

## Langages systèmes 2 - Introduction to Rust

---

Raphael Amiard - Adacore  
amiard@adacore.com

## Introduction to Rust

---

- 2006
- Personal project by Graydon Hoare (working @ Mozilla at the time)
- No specification, instead semantics are based on implementation
- Language changed *a lot* between 2006 and 2015 (and is still changing a lot by other languages' standards)
- Nowadays, maintained and evolved by the Rust foundation

- Safer alternative to C/C++ for systems programming
- Many inspirations, including ML family languages, C++
- Focus on safety, albeit with a different perspective when compared to Ada (memory safety being the most valued kind of safety)

- Use of Rust is spreading like wildfire
- Projects like Android, Linux
- Companies like Google, Amazon
- Well positioned to become a credible alternative to C++, and maybe even C
- Big list of industrial users here:  
<https://www.rust-lang.org/production/users>

- Rust making forays into the critical markets. Big players are assessing the use of Rust in their codebases.
- But lacking industrial support for now
- Will probably become mainstream in the coming decade

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

## Procedural language

---



## First, a note about philosophy

- In C/C++, very weak distinction between statements and expressions
  - You can use exprs as statements
- In Ada, strong distinction between statements and expressions
  - Statements are statements, expressions are expressions, not interchangeable
  - Procedures and functions are distinct
- In Rust, everything is an expression (and you generally cannot ignore their value)
  - Simpler than Ada, (much) safer than C/C++
  - But not always obvious what an expression returns
  - Complex type system tricks to make it work (what's the type of a loop?)

## For loops

```
fn main() {  
  for i in 1..10 {  
    //   ^ Range object (of type Range)  
    println!("Hello, World!");  
  }  
}
```

## While loops

```
fn main() {  
    let mut i = 1;  
    // ^ Declare a mutable variable (immutable by default)  
  
    // No parens around condition  
    while i < 10 {  
        println!("Hello, World!");  
        i += 1; // increment  
    }  
}
```

# Infinite loops

```
fn main() {  
    let mut i = 1;  
  
    loop {  
        println!("Hello, World!");  
        i += 1; // increment  
  
        if i == 5 {  
            // ^ equality operator  
            break;  
        }  
    }  
}
```

## Infinite loop w. return value

```
fn main() {  
    let mut i = 1;  
  
    let mut a = 0;  
    let mut b = 1;  
  
    let res = loop {  
        let c = a + b;  
        a = b;  
        b = c;  
        i += 1;  
        if i > 12 {  
            break a;  
        }  
    };  
    println!("{}", res);  
}
```

```
fn main() {  
  let mut i = 1;  
  loop {  
    if i == 5 || else i == 12 {  
      break;  
    } else if i < 5 && i > 2 {  
      println!("I = 3 or 4");  
    } else {  
      println!("Hello, World!");  
    }  
  }  
}
```

## If/else as an expression

```
fn main() {  
    let number = if true { 5 } else { 6 };  
  
    let error = if true { 5 } else { "six" };  
}
```

## Match expression

```
fn main() {  
  let mut i = 1;  
  
  loop {  
    match i {  
      5 | 12 => break,  
      1..=4 => println!("i in 1..4"),  
      7 | 9 => break,  
      _ => println!("Hello, World!")  
    }  
  
    i += 1;  
  }  
}
```



## Quizz

---

## Quizz 1: Is there a compilation error?

```
fn main() {  
    let a = loop {  
        println!("Pouet");  
    };  
  
    let b: u32 = a;  
}
```

## Quizz 2: Is there a compilation error?

```
fn main() {  
    let a = for n in 1..11 {  
        println!("Pouet");  
    };  
}
```

## Quizz 3: Is there a compilation error?

```
fn main() {  
    let a = for n in 1..11 {  
        println!("Pouet");  
    };  
  
    let b: u32 = a;  
}
```

## Quizz 4: Is there a compilation error?

```
fn main() {  
    let mut i = 1;  
  
    let a = loop {  
        println!("Pouet");  
  
        if i > 12 { break; }  
  
        i +=1;  
    };  
  
    let b: u32 = a;  
}
```

## Quizz 5: Is there a compilation error?

```
fn main() {  
    let mut i = 1;  
    loop {  
        println!(  
            "{}",  
            if i == 5 || i == 12 { "5 or 12" }  
            else { "everything else" }  
        );  
  
        i += 1;  
    };  
}
```

## Quizz 6: Is there a compilation error?

```
fn main() {  
    let mut i = 1;  
  
    loop {  
        println!(  
            "{}",  
            if i == 5 || i == 12 { "5 or 12" }  
            else if i == 15 { "15" }  
        );  
  
        i += 1;  
    };  
}
```

## Quizz 7: Is there a compilation error?

```
fn main() {  
    let mut i = 100;  
  
    while i {  
        i -= 1;  
  
        println!("{}", i);  
    }  
}
```



## Quizz 8: Is there a compilation error?

```
fn main() {  
  let mut i = 1;  
  
  loop {  
    match i {  
      1..=5 => println!("i in 1..=5"),  
      // ^ This is a PATTERN  
      5 | 12 => break,  
      7 | 9 => break,  
    }  
  
    i += 1;  
  }  
}
```

## Types

---

- Set of built-in types:
  - Integer types: i8, i16, i32, i64, i128
  - Unsigned types: u8, u16, u32, u64, u128
- No way to define custom integer types
- Statically/strongly typed
- Two floating point types: f32 and f64

- Boolean: Named `bool`, either `true` or `false`. Not an enum!
- Character: Named `char`, can be any valid Unicode value.
- All in all, less powerful than Ada, but also much simpler.

- In debug builds: raises an error
- In release builds: wrap around
- Heritage of C++'s zero-cost abstraction mentality

# Tuple type

- Most basic composite type
- Anonymous collection of elements.
- Structurally typed

```
fn main() {  
    let tp = (1, 2)  
    // ^ Type of this is (i32, i32)  
  
    let (x, y) = tp;  
    // ^ This is an irrefutable pattern  
  
    let f = tp.1;  
    // Access first value of tuple  
}
```

## Array type

- Homogeneous array type
- Index type is usize
- Bounds checked
- Very simple (dare I say primitive). No variable length arrays at all.
- 90% of the time one will use vectors

```
fn main() {  
    let a = [1, 2, 3, 4, 5];  
  
    println!("{}", a[4]);  
}
```

# Vectors

- As we said before, arrays in Rust are mostly useless
- In most cases you'll want to use vectors (`Vec<T>`)
- Vectors can be variable size, and are growable, *but*, they're always heap allocated

```
fn main() {  
    let mut a = [1, 2, 3, 4].to_vec();  
    //           ^ Transform an array or slice into a vector  
  
    let b = vec![1, 2, 3, 4];  
    // Same thing as above  
  
    let c = vec![1; 100];  
    // Vector of 100 elements, all "1"  
  
    println!("{:?}", a);  
    //           ^ Print vector via the Debug trait  
    //           If you can't print something, try this  
  
    a.push(5);  
    println!("{:?}", a);  
}
```



- Slices are a bit like arrays, but they just a view into a sequence. The type is written `[T]`, but is not used directly, but rather through pointers.

```
fn main() {  
    let a = [1, 2, 3, 4, 5, 6, 7];  
    let mut v = vec![1, 2, 3, 4, 5, 6, 7];  
  
    let b = &a[1 .. 3];  
    //      ^ Reference to a view of items 1 to 3 of the array a  
  
    let c = &v[3 .. 5];  
    //      ^ Reference to a view of items 3 to 5 of the vec v  
  
    println!("{:?}", c);  
    // By some ownership magic, after this statement, the lifetime of the  
    // reference c is over  
  
    v.clear();  
  
    println!("{:?}", b);  
}
```

There are two main string types in Rust

- `String` is similar to a `Vec<u8>`, except:
  - It always points to a valid utf-8 sequence
  - You cannot index it
- `str` is a slice type. It is always used through a reference (`&str`)
- An array of characters is *not* a `String`

```
fn main() {  
    let message: &str = "Hello world";  
  
    for c in message.chars() {  
        print!("{}", c);  
    }  
    println!("");  
}
```

## Quizz 1: Is there a compilation error?

```
fn main() {  
    let i: (i32, i32) = [1, 2];  
}
```

## Quizz 2: Is there a compilation error?

```
fn main() {  
    let i = [1, 2, 3, 4, 5.0];  
}
```

## Quizz 3: Is there a compilation error?

```
fn main() {  
    let i: [i32; 5] = [1, 2, 3, 4, 5];  
}
```

## Quizz 4: Is there a compilation error?

```
fn main() {  
    let i: [i32] = [1, 2, 3, 4, 5];  
}
```

## Quizz 5: Is there a compilation error?

```
fn main() {  
    let n: int = 5;  
    let i: [i32; n] = [1, 2, 3, 4, 5];  
}
```

## Quizz 6: Is there a compilation error?

```
fn main() {  
    let a = [1, 2, 3, 4, 5];  
  
    println!("{}", a[10]);  
}
```



## Quizz 7: Is there a compilation error?

```
fn main() {  
    let s: String = "Hai";  
    println!("{}", s);  
}
```

## Quizz 7: Is there a compilation error?

```
fn main() {  
    let s: &str = "Hai";  
    let s2: &str = &s[0..2];  
    println!("{}", s);  
}
```

- Main is always called `main`
- You can put other functions at the top-level in your main source file
- Order doesn't matter

```
fn main() {  
    println!("Pouet");  
    other_function();  
}  
  
fn other_function() {  
    println!("Pouet2");  
}
```

## Functions (2)

- Functions contain a (possibly empty) sequence of statements, followed by an optional expression
- Expression is used as the return value
- An expression followed by a semicolon *is a statement*

```
fn fib() -> i32 {  
    let mut i = 1;  
  
    let mut a = 0;  
    let mut b = 1;  
  
    loop {  
        let c = a + b;  
        a = b;  
        b = c;  
        i += 1;  
        if i > 12 {  
            break a;  
        }  
    }  
}
```

```
fn double(v: Vec<i32>) -> Vec<i32> {
    v.iter().map(|i| i * 2).collect()
    //           ^ Lambda function
    //           ^ Convert back to a vector
}

fn main() {
    let v: Vec<i32> = vec![1, 2, 3, 4];
    println!("{:?}", double(v));

    println!("{:?}", v); // :(
}
```

- Defining concept of Rust. Academic concept: Linear/Affine types
- By default, a value cannot be copied, only moved
- If you want to use it you either move it (as in the above example) or *borrow* it
- Two types of borrows: Mutable (only one at a time), and immutable (N at a time)

```
fn double(v: &Vec<i32>) -> Vec<i32> {  
    v.iter().map(|i| i * 2).collect()  
}  
  
fn main() {  
    let v: Vec<i32> = vec![1, 2, 3, 4];  
    println!("{:?}", double(&v));  
  
    println!("{:?}", v); // :(  
}
```

```
fn main() {  
    let mut v: Vec<i32> = vec![1, 2, 3, 4];  
    let v2 = &mut v[1..3];  
    v2[1] = 13;  
    println!("{:?}", v);  
}
```

- In many case you want to manipulate your data by reference but you can't use references
- In those cases you want to use a managed pointer type: either Box (owned) or Rc (shared).
- More details in next class



## Quizz 1: Is there a compilation error?

```
fn factorial(n: i64) -> i64 {  
    let mut ret = n;  
  
    for i in 1..n {  
        ret = ret * n;  
    }  
  
    ret;  
}
```

## Quizz 2: Is there a compilation error?

```
fn double(v: &mut Vec<i32>) {  
    for i in 0..v.len() {  
        v[i] = v[i] * 2;  
    }  
}  
  
fn main() {  
    let v: Vec<i32> = vec![1, 2, 3, 4];  
    double(&v);  
  
    println!("{:?}", v); // :(  
}
```

## Quiz 3: Is there a compilation error?

```
fn double(v: &mut Vec<i32>) {  
    for i in 0..v.len() {  
        v[i] = v[i] * 2;  
    }  
}  
  
fn main() {  
    let mut v: Vec<i32> = vec![1, 2, 3, 4];  
    double(&v);  
  
    println!("{:?}", v); // :(  
}
```

## Quiz 4: Is there a compilation error?

```
fn double(v: &mut Vec<i32>) {  
    for i in 0..v.len() {  
        v[i] = v[i] * 2;  
    }  
}  
  
fn main() {  
    let mut v: Vec<i32> = vec![1, 2, 3, 4];  
  
    let v2 = &mut v;  
    double(v2);  
  
    let v3 = &mut v;  
    double(v3);  
  
    println!("{:?}", v); // :(  
}
```

## Quiz 5: Is there a compilation error?

```
fn double(v: &mut Vec<i32>) {  
    for i in 0..v.len() {  
        v[i] = v[i] * 2;  
    }  
}  
  
fn main() {  
    let mut v: Vec<i32> = vec![1, 2, 3, 4];  
  
    let v2 = &mut v;  
    double(v2);  
  
    let v3 = &mut v;  
    double(v3);  
  
    println!("{:?}", v2); // :(  
}
```

# Structs

```
#[derive(Debug)]
// Magic that allows you to print structs
struct Point {
    x: i32,
    // Component of the struct
    y: i32
}

fn main() {
    let p = Point { x: 12, y: 12 };
    println!("{:?}", p);

    println!("{}", p.x);
    //           ^ Access the field x

    // You can define mutable structs
    let mut p2 = Point { x: 12, y: 12 };

    // You can mutate fields of structs via dot notation
    p2.x = 15;

    println!("{:?}", p2);
}
```

- Rust is not strictly an OOP language
- No inheritance
- No encapsulation
- BUT: You have method syntax :D

## Structs: methods

```
#[derive(Debug)]
struct Point {
    x: i32, y: i32
}

impl Point {
    fn invert(self: &Point) -> Point {
        Point {x: self.y, y: self.x}
    }

    fn double(&mut self) {
        // ^ Alias for self: &mut Point
        self.x = self.x * 2;
        self.y = self.y * 2;
    }
}

fn main() {
    let mut p = Point {x: 1, y: 2};
    p.double();

    println!("{:?}", p);
    println!("{:?}", p.invert());
}
```



- Enums in Rust are very powerful
- Akin to sum types in functional languages
- But can also be used to model simple stuff
- Can also have methods, like structs!

```
enum Color {  
    Yellow, Red, Green, Blue  
}  
  
fn main() {  
    let y = Color::Yellow;  
  
    match y {  
        Color::Yellow => println!("yellow!"),  
        Color::Red    => println!("red!");  
        _              => println!("other color!");  
    }  
}
```

## Complex enums (1/2)

```
#[derive(Debug)]
enum Operator {
    Plus, Minus, Divide, Multiply
}

#[derive(Debug)]
enum Expr {
    BinOp {
        l: Box<Expr>,
        op: Operator,
        r: Box<Expr>
    },
    Literal(i32)
}
```

## Complex enums (2/2)

```
fn main() {  
  let e =  
    Expr::BinOp {  
      l: Box::new(  
        Expr::BinOp {  
          l: Box::new(Expr::Literal(12)),  
          op: Operator::Plus,  
          r: Box::new(Expr::Literal(15))  
        },  
        op: Operator::Plus,  
        r: Box::new(Expr::Literal(12))  
      ),  
    };  
  
  println!("{:?}", e);  
}
```