Programando en Rust

Por : lae-laps

Bases

Esta es una pequeña guía sobre el lenguaje de programación rust. Está basada en la guía oficial de rust y sigue mucha de su estructura y contenidos. Como pequeña introducción al lenguaje en si, se trata de un lenguaje compilado y que aplica los paradigmas tanto funcionales como de orientación a objetos. Es relativamente nuevo, pero esta ganando fama rápidamente, por lo que es muy recomendable empezar a aprenderlo ahora.

Este lenguaje normalmente está clasificado entre c++ y java, ya que comparte la gran mayoría de funcionalidades de bajo nivel de c++ con la sintaxis organizada de java. En general es un lenguaje bastante bueno para aprender y muy versátil. Se suele aplicar en muchos campos desde el desarroyo de aplicaciones web hasta la creación de window managers, sistemas operativos o kernels enteros como son redox por ejemplo. También es muy adecuado para crear aplicaciones de command line, y aunque por el momento no tiene muchos frameworks popularizados y estandarizados, se puede usar para crear aplicaciones GUI.

Instalación

Rust es un programa compilado como hemos dicho antes. Por esto, lo que tenemos que instalar es su compilador para poder compilar los programas que iremos haciendo. Rust cuenta con una herramienta oficial llamada rustup para la instalación del compilador y otras utilidades comunes. Yo recomiendo el uso de esta herramienta ya que nos instalará otras herramientas que usaremos más adelante, aunque también es posible instalar rust a través de el paquete normal.

Para instalar rustup, podemos instalarlo a partir del paquete oficial, por ejemplo en arch linux : `sudo pacman -S rustup --needed`. La otra forma que es más recomendable es a través de un script de automatización. Se haría de la siguiente manera

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Después de haber instalado rustup, tenemos que usar los siguientes comandos para que rustup instale rust y el resto de herramientas que queremos

```
rustup update
rustup install stable
rustup default stable
rustc --version
```

El último comando debería darnos una version. Si lo hace, hemos logrado instalar rust correctamente.

Para instalar rust manualmente, solamente tenemos que instalar el paquete "rust", que vendrá por defecto en el gestor de paquetes de la mayoría de las distribuciones de GNU/Linux. Por ejemplo, en arch linux usaríamos el comando : `sudo pacman -S rust --needed`

Esto nos instalaría el compilador de rust. Para futuras referencias a herramientas adicionales, se explicará también su instalación, probablemente para arch linux.

En cuanto al editor de texto, se recomienda el uso de VSCode o vim, aunque no tiene mucha importancia ya que todos bienen a ser más o menos lo mismo si consideramos los programas que producimos. La compilación de los programas se hará enteramente a través de la terminal.

Intellij IDEA es una muy buena opción si preferimos un entorno más completo, ya que aunque esta principalmente orientado a java, con unos cuantos plugins se puede convertir en un IDE completo para rust. Para hacer esto solo hay que instalar el plugin para rust y reiniciar nuestro IDE. Al crear un nuevo proyecto se nos preguntará en que lenguaje queremos crearlo, y rust será una opción.

Hola Mundo

Es ya tradición que la primera cosa que se haga al comenzar en un nuevo lenguaje de programación, sea un pequeño programa, el "Hola mundo". Básicamente se trata de un programa que imprime esas palabras en pantalla.

Compilación

Primero de nada, vamos a abrir una terminal y vamos a crear un directorio para guardar todo nuestro código de rust (Esto no es necesario, pero es recomendable para mantenerlo todo organizado), luego navegaremos a este directorio y crearemos un archivo con el nombre que queramos, por ejemplo, hola_mundo.rs . Es importante que tenga la extensión.rs ya que esto indica que es un archivo de código fuente de rust.

Todo este proceso se haría así desde una terminal de unix estandard :

```
cd
mkdir rust
cd rust
touch hola_mundo.rs
```

Ahora que ya tenemos nuestro archivo, vamos a introducir este código en el. Para ello lo abriremos con nuestro editor de texto. Por ejemplo si usamos vim haríamos : `vim hola_mundo.rs` , mientras que si usamos VSCode : `code hola_mundo.rs` . Se haría lo mismo para nano, atom o casi cualquier otro editor sustituyendo el comando por el apropiado del editor en concreto. De aquí en adelante se asume que se sabe como abrir archivos desde la terminal.

```
fn main() {
    println!("Hola, Mundo")
}
```

Ahora guardamos nuestro archivo y volvemos a la terminal. Vamos a compilarlo. En el proceso de compilación, un software llamado compilador convierte nuestro código fuente a código máquina que puede ser ejecutado. Para llamar al compilador de rust, usaremos el comando rustc <filename.rs>, por ejemplo, haríamos `rustc hola_mundo.rs` .

Acabamos de compilar el archivo que contiene el código fuente, y si ahora introducimos el comando `ls` , para ver el contenido del directorio en el que estamos, deberíamos ver que hay un nuevo archivo llamado hola_mundo. Este archivo es lo que se llama un ejecutable, es el equivalente de un .exe en windows (en unix no necesitan una extensión para funcionar) . Si lo abrimos con un editor de texto, no entenderemos nada ya que está compuesto principalmente por caracteres no imprimibles que son legibles para el ordenador en forma de instrucciones.

Lo importante es que ahora podemos ejecutar este archivo, y se ejecutará el código que hemos escrito antes. Para ello, solo tenemos que poner ./nombre_de_archivo, o en este caso `./hola_mundo` . Esta es la forma en la que se ejecuta cualquier archivo con permisos de ejecución en linux, donde el ./ representa nuestro directorio actual. De igual manera, podríamos ejecutarlo

con el path absoluto hacia el fichero y no tendríamos que estar en el mismo directorio que el propio archivo de esta manera.

> *Hay que destacar que compilar y ejecutar son dos cosas distintas y que se hacen por separado*

Estructura del Hola Mundo

Al hacer esto, en la terminal, deberíamos ver que salen las palabras *"Hola, Mundo"* ya que lo que hace nuestro programa hasta ahora es imprimir esto. Vamos a analizarlo parte por parte. Lo primero de todo es esto: fn main() {}

Esta es la manera en la que se declara una función. En este caso la función de nombre main, el cual es un nombre reservado para la primera función que se ejecuta en cualquier programa de rust. la keyword fn nos dice que es una función, y los parentesis será donde más adelante pasemos lo que llamamos argumentos. Después de los paréntesis vemos unas llaves. En el programa inicial estas llaves acaban en la última línea. Básicamente estas llaves engloban todo el código de la función, se abren en la primera línea donde hemos visto y se cierran al final de todo. El código de la función puede ir indentado. Aunque a la hora de compilarlo no hay ninguna diferencia, ayuda enormemente a leerlo y es muy buena práctica indentarlo siempre.

Si ahora vamos a la segunda línea podemos ver la keyword `println` seguida de un símbolo de exclamación (!). El símbolo de exclamación nos indica que se trata de una macro, ya que si fuera una función, se pondría sin el símbolo de exclamación. Después podemos ver el texto Hola, Mundo entre comillas y paréntesis. Las comillas nos indican que es una string, y los paréntesis que es un argumento del macro println! . En esencia podemos ver que println! es el equivalente a print() en python, echo en bash o std::cout << en c++.

Cargo

Cargo es la herramienta más usada para compilar y administrar proyectos en rust. Rustup la instala por defecto, pero para comprobar si está instalada podemos usar el comando `cargo --version` que devolverá la versión de cargo que tenemos instalada. Cargo puede compilar automáticamente nuestros proyectos e instalar todas las librerías de las que dependen, llamadas dependencias. Para cualquier proyecto medianamente serio es casi necesario el uso de cargo, así que se explicará su uso aquí y se asumirá su uso en el resto de esta documentación.

Crear un proyecto con Cargo

Vamos a empezar por crear un nuevo proyecto con cargo. Para esto usaremos el comando `cargo new nombre_de_proyecto` . Por ejemplo podríamos llamarlo ejemplo. Ahora si usamos el comando ls, veremos que cargo ha creado un directorio con el nombre del proyecto que le hemos indicado. Si navegamos dentro de este directorio, veremos un archivo Cargo.toml y un directorio src donde estará el código fuente de nuestra aplicación. Si navegamos dentro de el directorio src veremos que cargo ya ha creado un archivo main.rs por defecto.

Cuando creamos un proyecto de cargo como acabamos de ver, cargo también inicializa un repositorio de git por defecto. Esto se puede cambiar para que use otra herramienta de control de versiones o que no use ninguna. Para ver todas las opciones que nos da cargo al crear un nuevo proyecto podemos hacer `cargo new --help`, pero como nota rápida, la manera en la que se crea un nuevo proyecto sin usar git ni nada por el estilo sería añadiendo esta flag de esta manera: `cargo new --vcs=git` .

Empecemos por abrir el archivo llamado *Cargo.toml* como se ha explicado antes. Este es el archivo de configuración de cargo para nuestro proyecto. la extensión toml son siglas en inglés para *"Tom's Obvious, Minimal Language"* que viene a significar "El lenguaje obvio y minimalista de Tom". A pesar de lo que indica su nombre no llega a ser un lenguaje y es más una sintaxis bastante estandard para determinar ciertos parametros que cargo necesita. Al abrir el archivo deberíamos ver algo así :

```
[package]
name = "hello_cargo"
version = "0.1.0"
edition = "2021"

[dependencies]
```

Cada línea que tiene una palabra entre [] es lo que llamamos una sección o cabecera. Tenemos dos secciones en este ejemplo por ahora, pero más adelante veremos como añadir más.

En la primera sección tenemos tres lineas que dan la información que rust necesita para compilar nuestro programa. Esto es el propio nombre, la versión de nuestro programa y la versión de rust con la que compilar. La segunda sección esta vacía. Aquí es donde pondremos más a delante nuestras dependencias cuando las tengamos, y rust se ocupará de hacer todo el trabajo por nosotros.

Ahora abriremos el archivo *main.rs* dentro del directorio src. Para hacer esto debemos entrar al directorio con `cd src` y luego abrir el archivo como se ha explicado antes. Dentro deberíamos ver un código muy similar al que hemos creado al principio con el hola mundo, solo que será *hello world* que es el equivalente en inglés.

Como breve explicación, podemos decir que cargo presupone que todo nuestro código va a estar siempre en /src/ (abreviatura de *"source"* en inglés que significa fuente o código fuente). El resto de archivos o información como los ficheros de configuración como el propio *Cargo.toml* o READMEs deben estar en el directorio principal del proyecto.

También es posible convertir un proyecto de rust creado en un principio manualmente a uno de cargo. Para hacer esto solo hay que poner todo el código fuente en un directorio src y crear un archivo de Cargo.toml funcional.

Compilación y Ejecución de proyectos cargo

Ahora vamos a ver la manera en la que compilaremos todo nuestro código fuente desde cargo. Para esto, nos situaremos con la terminal en el directorio principal de nuestro proyecto y usaremos el comando `cargo build` . Esto compilará nuestro código fuente en /src/ y generará un ejecutable. Este ejecutable lo podemos encontrar en */target/debug/proyecto* donde proyecto es el nombre que le hemos dado al proyecto. Al compilar nuestro código fuente, cargo genera nuevos directorios y archivos como acabamos de ver. Si ahora queremos ejecutar esto podemos hacer `./target/debug/proyecto` y se debería ejecutar el código compilado. Si no hemos tocado main.rs, por defecto debería imprimir en pantalla "Hello, world!" como hemos visto antes.

Una forma más rápida de hacer todo esto es con el comando `cargo run` . Este comando compila y ejecuta nuestro código automáticamente por lo que es muy útil para probarlo automáticamente. Estos dos comandos también detectan si se ha realizado algún cambio al código fuente, y en caso contrario, no recompilan todo. Esto nos ahorrará tiempo más de una vez, y es que cuando nuestros programas se hagan más grandes, los tiempos de compilación también.

También tenemos el comando `cargo check`. Este comando comprueba si nuestro código tiene algún error de compilación sin llegar a compilarlo en si, por lo que es mucho más rápido y nos vendrá muy bien cuando tengamos que comprobar si hay errores continuamente. Es bueno usarlo de vez en cuando mientras programamos para asegurarse de que no hay grandes errores.

Por último, es muy importante mencionar que una vez nuestro código funcione y queramos producir u ejecutable final, debemos usar `cargo build --release` ya que esto creará un ejecutable optimizado que será más rápido que los que produzcamos con *cargo build* sin mas. Sin embargo, mientras aún estamos programando, nos basta con *cargo build* sin más porque no necesitamos los ejecutables más optimizados solo para saber si nuestro código funciona. Compilar optimizadamente llevará más tiempo pero no importa porque solo tenemos que hacerlo una vez.

Aunque para el código que tenemos hasta ahora cargo no nos ayuda demasiado, para proyectos algo más grandes donde se usen dependencias, cargo va a ayudarnos muchísimo. Una cosa muy buena que tiene es que es igual para todo y todos incluso entre sistemas operativos, así que si encontramos un proyecto open source en rust que nos guste por ejemplo en github, y si está hecho con rust (cosa que es casi segura) simplemente tenemos que entrar en el directorio y hacer cargo build para tener el proyecto compilado, por lo que simplifica mucho el proceso de probar código, y lo que es más importante, lo hace uniforme para todo el mundo.

Primer pequeño programa en rust

Ahora crearemos un pequeño juego en rust y analizaremos el código. Para empezar, crearemos un nuevo proyecto de cargo con cargo new como hemos visto antes. Lo podemos llamar juego por ahora aunque el nombre no importa. Luego iremos dentro del directorio con cd. Abriremos el arcivo main.rs con nuestro editor de texto y pondremos dentro este código :

```
use std::io;

fn main() {
    println!("Acierta el número");
    println!("Introduce tu número:");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("Tu número fue: {}", guess);
}
```

Esta es la primera parte de nuestro código que ahora explicaremos. Por ahora nuestro código lo que hace es preguntar por un numero y imprimir el número que se inroduzca. Vamos a ver el código línea por línea.

En la primera línea tenemos use `std::io;` . Aquí estamos usando el tipo io que es la librería para input y outputs de ahí su nombre. Esta es parte de la librería estandard o std, pero no se invoca automáticamente en todos los programas por lo que debemos usarla con la keyword use. Esto nos permitirá hacer inputs más adelante.

En la segunda línea tenemos la función main() que ya vimos antes. Esta función es necesaria para declarar cual es la parte principal de nuestro programa.

Después de esto vemos dos lineas con el macro printl! que también vimos en nuestro programa de hola mundo. Básicamente este macro imprime en pantalla lo que se le pase como argumento.

Variables

Como para muchos otros conceptos en esta guía, se asume que se tiene un conocimiento mínimo acerca de algunos conceptos básicos en programación entre los que se encuentran las variables, pero como explicación muy simple, una variable es una forma que tenemos de guardar un valor de tipo determinado en programación.

Ahora en nuestro programa tenemos una declaración de una variable. Esta es `let mut guess = String::new();` . Podemos ver la keyword *"let"* que también se usa en otros lenguajes como JavaScript o Swift. Un ejemplo de una variable mucho más fácil de entender sería :

```
let dias = 23;
```

Aquí asignamos el valor numérico 23 a la variable de nombre *"dias"*. Y como se ve, no se admiten ciertos caracteres como los acentos o la ñ en los nombres de las variables.

Volviendo a nuestro ejemplo original, vemos que entre la keyword *let* y el nombre de nuestra varíable, "guess", tenemos otra keyword, *mut*. En rust, las variables por defecto no son *"variables"*, es decir, que su valor no puede ser cambiado una vez son definidas. Para cambiar esto, se usa la keyword *mut* que las hace *"mutables"* y nos permite cambiar su valor más a delante en el programa. Si aplicamos esto a nuestro ejemplo de antes, para hacer la variable mutable haríamos :

```
let mut dias = 23;
```

Ahora que ya conocemos algunos conceptos básicos, podemos volver a nuestro programa. Vemos que se está creando una variable mutable llamada guess, o acierto. Esto ya lo comprendemos, pero ahora veremos el valor que se le está asignando. Este es String::new(); . Aquí, estamos llamando al método new de la clase String. Esta pertenece a la librería estandard de rust y es una cadena de caracteres que se puede editar con encoding de UTF-8 o unicode por defecto. El método new de esta clase simplemente devuelve una nueva de estas cadenas, o strings vacía, por lo que nuestra variable tiene una string vacía asignada como valor.

Muchas clases que representan tipos de datos de la librería estandard tienen métodos new que hacen lo mismo que este, devuelven un dato de su tipo vacío. Más adelante veremos muchos ejemplos de esto.

Input

```
io::stdin()
    .read_line(&mut guess)
    .expect("Failed to read line");
```

Ahora pasemos a las siguientes tres lineas de nuestro programa. Estas tres lineas en realidad son funcionalmente una sola, pero es una buena práctica separar las líneas largas de nuestros programas en rust con una indentación, por los métodos, como se ve aquí. La función que cumplen estas tres lineas es la de recojer el input del usuario y guardarlo, o adjuntarlo a la variable que acabamos de crear.

Para empezar podemos ver que estamos llamando a la librería *io* que mencionamos al principio del programa. Si no hubiesemos importado esta librería al principio de nuestro programa, tendríamos que llamar al método completo ahora, lo que se haría con `std::io::stdin()`. Dentro del módulo de *io* que es la librería estandard para input y output, estamos llamando a stdin, que es un método que devuelve un handel al input estandard de la terminal del sistema. El siguiente método, `.read_line(&mut guess)`, que recoje el input de la terminal. A este método se le pasa un argumento, el cual es una referencia o puntero a la variable en la que se guardará el input que recoja.

El símbolo de & es el que indica que se trata de una referencia o puntero a la variable. El usar punteros nos permite acceder a un valor desde varias partes de nuestro código sin crear varias instancias de este valor en memoria, osea, copiar y pegar el valor en varias variables. También podemos ver la keyword mut en el putero, y esto es porque como las variables, los punteros son inmutables por defecto y tenemos que usar la keyword mut para hacerlos mutables.

Por último tenemos esto `.expect("Failed to read line");`. Antes hemos mencionado que read_line() mete en una string que se le pasa como argumento lo que sea que recoja como input, pero al ser un método, también tiene la capacidad de devolver un valor al igual que las funciones, como se hace en otros lenguajes. En este caso, el método read_line() devuelbe un valor del tipo `io::Result`. Su tipo es como decir una string o un numero entero, pero en este caso es otro que es un tipo propio de rust llamado así.

Rust tiene varios de estos tipos en su librería estandard, por ejemplo el *Result* general, aunque tiene también otros tipos para librerías específicas, como este de aquí que pertenece a la librería *io*. Los tipos Result son lo que se llaman *enumeraciones* o *enums*. Un ejemplo de otros lenguajes para una enumeración sería una booleana, donde el valor solo puede ser Verdadero, Falso o Ninguno (True / False / None). Los valores que puede tener una enumeración se llaman variantes de esta. En el caso de Result en rust, sus variantes son *Ok* y *Err*. Como se puede deducir por sus nombres, Ok indica que la operación se completó sin problemas, mientras que Err indica que hubo un error. Si se da el caso de error, vendrá información acerca del error específico acompañada de el valor de Err.

> Lo que hay que tener en cuenta aquí, especialmente para la gente que está acostumbrada a la sintaxis de python, es que en python la función input equivalente a read_linDoes IntelliJ Rust use rust analyzer?e(), devuelve el input que recoje, mientras que aquí la función read_line devuelve Err o Ok, y el input que recoje se guarda en la variable a la que se le pasa un puntero como argumento. En python, al input se le pasa como argumento el prompt, pero aquí, tendremos que imprimir en pantaya el prompt antes simplemente.

El método expect en si, es simplemente un método que tienen los tipos Result. Este método ve si result devuelve un Err o Ok. En caso de ser un Err, expect crea un crash del programa e imprime en pantaya los detalles del error, mientras que si se trata de un Ok, devuelve el valor que Ok contiene, en este caso sería el número de bytes del input del usuario. No es necesario llamar a expect, pero se nos dara un aviso o warning en tiempo de compilación si no lo hacemos.

Uso básico de println!

Por último, nuestra última línea es un *println!()*. Este println usa valores externos. Para hacer esto usamos algo llamado placeholders. Como podemos ver en el contenido de la string que imprime este println (`println!("Tu número fue: {}", guess);`), tenemos dos corchetes. Estos corchetes son nuestro placeholder, y podemos tener varios en nuestra string. La idea es que ponemos esto donde queremos insertar nuestro valor, y luego pasamos nuestro valor como un segundo argumento. Es decir, si tenemos 3 corchetes en la string de el primer argumento, el

segundo argumento será correspondiente a los primeros corchetes, el tercero a los segundos y el cuarto a los terceros.

Veamos un ejemplo:

```
fn main() {

    let manzanas = 5;
    let naranjas = 7;

    println!("Tengo {} manzanas y  {} naranjas.", manzanas, naranjas);
}
```

Si compilamos y ejecutamos este código, veremos que nos da el output *"Tengo 5 manzanas y 7 naranjas."*. Podemos ver que esto es porque el primer placeholder corresponde a la variable manzanas, y representa su valor, 5 y el segundo a naranjas, que es 7.

Usando dependencias

Ahora ya comprendemos todo el código que llevamos hasta ahora, así que vamos a ver la última cosa que necesitamos para acabar nuestro programa. Esto es generar un número aleatorio cada vez. Rust cuenta con librerías para generar números aleatorios pero estas no están en la librería estandard, por ello hemos de incorporarlas a nuestro programa y descargarlas como dependencias. Aquí es donde entra cargo, y como vimos antes, tenemos una sección de dependencias en nuestro *Cargo.toml*.

Abriremos este archivo y nos dirijiremos a la parte de *[dependencies]* que está vacía por defecto. Aquí pondremos nuestras dependencias. En este caso la librería que necesitamos se llama rand, por lo que la especificaremos junto a su versión, así `rand = "0.8.3"` una línea por debajo de la cabezera de dependencies. El modo de notación de las versiones de las librerías que usa cargo se llama versionado semántico. En este caso, cargo buscará la versión mas alta posible que 0.8.3 pero más pequeña que 0.9.0. Estas versiones siempre serán compatibles con nuestro código por lo que siempre estaremos en la última versión compatible.

Todas estas librerías o crates, podemos encontrarlas en Crates.io, una website donde la gente cuelga sus librerías y crates open source para la comunidad.

El otro fichero que se puede ver al compilar nuestro proyecto es uno llamado *Cargo.lock*. Este archivo guarda las versiones de todas nuestras dependencias cuando las compilamos por primera vez. La próxima vez que compilemos o alguien compile este proyecto sea mañana o en 5 años, cargo se descargará las dependencias de la version de la primera vez de cmpilación al menos que se le indique lo contrario. Esto hace que nuestro código no se rompa con las nuevas actualizaciones de ls librerías.

Para hacer que cargo sí busque las dependencias más nuevas según nuestras especificaciones, podemos usar el comando `cargo update`, el cual hará esto y mandará las nuevas versiones al *Cargo.lock* donde quedarán guardadas otra vez.

Si compilamos ahora nuestro proyecto con cargo build, veremos que tarda bastante más y que se compilan un monton de archivos. Estos archivos son los de la librería rand y sus dependencias a su vez. Cargo descarga su código fuente y despues lo compila para que lo podamos usar. Esto es porque ahora está descargando las dependencias que le hemos indicado en Cargo.toml. Sin embargo, solo las descargará una vez, al menos que hagamos update, por lo que una vez descargadas nuestro proyecto podrá ser compilado rápidamente de nuevo.

Generación de números aleatorios

Ahora veremos como vamos a integrar la librería rand en nuestro programa.

```
use std::io;
use rand::Rng;

fn main() {
    println!("Adivina el número");

    let random_number = rand::thread_rng().gen_range(1..101);

    println!("Introduce un número : ");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("Tu número era: {}", guess);

    println!("El número secreto era: {}", random_number);
}
```

Primero de todo definimos una nueva variable. No la hacemos mutable ya que no cambiaremos su valor más adelante en el programa. le damos el nombre de random_number ya que será un número que generaremos aleatoriamente. En cuanto al valor que le damos, primero de todo llamamos a la librería rand que hemos instalado antes. Dentro llamamos al método *thread_rng()* el cual nos devuelve un número aleatorio en el rango que se especifique. Para generar el rango usamos el comando *gen_range()* y le pasamos nuestro rango como atributo. Este tiene que estar en el formato de *comienzo..fin*. El comienzo empieza en el número que se especifique, pero el fin acaba un número por abajo. En este caso, 1..101 generaría un número entre 1 y 100.

Comparación entre ambos números

Ahora que nuestro programa ya genera un número aleatorio y recoje nuestro input, la última cosa que nos queda por hacer es comparar ambos para saber si son iguales. Para ello vamos a usar otro módulo de la librería estandard llamado *Ordering*. Es otra enumeración como las que vimos antes. Tiene tres variantes, Less, Greater y Equal, osea, mayor, menor e igual.

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {

    // Nuestro código de antes

    match guess.cmp(&random_number) {
        Ordering::Less => println!("Muy pequeño"),
        Ordering::Greater => println!("Muy grande"),
        Ordering::Equal => println!("Acertaste!"),
    }
}
```

Para empezar añadimos un use statement de el módulo. Después llamamos al módulo cmp que compara las variables guess y random_number a la que pasamos un puntero. Debajo de esto podemos ver las tres posibilidades que hay, y el código que se ejecutará en cada una de ellas después del => . El método match aquí es una función muy útil que tenemos en rust. Imaginemos que ejecutamos nuestro programa y introducimos como número 64, mientras que el número aleatorio es 20. En este caso, cmp devolvera un *Ordering::Greater* porque 64 es más que 20. Entonces match ira a la primera de nuestras 3 lineas de orderings a las que se llaman arms. Vera que es Ordering::Less y como no es lo que busca pasará a la siguiente. Aquí sí encontrará a Ordering::Greater que es lo que busca. Por lo tanto, ejecutara la línea de código siguiente, que es un println!() diciendo que el número es muy grande.

Nuestro programa esta casi completo, pero en su estado actual, nos dará un error de compilación.

```
$ cargo build
   Compiling libc v0.2.86
   Compiling getrandom v0.2.2
   Compiling cfg-if v1.0.0
   Compiling ppv-lite86 v0.2.10
   Compiling rand_core v0.6.2
   Compiling rand_chacha v0.3.0
   Compiling rand v0.8.3
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
  --> src/main.rs:22:21
   |
22 |     match guess.cmp(&random_number) {
   |                     ^^^^^^^^^^^^^^^ expected struct `String`, found integer
   |
   = note: expected reference `&String`
              found reference `&{integer}`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `guessing_game` due to previous error
```

Si leemos nuestro error dice que no ha podido comparar los dos valores porque uno era un número entero y otro una string. Si nos fijamos en la línea en la que definimos la variable guess, vemos que le damos una string vacía de valor. Rust es un lenguaje fuertemente tipado pero infiere los tipos. Esto quiere decir que no es necesario decirle el tipo de una variable al definirla como haríamos en c, y la infiere el compilador como en python. Para solucionar nuestro problema, la solución más fácil es convertir la variable guess a tipo int.

En rust tenemos varios tipos de números enteros. Por defecto son i32, números de 32 bits, pero se les puede indicar que sean i64 (de 64 bits) o u32 que sería un número unsigned de 32 bits. Ahora que sabemos esto, vamos a hacer algo llamado shadowing. Esto se trata de redefinir una variable con el mismo nombre pero diferente valor. Usaremos esta línea.

```
let guess: u32 = guess.trim().parse().expect("Introduzca un número");
```

Esta linea (que meteremos en nuestro programa después de nuestro input) redefine la variable guess que ya hemos definido en nuestro programa, pero le da un tipo de u32, osea, número de 32 bits unsigned. El que sea unsigned significa que solo se admite 0 y números positivos entre sus valores. A la variable le damos el valor guess.trim().parse() . Ese guess hace referencia al guess original en forma de string en nuestro programa. El método trim() elimina cualquier espacio en

blanco en el principio o final de una string, lo cual es necesario para nuestra conversión. Esto es porque cuando el usuario introduce su número, tiene que darle a enter para que read_line lea el valor y continúe el programa, por lo cual se introduce un caracter de newline en el input recojido. Esto significa que si el usuario mete un 10 y luego le da enter, la string generada sería "10\n" (ese es el caracter de nueva línea). El método trim eliminará este caracter.

Después tenemos el método parse el cual se encarga de convertir nuestro valor a uno de los tipos de números de rust. Como hay muchos tipos de números en rust, es necesario especificar a cual queremos que se convierta, como hemos hecho al decirle : u32. El u32 es una buena opción para guardar números pequeños que sabemos que siempre serán positivos. El compilador de rust también deducira que random_number debería ser un u32 y por lo tanto se hará una comparación entre tipos iguales.

Como hemos dicho antes, la string que creamos es una UTF-8 por defecto, lo que significa que admite todos los caracteres unicode. Esto incluye todo desde caracteres chinos hasta la ñ o incluso emojis. La gran mayoría de estos caracteres no pueden ser convertidos a números, por lo que el método parse() es bastante propenso a fallar. Es por esto que también incluimos el método expect() que ya vimos antes. parse() también nos devuelve un valor Result, y expect se encarga de manejarlo.

Bucles

Técnicamente nuestro juego ya funciona, pero solo se puede intentar acertar 1 vez el número. Es por esto que ahora introduciremos los bucles para que se pueda hacer varias veces hasta acertar.

```
    // Principio del programa

    loop {
        println!("Introduzca su número : ");

        // Más código

        match guess.cmp(&random_number) {
            Ordering::Less => println!("Muy pequeño"),
            Ordering::Greater => println!("Muy grande"),
            Ordering::Equal => println!("Acertaste!"),
        }
    }
}
```

Añadiremos este código a nuestro programa. Podemos ver la keyword loop que crea un bucle infinito por defecto. Esto significa que nuestro programa nos seguirá pidiendo números infinitamente y comparandolos, incluso una vez adivinemos el número aleatorio. Pero como antes hemos visto, si introducimos un caracter no numérico se produce una excepción y crashea el programa. Podemos usar esto como forma de salir de nuestro programa.

Pero si queremos sofisticar algo más el programa, podríamos hacer que este se cierre después de acertar el número. Para esto tenemos que modificar el código que se ejecuta si los dos números son iguales. Para poner código multilinea aquí usamos llaves.

```
match guess.cmp(&random_number) {
    Ordering::Less => println!("Muy pequeño"),
    Ordering::Greater => println!("Muy grande"),
    Ordering::Equal => {
        println!("Ganaste!");
        break;
    }
}
```

Break

De esta manera. El statement o keyword break, hace que salga del bucle, al igual que en python. Y ya que el bucle es la última parte de la función main, significa que el programa se acaba también. El statement break es muy útil y lo usaremos en muchas ocasiones, pero como norma general, hace que se salga del bucle en el cual se encuentra.

Control de excepciones

La última cosa que vamos a hacer en nuestro programa es que en vez de que se crashee al meter un input no numérico, simplemente lo ignore. Para esto usaremos otra vez match. Solo tenemos que cambiar la line donde se convierte la variable guess de string a u32.

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```

La reemplazaremos por esto. Vemos que en vez del expect que había antes, ahora hay un match, como el que tenemos para comparar los números. Este match lee el valor que devuelve parse, que es un Result (Ok / Err) y hace lo que se le indica en cada uno de los arms. Si parse funciona correctamente, match devuelve el número sin más como vemos. En caso contrario de que haya un error, se usa el statement continue. Al igual que en python, este statement salta a la siguiente iteración de el bucle en el que se encuentra. el _ entre los paréntesis es una expresión catchall que significa que acepte cualquier error, sea cual sea el valor que traiga, si es Err se ejecutará.

Código final

Ahora nuestro juego ya está terminado y podemos compilarlo y probarlo. Como recordatorio debemos ir a el directorio base de nuestro proyecto y hacer el comando `cargo run`, se compilará y ejecutará nuestro juego. Si queremos compilarlo en un ejecutable optimizado para guardar, podemos usar el comando podemos hacer `cargo build --release` como se explicó al principio de todo. Vamos al directorio */target/debug/* y deberíamos ver nuestro ejecutable que ya podemos usar y probar.

Aquí está el código completo que hemos creado :

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    println!("Adivina el número");

    let random_number = rand::thread_rng().gen_range(1..101);
```

```
    loop {
        println!("Introduce tu número : ");

        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("Tu número es : {}", guess);

        match guess.cmp(&random_number) {
            Ordering::Less => println!("Muy pequeño!"),
            Ordering::Greater => println!("Muy grande"),
            Ordering::Equal => {
                println!("Ganaste!");
                break;
            }
        }
    }
}
```