Rust desde cero

Gorka Urrutia Landa

Version 1.0, 14/10/2022

Contenido

Presentación

¿Por qué este innecesario libro?

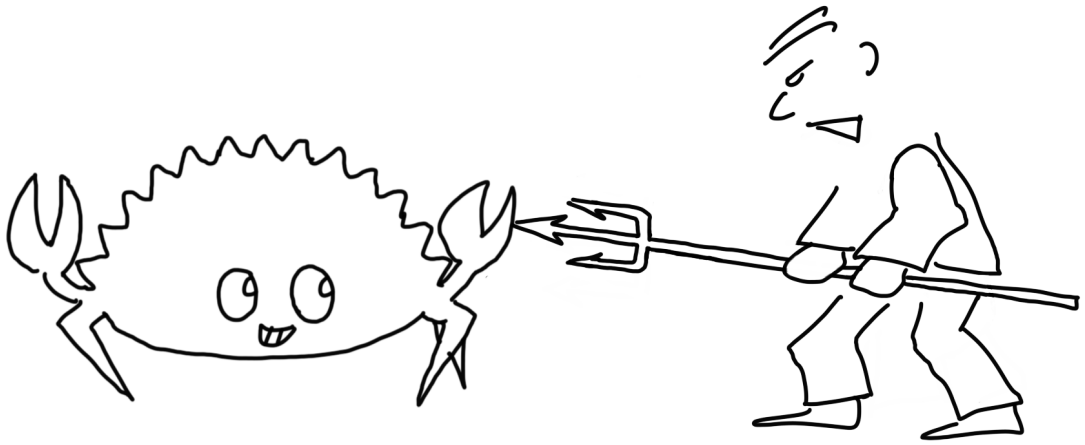
¿Por qué digo que es innecesario? Pues porque ya hay un libro oficial que es muy completo y está muy bien explicado.

Y entonces ¿por qué escribir otro libro de Rust?

Pues tengo varias razones:

- Escribirlo en un libro que voy a compartir me obliga a no saltarme ninguna parte.
- Me ayuda a entender bien cada uno de los conceptos. Creo que entiendes mejor las cosas cuando haces el esfuerzo de explicárselo a otros.
- Me encanta escribir.
- Tengo una excusa para meter mis dibujillos.
- Ansias de protagonismo.

Pero, por encima de todo... me apetece hacerlo.



Sobre el autor

El autor de esta ignominia es Gorka Urrutia Landa.

Una rápida introducción

Instalación de Rust

Como eso de las instrucciones de instalación dependen del sistema y está explicado en mil sitios prefiero pasar de puntillas sobre el tema y dejarte el enlace oficial sobre el tema:

https://doc.rust-lang.org/book/ch01-01-installation.html

Los preparativos

$ cargo new ejemplo-1 $ cd ejemplo-1

Esto nos creará el proyecto en el que aparecerán por arte de magia dos ficheros:

- src/main.rs
- Cargo.toml

Por ahora el que nos importa es src/main.rs:

```
fn main() {
    println!("Hello world!");
}
```

cap-01/ejemplo-01/src/main.rs

Compilamos y lanzamos el programa:

```
$ cargo run
```

y esto nos mostrará, como habrás imaginado tras un colosal esfuerzo mental:

```
Hola mundo!
```

¿Qué es ese signo de exclamación? Las *macros*

No hemos hecho más que empezar y ya se pueden ver cosas "raras" en el código.

Por un lado tenemos un signo de exclamación después del *println* ¿y eso qué porras es? ¿acaso se me ha colado? Pues no, es la forma que tiene Rust de identificar una **macro**.

¿Y qué es una macro? Pues es una forma de no repetir código pero sin tener que usar funciones.

¿Y por qué es mejor una macro que una función? Pues porque una macro es más eficiente que una función.

Ya veremos las macros más adelante, por ahora basta con saber que eso del signo de exclamación es una macro, que Rust incorpora unas cuantas que podemos usar y que podemos crear nuestras propias macros.

Nuestra primera variable

Para que el ejemplo sea un poco menos tonto vamos a añadir la variable *nombre* y vamos a mostrar su valor:

```
fn main() {
    let cabezas = 1;
    println!("Gorka tiene {} cabeza.", cabezas);
}
```

cap-01/ejemplo-02/src/main.rs

Bueno, aquí varios apuntes con respecto a las variables:

Las variables se definen usando *let*. No te olvides del let.

Rust es un lenguaje tipado. Eso quiere decir que todas las variables tienen que ser de algún tipo (booleano, entero, coma flotante, carácter, o alguno de los otros tipos raros que tiene Rust).

Pero, si es tipado y en el ejemplo no hemos puesto el tipo ¿por qué el programa no falla? Eso se debe a que Rust es muy listo y es capaz de adivinar el tipo con el valor que hemos asignado (el '1').

Ah, y una vez que definimos una variable ya no podemos cambiarle el tipo.

Y otra cosa... los nombres de las variables usan el formato *snake_case*. Y si no lo haces el complilador se va a quejar. En el mundo de Rust no hay peleas del formato a usar en nombres de variables porque si no usas snake_case viene el cangrejo que tienen por mascota y te pellizca.

Para mostrar el valor de una variable con *println!* usaremos las llaves:

```
println!("Gorka tiene {} cabeza.", cabezas);
```

Pero también se puede hacer así:

```
println!("Gorka tiene {cabezas} cabeza.");
```

Tendremos que poner tantas llaves como valores queramos mostrar:

```
fn main() {
    let cabezas = 1;
    let narices = 1;
    println!("Gorka tiene {} cabeza y {} narices.", cabezas, narices);
}
```

Los tipos de datos

cap-01/ejemplo-03/src/main.rs

He dicho antes que Rust es muy listo y no hace falta meter los tipos de datos. Pero a veces nos interesa especificar el tipo de dato. Si queremos decirle a Rust que queremos que esas variables sean de tipo entero Y sin signo podemos hacerlo añadiendo el tipo:

```
fn main() {
    let cabezas : u64 = 1;
    let narices : u64 = 1;
    println!("Gorka tiene {} cabeza y {} narices.", cabezas, narices);
}
```

cap-01/ejemplo-04/src/main.rs

Más cosas poco habituales de Rust: el tipo de especifica **después del nombre variable**.

En otro capítulo veremos los tips de datos disponbibles.

Constantes

Además de variables podemos usar valores directamente:

```
fn main() {
    let edad = 50;
    println!("Gorka ha vivido {} días.", edad * 365);
}
```

cap-01/ejemplo-05/src/main.rs

NOTE::Sí, ya se ¿qué pasa con los años bisiestos? Si eso, otro día.

Pero yo creo que ya tenemos la suficiente edad como para saber que ese '365' no está bien ahí. Eso de meter valores sin significado en el código está feo y lo sabes (o deberías saberlo). Vale, vale, todo el mundo sabe que eso son 365 días que hay en un año. Pero ¿por qué no hacer que el código sea más claro usando una constante? Esos son los pequeños detalles que hacen que tu código sea mejor.

```
fn main() {
    let edad = 50;
    const DIAS_POR_ANYO : u32 = 365;
    println!("Gorka ha vivido {} días.", edad * DIAS_POR_ANYO);
}
```

cap-01/ejemplo-06/src/main.rs

Y ahora unos comentarios sobre las constantes:

- Las constantes se escriben en mayúsculas. Y si no lo haces el compilador de Rust, que es un quejica, te lo va a recordar. Haz la prueba, pon la constante en minúsculas.
- En las constantes **es obligatorio** indicar el tipo de dato que usamos. Con las constantes Rust ya no es tan listo.

¡No puedes cambiar la variable!

Más cosas curiosas de Rust... no puedes modificar el valor de una variable. Como lo oyes. El siguiente código da un error:

```
fn main() {
    let edad_gorka = 50;
    edad_gorka += 1;

    println!("Gorka tendrá {} años el año que viene.", edad_gorka);
}
```

Bueno, sí, sí puedes con *mut*

Este código no funciona.

Pero no nos desgarremos las vestiduras aún. Esto es algo bueno para nosotros como programadores que somos. Si queremos modificar una variable basta con indicar que queremos tener la posibilidad de cambiar su valor.

Y eso se hace añadiendo el modificador **mut**:

```
fn main() {
    let mut edad_gorka = 50;
    edad_gorka += 1;

    println!("Gorka tendrá {} años el año que viene.", edad_gorka);
}
```

¡Ni puedes copiar su valor!

cap-01/ejemplo-07/src/main.rs

Y ahora es cuando te vas a ir a llorar a una esquinita. Con **algunos tipos de datos** no se puede copiar alegremente el valor de una variable directamente. Hay unos conceptos como *borrow* (prestar), *ownership* (propiedad) que al principio puede que te hagan explotar la cabeza. Pero, créeme, son cosas buenas para tí (lo de explotar la cabeza no, lo otro).

Pero lo veremos más adelante. Para que no te quedes con mal sabor de boca te dejo un ejemplo donde sí se pueden copiar valores de variables:

```
fn main() {
    let edad_gorka = 50;
    let edad_clon = edad_gorka;

    println!("Gorka tiene {} años.", edad_gorka);
    println!("El clon tiene {} años.", edad_clon);
}
```

cap-01/ejemplo-08/src/main.rs

Y ahora practica un poco lo que hemos visto para familiarizarte con la sistaxis de Rust. Te recomiendo que no hagas copia/pega con los ejemplos sino que los escribas con tus manitas para que se te vaya quedando.

Variables en Rust

No me toques las variables

Cuando, como yo, llegas a Rust desde otros lenguajes (PHP, JavaScript, Python o C) te encuentras con algunas cosas que te sorprenden bastante. A mí me sorprendió la **inmutabilidad** de las variables. ¡Por defecto una variable no puede cambiar!

Ya te lo he dejado caer en el capítulo anterior; las variables, si no indicamos lo contrario, no se pueden modificar. Pruébalo con este código que no compila:

cap-02/ejemplo-01/src/main.rs (Este código no compila)

```
fn main() {
    let edad_gorka = 50;

    edad_gorka = edad_gorka + 1;

    println!("Gorka tiene tendrá {} años.", edad_gorka);
}
```

¿Y por qué hace eso Rust? ¿Acaso quienes lo diseñaron odian a la humanidad?

A ver, no digo que no odien a la humanidad, pero lo hicieron por nuestro bien. Según dicen cabezas mejor amuebladas que la mía es un tema de seguridad y para evitar errores difíciles de localizar.

Así que, si no decimos nada, nuestras variables serán *inmutables*.

Hay una cosa para solucionar ésto que también me llamó la atención. Podemos hacer funcionar el código anterior añadiendo un siemple *let*:

cap-02/ejemplo-02/src/main.rs

```
fn main() {
    let edad_gorka = 50;

    let edad_gorka = edad_gorka + 1;

    println!("Gorka tiene tendrá {} años.", edad_gorka);
}
```

Aquí estamos creando una nueva variable *edad_gorka* que va a sustituir a la anterior. A ésto se le llamada hacer **shadowing** (¿hacer sombra?). Luego volvemos con el shadowing.

Pero creo que casi es mejor añadir el *mut*:

```
fn main() {
    let mut edad_gorka = 50;

    edad_gorka = edad_gorka + 1;

    println!("Gorka tiene tendrá {} años.", edad_gorka);
}
```

¡Ah! Entonces las variables con el *mut* en realidad son constantes

cap-02/ejemplo-03/src/main.rs

Muy buena apreciación. Pero no. No es correcto.

Las constantes son una cosa diferente a las variables inmutables. Para empezar se definen de manera diferente, tal y como vimos en el capítulo anterior:

cap-02/ejemplo-04/src/main.rs

```
fn main() {
    let edad = 50;
    const DIAS_POR_ANYO : u32 = 365;
    println!("Gorka ha vivido {} días.", edad * DIAS_POR_ANYO);
}
```

La diferencia entre las constantes y las variables inmutables

La diferencia fundamental entre ambas es que las constantes se definen en el momento de la compilación, las variables en tiempo de ejecución. Así que si una constante depende de un valor que no está disponible en tiempo

Un par de ejemplos de casos en los que no podemos usar una constante:

- No podemos guardar un dato introducido por el usuario en una constante.
- Tampoco podremos guardar en una constante datos que extraigamos de un fichero que leamos.

Así que las constantes pueden almacenar valores que tienen que estar disponibles en el momento de la **compilación**. Y las variables pueden almacenar varlores que pueden solo estar disponibles en el momento de la **ejecución**.

Digamoslo aún más claro: **una constante solo puede almacenar valores constantes**.

Es decir, esto no se puede hacer:

```
let variable = 10;
const CONSTANTE : i32 = variable * 2;
```

Pero esto sí:

```
const CONSTANTE_1 = 10;
const CONSTANTE_2 : i32 = CONSTANTE_1 * 2;
```

Shadowing

Hemos visto antes como modificar el valor de una variable *inmutable*:

cap-02/ejemplo-02/src/main.rs

```
fn main() {
    let edad_gorka = 50;

    let edad_gorka = edad_gorka + 1;

    println!("Gorka tiene tendrá {} años.", edad_gorka);
}
```

Y he comentado que a ésto se le llama hacer *shadowing*. En realidad estamos creando una nueva variable pero que tiene el mismo nombre que la primera. Se dice que la segunda variable "hace sombra" (shadow) a la primera.

Y esta es una variable totalmente nueva. Y puede incluso ser de otro tipo como en este ejemplo:

```
fn main() {
    let edad = 50;
    println!("Gorka tiene {} años.", edad);

    let edad = "cincuenta";
    println!("Gorka tiene {} años.", edad);
}
```

cap-02/ejemplo-05/src/main.rs

En este caso tenemos que la primera variable *edad* es de tipo entero y la segunda es una cadena de texto.

Ojo con esto del *shadowing* que va a ser importante entenderlo bien cuando hablemos de referencias.

Variables mutables por defecto

Hemos visto que una variable a la que le damos un valor se convierte en inmutable salvo que la definamos con `mut`. Pero ¿y qué pasa con una variable sin valor inicial? En ese caso Rust entiende que quieres que sea una variable *mutable*. Tiene lógica, si no tiene ningún valor ¿para qué te sirve esa variable.

```
fn main() {
    let edad;

    edad = 50;

    println!("Si tienes más de {edad} años puedes sentirte joven... pero no lo
eres.");
}
```

cap-02/ejemplo-06/src/main.rs

En este caso Rust se quejará si la defines con *mut*.

Tipos de datos escalares

Si has leído escaladores en lugar de escalares deja este libro que no te estás centrando.

Los tipos escalares (scalar) son aquellos que solo contienen un único valor. Bueno, esto ya deberías saberlo si tienes experiencia en cualquier otro lenguaje de programación.

Ejemplos de escalares:

- Números enteros.
- Números en coma flotante.
- Booleanos (eso del true o false).
- Caracteres. Ojo, no cadenas de caracteres, que esos son *strings*.

Números enteros

Un número entero es aquel al que no le falta ningún trozo. Bueno, mejor dejo de hacer chistes sin gracia y me centro en el libro.

Los números enteros puede ser con signo (signed) o sin signo (unsigned) ¿La diferencia? Pues que los *unsigned* van desde el 0 al X y los *signed* incluyen números negativos.

Las variables que van a contener números enteros con signo (signed) se etiquetan usando una *i*. Por ejemplo si queremos usar números enteros con signo de 16 bits usaremos *i16* cuando creemos una variable.

Las variables para enteros sin signo (unsigned) llevan el prefijo *u*.

Imagina que eres una persona a la que el dinero no le falta y llevas 1000 euros en el bolsillo. Podrías crear un programa en Rust para presumir de ello así:

cap-03/ejemplo-01/src/main.rs

```
fn main() {
    let dinero : u16 = 1000;

    println!("Tengo {dinero} euros en el bolsillo.");
}
```

Ahora prueba a cambiar *u16* por *u8*. Verás que el programa falla porque tus 1000 euros no caben en una variable de tipo *u8*. El tamaño máximo que cabe en un *u8* es 255.

Y ahora imagina que vas con tus mil euros al casino y te los juegas en la ruleta. Y los pierdes. Pides otros mil euros prestados y los pierdes también. Ahora tendrás *-1000* euros. Así que tienes que modificar tu programa porque el tipo *u16* no admite números negativos. Prueba a poner -1000 en el ejemplo anterior y verás que no compila.

cap-03/ejemplo-02/src/main.rs

```
fn main() {
    let dinero : i16 = -1000;

    println!("Tengo {dinero} euros en el bolsillo.");
}
```

Rust es bueno. Nos cuida y nos vigila para que no cometamos errores. Creo que es una de las razones por las que Rust gusta tanto.

Así que atención al tamaño y tipo que usas para tus variables. Ojo con quedarse corto pero tampoco malgastes memoria.

TODO - Tabla con los tipos de datos y el número máximo y mínimo de cada uno de ellos.

Si queremos ir a lo loco y usar el máximo número que permita el ordenador donde se esté ejecutando nuestro programa podemos usar *isize* o *usize*.

Tipos de datos compuestos

Aquí tenemos las tuplas y los arrays

Funciones

Las funciones en Rust tienen este formato:

```
fn nombre_funcion(nombre_parametro :tipo_variable) -> tipo_de_valor_develto {
  // código
}
```

Y, como en casi cualquier lenguaje, podemos llamar a una función dentro de otra y, por supuesto, tenemos recursividad.

Recuerda que si ves un signo de exclamación (!) no es una función sino una macro. Por ejemplo *println!()*.

Un ejemplo sencillo de una función:

cap-05/ejemplo-01/src/main.rs

```
fn main() {
    println!("El doble de 10 es: {}", duplica(10));
}

fn duplica(numero: i32) -> i32 {
    return numero * 2;
}
```

¿Quieres más de un parámetro? Pues añade una coma y listo:

cap-05/ejemplo-02/src/main.rs

```
fn main() {
    println!("10 + 20 = {}", suma(10, 20));
}

fn suma(numero1: i32, numero2: i32) -> i32 {
    return numero1 + numero2;
}
```

Snake_case, por favor

Prueba a usar un nombre de función con formato camelCase y verás cómo el compilador de Rust se queja. En esta comunidad no hay discusión sobre si usar camelCase o snake_case; esa pelea ya está decidida.

Todos los parámetros con tipo

En Rust es obligatorio poner el tipo de variable en todos y cada uno de los parámetros. Pero si te olvidas de ellos no te preocupes, el compilador te dará un tirón de orejas.

Los valores devueltos

Vamos a ver una cosa curiosa que tiene Rust pero que no se suele ver en otros lenguajes.

Mira este código. Nada raro, tenemos una función con un *return*. Todo en orden:

cap-05/ejemplo-03/src/main.rs

```
fn main() {
    let z = suma(10, 20);
    println!("10 + 20 = {z}");
}

fn suma(x :i64, y :i64) -> i64 {
    return x + y;
}
```

Y si nos olvidamos el punto y coma final ¡oh sorpresa! sigue funcionando:

cap-05/ejemplo-04/src/main.rs

```
fn main() {
    let z = suma(10, 20);
    println!("10 + 20 = {z}");
}

fn suma(x :i64, y :i64) -> i64 {
    return x + y
}
```

Pero vamos a quitar el *return* a ver qué pasa:

cap-05/ejemplo-05/src/main.rs

```
fn main() {
    let z = suma(10, 20);
    println!("10 + 20 = {z}");
}

fn suma(x :i64, y :i64) -> i64 {
    x + y
}
```

Pues resulta que sigue funcionando. ¿Cómo es posible? Pues funciona porque en Rust, si no añadimos un punto y como al final se considera que es una ***expresion*** Y las expresiones se convierten en un valor. Lo vamos a ver con más detalle y ejemplos en el capítulo siguiente.

Por cierto, esto no funciona:

```
fn main() {
    let z = suma(10, 20);
    println!("10 + 20 = {z}");
}

fn suma(x :i64, y :i64) -> i64 {
    x + y;
}
```

No funciona porque hemos añadido punto y como y ya no es una expresión.

En resumen, en Rust no es necesario usar *return*. Basta con no añadir punto y coma al final de una sentencia y lo que tengamos justo antes se convertirá en un valor. En nuestro ejemplo `x + y` se convierte en el resultado de la suma.

Este capítulo está basado en el siguiente capítulo del libro de Rust: https://doc.rust-lang.org/book/ch03-03-how-functions-work.html

Expresiones y statements

En el capítulo anterior vimos que una función que debe devolver un valor no necesita que añadamos la palabra *return*, basta con no añadir punto y coma al final.

Ojo, que aquí viene una de las peculiaridades de Rust que lo hace diferente a la mayoría de los lenguajes de programación.

Por resumir:

- Si al final de una instrucción no añadimos punto y coma tenemos una **expresión** y ésta **devuelve un valor**.
- Si al final de una instrucción añadimos punto y coma tenemos un **statement** y **no devuelve un valor**.

Tengo una duda que me atormenta ¿Cómo se traduce *statement*?

Vamos a ver unos ejemplos:

cap-06/ejemplo-01/src/main.rs

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

En este caso `{ 5 + 5 }` es una expresión por no tener punto y coma al final. Así que ese bloque nos devolverá `10`.

Pero si añadimos un punto y coma ya no compilará:

cap-06/ejemplo-02/src/main.rs (no compila)

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

Esto se debe a que el bloque `5 + 5;` no es una expresión y ya no devolverá `10`.

Puede que te preguntes ¿vale aquí lo que hemos hecho en el capítulo de "Funciones" de añadir un *return*? Pues no, no vale. Haz la prueba:

cap-06/ejemplo-03/src/main.rs (no compila)

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

Esto no compila y además estamos metiendo un *return* en la función que hará que termine en ese punto. Por lo tanto, el `println!` no se ejecutaría nunca.

En Rust no puedes hacer cosas como:

cap-06/ejemplo-04/src/main.rs (no compila)

```
fn main() {
    let x = let y = 6;
}
```

Porque `let y = 6` es un *statement*, no una *expresión* así que **no devuelve ningún valor** que se pueda usar para `x`. Lo que sí podemos hacer es meter un bloque acabado en una expresión:

cap-06/ejemplo-05/src/main.rs

```
fn main() {
    let x = { let y = 6; y };
    println!("x = {x}");
}
```

Recuerda: una expresión devuelve un valor, un statement no.

No podía faltar el if

If, siempre if.

Bucles con *loop*

Bucles con while

Bucles con for

[[011]] == La pila y el montón

Seguro que si te hablo de la pila y el "montón" no tienes ni idea de lo que te estoy hablando. Pero resulta que son los nombres que me he encontrado que se usan en castellano para **heap** y **stack**. Y, la verdad, eso de "montón" es un nombre absurdo pero es lo mejor que he encontrado.

Es muy probable que no te hayas tropezado mucho en tu vida laboral con estos dos conceptos (salvo que trabajes programando sistemas). Puede que los hayas visto cuando estudiabas. Pero resulta que son dos conceptos que cuando trabajas con lenguajes como Rust es conveniente tener claros.

Así que vamos a por ellos.

Como seguro que ya sabes cuando definimos una variable lo que hacemos es pedirle al sistema que nos reserve parte de la memoria para almacenar allí un valor. Por ejemplo, al definir una variable de tipo i32 le estamos pidiendo al sistema que nos reserve 1 byte (8 bits) de memoria para poder almacenar allí un número. Esa parte de la memoria queda reservada para esa variable y nadie más la puede tocar (mientras exista esa variable).

Cuando creamos una variable de tipo "fijo" (enteros, float, boolean, character) no tenemos que preocuparnos por la memoria. Esto se hace automáticamente.

Pero hay ocasiones en las que puede interesarnos reservar x bytes de memoria. Por ahora no te preocupes de ésto, ya veremos este tipo de situaciones. Simplemente necesitas saber que en ocasiones necesitamos un tamaño especial de memoria. Aquí es donde todo se complica.

Stack (la pila)

La pila (stack) es una zona de la memoria donde podemos guardar las variables cuyo tamaño es fijo y conocido desde el principio. Recuerda (esto es importante) **en la pila solo se pueden almacenar datos que tienen un tamaño fijo**.

Por ejemplo, las variables locales de una función, los parámetros que pasamos a una función.

La pila es un tipo de memoria muy rápida que funciona con el método LIFO (Last In First Out). Es decir, que el último valor que metemos es el primero que sacamos.

Cuando metemos un valor en la pila decimos que hacemos **push** y cuando sacamos un valor decimos que hacemos **pop**.

Heap (el montón)

No tiene orden. Puedes solicitar un tamaño determinado. Memory allocator. Punteros. Lento porque tiene que buscar el dato.

Entra en acción el tipo String

El tipo **String** es un tipo de dato especial. Se usa para almacenar cadenas de texto y tiene varias peculiaridades que no afectan a otros tipos de datos que hemos visto.

Veamos un ejemplo de uso de un *String* creando la variable `nombre`:

cap-12/ejemplo-01/src/main.rs

```
fn main() {
    let nombre = String::from("Gorka");

    println!("Hola {nombre}.");
}
```

Aquí vemos cosas nuevas:

```
String::from("Gorka");
```

Por ahora basta con saber que cuando veas una construcción como ésta quiere decir: Usar la función `from` que se encuentra en `String`. Y esto de `String` por ahora podemos imaginarlo como un lugar donde tenemos definidas unas cuantas funciones relacionadas con el manejo de cadenas.

Esta función `from` lo que hace es crearnos una variable de tipo `String` a partir de la frase "Gorka". Vamos, que nos crea una variable que contiene la frase "Gorka".

Los Strings son mutables

Una cadena de caracteres (o String) puede ser mutable añadiendo a nuestro querido `mut`:

cap-12/ejemplo-02/src/main.rs

```
fn main() {
    let mut nombre = String::from("Gorka");

    nombre.push_str(" Urrutia");

    println!("Hola {nombre}.");
}
```

Venga, más cosas nuevas:

```
nombre.push_str(" Urrutia");
```

Hemos visto arriba que `String::from()` nos creaba una variable de tipo `String`. Pues resulta que las variables de este tipo tienen varias funciones asociadas (como la de `from` que ya habíamos visto). Si

tienes curiosidad por saber qué otras funciones tiene String puedes echar un vistazo aquí.

Otra de esas funciones que tiene asociadas es ésta de `push_str`. Lo que hace es añadir a nuestra cadena otra cadena. En este caso le añade mi apellido.

Un detalle importante, cuando una función está dentro de estas "agrupaciones" nos referimos a ella como **método (method)** en lugar de función.

Para usar un método que tiene asociado una variable basta con usar un punto (.) y el nombre del método. ¿Hacía falta decir ésto?

Ya lo veremos más adelante, pero es posible que te preguntes por qué unas veces usamos:

```
String::from("Gorka");
```

y otras:

```
nombre.push_str(" Urrutia");
```

Pues esto es porque en cuanto definimos la variable `nombre` ésta ya tiene asociados sus "métodos String". Pero cuando la estamos definiendo no tenemos otra forma de indicar que queremos usar el método `from` que hay definido en `String`.

Las cadenas y sus cosas

Len y cap

¿Dónde se guardan?

https://stackoverflow.com/questions/47179667/is-a-string-array-e-g-string-3-stored-on-the-stack-or-heap

fn main() { let cadena_original = String::from("Hola"); println!("cadena_original = {cadena_original}"); println!("cadena_original = {}", cadena_original.len()); println!("Dirección donde se almacena la variable cadena_original = {:p}", &cadena_original); println!("Dirección donde se almacena la cadena = {:p}", cadena_original.as_ptr()); }

Propiedad (Ownership)

Una vez entendido bien que hay dos tipos de memoria que podemos usar (*stack* y *heap*) vamos a meternos con el concepto de **ownership** (propiedad) que seguro que, como a mí, al principio te resultará muy extraño.

En Rust tenemos este peculiar concepto de propiedad que tiene una serie de **reglas** que conviene recordar:

- Cada valor en Rust tiene un propietario (owner).
- Solo puede haber un propietario cada vez.
- Cuando un propietario deja de existir su valor también.

Esto me recuerda a esas culturas en las que al fallecer una persona se la entierra con sus propiedades (incluidas las personas de "su propiedad").

Moviendo variables (move)

Vamos a ver qué es esto de la propiedad y cómo nos afecta. Empecemos con un sencillo ejemplo en el que tenemos dos variables `x` e `y` donde `y` copia el valor de `x`.

cap-13/ejemplo-01/src/main.rs

```
fn main() {
    let x = 10;
    let y = x;

    println!("x = {x}, y = {y}");
}
```

Como podrás comprobar esto compila sin problemas. Ahora vamos a hacer lo mismo pero con dos Strings:

cap-13/ejemplo-02/src/main.rs (no compila)

```
fn main() {
    let s = String::from("Hola");
    let s2 = s;
    println!("s = {s}, s2 = {s2}");
}
```

Este ejemplo no se puede compilar porque Rust se queja de que el valor de `s` se ha movido a `s2`. ¿Y qué es eso de que se ha movido? Pues porque *Rust* se preocupa de que no hagamos las cosas mal (aunque parezca que nos está complicando la vida gratuitamente).

Las variables que se guardan en la *heap* pueden ser problemáticas y gestionarlas no es algo trivial. Para evitarnos problemas *Rust* nos impide hacer chapucillas con ellas. Una de las cosas que hace

Rust es que cuando asignamos el valor de una variable de este tipo a otra variable lo que hacemos en realidad es **moverla**. En este caso decimos que movemos el contenido de la variable `s` a `s2`.

Como el valor se ha movido a `s2`, ya no tiene sentido usar `s` y de ahí este error.

Si queremos que el programa compile tendremos que eliminar las referencias a `s`:

cap-13/ejemplo-03/src/main.rs

```
fn main() {
    let s = String::from("Hola");
    let s2 = s;
    println!("s2 = {s2}");
}
```

Clonar una variable

Pero ¿y si queremos seguir usando tanto `s` como `s2`? Buena pregunta. En ese caso lo que tenemos que hacer es clonar la variable:

cap-13/ejemplo-04/src/main.rs (no compila)

```
fn main() {
    let s = String::from("Hola");
    let s2 = s.clone();
    println!("s = {s}, s2 = {s2}");
}
```

Aquí le estamos diciendo al compilador: "Oye *Rust*, quiero que me copies el valor de `s` en `s2` para poder seguir usando las dos variables".

Copiado automático

Una buena pregunta ¿Por qué no había este problema en el ejemplo del principio? Porque lo que hemos visto de mover variables afecta solo a las variables que se almacenan en la heap, no las de la pila.

Las variables que se guardan en el *stack* se copian siempre directamente. Implementan el *trait* `copy()` (ya lo veremos más adelante).

Para tener una referencia estas son las varaibles que incluyen el `copy()` y de las que no tenemos que preocuparnos por esto de que se *mueven* sus valores:

- Tipos enteros (u8, u16, u32, etc y i8, i16, i32, etc).
- Tipo *bool*.
- Coma flotante (por ejemplo *f64*).
- Tipo *char*.

- Tuplas que incluyen unicamente los tipos indicados en esta lista.

Patrocinadores

Urlan Heat

Este libro por ahora tiene un único pero inigualable patrocinador: Urlan Heat. Es, sin duda, la empresa de desarrollo de software más importante del mundo. Lo es, al menos, para mí y para mis amistades y mi familia.

https://urlanheat.com/blog/quienes-somos/

Términos y definiciones

Heap

Es un tipo de memoria donde se almacenan las variables cuyo tamaño no se conoce a priori. Más información aquí: Heap (el montón).

Stack

Es un tipo de memoria donde se almacenan las variables cuyo tamaño se conoce a priori. Más información aquí: Stack (la pila).

Rust

¿En serio has buscado qué significa *Rust* en un libro sobre *Rust*?