The Rust programming language

Paolo Bettelini

Contents

1 Data Types **2**

1.1 Basic Types 2

1.2 Tuples 2

1.3 Arrays 2

1.3.1 Definition 2

1.3.2 Indexing 2

1.3.3 Slices 2

2 Loops **4**

2.1 Returning from loops 4

2.2 Labels 4

2.3 Returning from labelled loops 4

3 Pattern Matching **4**

3.1 Basic 4

3.2 Destructuring 6

3.3 Ignoring values 6

3.4 Match guards 7

3.5 @ bindings 7

4 Option **8**

5 Result **8**

5.1 Definition 8

5.2 ? operator 9

1 Data Types

1.1 Basic Types

```
// boolean
bool

// signed integers
i8, i16, i32, i64, i128, isize

// unsigned integers
u8, u16, u32, u64, u128, usize

// floating points
f32, f64

// Text
char, String, str
```

1.2 Tuples

Tuples are a combination of multiple types. Tuples can contain any number of types and/or other tuples.

```
let coordinates = (101, 3, 4);
let person = ("Paolo", "Bettelini", 18);
let status: (bool, (u128, i32)) = (true, (1u128, 2));
```

1.3 Arrays

1.3.1 Definition

An array is defined by its type and length.

```
let values = [1, 2, 3, 4, 5];
// with explicit type
let values: [i32; 5] = [1, 2, 3, 4, 5];
```

We can also initialize an array by specifying its default value and length

```
let values = [0; 5]; // [0, 0, 0, 0, 0]
```

1.3.2 Indexing

We can index an array element using the square brackets

```
let first = values[0];
let second = values[1];
```

1.3.3 Slices

We can point to a portion of the array using slices

```
let slice = &values[1..5];
let slice = &values[1..=5];
let slice = &values[..5];
let slice = &values[1..];
let slice = &values[..];
```

2 Loops

2.1 Returning from loops

```
let mut counter = 0;

let result = loop {
    counter += 1;

    if counter == 10 {
        break counter;
    }
};
```

2.2 Labels

```
'outer: loop {
    'inner: loop {
        // This breaks the inner loop
        break;
        // This breaks the outer loop
        break 'outer;
    }
}
```

2.3 Returning from labelled loops

```
let mut counter = 0;

let result = 'outer: loop {
    counter += 1;

    if counter == 10 {
        break 'outer counter;
    }
};
```

3 Pattern Matching

3.1 Basic

```
    let x = 5;

```

```
match x {
  // matching literals
  1 => println!("one"),
  // matching multiple patterns
  2 | 3 => println!("two or three"),
  // matching ranges
  4..=9 => println!("within range"),
  // matching named variables
  x => println!("{}", x),
  // default case (ignores value)
  _ => println!("default Case")
}
```

3.2 Destructuring

```
struct Point {
    x: i32,
    y: i32,
  }

  let p = Point { x: 0, y: 7 };

  match p {
    Point { x, y: 0 } => {
      println!("{}" , x);
    },
    Point { x, y } => {
      println!("{} {}" , x, y);
    },
  }

  enum Shape {
    Rectangle { width: i32, height: i32 },
    Circle(i32),
  }

  let shape = Shape::Circle(10);

  match shape {
    Shape::Rectangle { x, y } => //...
    Shape::Circle(radius) => //...
  }
```

3.3 Ignoring values

```
struct SemVer(i32, i32, i32);

let version = SemVer(1, 32, 2);

match version {
  SemVer(major, _, _) => {
    println!("{}", major);
  }
}

let numbers = (2, 4, 8, 16, 32);

match numbers {
  (first, .., last) => {
    println!("{}, {}", first, last);
  }
}
```

3.4 Match guards

```
  let num = Some(4);

  match num {
    Some(x) if x < 5 => println!("less than five: {}", x),
    Some(x) => println!("{}", x),
    None => (),
  }
```

3.5 @ bindings

Bind value to a name

```
match beaufort() {
  v @ 0..1    => println!("Calm : {} km/h", v),
  v @ 1..=5   => println!("Light Air : {} km/h", v),
  v @ 5..=11  => println!("Light Breeze : {} km/h", v),
  v @ 11..=19 => println!("Gentle Breeze : {} km/h", v)
}
```

4 Option

A function that may fail might enclose its return value in an **Option** enum, to notify wheter the action was successful.

```
fn sqrt(v: f64) -> Option<(f64, f64)> {
  if v < 0.0 {
    return None;
  }

  let sqrt = v.sqrt();
  Some((sqrt, -sqrt))
}
```

5 Result

5.1 Definition

The **Result** enum is similar to **Option** but it specifies why the function has failed.

When the function doesn't really need to return anything other than the **Result** status, **()** can be used.

```
enum ErrorType {
  NegativeBase,
  NegativeArgument,
  BaseOne
}

fn log(base: f64, arg: f64) -> Result<f64, ErrorType> {
  if base <= 0.0 {
    return Err(ErrorType::NegativeBase);
  }

  if base == 1.0 {
    return Err(ErrorType::BaseOne);
  }

  if arg <= 0.0 {
    return Err(ErrorType::NegativeArgument);
  }

  let result = arg.log(base);
  Ok(result)
}
```

5.2 ? operator

The **?** operator is syntax sugar for **Result** handling.
This operator can be placed at the end of a **Result** type. If the result is an error, the functions returns it, otherwise unwraps its value.

```
fn log(base: f64, arg: f64) -> Result<f64, ErrorType> { ... }

fn something() -> Result<f64, ErrorType> {
  let v = match log(2.718, 3.14) {
    Ok(v) => v,
    Err(e) => return Err(e)
  };

  // use `v`
}
```

can be written as

```
fn log(base: f64, arg: f64) -> Result<f64, ErrorType> { ... }

fn something() -> Result<f64, ErrorType> {
  let v = log(2.718, 3.14)?;

  // use `v`
}
```