Rust-101, Part 13: Concurrency, Arc, Send

Our next stop are the concurrency features of Rust. We are going to write our own small version of "grep", called *rgrep*, and it is going to perform three jobs concurrently: One thread reads the input files, one thread does the actual matching, and one thread writes the output. I already mentioned in the beginning of the course that Rust's type system (more precisely, the discipline of ownership and borrowing) will help us to avoid a common pitfall of concurrent programming: data races. We will see how that works concretely.

```
use std::io::prelude::*;
use std::{io, fs, thread};
use std::sync::mpsc::{sync_channel, SyncSender, Receiver};
use std::sync::Arc;
```

Before we come to the actual code, we define a data-structure `Options` to store all the information we need to complete the job: Which files to work on, which pattern to look for, and how to output. Besides just printing all the matching lines, we will also offer to count them, or alternatively to sort them.

```
#[derive(Clone,Copy)]
pub enum OutputMode {
    Print,
    SortAndPrint,
    Count,
}
use self::OutputMode::*;

pub struct Options {
    pub files: Vec<String>,
    pub pattern: String,
    pub output_mode: OutputMode,
}
```

Now we can write three functions to do the actual job of reading, matching, and printing, respectively. To get the data from one thread to the next, we will use *message passing*. We will establish communication channels between the threads, with one thread *sending* data, and the other one *receiving* it. `SyncSender<T>` is the type of the sending end of a synchronous channel transmitting data of type `T`. *Synchronous* here means that the `send` operation could block, waiting for the other side to make progress. We don't want to end up with the entire file being stored in the buffer of the channels, and the output not being fast enough to keep up with the speed of input.

We also need all the threads to have access to the options of the job they are supposed to do. Since it would be rather unnecessary to actually copy these options around, we will use reference-counting to share them between all threads. `Arc` is the thread-safe version of `Rc`, using atomic operations to keep the reference count up-to-date.

The first function reads the files, and sends every line over the `out_channel`.

```
fn read_files(options: Arc<Options>, out_channel: SyncSender<String>) {
    for file in options.files.iter() {
```

First, we open the file, ignoring any errors.

```
        let file = fs::File::open(file).unwrap();
```

Then we obtain a `BufReader` for it, which provides the `lines` function.

```
        let file = io::BufReader::new(file);
        for line in file.lines() {
            let line = line.unwrap();
```

Now we send the line over the channel, ignoring the possibility of `send` failing.

```
            out_channel.send(line).unwrap();
        }
    }
}
```

When we drop the `out_channel`, it will be closed, which the other end can notice.

The second function filters the lines it receives through `in_channel` with the pattern, and sends matches via `out_channel`.

```
fn filter_lines(options: Arc<Options>,
                in_channel: Receiver<String>,
                out_channel: SyncSender<String>) {
    for line in in_channel.iter() {
```

We can simply iterate over the channel, which will stop when the channel is closed.

`contains` works on lots of types of patterns, but in particular, we can use it to test whether one string is contained in another. This is another example of Rust using traits as substitute for overloading.

```
        if line.contains(&options.pattern) {
            out_channel.send(line).unwrap();
        }
    }
}
```

The third function performs the output operations, receiving the relevant lines on its `in_channel`.

```
fn output_lines(options: Arc<Options>, in_channel: Receiver<String>) {
    match options.output_mode {
```

Here, we just print every line we see.

```
        Print => {
            for line in in_channel.iter() {
                println!("{}", line);
            }
        },
```

We are supposed to count the number of matching lines. There's a convenient iterator adapter that we can use for this job.

```
        Count => {
            let count = in_channel.iter().count();
            println!("{} hits for {}.", count, options.pattern);
        },
```

We are asked to sort the matching lines before printing. So let's collect them all in a local vector...

```
        SortAndPrint => {
            let mut data: Vec<String> = in_channel.iter().collect();
```

...and implement the actual sorting later.

```
            unimplemented!()
        }
    }
}
```

With the operations of the three threads defined, we can now implement a function that performs grepping according to some given options.

```
pub fn run(options: Options) {
```

We move the `options` into an `Arc`, as that's what the thread workers expect.

```
    let options = Arc::new(options);
```

This sets up the channels. We use a `sync_channel` with buffer-size of 16 to avoid needlessly filling RAM.

```
    let (line_sender, line_receiver) = sync_channel(16);
    let (filtered_sender, filtered_receiver) = sync_channel(16);
```

Spawn the read thread: `thread::spawn` takes a closure that is run in a new thread. The `move` keyword again tells Rust that we want ownership of captured variables to be moved into the closure. This means we need to do the `clone` *first*, otherwise we would lose our `options` to the new thread!

```
    let options1 = options.clone();
    let handle1 = thread::spawn(move || read_files(options1, line_sender));
```

Same with the filter thread.

```
    let options2 = options.clone();
    let handle2 = thread::spawn(move || {
        filter_lines(options2, line_receiver, filtered_sender)
    });
```

And the output thread.

```
    let options3 = options.clone();
    let handle3 = thread::spawn(move || output_lines(options3, filtered_receiver));
```

Finally, wait until all three threads did their job. Joining a thread waits for its termination. This can fail if that thread panicked: In this case, we could get access to the data that it provided to `panic!`. Here, we just assert that they did not panic - so we will panic ourselves if that happened.

```
    handle1.join().unwrap();
    handle2.join().unwrap();
    handle3.join().unwrap();
}
```

Now we have all the pieces together for testing our rgrep with some hard-coded options. We need to call `to_string` on string literals to convert them to a fully-owned `String`.

```
pub fn main() {
    let options = Options {
        files: vec!["src/part10.rs".to_string(),
                    "src/part11.rs".to_string(),
                    "src/part12.rs".to_string()],
        pattern: "let".to_string(),
        output_mode: Print
    };
    run(options);
}
```

Exercise 13.1: Change rgrep such that it prints not only the matching lines, but also the name of the file and the number of the line in the file. You will have to change the type of the channels from `String` to something that records this extra information.

Ownership, Borrowing, and Concurrency

The little demo above showed that concurrency in Rust has a fairly simple API. Considering Rust has closures, that should not be entirely surprising. However, as it turns out, Rust goes well beyond this and actually ensures the absence of data races.

A data race is typically defined as having two concurrent, unsynchronized accesses to the same memory location, at least one of which is a write. In other words, a data race is mutation in the presence of aliasing, which Rust reliably rules out! It turns out that the same mechanism that makes our single-threaded programs memory safe, and that prevents us from invalidating iterators, also helps secure our multi-threaded code against data races. For example, notice how `read_files` sends a `String` to `filter_lines`. At run-time, only the pointer to the character data will actually be moved around (just like when a `String` is passed to a function with full ownership). However, `read_files` has to *give up* ownership of the string to perform `send`, to it is impossible for the string to still be borrowed. After it sent the string to the other side, `read_files` has no pointer into the string content anymore, and hence no way to race on the data with someone else.

There is a little more to this. Remember the `'static` bound we had to add to `register` in the previous parts, to make sure that the callbacks do not reference any pointers that might become invalid? This is just as crucial for spawning a thread: In general, that thread could last for much longer than the current stack frame. Thus, it must not use any pointers to data in that stack frame. This is achieved by requiring the `FnOnce` closure passed to `thread::spawn` to be valid for lifetime `'static`, as you can see in its documentation. This avoids another kind of data race, where the thread's access races with the callee deallocating its stack frame. It is only thanks to the concept of lifetimes that this can be expressed as part of the type of `spawn`.

Send

However, the story goes even further. I said above that `Arc` is a thread-safe version of `Rc`, which uses atomic operations to manipulate the reference count. It is thus crucial that we don't use `Rc` across multiple threads, or the reference count may become invalid. And indeed, if you replace `Arc` by `Rc` (and add the appropriate imports), Rust will tell you that something is wrong. That's great, of course, but how did it do that?

The answer is already hinted at in the error: It will say something about `Send`. You may have noticed that the closure in `thread::spawn` does not just have a `'static` bound, but also has to satisfy `Send`. `Send` is a trait, and just like `Copy`, it's just a marker - there are no functions provided by `Send`. What the trait says is that types which are `Send` can be safely sent to another thread without causing trouble. Of course, all the primitive data-types are `Send`. So is `Arc`, which is why Rust accepted our code. But `Rc` is not `Send`, and for a good reason! If we had two `Rc`s to the same data, and sent one of them to another thread, things could go havoc due to the lack of synchronization.

Now, `Send` as a trait is fairly special. It has a so-called *default implementation*. This means that *every type* implements `Send`, unless it opts out. Opting out is viral: If your type contains a type that opted out, then you don't have `Send`, either. So if the environment of your closure contains an `Rc`, it won't be `Send`, preventing it from causing trouble. If however every captured variable *is* `Send`, then so is the entire environment, and you are good.