Rust: Pass-By-Value or Pass-By-Reference? 👐

Ryan Levick · 2018.9.12

The other day, a friend of mine who is learning Rust asked if Rust is a pass-by-value or a pass-by-reference language. For the unfamiliar, pass-by-value means that when passing an argument to a function it gets copied into the new function so that the value in the calling function and the value in the called function are two separate values. Changes to one will not in turn result in the same change to the other. Pass-by-reference, on the other hand, means that when passing an argument to a function, it doesn't copy the value but rather the argument is merely a transparent reference to the original value. This means changes to the value in the called function change the value in the calling function since they are the same value.

A lot of languages have a little bit of both. Take Swift for example. Structs in Swift have call-by-value semantics. If you pass a struct to a function, the struct is copied and a new value is created. If you mutate a struct that was passed to a function the value in the calling function will not be changed. Classes in Swift, on the other hand, are pass-by-reference. Passing a class to a function in Swift does not copy that class. This means if a function mutates a class the class will not only be changed in the called function but also in the calling function.

Rust

So now that we have a good understanding of pass-by-value vs. pass-by-reference, what is Rust? Rust is strictly a pass-by-value language. This might surprise you since references play a big part in the language. However, references in Rust are first class citizens. They, themselves, are values that can be passed - by value - to a function.

Let's take a look at some examples:

```
fn main() {
  let n_main: usize = 100;
  println!("{}", inc(n_main));
}

fn inc(n_inc: usize) -> usize {
  n_inc + 1
}
```

In the above example, `n_main` is a number allocated on the stack of the `main` function. When we pass `n_main` to the `inc` function it is copied to the stack of the `inc` function. `n_main` and `n_inc` are two separate values at two places in memory.

```
fn main() {
  let n_main: usize = 100;
  let n_main_ref: &usize = &n_main;
  println!("{}", inc_ref(n_main_ref));
}

fn inc_ref(n_inc_ref: &usize) -> usize {
  n_inc_ref + 1
}
```

In this example, the function `inc_ref` works the same way as `inc` did before just now with the value being a reference instead of a number of type `usize`. When we pass `n_main_ref` to `inc_ref`, the reference value `n_main_ref` (not the value that it's pointing to) is copied to the stack of `inc_ref`. The reference is a value and it behaves with pass-by-value semantics just like the value of type `usize` did.

Let's take a look at another example to drive the case home:

```
fn main() {
  let array_main: [Vec<u8>; 3] = [vec![1], vec![2, 4], vec![]];
  print(array_main);
}

fn print(array: [Vec<u8>; 3]) {
  for e in array.iter() {
    println!("{:?}", e)
  }
}
```

This example is the exact same as the above examples except the underlying value we're looking at is a fixed length array with elements that are `Vec`s. In the case of the `print` function, the array will be copied from the stack of the main function to the stack of the `print` function. This means that each of the three elements (the `Vec`s) will be copied to the new stack frame.

It's important to be clear what "copying" means here. While logically we might think of a `Vec` as a growable array on the heap, in reality is is just a struct that contains three values: a length, a capacity and a pointer to the actual data on the heap. When we say the `Vec` is copied - we mean *this struct* is copied. The data on the heap, however, is *not copied*.

Now if Rust didn't have the borrow checker this could present a problem. Why? Copying one of the `Vec`s would lead to two pointers (a.k.a. aliased pointers) to the same data on the heap. If we then "dropped" (a.k.a freed, a.k.a. deallocated, a.k.a. destroyed) one of the `Vec`s (and in turn free the heap allocated memory), the pointer in other `Vec` would now point to junk data. Luckily, the borrow checker makes it impossible to use `array_main` after it's been "moved" to the print function.

Ok, now let’s look again at passing a reference by value:

```
fn main() {
  let array_main: [Vec<u8>; 3] = [vec![1], vec![2, 4], vec![]];
  let array_main_ref: &[Vec<u8>] = &array_main;
  print(array_main_ref);
}

fn print(array_ref: &[Vec<u8>]) {
  for e in array.iter() {
    println!("{:?}", e)
  }
}
```

This example again is pass-by-value semantics. The value in question is the reference to our stack allocated array. The reference `array_main_ref` will be copied from `main`’s stack to the stack of the `print` function, but of course, the array itself will not be copied and will stay on the stack of the `main` function.

What’s Actually Going On

Everything I’ve said above is true absent of an optimizing compiler. The Rust compiler is free to do a whole bunch of optimizations that make the above not necessarily true in practice. However, any assumptions we can make with the pass-by-value semantics we described above must remain true - and the Rust compiler guarantees that it won’t pull the rug out from underneath us.

If you’d like to learn more about how Rust works, let me know on Twitter!