Arrays and Vectors

In this lab, you will learn how to do the following:

- Manually iterate the lines of a filehandle using `Iterator::next`
- `match` on combinations of possibilities using a tuple
- Use `std::cmp::Ordering` when comparing strings

Consult **"Chapter 10"** from the book "Command-Line Rust" (by Ken Youens-Clark) for more detailed information.

1. Creating a crate for the `comm` program

As with the previous labs, we will start by creating a project called `commr` for a Rust version of `comm`. Your *Cargo.toml* would be as follows:

```
[package]
name = "commr"
version = "0.1.0"
edition = "2021"

[dependencies]
clap = "2.33"

[dev-dependencies]
assert_cmd = "2"
predicates = "2"
rand = "0.8"
```

Then copy the *tests* directory provided for this lab into your project directory. You should get the new project with the following structure:

```
commr
├── Cargo.toml
├── src
│   ├── lib.rs
│   └── main.rs
└── tests
    ├── cli.rs
    ├── expected
    └── inputs
```

Make sure the program compiles and then run `cargo test` to run the tests, which should all fail.

Suppose you have a file containing a list of cities where your favorite band played on their last tour:

```
$ cd commr/tests/inputs
$ cat cities1.txt
Jackson
Denton
Cincinnati
Boston
Santa Fe
Tucson
```

Another file lists the cities on their current tour:

```
$ cat cities2.txt
San Francisco
Denver
Ypsilanti
Denton
Cincinnati
Boston
```

You can use `comm` to find which cities occur in both sets by suppressing columns 1 (the lines unique to the first file) and 2 (the lines unique to the second file) and only showing column 3 (the lines common to both files). This is like an *inner join* in SQL, where only data that occurs in both inputs is shown. Note that both files need to be sorted first:

```
$ comm -12 <(sort cities1.txt) <(sort cities2.txt)
Boston
Cincinnati
Denton
```

If you wanted the cities the band played only on the first tour, you could suppress columns 2 and 3:

```
$ comm -23 <(sort cities1.txt) <(sort cities2.txt)
Jackson
Santa Fe
Tucson
```

Finally, if you wanted the cities they played only on the second tour, you could suppress columns 1 and 3:

```
$ comm -13 <(sort cities1.txt) <(sort cities2.txt)
Denver
San Francisco
Ypsilanti
```

The first or second file can be `STDIN`, as denoted by a filename consisting of a dash (`-`):

```
$ sort cities2.txt | comm -12 <(sort cities1.txt) -
Boston
Cincinnati
Denton
```

As with the GNU `comm`, only one of the inputs may be a dash with the challenge program. Note that BSD `comm` can perform case-insensitive comparisons when the `-i` flag is present. For instance, we can put the first tour cities in lowercase:

```
$ cat cities1_lower.txt
jackson
denton
cincinnati
boston
santa Fe
tucson
```

and the second tour cities in uppercase:

```
$ cat cities2_upper.txt
SAN FRANCISCO
DENVER
YPSILANTI
DENTON
CINCINNATI
BOSTON
```

Then we can use the `-i` flag to find the cities in common:

```
$ comm -i -12 <(sort cities1_lower.txt) <(sort cities2_upper.txt)
boston
cincinnati
denton
```

On Unix based OS, you can use `man comm` command to read the manual page for `comm`. You can also use `comm --help` to get the brief usage information.

As usual, we will use the following structure for *src/main.rs*:

```
fn main() {
    if let Err(e) = commr::get_args().and_then(commr::run) {
        eprintln!("{}", e);
        std::process::exit(1);
    }
}
```

Following is how we would start *src/lib.rs*:

```
use clap::{App, Arg};
use std::error::Error;

type MyResult<T> = Result<T, Box<dyn Error>>;

#[derive(Debug)]
pub struct Config {
    file1: String, (1)
    file2: String, (2)
    show_col1: bool, (3)
    show_col2: bool, (4)
    show_col3: bool, (5)
    insensitive: bool, (6)
    delimiter: String, (7)
}
```

(1) The first input filename is a `String`.

(2) The second input filename is a `String`.

(3) A Boolean for whether or not to show the first column of output.

(4) A Boolean for whether or not to show the second column of output.

(5) A Boolean for whether or not to show the third column of output.

(6) A Boolean for whether or not to perform case-insensitive comparisons.

(7) The output column delimiter, which will default to a tab.

You can start your `get_args` by filling in the missing parts on the following skeleton:

```
pub fn get_args() -> MyResult<Config> {
    let matches
    =   App::new("commr")
            .version("0.1.0")
            .author("SE15 <se15@kmitl.ac.th>")
            .about("Rust comm")
            // What goes here?
            .get_matches();

    Ok(Config {
        file1: ...
        file2: ...
        show_col1: ...
        show_col2: ...
        show_col3: ...
        insensitive: ...
        delimiter: ...
    })
}
```

Start your `run` by printing the `config`:

```
pub fn run(config: Config) -> MyResult<()> {
    println!("{:#?}", &config);
    Ok(())
}
```

Following is the expected usage for the program:

```
$ cargo run -- --help
commr 0.1.0
SE15 <se15@kmitl.ac.th>
Rust comm

USAGE:
    commr [FLAGS] [OPTIONS] <FILE1> <FILE2>

FLAGS:
    -h, --help       Prints help information
    -i               Case-insensitive comparison of lines
    -1               Suppress printing of column 1
    -2               Suppress printing of column 2
    -3               Suppress printing of column 3
    -V, --version    Prints version information

OPTIONS:
    -d, --output-delimiter <DELIM>    Output delimiter [default:        ]

ARGS:
    <FILE1>    Input file 1
    <FILE2>    Input file 2
```

If you run your program with no arguments, it should fail with a message that the two file arguments are required:

```
$ cargo run
error: The following required arguments were not provided:
    <FILE1>
    <FILE2>

USAGE:
    commr <FILE1> <FILE2> --output-delimiter <DELIM>

For more information try --help
```

If you supply two positional arguments, you should get the following output:

```
$ cargo run -- tests/inputs/file1.txt tests/inputs/file2.txt
Config {
    file1: "tests/inputs/file1.txt", (1)
    file2: "tests/inputs/file2.txt",
    show_col1: true, (2)
    show_col2: true,
    show_col3: true,
    insensitive: false,
    delimiter: "\t",
}
```

(1) The two positional arguments are parsed into `file1` and `file2`.

(2) All the rest of the values use defaults, which are `true` for the Booleans and the tab character for the output delimiter.

Verify that you can set all the other arguments as well:

```
$ cargo run -- tests/inputs/file1.txt tests/inputs/file2.txt -123 -d , -i
Config {
    file1: "tests/inputs/file1.txt",
    file2: "tests/inputs/file2.txt",
    show_col1: false, (1)
    show_col2: false,
    show_col3: false,
    insensitive: true, (2)
    delimiter: ",", (3)
}
```

(1) The `-123` sets each of the *show* values to `false`.

(2) The `-i` sets `insensitive` to `true`.

(3) The `-d` option sets the output delimiter to a comma (`,`).

At this point, get your program to match the preceding output.

2. Completing the `comm` program

The next step is checking and opening the input files. Let's modify to the `open` function used in several previous exercises:

```
fn open(filename: &str) -> MyResult<Box<dyn BufRead>> {
    match filename {
        "-" => Ok(Box::new(BufReader::new(io::stdin()))),
        _ => Ok(Box::new(BufReader::new(
            File::open(filename)
                .map_err(|e| format!("{}: {}", filename, e))? (1)
        ))),
    }
}
```

(1) Incorporate the `filename` into the error message.

This will require you to expand your imports with the following:

```
use std::{
    error::Error,
    fs::File,
    io::{self, BufRead, BufReader},
};
```

As noted earlier, only one of the inputs is allowed to be a dash, for `STDIN`. You can use the following code for your run that will check the filenames and then open the files:

```
pub fn run(config: Config) -> MyResult<()> {
    let file1 = &config.file1;
    let file2 = &config.file2;

    if file1 == "-" && file2 == "-" { (1)
        return Err(From::from("Both input files cannot be STDIN (\"-\")"));
    }

    let _file1 = open(file1)?; (2)
    let _file2 = open(file2)?;
    println!("Opened {file1} and {file2}"); (3)

    Ok(())
}
```

(1) Check that both of the filenames are not a dash (`-`).

(2) Attempt to open the two input files.

(3) Print a message so you know what happened.

Your program should reject two `STDIN` arguments:

```
$ cargo run -- - -
Both input files cannot be STDIN ("-")
```

It should be able to print the following for two good input files:

```
$ cargo run -- tests/inputs/file1.txt tests/inputs/file2.txt
Opened tests/inputs/file1.txt and tests/inputs/file2.txt
```

It should reject a bad file for either argument, such as the nonexistent *blargh*:

```
$ cargo run -- tests/inputs/file1.txt blargh
blargh: No such file or directory (os error 2)
```

At this point, your program should pass all the tests for `cargo test dies` that check for missing or bad input arguments:

```
$ cargo test dies
running 4 tests
test dies_no_args ... ok
test dies_both_stdin ... ok
test dies_bad_file1 ... ok
test dies_bad_file2 ... ok
```

Your program can now validate all the arguments and open the input files, either of which may be `STDIN`. Next, you need to iterate over the lines from each file to compare them. The files in *commr/tests/inputs* that are used in the tests are:

- *empty.txt*: an empty file
- *blank.txt*: a file with one blank line
- *file1.txt*: a file with four lines of text
- *file2.txt*: a file with two lines of text

You may use `BufRead::lines` to read files as it is not necessary to preserve line endings. Start simply, perhaps using the *empty.txt* file and *file1.txt*. Try to get your program to reproduce the following output from `comm`:

```
$ cd tests/inputs
$ comm file1.txt empty.txt
a
b
c
d
```

Then reverse the argument order and ensure that you get the same output, but now in column 2, like this:

```
$ comm empty.txt file1.txt
        a
        b
        c
        d
```

The order of the lines shown in the following command is the expected output for the challenge program:

```
$ comm file1.txt file2.txt
        B
a
b
                c
d
```

Next, consider how you will handle the *blank.txt* file that contains a single blank line. In the following output, notice that the blank line is shown first, then the two lines from *file2.txt*:

```
$ comm tests/inputs/blank.txt tests/inputs/file2.txt

        B
        c
```

You should start by trying to read a line from each file. The documentation for `BufRead::lines` notes that it will return a `None` when it reaches the end of the file. Starting with the empty file as one of the arguments will force you to deal with having an uneven number of lines, where you will have to advance one of the filehandles while the other stays the same. Then, when you use two nonempty files, you'll have to consider how to read the files until you have matching lines and move them independently otherwise.

Use the test suite to guide you and try to do your best before looking for the solution in **"Chapter 10"** from the book "Command-Line Rust" (by Ken Youens-Clark).

3. Categorizing Points

In this exercise, we will randomly generate a list of points from customizable configuration and categorize each point in the list based on whether the point is located inside or outside a circle.

The following code show an example outline for a program that randomly generates a point list from the config specified as `RandConfig` and prints the location information for each point in the list:

```
fn main() {
    let mut rng = rand::thread_rng();
    let cfg = RandConfig{
        x_min: -1.5, x_max: 1.5,
        y_min: -1.5, y_max: 1.5
    };

    let c = Circle{center: Point{x: -0.1, y: -0.1}, radius: 0.8};
    let pt_list: Vec<_> = gen_point_list(&mut rng, &cfg, 50);
    let loc_list = locate_n_point(&c, &pt_list);
    for loc in loc_list {
        println!("{loc:?}");
    }

    // let (w, h) = (600, 600);    // SVG image size
    // let scale = (h as f64) / (cfg.y_max - cfg.y_min);
    // let pt_map = |pt: &Point| {
    //     let x = (pt.x - cfg.x_min) * scale;
    //     let y = (-pt.y - cfg.y_min) * scale;
    //     (x as i64, y as i64)    // map (x, y) to SVG output
    // };
}
```

The expected output should be similar to the following output:

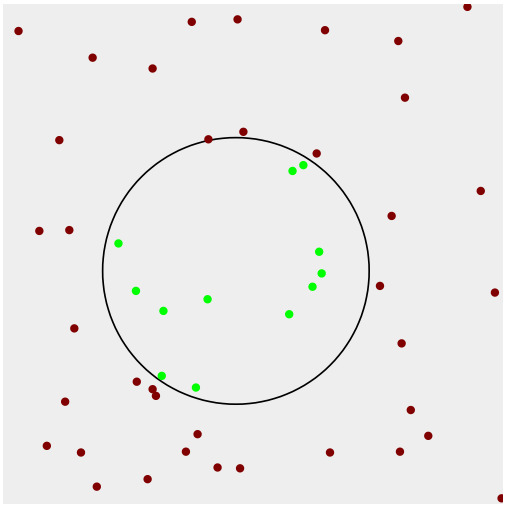
```
Outside((Point { x: -1.27, y: 1.01 }, 1.62))
Outside((Point { x: -1.35, y: -1.02 }, 1.56))
Outside((Point { x: 1.39, y: 0.05 }, 1.50))
Outside((Point { x: 1.41, y: -0.01 }, 1.51))
Outside((Point { x: -0.99, y: -1.08 }, 1.32))
Outside((Point { x: -0.53, y: -1.23 }, 1.21))
Inside((Point { x: 0.65, y: -0.13 }, 0.75))
Outside((Point { x: -0.88, y: 0.92 }, 1.29))
Outside((Point { x: -1.08, y: -0.87 }, 1.25))
Inside((Point { x: -0.61, y: 0.41 }, 0.73))
    ...
```

3.1) Write the function `gen_point_list(rng, cfg, n)` that randomly generates a point list of `n` points using the generator `rng` with each point having the range as specified in `cfg`.

3.2) Write the function `locate_n_point(c, pt_list)` that scan each point in a list `pt_list` and locate whether the point is within a circle `c` or is outside. The result should be a list of enum that indicates the location of each point along with the distance from the center of a circle `c` similar to the preceding example output. Add a unit test for the function to ensure the correctness.

3.3) Write a program that generates a point list, check whether each point in the list is inside or outside a circle, and use it to generate SVG image output. Use different fill color for the point inside and outside a circle and choose the comfortable size for `<circle>` radius that represents the point. The program should takes the **configuration** including random `x` range and `y` range for sampling the point, the number of points `n` in a generated point list, and a circle `((x, y), r)` for categorizing the point from the **command-line arguments**.

The expected SVG output should be similar to the following image:



4. Categorizing Points (Part II)

4.1) Write the function `locate_n_point2(b, pt_list)` based on the function in **3.2)** that takes two circles packed in a boundary `b` which is the structure `Bound { circle1, circle2 }` instead and locate whether each point in `pt_list` is within `Both` circles, on the `First` circle, on the `Second` circle, or `Outside` both circles.

4.2) Extend the program in **3.3)** by categorizing points using the function in **4.1)** and generate SVG output that visualizes a randomly generated point list. The program will **take two circles** and the **configuration** from the **command-line arguments**.

Homework Exercises

1. Write functions that flip lines of text

1.1) Write the function `vflip(img)` that takes a list of strings `img` representing ASCII image and **flip** (reverse) the contents of `img` **vertically**.

1.2) Write the function `hflip(img)` that takes a list of strings `img` representing ASCII image and **flip** (reverse) the contents of `img` **horizontally**.

The expected result from `vflip` and `hflip` should at least pass the following test:

```
#[test]
fn test_img_flip() {
    let emp = [""].to_string(); 0];
    assert_eq!(vflip(&emp), [""; 0]);
    assert_eq!(hflip(&emp), [""; 0]);

    let data = [
        "<--",
        "#####",
        "<=="
    ].map(|v| v.to_string());

    assert_eq!(
        vflip(&data), [
            "<==",
            "#####",
            "<--"
        ]);
    assert_eq!(
        hflip(&data), [
            "  --<",
            "#####",
            "  ==<"
        ]);
}
```

2. Write functions that concatenate lines of text

2.1) Write the function `vcat(img1, img2)` that takes a list of strings `img1` and `img2` representing two ASCII images and **concatenate** (join) the contents of `img1` and `img2` **vertically**.

2.2) Write the function `hcat(img1, img2)` that takes a list of strings `img1` and `img2` representing two ASCII images and **concatenate** (join) the contents of `img1` and `img2` **horizontally**.

The expected result from `vcat` and `hcat` should at least pass the following test:

```
#[test]
fn test_img_cat() {
    let emp = [""].to_string(); 0];
    assert_eq!(vcat(&emp, &emp), [""; 0]);
    assert_eq!(hcat(&emp, &emp), [""; 0]);

    let data = [
        "<--",
        "#####",
        "<=="
    ].map(|v| v.to_string());
    assert_eq!(vcat(&emp, &data), data);
    assert_eq!(vcat(&data, &emp), data);

    assert_eq!(
        vcat(&data, &data), [
            "<--",
            "#####",
            "<==",
            "<--",
            "#####",
            "<=="
        ]);

    assert_eq!(
        hcat(&data, &data[..2]), [
            "<--  <--",
            "##########",
            "<=="
        ]);
    assert_eq!(
        hcat(&data[..2], &data), [
            "<--  <--",
            "##########",
            "     <=="
        ]);
}
```