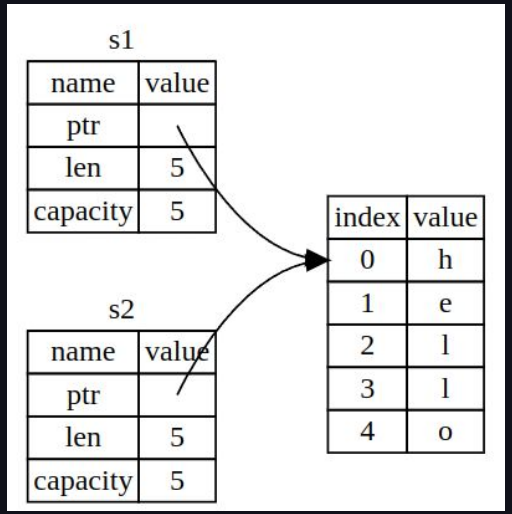
Variables and Data Interacting with Move

```
let x = 5;
let y = x;
```

5 is a literal. It makes a copy

```
let s1 = String::from("hello");
let s2 = s1;
```

But using a non-literal String..



Other Langs: Shallow Copy ✗

Rust: Move ✓

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

Ownership: Functions

Functions and return values

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

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
                                    // use x afterward

} // Here, x goes out of scope, then s. But because s's value was moved, nothing
  // special happens.

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

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

The usage of s after the call to takes_ownership would throw a compile-time error

```
fn main() {
    let s1 = gives_ownership();         // gives_ownership moves its return
                                        // value 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, which also
                                        // moves its return value into s3
} // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing
  // happens. s1 goes out of scope and is dropped.

fn gives_ownership() -> String {             // gives_ownership will move its
                                             // return value into the function
                                             // that calls it

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

    some_string                              // some_string is returned and
                                             // moves out to the calling
                                             // function
}

// 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

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

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

References allow you to refer to some value without taking ownership of it.

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

    change(&s);
}

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

Compilation Error: some_string is not mutable

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
 --> src/main.rs:8:5
  |
7 | fn change(some_string: &String) {
  |                        ------- help: consider changing this to be a mutable referenc
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `some_string` is a `&` reference, so the data it
```

Mutable References

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

    change(&mut s);
}

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

It allows to mutate borrowed values

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

let r1 = &mut s;
let r2 = &mut s;

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

Data Race happens when:

- Two or more pointers access the same data at the same time.
- At least one of the pointers is being used to write to the data.
- There's no mechanism being used to synchronize access to the data.

Slice Type: Some context

```rust
fn main(){

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

    let word = first_word(&s);      // word will get the value 5

    s.clear();                      // this empties the String, making it equal to ""

    // word still has the value 5 here, but there's no more string that
    // we could meaningfully use the value 5 with. word is now totally invalid!
}

fn first_word(s: &String) -> usize {   // returns a usize
    let bytes = s.as_bytes();          // convert String to an array of bytes

    for (i, &item) in bytes.iter().enumerate() {    // iter() returns each element in a collection
                                                    // enumerate() returns a tuple of the index and the element
        if item == b' ' {                           // b' ' is a byte literal
            return i;
        }
    }

    s.len()
}
```

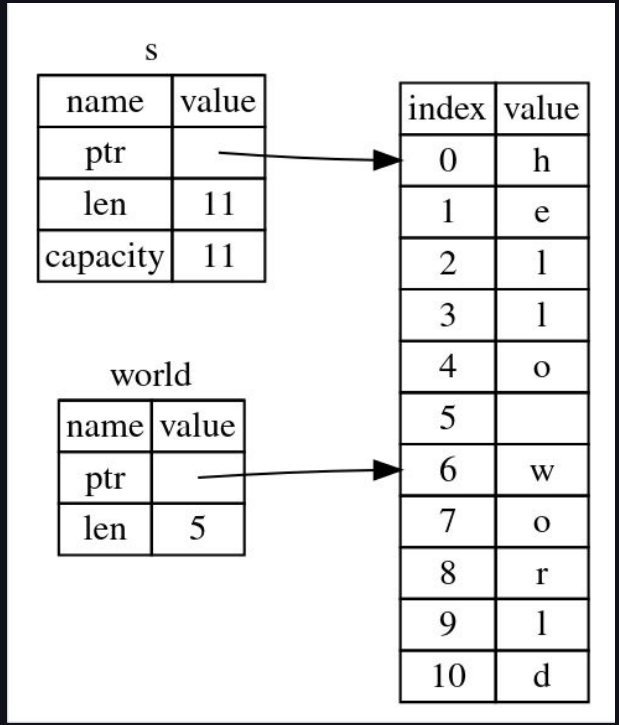
Slice Type: String Slices

A String Slice is a part of a String

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

let hello = &s[0..5];
let world = &s[6..11];
```

- &s[0..5] is the same as &s[..5]
- &s[6..11] is the same as &s[6..]

(this excludes the last index)



Slice Type: String Slices

```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];
        }
    }

    &s[..]
}
```

&String pode ser convertido para &str, mas não vice versa.

```rust
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // error!

    println!("the first word is: {}", word);
}
```

- The first code had a logical flaw that didn't generate immediate errors.
- Issues would arise later if we continued using the first word index with an emptied string.
- Slices prevent this bug and prompt us to identify the issue earlier in our code.

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
  --> src/main.rs:18:5
   |
16 |     let word = first_word(&s);
   |                           -- immutable borrow occurs here
17 |
18 |     s.clear(); // error!
   |     ^^^^^^^^^ mutable borrow occurs here
19 |
20 |     println!("the first word is: {}", word);
   |                                       ---- immutable borrow later used here
```

Slice Type: Other Slices

Array Slices

```
let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];
```

Type: &[i32]

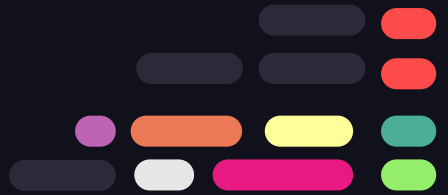
Structs

Definition

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

Creating an instance and updating

```
let mut user1 = User {
    active: true,
    username: String::from("someusername123"),
    email: String::from("someone@example.com"),
    sign_in_count: 1,
};

user1.email = String::from("anotheremail@example.com");
```



Structs: Field Init Shorthand

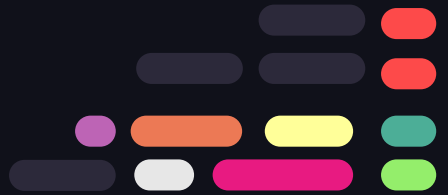
Function returning a instance of User

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username: username,
        email: email,
        sign_in_count: 1,
    }
}
```

Using the Field Init Shorthand

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username,
        email,
        sign_in_count: 1,
    }
}
```

Having to repeat the email and username field names and variables is a bit tedious...



Structs: Creating Instances from other instances

```
let user2 = User {
    active: user1.active,
    username: user1.username,
    email: String::from("another@example.com"),
    sign_in_count: user1.sign_in_count,
};
```

Using the Struct Update Syntax

```
let user2 = User {
    email: String::from("another@example.com"),
    ..user1
};
```

Note: In this example, we can no longer use **user1** as a whole after creating **user2** because the **String** in the **username** field of **user1** was moved into **user2**. To prevent this, we should only used the **active** and **sign_in_count** values from **user1** (types that implement the Copy trait).

Structs: Methods

```rust
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

```rust
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Structs: Other

Tuple Structs

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

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

Useful when you want to give the whole tuple a name and make the tuple a different type from other tuples.

Unit-Like Structs

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

Function as concise markers or signals. They are valuable for representing concepts without the need for data storage, serving well to indicate events, initializations, or units.

Error Handling

Error Categories:

- Recoverable (e.g file not found)
- Unrecoverable (caused by bugs)

Rust doesn’t have exceptions:

- For recoverable Errors -> type `Result<T, E>`
- For unrecoverable Errors -> `panic!` macro

panic!

Calling panic!

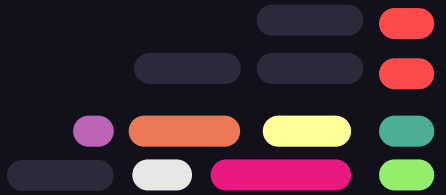
```
fn main() {
    panic!("crash and burn");
}
```

Caused by a bug

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s
     Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

A backtrace is a list of all the functions that have been called to get to this point.



Errors: Result

Result enum

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

Handling Recoverable Errors

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

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

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

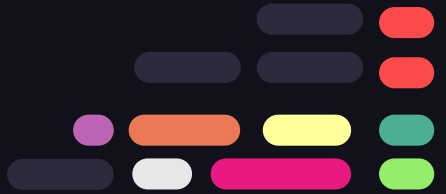
`Ok` => this code will return the inner `file` value out of the `Ok` variant, and we then assign that file handle value to the variable `greeting_file`.

`Err` => handles the case where we get an `Err` value from `File::open`.

Generic Data: Some context

```
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

- ➔ Duplicating code is tedious and error prone
- ➔ Have to update the code in multiple places when we want to change it



Generic Data: Functions

```rust
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

- This function is generic over some type **T**.
- Has a parameter **list**, which is a slice of values of type **T**.
- Return a reference to a value of type **T**.


```
error[E0369]: binary operation `>` cannot be applied to type `&T`
 --> src/main.rs:5:17
  |
5 |         if item > largest {
  |            ---- ^ ------- &T
  |            |
  |            &T
  |
help: consider restricting type parameter `T`
  |
1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
  |             ++++++++++++++++++++++
```

Using **std::cmp::PartialOrd** trait we restrict the types valid for **T** to only those that can be ordered.

Generic Data: Other contexts

In Structs Definitions

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

In Enums Definitions

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

In Methods Definitions

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

Traits

A Trait defines functionality a particular type has and can share with other types. Defines shared behavior in an abstract way.

Trait Definition

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

Trait Application

```
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());
}
```

Trait Implementation

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

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

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

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

Traits

```
trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Sometimes it’s useful to have default behavior for some or all of the methods in a trait instead of requiring implementations for all methods on every type. We can choose to keep or overwrite these behaviors when implementing.

```
trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

Default implementations can call other methods in the same trait, even if those other methods don’t have a default implementation.

Traits as Parameters

Without Trait Bound Syntax

Single Parameter

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

Multiple Parameters

```rust
fn notify(item1: &impl Summary, item2: &impl Summary) {
    // ...
}
```

Multiple Trait Bonds

```rust
fn notify(item: &(impl Summary + Display)) {
    // ...
}
```

With Trait Bound Syntax

Single Parameter

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

Multiple Parameters

```rust
fn notify<T: Summary>(item1: &T, item2: &T) {
    // ...
}
```

Multiple Trait Bonds

```rust
fn notify<T: Summary + Display>(item: &T) {
    // ...
}
```

Using **impl** Trait is a concise shorthand for straightforward cases, while the equivalent **Trait Bound Syntax** is more verbose and allows expressing more complexity in other situations.

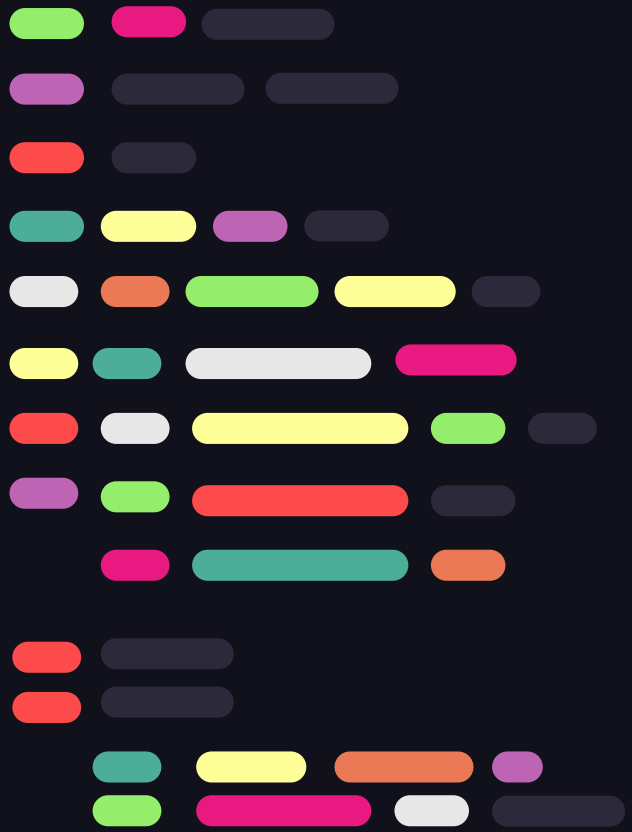
Lifetimes

Definition: named regions of code that a reference must be valid for.

Liveness: A variable is live if its current value may be used later in the program.

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

```
fn example_2() {                  //
    let foo = 69;                 //
    let mut r;                    //
                                  //
    {                             //
        let x = 42;               //
        r = &x;                   // ----------+-- 'r = { &x }
                                  //           |
        println!("{}", *r);       // ----------+
    }                             //
                                  //
    r = &foo;                     // ----------+-- 'r = { &foo }
                                  //           |
    println!("{}", *r);           // ----------+
}                                 //
```



Advanced Features!

Unsafe Rust

Keyword used to mark code to be ignored by the compiler for memory safety.

Doesn’t automatically mean that your code is unsafe, use precaution.

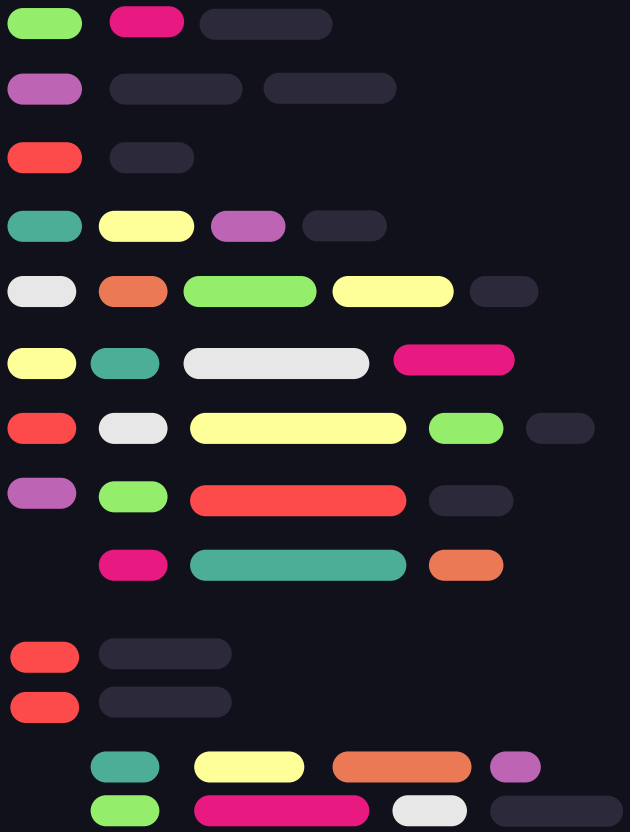
Useful in many situations, like when interfacing with C libraries, and more!

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

Macros

Rust macros provide powerful metaprogramming capabilities.
Create reusable code patterns for enhanced productivity and maintainability in complex projects.

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```



Rust!}

*