Generics

This lesson introduces generics.

We'll cover the following

- What Are Generics?
- Syntax
- Example 1: Generic Function
- Example 2: Generic Vector
- Example 3: Generic Struct
- Quiz

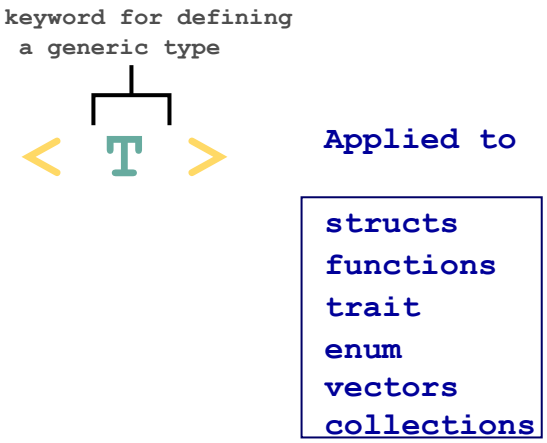
What Are Generics?

Generics are a way of generalizing types; they define the data type at run time. Generics are called **parametric polymorphism** in type theory. 'Poly' is multiple, 'morph' is form over a given parameter ('parametric') meaning multiple forms of a given parameter.

They can be applied to methods, functions, structures, enumerations, collections, and traits. This helps to reuse the same code but with a different type.

Syntax

The `<T>` is known as the **type parameter** and is used to declare a generic construct. `T` can be any data-type.



Example 1: Generic Function

The following example defines a generic function that displays the value passed to it as a parameter. The first value type is a `string` and the second value is an integer of type `i32`.

> **Note:** For printing a value of the passed parameter, write `use std::fmt::Display` prior to function definition and after the generic function's name write `<T:Display>` following the function name prior to writing the passing parameters.

> **Note:** `Display` **is a trait**

```
fn main(){
   println!("- Passing a string literal");
   concatenate(" Rust ", " Programming ");
   println!("- Passing an integer");
   concatenate(10 as i32, 1 as i32);

}
use std::fmt::Display;
fn concatenate<T:Display>(t:T, s:T){
   let result = format!("{}{}", t , s);
   println!("{}", result);
}
```

- On **line 3**, function `concatenate` takes a value of type `String`.
- On **line 5**, function `concatenate` takes a value of type `i32`.
- `T` is declared as a generic type specifier on **line 9**. This line tells the compiler to replace `T` with the type of value with which the function is invoked.

Example 2: Generic Vector

The following example creates a vector `my_int_vector` of type `i32`:

```
fn main(){
   let mut my_int_vector: Vec<i32> = vec![1,2];
   my_int_vector.push(3);
   println!("{:?}",my_int_vector);
   // my_int_vector.push("Rust"); // mismatched types error
}
```

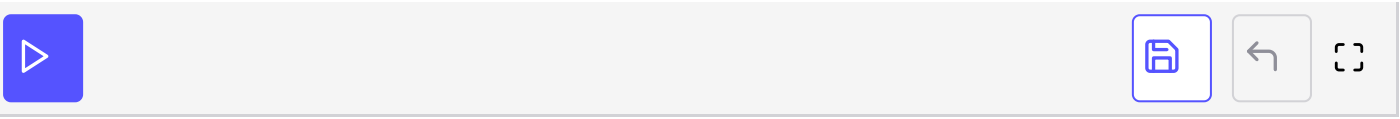
`main` function

- On **line 2**, vectors of type integer type are initialized.
- On **line 3**, an integer number is pushed in the vector and printed on **line 4**.
- If you uncomment the **line 5**, an error, ✕, will be thrown by the compiler because if you try to push a string value into the collection, the compiler will return an error.

> **Note** while making a vector in the same function, we cannot actually do this. However, a vector of type `T` can be passed to the function.

```
use std::fmt::Display;
fn print_vec<T:Display>(v: &[T]) {
  for i in v.iter() {
      print!("{}", i)
  }
  println!("");
}

fn main() {
  let int_vec = [1, 2, 3, 4, 5]; // define a vector of type integer

  println!("Call to the function with vector of integers");

  print_vec(& int_vec); // pass vector of type integer to the function
```

```
  println!("Call to the function with vector of strings");

  let str_vec = ["Rust", "Programming"]; // define a vector of type string

  print_vec(&str_vec); // pass vector of type String to the function
}
```

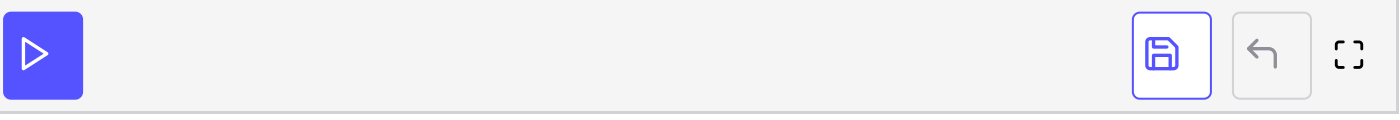


- `main` **function**
  - On **line 10**, vectors of type integer type `int_vec`, is initialized.
  - On **line 14**, `int_vec` is passed to the function `print_vec`.
  - On **line 18**, vectors of String type `str_vec` is initialized.
  - On **line 20**, `str_vec` is passed to the function `print_vec`.
- `print_vec` **function**
  - On **line 2**, `print_vec` is defined with `v` as a parameter to the function of type `&T`.
  - From **line 3 to 5**, `v` is traversed using `v.iter()` and the value is printed on **line 4**.

Example 3: Generic Struct

The following example creates a `struct` `Rectangle`. The struct gets invoked with type instances of type `i32` and `f32` respectively:

```
struct Rectangle<T> {
   width:T,
   height:T
}
fn main() {
   //generic type of i32
   let r1:Rectangle<i32> = Rectangle{width:250, height:150};
   println!("Width:{}, Height:{}", r1.width, r1.height);
   //generic type of String
   let r2:Rectangle<f32> = Rectangle{width:240.0, height:250.0};
   println!("Width:{}, Height:{}", r2.width, r2.height);

}
```



- `struct` **Rectangle**

- `T` is declared as a generic type specifier for `struct Rectangle` on **line 1**.

- `main` **function**

  - On **lines 2 and 3**, the variables `width` and `height` are declared to be of type `T`. The type itself is not determined until a variable of type Rectangle is defined.
  - On **line 7**, we define a variable `r1` of type `Rectangle<i32>`. This line tells the compiler to replace `T` with `i32` in the `Rectangle struct`.
  - On **line 10**, we define a variable `r1` of type `Rectangle<f32>`. This line tells the compiler to replace `T` with `f32` in the `Rectangle struct`.

Quiz

Test your understanding of generics in Rust.

Quick Quiz on Generics!

Q Which of the following types are not allowed to be made generic in Rust!

Retake Quiz

Now that you have learned about generics, let’s test your knowledge in the upcoming challenge.