

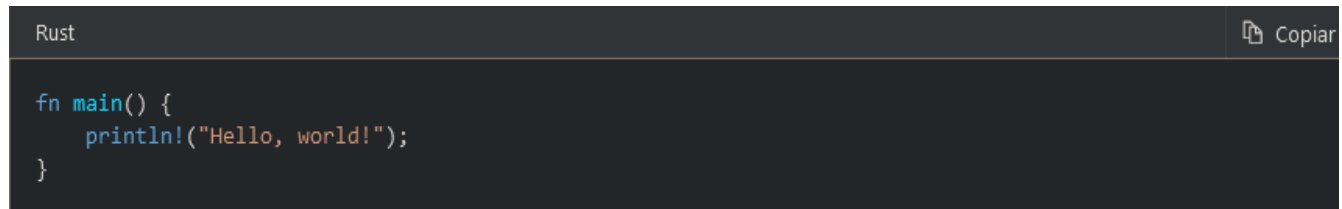
Descripción de la estructura básica de programas de Rust.

En esta unidad, se revisa cómo se estructura un programa simple de Rust.

Funciones en Rust

Una función es un bloque de código que realiza una tarea específica. Separamos el código de nuestro programa en bloques basados en tareas. Esta separación hace que el código sea más fácil de entender y mantener. Después de definir una función para una tarea, podemos llamar a la función cuando sea necesario realizar esa tarea.

Cada programa de Rust debe tener una función llamada `main`. El código de la función `main` siempre es el primer código que se ejecuta en un programa con Rust. Podemos llamar a otras funciones desde la función `main` o desde otras funciones.

A screenshot of a code editor window titled "Rust". In the top right corner, there is a "Copiar" button with a clipboard icon. The editor contains the following Rust code:

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

Para declarar una función en Rust, usamos la palabra clave `fn`.

Después del nombre de la función, se le indica al compilador cuántos parámetros o *argumentos* espera la función como entrada. Los argumentos se enumeran entre paréntesis `()`. El *cuerpo de la función* es el código que realiza la tarea de una función y se define entre llaves `{}`. Un procedimiento recomendado consiste en aplicar formato al código para que la llave de apertura del cuerpo de la función aparezca justo después de la lista de argumentos entre paréntesis.

Sangría del código

En el cuerpo de la función, la mayoría de las instrucciones de código terminan con un punto y coma `;`. Rust procesa estas instrucciones una tras otra, por orden. Cuando una instrucción de código no termina con un punto y coma, Rust sabe que la línea de

código siguiente debe ejecutarse antes de que se pueda completar la instrucción inicial.

Para ayudar a ver las relaciones de ejecución en el código, usamos la sangría. Este formato muestra cómo se organiza el código y revela el flujo de pasos necesarios para completar la tarea de la función. A una instrucción de código inicial se le aplica una sangría de cuatro espacios desde el margen izquierdo. Cuando el código no termina con un punto y coma, a la siguiente línea de código que se va a ejecutar se le aplica una sangría de cuatro espacios más.

Veamos un ejemplo:

Rust

Copiar

```
fn main() { // The function declaration is not indented

    // First step in function body
    // Substep: execute before First step can be complete

    // Second step in function body
    // Substep A: execute before Second step can be complete
    // Substep B: execute before Second step can be complete
    // Sub-substep 1: execute before Substep B can be complete

    // Third step in function body, and so on...
}
```

To do! (macro).

Cuando trabaje en los ejercicios de los módulos de Rust, observará que en el código de ejemplo se suele usar la macro `(todo!)`.

En Rust, una macro es como una función la cual toma un número variable de argumentos de entrada. La macro `(todo!)` se usa para identificar código sin terminar en el programa de Rust. La macro es útil para crear prototipos, o bien cuando se quiere indicar un comportamiento que no está completo.

Este es un ejemplo de cómo se usa la macro `todo!` en los ejercicios:

Rust

Copiar

```
fn main() {
    // Display the message "Hello, world!"
    todo!("Display the message by using the println!() macro");
}
```

Al compilar código en el que se usa la macro `todo!`, el compilador puede devolver un mensaje de alarma en el que espera encontrar la funcionalidad completada:

```
Resultados Copiar
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 1.50s
Running `target/debug/playground`
thread 'main' panicked at 'not yet implemented: Display the message by using the println!() macro', src/main.rs:1:1
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

println! (macro);

Nuestra función `main()` realiza una tarea. Llama a la macro `(println!)` predefinida en Rust. La macro `(println!)` espera uno o varios argumentos de entrada, que se muestran en la pantalla o en la *salida estándar*. En nuestro ejemplo, pasamos un argumento de entrada a la macro, la cadena de texto "Hello, world!".

```
Rust Copiar
fn main() {
    // Our main function does one task: show a message
    // println! displays the input "Hello, world!" to the screen
    println!("Hello, world!");
}
```

Sustitución de valores para argumentos {}

En las lecciones del módulo de Learn de Rust, a menudo llamamos a la macro `(println!)` con una lista de argumentos que incluye cadenas de texto con instancias de corchetes `{}` y otros valores. La macro `(println!)` reemplaza cada instancia de llaves `{}` dentro de una cadena de texto por el valor del argumento siguiente de la lista.

Veamos un ejemplo:


```
Rust Copiar
fn main() {
    // Call println! with three arguments: a string, a value, a value
    println!("The first letter of the English alphabet is {} and the last letter is {}. ", 'A', 'Z');
}
```

Llamamos a la macro `(println!)` con tres argumentos: una cadena, un valor y otro valor. La macro procesa los argumentos por orden. Cada

instancia de llaves {} dentro de una cadena de texto se reemplaza por el valor del argumento siguiente de la lista.

La salida es la siguiente:

Resultados

 Copiar

```
The first letter of the English alphabet is A and the last letter is Z.
```