# Week 2 Exercises: Ownership and structs

Congratulations on making it to week 2! You rock!

## Purpose

This week's exercises will continue easing you into Rust. You'll get some practice with handling ownership/references and working with `Option` and `Result`, and you'll also get some light exposure to object-oriented programming in Rust! The primary exercise involves implementing a simple version of the `diff` utility to compare two files. Optionally, you can also implement the `wc` (word count) utility, or try out some graphics and implement Conway's Game of Life.

Let's build some practical tools, shall we?

**Due date:** Tuesday, April 21, 11:59pm (Pacific time)

*Ping us on Slack if you are having difficulty with this assignment. We would love to help clarify any misunderstandings, and we want you to sleep!*

## Getting the code

The starter code is available on GitHub here.

## Part 1: Ownership short-answer exercises

Let's warm up with some short-answer questions to get you thinking about ownership. For each of the following examples answer: Does this code compile? Explain why or why not. If it doesn't compile, what could you change to make it compile? (If you're not sure, you can run the compiler and try things out, but you need to provide a high-level English explanation of why it does/doesn't work.)

- Example 1:

```rust
fn main() {
    let mut s = String::from("hello");
    let ref1 = &s;
    let ref2 = &ref1;
    let ref3 = &ref2;
    s = String::from("goodbye");
    println!("{}", ref3.to_uppercase());
}
```

- Example 2:

```
fn drip_drop() -> &String {
    let s = String::from("hello world!");
    return &s;
}
```

- Example 3:

```
fn main() {
    let s1 = String::from("hello");
    let mut v = Vec::new();
    v.push(s1);
    let s2: String = v[0];
    println!("{}", s2);
}
```

Please provide your answers in ownership.txt.

## Part 2: rdiff

In this exercise, we will implement a simple version of the `diff` command-line utility to compare two files. (When you're finished, try using your tool to check your CS 110 assignments' output against the sample solution! It's pretty handy!)

Finding the diff between two files is a well-studied problem. Most implementations use Myers' algorithm, but in this exercise, we will have you compute the longest common subsequence between the two files and use that information to compute your diff. (LCS actually serves as the basis for Myers' algorithm!)

We have split this program into milestones in order to make it easier to implement and debug.

### Milestone 1: Reading the two files into vectors of lines

In this milestone, you should edit main.rs and implement the function `read_file_lines`, which takes a path to a file and returns a vector of lines in the file.

If you are starting this before Thursday's lecture, you may want to take a quick look at Tuesday's lecture notes to familiarize yourself with the ideas of `panic` and `Result`. This milestone will help you familiarize yourself with these concepts.

First, you'll need to open the file by calling `File::open(filename)`. `File::open` returns a `Result`, since opening the file may fail (e.g. if the filename is invalid).

You could open the file like this:

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

However, it's generally good style to avoid calling `unwrap` or `expect` unless the error absolutely should never occur, or unless you're writing the code in `main` or some other high-level function that won't be reused in your program. If you use `unwrap` or `expect` in helper functions, then code might use those helper functions without realizing they could cause panics.

The idiomatic way to deal with this is to write something like the following:

```
let file = File::open(filename)?;
```

The `?` operator is commonly used in Rust to propagate errors without having to repeatedly write a lot of code to check a `Result`. That line of code really expands to this:

```
let file = match File::open(filename) {
    Ok(file) => file,
    Err(err) => return Err(err),
};
```

If the `Result` was `Ok`, then the `?` gives you the returned value, but if the result was an `Err`, it causes your function to return that `Err` (propagating the error through the call chain).

Once you have your file open, you can read the lines like so:

```
for line in io::BufReader::new(file).lines() {
    let line_str = line?;
    // do something with line_str
}
```

In this code, `line` is a `Result<String, io::Error>`. We can safely unwrap its `Ok` value using `?`, as we did when opening the file. Then, you can add the string to a vector.

Finish implementing `read_file_lines`. Then, test your implementation by running `cargo test test_read_file_lines`. (The test definition is at the bottom of the file if you want to look at it.)

If you would like to print a vector for debugging purposes, you can do so as follows:

```
let v = vec![1, 2, 3];
println!("{:?}", v);
```

## Milestone 2: Implementing the Grid interface

To make this problem easier to solve, we are going to introduce a `Grid` struct, which internally stores data as a flat vector. In this milestone, you will implement two small methods on this struct.

Open `grid.rs` and have a look at the code that is already there. The `new` function initializes a vector with enough space to store `num_rows * num_cols` elements. The `display` function prints out the contents of the `Grid`, and the `clear` function zeroes all the elements.

Your job is to implement the `get` and `set` functions. `get` should return `Some(value)` if the provided location is within bounds, and `None` if it is out of bounds. `set` should return `Ok(())` (`()` is the "void type" in Rust, indicating "nothing to return") on success, or `Err(some error message)` if the provided location is out of bounds.

You'll need to come up some mapping from 2D `Grid` locations to 1D `Vec` locations. Accessing struct members uses syntax similar to Python; for example, to get the number of rows, you would write `self.num_rows`.

To test your code, run `cargo test test_grid -- --nocapture`.

## Milestone 3: Implementing Longest Common Subsequence

Longest Common Subsequence is a common algorithms problem: given two sequences, what is the longest *subsequence* that appears in both? In the character sequence "a b c d" and "a d b c", the longest common subsequence is "a b c", because those characters appear in order in both "a b c d " and "a d b c". (Note that a subsequence needn't be consecutive, but it does need to be in order.)

When diffing two files, we want to determine which lines were added or removed between them. To do this, we need to identify the lines that are common between both files. We can frame this as an LCS problem! We have two sequences of *lines*, and we want to find the longest subsequence of lines that appears in both files; those lines are the lines that were unmodified, and the other lines are those that were added or removed.

LCS is generally solved using dynamic programming. If you have taken CS 161, you may have seen this problem before. Wikipedia has an explanation here, and there is a decent YouTube video explaining the solution here. The solution involves filling out a `Grid` with subsequence lengths. However, you really don't need to understand how the algorithm works for our purposes, as long as you can implement the pseudocode below (adapted from Wikipedia).

```
let X and Y be sequences
let m be the length of X, and let n be the length of Y

C = grid(m+1, n+1)
// here, the .., like in Rust, is inclusive on the left bound and exclusive on
// the right bound
for i := 0..m+1
    C[i,0] = 0
for j := 0..n+1
    C[0,j] = 0
for i := 0..m
    for j := 0..n
        if X[i] = Y[j]
            C[i+1,j+1] := C[i,j] + 1
        else
            C[i+1,j+1] := max(C[i+1,j], C[i,j+1])

return C
```

You should implement this algorithm in `lcs` in `main.rs`. (Feel free to use better variable names in your code, as this is only pseudocode.) You can test your implementation by running `cargo test test_lcs -- --nocapture`.

## Milestone 4: Using LCS to construct the full diff

Time to put everything together! In the `main` function, you'll need to do the following:

- Call your `read_file_lines` function to read the contents of the two files. You'll want to call `expect` here with a suitable error message to handle the case where one of the supplied filenames is invalid.
- Call `lcs` to get an LCS `Grid`.
- Implement and call the following pseudocode (adapted from Wikipedia) to print the diff:

```
* let C be the grid computed by lcs()
* let X and Y be sequences
* i and j specify a location within C that you want to look at when reading out
  the diff. (This makes more sense if you understand the LCS algorithm, but
  it's not important.) When you call this function initially, just pass
  i=len(X) and j=len(Y).
function print_diff(C, X, Y, i, j)
    if i > 0 and j > 0 and X[i-1] = Y[j-1]
        print_diff(C, X, Y, i-1, j-1)
        print "  " + X[i-1]
    else if j > 0 and (i = 0 or C[i,j-1] ≥ C[i-1,j])
        print_diff(C, X, Y, i, j-1)
        print "> " + Y[j-1]
    else if i > 0 and (j = 0 or C[i,j-1] < C[i-1,j])
        print_diff(C, X, Y, i-1, j)
        print "< " + X[i-1]
    else
        print ""
```

Notice that Wikipedia’s solution is recursive. This is not optimal, since the compiler is not guaranteed to optimize this recursion away. This means you could quickly run out of stack space if the files are large. If you’re up for an extra challenge, see if you can figure out a way to convert this recursive defintion into an iterative one (although you’re certainly not required to do this).

You can run your program like so:

```
cargo run simple-a.txt simple-b.txt
```

You should get the following output:

```
  a
> added
  b
  c
> added
  d
> added
  e
```

You should also try on something slightly more complex:

```
cargo run handout-a.txt handout-b.txt
```

You should see this output:

```
> You'll be learning and practicing a lot of new Rust concepts this week by
< This week's exercises will continue easing you into Rust and will feature some
< components of object-oriented Rust that we're covering this week. You'll be
  writing some programs that have more sophisticated logic that what you saw last
  with last week's exercises. The first exercise is a warm-up: to implement the w
  command line utility in Rust. The second exercise is more challenging: to
  implement the diff utility. In order to do so, you'll first find the longest
> subsequence that is common.
< common subsequence (LCS) of lines between the two files and use this to inform
< how you display your diff.
```

It's fine if the `>` and `<` lines are printed in a different order, as long as the diff is valid.

Congratulations! You have a mini version of `diff` working in Rust!

## Optional: rwc

If you've made it this far and still have some time left over, give this next small program a try! It's smaller and easier than `rdiff`, but will give you more great practice.

Implement the most basic form of the `wc` command – given an input file, output the number of words, lines, and characters in the file. There is some minimal starter code in `rwc/src/main.rs`!

## Optional challenge: Conway's Game of Life

Haven't had enough Rust for the week? Well this exercise is for you!

Implement a Conway's Game of Life animation in Rust! Provide a way for the user to specify what cells are on at the start, to start/stop the animation, specify the speed of the animation, and run the animation for a certain number of steps. You might be interested in using this GUI library You should consider using your Grid type from the rdiff portion (we haven't talked about generics in Rust yet, but you can extend your implementation with generics if you feel inclined to look into them!).

This exercise will give you experience using other people's crates.

Be sure to run `git add .` before committing in order to make sure that any new files you've created get included in your submission.

## Part 3: Weekly survey

Please let us know how you're doing using this survey.

When you have submitted the survey, you should see a password. Put this code in `survey.txt` before submitting.

## Submitting your work

As with last week, you can commit your progress using `git`:

```
git commit -am "Type some title here to identify this snapshot!"
```

In order to submit your work, commit it, then run `git push`. This will upload your commits (snapshots) to Github, where we can access them. You can verify that your code is submitted by visiting https://github.com/cs110l/week2-yourSunetid and browsing the code there. You can `git push` as many times as you'd like.

## Grading

Part 1 (ownership exercises) will be worth 20%, each Part 2 (`rdiff`) milestone will be worth 15%, and Part 3 (survey) will be worth 20%. You'll earn the full credit for each part if we can see that you've made a good-faith effort to complete it.