More moves, copies and clones

Moves

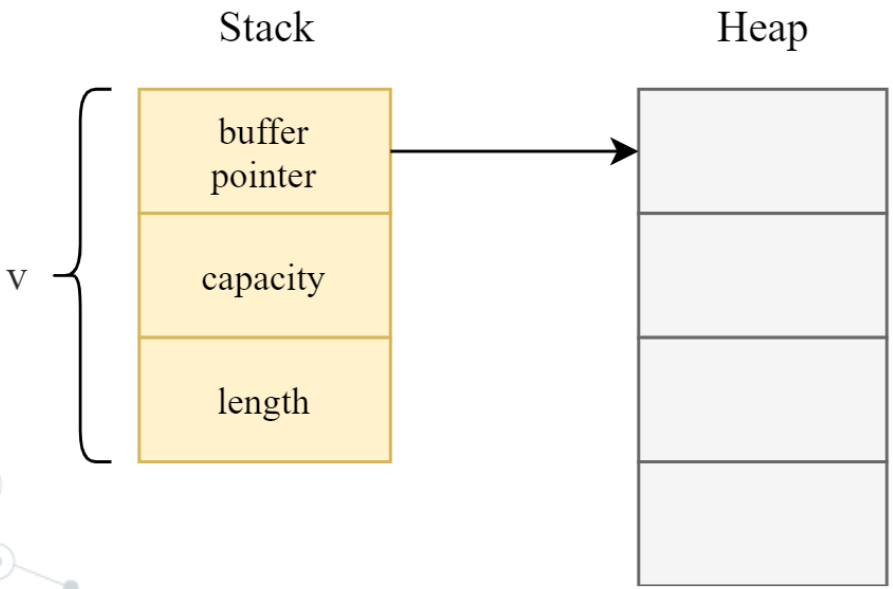
- Assigning one variable to another transfers the ownership to the assignee.

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

- v is moved to v1. But what does it mean to move v?

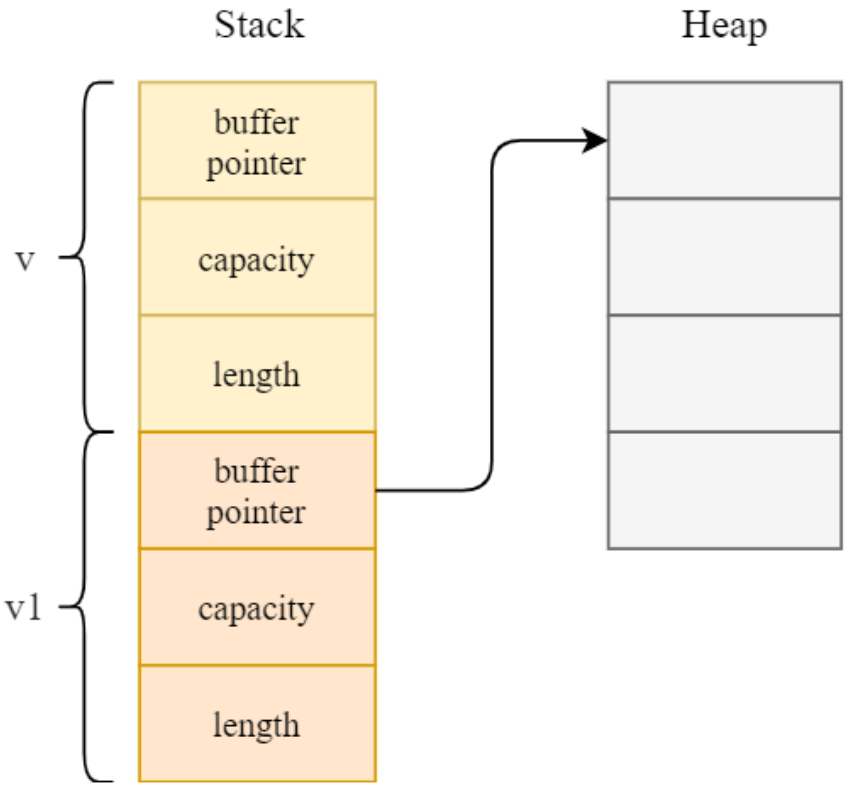
Moves

- A Vec maintains a dynamically growing /shrinking buffer. This buffer is allocated on the **heap**
- 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.



Moves

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



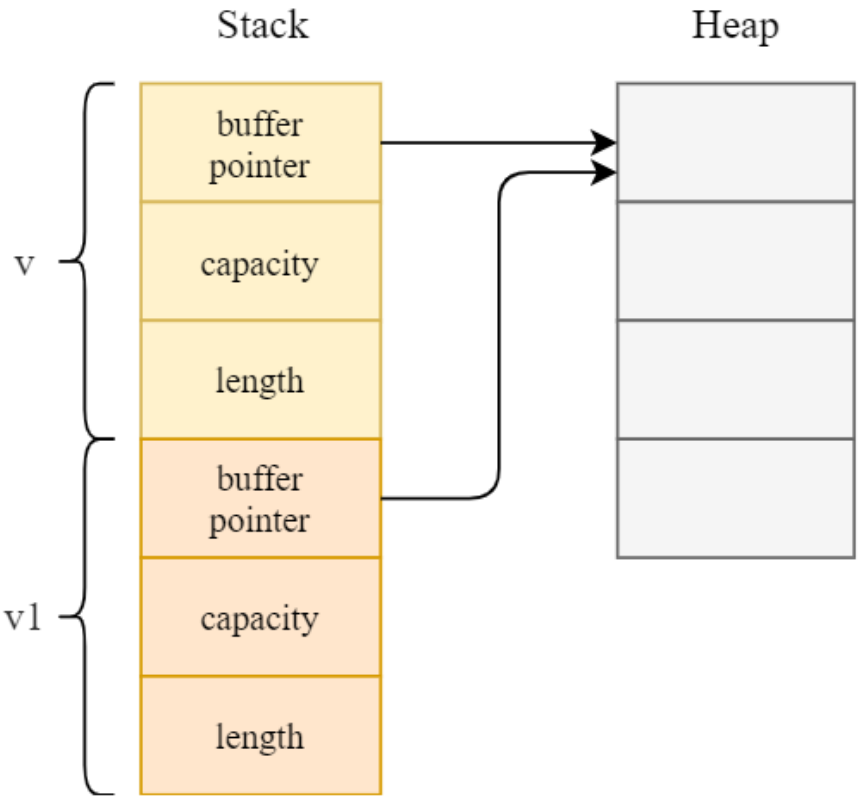
Moves

- 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: `v`
```

Moves

- 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:



Copies

- Remember this example?:

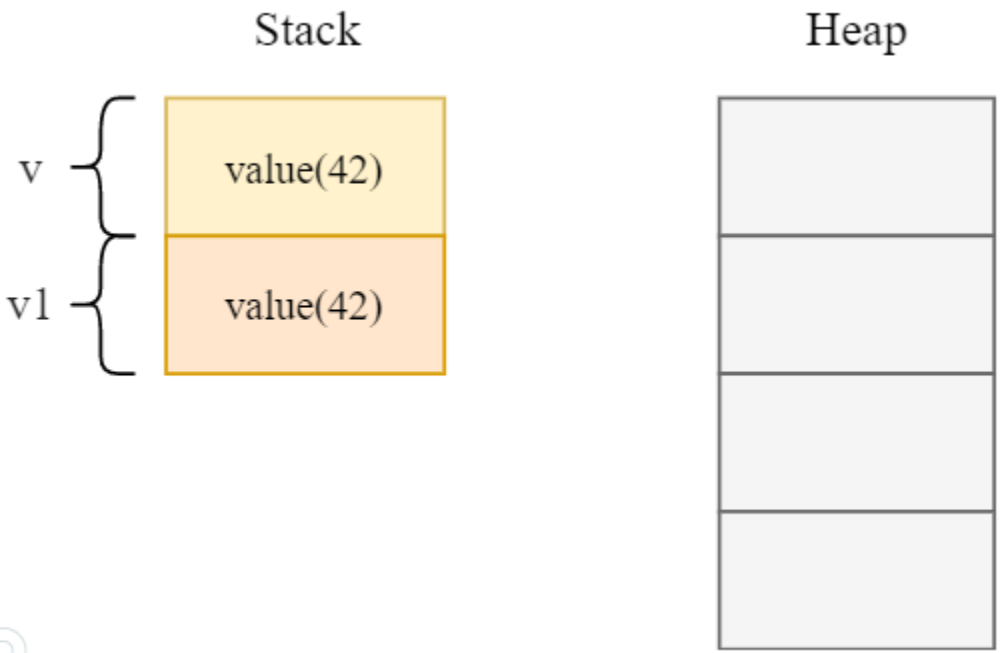
```
let v: Vec<i32> = Vec::new();
let v1 = v;
println!("v's length is {}", v.len());//error:
borrow of moved value: `v`
```

- 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!
```

Copies

- This is almost the same code. Why doesn't the assignment operator move v into v1 this time?



Copies

- **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.

Copies

- Any type that implements **Drop** cannot be Copy,
- Drop is implemented by types which own some resource and hence cannot be simply bitwise copied.
- Copy types must be trivially copyable. Hence, Drop and Copy don't mix well.

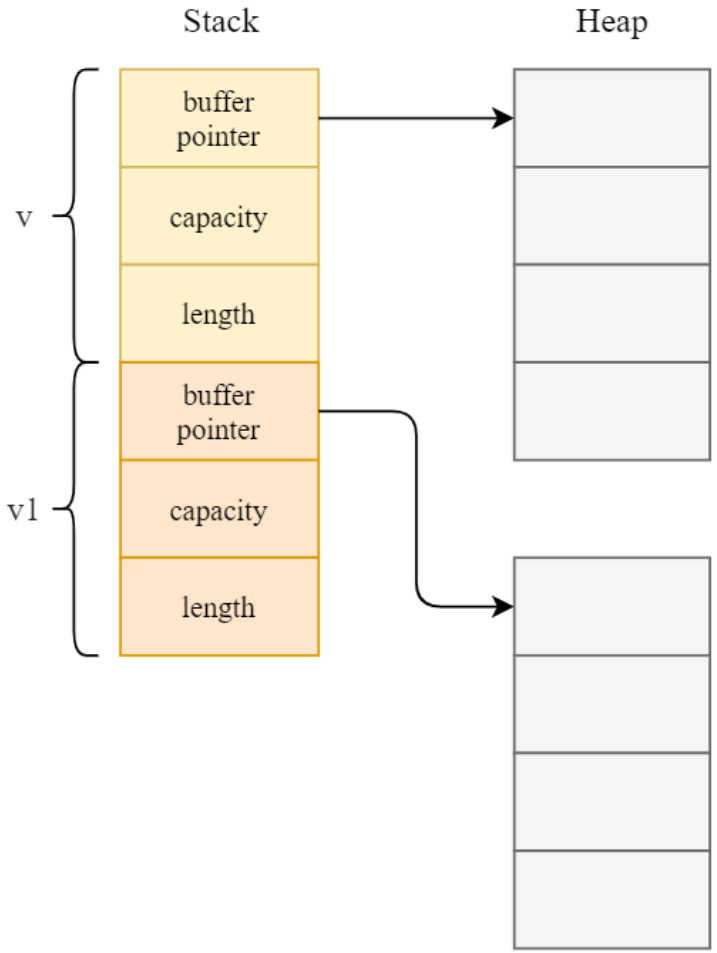
Clones

- When a value is moved, Rust does a shallow copy.
- What if you want to create a deep copy?
- 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
```

Clones

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



Clones

- Due to deep copying, both v and v1 are free to independently drop their heap buffers.
- Types are free to implement clone any way they want, but semantically it should be close enough to the meaning of duplicating an object.