Arrays

Un array en Rust es una colección de tamaño fijo de valores del mismo tipo.

Los arrays de tamaño fijo siempre se almacenan en el stack.

Esto los hace muy rápidos, pero limita su uso a tamaños pequeños o medianos.

Una vez definido, su tamaño no puede cambiar durante la ejecución del programa.

Sintaxis general

```
let nombre: [Tipo; tamaño] = [valor1, valor2, valor3, ...];
```

Creación

Se definen entre corchetes '`[ ]`'.

```
let notas: [i32; 4] = [15, 18, 20, 17];
            // Índice:  0,  1,  2,  3
```

- `let notas: [i32; 4]` se crea un array con 4 elementos `i32`
- `[15, 18, 20, 17]` se establecen los 4 números separados por '`,`'.

Debug e impresión

Puedes imprimir arrays fácilmente con `{:?}` o `{:#?}` para formato legible.

```
let datos = [10, 20, 30];
println!("{:?}", datos);   // [10, 20, 30]
println!("{:#?}", datos);  // formato en líneas
```

Resultado con `{:#?}`:

```
[
    10,
    20,
    30,
]
```

Tipos de datos

Los arrays pueden ser de cualquier tipo conocido en tiempo de compilación:

```
let cadenas = ["uno", "dos", "tres"];
let booleanos = [true, false, true];
let caracteres = ['a', 'b', 'c'];
let flotantes = [1.5, 2.7, 3.9];
```

Si mezclas tipos distintos, Rust no compila:

```
// Error de compilación:
let mixto = [1, "dos", 3.0];
```

Acceso por índice

```
let notas = [15, 18, 20, 17];

println!("Primera nota: {}", notas[0]);  // 15
println!("Segunda nota: {}", notas[1]);  // 18
println!("Tercera nota: {}", notas[2]);  // 20
println!("Cuarta nota: {}", notas[3]);   // 17
```

Los índices empiezan en 0, igual que en tuplas.

Acceso seguro

Rust evita errores de índice fuera de rango en tiempo de compilación cuando es posible.

```
let arr = [1, 2, 3];

println!("{}", arr[5]); // panic!
```

Acceder fuera del rango genera un panic!.

Para acceso seguro sin panic, usa métodos como `get()`:

```
let arr = [1, 2, 3];

match arr.get(5) {
    Some(valor) => println!("Valor: {}", valor),
    None => println!("Índice fuera de rango"),
}
// Imprime: Índice fuera de rango
```

Declaración de arrays

Inicialización explícita

```
let valores = [5, 10, 15];  // tipo inferido: [i32; 3]
```

Inicialización repetida

```
let ceros = [0; 5];  // [0, 0, 0, 0, 0]
```

Esto crea un array de 5 elementos, todos con valor 0.

Útil para inicializar arrays grandes:

```
let buffer = [0u8; 1024];  // 1024 bytes en cero
```

Mutabilidad

Para modificar un array, debe ser mutable con `mut`.

```
let mut numeros = [1, 2, 3, 4];

println!("{:?}", numeros);  // Antes: [1, 2, 3, 4]

numeros[2] = 99;  // modificamos el valor 3 por 99

println!("{:?}", numeros);  // Después: [1, 2, 99, 4]
```

No puedes cambiar el tamaño, solo los valores:

```
let mut arr = [1, 2, 3];
// arr[3] = 4;  // Error: índice fuera de rango
```

Desestructuración

```
let numeros = [1, 2, 3, 4, 5];

// Desestructuración completa
let [a, b, c, d, e] = numeros;
```

```
println!("a={}, b={}, c={}, d={}, e={}", a, b, c, d, e);
// a=1, b=2, c=3, d=4, e=5
```

El número de variables debe coincidir con el tamaño del array.

Desestructuración parcial

Si no te interesan todos los valores, puedes usar _ para ignorar:

```
let numeros = [10, 20, 30, 40];

let [primero, _, tercero, _] = numeros;

println!("Primero = {}, Tercero = {}", primero, tercero);
// Primero = 10, Tercero = 30
```

Rest pattern

Puedes tomar los primeros elementos y manejar el resto:

```
let numeros = [100, 200, 300, 400, 500];

// Ignorar el resto
let [x, y, ..] = numeros;
println!("x = {}, y = {}", x, y);

// Capturar el resto en una slice
let [a, b, rest @ ..] = numeros;
println!("a = {}, b = {}, resto = {:?}", a, b, rest);
```

Resultado:

```
x = 100, y = 200
a = 100, b = 200, resto = [300, 400, 500]
```

También puedes capturar desde el final:

```
let nums = [1, 2, 3, 4, 5];
let [.., penultimo, ultimo] = nums;

println!("{}, {}", penultimo, ultimo);  // 4, 5
```

Comparaciones

Puedes comparar arrays directamente si sus elementos implementan `PartialEq` o `Ord`.

```
let a = [1, 2, 3];
let b = [1, 2, 3];
let c = [1, 2, 4];

println!("{}", a == b);  // true
println!("{}", a != c);  // true
println!("{}", a < c);   // true (comparación lexicográfica)
```

La comparación es elemento por elemento, de izquierda a derecha.

Copia y movimiento

Los arrays se copian completamente si contienen tipos que implementan el trait `Copy` como enteros, booleanos o caracteres.

```
let a = [1, 2, 3];
let b = a;  // copia completa del array

println!("{:?}", a);  // válido, no se mueve
println!("{:?}", b);  // [1, 2, 3]
```

Esto ocurre porque los arrays de tamaño fijo están en el stack y son baratos de copiar.

Arrays con tipos que no implementan Copy

```
let a = [String::from("Hola"), String::from("Mundo")];
// let b = a;  //Error: String no implementa Copy

// Solución: clonar explícitamente
let b = a.clone();
```

Arrays multidimensionales

```
let matriz: [[i32; 3]; 2] = [
    [1, 2, 3],
    [4, 5, 6],
];

println!("Elemento fila 0, col 1: {}", matriz[0][1]);  // 2
println!("Elemento fila 1, col 2: {}", matriz[1][2]);  // 6
```

Se lee de derecha a izquierda: `[[i32; 3]; 2]` = array de 2 filas, cada fila con 3 elementos `i32`.

Arrays 3D:

```
let cubo: [[[i32; 2]; 2]; 2] = [
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]],
];

println!("{}", cubo[1][0][1]);  // 6
```

Métodos útiles

```
let arr = [5, 10, 15, 20];

println!("Longitud: {}", arr.len());          // 4
println!("Está vacío? {}", arr.is_empty());   // false
println!("Primer elemento: {:?}", arr.first());  // Some(5)
println!("Último elemento: {:?}", arr.last());   // Some(20)
```

Resultado:

```
Longitud: 4
Está vacío? false
Primer elemento: Some(5)
Último elemento: Some(20)
```

Métodos adicionales útiles

```
let arr = [1, 2, 3, 4, 5];
```

```rust
// Verificar si contiene un valor
println!("{}", arr.contains(&3));  // true


// Obtener slice
let slice = &arr[1..4];
println!("{:?}", slice);  // [2, 3, 4]
```

Conversión entre tuplas y arrays

De tupla a array

```rust
let tupla: (u32, u32, u32) = (1, 2, 3);
let array: [u32; 3] = tupla.into();

println!("{:?}", array);  // [1, 2, 3]
```

Esto solo funciona para tuplas con elementos del mismo tipo.

De array a tupla

```rust
let array = [1, 2, 3];
let tupla = (array[0], array[1], array[2]);

println!("{:?}", tupla);  // (1, 2, 3)
```

Límite práctico de tamaño

Aunque técnicamente no hay límite de tamaño para arrays, hay consideraciones prácticas:

- Arrays muy grandes (>10,000 elementos) pueden causar stack overflow
- Para colecciones grandes o dinámicas, usa `Vec<T>` que vive en el heap

```rust
// ✓ Bien para arrays pequeños
let pequeno = [0; 100];

// Riesgoso para arrays grandes en el stack
// let enorme = [0; 1_000_000];  // Puede causar stack overflow

// Mejor usar Vec para colecciones grandes
let grande = vec![0; 1_000_000];
```