# Rust

## Functional Programming

# Rust

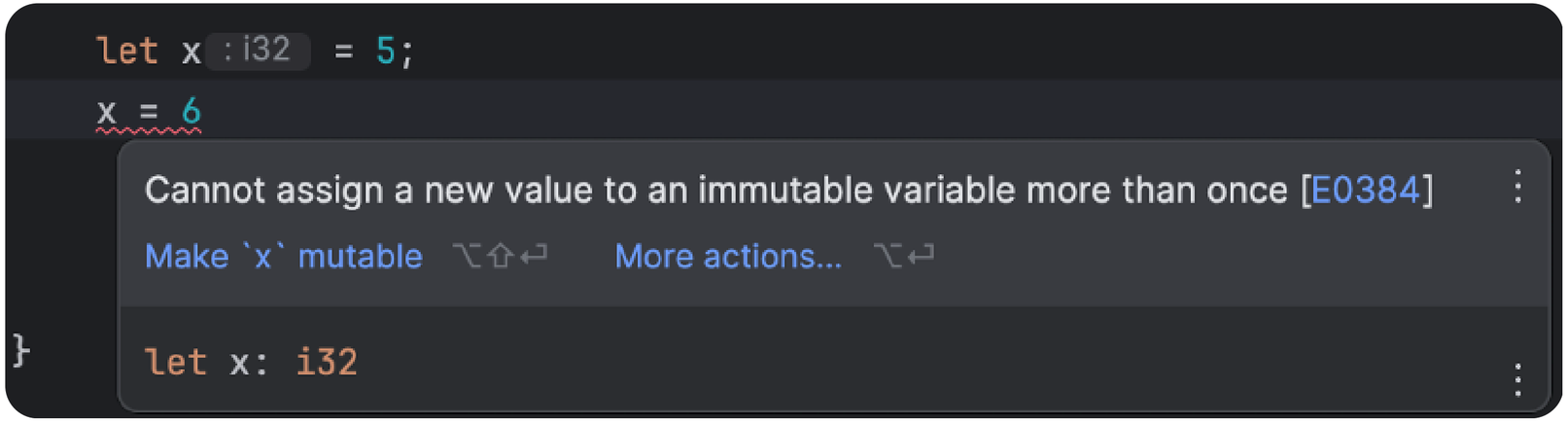
**Multi-paradigm programming language focused on safety and performance.**

**Unique approach to memory management and concurrency.**

**Incorporates functional programming features for concise, readable, and reliable code.**

# Immutability by Default

**Variables in Rust are immutable by default. This means that once a variable is set to a value, it cannot be changed:**




**To allow a variable's value to be changed, it must be explicitly declared as mutable:**

```
let mut x :i32 = 5;
x = 6
```

# Clear Function Input and Output

**Specifying clear types for inputs and outputs, along with using pure functions, enhances code safety and clarity. Pure functions ensure consistent outputs with no side effects, further reducing errors and making the code easier to understand.**

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

# Handling Optional Values

**Rust's Option are used to handle the presence or absence of a value in a type-safe manner. This approach eliminates the risk of null pointer exceptions common in many other programming languages.**

```rust
let find_person: Option<&str> = Some("Alice");

println!("{:?}", find_person);

// Output: Some("Alice")
```

# First-Class Functions and Closures

**In Rust, functions are treated as "first-class citizens," meaning they can be passed as arguments, returned from other functions, and assigned to variables, much like any other data type.**

```
let add_one : fn(i32) → i32 = |x: i32| x + 1;
println!("The result is: {}", add_one(5));
```

# First-Class Functions and Closures

- **Fn: Can be called multiple times without mutating state.**
- **FnMut: Can be called multiple times and may mutate state.**
- **FnOnce: Can be called once and might consume the variable, so it cannot be used again.**

```rust
// Fn example: Closure that does not mutate state.
let add_ten : fn(i32) -> i32 = |x: i32| x + 10;
println!("add_ten: {}", add_ten(5));

// FnMut example: Closure that mutates state.
let mut counter : i32 = 0;
let mut increment_counter : fn() -> i32 = || {
    counter += 1; // Mutates the `counter` variable.
    counter
};
println!("increment_counter: {}", increment_counter());
println!("increment_counter: {}", increment_counter());

// FnOnce example: Closure that consumes captured variable.
let text : String = "Hello".to_string();
let consume_text : fn() = move || println!("consume_text: {}", text);
consume_text();
// `consume_text` cannot be used again after this point.
```

# Pattern Matching

**Pattern matching in Rust is a powerful feature that allows you to check a value against a series of patterns and execute code based on which pattern matches.**

```rust
let pair : (i32, i32) = (2, -2);

match pair {
    (x : i32, y : i32) if x == y => println!("Equal"),
    (x : i32, 0) => println!("x axis: {}", x),
    (0, y : i32) => println!("y axis: {}", y),
    _ => println!("Anywhere"),
}
```

# Pattern Matching

- **Exhaustiveness Checking: Rust's compiler ensures that all possible cases are covered in a match expression, reducing the risk of bugs.**
- **Destructuring: Patterns can destructure enums, structs, tuples, and references, which allows for extracting parts of complex data types.**
- **Binding: Pattern matching can bind values to names, making them available in the associated code block.**
- **Guards: Patterns can be augmented with guards, additional conditions specified after the if keyword that must also be met for the matching arm to be selected.**

# Pattern Matching

```
5 usages
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
}

// Here, the enumeration Message is defined, which can be one of three kinds:

// Quit – a simple variant, carrying no data.
// Move – contains named fields x and y, both of type i32.
// This variant is used to represent a movement message with two coordinates.

// Write – contains a single unnamed field of type String.
// This variant represents a message with text.


let msg1 = Message::Write(String::from( s: "hello"));

let msg2 = Message::Move { x: 23, y: 12 };

// Here, a variable `msg1` and `msg2` is created, which is assigned the value of
// `Message::Write` with the string "hello" and
// `Message::Move` with { x: 23, y: 12 } as an argument.
// This demonstrates how instances of different enumeration variants can be created.

match msg1 {
    Message::Quit => println!("The Quit variant"),
    Message::Move { x :i32 , y :i32 } => println!("Move in the x: {} and y: {}", x, y),
    Message::Write(text :String ) => println!("Text message: {}", text),
}

// The `match` construct here is used for branching the program depending on the enum variant stored in `msg1` and `msg2`. `match` checks each variant:
//
//If `msg` is `Message::Quit`, a message about the Quit variant is printed.
//If `msg` matches `Message::Move`, the fields `x` and `y` are extracted, and a message with these coordinates is printed.
//If `msg` matches `Message::Write`, the `text` string is extracted, and a message with this text is printed.
```

# Algebraic Data Types and Pattern Matching

**Rust's enums and pattern matching allow for expressive and safe handling of various states and conditions. This feature is foundational to Rust's error handling and optional values, via the Result and Option types.**

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

fn divide(numerator: f64, denominator: f64) -> Result<f64, &'static str> {
    if denominator == 0.0 {
        Err("Cannot divide by zero.")
    } else {
        Ok(numerator / denominator)
    }
}
```

# Higher-Order Functions

**In Rust, higher-order functions (HOFs) are functions that can take one or more functions as arguments and/or return a function.**

```rust
// `apply_twice` takes a function and returns a new function that applies the original twice.
1 usage
fn apply_twice(f: fn(i32) -> i32) -> Box<dyn Fn(i32) -> i32> { Box::new(move |x: i32| f(f(x))) }

// `add_one` is a simple function that adds 1 to its input, showcasing function as a value.
fn add_one(x: i32) -> i32 { x + 1 }

// Applying `add_one` twice through `apply_twice`, demonstrating higher-order functions.
let double_add_one: Box<dyn Fn(...)> = apply_twice(add_one);
println!("Result: {}", double_add_one(5));
```

# Currying and partial application

* We can implement curried functions in Rust, but it’s not idiomatic.

```
2 usages
fn apply<F>(f: F) -> impl Fn(i32) -> i32
    where
        F: Fn(i32) -> i32,
{
    move |x :i32 | f(x)
}
1 usage
fn double(n: i32) -> i32 {
    n * 2
}
1 usage
fn sqr(n: i32) -> i32 {
    n * n
}

println!("Apply sqr 6 : {}", apply(sqr)(6));
println!("Apply double 2 : {}", apply(double)(2));

// Apply sqr 6 : 36
// Apply double 2 :4
```

# Iterators and Combinators

**Rust's iterators and combinators support a functional approach to operating on sequences of items. They are lazy, only executing when consumed, and support a range of operations like map, filter, and fold.**

```rust
let vec : Vec<i32> = vec![1, 2, 3, 4];
let iter : Iter<i32> = vec.iter();
println!("vec: {:?}", vec);
println!("iter: {:?}", iter);

let squares: Vec<i32> = vec.iter().map(|&x : i32 | x * x).collect();
println!("squares: {:?}", squares);

let evens: Vec<i32> = vec.iter().filter(|&x : &i32 | x % 2 == 0).cloned().collect();
println!("evens: {:?}", evens);

let sum: i32 = vec.iter().fold( init: 0,  f: |acc : i32 , x : &i32 | acc + x);
println!("sum: {}", sum);

// vec: [1, 2, 3, 4]
// iter: Iter([1, 2, 3, 4])
// squares: [1, 4, 9, 16]
// evens: [2, 4]
// sum: 10
```

# Merge Sort

```rust
2 usages
fn split_helper(sub_arr: &[i32]) -> (Vec<i32>, Vec<i32>) {
    match sub_arr {
        [] => (vec![], vec![]),
        [front : &i32 , rest : &[i32] @ ..] => {
            let (half_one : Vec<i32> , half_two : Vec<i32> ) = split_helper(rest);
            (half_two, std::iter::once(*front).chain(half_one.into_iter()).collect())
        }
    }
}
```

```
3 usages
fn merge(left: Vec<i32>, right: Vec<i32>) -> Vec<i32> {
    match (left.as_slice(), right.as_slice()) {
        ([], []) => vec![],
        ([], right_slice :&[i32] ) => right_slice.to_vec(),
        (left_slice :&[i32] , []) => left_slice.to_vec(),
        ([left_first :&i32 , left_rest :&[i32] @ ..], [right_first :&i32 , right_rest :&[i32] @ ..])
            if right_first >= left_first => {
                [vec![*left_first], merge(left_rest.to_vec(), right.to_vec())].concat()
            }
        ([left_first :&i32 , left_rest :&[i32] @ ..], [right_first :&i32 , right_rest :&[i32] @ ..]) =>
            {
                [vec![*right_first], merge(left.to_vec(), right_rest.to_vec())].concat()
            }
    }
}
```

```rust
fn merge_sort(arr: Vec<i32>) -> Vec<i32> {
    match arr.len() {
        0 => vec![],
        1 => arr,
        _ => {
            let (left : Vec<i32> , right : Vec<i32> ) = split_helper(&arr);
            let left_sorted : Vec<i32> = merge_sort(left);
            let right_sorted : Vec<i32> = merge_sort(right);
            merge(left_sorted, right_sorted)
        }
    }
}
```

# Conclusion

**Rust integrates functional programming elements at its foundation, which enable Rust code to adopt functional-style patterns. These elements enhance code clarity, optimize performance, and minimize potential bugs.**

**While Rust includes fundamental concepts of functional programming, it remains distinct from purely functional languages like Haskell or Elm. But still, it might be a good practice to selectively incorporate these aspects to refine specific code blocks without fully committing to a functional programming paradigm.**