Problemas de Rust

Guía Práctica de Programación

Una colección curada de 100 problemas

M.H. Alberto

29/12/2025

v1.0

Índice

Problemas 1 - 10 2
Problemas 11 - 20 13
Problemas 21 - 30 24

Problemas 1 - 10

Problema 1 El Contador Inmutable

El siguiente código intenta incrementar una variable, pero el programador olvidó las reglas de mutabilidad. Corrígelo de dos formas distintas en tu mente, pero aquí escribe la solución usando **shadowing** (re-declarando con `let`).

```
fn main() {
    let pasos = 0;
    pasos = pasos + 1;
    pasos = pasos + 1;
    println!("Pasos dados: {pasos}");
}
```

Problema 2 La Constante Prohibida

Rust es muy estricto con las constantes. Encuentra los **dos** errores en este código y corrígelos para que compile.

```
fn main() {
    const mut LIMITE_USUARIOS: u32 = 1000;
    LIMITE_USUARIOS = 1100;
    println!("Límite: {LIMITE_USUARIOS}");
}
```

Problema 3 Shadowing y Tipos Cruzados

El shadowing permite cambiar el tipo de dato. ¿Qué imprimirá este código? Analiza el cambio de `&str` a `usize`.

```
fn main() {
    let dato = "42";
    let dato = dato.len();
    {
        let dato = dato * 2;
        println!("Valor en bloque: {dato}");
    }
    println!("Valor final: {dato}");
}
```

Problema 4 El Techo de Cristal de las Constantes

Las constantes deben conocerse en tiempo de compilación. ¿Por qué falla este código y cómo lo arreglarías si `IVA` debe ser siempre el mismo?

```
fn main() {
    let tasa_dinamica = 0.16;
    const IVA: f64 = tasa_dinamica;
    println!("El impuesto es: {IVA}");
}
```

Problema 5 Variables Olvidadas en Scopes

En Rust, una variable sombreada dentro de un bloque «muere» al salir de este. Predice el resultado de este código:

```
fn main() {
    let nivel = 1;
    {
        let nivel = nivel + 5;
        println!("Nivel interno: {nivel}");
    }
    println!("Nivel externo: {nivel}");
}
```

Problema 6 Mutabilidad vs Shadowing

Si usas `mut`, puedes cambiar el valor pero no el tipo. Corrige este código para que `espacios` pueda pasar de ser un texto a ser un número (su longitud).

```
fn main() {
    let mut espacios = "    ";
    espacios = espacios.len(); // ¡Error de tipos!
    println!("Espacios: {espacios}");
}
```

Problema 7 Sombreado de Sombreado

¿Cuántas veces se ha creado la variable `x` en memoria en este ejemplo y cuál es su valor final?

```
fn main() {
    let x = 2;
    let x = x * x;
    let x = x * x;
    let x = x * x;
    // Pista: No es una mutación, es creación tras creación.
    println!("Resultado: {x}");
}
```

Problema 8 La Constante Global

Las constantes pueden vivir fuera del `main`. Completa el código declarando una constante `PI` de tipo `f32` fuera de la función.

```
// Escribe aquí tu constante

fn main() {
    let radio = 10.0;
    let area = PI * (radio * radio);
    println!("Área: {area}");
}
```

Problema 9 Confusión de Identidad

Este código intenta usar mutabilidad y shadowing al mismo tiempo de forma caótica. ¿Cuál es el valor final de `z`?

```
fn main() {
    let mut z = 5;
    z = z + 1;
    let z = z * 2;
    // z = z + 1; // ¿Qué pasaría si descomento esta línea?
    println!("Z es: {z}");
}
```

Problema 10 El Contrato de la Constante

Dada la siguiente expresión, ¿es válida para una constante? Justifica tu respuesta basándote en que Rust permite «operaciones limitadas» en tiempo de compilación.

```
const MINUTOS_EN_UN_DIA: u32 = 60 * 24;

fn main() {
    println!("Minutos: {MINUTOS_EN_UN_DIA}");
}
```

Problemas 11 - 20

Problema 11 La Inferencia Ambigua

Rust es estáticamente tipado, pero a veces el compilador no puede adivinar qué tipo queremos. Corrige el siguiente código añadiendo una anotación de tipo explícita para que el método `.parse()` sepa a qué tipo convertir el string.

```
fn main() {
    let porcentaje = "85".parse().expect("No es un número");
    println!("El éxito es del {}%", porcentaje);
}
```

Problema 12 El Desbordamiento Silencioso (Release vs Debug)

Basado en la teoría, ¿qué sucede con la variable `byte` si este código se ejecuta en modo **release** (`--release`)? Explica el concepto de **two's complement wrapping**.

```
fn main() {
    let mut byte: u8 = 255;
    byte = byte + 1;
    println!("Valor del byte: {byte}");
}
```

Problema 13 Precisión de Punto Flotante

Rust tiene dos tipos de punto flotante. Declara una variable `precio` con el tipo de **menor precisión** y otra `pi` con el tipo **por defecto** de 64 bits.

```
fn main() {
    // Declara precio (32 bits) y pi (64 bits)

    println!("Precio: {precio}, PI: {pi}");
}
```

Problema 14 Truncamiento en División Entera

Predice el valor de `resultado`. ¿Por qué no es ?

```
fn main() {
    let resultado = 5 / 3;
    println!("El resultado es: {resultado}");
}
```

Problema 15 Caracteres Unicode

El tipo `char` en Rust no es solo ASCII. ¿Cuál es el tamaño en bytes de un `char` y por qué el siguiente código es válido?

```
fn main() {
    let corazon = '❤';
    let jap = '世';
    println!("{corazon} Hola Mundo {jap}");
}
```

Problema 16 Destructuración de Tuplas

Usa el patrón de destructuración para extraer los tres valores de la tupla `datos` en variables individuales llamadas `x`, `y` y `z`.

```
fn main() {
    let datos = (500, 6.4, 1);

    // Escribe la línea de destructuración aquí

    println!("El valor intermedio es: {y}");
}
```

Problema 17 Acceso por Índice de Tupla

Sin usar destructuración, accede directamente al **tercer** elemento de la tupla `empaquetado` usando la sintaxis del punto (`.`) y asígnalo a una variable.

```
fn main() {
    let empaquetado: (i32, f64, u8) = (10, 3.14, 255);

    let ultimo = // Tu código aquí

    println!("El último valor es: {ultimo}");
}
```

Problema 18 Inicialización de Arrays

Escribe de forma concisa (usando la sintaxis `[valor; tamaño]`) la declaración de un array llamado `buffer` que contenga **500 elementos**, todos inicializados con el número **0**.

```
fn main() {
    let buffer = // Tu código aquí

    println!("Primer elemento: {}", buffer[0]);
    println!("Longitud: {}", buffer.len());
}
```

Problema 19 Tipado de Arrays

Corrige la anotación de tipo del array `semana` para que coincida con el contenido y el tamaño correcto.

```
fn main() {
    let semana: [i32; 3] = [1, 2, 3, 4, 5];
    println!("Días procesados: {}", semana.len());
}
```

Problema 20 Pánico en el Índice

Analiza el siguiente código. ¿En qué momento fallará (compilación o ejecución) y qué mensaje de error lanzará Rust para proteger la memoria?

```
fn main() {
    let meses = ["Ene", "Feb", "Mar"];
    let indice = 3;

    let valor = meses[indice];
    println!("El mes es: {valor}");
}
```

Problemas 21 - 30

Problema 21 El Orden de Definición

¿Compilará el siguiente código a pesar de que `saludar` se define después de ser llamada? Explica por qué basándote en la teoría de **scopes** de Rust.

```
fn main() {
    saludar();
}

fn saludar() {
    println!("¡Hola desde una función!");
}
```

Problema 22 Anotación de Tipos Obligatoria

El siguiente código falla porque el compilador de Rust no infiere tipos en las firmas de las funciones. Añade el tipo de dato necesario para que acepte un entero de 32 bits.

```
fn main() {
    duplicar(10);
}

fn duplicar(n) { // Error aquí
    println!("El doble es: {}", n * 2);
}
```

Problema 23 Múltiples Parámetros

Crea una función llamada `mostrar_edad` que reciba un carácter (inicial del nombre) y un entero `u8` (edad).

```
fn main() {
    mostrar_edad('G', 25);
}

// Escribe la función mostrar_edad aquí
```

Problema 24 Sentencias vs. Expresiones

Identifica cuál de las siguientes líneas dentro de `main` es una expresión y cuál es una sentencia. Explica la diferencia fundamental según el texto.

```
fn main() {
    let x = 10;          // A
    let y = { x + 5 };   // B
}
```

Problema 25 El Bloque como Expresión

Analiza el siguiente código. ¿Cuál será el valor final de `z`? Presta mucha atención a la ausencia de punto y coma al final del bloque.

```
fn main() {
    let z = {
        let a = 5;
        let b = 10;
        a * b
    };
    println!("El valor de z es: {z}");
}
```

Problema 26 Retorno Implícito

Modifica esta función para que devuelva el área de un cuadrado () sin usar la palabra clave `return`.

```
fn area_cuadrado(lado: i32) -> i32 {
    // Escribe la expresión de retorno aquí
}

fn main() {
    let resultado = area_cuadrado(4);
    assert_eq!(resultado, 16);
}
```

Problema 27 El Error del Punto y Coma

Este código genera un error de «mismatched types». ¿Por qué el compilador dice que encontró `()` cuando esperaba `i32`? Corrígelo.

```
fn suma(a: i32, b: i32) -> i32 {
    a + b;
}

fn main() {
    let s = suma(5, 5);
}
```

Problema 28 Retorno Temprano

Aunque Rust prefiere expresiones finales, a veces usamos `return` para salir antes. ¿Qué imprimirá este código?

```
fn prueba_retorno(n: i32) -> i32 {
    if n > 10 {
        return n;
    }
    n * 2
}

fn main() {
    println!("{}", prueba_retorno(15));
}
```

Problema 29 Unidad como Retorno

Si una función no tiene una expresión de retorno ni la flecha `->`, ¿qué tipo de dato devuelve técnicamente? Escribe el símbolo que lo representa.

```
fn funcion_vacia() {
    // No hace nada
}

fn main() {
    let x = funcion_vacia();
    // ¿Qué tipo es x?
}
```

Problema 30 Símbolos en la Firma

Escribe una función completa llamada `calcular_minutos` que reciba `horas` (i32) y devuelva el total en minutos (i32). Recuerda usar la sintaxis de flecha `->`.

```
// Tu función aquí

fn main() {
    let m = calcular_minutos(2);
    println!("2 horas son {m} minutos");
}
```

Siguiente paso sugerido: Hemos cubierto variables, tipos de datos y funciones (el núcleo de la sintaxis). ¿Quieres que revisemos alguna respuesta o prefieres que pasemos al siguiente tema: Comments and Control Flow (if/loops)?