Rust-101, Part 04: Ownership, Borrowing, References

Rust aims to be a "safe systems language". As a systems language, of course it provides *references* (or *pointers*). But as a safe language, it has to prevent bugs like this C++ snippet.

```
/*
  void foo(std::vector<int> v) {
      int *first = &v[0];
      v.push_back(42);
      *first = 1337; // This is bad!
  }
*/
```

What's going wrong here? `first` is a pointer into the vector `v`. The operation `push_back` may re-allocate the storage for the vector, in case the old buffer was full. If that happens, `first` is now a dangling pointer, and accessing it can crash the program (or worse).

It turns out that only the combination of two circumstances can lead to such a bug: *aliasing* and *mutation*. In the code above, we have `first` and the buffer of `v` being aliases, and when `push_back` is called, the latter is used to perform a mutation. Therefore, the central principle of the Rust typesystem is to *rule out mutation in the presence of aliasing*. The core tool to achieve that is the notion of *ownership*.

Ownership

What does that mean in practice? Consider the following example.

```
fn work_on_vector(v: Vec<i32>) { /* do something */ }
fn ownership_demo() {
    let v = vec![1,2,3,4];
    work_on_vector(v);
    /* println!("The first element is: {}", v[0]); */               /* BAD! */
}
```

Rust attaches additional meaning to the argument of `work_on_vector`: The function can assume that it entirely *owns* `v`, and hence can do anything with it. When `work_on_vector` ends, nobody needs `v` anymore, so it will be deleted (including its buffer on the heap). Passing a `Vec<i32>` to `work_on_vector` is considered *transfer of ownership*: Someone used to own that vector, but now he gave it on to `take` and has no business with it anymore.

If you give a book to your friend, you cannot just come to his place next day and get the book! It's no longer yours. Rust makes sure you don't break this rule. Try enabling the commented line in `ownership_demo`. Rust will tell you that `v` has been *moved*, which is to say that ownership has been transferred somewhere else. In this particular case, the buffer storing the data does not even exist anymore, so we are lucky that Rust caught this problem! Essentially, ownership rules out aliasing, hence making the kind of problem discussed above impossible.

Borrowing a shared reference

If you go back to our example with `vec_min`, and try to call that function twice, you will get the same error. That's because `vec_min` demands that the caller transfers ownership of the vector. Hence, when `vec_min` finishes, the entire vector is deleted. That's of course not what we wanted! Can't we somehow give `vec_min` access to the vector, while retaining ownership of it?

Rust calls this *a reference* to the vector, and it considers references as *borrowing* ownership. This works a bit like borrowing does in the real world: If your friend borrows a book from you, your friend can have it and work on it (and you can't!) as long as the book is still borrowed. Your friend could even lend the book to someone else. Eventually however, your friend has to give the book back to you, at which point you again have full control.

Rust distinguishes between two kinds of references. First of all, there's the *shared* reference. This is where the book metaphor kind of breaks down... you can give a shared reference to *the same data* to lots of different people, who can all access the data. This of course introduces aliasing, so in order to live up to its promise of safety, Rust generally does not allow mutation through a shared reference.

So, let's re-write `vec_min` to work on a shared reference to a vector, written `&Vec<i32>`. I also took the liberty to convert the function from `SomethingOrNothing` to the standard library type `Option`.

This time, we explicitly request an iterator for the vector `v`. The method `iter` just borrows the vector it works on, and provides shared references to the elements.

In the loop, `e` now has type `&i32`, so we have to dereference it to obtain an `i32`.

```
fn vec_min(v: &Vec<i32>) -> Option<i32> {
    use std::cmp;

    let mut min = None;

    for e in v.iter() {


        min = Some(match min {
            None => *e,
            Some(n) => cmp::min(n, *e)
        });
    }
    min
}
```

Now that `vec_min` does not acquire ownership of the vector anymore, we can call it multiple times on the same vector and also do things like

```
fn shared_ref_demo() {
    let v = vec![5,4,3,2,1];
    let first = &v[0];
    vec_min(&v);
    vec_min(&v);
    println!("The first element is: {}", *first);
}
```

What's going on here? First, `&` is how you lend ownership to someone - this operator creates a shared reference. `shared_ref_demo` creates three shared references to `v`: The reference `first` begins in the 2nd line of the function and lasts all the way to the end. The other two references, created for calling `vec_min`, only last for the duration of that respective call.

Technically, of course, references are pointers. Notice that since `vec_min` only gets a shared reference, Rust knows that it cannot mutate `v`. Hence the pointer into the buffer of `v` that was created before calling `vec_min` remains valid.

Unique, mutable references

There is a second way to borrow something, a second kind of reference: The *mutable reference*. This is a reference that comes with the promise that nobody else has *any kind of access* to the referee - in contrast to shared references, there is no aliasing with mutable references. It is thus always safe to perform mutation through such a reference. Because there cannot be another reference to the same data, we could also call it a *unique* reference, but that is not their official name.

As an example, consider a function which increments every element of a vector by 1. The type `&mut Vec<i32>` is the type of mutable references to `Vec<i32>`. Because the reference is mutable, we can use a mutable iterator, providing mutable references to the elements.

Here's an example of calling `vec_inc`.

```
fn vec_inc(v: &mut Vec<i32>) {
    for e in v.iter_mut() {
        *e += 1;
    }
}

fn mutable_ref_demo() {
    let mut v = vec![5,4,3,2,1];
    /* let first = &v[0]; */
    vec_inc(&mut v);
    vec_inc(&mut v);
    /* println!("The first element is: {}", *first); */             /* BAD! */
}
```

`&mut` is the operator to create a mutable reference. We have to mark `v` as mutable in order to create such a reference: Even though we completely own `v`, Rust tries to protect us from accidentally mutating things. Hence owned variables that you intend to mutate have to be annotated with `mut`. Because the reference passed to `vec_inc` only lasts as long as the function call, we can still call `vec_inc` on the same vector twice: The durations of the two references do not overlap, so we never have more than one mutable reference - we only ever borrow `v` once at a time. However, we can *not* create a shared reference that spans a call to `vec_inc`. Just try enabling the commented-out lines, and watch Rust complain. This is because `vec_inc` could mutate the vector structurally (i.e., it could add or remove elements), and hence the reference `first` could become invalid. In other words, Rust keeps us safe from bugs like the one in the C++ snippet above.

Above, I said that having a mutable reference excludes aliasing. But if you look at the code above carefully, you may say: "Wait! Don't the `v` in `mutable_ref_demo` and the `v` in `vec_inc` alias?" And you are right, they do. However, the `v` in `mutable_ref_demo` is not actually usable, it is not *active*. As long as `v` is borrowed, Rust will not allow you to do anything with it.

Summary

The ownership and borrowing system of Rust enforces the following three rules:

- There is always exactly one owner of a piece of data
- If there is an active mutable reference, then nobody else can have active access to the data
- If there is an active shared reference, then every other active access to the data is also a shared reference

As it turns out, combined with the abstraction facilities of Rust, this is a very powerful mechanism to tackle many problems beyond basic memory safety. You will see some examples for this soon.