Rust-101, Part 14: Slices, Arrays, External Dependencies

To complete rgrep, there are two pieces we still need to implement: Sorting, and taking the job options as argument to the program, rather than hard-coding them. Let's start with sorting.

Slices

Again, we first have to think about the type we want to give to our sorting function. We may be inclined to pass it a `Vec<T>`. Of course, sorting does not actually consume the argument, so we should make that a `&mut Vec<T>`. But there's a problem with that: If we want to implement some divide-and-conquer sorting algorithm (say, Quicksort), then we will have to *split* our argument at some point, and operate recursively on the two parts. But we can't split a `Vec`! We could now extend the function signature to also take some indices, marking the part of the vector we are supposed to sort, but that's all rather clumsy. Rust offers a nicer solution.

`[T]` is the type of an (unsized) *array*, with elements of type `T`. All this means is that there's a contiguous region of memory, where a bunch of `T` are stored. How many? We can't tell! This is an unsized type. Just like for trait objects, this means we can only operate on pointers to that type, and these pointers will carry the missing information - namely, the length (they will be *fat pointers*). Such a reference to an array is called a *slice*. As we will see, a slice can be split. Our function can thus take a mutable slice, and promise to sort all elements in there.

We decide that the element at 0 is our pivot, and then we move our cursors through the rest of the slice, making sure that everything on the left is no larger than the pivot, and everything on the right is no smaller.

```
pub fn sort<T: PartialOrd>(data: &mut [T]) {
    if data.len() < 2 { return; }

    let mut lpos = 1;
    let mut rpos = data.len();
    /* Invariant: pivot is data[0]; everything with index (0,lpos) is <= pivot;
       [rpos,len) is >= pivot; lpos < rpos */
    loop {
        unimplemented!()
    }

    data.swap(0, lpos-1);

    let (part1, part2) = data.split_at_mut(lpos);

    sort(&mut part1[..lpos-1]);
    sort(part2);
}

fn sort_nums(data: &mut Vec<i32>) {

    sort(&mut data[..]);
}

fn sort_array() {
    let mut array_of_data: [f64; 5] = [1.0, 3.4, 12.7, -9.12, 0.1];
    sort(&mut array_of_data);
}
```

Exercise 14.1: Complete this Quicksort loop. You can use `swap` on slices to swap two elements. Write a test function for `sort`.

Once our cursors met, we need to put the pivot in the right place.

Finally, we split our slice to sort the two halves. The nice part about slices is that splitting them is cheap: They are just a pointer to a start address, and a length. We can thus get two pointers, one at the beginning and one in the middle, and set the lengths appropriately such that they don't overlap. This is what `split_at_mut` does. Since the two slices don't overlap, there is no aliasing and we can have both of them as unique, mutable slices.

The index operation can not only be used to address certain elements, it can also be used for *slicing*: Giving a range of indices, and obtaining an appropriate part of the slice we started with. Here, we remove the last element from `part1`, which is the pivot. This makes sure both recursive calls work on strictly smaller slices.

Exercise 14.2: Since `String` implements `PartialEq`, you can now change the function `output_lines` in the previous part to call the sort function above. If you did exercise 13.1, you will have slightly more work. Make sure you sort by the matched line only, not by filename or line number!

Now, we can sort, e.g., an vector of numbers.

Vectors support slicing, just like slices do. Here, `..` denotes the full range, which means we want to slice the entire vector. It is then passed to the `sort` function, which doesn't even know that it is working on data inside a vector.

Arrays

An *array* in Rust is given by the type `[T; n]`, where `n` is some *fixed* number. So, `[f64; 10]` is an array of 10 floating-point numbers, all one right next to the other in memory. Arrays are sized, and hence can be used like any other type. But we can also borrow them as slices, e.g., to sort them.

External Dependencies

This leaves us with just one more piece to complete rgrep: Taking arguments from the command-line. We could now directly work on `std::env::args` to gain access to those arguments, and this would become a pretty boring lesson in string manipulation. Instead, I want to use this opportunity to show how easy it is to benefit from other people's work in your program.

For sure, we are not the first to equip a Rust program with support for command-line arguments. Someone must have written a library for the job, right? Indeed, someone has. Rust has a central repository of published libraries, called crates.io. It's a bit like PyPI or the Ruby Gems: Everybody can upload their code, and there's tooling for importing that code into your project. This tooling is provided by `cargo`, the tool we are already using to build this tutorial. (`cargo` also has support for *publishing* your crate on crates.io, I refer you to the documentation for more details.) In this case, we are going to use the `docopt` crate, which creates a parser for command-line arguments based on the usage string. External dependencies are declared in the `Cargo.toml` file.

I already prepared that file, but the declaration of the dependency is still commented out. So please open `Cargo.toml` of your workspace now, and enable the two commented-out lines. Then do `cargo build`. Cargo will now download the crate from crates.io, compile it, and link it to your program. In the future, you can do `cargo update` to make it download new versions of crates you depend on. Note that crates.io is only the default location for dependencies, you can also give it the URL of a git repository or some local path. All of this is explained in the Cargo Guide.

I disabled the following module (using a rather bad hack), because it only compiles if `docopt` is linked. Remove the attribute of the `rgrep` module to enable compilation.

```
#[cfg(feature = "disabled")]
pub mod rgrep {

    extern crate docopt;
    use self::docopt::Docopt;
    use part13::{run, Options, OutputMode};
    use std::process;

    static USAGE: &'static str = "
Usage: rgrep [-c] [-s] <pattern> <file>...

Options:
    -c, --count  Count number of matching lines (rather than printing them).
    -s, --sort   Sort the lines before printing.
";

    fn get_options() -> Options {

        let args = Docopt::new(USAGE).and_then(|d| d.parse()).unwrap_or_else(|e| e.exit());

        let count = args.get_bool("-c");
        let sort = args.get_bool("-s");
        let pattern = args.get_str("<pattern>");
        let files = args.get_vec("<file>");
        if count && sort {
            println!("Setting both '-c' and '-s' at the same time does not make any sense.");
            process::exit(1);
        }
        let mode = if count {
            OutputMode::Count
        } else if sort {
            OutputMode::SortAndPrint
        } else {
            OutputMode::Print
        };
        Options {
            files: files.iter().map(|file| file.to_string()).collect(),
            pattern: pattern.to_string(),
            output_mode: mode,
        }
    }

    pub fn main() {
        run(get_options());
    }
}
```

This function extracts the rgrep options from the command-line arguments.

This parses `argv` and exit the program with an error message if it fails. The code is taken from the `docopt` documentation.
The function `and_then` takes a closure from `T` to `Result<U, E>`, and uses it to transform a `Result<T, E>` to a `Result<U, E>`. This way, we can chain computations that only happen if the previous one succeeded (and the error type has to stay the same). In case you know about monads, this style of programming will be familiar to you. There's a similar function for `Option`. `unwrap_or_else` is a bit like `unwrap`, but rather than panicking in case of an `Err`, it calls the closure.

Now we can get all the values out.

We need to make the strings owned to construct the `Options` instance. If you check all the types carefully, you will notice that `pattern` above is of type `&str`. `str` is the type of a UTF-8 encoded string, that is, a bunch of bytes in memory (`[u8]`) that are valid according of UTF-8. `str` is unsized. `&str` stores the address of the character data, and their length. String literals like "this one" are of type `&'static str`: They point right to the constant section of the binary, so the reference is valid for the entire program. The bytes pointed to by `pattern`, on the other hand, are owned by someone else, and we call `to_string` on it to copy the string data into a buffer on the heap that we own.

Finally, we can call the `run` function from the previous part on the options extracted using `get_options`. Edit `main.rs` to call this function. You can now use `cargo run -- <pattern> <files>` to call your program, and see the argument parser and the threads we wrote previously in action!

Exercise 14.3: Wouldn't it be nice if rgrep supported regular expressions? There's already a crate that does all the parsing and matching on regular expression, it's called regex. Add this crate to the dependencies of your workspace, add an option ("-r") to switch the pattern to regular-expression mode, and change `filter_lines` to honor this option. The documentation of regex is available from its crates.io site. (You won't be able to use the `regex!` macro if you are on the stable or beta channel of Rust. But it wouldn't help for our use-case anyway.)