

Definición de colecciones de datos mediante tuplas y estructuras:

En esta unidad, se explorarán dos tipos de datos que son útiles para trabajar con colecciones de datos o datos compuestos: las tuplas y las estructuras.

Tuplas (Tuples):

Una tupla es una **agrupación de valores de distintos tipos** recopilados en un valor compuesto. Los valores individuales de una tupla se denominan **elementos**. Los valores se especifican como una lista separada por comas entre paréntesis (<value>, <value>, ...).

Una tupla tiene una longitud fija, que es igual a su **número de elementos**. Una vez declarada una tupla, no puede aumentar ni reducir su tamaño. No se pueden agregar ni quitar elementos. El tipo de dato de una tupla se define mediante la secuencia de los tipos de datos de los elementos.

Definición de una tupla

Este es un ejemplo de una tupla con tres elementos:

```
Rust Copiar  
  
// Tuple of length 3  
let tuple_e = ('E', 5i32, true);
```

	Elemento	Value	Tipo de datos
0	E		char
1	5		i32
2	true		bool


Acceso a elementos de una tupla:

Se puede acceder a los elementos de una tupla por la posición del índice, a partir de cero. Este proceso se conoce como *indexación de*

tupla. Para acceder a un elemento de una tupla, usamos la sintaxis `<tupla>.<index>` → (dot notation).

En el ejemplo siguiente se muestra cómo acceder a los elementos de la tupla usando la indexación:


Rust

 Copiar

```
// Declare a tuple of three elements
let tuple_e = ('E', 5i32, true);

// Use tuple indexing and show the values of the elements in the tuple
println!("Is '{}' the {}th letter of the alphabet? {}", tuple_e.0, tuple_e.1, tuple_e.2);
```

Resultados

 Copiar

```
Is 'E' the 5th letter of the alphabet? true
```

Las tuplas resultan útiles cuando se quieren combinar tipos distintos en un único valor. Las funciones pueden utilizar tuplas para devolver varios valores porque las tuplas pueden contener cualquier número de valores.

Estructuras(Structs):

Una estructura es un tipo compuesto por otros tipos. Los elementos de una estructura se denominan *campos*. Al igual que las tuplas, los campos de una estructura pueden tener tipos de datos diferentes. Una ventaja importante de las estructuras es que pueden asignar un nombre a cada campo, por lo que queda claro lo que significa el valor.

Para trabajar con estructuras en Rust, en primer lugar se debe definir la estructura por nombre y especificar el tipo de dato de cada campo. Después se debe crear una *instancia* de la estructura con otro nombre. Al declarar la instancia, se proporcionan los valores específicos para los campos.

Rust admite tres tipos de estructura: *clásicas*, *de tupla* y *de unidad*. Estos tipos de estructura admiten diferentes maneras de agrupar y trabajar con los datos.

- Las [estructuras de C](#) clásicas: Son las más utilizadas. Cada campo de la estructura tiene un nombre y un tipo de dato. Una vez definida una estructura clásica, se puede acceder a los campos de

la estructura usando la sintaxis `<struct>.<field>.>` (Dot notation).

- Las estructuras de tupla: Son parecidas a las clásicas, pero sus campos no tienen nombres. A fin de acceder a los campos de una estructura de tupla, usamos la misma sintaxis que para indexar una tupla: `<tuple>.<index>.>` (Dot notation). Al igual que con las tuplas, los valores de índice de la estructura de tupla empiezan por cero.
- Las estructuras de unidad: Suelen usarse como marcadores. Obtendremos más información sobre por qué las estructuras pueden resultar útiles cuando descubramos la característica *traits* de Rust.

En el código siguiente se muestran definiciones de ejemplo para las tres variedades de tipos de estructura:

```
Rust Copiar

// Classic struct with named fields
struct Student { name: String, level: u8, remote: bool }

// Tuple struct with data types only
struct Grades(char, char, char, char, f32);

// Unit struct
struct Unit;
```

Definición de una estructura:


Para definir una estructura, se escribe la palabra clave `struct` seguida del nombre de la estructura. Elija un nombre para el tipo de estructura que describa la característica significativa de los datos agrupados. A diferencia de la convención de nomenclatura que hemos usado hasta ahora, el nombre de un tipo de estructura se escribe en mayúsculas.

En Rust las estructuras se definen a menudo fuera de la función `main` y de otras funciones. Por este motivo, al inicio de la definición de la estructura no se le aplica sangría desde el margen izquierdo. Solo se le aplica sangría a la parte interna de la definición para mostrar cómo se organizan los datos.

Estructura clásica:

Al igual que una función, el cuerpo de una estructura clásica se define entre llaves `{}`. A cada campo de la estructura clásica se le asigna un nombre único dentro de la estructura. El tipo de cada campo se especifica con la sintaxis `: <type>`. Los campos de la estructura clásica se especifican como una lista separada por comas `<field>`, `<field>`, **Una definición de estructura clásica no termina con punto y coma.**

Rust

 Copiar


```
// Classic struct with named fields
struct Student { name: String, level: u8, remote: bool }
```

Estructura de tupla:

Al igual que una tupla, el cuerpo de una estructura de tupla se define entre paréntesis `()`. **Los paréntesis van inmediatamente después del nombre de la estructura.** No hay espacio entre el nombre de la estructura y el paréntesis de apertura.

A diferencia de una tupla, la definición de estructura de tupla incluye solo el tipo de dato de cada campo. Los tipos de datos de la estructura de tupla se especifican como una lista separada por comas `<type>`, `<type>`,

Rust

 Copiar


```
// Tuple struct with data types only
struct Grades(char, char, char, char, f32);
```

Creación de una instancia de una estructura

Después de definir un tipo de estructura, para usar la estructura se crea una instancia del tipo y se especifican valores para cada campo. Al establecer los valores de campo, no es necesario especificar los campos con el mismo orden con el que están definidos.

En el ejemplo siguiente se usan las definiciones que hemos creado para los tipos de estructura `Student` y `Grades`.

Rust

 Copiar

```
// Instantiate classic struct, specify fields in random order, or in specified order
let user_1 = Student { name: String::from("Constance Sharma"), remote: true, level: 2 };
let user_2 = Student { name: String::from("Dyson Tan"), level: 5, remote: false };

// Instantiate tuple structs, pass values in same order as types defined
let mark_1 = Grades('A', 'A', 'B', 'A', 3.75);
let mark_2 = Grades('B', 'A', 'A', 'C', 3.25);

println!("{}", level {}. Remote: {}. Grades: {}, {}, {}, {}. Average: {},
    user_1.name, user_1.level, user_1.remote, mark_1.0, mark_1.1, mark_1.2, mark_1.3, mark_1.4);
println!("{}", level {}. Remote: {}. Grades: {}, {}, {}, {}. Average: {},
    user_2.name, user_2.level, user_2.remote, mark_2.0, mark_2.1, mark_2.2, mark_2.3, mark_2.4);
```

Conversión de un String literal en un String:

Los datos de cadena que se almacenan dentro de otra estructura de datos, como una estructura o un vector, se deben convertir de una referencia literal de cadena (&str) a un tipo String. Para realizar la conversión, se usa el método `String::from(&str)` estándar. Observe cómo se usa este método en este ejemplo:


Rust

 Copiar

```
// Classic struct with named fields
struct Student { name: String, level: u8, remote: bool }
...
let user_2 = Student { name: String::from("Dyson Tan"), level: 5, remote: false };
```

Si no se convierte el tipo antes de asignar el valor, el compilador emite un error:

Resultados

 Copiar

```
error[E0308]: mismatched types
  --> src/main.rs:24:15
   |
24 |         name: "Dyson Tan",
   |         ^^^^^^^^^^^^^
   |         |
   |         expected struct `String`, found `&str`
   |         help: try using a conversion method: `"Dyson Tan".to_string()`

error: aborting due to previous error
```

El compilador sugiere que se puede usar la función `.to_string()` para realizar la conversión. En los ejemplos, se usa el método `String::from(&str)`.