Rust-101, Part 05: Clone

Big Numbers

In the course of the next few parts, we are going to build a data-structure for computations with *big* numbers. We would like to not have an upper bound to how large these numbers can get, with the memory of the machine being the only limit.

We start by deciding how to represent such big numbers. One possibility here is to use a vector "digits" of the number. This is like "1337" being a vector of four digits (1, 3, 3, 7), except that we will use `u64` as type of our digits, meaning we have 2^64 individual digits. Now we just have to decide the order in which we store numbers. I decided that we will store the least significant digit first. This means that "1337" would actually become (7, 3, 3, 1).
Finally, we declare that there must not be any trailing zeros (corresponding to useless leading zeros in our usual way of writing numbers). This is to ensure that the same number can only be stored in one way.

To write this down in Rust, we use a `struct`, which is a lot like structs in C: Just a bunch of named fields. Every field can be private to the current module (which is the default), or public (which is indicated by a `pub` in front of the name). For the sake of the tutorial, we make `data` public - otherwise, the next parts of this course could not work on `BigInt`s. Of course, in a real program, one would make the field private to ensure that the invariant (no trailing zeros) is maintained.

```
pub struct BigInt {
    pub data: Vec<u64>, // least significant digit first, no trailing zeros
}
```

Now that we fixed the data representation, we can start implementing methods on it.

```
impl BigInt {
```

Let's start with a constructor, creating a `BigInt` from an ordinary integer. To create an instance of a struct, we write its name followed by a list of fields and initial values assigned to them.

```
    pub fn new(x: u64) -> Self {
        if x == 0 {
            BigInt { data: vec![] }
        } else {
            BigInt { data: vec![x] }
        }
    }
```

It can often be useful to encode the invariant of a data-structure in code, so here is a check that detects useless trailing zeros.

```
    pub fn test_invariant(&self) -> bool {
        if self.data.len() == 0 {
            true
        } else {
            self.data[self.data.len() - 1] != 0
        }
    }
```

We can convert any little-endian vector of digits (i.e., least-significant digit first) into a number, by removing trailing zeros. The `mut` declaration for `v` here is just like the one in `let mut ...`: We completely own `v`, but Rust still asks us to make our intention of modifying it explicit. This `mut` is *not* part of the type of `from_vec` - the caller has to give up ownership of `v` anyway, so they don't care anymore what you do to it.

```
    pub fn from_vec(mut v: Vec<u64>) -> Self {
        unimplemented!()
    }
}
```

Exercise 05.1: Implement this function.

Hint: You can use `pop` to remove the last element of a vector.

Cloning

If you take a close look at the type of `BigInt::from_vec`, you will notice that it consumes the vector `v`. The caller hence loses access to its vector. However, there is something we can do if we don't want that to happen: We can explicitly `clone` the vector, which means that a full (or *deep*) copy will be performed. Technically, `clone` takes a borrowed vector in the form of a shared reference, and returns a fully owned one.

```
fn clone_demo() {
    let v = vec![0,1 << 16];
    let b1 = BigInt::from_vec((&v).clone());
    let b2 = BigInt::from_vec(v);
}
```

Rust has special treatment for methods that borrow their `self` argument (like `clone`, or like `test_invariant` above): It is not necessary to explicitly borrow the receiver of the method. Hence you could replace `(&v).clone()` by `v.clone()` above. Just try it!

To be clonable is a property of a type, and as such, naturally expressed with a trait. In fact, Rust already comes with a trait `Clone` for exactly this purpose. We can hence make our `BigInt` clonable as well.

```
impl Clone for BigInt {
    fn clone(&self) -> Self {
        BigInt { data: self.data.clone() }
    }
}
```

Making a type clonable is such a common exercise that Rust can even help you doing it: If you add `#[derive(Clone)]` right in front of the definition of `BigInt`, Rust will generate an implementation of `Clone` that simply clones all the fields. Try it! These `#[...]` annotations at types (and functions, modules, crates) are called *attributes*. We will see some more examples of attributes later.

We can also make the type `SomethingOrNothing<T>` implement `Clone`. However, that can only work if `T` is `Clone`! So we have to add this bound to `T` when we introduce the type variable.

```
use part02::{SomethingOrNothing,Something,Nothing};
impl<T: Clone> Clone for SomethingOrNothing<T> {
    fn clone(&self) -> Self {
        match *self {
            Nothing => Nothing,
            Something(ref v) => Something(v.clone()),
        }
    }
}
```

In the second arm of the match, we need to talk about the value `v` that's stored in `self`. However, if we were to write the pattern as `Something(v)`, that would indicate that we *own* `v` in the code after the arrow. That can't work though, we have to leave `v` owned by whoever called us - after all, we don't even own `self`, we just borrowed it. By writing `Something(ref v)`, we borrow `v` for the duration of the match arm. That's good enough for cloning it.

Again, Rust will generate this implementation automatically if you add `#[derive(Clone)]` right before the definition of `SomethingOrNothing`.

Exercise 05.2: Write some more functions on `BigInt`. What about a function that returns the number of digits? The number of non-zero digits? The smallest/largest digit? Of course, these should all take `self` as a shared reference (i.e., in borrowed form).

Mutation + aliasing considered harmful (part 2)

Now that we know how to create references to contents of an `enum` (like `v` above), there's another example we can look at for why we have to rule out mutation in the presence of aliasing. First, we define an `enum` that can hold either a number, or a string.

```
enum Variant {
    Number(i32),
    Text(String),
}
```

Now consider the following piece of code. Like above, `n` will be a reference to a part of `var`, and since we wrote `ref mut`, the reference will be unique and mutable. In other words, right after the match, `ptr` points to the number that's stored in `var`, where `var` is a `Number`. Remember that `_` means "we don't care".

```
fn work_on_variant(mut var: Variant, text: String) {
    let mut ptr: &mut i32;
    match var {
        Variant::Number(ref mut n) => ptr = n,
        Variant::Text(_) => return,
    }
    /* var = Variant::Text(text); */                              /* BAD! */
    *ptr = 1337;
}
```

Now, imagine what would happen if we were permitted to also mutate `var`. We could, for example, make it a `Text`. However, `ptr` still points to the old location! Hence `ptr` now points somewhere into the representation of a `String`. By changing `ptr`, we manipulate the string in completely unpredictable ways, and anything could happen if we were to use it again! (Technically, the first field of a `String` is a pointer to its character data, so by overwriting that pointer with an integer, we make it a completely invalid address. When the destructor of `var` runs, it would try to deallocate that address, and Rust would eat your laundry - or whatever.)

I hope this example clarifies why Rust has to rule out mutation in the presence of aliasing *in general*, not just for the specific case of a buffer being reallocated, and old pointers becoming hence invalid.