.UBAfiuba
FACULTAD DE INGENIERÍA

# Técnicas de Programación Concurrente I

## Introducción a Rust

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

Facultad de Ingeniería
Universidad de Buenos Aires

# Índice

# 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**

# Índice

# 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();
```

# Funciones

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
}
```

# Estructuras

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,
}
```

# Enums

**Variante 1:** enumeración de ítems

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

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

# 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)
}
```

.UBAfiuba FACULTAD DE INGENIERÍA

# Índice

# Ownership

**Motivación:** eliminar clases enteras de errores (punteros nulos, uso posterior a liberación, doble liberación, saturación de búfer (*buffer overflow*), invalidación de iterador, carreras de datos) al restringir programas válidos, sin incurrir en gastos generales de ejecución.
**Inspiración:** C ++ RAII

1. Un recurso se inicializa cuando se asigna.
2. La inicialización la realizan los constructores de clases.
3. La limpieza se realiza mediante destructores de clases.
4. Los destructores se llaman automáticamente cuando el objeto sale de su alcance.

# Ownership

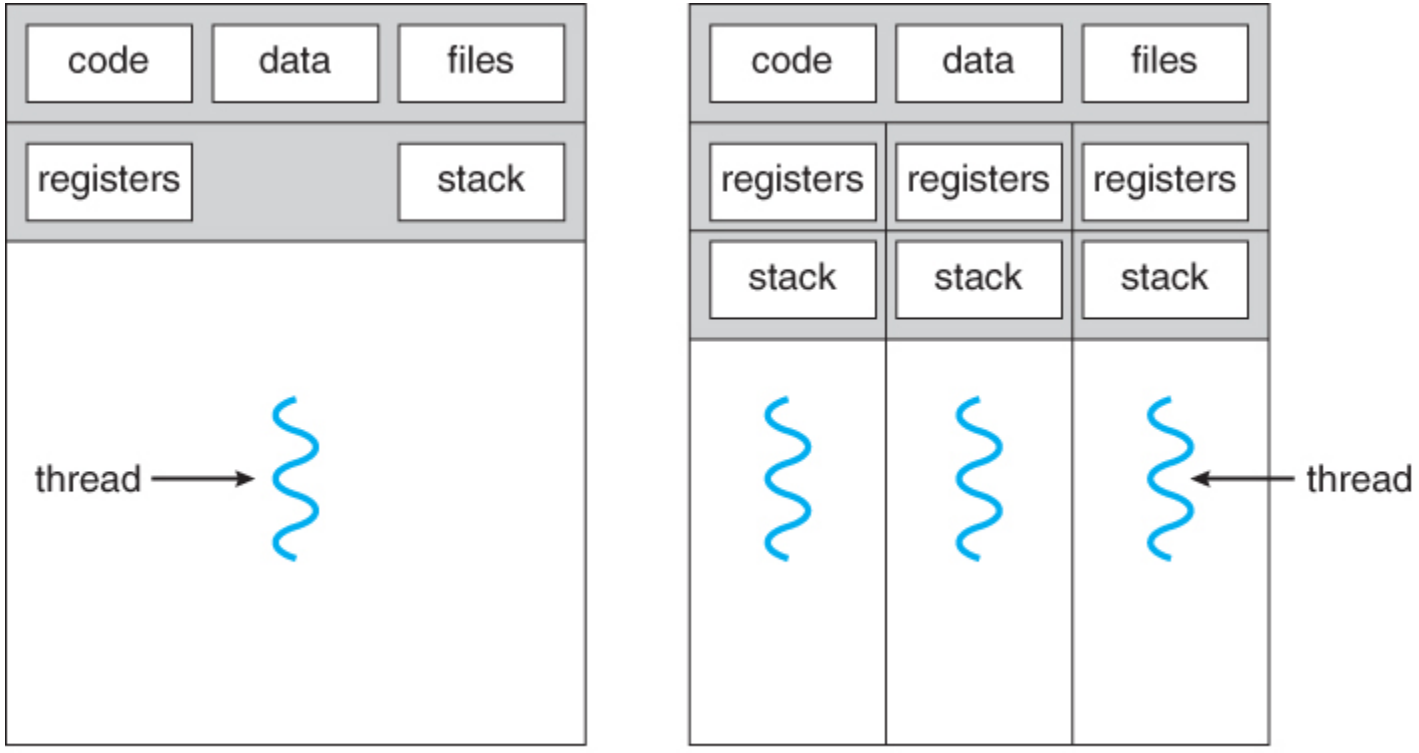
1. Cada valor en Rust tiene una variable, que es llamada su dueño (*owner*).
2. Puede haber un único dueño a la vez.
3. Cuando el dueño sale fuera de su scope, el valor es eliminado (*dropped*).

.UBAfiuba FACULTAD DE INGENIERÍA

# Índice

# Threads Introducción

Los Threads comparten los recursos del proceso, entre ellos, el espacio de memoria.
Cada thread mantiene su propia información de estado (stack, PC, registros).



# Threads en Rust (I)

Rust permite crear Threads del SO, utilizando la función `thread::spawn()`.
Recibe como parámetro un *moving closure*.
Retorna el handle del thread (*JoinHandle*).

```
let join_handle: thread::JoinHandle<_> =
    thread::spawn(|| {
  // codigo del thread hijo
});
```

Este mecanismo, basado en el sistema de tipos y ownership, asegura que no hay condiciones de carrera sobre los datos, se verifica en tiempo de compilación.
Los threads son programados por el shceduler del SO.

# Threads en Rust (II)

El thread padre puede esperar a que un thread hijo creado finalice utilizando la función join().

```
pub fn join(self) -> Result<T>
```

Se invoca sobre el handle del thread obtenido con spawn:

```
child.join();
```

# Threads en Rust (III) - Ejemplo

```
use std::thread;
static NTHREADS: i32 = 10;

// main thread
fn main() {
    // Vector para almacenar handles de los threads
    let mut children = vec![];

    for i in 0..NTHREADS {
        // Crear otro thread
        children.push(thread::spawn(move || {
            println!("this is thread number {}", i);
        }));
    }

...
```

# Threads en Rust (III) - Ejemplo

```
...
    for child in children {
            // Esperar que terminen los threads
            let _ = child.join();
    }
}
```

# Threads en Rust (IV) - Otras funciones

- `thread::sleep();`
  Suspende la ejecución del thread durante el tiempo dado. El tiempo se especifica con funciones de std::time::Duration, por ejemplo:

```
thread::sleep(Duration::from_millis(1));
```

- `thread::yield_now();`
  Cede el timeslice al scheduler del SO.

# Threads en Rust (V) - Otras características

- Los threads pueden tener un nombre: Sirve para identificar un thread que ejecuta `panic!`. El nombre puede obtenerse y usarse.
  Se crea el Thread con `thread::Builder`.

```
let builder = thread::Builder::new()
    .name("mi_hilo".into());

let handler = builder.spawn(|| {
  assert_eq!(thread::current().name(),
    Some("hilo"))
}).unwrap();
```

# Threads en Rust (V) - Otras características

- También se puede configurar el tamaño del stack del thread (`stack_size()`).

# Bibliografía


- **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.