part06.rs

Rust-101, Part 06: Copy, Lifetimes

We continue to work on our `BigInt`, so we start by importing what we already established.

With `BigInt` being about numbers, we should be able to write a version of `vec_min` that computes the minimum of a list of `BigInt`. First, we have to write `min` for `BigInt`.

Just to be sure, we first check that both operands actually satisfy our invariant. `debug_assert!` is a macro that checks that its argument (must be of type `bool`) is `true`, and panics otherwise. It gets removed in release builds, which you do with `cargo build --release`.

Now our assumption of having no trailing zeros comes in handy: If the lengths of the two numbers differ, we already know which is larger.

Exercise 06.1: Fill in this code.

Now we can write `vec_min`.

If `v` is a shared reference to a vector, then the default for iterating over it is to call `iter`, the iterator that borrows the elements.

Now, what's happening here? Why do we have to to make a full (deep) copy of `e`, and why did we not have to do that in our previous version?

The answer is already hidden in the type of `vec_min`: `v` is just borrowed, but the Option that it returns is *owned*. We can't just return one of the elements of `v`, as that would mean that it is no longer in the vector! In our code, this comes up when we update the intermediate variable `min`, which also has type `Option<BigInt>`. If you get rid of the `e.clone()`, Rust will complain "Cannot move out of borrowed content". That's because `e` is a `&BigInt`. Assigning `min = Some(*e)` works just like a function call: Ownership of the underlying data is transferred from `e` to `min`. But that's not allowed, since we just borrowed `e`, so we cannot empty it! We can, however, call `clone` on it. Then we own the copy that was created, and hence we can store it in `min`.
Of course, making such a full copy is expensive, so we'd like to avoid it. We'll come to that in the next part.

Copy types

But before we go there, I should answer the second question I brought up above: Why did our old `vec_min` work? We stored the minimal `i32` locally without cloning, and Rust did not complain. That's because there isn't really much of an "ownership" when it comes to types like `i32` or `bool`: If you move the value from one place to another, then both instances are "complete". We also say the value has been *duplicated*. This is in stark contrast to types like `Vec<i32>`, where moving the value results in both the old and the new vector to point to the same underlying buffer. We don't have two vectors, there's no proper duplication.

Rust calls types that can be easily duplicated `Copy` types. `Copy` is another trait, and it is implemented for types like `i32` and `bool`. Remember how we defined the trait `Minimum` by writing `trait Minimum : Copy { ... }`? This tells Rust that every type that implements `Minimum` must also implement `Copy`, and that's why the compiler accepted our generic `vec_min` in part 02. `Copy` is the first *marker trait* that we encounter: It does not provide any methods, but makes a promise about the behavior of the type - in this case, being duplicable.

If you try to implement `Copy` for `BigInt`, you will notice that Rust does not let you do that. A type can only be `Copy` if all its elements are `Copy`, and that's not the case for `BigInt`. However, we can make `SomethingOrNothing<T>` copy if `T` is `Copy`.

Again, Rust can generate implementations of `Copy` automatically. If you add `#[derive(Copy,Clone)]` right before the definition of `SomethingOrNothing`, both `Copy` and `Clone` will automatically be implemented.

An operational perspective

Instead of looking at what happens "at the surface" (i.e., visible in Rust), one can also explain ownership passing and how `Copy` and `Clone` fit in by looking at what happens on the machine.
When Rust code is executed, passing a value (like `i32` or `Vec<i32>`) to a function will always result in a shallow copy being performed: Rust just copies the bytes representing that value, and considers itself done. That's just like the default copy constructor in C++. Rust, however, will consider this a destructive operation: After copying the bytes elsewhere, the original value must no longer be used. After all, the two could now share a pointer! If, however, you mark a type `Copy`, then Rust will *not* consider a move destructive, and just like in C++, the old and new value can happily coexist. Now, Rust does not allow you to overload the copy constructor. This means that passing a value around will always be a fast operation, no allocation or any other kind of heap access will happen. In the situations where you would write a copy constructor in C++ (and hence incur a hidden cost on every copy of this type), you'd have the type *not* implement `Copy`, but only `Clone`. This makes the cost explicit.

Lifetimes

To fix the performance problems of `vec_min`, we need to avoid using `clone`. We'd like the return value to not be owned (remember that this was the source of our need for cloning), but *borrowed*. In other words, we want to return a shared reference to the minimal element.

The function `head` demonstrates how that could work: It returns a reference to the first element of a vector if it is non-empty. The type of the function says that it will either return nothing, or it will return a borrowed `T`. We can then obtain a reference to the first element of `v` and use it to construct the return value.

Technically, we are returning a pointer to the first element. But doesn't that mean that callers have to be careful? Imagine `head` would be a C++ function, and we would write the following code.

This is very much like our very first motivating example for ownership, at the beginning of part 04: `push_back` could reallocate the buffer, making `first` an invalid pointer. Again, we have aliasing (of `first` and `v`) and mutation. But this time, the bug is hidden behind the call to `head`. How does Rust solve this? If we translate the code above to Rust, it doesn't compile, so clearly we are good - but how and why? (Notice that we use `unwrap` to explicitly assert that `first` is not `None`, whereas the C++ code above would silently dereference a `NULL`-pointer. But that's another point.)

To give the answer to this question, we have to talk about the *lifetime* of a reference. The point is, saying that you borrowed your friend a `Vec<i32>`, or a book, is not good enough, unless you also agree on *how long* your friend can borrow it. After all, you need to know when you can rely on owning your data (or book) again.

Every reference in Rust has an associated lifetime, written `&'a T` for a reference with lifetime `'a` to something of type `T`. The full type of `head` reads as follows: `fn<'a, T>(&'a Vec<T>) -> Option<&'a T>`. Here, `'a` is a *lifetime variable*, which represents for how long the vector has been borrowed. The function type expresses that argument and return value have *the same lifetime*.

When analyzing the code of `rust_foo`, Rust has to assign a lifetime to `first`. It will choose the scope where `first` is valid, which is the entire rest of the function. Because `head` ties the lifetime of its argument and return value together, this means that `&v` also has to borrow `v` for the entire duration of the function `rust_foo`. So when we try to create a unique reference to `v` for `push`, Rust complains that the two references (the one for `head`, and the one for `push`) overlap, so neither of them can be unique. Lucky us! Rust caught our mistake and made sure we don't crash the program.

So, to sum this up: Lifetimes enable Rust to reason about *how long* a reference is valid, how long ownership has been borrowed. We can thus safely write functions like `head`, that return references into data they got as argument, and make sure they are used correctly, *while looking only at the function type*. At no point in our analysis of `rust_foo` did we have to look *into* `head`. That's, of course, crucial if we want to separate library code from application code. Most of the time, we don't have to explicitly add lifetimes to function types. This is thanks to *lifetime elision*, where Rust will automatically insert lifetimes we did not specify, following some simple, well-documented rules.


```
use part05::BigInt;


impl BigInt {
    fn min_try1(self, other: Self) -> Self {



        debug_assert!(self.test_invariant() && other.test_invariant());




        if self.data.len() < other.data.len() {
            self
        } else if self.data.len() > other.data.len() {
            other
        } else {

            unimplemented!()
        }
    }
}
fn vec_min(v: &Vec<BigInt>) -> Option<BigInt> {
    let mut min: Option<BigInt> = None;

    for e in v {
        let e = e.clone();
        min = Some(match min {
            None => e,
            Some(n) => e.min_try1(n)
        });
    }
    min
}
```

```
use part02::{SomethingOrNothing,Something,Nothing};
impl<T: Copy> Copy for SomethingOrNothing<T> {}
```

```
fn head<T>(v: &Vec<T>) -> Option<&T> {
    if v.len() > 0 {
        Some(&v[0])
    } else {
        None
    }
}
/*
  int *foo(std::vector<int> v) {
    int *first = head(v);
    v.push_back(42);
    return *first;
  }
*/
fn rust_foo(mut v: Vec<i32>) -> i32 {
    let first: Option<&i32> = head(&v);
    /* v.push(42); */
    *first.unwrap()
}
```