part02.rs

Rust-101, Part 02: Generic types, Traits

Let us for a moment reconsider the type `NumberOrNothing`. Isn't it a bit annoying that we had to hard-code the type `i32` in there? What if tomorrow, we want a `CharOrNothing`, and later a `FloatOrNothing`? Certainly we don't want to re-write the type and all its inherent methods.

Generic datatypes

The solution to this is called *generics* or *polymorphism* (the latter is Greek, meaning "many shapes"). You may know something similar from C++ (where it's called *templates*) or Java, or one of the many functional languages. So here, we define a generic type `SomethingOrNothing`.

Instead of writing out all the variants, we can also just import them all at once.

What this does is define an entire family of types: We can now write `SomethingOrNothing<i32>` to get back our `NumberOrNothing`.

However, we can also write `SomethingOrNothing<bool>` or even `SomethingOrNothing<SomethingOrNothing<i32>>`. In fact, a type like `SomethingOrNothing` is so useful that it is already present in the standard library: It's called an *option type*, written `Option<T>`. Go check out its documentation! (And don't worry, there's indeed lots of material mentioned there that we have not covered yet.)

```
pub enum SomethingOrNothing<T>  {
    Something(T),
    Nothing,
}


pub use self::SomethingOrNothing::*;

type NumberOrNothing = SomethingOrNothing<i32>;
```

Generic `impl`, Static functions

The types are so similar, that we can provide a generic function to construct a `SomethingOrNothing<T>` from an `Option<T>`, and vice versa.

Notice the syntax for giving generic implementations to generic types: Think of the first `<T>` as *declaring* a type variable ("I am doing something for all types `T`"), and the second `<T>` as *using* that variable ("The thing I do, is implement `SomethingOrNothing<T>`").

Inside an `impl`, `Self` refers to the type we are implementing things for. Here, it is an alias for `SomethingOrNothing<T>`. Remember that `self` is the `this` of Rust, and implicitly has type `Self`.

Observe how `new` does *not* have a `self` parameter. This corresponds to a `static` method in Java or C++. In fact, `new` is the Rust convention for defining constructors: They are nothing special, just static functions returning `Self`.

You can call static functions, and in particular constructors, as demonstrated in `call_constructor`.

```
impl<T> SomethingOrNothing<T> {
    fn new(o: Option<T>) -> Self {
        match o { None => Nothing, Some(t) => Something(t) }
    }

    fn to_option(self) -> Option<T> {
        match self { Nothing => None, Something(t) => Some(t) }
    }
}


fn call_constructor(x: i32) -> SomethingOrNothing<i32> {
    SomethingOrNothing::new(Some(x))
}
```

Traits

Now that we have a generic `SomethingOrNothing`, wouldn't it be nice to also have a generic `vec_min`? Of course, we can't take the minimum of a vector of *any* type. It has to be a type supporting a `min` operation. Rust calls such properties that we may demand of types *traits*.

So, as a first step towards a generic `vec_min`, we define a `Minimum` trait. For now, just ignore the `Copy`, we will come back to this point later. A `trait` is a lot like interfaces in Java: You define a bunch of functions you want to have implemented, and their argument and return types.
The function `min` takes two arguments of the same type, but I made the first argument the special `self` argument. I could, alternatively, have made `min` a static function as follows: `fn min(a: Self, b: Self) -> Self`. However, in Rust one typically prefers methods over static functions wherever possible.

Next, we write `vec_min` as a generic function over a type `T` that we demand to satisfy the `Minimum` trait. This requirement is called a *trait bound*. The only difference to the version from the previous part is that we call `e.min(n)` instead of `min_i32(n, e)`. Rust automatically figures out that `e` is of type `T`, which implements the `Minimum` trait, and hence we can call that function.

There is a crucial difference to templates in C++: We actually have to declare which traits we want the type to satisfy. If we left away the `Minimum`, Rust would have complained that we cannot call `min`. Just try it!
This is in strong contrast to C++, where the compiler only checks such details when the function is actually used.

Here, we can now call the `min` function of the trait.

```
pub trait Minimum : Copy {
    fn min(self, b: Self) -> Self;
}


pub fn vec_min<T: Minimum>(v: Vec<T>) -> SomethingOrNothing<T> {
    let mut min = Nothing;
    for e in v {
        min = Something(match min {
            Nothing => e,
            Something(n) => {
                e.min(n)
            }
        });
    }
    min
}
```

Before going on, take a moment to ponder the flexibility of Rust's take on abstraction: We just defined our own, custom trait (interface), and then implemented that trait *for an existing type*. With the hierarchical approach of, e.g., C++ or Java, that's not possible: We cannot make an existing type also inherit from our abstract base class after the fact.

In case you are worried about performance, note that Rust performs *monomorphisation* of generic functions: When you call `vec_min` with `T` being `i32`, Rust essentially goes ahead and creates a copy of the function for this particular type, filling in all the blanks. In this case, the call to `T::min` will become a call to our implementation *statically*. There is no dynamic dispatch, like there would be for Java interface methods or C++ `virtual` methods. This behavior is similar to C++ templates. The optimizer (Rust is using LLVM) then has all the information it could want to, e.g., inline function calls.

Trait implementations

To make `vec_min` usable with a `Vec<i32>`, we implement the `Minimum` trait for `i32`.

We again provide a `print` function. This also shows that we can have multiple `impl` blocks for the same type (remember that `NumberOrNothing` is just a type alias for `SomethingOrNothing<i32>`), and we can provide some methods only for certain instances of a generic type.

```
impl Minimum for i32 {
    fn min(self, b: Self) -> Self {
        if self < b { self } else { b }
    }
}

impl NumberOrNothing {
    pub fn print(self) {
        match self {
            Nothing => println!("The number is: <nothing>"),
            Something(n) => println!("The number is: {}", n),
        };
    }
}

fn read_vec() -> Vec<i32> {
    vec![18,5,7,3,9,27]
}
pub fn main() {
    let vec = read_vec();
    let min = vec_min(vec);
    min.print();
}
```

Now we are ready to run our new code. Remember to change `main.rs` appropriately. Rust figures out automatically that we want the `T` of `vec_min` to be `i32`, and that `i32` implements `Minimum` and hence all is good.

If this printed `3`, then your generic `vec_min` is working! So get ready for the next part.

Exercise 02.1: Change your program such that it computes the minimum of a `Vec<f32>` (where `f32` is the type of 32-bit floating-point numbers). You should not change `vec_min` in any way, obviously!