>>> Introducción a Rust

>>> t3chfest 2023

Martín Pozo

25 de febrero

Presentación

>>> Presentación

- Doctorando en Inteligencia Artificial
- Casi 10 años de experiencia profesional
  - Desarrollador
  - Administrador de sistemas
- Programando en Rust proyectos FOSS desde 2017

Instalación y configuración del entorno

>>> rustup

- `rustup` permite instalar distintos canales de Rust (estable, beta y *nightly*) y sus componentes en *toolchains*
- Seguir los pasos de `https://rustup.rs/` para instalar `rustup` en el equipo
- Más información sobre `rustup` en el libro oficial
  - `https://rust-lang.github.io/rustup/`

>>> Instalación de componentes de desarrollo en la rama principal

```
$ rustup component add clippy rust-analyzer rust-docs rustfmt
```

- Este comando instala los siguientes componentes (algunos quizás ya instalados):
  - `clippy`: consejos para detectar errores comunes y mejorar el código
  - `rust-analyzer`: LSP (*Language Server Protocol*) recomendado
  - `rust-docs`: documentación sobre la biblioteca estándar y el lenguaje
  - `rustfmt`: da formato al código siguiendo las reglas de estilo oficiales

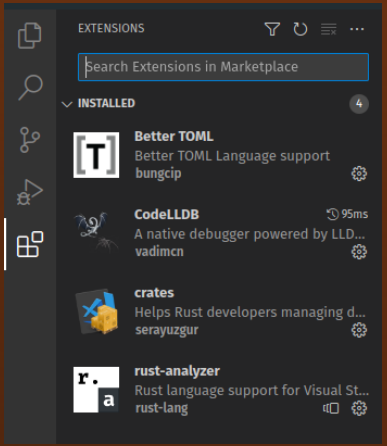
>>> Creación de un proyecto (*crate*) nuevo

$ cargo new --bin <nombre_del_proyecto>

- Este comando crea una carpeta nueva con los siguientes elementos:
  - `.git`: se ha creado un repositorio de Git nuevo
  - `.gitignore`: se excluye del repositorio la carpeta `target` (binarios)
  - `Cargo.toml`: configuración del proyecto
  - `src/main.rs`: módulo principal implementando un 'Hello World'
- Si en lugar de `--bin` se usa la opción `--lib` crea un módulo `lib.rs`
- El proyecto se compila con `cargo build` y se ejecuta con `cargo run`
  - El argumento `--release` optimiza la compilación en ambos comandos
  - También se puede ejecutar el binario dentro de `target/debug` o `target/release`
  - Otra forma es instalarlo en `~/.cargo/bin` con `cargo install --path .`

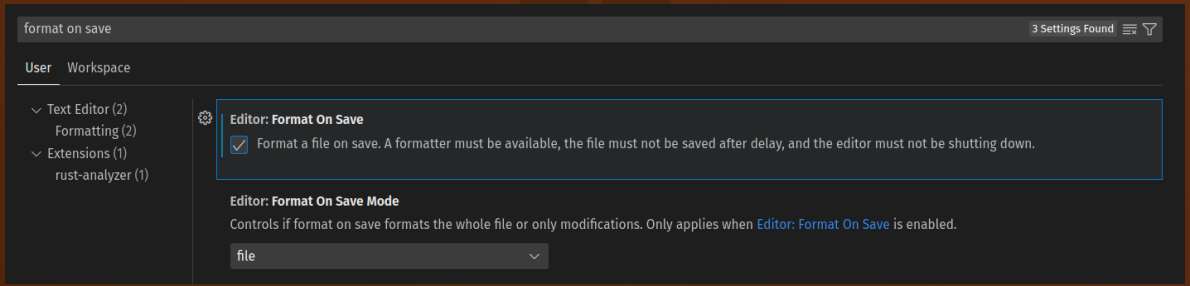
>>> Extensiones de VSCode/ium

- `rust-analyzer`
  - LSP, clippy y da formato al código con rustfmt
- `CodeLLDB`
  - Depuración (*debugging*) de código en Linux y Mac
- `Better TOML`
  - Soporte de ficheros de configuración TOML
- `crates`
  - Comprobación de que cada dependencia está en la última versión



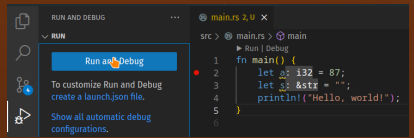
>>> Configuración

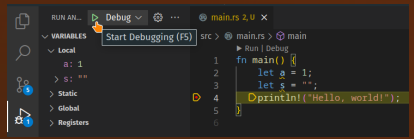
- `rust-analyzer` tiene muchas opciones pero los valores por defecto están bien
- La única opción que conviene modificar es la de dar formato al guardar:



>>> Depuración

1. Poner un punto de interrupción en el lugar deseado
2. Hacer clic en el botón de ejecución y aceptar la creación de un fichero `launch.json` usando el fichero `Cargo.toml`
3. Hacer clic en el botón de ejecución con el perfil de depuración





Fundamentos de Rust

>>> Introducción

- Rust es un lenguaje compilado con seguridad de memoria sin recolector de basura
  - La seguridad de memoria se garantiza mediante reglas de compilación
- Esto lo hace más difícil de aprender al principio pero tiene muchas ventajas:
  - Rendimiento en tiempo de ejecución
    - Además usa muchas abstracciones de coste cero
  - Concurrencia sin miedo
    - Estas reglas de compilación también evitan condiciones de carrera
  - Menos errores en tiempo de ejecución porque se detectan al compilar
    - Al compilar se detectan siempre, en ejecución solo cuando se cumplen ciertas condiciones

>>> Sintaxis básica

```rust
//! Esto es un comentario de documentación para el fichero

/// Recibe dos números enteros de 64 bits y devuelve otro
fn add(o1: i64, o2: i64) -> i64 {
    // Las expresiones que no acaban en `;` se retornan. Esto es equivalente a:
    // return o1 + o2;
    o1 + o2
}

/// Esto es un comentario de documentación para la función
fn main() {
    // Los tipos para enteros son (`u` para `unsigned` e `i` para enteros)
    // u8, u16, u32, u64, u128, usize
    // i8, i16, i32, i64, i128, isize
    // donde el número indica el tamaño en bits y `usize` e `isize` usan el tamaño de la arquitectura
    //
    // Los tipos para números de coma flotante son f32 y f64
    // El tipo se puede omitir cuando el compilador lo puede predecir
    // let result: i64 = add(1, 1);
    let result = add(1, 1);

    // Macro calls are ended with `!`
    println!("1 + 1 = {}", result);
}
```

>>> Tuplas y desestructuración de patrones

```rust
fn main() {
    let my_tuple: (u64, i64, f64, String) = (1, -1, 1.0, "hola".string());

    println!("{}", tuple.0); // 1
    println!("{}", tuple.3); // "hola"

    // Extracción de los valores de la tupla a variables
    // `_` se puede usar para ignorar valores
    let (one, minus_one, _, hola) = my_tuple;
}
```

>>> Strings

```rust
fn main() {
    let mut string1 = String::new();
    // Para combinar strings se puede usar el método `push_str` o la macro `format`
    string1.push_str("hola");
    let string3 = format!("{} {}", string1, "mundo");
    // Es importante diferenciar entre `String` y `&str`, esto es un &str
    let my_str = "Hola";
    // Se puede convertir en String con el método `to_string()` o con la función `String::from`
    let string5 = my_str.to_string();
    // Un str se puede extender en varias líneas usando `\` al final de la línea
    let multiline_str = "Esto es un string \
        de varias líneas";
    // `r`"" se usa para escribir contenido literal sin escapar
    let literal_str = r"Los saltos de línea se escriben como \n";
    // Se puede escribir contenido en bytes con `b""`
    let bytes = b"Hola";
    // `r#""#` se usa para cadenas literales
    let raw = r#"Esto es un string
literal"#;
    // Los strings en Rust son UTF-8, esto implica que no se puedan obtener caracteres directamente
    // porque algunos caracteres se representan con varios bytes, pero hay una función para iterar sobre chars
    for ch in raw.chars() {
        println!("{}", ch);
    }
}
```

>>> Vectores, *arrays* y *slices*

```rust
fn main() {
    // La macro `vec!` construye un `Vec` con los elementos indicados
    let vector1: Vec<i32> = vec![0, 5, 10, 100];
    // También se puede crear un vector vacío y añadir elementos después
    let mut vector2 = Vec::new();
    vector2.push(0);
    vector2.push(5);
    vector2.push(10);
    vector2.push(100);

    // Los 'arrays' son menos flexibles porque tienen un tamaño fijo
    let array: [i32; 4] = [0, 5, 10, 100];

    // Un 'slice' es una referencia a una sección contigua de una colección
    let slice: &[i32] = &vector1[1..3]; // &[5, 10]
}
```

>>> Estructuras

```rust
// Estructura básica con campos
struct WithFields {
    field1: u64,
    field2: String,
}
// Estructura sin datos (puede ser útil por reglas de compilación, por ejemplo una máquina de estados)
struct UnitStruct;
// Estructura tupla, los campos son accedidos con `.0`, `.1`, etc.
struct TupleStruct(u64, String);

fn main() {
    let with_fields = WithFields {
        field1: 10,
        field2: String::from("Ten"),
    };
    println!("{}", with_fields.field1);
    let with_fields_2 = WithFields {
        field1: 10,
        // Copia el valor de los otros campos de la otra estructura
        ..with_fields
    };
    let empty = UnitStruct;
    let tuple_struct = TupleStruct(10, String::from("Ten"));
    println!("{}", tuple_struct.0);
}
```

>>> Enumeraciones

```rust
// Las enumeraciones pueden tener variantes sin datos, con campos sin nombre tipo tupla o con campos con nombre
enum Enum1 {
    Variant1,
    Variant2(String, i32),
    Variant3 {
        name: String,
        number: i32,
    },
}

fn main() {
    let enum1: Enum1 = Enum1::Variant1;
    let enum2 = Enum1::Variant2("texto".to_string(), 10);
    let enum3 = Enum1::Variant3 { name: "texto".to_string(), number: 10 };
}
```

>>> Visibilidad y mutabilidad

```rust
// Por defecto todo es privado (accesible solo desde el propio módulo).
// Existen las visibilidades adicionales `pub` (desde cualquier sitio) y `pub(crate)` (desde cualquier módulo del paquete).

// La estructura es visible desde cualquier sitio pero algunos de sus campos no
pub struct VisibilityTest {
    private_field: i32,
    pub public_field: i32,
    pub(crate) public_in_crate_field: i32,
}

fn main() {
    // Las variables son inmutables por defecto, esto evita modificaciones indeseadas, se declaran mutables con `mut`

    // Este valor no se puede modificar
    let inmutable = 10;

    let mut mutable = 1;
    mutable += 10; // Ahora mutable vale 11
}
```

>>> Métodos

```rust
// Los métodos se implementan dentro de bloques `impl` y son válidos en estructuras y enumeraciones
struct StructWithMethods {
    field1: String,
    field2: i32,
}

impl StructWithMethods {
    // Las funciones de estructura no tienen parámetro `self`
    fn struct_fn() {
        println!("Esta función se invoca como StructWithMethods::struct_fn()");
    }

    // Los métodos pueden tener como primer parámetro `&self` para pasar el objeto como referencia,
    // `&mut self` para pasarlo como referencia y modificarlo o `self` para pasarlo totalmente
    fn my_unit_method(&self) -> i32 {
        self.field2 * 2
    }

    fn modificator_method(&mut self, new_value: i32) {
        self.field2 = new_value;
    }
}
```

>>> *Traits*

```rust
// Los 'traits' definen interfaces de funciones y cualquier estructura o enumeración puede implementarlos.
// Para implementar un 'trait' tienen que implementarse todas sus funciones.

// Rust no soporta herencia, en su lugar promueve la composición.

// Todas las funciones definidas por un 'trait' son públicas.

pub trait Interface {
    // Se pueden definir implementaciones por defecto para que esa función no sea necesario implementarla
    fn fn_default_impl(param1: i32) -> i32 {
        param1
    }
    fn another_fn(param: i32) -> i32;
}

struct MyStruct;
impl Interface for MyStruct {
    fn another_fn(param: i32) -> i32 {
        param * 2
    }
    // Esta implementación sobrescribe la implementación por defecto pero no es necesaria para implementar el 'trait'
    fn fn_default_impl(param1: i32) -> i32 {
        param1 * 4
    }
}
```

>>> *Generics y trait objects*

```rust
// Ambos conceptos permiten usar parámetros genéricos pero son totalmente distintos en implementación:
//
// - Los 'generics' son como las plantillas de C++ y replican el código para todos los tipos afectados
// en tiempo de compilación. No tienen impacto en tiempo de ejecución pero incrementan el tamaño del ejecutable.
//
// - Los `trait objects` son objetos que implemtan el `trait` pero se determinan en tiempo de ejecución y por
// tanto tienen una leve penalización en el rendimiento.

// Esta función usa 'generics', en tiempo de compilación se genera una copia de la función para cada tipo posible.
// Esto es equivalente a
// fn print_generic(param: impl std::fmt:Display) {
fn print_generic<T: std::fmt::Display>(param: T) {
    println!("{}", param.to_string());
}

// Esta función recibe un 'trait object', la función `to_string()` real se obtiene en tiempo de ejecución.
// Los 'trait objects' tienen que ser referencias porque no se conoce el tipo en tiempo de compilación.
fn print_trait(param: &dyn std::fmt::Display) {
    println!("{}", param.to_string());
}
```

>>> Atributos

```rust
// Los atributos se escriben como `#[atributo]` y añaden funcionalidad a estructuras, enumeraciones, campos y variantes.

// Un tipo de atributo muy útil y habitual es `#[derive()]`, que implementa automáticamente el 'trait' especificado.

// Estos atributos ejecutan macros procedimentales, pero cómo implementar estas macros está fuera del alcance del taller.

// Esta estructura implementa `.to_string()`, `.clone()` y se puede imprimir con `{:?}` en `println!`
#[derive(Display, Debug, Clone)]
struct MyStruct {
    field1: i32,
}

// Esta estructura se puede seriailzar y deserializar en distintos formatos usando el crate `serde`
#[derive(Seriailze, Deserialize)]
struct AnotherStruct {
    // Este atributo adquiere un valor por defecto si no se especifica valor y se renombra como 'name'
    #[serde(default)]
    #[serde(rename="name")]
    field1: String,
}
```

>>> Condicionales y bucles

```rust
fn main() {
    let mut a = 5;

    if a < 10 {
        println!("a is less than 10");
    } else if a >= 10 {
        println!("a is greater or equal than 10");
    } else {
        println!("other value");
    }
    while a < 10 {
        if a == 10 {
            continue;
        }
        a += 1;
    }
    // Esto es equivalente a `while true`
    loop {
        if a == 20 {
            break;
        }
        a += 1;
    }
}
```

>>> Bucles for e iteradores

```rust
fn main() {
    // 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    for i in 0..10 {
        println!("{}", i);
    }

    let v: Vec<i32> = vec![0, 1, 5, 10];
    for item in v.iter() {
        println!("{}", item);
    }

    // Rust incorpora varias características de lenguajes funcionales.
    // v2 = vec![2, 10, 20]
    let v2: Vec<i32> = v.iter().filter(|item| item > 0).map(|item| item * 2).collect();
}
```

>>> Comprobación de patrones

```rust
// `match` es similar a `switch` en otros lenguajes de programación pero más completo, comprueba patrones.

enum MyEnum {
    Variant1,
    Variant2(bool, String),
    Variant3 {
        enabled: bool,
        name: String,
    },
    Variant4,
}

fn main() {
    let my_enum = MyEnum::Variant2(false, "hola".to_string());

    match my_enum {
        MyEnum::Variant1 => println!("Variant1"),
        MyEnum::Variant2(enabled, name) => println!("{}", name),
        MyEnum::Variant3 {enabled, name} => println!("Variant3"),
        // Cualquier otro patrón, el compilador fuerza en nivel de compilación que se comprueben todas las variantes,
        // por lo que no incluir esta última opción u otra rama con `Variant4` fallaría al compilar
        _ => println!("Another thing"),
    }
}
```

>>> Option y Result

```rust
// Rust no tiene `null` ni `None`, ¡se acabaron las `NullPointerException` y los `NoneError`!
// ¿Pero entonces cómo indicar la posible ausencia de valor? Con la enumeración `Option`, con variantes `Some(T)` y `None`.

// Tampoco tiene excepciones, sino que las funciones retornan `Result`, ya no más excepciones no comprobadas.
// `Result` es una enumeración con variantes `Ok(T)` y `Err(E)`.
fn print(value: Option<i32>) -> Result<(), String> {
    match value {
        Some(val) => println!("{}", val),
        None => return Err("El argumento no tiene valor".to_string()),
    }

    // Variante `Ok` con una tupla vacía como primer campo
    Ok(())
}

fn main() {
    // Si no se comprueba el resultado el compilador genera una advertencia.
    // `unwrap()` obtiene el valor y lanza un `panic!` si es un error (o `None` con los `Option`).
    // `except()` es similar pero con un mensaje de error como argumento.
    match print(Some(10)) {
        Ok(_) => {},
        Err(msg) => println!("{}", msg),
    }
}
```

>>> if let y while let

```rust
// Sintaxis alternativa a `match` útil sobre todo con `Option` y `Result`.

fn main() {
    let mut a = Some(10);

    if let Some(val) = a {
        println!("{}", val);
    } else {
        println!("None");
    }

    while let Some(val) = a {
        if val > 0 {
            a = Some(val - 1);
        } else {
            a = None;
        }
    }
}
```

>>> El operador ?

```rust
// El operador `?` retorna el error si el `Result` es un `Err` (propaga el error) y sigue la ejecución
// si es válido, ahorrando mucho código. Para ello el tipo de error retornado tiene que ser el mismo que el obtenido.

use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt")?;
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
```

>>> anyhow y thiserror

```rust
// `anyhow` y `thiserror` son dos crates que facilitan la gestión de errores y el uso del operador `?`.

// `anyhow` facilita la gestión mediante anyhow::Result y es más adecuada para binarios que no exponen una interfaz.
use anyhow::{Context, Result};
fn main() -> Result<()> {
    let config = std::fs::read_to_string("cluster.json")?;
    let it = Some("");
    it.context("Failed to detach the important thing")?;
    let content = std::fs::read(path)
        .with_context(|| format!("Failed to read instrs from {}", path))?;
}
```

```rust
// Por el contrario, `thiserror` facilita la creación de errores propios y es más útil en interfaces que usarán otros.
use thiserror::Error;
#[derive(Error, Debug)]
pub enum DataStoreError {
    #[error("the data for key `{0}` is not available")]
    Redaction(String),
    #[error("invalid header (expected {expected:?}, found {found:?})")]
    InvalidHeader {
        expected: String,
        found: String,
    },
}
```

>>> Reglas de propiedad y transferencia de valores (I)

- Para usar Rust conviene pensar en los valores en memoria (sin importar si en registro, pila, montículo, etc.) como entidades independientes. Rust sigue estas reglas:
  - Cada valor puede pertenecer a un único propietario
  - Cuando el ámbito del propietario se acaba el valor se elimina de memoria
    - Cuando eso sucede se invoca la función `drop` del *trait* `Drop`, pero generalmente no hace falta implementarlo porque la memoria se gestiona bien
- En cuanto a las referencias, en cada momento puede haber:
  - Una referencia mutable
  - Varias inmutables
- Estas reglas también previenen las condiciones de carrera en programas concurrentes

>>> Reglas de propiedad y transferencia de valores (II)

```rust
fn main() {
  let a = String::from("Hola");
  // Esta sentencia hace que `b` sea la propietaria del String, y que `a` no tenga ningún valor asociado.
  // Si el valor de `a` fuera un tipo básico, el valor se copiaría en lugar de moverse y ambas serían iguales.
  let b = a;

  // La siguiente sentencia supondría un error de compilación porque el valor de `a` se ha movido a `b`
  // println!("{}", a);

  let c = String::from("mundo");
  // Para copiar un valor no `Copy` se puede usar el método `clone()` (si el tipo lo ha implementado)
  let d = c.clone();

  println!("{} {}", c, d); // "mundo mundo"

  while true {
      let new_var = 0;

      break;
  }

  // La siguiente sentencia supondría un error de compilación porque `new_var` se ha borrado al acabar el bucle
  // println!("{}", new_var);
}
```

>>> Tiempos de vida de las referencias

- Todas las referencias tienen un tiempo de vida asociado (el de su propietario)
  - En el caso de los `&str` escritos en el código es `'static`, válido durante todo el programa
- En la mayoria de ocasiones no es necesario especificarlo porque el compilador deduce el mismo para todas las referencias de argumentos y retorno
- Pero en ocasiones hay que especificar los tiempos de vida y la relación entre ellos:

```
// La cláusula `where` indica que 'long vive igual o más que 'short, sin ella habría un error de compilación.
// Del mismo modo, si la función se invoca con un segundo parámetro que vive menos que el primero fallará la compilación.
fn select<'short, 'long>(s1: &'short str, s2: &'long str, second: bool) -> &'short str
where
    'long: 'short,
{
    if second { s2 } else { s1 }
}
```

>>> ¿Paso por referencia o por valor?

- Al llamar a una función con un parámetro sin referencia se está moviendo la propiedad a la función (a no ser que sea Copy)
  - El valor no se copiará por lo que no hay penalización de rendimiento, pero ya no se podrá usar tras llamar a la función
    - A no ser que la función retorne el valor y se vuelva a mover en la llamada:

```
...
a = function(a);
...
```

- Por tanto, hay que usar la opción menos restrictiva posible al definir funciones:
  - referencias
  - si no es posible referencias mutables
  - y si no valores sin referencia

>>> Algunos trucos prácticos

```rust
// En los campos de las estructuras y en los valores de retorno generalmente se usarán valores sin referencia.

// Es posible definir campos como referencias pero la interfaz se hace demasiado engorrosa rápidamente.

// En el caso del valor de retorno de las funciones, la referencia debe ser a alguno de los parámetros, ya que
// lo que se define dentro de la función se elimina al finalizar su ejecución (se acaba su ámbito).
// Es mejor usar el tipo más genérico posible: `&str` en lugar de `&String`, `&[u8]`, en vez de `&Vec<u8>`, etc.
fn concat(a: &str, b: &str) -> String {
    format!("{}{}", a, b)
}

// En estructuras puede ser útil no almacenar los datos o campos con 'trait objects',
// en ese caso generalmente es más práctico usar 'Box' en lugar de referencias.
// `std::boxed::Box` es una estructura que gestiona la referencia por nosotros.
struct WithBoxes {
    field1: Box<str>,
    field2: Box<dyn MyTrait>,
}

fn main() {
    let my_struct = WithBoxes {
        field1: Box::new("hola"),
        field2: Box::new(MyStructImplementingMyTrait::new())
    };
}
```

>>> Clausuras

- Una clausura es una función anónima que captura las variables del entorno

- Usando la palabra clave `move` delante de la clausura la propiedad de las variables del entorno se mueve en lugar de prestarse

- Las clausuras implementan los siguientes `traits`:
  - `FnOnce` las que mueven los valores capturados y las de abajo
  - `FnMut` las que mutan los valores capturados sin apropiarse de ellos y las de abajo
  - `Fn` las que no mutan ni se apropian de ellos

```rust
// Esta función ejecuta la función `f` perezosamente solo con valores `None`
pub fn unwrap_or_else<F>(self, f: F) -> T where F: FnOnce() -> T {
    match self {
        Some(x) => x,
        None => f(),
    }
}
```

>>> Estructuración del proyecto en módulos

- Tener todo el código en un único fichero no es sostenible, Rust permite dividir un proyecto en módulos de dos formas:
  - Una carpeta con el nombre del módulo y un fichero principal llamado `mod.rs`
  - Una carpeta con el nombre del módulo y un fichero principal llamado `nombre_modulo.rs` en el mismo nivel de carpetas

```
/*
Las posibles organizaciones de ficheros son:

|- lib.rs                |- lib.rs
|- test_mods             |- test_mods
    |- mod.rs            |   |- test_mods_mods
    |- test_mods_mods    |- test_mods.rs
*/

mod test_mods;

// Con esta setencia se pueden usar desde otros paquetes los contenidos como si estuvieran en el principal
pub use test_mods::*
```

>>> Tests

- Los tests se ejecutan con el comando `cargo test`
  - Este comando también ejecuta el código incluido en los comentarios de documentación
  - Los tests unitarios se escriben en distintos módulos
  - Los de integración en ficheros `.rs` dentro de una carpeta `tests` al mismo nivel que `src`

```rust
// Los módulos de test llevan el siguiente atributo y no se incluyen en el binario final
#[cfg(test)]
mod tests {
    // Las funciones de test llevan el siguiente atributo
    #[test]
    fn test_add() {
        assert_eq!(4, add(2, 2), "add(2, 2) should be 4");

        assert!(true, fn_bool(true));
    }

    #[test]
    #[should_panic]
    fn test_panic() {
        panic!("Falla")
    }
}
```

Creación de un algoritmo genético usando oxigen

>>> Conceptos básicos sobre los algoritmos genéticos

- Los algoritmos genéticos son una metaheurística para buscar soluciones en un espacio de problemas imitando la evolución natural

- Sobre una población de individuos se ejecutan diferentes generaciones hasta alcanzar el criterio de parada aplicando los siguientes operadores:
  - Selección: seleccionar estocásticamente a los individuos más aptos
  - Cruce: combinar por parejas los seleccionados generando dos hijos
  - Mutación: cambiar genes aleatoriamente con probabilidad baja
  - Presión de supervivencia: reducir el tamaño de la población matando a los menos aptos

>>> Pasos para la creación de un algoritmo genético

- Codificar el problema en genes (genotipo)
  - Representar el problema de la manera más compacta posible para reducir el tamaño del espacio de búsqueda usando *bits* o números enteros
- Creación de una función de evaluación
  - Otorgar una puntuación numérica a cada individuo en función de sus capacidades para resolver el problema
- Selección de los operadores y el criterio de parada a utilizar
  - Número máximo de generaciones, solución encontrada, etc.

>>> Oxigen

- *Framework* de código abierto para la creación de algoritmos genéticos
- Paralelo, flexible y escrito en Rust
- Proporciona los algoritmos más usados para todos los operadores y permite añadir operadores nuevos
- Para crear un algoritmo genético solo hay que implementar el *trait* `Genotype` y elegir los parámetros de la ejecución
- `https://github.com/Martin1887/oxigen`

>>> Ejemplo del problema *onemax* en Oxigen (I)

- Solución del problema: individuo con todos los genes con valor 1
- Codificación (`true` representa 1 y `false` representa 0):

```rust
use std::fmt::Display;

/// A vector of booleans represent an individual, where each gen is 0 or 1
#[derive(Clone)]
struct OneMax(Vec<bool>);

// Oxigen requires implementing the `Display` trait to implement `Genotype`
impl Display for OneMax {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> Result<(), std::fmt::Error> {
        write!(f, "{:?}", self.0)
    }
}
```

>>> Ejemplo del problema *onemax* en Oxigen (II)

```rust
impl Genotype<bool> for OneMax {
    type ProblemSize = usize;

    fn iter(&self) -> std::slice::Iter<bool> {
        self.0.iter()
    }
    fn into_iter(self) -> std::vec::IntoIter<bool> {
        self.0.into_iter()
    }
    fn from_iter<I: Iterator<Item = bool>>(&mut self, genes: I) {
        self.0 = genes.collect();
    }
    fn generate(size: &Self::ProblemSize) -> Self {
        let mut individual = Vec::with_capacity(*size as usize);
        let mut rgen = SmallRng::from_entropy();
        for _i in 0..*size {
            individual.push(rgen.sample(Standard));
        }
        OneMax(individual)
    }
```

>>> Ejemplo del problema *onemax* en Oxigen (III)

- La función de evaluación es el número de unos
- La mutación consiste en cambiar el valor del gen
- Un individuo es solución si tiene el máximo *fitness*

```
    fn fitness(&self) -> f64 {
        (self.0.iter().filter(|el| **el).count()) as f64
    }

    fn mutate(&mut self, _rgen: &mut SmallRng, index: usize) {
        self.0[index] = !self.0[index];
    }

    fn is_solution(&self, fitness: f64) -> bool {
        fitness as usize == self.0.len()
    }
}
```

>>> Ejemplo del problema *onemax* en Oxigen (IV)

```rust
fn main() {
    let problem_size: usize = std::env::args()
        .nth(1).expect("Enter a number bigger than 1")
        .parse().expect("Enter a number bigger than 1");
    let population_size = problem_size * 8;
    let log2 = (f64::from(problem_size as u32) * 4_f64).log2().ceil();
    let (solutions, generation, _progress, _population) = GeneticExecution::<bool, OneMax>::new()
        .population_size(population_size)
        .genotype_size(problem_size)
        .mutation_rate(Box::new(MutationRates::Linear(SlopeParams {
            start: f64::from(problem_size as u32) / (8_f64 + 2_f64 * log2) / 100_f64,
            bound: 0.005,
            coefficient: -0.0002,
        })))
        .selection_rate(Box::new(SelectionRates::Linear(SlopeParams {
            start: log2 - 2_f64,
            bound: log2 / 1.5,
            coefficient: -0.0005,
        })))
        .select_function(Box::new(SelectionFunctions::Cup))
        .run();

    println!("Finished in the generation {}", generation);
}
```

>>> Ejercicio: crear un algoritmo genético con Oxigen

- Elegir un problema y resolverlo mediante un algoritmo genético con Oxigen
- Quien no encuentre ninguno puede resolver el problema de las N reinas:
  - Colocar N reinas en un tablero de ajedrez de $N \times N$ casillas sin que ninguna se ataque
    - No puede haber ninguna reina en la misma fila, columna o diagonal
- El ejemplo del problema *onemax* se puede descargar en `https://github.com/Martin1887/oxigen/blob/master/onemax-oxigen/src/main.rs`
- Antes de nada hay que especificar la dependencia en el `Cargo.toml`

```
[dependencies]
#La última versión 2.x disponible. "2.2" seleccionaría la última versión 2.2.x disponible
oxigen = "2"
```

Documentación con mdBook y publicación en crates.io

>>> Documentación automática del código usando los comentarios

- El comando `cargo doc` genera la documentación en HTML
  - Al publicar el paquete en crates.io se sube la documentación automáticamente a `https://docs.rs`
- Los comentarios de documentación admiten Markdown
- No es necesario especificar los tipos de los parámetros y los valores de retorno, se muestran en la documentación automáticamente
- Secciones y convenciones recomendables: `https://deterministic.space/machine-readable-inline-markdown-code-cocumentation.html`

>>> mdBook

- Documentar el proyecto es importante para facilitar el uso del proyecto
  - A veces un *README* es suficiente
  - Pero si es relativamente grande es mejor usar otras herramientas
- Crea libros en HTML con funciones de búsqueda y varios temas desde varios MD
  - Instalar el binario desde la última *release*: `https://github.com/rust-lang/mdBook/releases`
  - Crear un mdBook nuevo con `mdbook init`
  - Añadir capítulos al `SUMMARY.md` y editar el contenido de los ficheros
  - Generar el HTML con el comando `mdbook build`
  - Más información incluyendo *workflows* de GitHub y GitLab en el mdBook oficial: `https://rust-lang.github.io/mdBook/`

>>> mdbook-pandoc y pandozed

- `mdbook-pandoc` es un *backend* de mdBook usando `pandozed`
  - Un único `mdbook build` para convertir a PDF, EPUB, etc. usando distintas plantillas
  - Una única fuente para todos los formatos de documentación usando tu editor favorito
  - `https://github.com/Martin1887/mdbook-pandoc`
- `pandozed` es un envoltorio de Pandoc escrito en Rust con pilas incluidas
  - Configuración en TOML fácil y cómoda para cada formato de salida
  - Con las plantillas y los filtros más populares incluidos en el binario
  - Configuraciones precargadas para cada plantilla
  - Comando para exportar e importar configuraciones personalizadas
  - `https://github.com/Martin1887/pandozed`

>>> Registro en crates.io

- El repositorio oficial para paquetes de Rust

- El primer paso es crear una cuenta:

  1. Acceder a `https://crates.io/`
  2. Hacer clic en 'Login with GitHub' (el único método de autenticación actualmente)
  3. Acceder a `https://crates.io/me/` y copiar la `API key`
  4. Ejecutar en un terminal `cargo login <API_key>`
  5. Cargo ha guardado las credenciales en `~/.cargo/credentials`
     - Estas credenciales son secretas, si se ven comprometidas hay que regenerar la `API key`

>>> Publicación en crates.io

- Antes de publicar el paquete conviene añadir información relevante al `Cargo.toml`:
  - `version`
  - `license`
  - `description`
  - `readme`
  - `repository`
  - `authors`
  - `edition`
  - `keywords`
  - `categories`

- Con el repositorio en un estado limpio (sin cambios fuera de *commits*) ejecutar el comando `cargo publish`

Conclusiones

>>> Aspectos no cubiertos en el taller

- *Traits* avanzados
- Mutabilidad interior (`Cell` y `Refcell`)
- Programación concurrente
  - `rayon`
  - `crossbeam`
- Macros y macros procedimentales
- Programación asíncrona (*futures* y `async`)
- `unsafe`
- *Foreign Function Interface* (FFI)
- Programación para sistemas embebidos
- Desarrollo web
- WebAssembly
- Interfaces para terminal (TUI)
  - `tui-rs`
  - `Cursive`
- Interfaces CLI (`clap`)
- Interfaces gráficas (GUI)
- Compilación en múltiples SOs
- Medición de rendimiento y *profiling*
- Etc.

>>> Recursos para seguir aprendiendo

- *Comprehensive Rust* (curso de 4 días creado por Google)
- El libro de Rust (también disponible en castellano)
- *Rust by Example*
- *Rustlings* (ejercicios para aprender Rust)
- *Awesome Rust* (paquetes más importantes y recursos de aprendizaje)
- *Rust Concepts I Wish I Learned Earlier*
- *The Little Book of Rust Macros*
- *Nine Rules for Creating Procedural Macros in Rust*
- *Rust Async Book*
- *The Rustonomicon* (las artes oscuras de *unsafe Rust*)

Turno de preguntas

GRACIAS

https://github.com/Martin1887/rust-introduction-workshop