

Exploración del tipo de datos vectoriales:

Al igual que con las matrices, los vectores almacenan valores que tienen el mismo tipo de dato. A diferencia de las matrices, el tamaño o la longitud de un vector puede aumentar o reducirse en cualquier momento. La capacidad de cambiar el tamaño con el tiempo está implícita en tiempo de compilación. Por lo tanto, Rust no puede impedir que se acceda a una posición no válida en el vector como lo hace para el acceso en matrices fuera de los límites.

Definición de un vector:


Al leer código en Rust, observará la sintaxis `<T>`. Esta sintaxis representa el uso de un tipo genérico T. Usamos una declaración de tipo genérico cuando todavía no se conoce el tipo de dato real.

La sintaxis de tipo genérico se usa para declarar vectores. La sintaxis `<vector><T>` declara un tipo de vector compuesto de un tipo de datos T genérico (aún no conocido).

Para crear realmente un vector, usamos un tipo concreto como por ejemplo, `<vector>u32`, un vector de tipo u32 o `<vector>String`, para un vector de tipo cadena.

Una manera común de declarar e inicializar un vector es con la macro `vec!`. Esta macro también acepta la misma sintaxis que el constructor de la matriz.

Rust

 Copiar

```
// Declare vector, initialize with three values
let three_nums = vec![15, 3, 46];
println!("Initial vector: {:?}", three_nums);

// Declare vector, value = "0", length = 5
let zeroes = vec![0; 5];
println!("Zeroes: {:?}", zeroes);
```

La salida es la siguiente:

```
Resultados Copiar
Initial vector: [15, 3, 46]
Zeroes: [0, 0, 0, 0, 0]
```

En este ejemplo, se usa la sintaxis de dos puntos y signo de interrogación `{:?}` con la macro `println!`. Rust no sabe cómo mostrar un vector de enteros. Si intentamos mostrar la información del vector sin usar un formato especial, el compilador emite un error. Usamos las llaves vacías `{}` para ayudar a mostrar los valores del vector.

Los vectores también se pueden crear mediante el método `Vec::new()`. Este método de creación de vectores nos permite agregar y quitar valores al final del vector. Para admitir este comportamiento, declaramos la variable de vector como mutable con la palabra clave `mut`.

```
Rust Copiar
// Create empty vector, declare vector mutable so it can grow and shrink
let mut fruit = Vec::new();
```

Valores push y pop

Cuando creamos un vector con el método `Vec::new()`, podemos agregar y quitar valores al final del vector.

Para agregar un valor al final del vector, usamos el método `push(<value>)`.

```
Rust Copiar
// Push values onto end of vector, type changes from generic `T` to String
fruit.push("Apple");
fruit.push("Banana");
fruit.push("Cherry");
println!("Fruits: {:?}", fruit);
```

En la salida, observe que el sistema muestra los corchetes del vector y que cada valor String está entre comillas:

```
Resultados Copiar  
Fruits: ["Apple", "Banana", "Cherry"]
```

Una vez que el tipo de un vector se establece en un tipo concreto, solo se pueden agregar valores de ese tipo específico al vector. Si intentamos agregar un valor de un tipo diferente, el compilador devuelve un error.


```
Resultados Copiar  
error[E0308]: mismatched types  
--> src/main.rs:11:17  
|  
11 |     fruit.push(1);  
|                ^ expected `&str`, found integer  
  
error: aborting due to previous error
```

Para quitar el valor al final del vector, usamos el método `pop()`.

```
Rust Copiar  
  
// Pop off value at end of vector  
// Call pop() method from inside println! macro  
println!("Pop off: {:?}", fruit.pop());  
println!("Fruits: {:?}", fruit);
```

En la salida se muestra que el valor "Cherry" se ha quitado y no está asociado a un vector:

Resultados


 Copiar

```
Pop off: Some("Cherry")  
Fruits: ["Apple", "Banana"]
```

Indexación en un vector:

Los vectores admiten la indexación de la misma manera que las matrices. Podemos acceder a los valores de elemento del vector mediante un índice. El primer elemento está en el índice 0 y el último en la longitud de vector: 1.


Rust

 Copiar

```
// Declare vector, initialize with three values  
let mut index_vec = vec![15, 3, 46];  
let three = index_vec[1];  
println!("Vector: {:?}, three = {}", index_vec, three);
```

La salida es la siguiente:


Resultados

 Copiar

```
Vector: [15, 3, 46], three = 3
```

Dado que los valores del vector son **mutables**, podemos cambiar un valor en su lugar accediendo al valor del elemento con el índice:


Rust

 Copiar

```
// Add 5 to the value at index 1, which is 5 + 3 = 8  
index_vec[1] = index_vec[1] + 5;  
println!("Vector: {:?}", index_vec);
```

La salida es la siguiente:

Resultados

 Copiar


Vector: [15, 8, 46]

Búsqueda de valores de índice fuera de los límites:

Al igual que con las matrices, no se puede acceder a un elemento de un vector con un índice que no esté en el intervalo permitido. Este tipo de expresión para una matriz hace que el compilador devuelva un error. En el caso de los vectores, **la compilación pasa, pero el programa entra en un estado de pánico irre recuperable en la expresión y detiene la ejecución del programa.**

Para nuestro vector de ejemplo que tiene tres elementos, ¿qué ocurre si intentamos acceder al elemento en el índice 10?


Rust

 Copiar

```
// Access vector with out-of-bounds index
let beyond = index_vec[10];
println!("{}", beyond);
```

El programa se anula con el mensaje de error siguiente:

Resultados

 Copiar

```
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 10'...
```

En otro módulo, analizamos cómo acceder de forma segura a un elemento del vector sin provocar una alerta de pánico (panic) en el programa.