Variables y tipos de datos en Rust:

Seguimos en Rust 101, el tutorial de Rust en español, y ahora vamos a ver como trabaja Rust con las variables. En este aspecto no se distingue demasiado de otros lenguajes imperativos, aunque vamos a ver que la inferencia de tipos nos va a ayudar. También vamos a ver tipos de datos más complejos.

Declaración de variables:

En Rust las variables de declaran usando la palabra reservada **let**. El tipo de la variable puede inferirse pero en ocasiones podemos ayudar al compilador indicando el tipo del que queramos que sea la variable.

```
fn main(){
    let x = 42;
    let url = "http://adrianistan.eu";

    println!("{}",url);
}
```

Sin embargo hay una diferencia importante respecto a otros lenguajes y es que aquí las variables serán por defecto inmutables. Tratar de hacer esto:

```
x += 5;
```

Provocará un error de compilación. No obstante, esta limitación puede saltarse aplicando el modificador **mut**. Aunque las variables puedan volverse mutables esto ya nos avisa de que Rust tiene muy presente conceptos de programación funcional más pura (como podemos tener en Haskell, Elm, F# u OCaml).

```
fn main(){
    let mut x = 42;
    x += 5;
    println!("{}",x);
}
```

Para espeficicar un tipo usamos una notación de dos puntos después de la palabra clave let.

```
fn main(){
    let x: i32 = 5; // int 32 bits
    let y: f64 = 5; // float 64 bits
}
```

Arrays y vectores.

Hasta ahora hemos podido ver en acción variables de tipos de datos más o menos primitivos. Ahora vamos a ver dos estructuras de datos similares, aunque diferentes. Digamos para simplificar, que los arrays tienen tamaño fijo y los vectores no.

```
fn main() {
        let mut notas_array: [i32;5] = [0;5];
        notas_array[0] = 1;
        notas_array[1] = 6;

        let mut notas_vec: Vec<i32> = vec!();
        notas_vec.push(1);
        notas_vec.push(6);

        println!("Nota 2: {}",notas_array[1]);
        println!("Nota 2: {}",notas_vec[1]);
}
```

En este caso inicializamos un array todo a cero de 5 posiciones y también un vector. He dejado las marcas de los tipos, aunque no son necesarias.

Constantes, let shadowing y casting

Rust soporta constantes con la palabra reservada **const**. La convención es usar SCREAMING_SNAKE_CASE para las constantes. El valor de una constante es inmutable y a diferencia de **let**, no se puede reescribir. ¿Espera, las variables con **let** se pueden reescribir? En efecto, es posible definir una variable posteriormente, de tipo totalmente distinto con el mismo nombre. En este ejemplo vamos a ver el uso de **const** y el uso de *let shadowing*.

```
const PI: f64 = 3.14;

fn main(){
    let x = 42;
    let x = (x as f64) + PI;
    println!("{}",x);

}
```

Además podemos apreciar como se realiza **casting** en Rust. Se usa la palabra reservada **as** y se indica el tipo al que queremos hacer cast. En este ejemplo de *let shadowing* podemos apreciar como no solo hemos redefinido x, sino que lo hemos hecho de otro tipo. En la primera línea, x es de tipo i32, en la segunda pasa a ser f64.

Tuplas

Las tuplas pueden resultar algo novedoso a quien venga de ciertos lenguajes, pero no son elementos realmente nuevos. Podría entenderse una tupla como una estructura sin nombre. Se trata de una especie de array donde cada elemento puede ser de un tipo, pero espeificado de antemano. Lo veremos mejor con ejemplos.

```
fn main(){
    let tupla = (42,"Adrianistán",true);
    let (random,country,has_beers) = tupla;
    println!("{}",random);
    let (_,country,_) = tupla;
    println!("{}",country);
    let has_beers = tupla.2;

}
```

Aquí podemos observar como se forma una tupla, compuesta por un entero, un texto y un booleano. La sentencia con let desestructura la tupla y se obtienen 3 variables con los valores de los 3 campos de la tupla. Es posible que en algunas situaciones no nos importe algún campo. Para ignorar algún campo, dejamos una barra baja. También existe la sintaxis punto para acceder a elementos de la tupla. Para ello indicamos el orden dentro de la tupla del elemento que queremos. No obstante, no es tan claro como desestructurar la tupla.

Expresiones avanzadas con let

let admite cualquier expresión. Esto es una práctica habitual en muchos lenguajes imperativos, sin embargo, las expresiones en Rust son más amplias de lo que vemos a simple vista.

```
fn main(){
    let age = 65;
    let x = if age > 17 {
        "Mayor de edad"
    }else{
        "Menor de edad"
    };
    println!("{}",x);

}
```

Por ejemplo, una estructura condicional if-else es una expresión. Un ojo avizado podría decir que este código está mal, que falta el punto y coma. Realmente no es así, en Rust si algo no lleva punto y coma se vuelve una expresión. Más bien, al no llevar punto y coma no se vuelve un *statement*. Esto puede aplicarse a cualquier tipo de código que no deja de ser una expresión.

```
fn main(){
    let age = 65;
    let a = 10;
    let b = 25;
    let x = {
        let u = a*b;
        u+age
    };
    println!("{}",x);

}
```

Y como véis, se puede complicar hasta niveles muy elevados. Al principio puede no ser muy natural, pero es bastante cómodo en muchas situaciones.

Con esto acabamos nuestra visita a las variables.