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

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");
}
```

Problemas 31 - 40

Problema 31 Comentarios de Fin de Línea

Los comentarios pueden ir al final de una sentencia. Limpia el siguiente código: mueve el comentario a la línea correcta y asegúrate de que el nombre de la variable siga la convención `snake_case`.

```
fn main() {
    let NumeroSecreto = 7; // Este es el número para el juego
    // Imprimir el valor
    println!("{NumeroSecreto}");
}
```

Problema 32 Desactivación de Código

Una utilidad común de los comentarios es «comentar» código que causa errores. Comenta **únicamente** las líneas que impiden que este programa compile debido a las reglas de constantes y tipos.

```
fn main() {
    const VELOCIDAD: i32 = 100;
    let x = 10;

    VELOCIDAD = 110;
    x = x + 5;

    println!("{x}");
}
```

Problema 33 El Tipo 'Unit' en Funciones

Cuando una función no devuelve nada, su tipo de retorno es `()`. Basándote en el código de abajo, añade un comentario arriba de la variable `resultado` explicando qué valor técnico almacena.

```
fn emitir_alerta() {
    println!("¡Alerta de sistema!");
}

fn main() {
    let resultado = emitir_alerta();
    // ¿Qué valor tiene 'resultado'? Escríbelo en un
comentario.
}
```

Problema 34 Sombreado y Tipos de Datos

A diferencia de `mut`, el sombreado nos permite cambiar el tipo. Completa el código usando `let` para transformar el string en un entero de 64 bits (`i64`).

```
fn main() {
    let valor = "1000"; // String

    // Transforma 'valor' a i64 usando shadowing
y .parse().unwrap()

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

Problema 35 Sufijos de Literales Numéricos

Rust permite definir el tipo de un número pegando el tipo al literal (ej: `10u8`). Corrige la llamada a la función para que los argumentos coincidan exactamente con los tipos de la firma.

```
fn procesar_datos(a: u16, b: i64) {
    println!("Suma: {}", (a as i64) + b);
}

fn main() {
    // Usa sufijos para pasar un u16 y un i64
    procesar_datos(500, 1000);
}
```

Problema 36 Acceso Compuesto: Tuplas

Las tuplas pueden contener diferentes tipos. Accede al elemento booleano de la siguiente tupla e úsalo para decidir si se imprime el mensaje, usando un comentario para explicar qué índice estás usando.

```
fn main() {
    let configuracion = (true, 10.5, 'A');

    // Accede al índice 0
    let activado =

    if activado {
        println!("Sistema activo");
    }
}
```

Problema 37 Arrays y Longitud Constante

El tamaño de un array debe conocerse en tiempo de compilación. ¿Por qué falla este código? Corrígelo usando una constante para definir el tamaño.

```
fn main() {
    let tamaño = 3;
    let datos: [i32; tamaño] = [1, 2, 3];
}
```

Problema 38 Evaluación de Expresiones en Bloques

¿Cuál es el valor de y? Analiza cuidadosamente los puntos y comas internos.

```
fn main() {
    let x = 5;
    let y = {
        let x = 3;
        x + 1; // ¿Hay un punto y coma aquí?
        x + 5
    };
    println!("El valor de y es: {y}");
}
```

Problema 39 Parámetros de Función y Shadowing

Este código es confuso. Añade comentarios que sigan el flujo del valor de `n` para explicar por qué el resultado final es `10` y no `20`.

```
fn calcular(n: i32) -> i32 {
    let n = n + 5;
    let n = n * 2;
    n
}

fn main() {
    let n = 0;
    calcular(n); // El resultado no se asigna
    println!("n es: {n}");
}
```

Problema 40 Diferencia entre f32 y f64

Por defecto, los decimales son `f64`. Crea una función llamada `promediar` que reciba dos números `f32` y devuelva su promedio. Asegúrate de anotar los tipos en la firma.

```
// Escribe la función aquí

fn main() {
    let a: f32 = 10.0;
    let b: f32 = 20.0;
    println!("Promedio: {}", promediar(a, b));
}
```

Problemas 41 - 50

Problema 41 La Verdad Estricta

A diferencia de otros lenguajes, Rust no convierte números a booleanos automáticamente. Corrige el siguiente código para que el bloque `if` se ejecute si la variable `stock` es mayor a cero.

```
fn main() {
    let stock = 10;

    if stock { // Error de tipo
        println!("Tenemos productos disponibles");
    }
}
```

Problema 42 Asignación Condicional

Usa un `if` como expresión para asignar un valor a la variable `estado`. Si `temperatura` es mayor a 30, el estado debe ser «Caliente», de lo contrario, «Normal».

```
fn main() {
    let temperatura = 35;

    let estado = // Escribe aquí el if como expresión

    println!("El clima está: {estado}");
}
```

Problema 43 El Tipo Coherente en el else

Este código falla porque Rust necesita saber el tipo de la variable en tiempo de compilación. Corrige el error para que ambos «brazos» (arms) del `if` devuelvan el mismo tipo de dato.

```
fn main() {
    let condicion = true;
    let numero = if condicion { 5 } else { "seis" };

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

Problema 44 Retorno de Valor desde loop

El `loop` puede devolver un valor mediante la palabra clave `break`. Completa el código para que el bucle se detenga cuando `contador` sea 5 y devuelva el valor de `contador` multiplicado por 10.

```
fn main() {
    let mut contador = 0;

    let resultado = loop {
        contador += 1;

        if contador == 5 {
            // Escribe el break con el valor de retorno aquí
        }
    };

    assert_eq!(resultado, 50);
}
```

Problema 45 Etiquetas de Bucle (Loop Labels)

Rust permite nombrar bucles para romper el bucle exterior desde uno interior. Modifica el código para que el `break` termine el bucle `'externo` cuando `j` sea igual a 1.

```
fn main() {
    'externo: loop {
        println!("Entrando al bucle externo");
        loop {
            println!("Entrando al bucle interno");
            let j = 1;
            if j == 1 {
                // Termina el bucle 'externo' aquí
            }
        }
    }
    println!("Salida exitosa");
}
```

Problema 46 Contador con while

Usa un bucle `while` para imprimir los números del 5 al 1 (cuenta regresiva). El bucle debe detenerse cuando la variable sea 0.

```
fn main() {
    let mut n = 5;

    // Escribe el while aquí

    println!("¡Ignición!");
}
```

Problema 47 Iteración Segura con for

Es peligroso iterar un array usando un índice manual en un `while`. Reescribe el siguiente código usando un bucle `for` para que sea más seguro y eficiente.

```
fn main() {
    let colores = ["rojo", "verde", "azul"];
    let mut i = 0;

    // Sustituye este while por un for
    while i < 3 {
        println!("Color: {}", colores[i]);
        i += 1;
    }
}
```

Problema 48 Rangos y Reversa

Usa un rango `(..)` y el método `.rev()` para imprimir los números del 1 al 10 en orden descendente (del 10 al 1).

```
fn main() {
    // Escribe el for usando un rango invertido aquí
    for numero in // ...
    {
        println!("{numero}");
    }
}
```

Problema 49 Divisibilidad y else if

Crea un programa que verifique si un número es divisible por 5, por 3, o por ambos. Recuerda que Rust solo ejecuta el **primer** brazo cuya condición sea verdadera.

```
fn main() {
    let n = 15;

    if n % 5 == 0 && n % 3 == 0 {
        println!("Divisible por ambos");
    } else if n % 5 == 0 {
        println!("Divisible por 5");
    } // Añade el caso para el 3 y un else final
}
```

Problema 50 Lógica de continue

El `continue` salta el resto del código en la iteración actual. Completa el código para que el bucle imprima solo los números **impares** entre 1 y 6.

```
fn main() {
    for num in 1..7 {
        if num % 2 == 0 {
            // Usa una palabra clave aquí para saltar los pares
        }
        println!("Impar: {num}");
    }
}
```

Problemas 51 - 60

Problema 51 Stack vs Heap

Basado en la teoría, ¿cuál de estas variables se almacena completamente en el **stack** y cuál involucra datos en el **heap**? Añade un comentario al lado de cada una.

```
fn main() {
    let x = 42;
    let s = String::from("Rust");

    // x está en:
    // s tiene datos en:
}
```

Problema 52 El Error del Move

El siguiente código falla. Explica qué sucede con `s1` cuando se asigna a `s2` y por qué el `println!` no funciona.

```
fn main() {
    let s1 = String::from("hola");
    let s2 = s1;

    println!("{}, mundo!", s1);
}
```

Problema 53 Clonación Explícita

Si realmente necesitamos que `s1` siga siendo válido después de la asignación, debemos hacer una «copia profunda» (deep copy). Corrige el código anterior usando el método adecuado.

```
fn main() {
    let s1 = String::from("hola");
    // Haz una copia profunda de s1 aquí
    let s2 =

    println!("s1: {}, s2: {}", s1, s2);
}
```

Problema 54 El Trait Copy

¿Por qué el siguiente código sí compila sin errores, a diferencia del ejemplo con `String`? Menciona el nombre del **trait** que permite este comportamiento.

```
fn main() {
    let x = 5;
    let y = x;

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

Problema 55 Transferencia de Propiedad a Funciones

Al pasar `s` a la función, la propiedad se mueve. ¿Qué línea causará un error de compilación si intentamos usar `s` después de llamar a la función?

```
fn main() {
    let s = String::from("texto");
    imprimir(s);

    // Intenta imprimir s aquí
    println!("¿Sigue aquí?: {s}");
}

fn imprimir(texto: String) {
    println!("{texto}");
}
```

Problema 56 Retorno de Propiedad

Podemos recuperar la propiedad si la función devuelve el valor. Completa la función para que reciba un `String` y lo devuelva de vuelta al `main`.

```
fn main() {
    let s1 = String::from("regreso");
    let s2 = devolver(s1);

    println!("s2 tiene la propiedad: {s2}");
}

fn devolver(s: String) -> String {
    // Tu código aquí
}
```

Problema 57 Tuplas para Múltiples Retornos

A veces queremos devolver la propiedad y un dato extra (como la longitud). Completa la tupla de retorno.

```
fn calcular_longitud(s: String) -> (String, usize) {
    let largo = s.len();
    // Devuelve s y largo en una tupla
}

fn main() {
    let s1 = String::from("rust");
    let (s2, len) = calcular_longitud(s1);
}
```

Problema 58 Copy en Tuplas Mixtas

Según la teoría, una tupla es `Copy` solo si todos sus elementos lo son. ¿Cuál de estas asignaciones provocará un «move» (y por tanto invalidará la variable original)?

```
fn main() {
    let a = (10, 20);
    let b = a; // ¿Es move o copy?

    let c = (10, String::from("hola"));
    let d = c; // ¿Es move o copy?

    // println!("{:?}", c); // Si descomento esto, ¿falla?
}
```

Problema 59 Drop Prematuro

Rust libera la memoria inmediatamente cuando asignamos un nuevo valor a una variable mutable que ya poseía datos en el heap. Explica qué ocurre con la memoria de «hola» en la línea 3.

```
fn main() {
    let mut s = String::from("hola");
    s = String::from("adiós"); // ¿Qué pasa con "hola"?

    println!("{s}");
}
```

Problema 60 Scope y Drop

Dibuja o explica en qué línea exacta se llama a la función `drop` para la variable `s` en este código.

```
fn main() {
    {
        let s = String::from("dentro");
        println!("{s}");
    } // <-- ¿A?

    println!("Fuera del bloque");
} // <-- ¿B?
```

Problemas 61 - 70

Problema 61 Referencia Básica

Modifica la función `main` para pasar una referencia de `s1` a la función `calcular_longitud`. No debes mover la propiedad, de modo que `s1` pueda ser impreso al final.

```
fn main() {
    let s1 = String::from("referencia");

    // Pasa una referencia aquí
    let len = calculate_length( /* ... */ );

    println!("La longitud de '{}' es {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Problema 62 El Préstamo es Inmutable

Por defecto, no puedes modificar lo que pides prestado. ¿Qué error dará este código y cómo lo explicarías basándote en la analogía de «pedir prestado»?

```
fn main() {
    let s = String::from("hola");
    intentar_cambiar(&s);
}

fn intentar_cambiar(un_string: &String) {
    un_string.push_str(", mundo");
}
```

Problema 63 Referencias Mutables

Para modificar un valor prestado, necesitamos una **referencia mutable**. Corrige las tres partes necesarias: la declaración de `s`, el paso del argumento y la firma de la función.

```
fn main() {
    let s = String::from("hola");

    cambiar(&s);
}

fn cambiar(un_string: &String) {
    un_string.push_str(", mundo");
}
```

Problema 64 La Regla de Oro de la Mutabilidad

Rust prohíbe tener dos referencias mutables al mismo tiempo. ¿Cuál de estas líneas causará que el compilador falle y por qué?

```
fn main() {
    let mut s = String::from("valor");

    let r1 = &mut s;
    let r2 = &mut s; // <-- ¿Qué pasa aquí?

    println!("{}, {}", r1, r2);
}
```

Problema 65 Mezcla Prohibida

No podemos tener una referencia mutable si ya existe una inmutable. Explica por qué el siguiente código es peligroso para los usuarios de `r1` y `r2`.

```
fn main() {
    let mut s = String::from("dato");

    let r1 = &s;
    let r2 = &s;
    let r3 = &mut s; // <-- EL GRAN PROBLEMA

    println!("{}, {}, y {}", r1, r2, r3);
}
```

Problema 66 Scopes de Referencias

Afortunadamente, el scope de una referencia termina en su **último uso**. ¿Por qué el siguiente código **sí** compila?

```
fn main() {
    let mut s = String::from("hola");

    let r1 = &s;
    let r2 = &s;
    println!("{} y {}", r1, r2);

    let r3 = &mut s; // Esto funciona. ¿Por qué?
    println!("{}", r3);
}
```

Problema 67 Punteros Colgantes (Dangling)

El siguiente código intenta devolver una referencia a una variable que deja de existir al final de la función. Explica por qué Rust lanza un error y qué sucede con la memoria de `s` al llegar al cierre de llave `}`.

```
fn main() {
    let referencia = colgar();
}

fn colgar() -> &String {
    let s = String::from("adiós");
    &s
}
```

Problema 68 Solución al Dangling

Corrige la función `colgar` del problema anterior para que sea segura. En lugar de devolver una referencia, devuelve el objeto completo (mueve la propiedad).

```
fn no_cuelgues() -> String {
    let s = String::from("seguro");
    // Tu código aquí
}
```

Problema 69 Carreras de Datos (Data Races)

Menciona los tres comportamientos que deben ocurrir simultáneamente para que Rust considere que hay una **carrera de datos**, la cual previene no permitiendo múltiples referencias mutables.

1. ...
2. ...
3. ...

Problema 70 Scopes con Llaves

Podemos usar bloques de código `{ }` para crear nuevos scopes y así tener varias referencias mutables (pero no simultáneas). Completa el código.

```
fn main() {
    let mut s = String::from("limpio");

    {
        let r1 = &mut s;
        r1.push_str("!");
    } // r1 sale de scope aquí

    // Crea r2 como referencia mutable aquí
    let r2 =

    println!("{}", r2);
}
```

Problemas 71 - 80

Problema 71 Sintaxis de Rangos

Dada la cadena `s`, crea tres slices distintos usando la sintaxis abreviada de rangos (`..`): uno que contenga «Hola», otro que contenga «Rust» y otro que contenga toda la cadena.

```
fn main() {
    let s = String::from("Hola Rust");

    let hola = // ...
    let rust = // ...
    let todo = // ...
}
```

Problema 72 El Tipo &str

¿Cuál es la diferencia técnica entre `String` y `&str`? Añade un comentario explicando qué almacena internamente un slice (pista: son dos datos).

```
fn main() {
    let s_propio = String::from("Contenedor");
    let s_slice: &str = &s_propio[0..4];

    // El slice almacena:
    // 1.
    // 2.
}
```

Problema 73 Slices y Seguridad en Compilación

El siguiente código intenta imprimir un slice después de limpiar el `String` original. Explica qué error lanzará el compilador y por qué el «Borrow Checker» protege aquí la memoria.

```
fn main() {
    let mut s = String::from("mensaje largo");
    let slice = &s[0..7];

    s.clear();

    println!("El slice es: {slice}");
}
```

Problema 74 Literales como Slices

¿Por qué los literales de cadena (como `"hola"`) son inmutables en Rust? Explica su relación con el tipo `&str` y el binario del programa.

```
fn main() {
    // ¿De qué tipo es 'literal'?
    let literal = "Soy inmutable";
}
```

Problema 75 Firmas de Función Flexibles

Mejora la firma de esta función para que acepte tanto `&String` como `&str` de forma eficiente (deref coercion).

```
// Cambia la firma aquí
fn procesar_texto(s: &String) {
    println!("Procesando: {s}");
}

fn main() {
    let s = String::from("Hola");
    procesar_texto(&s);
    procesar_texto("Mundo");
}
```

Problema 76 Slices de Arrays

Los slices no son solo para strings. Crea un slice que tome los elementos `20` y `30` del siguiente array. ¿Cuál es el tipo de dato resultante?

```
fn main() {
    let numeros = [10, 20, 30, 40, 50];

    let slice = // ...

    assert_eq!(slice, &[20, 30]);
}
```

Problema 77 Lógica del primer_word

Completa la lógica de esta función para que devuelva un slice de la primera palabra. Si no hay espacios, debe devolver el string completo.

```
fn primera_palabra(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            // Retorna el slice aquí
        }
    }

    // Retorna el string completo aquí
}
```

Problema 78 Límites de UTF-8

El texto advierte sobre los índices en los slices. ¿Qué sucede si intentas obtener un slice como `&s[0..1]` en una cadena donde el primer carácter es un emoji o un carácter multiocteto?

```
fn main() {
    let s = "😍";
    // let slice = &s[0..1]; // ¿Qué pasaría al ejecutar esto?
}
```

Problema 79 Iteradores y Enumeración

En la expresión `for (i, &item) in bytes.iter().enumerate()`, explica brevemente qué hace cada método:

1. `.iter()`:
2. `.enumerate()`:

Problema 80 — Desafío Final: Ownership y Slices

Analiza este código. ¿Compilará? Si no, ¿cómo lo arreglarías sin usar `.clone()`?

```
fn obtener_prefijo(s: String) -> &str {
    &s[..3]
}

fn main() {
    let texto = String::from("Rustacean");
    let prefijo = obtener_prefijo(texto);
    println!("{prefijo}");
}
```

Problemas 81 - 90

Problema 81 El Convertidor de Temperatura

Escribe una función que convierta Celsius a Fahrenheit. Usa `f64` y asegúrate de que el resultado se asigne usando un `if` como expresión para determinar si el agua estaría congelada o no.

```
fn celsius_a_fahrenheit(c: f64) -> f64 {
    (c * 9.0 / 5.0) + 32.0
}

fn main() {
    let temp_c = 0.0;
    let temp_f = celsius_a_fahrenheit(temp_c);

    let estado = if temp_f <= 32.0 { "Congelado" } else
{ "Líquido" };
    println!("Estado: {estado}");
}
```

Problema 82 Préstamo de Perímetro

Crea una función que calcule el perímetro de un rectángulo. La función debe recibir una **referencia** a una tupla `(u32, u32)` para no tomar propiedad de ella.

```
fn calcular_perimetro(rect: &(u32, u32)) -> u32 {
    // Tu código aquí: 2 * (ancho + alto)
}

fn main() {
    let rectangulo = (10, 20);
    let p = calcular_perimetro(&rectangulo);
    println!("Perímetro: {p}. Tupla original sigue aquí: {:?}",
rectangulo);
}
```

Problema 83 Shadowing de Entrada

Simula el procesamiento de una entrada de usuario. Primero declara `input` como un `&str` con espacios, luego sombréalo (shadowing) para que sea un `usize` con su longitud tras aplicar `.trim().len()`.

```
fn main() {
    let input = "  42  ";
    // Tu código aquí (shadowing)

    println!("La longitud real es: {input}");
}
```

Problema 84 FizzBuzz con Retorno

Crea una función que reciba un número y devuelva un `String`. Si es divisible por 3 devuelve «Fizz», por 5 «Buzz», por ambos «FizzBuzz» y si no, el número como texto usando `n.to_string()`.

```
fn fizz_buzz(n: i32) -> String {
    // Usa if / else if / else como expresión de retorno
}
```

Problema 85 El Array Inmutable

Dado un array de 5 enteros, usa un bucle `for` y un `if` para sumar solo los números que sean mayores a 10.

```
fn main() {
    let numeros = [5, 12, 8, 15, 3];
    let mut suma = 0;
    // Tu bucle aquí
    println!("Suma: {suma}");
}
```

Problema 86 Slices de Usuario

Escribe una función que reciba un `&str` (nombre completo) y devuelva un slice con solo el primer nombre (hasta el primer espacio).

```
fn obtener_primer_nombre(nombre: &str) -> &str {
    // Usa .as_bytes() e iteradores
}
```

Problema 87 Ownership en el Bucle

¿Por qué falla este código? Corrígelo para que `s` pueda imprimirse dentro del bucle sin perder la propiedad en la primera iteración.

```
fn procesar(texto: String) {
    println!("Procesando: {texto}");
}

fn main() {
    let s = String::from("Hola");
    for _ in 0..3 {
        procesar(s); // Error aquí
    }
}
```

Problema 88 La Constante de Tiempo

Define una constante global para los minutos de un día escolar (6 horas). Luego, en `main`, usa esa constante para calcular cuántos segundos son.

```
// Define la constante aquí

fn main() {
    let segundos = MINUTOS_ESCUELA * 60;
}
```

Problema 89 Referencia Mutable de String

Crea una función llamada `limpiar_y_anadir` que reciba una referencia mutable a un `String`. La función debe vaciarlo con `.clear()` y añadirle «Nuevo contenido».

```
fn main() {
    let mut base = String::from("Viejo");
    limpiar_y_anadir(&mut base);
    println!("{base}");
}
```

Problema 90 Validación de Rango de Array

Crea una función que reciba un array de 3 elementos y un índice. Si el índice es válido, devuelve el valor; si no, devuelve `0`. Usa `if` para prevenir un pánico.

```
fn obtener_seguro(arr: [i32; 3], idx: usize) -> i32 {
    // Tu lógica aquí
}
```

Problemas 91 - 100

Problema 91 Sombreado de Scopes Internos

Predice la salida de este código que mezcla scopes y shadowing.

```
fn main() {
    let x = 7;
    {
        let x = x + 3;
        println!("Interno: {x}");
    }
    let x = x * 2;
    println!("Final: {x}");
}
```

Problema 92 Loop con Acumulador

Usa un `loop` (no `while`) para encontrar el primer número divisible por 7 que sea mayor a 100. Devuelve ese valor usando `break`.

```
fn main() {
    let mut n = 100;
    let resultado = loop {
        // Tu lógica aquí
    };
    println!("Encontrado: {resultado}");
}
```

Problema 93 Múltiples Referencias Inmutables

Demuestra que puedes tener dos referencias inmutables funcionando al mismo tiempo, pero no una mutable mientras existan las inmutables.

```
fn main() {
    let mut s = String::from("Dato");
    let r1 = &s;
    let r2 = &s;
    println!("{r1} y {r2}");
    // Crea r3 (mutable) aquí después de que r1 y r2 ya no se
usen
}
```

Problema 94 Slices de Array de Enteros

Escribe una función que reciba un slice de enteros `&[i32]` y devuelva la suma del primer y el último elemento del slice.

```
fn sumar_extremos(slice: &[i32]) -> i32 {
    slice[0] + slice[slice.len() - 1]
}
```

Problema 95 Diferencia entre String y &str

Escribe una función que acepte un parámetro de tipo `&str` y otra que acepte `String`. Explica en un comentario cuál permite pasar tanto literales como objetos `String`.

```
fn f1(s: &str) {}
fn f2(s: String) {}

fn main() {
    let texto = String::from("Hola");
    f1(&texto);
    f2(texto);
    // ¿Cuál es más flexible?
}
```

Problema 96 Tuplas y Shadowing

Crea una tupla con un `i32` y un `bool`. Usa shadowing para extraer solo el valor numérico e incrementarlo.

```
fn main() {
    let datos = (10, true);
    let (valor, _) = datos;
    let valor = valor + 1;
}
```

Problema 97 El Puntero Colgante

Explica por qué este código no compila. ¿Qué sucede con la memoria de `res` al terminar la función?

```
fn crear_referencia() -> &String {
    let res = String::from("Error");
    &res
}
```

Problema 98 Bucle sobre Rango Invertido

Usa un `for` para imprimir una cuenta atrás desde 10 hasta 1, pero salta el número 5 usando `continue`.

```
fn main() {
    for i in (1..11).rev() {
        // Tu lógica aquí
    }
}
```

Problema 99 Anidación de Loops y Labels

Usa un loop con etiqueta para salir de dos bucles anidados cuando una variable externa llegue a 5.

```
fn main() {
    let mut x = 0;
    'exterior: loop {
        x += 1;
        loop {
            if x == 5 { break 'exterior; }
            break;
        }
    }
}
```

Problema 100 El Tipo usize

Explica en un comentario por qué usamos `usize` para los índices de arrays y longitudes de Strings en lugar de `u32` o `i32`.

```
fn main() {
    let s = String::from("Rust");
    let len: usize = s.len();
    // Comentario:
}
```