part03.rs

Rust-101, Part 03: Input

In part 00, I promised that we would eventually replace `read_vec` by a function that actually asks the user to enter a bunch of numbers. Unfortunately, I/O is a complicated topic, so the code to do that is not exactly pretty - but well, let's get that behind us.

I/O is provided by the module `std::io`, so we first have to import that with `use`. We also import the I/O *prelude*, which makes a bunch of commonly used I/O stuff directly available.

```rust
use std::io::prelude::*;
use std::io;
```

Let's now go over this function line-by-line. First, we call the constructor of `Vec` to create an empty vector. As mentioned in the previous part, `new` here is just a static function with no special treatment. While it is possible to call `new` for a particular type (`Vec::<i32>::new()`), the common way to make sure we get the right type is to annotate a type at the *variable*. It is this variable that we interact with for the rest of the function, so having its type available (and visible!) is much more useful. Without knowing the return type of `Vec::new`, specifying its type parameter doesn't tell us all that much.

```rust
fn read_vec() -> Vec<i32> {
    let mut vec: Vec<i32> = Vec::<i32>::new();
```

The central handle to the standard input is made available by the function `io::stdin`.

```rust
    let stdin = io::stdin();
    println!("Enter a list of numbers, one per line. End with Ctrl-D (Linux) or Ctrl-Z (Windows).");
```

We would now like to iterate over standard input line-by-line. We can use a `for` loop for that, but there is a catch: What happens if there is some other piece of code running concurrently, that also reads from standard input? The result would be a mess. Hence Rust requires us to `lock` standard input if we want to perform large operations on it. (See the documentation for more details.)

```rust
    for line in stdin.lock().lines() {
```

Rust's type for (dynamic, growable) strings is `String`. However, our variable `line` here is not yet of that type: It has type `io::Result<String>`. The problem with I/O is that it can always go wrong. The type of `line` is a lot like `Option<String>` ("a `String` or nothing"), but in the case of "nothing", there is additional information about the error. Again, I recommend to check the documentation. You will see that `io::Result` is actually just an alias for `Result`, so click on that to obtain the list of all constructors and methods of the type.

We will be lazy here and just assume that nothing goes wrong: `unwrap` returns the `String` if there is one, and panics the program otherwise. Since a `Result` carries some details about the error that occurred, there will be a somewhat reasonable error message. Still, you would not want a user to see such an error, so in a "real" program, we would have to do proper error handling. Can you find the documentation of `Result::unwrap`?

I chose the same name (`line`) for the new variable to ensure that I will never, accidentally, access the "old" `line` again.

```rust
        let line = line.unwrap();
```

Now that we have our `String`, we want to make it an `i32`. We first `trim` the `line` to remove leading and trailing whitespace. `parse` is a method on `String` that can convert a string to anything. Try finding its documentation!

In this case, Rust *could* figure out automatically that we need an `i32` (because of the return type of the function), but that's a bit too much magic for my taste. We are being more explicit here: `parse::<i32>` is `parse` with its generic type set to `i32`.

```rust
        match line.trim().parse::<i32>() {
```

`parse` returns again a `Result`, and this time we use a `match` to handle errors (like, the user entering something that is not a number). This is a common pattern in Rust: Operations that could go wrong will return `Option` or `Result`. The only way to get to the value we are interested in is through pattern matching (and through helper functions like `unwrap`). If we call a function that returns a `Result`, and throw the return value away, the compiler will emit a warning. It is hence impossible for us to *forget* handling an error, or to accidentally use a value that doesn't make any sense because there was an error producing it.

```rust
            Ok(num) => {
                vec.push(num)
            },
```

We don't care about the particular error, so we ignore it with a `_`.

```rust
            Err(_) => {
                println!("What did I say about numbers?")
            },
        }
    }

    vec
}
```

So much for `read_vec`. If there are any questions left, the documentation of the respective function should be very helpful. Try finding the one for `Vec::push`. I will not always provide the links, as the documentation is quite easy to navigate and you should get used to that.

For the rest of the code, we just re-use part 02 by importing it with `use`. I already sneaked a bunch of `pub` in part 02 to make this possible: Only items declared public can be imported elsewhere.

```rust
use part02::{SomethingOrNothing,Something,Nothing,vec_min};
```

If you update your `main.rs` to use part 03, `cargo run` should now ask you for some numbers, and tell you the minimum. Neat, isn't it?

```rust
pub fn main() {
    let vec = read_vec();
    let min = vec_min(vec);
    min.print();
}
```

Exercise 03.1: The goal is to write a generic version of `SomethingOrNothing::print`. To this end, define a trait `Print` that provides (simple) generic printing, and implement that trait for `i32`. Then define `SomethingOrNothing::print2` to use that trait, and change `main` above to use the new generic `print2` function. I will again provide a skeleton for this solution. It also shows how to attach bounds to generic implementations (just compare it to the `impl` block from the previous exercise). You can read this as "For all types `T` satisfying the `Print` trait, I provide an implementation for `SomethingOrNothing<T>`".

Notice that I called the function on `SomethingOrNothing` `print2` to disambiguate from the `print` defined previously.

Hint: There is a macro `print!` for printing without appending a newline.

```rust
pub trait Print {
    /* Add things here */
}
impl Print for i32 {
    /* Add things here */
}
impl<T: Print> SomethingOrNothing<T> {
    fn print2(self) {
        unimplemented!()
    }
}
```

Exercise 03.2: Building on exercise 02.2, implement all the things you need on `f32` to make your program work with floating-point numbers.