Estructuras de control en Rust.

Los lenguajes de programación imperativos no serían nada sin sus estructuras de control. En Rust no son muy diferentes respecto a otros lenguajes. Ya hemos visto de pasada que existe **if-else** y **for-in**. Ahora veamos la lista entera junto con un operador muy importante en Rust, **match**.

Condicionales, if, if-else, if-let.

En Rust **if-else** funciona de forma parecida a otros lenguajes, quizá lo único destacable sea que no hacen falta los paréntesis.

```
fn main(){
    let ano = 1998;
    if ano > 1975{
        println!("Deberías escuchar a Ennio Morricone");
    }else if ano > 1600{
        println!("Deberías escuchar a Bach");
    }else{
        println!("Deberías escuchar a tu monje benedictino más próximo");
    }
}
```

En realidad la sintaxis del **if** no es tan simple, ya que existe una versión vitaminada, pensada para tratar con Option y Result. Se trata de **if-let**, una construcción tomada de Swift y que sirve para ejecutar una porción de código sólo si el valor existe (en el caso de Option) o no ha habido fallos (con Result).

```
fn main(){
    let url = Some("https://blog.adrianistan.eu");

    if let Some(url) = url {
        println!("{}",url);
    }

    let magic_box: Result<String,String> = Ok(String::from("Aquí no hay
nada"));
```

```
    if let Ok(magic_box) = magic_box {
        println!("{}",magic_box);
    }

    let url: Option<&'static str> = None;
    if let Some(url) = url {
        println!("{}",url);
    }

    let magic_box: Result<String,String> = Err(String::from("No tienes la
llave de la caja"));
    if let Ok(magic_box) = magic_box {
        println!("{}",magic_box);
    }

}
```

Condicionales, match.

match es un potente condicional basado en concordancia de patrones. Podemos pensar en ello como un **switch** vitaminado, pero realmente es capaz de hacer muchas más cosas. Como curiosidad, mencionar que ciertos lenguajes de programación como Haskell usan la concordancia de patrones muy a menudo y consiguen obtener un código muy claro.

Para demostrar el uso del match primero vamos a construir una tupla con cuatro valores.

```
fn buscar(t: (bool,&'static str,&'static str,i32)) -> &'static str{
    match t {
        (true,"Mike Oldfield","The Bell",1992) => "Gran canción",
        (true,"Mike Oldfield",_,1992) => "Tubular Bells II probablemente",
        (true,autor,_,_) => autor,
        (false,..) => "Disco no existente",
        _ => "¿Qué me has pasado exactamente?"
    }
}

fn main(){
    let tupla = (true,"Mike Oldfield","The Bell",1992);

    let busqueda = buscar(tupla);
    println!("{}",busqueda);

}
```

match realiza comparaciones de arriba a abajo y devuelve la expresión después de la flecha. Como os habréis imaginado, al ser una expresión directamente no se pone punto y coma y no hace falta hacer return en la función de ningún tipo.

La primera condición del match es clara: si es exactamente igual la tupla a la descrita se devuelve *Gran canción*. En la segunda vemos una barra baja. Ya hemos hablado de él, pero vamos a recordarlo. Se trata del operador que usamos cuando algo nos da igual. Similar a un asterisco en la terminal de Linux.

El tercer caso es interesante pues nos permite comprobar concordancia en el primer campo y a su vez nos permite extraer el valor del segundo, que queda guardado en la variable autor.

El cuarto caso es para los perezosos, simplemente con que cumpla una condición, el resto ni nos molestamos en escribir. Usando los dos puntos hacemos que todos los false vayan al mismo sitio.

Por último pero no menos importante, tenemos la barra baja. Cualquier tupla que no haya caído antes en otro caso caerá en este. (En este ejemplo no puede llegar a funcionar, pues con los casos anteriores ya hemos cubierto el espectro. El compilador de Rust nos avisa de que no hay posibilidad de que alguna vez se alcance ese código).

Existen más opciones, una barra horizontal **|** nos servirá para hacer un OR. Si trabajamos con números, **1..10** representa un intervalo en el que caen los valores admitidos.

Por descontando, **match** no solo funciona en tuplas, sino que tiene una gran variedad de usos.

Bucles, loop y while.

Estos bucles funcionan prácticamente igual que en otros lenguajes. **loop** es un bucle infinito que solo se rompe con **break**. **while** tiene una condición que se comprueba al inicio.

```
fn main(){
    let mut n = 0;
    loop {
        if n > 100{
            break;
        }
        println!("No soy pesado");
        n+=1;
    }

    while n < 200{
        println!("No soy pesado");
        n+=1;
    }
}
```

Es conveniente usar los bucles **while** cuando tenemos una bandera o una condición inesperada, pero nunca para recorrer elementos de una lista o similares.

Bucles, for-in.

En Rust se toma el ejemplo de lenguajes como Python y se deja atrás el bucle **for** de C. El bucle **for-in** recorre todos los elementos de un **Iterable**. Un **Iterable** es cualquier cosa que implemente la trait **Iterable**, y que como en lenguajes como Python y JavaScript con los generadores, puede no ser necesariamente una lista, simplemente algo sobre lo que se pueda iterar. Estos elementos son perezosos y solo se calculan cuando son necesarios, abriendo la posibilidad a listas infinitas. Los vectores son **Iterables**, veamos como usarlos.

```
fn main(){
    let v = vec!["Haskell","Elm","Python","C++","JavaScript","Rust","Java"];
    for s in v{
        println!("{}",s);
    }
}
```

Como vemos, es bastante simple. Este programa imprime cada lenguaje de programación por pantalla. El código no es muy complicado.

¿Y si resulta que quiero iterar N veces y no tengo un **Iterable** a mano? Similar al **range** de Python, existe una sintaxis compacta y clara.

```
fn main(){
    let n = 100;
    for i in 0..n{
        println!("{}",i);
    }
}
```

La sintaxis de dos puntos permite iterar en un rango en Z cerrado por la izquierda y abierto por la derecha. Muchas cosas de la librería estándar son iterables y cualquier array o vector puede serlo, así que este será probablemente el bucle que más uses.

Con esto acabamos las estructuras de control, listos para entrar en temas más avanzados.