

Taller de Programación I

Introducción a Rust

Ing. Pablo A. Deymonnaz
pdeymon@fi.uba.ar

Facultad de Ingeniería
Universidad de Buenos Aires



1. Introducción a Rust

Qué es Rust? Por qué Rust?
cargo

2. Sintaxis

3. Organización del código

Qué es Rust? Por qué Rust?

- ▶ Objetivos de Rust: velocidad, seguridad y concurrencia.
- ▶ Se siente como un lenguaje expresivo de alto nivel, mientras que alcanza muy alta performance.
- ▶ Adhiere al principio de “zero-cost abstractions”.
- ▶ Palabras de la comunidad: “empoderar a todos”, extremadamente rápido, concurrencia sin miedo, retrocompatibilidad.

Qué es Rust? Por qué Rust?

Funcionalidades que lo distinguen: **seguridad**

- ▶ Prevenir el acceso a datos/memoria inválida, en tiempo de compilación (memory safe sin costo en tiempo de ejecución) (*buffer overflow*).
- ▶ No existen los “punteros sueltos” (*dangling pointers*), las referencias a datos inválidos son descartadas.
- ▶ Previene condiciones de carrera sobre los datos al usar concurrencia.

productividad: funcionalidades ergonómicas para el desarrollador, ayudas del compilador, tipos de datos sofisticados, pattern matching, etc. **MANTENIBILIDAD.**

cargo

El compilador se llama **rustc** y lo acompaña una familia de herramientas. **cargo** es el gestor de paquetes y el sistema de construcción (similar a *make*)

Algunos comandos:

- ▶ Crear un nuevo proyecto: **cargo new**
- ▶ Crear un nuevo proyecto en un directorio existente: **cargo init**
- ▶ Compilar el proyecto: **cargo build**
- ▶ Compilar el proyecto en modo release: **cargo build --release**
- ▶ Ejecutar el proyecto: **cargo run**
- ▶ Ejecutar los tests: **cargo test**
- ▶ Generar la documentación HTML: **cargo doc**
- ▶ Analizar el proyecto, sin compilar: **cargo check**
- ▶ Formatear el código: **cargo fmt**
- ▶ linter: **cargo clippy**

1. Introducción a Rust

2. Sintaxis

- Tipos de datos

- Variables

- Estructuras

- Enums

- Estructuras de control

- Traits

3. Organización del código

Tipos de datos

Tipos numéricos:

- ▶ i8, i16, i32, i64
- ▶ u8, u16, u32, u64
- ▶ f32, f64
- ▶ usize, isize

Otros tipos nativos:

- ▶ bool
- ▶ char (Unicode)

Arrays: [u8; 3]

Tuplas: (char, u8, i32)

Tipos dinámicos: Vec<T>, String

Variables

Las variables son inmutables por default. Las llamamos *bindings*.

```
let t = true;
```

Para hacerlas mutables, usamos **mut**.

```
let mut punto = (1_u8, 2_u8);
```

shadow: se puede reutilizar el nombre de la variable, se descarta el valor anterior.

Rust implementa inferencia de tipos.

```
fn retornarentero() -> i32 {  
    42  
}
```

```
let v = retornarentero();
```


La sintaxis de las funciones es (la última línea, sin ; y sin *return* es el valor de retorno):

```
fn sumar_uno(a: i32) -> i32 {  
    a + 1  
}
```

Son un conjuntos de elementos puestos juntos, que son tratados como una unidad. Se le pueden definir operaciones (métodos).

Variante 1: estructura con nombres de campos

```
struct Persona {  
    nombre: String ,  
    apellido: String ,  
}
```

Estructuras (cont.)

Variante 2: estructura tipo tupla

```
struct Celsius(f32);  
struct Fahrenheit(f32);
```

permiten representar cosas como:

```
fn convertir_a_fahrenheit(temp: Celsius)  
  -> Fahrenheit {  
  ...  
}
```

Variante 3: estructura tipo *unit*

```
struct AlgunaCosa;
```

Variante 1: enumeración de ítems

```
enum Palo {  
    Oro ,  
    Copa ,  
    Espada ,  
    Basto ,  
}
```

```
let palo: Palo = Palo::Oro;
```

Enums - tagged union

Variante 2: Tipo tupla

```
enum Carta {  
    Comodin ,  
    Oro(u8) ,  
    Copa(u8) ,  
    Espada(u8) ,  
    Basto(u8) ,  
}  
let mi_carta: Carta = Carta::Oro(9);  
let mi_comodin: Carta = Carta::Comodin;
```

Enums - tagged union

Variante 3: Tipo estructura

```
enum Jugada {  
    Carta { palo: Palo, numero: u8 },  
    Paso,  
}  
  
let mi_jugada: Jugada =  
    Jugada::Carta { palo: Palo::Copa, numero: 6};
```

Option

Un elemento que puede contener algún valor o nada / None.

T es un *generic*, puede ser cualquier tipo de valor.

```
enum Option<T> {  
    Some(T),  
    None,  
}
```

```
// ——  
fn dividir(num: f64, den: f64) -> Option<f64> {  
    if den == 0.0 {  
        None  
    } else {  
        Some(num / den)  
    }  
}
```

Result

Representar el estado de error.

T es un *generic*, puede ser cualquier tipo de valor.

E es un *generic* que puede ser cualquier error.

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

```
// -----  
fn contar_lineas_de_archivo(path: &str)  
    -> Result<u64, String> {  
    // ....
```

```
Ok(42)  
}
```


Operaciones útiles

- ▶ operador ?
- ▶ *unwrap()*, *unwrap_or(42)*
- ▶ *expect("...")*
- ▶ *is_some()*, *is_none()*, *is_ok()*, *is_err()*

match

match provee *pattern matching* (similar al switch de C), se ejecuta la primera rama que cumple la condición.

Ejemplo:

```
...  
let mi_carta: Carta = Carta::Oro(9);  
  
match mi_carta {  
    Carta::Oro(valor) =>  
        println!("La carta es Oro de valor {}",  
            valor),  
    _ => {}  
}
```

match (cont.)

...

```
let punto: (i32, i32, i32) = (0, -2, 3);
```

```
match punto {  
  (0, y, z) =>  
    println!("En el eje x. y:{}", z: {} ", y, z),  
  (_, 0, _) =>  
    println!("El punto esta sobre el eje y"),  
  _ => {},  
}
```

```
let number = Some(42)

if let Some(i) = number {
    println!("valor {}", i);
}
```

loop

loop permite indicar un loop infinito.

```
loop {  
    count += 1;  
    if count == 3 {  
        println!("tres");  
        // salteamos el resto  
        // de la iteracion  
        continue;  
    }  
    println!("{}", count);  
  
    if count == 5000 {  
        println!("Listo");  
        break; // salir del loop  
    }  
}
```

for permite iterar sobre un *iterador*.

```
for n in 1..101 {  
  if n % 3 == 0 {  
    println!("multiplo de 3");  
  } else if n % 4 == 0 {  
    println!("multiplo de 4, pero no de 3");  
  } else {  
    println!("{}", n);  
  }  
}
```

for (cont.)

for iterando sobre un iterador:

```
let mut nombres = vec!["Juan", "Pedro", "Felipe"];

for nombre in nombres.iter() {
    println!("{}", nombre);
}
```

while realiza un bucle mientras la condición sea *true*.

while let realiza un bucle mientras la condición de *if let* sea verdadera.

```
while let Some(i) = opcional {
    // ...
}
```

Impls

impl se usa para definir métodos a enums y structs.

```
struct Persona {  
    ...  
}  
  
impl Persona {  
    fn nombre_completo(&self) -> String {  
        format!("{}", self.nombre, self.apellido)  
    }  
}  
  
let nombre = p.nombre_completo();
```


Impls - funciones asociadas

En otros lenguajes, se llaman funciones estáticas.

```
impl Persona {  
  fn new(first_name: String , last_name: String)  
    -> Persona {  
    Persona {  
      first_name: first_name ,  
      last_name: last_name ,  
    }  
  }  
}
```

```
let p = Persona::new("Pedro".to_string() ,  
  "Arce".to_string());
```

Traits

Traits permiten definir comportamiento común a las estructuras, similar a las interfaces en otros lenguajes.

Representan una *capacidad*, algo que el tipo de dato puede hacer.

```
trait NombreCompleto {  
    fn nombre_completo(&self) -> String;  
}  
  
impl NombreCompleto for Persona {  
    fn nombre_completo(&self) -> String {  
        format!("{}", self.nombre, self.apellido)  
    }  
}
```

Traits y generics

Permiten realizar polimorfismo, aceptando tipos de datos probablemente desconocidos que implementen Traits.

Ejemplo:

```
fn mostrar_nombre<W: NombreCompleto>(v: W) {  
  
}
```

Traits utilitarios

Existen Traits utilitarios, definidos en la std.

Default permite definir una construcción con valores default.

```
impl std::default::Default for Persona {  
    fn default() -> Persona {  
        nombre: "".string(),  
        apellido: "".string(),  
    }  
}
```

Se puede utilizar una implementación default del Trait:

```
#[derive(Default, Debug)]
```

From e **Into** permiten implementar la conversión de un tipo de dato a otro.

```
impl From<String> for Persona {  
    fn from(s: String) -> Persona {  
        Persona {  
            . . .  
        }  
    }  
}
```

1. Introducción a Rust
2. Sintaxis
3. Organización del código
 - Módulos
 - Testing

Módulos

Permiten organizar el código en espacios de nombre.

En el mismo archivo:

```
fn main() {  
    saludos :: hola();  
}  
  
mod saludos {  
    // por default, todo lo que esta dentro de  
    // un modulo es privado  
    pub fn hola() {  
        println!("Hola mundo!");  
    }  
}
```

self se refiere al mismo módulo y **super** al padre.

Módulos (cont.)

En otro archivo del mismo directorio

```
// main.rs  
mod saludos; // importar el modulo
```

```
fn main() {  
    saludos::hola();  
}
```

```
// saludos.rs  
// la funcion debe ser publica para ser  
// accedida desde otro modulo  
pub fn hola() {  
    println!("Hola mundo!");  
}
```


Módulos (cont.)

En otro archivo de en un subdirectorio

```
// main.rs
mod phrases;
fn main() {
    frases::saludos::hola();
}
```

```
// —
// frases/mod.rs
pub mod saludos;
```

```
// —
// frases/saludos.rs
pub fn hola() {
    println!("Hola Mundo!");
}
```

Importar crates externos.

En el archivo **Cargo.toml**:

```
[dependencies]  
log = "0.4"
```

Para usarlo:

```
use log::info;
```

```
...
```

```
info!("logueando evento!");
```

Testing

- ▶ Los test son funciones de Rust que verifican que el resto del código funciona de la manera esperada.
- ▶ En el cuerpo del test se hace:
 - ▶ Setup de los datos y estado necesarios para el test.
 - ▶ Ejecutar el código que se quiere testear.
 - ▶ Afirmar (*assert*) los resultados esperados.

Se identifica a las funciones test con una anotación antes de la línea **fn**:

```
#[test]
```

Para realizar las afirmaciones, se utilizan las macros de la familia *assert*.

```
assert!(persona.edad > 18);
```

Testing

- ▶ Los test se organizan en un módulo test:

```
#[cfg(test)]  
mod tests {  
    ...  
}
```

- ▶ Tests de integración
 - ▶ Se colocan en el directorio **tests/**, al lado de **src/**.
 - ▶ Se compila cada archivo como un crate separado. Debemos incluir como crate nuestro código.
 - ▶ No pueden testear funciones de **src/main.rs**.

- ▶ **The Rust Programming Language**,
<https://doc.rust-lang.org/book/>
- ▶ **Programming Rust: Fast, Safe Systems Development**,
1st Edition, Jim Blandy, Jason Orendorff. 2017.
- ▶ **Rust in Action**, Tim McNamara, Manning. 2020.