Moves, copies and clones in Rust

2020-08-12

Introduction

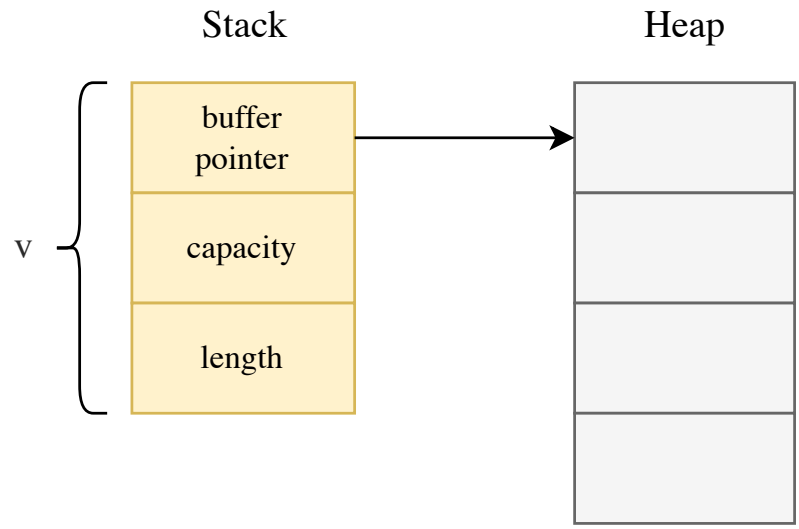
Moves and copies are fundamental concepts in Rust. These might be completely new to programmers coming from garbage collected languages like Ruby, Python or C#. While these terms do exist in C++, their meaning in Rust is subtly different. In this post I'll explain what it means for values to be moved, copied or cloned in Rust. Let's dive in.

Moves

As shown in *Memory safety in Rust - part 2*, assigning one variable to another transfers the ownership to the assignee:

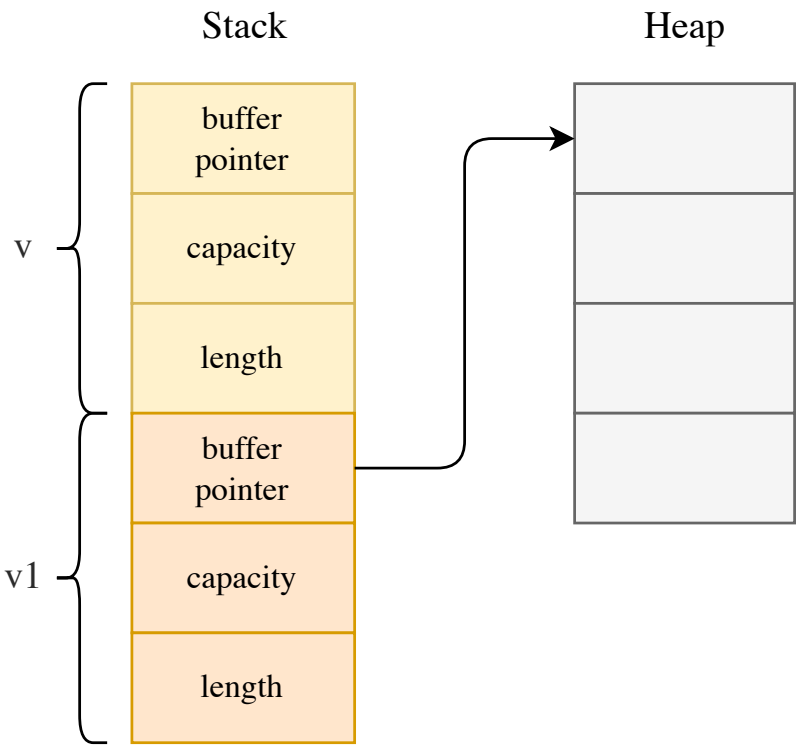
```
let v:Vec<i32> = Vec::new();
let v1 = v;//v1 is the new owner
```

In the above example, `v` is moved to `v1`. But what does it mean to *move* `v`? To understand that, we need to see how a `Vec` is laid out in memory:



A `Vec` has to maintain a dynamically growing or shrinking buffer. This buffer is allocated on the heap and contains the actual elements of the `Vec`. In addition, a `Vec` also has a small object on the stack. This object contains some housekeeping information: a pointer to the buffer on the heap, the capacity of the buffer and the length (i.e. how much of the capacity is currently filled).

When the variable `v` is moved to `v1`, the object on the stack is bitwise copied:



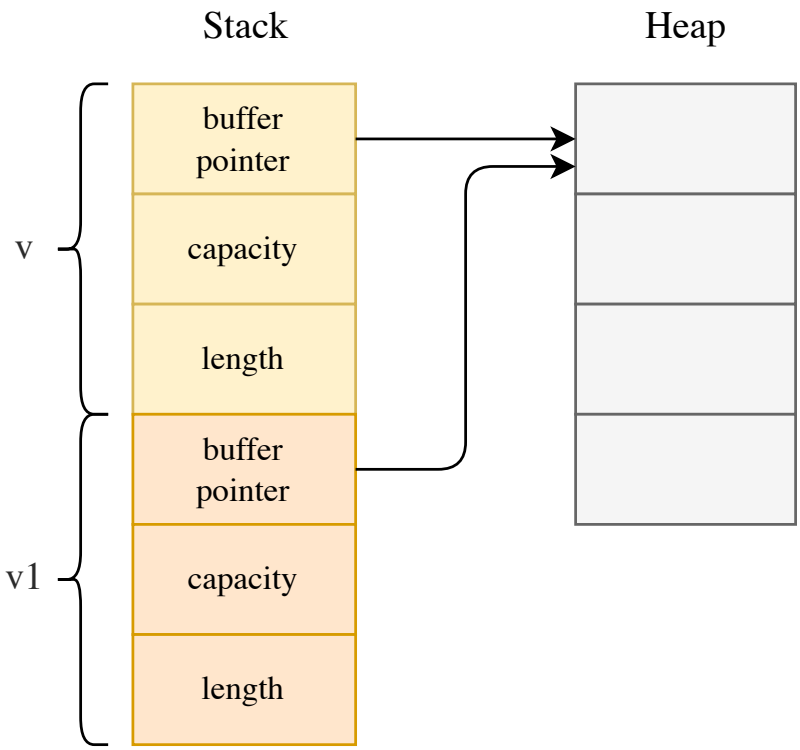
NOTE

What has essentially happened in the previous example is a shallow copy. This is in stark contrast to C++, which makes a deep copy when a vector is assigned to another variable.

The buffer on the heap stays intact. This is indeed a move: it is now `v1`'s responsibility to drop the heap buffer and `v` can't touch it:

```
let v: Vec<i32> = Vec::new();
let v1 = v;
println!("v's length is {}", v.len());//error: borrow of moved value:
```

This change of ownership is good because if access was allowed through both `v` and `v1` then you will end up with two stack objects pointing to the same heap buffer:



Which object should drop the buffer in this case? Because that is not clear, Rust prevents this situation from arising at all.

Assignment is not the only operation which involves moves. Values are also moved when passed as arguments or returned from functions:

```
let v: Vec<i32> = Vec::new();
//v is first moved into print_len's v1
//and then moved into v2 when print_len returns it
let v2 = print_len(v);
fn print_len(v1: Vec<i32>) -> Vec<i32> {
    println!("v1's length is {}", v1.len());
    v1//v1 is moved out of the function
}
```

Or assigned to members of a struct or enum:

```
struct Numbers {
    nums: Vec<i32>
}
let v: Vec<i32> = Vec::new();
//v moved into nums field of the Numbers struct
let n = Numbers { nums: v };

enum NothingOrString {
    Nothing,
    Str(String)
}
let s: String = "I am moving soon".to_string();
//s moved into the enum
let nos = NothingOrString::Str(s);
```

That's all about moves. Next let's take a look at copies.

Copies

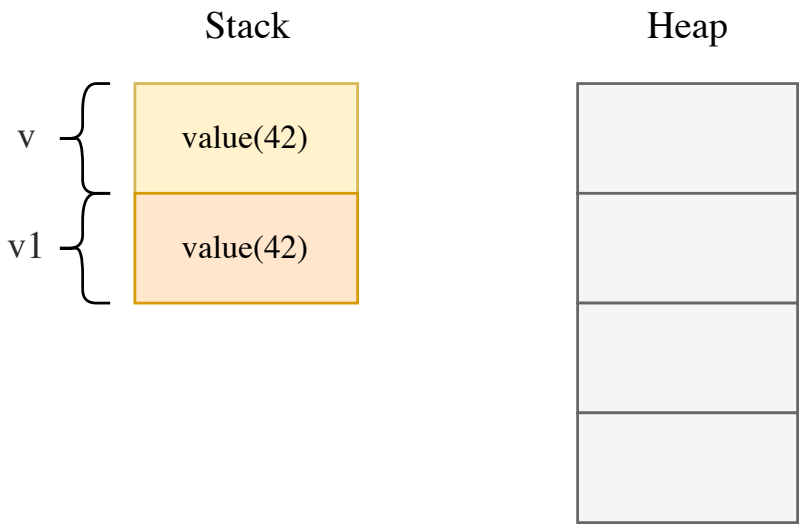
Remember this example from above?:

```
let v: Vec<i32> = Vec::new();
let v1 = v;
println!("v's length is {}", v.len());//error: borrow of moved value:
```

What happens if we change the type of the variables `v` and `v1` from `Vec` to `i32` :

```
let v: i32 = 42;
let v1 = v;
println!("v is {}", v);//compiles fine, no error!
```

This is almost the same code. Why doesn't the assignment operator move `v` into `v1` this time? To see that, let's take a look at the memory layout again:



In this example the values are contained entirely in the stack. There is nothing to own on the heap. That is why it is ok to allow access through both `v` and `v1` — they are completely independent copies.

Such types which do not own other resources and can be bitwise copied are called `Copy` types. They implement the `Copy` marker trait. All primitive types like integers, floats and characters are `Copy` . Structs or enums are not `Copy` by default but you can derive the `Copy` trait:

```
#[derive(Copy, Clone)]
struct Point {
    x: i32,
    y: i32,
}

#[derive(Copy, Clone)]
enum SignedOrUnsignedInt {
    Signed(i32),
    Unsigned(u32),
}
```

NOTE

`Clone` in the derive clause is needed because `Copy` is defined like this: `pub trait Copy: Clone {}`

For `#[derive(Copy, Clone)]` to work, all the members of the struct or enum must be `Copy` themselves. For example, this will not work:

```
//error:the trait `Copy` may not be implemented for this type
//because its nums field does not implement `Copy`
#[derive(Copy, Clone)]
struct Numbers {
    nums: Vec<i32>
}
```

You can of course also implement `Copy` and `Clone` manually:

```
struct Point {
    x: i32,
    y: i32,
}

//no method in Copy because it is a marker trait
impl Copy for Point {}
```

```
impl Clone for Point {
    fn clone(&self) -> Point {
        *self
    }
}
```

In general, any type that implements `Drop` cannot be `Copy` because `Drop` is implemented by types which own some resource and hence cannot be simply bitwise copied. But `Copy` types should be trivially copyable. Hence, `Drop` and `Copy` don't mix well.

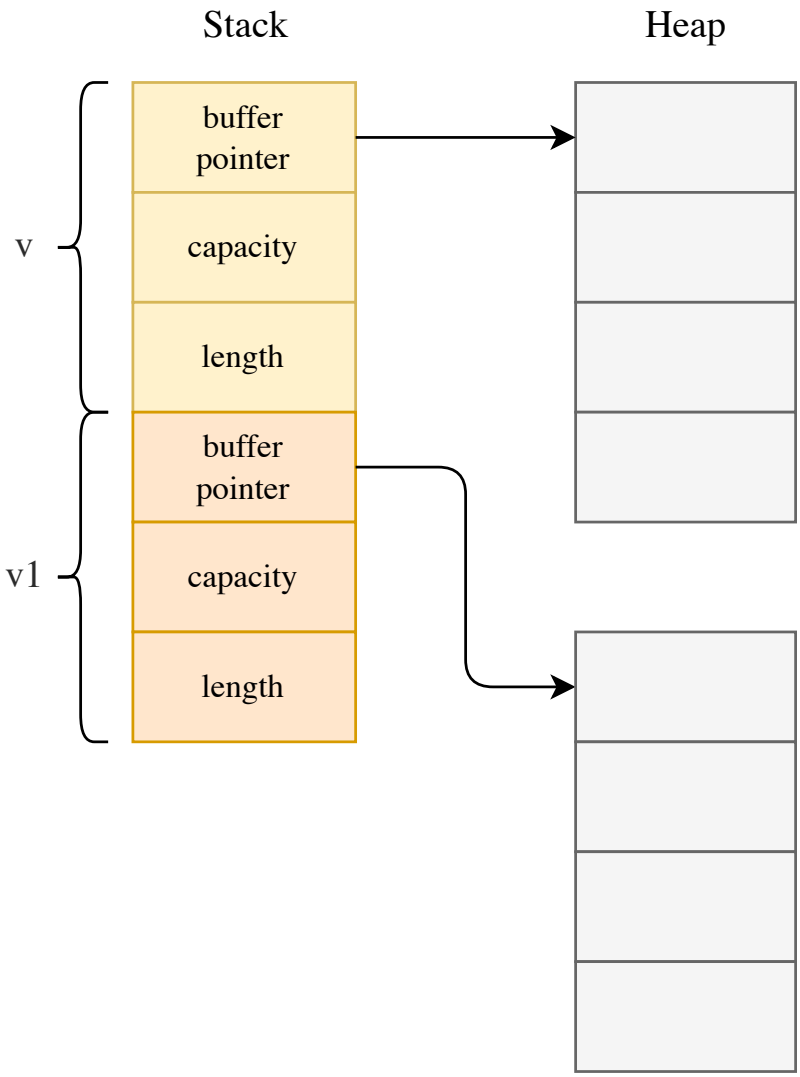
And that's all about copies. On to clones.

Clones

When a value is moved, Rust does a shallow copy; but what if you want to create a deep copy like in C++? To allow that, a type must first implement the `Clone` trait. Then to make a deep copy, client code should call the `clone` method:

```
let v: Vec<i32> = Vec::new();
let v1 = v.clone();//ok since Vec implements Clone
println!("v's length is {}", v.len());//ok
```

This results in the following memory layout after the `clone` call:



Due to deep copying, both `v` and `v1` are free to independently drop their heap buffers.

NOTE

The `clone` method doesn't always create a deep copy. Types are free to implement `clone` any way they want, but semantically it should be close enough to the meaning of duplicating an object. For example, `Rc` and `Arc` increment a reference count instead.

And that's all about clones.

Conclusion

In this post I took a deeper look at semantics of moves, copies and clones in Rust. I have tried to capture the nuance in meaning when compared with C++.

Rust is great because it has great defaults. For example, the assignment operator in Rust either moves values or does trivial bitwise copies. In C++, on the other hand, an innocuous looking assignment can hide loads of code that runs as part of overloaded assignment operators. In Rust, such code is brought into the open because the programmer has to explicitly call the `clone` method.

One could argue that both languages make different trade-offs but I like the extra safety guarantees Rust brings to the table due to these design choices.

GET NOTIFIED WHEN A NEW POST COMES OUT.

FIRST NAME

EMAIL

SUBSCRIBE