


Exploración de tipos de datos para números, texto y valores true/false.

Rust es un lenguaje de tipado estatico. El compilador debe conocer el tipo de dato exacto de todas las variables del código para que el programa se compile y ejecute. Normalmente, el compilador puede inferir el tipo de datos de una variable en función del valor enlazado. No siempre es necesario indicar de forma explícita el tipo en el código. Cuando son posibles muchos tipos, debe informar al compilador del tipo específico mediante *anotaciones de tipo*.

En el ejemplo siguiente, se le indica al compilador que cree la variable `number` como un entero de 32bits. Especificamos el tipo de datos (`u32`) después del nombre de la variable. Observe que después del nombre de la variable se usa el carácter de dos puntos:

Rust


 Copiar

```
let number: u32 = 14;  
println!("The number is {}.", number);
```

Si ponemos el valor de la variable entre comillas dobles, el compilador interpreta el valor como texto en lugar de como un número.

El tipo de datos deducido del valor no coincide con el tipo de datos `u32` especificado para la variable, por lo que el compilador emite un error:

Rust

 Copiar

```
let number: u32 = "14";
```

Error:

```
Resultados Copiar

Compiling playground v0.0.1 (/playground)
error[E0308]: mismatched types
--> src/main.rs:2:23
|
2 |     let number: u32 = "14";
  |                   ---   ^^^^ expected `u32`, found `&str`
  |                   |
  |                   expected due to this
error: aborting due to previous error
```

Built-in data types:

Rust incluye algunos tipos de datos primitivos integrados para expresar números, texto y veracidad. Algunos de estos tipos se conocen como *escalares*, porque representan un solo valor:

- Números enteros
- Números de punto flotante
- Valores booleanos
- Characters

Rust también ofrece tipos de datos más complejos para trabajar con series de datos, como strings y tuplas.

Números: valores enteros y de punto flotante:

Los enteros en Rust se identifican por el **tamaño en bits** y la **propiedad signed**. Un entero con signo puede ser un número positivo o negativo. Un entero sin signo solo puede ser un número positivo.

	Length	Signed	Unsigned
8	bits	i8	u8
16	bits	i16	u16
32	bits	i32	u32
64	bits	i64	u64
128	bits	i128	u128
	<i>dependiente de la arquitectura</i>	isize	usize

Los tipos `isize` y `usize` dependen del tipo de equipo en el que se ejecuta el programa. El tipo de 64 bits se usa en una arquitectura de 64 bits y el tipo de 32 bits, en una arquitectura de 32 bits. Si no especifica el tipo para un entero, y el sistema no puede deducir el tipo, asigna el tipo `i32` (un entero de 32 bits con signo) de forma predeterminada.

Rust tiene dos tipos de datos de punto flotante para los valores decimales: `f32` (32 bits) y `f64` (64 bits). El tipo de punto flotante predeterminado es `f64`. En las CPU modernas, el tipo `f64` tiene aproximadamente la misma velocidad que el tipo `f32`, pero cuenta con una mayor precisión.


Rust

 Copiar

```
let number_64 = 4.0;           // compiler infers the value to use the default type f64
let number_32: f32 = 5.0; // type f32 specified via annotation
```

Todos los tipos de números primitivos en Rust admiten operaciones matemáticas como la suma, resta, multiplicación y división.

Rust

 Copiar

```
// Addition, Subtraction, and Multiplication
println!("1 + 2 = {} and 8 - 5 = {} and 15 * 3 = {}", 1u32 + 2, 8i32 - 5, 15 * 3);

// Integer and Floating point division
println!("9 / 2 = {} but 9.0 / 2.0 = {}", 9u32 / 2, 9.0 / 2.0);
```

📌 Nota

Cuando llamamos a la macro `println`, agregamos el sufijo de tipo de datos a cada número literal para informar a Rust sobre el tipo de datos. La sintaxis `1u32` indica al compilador que el valor es el número 1 y que interprete el valor como un entero de 32 bits sin signo.

Si no se proporcionan anotaciones de tipo, Rust intenta deducir el tipo a partir del contexto. Cuando el contexto es ambiguo, asigna el tipo `i32` (un entero de 32 bits con signo) de forma predeterminada.


Valores booleanos: true o false:

El tipo booleano en Rust se usa para almacenar la veracidad. El tipo `bool` tiene dos valores posibles: `true` o `false`. Los valores booleanos se usan de forma generalizada en expresiones condicionales.

Si una instrucción `bool` o un valor es `true`, realice esta acción; de lo contrario (la instrucción o el valor es `false`), realice una acción distinta. Una comprobación de comparación suele devolver un valor booleano.

En el ejemplo siguiente, usamos el operador mayor que `>` para probar dos valores. El operador devuelve un valor booleano que muestra el resultado de la prueba.

Rust

 Copiar

```
// Declare variable to store result of "greater than" test, Is 1 > 4? -- false
let is_bigger = 1 > 4;
println!("Is 1 > 4? {}", is_bigger);
```

Texto: characters y strings:

Rust admite valores de texto con dos tipos de cadena básicos y un tipo de carácter. Un carácter es un elemento único, mientras que una cadena es una serie de caracteres. Todos los tipos de texto son representaciones UTF-8 válidas.

El tipo `char` es el más primitivo de los tipos de texto. El valor se especifica poniendo el elemento entre comillas simples:

```
let uppercase_s = 'S';  
let lowercase_f = 'f';  
let smiley_face = '😄';
```

📌 Nota

Algunos lenguajes tratan sus tipos `char` como enteros de 8 bits sin signo, que es el equivalente del tipo `u8` de Rust. El tipo `char` de Rust contiene puntos de código Unicode, pero no usan la codificación UTF-8. `char` en Rust es un entero de 21 bits que se ha agregado para ampliar a 32 bits. `char` contiene directamente el valor de punto de código sin formato.

Strings:

El tipo `str`, también conocido como *string slice*, es una vista de los datos de la cadena. La mayoría de las veces, se hace referencia a estos tipos usando la sintaxis del estilo de referencia que precede al tipo con el símbolo **ampersand** (`&str`). Trataremos las referencias en los siguientes módulos. Por ahora, puede imaginarse `&str` como un puntero a datos de cadena inmutables. Los literales de cadena son todos de tipo `&str`.

Aunque los literales de cadena son convenientes para usarlos en ejemplos de introducción de Rust, no son adecuados para todas las situaciones en las que podríamos querer usar texto. No todas las cadenas pueden conocerse en tiempo de compilación. Un ejemplo se da cuando un usuario interactúa con un programa en tiempo de ejecución y envía texto mediante un terminal.

En estos escenarios, Rust tiene un segundo tipo de cadena denominado **String**. Este tipo se asigna en el (heap), ya que cuando se usa el tipo **String**, no es necesario conocer la longitud de la cadena (número de caracteres) antes de compilar el código.

① Nota

Si está familiarizado con un lenguaje de recolección de elementos no utilizados, es posible que se pregunte por qué Rust tiene dos tipos de cadena.¹ Las cadenas son tipos de datos extremadamente complejos. La mayoría de los lenguajes usan sus recolectores de elementos no utilizados para atenuar esta complejidad. Rust, como lenguaje de un sistema, expone parte de la complejidad inherente de las cadenas. La complejidad adicional conlleva una cantidad de control muy específica sobre cómo se usa la memoria en el programa.

¹ _En realidad, Rust tiene más de dos tipos de cadena. En este módulo, solo se describen los tipos `String` y `&str`. Puede obtener más información sobre los tipos de cadena que se ofrecen en la [documentación de Rust](#).

*"Recolección de elementos no utilizados" → Garbage Collector.

No tendremos una idea completa de la diferencia entre `String` y `&str` hasta que comprendamos los sistemas **ownership y borrowing** de Rust. Hasta entonces, podemos pensar en los datos de tipo `String` como datos de texto que pueden cambiar a medida que se ejecuta el programa y en las referencias `&str` como vistas inmutables en los datos de texto que no cambian a medida que se ejecuta este.

Ejemplo de texto

En el ejemplo siguiente se muestra cómo usar los tipos de datos **`char`** y **`&str`** en Rust.

- Se declaran dos variables de caracteres con la sintaxis de anotación **`: char`**. Los valores se especifican usando **comillas simples**.
- Se declara una tercera variable de caracteres y se enlaza a una sola imagen. Para esta variable, se permite que el compilador deduzca el tipo de dato.
- Se declaran dos strings y se les asignan sus respectivos valores. Los valores de los strings se asignan con comillas dobles.
- Una de los strings se declara con la sintaxis de anotación **`(&str)`** para especificar el tipo de dato. El tipo de datos de la otra variable se deja sin especificar. El compilador deducirá el tipo de datos de esta variable en función del contexto.

Observe que la variable `string_1` incluye un espacio vacío al final de la serie de caracteres.

 Copiar

Esta es la salida de nuestro ejemplo:

 Copiar

Si no utilizasemos la notacion &str el compilador devolveria el siguiente error:

[illegible]