CSC 411 Assignment:

Interfaces, Implementations, and Images

Interfaces and design for part A Friday, Sept 29th at 11:59PM. Full assignment (all of parts A, B, and C) due **Friday, Oct 6th** at 11:59PM.

Please read the entire assignment before starting work.

Purpose

This assignment has four goals:

1. To spur you to think more deeply about programming technique
2. To give you practice designing your own interfaces, not just using interfaces designed by other people
3. To give you practice thinking about what familiar algorithms and data structures you can use to solve new problems
4. To lay a foundation for future assignments. In these future assignments,
    - You will learn about *locality*, its effects on performance, and how to change the locality of a program.
    - You will understand how data structures in a high-level language map to machine structures, and how to improve space performance by programming directly with machine structures.
    - You will learn to improve the performance of programs by *code tuning*

The abstractions you build in this assignment will help you represent and manipulate digital images.

Preliminaries

- Be sure you are familiar with the `Vec` interface provided by the standard library: https://doc.rust-lang.org/std/vec/struct.Vec.html
- Rust By Example has a good chapter on Iterators that is worth reading: https://doc.rust-lang.org/rust-by-example/trait/iter.html

Part A: Two-Dimensional, Polymorphic Arrays

The Rust standard library provides the `Vec`, which implements polymorphic one-dimensional arrays. Now, a two-dimensional analog of a vector is usually referred to as a *matrix* but the term 'matrix' carries the connotation of containing *numbers* and being used for linear algebra. Since the purpose here will be to support images (2-dimensional collections of pixels), we will use the term *array*. For this part of the assignment, you'll adapt the vector abstraction to support *two*-dimensional arrays. Your adaptation will be called `Array2` and should be polymorphic, just as `Vec<T>` is polymorphic. Your adaptation should include the following changes when compared to a `Vec`:

- Instead of a *length*, an `Array2` will have a *width* and a *height*.
- Instead of being identified by a single index, an element of an `Array2` will be identified by *two* indices: the *column* or x index measures the distance between the element and the left edge of the array, while the *row* or y index measures the distance between the element and the top row of the array. Thus the top left element is always indexed by the pair *(0,0)*.
- You will want to support several kinds of iterators on your array. In particular, you must support ***row-major*** and ***column-major*** iteration. We suggest you name these functions `iter_row_major` and `iter_col_major`, respectively. Row-major refers to visiting every element of the first row in order by column, followed by every element of the second row, and so forth. Column-major refers to visiting every element of the first (left-most) column, followed by every element of the second column, and so forth.
- Recall that an `Iterator` implementation has a `next()` function, as well as an `Item` defined.
  - An `Item` need not just be the `T` element contained within your `array2`; it could be a data structure with more information about where that `T` lives in your `array2`, for instance.
  - A common "lightweight" data structure to use here is a `tuple`, whose elements can be accessed by index rather than name.

Hints:

- The key to this problem is to set up an implementation in which the elements of your two-dimensional array are in one-to-one correspondence with elements of one or more one-dimensional `Vec`s. The key question to answer is:

  > *How do you relate each element in a two-dimensional array to a corresponding element in some one-dimensional `Vec`?*

  If you have a precise answer to this question, the code is pretty easy to get right.

- Don't worry about performance; aim for simplicity. If you feel compelled to worry about performance, you may make simple code improvements provided you *justify* them. Don't try anything radical; premature optimization is the root of much evil.

- Think carefully about how the `next()` function for the `Iterator` implementation should behave.

  - Depending on your choice of underlying representation, one map function might be blindingly simple, while the other might be more work.

- Don't worry about the macros that can be used for literal vector construction. You might *use* those macros in your constructors, but you do not need to implement a macro to create an `Array2`.

- You will need some reasonable constructors:
  - Something like a `from_row_major` constructor might be useful; it might take the elements of an array in row-major order as the name implies.
  - You might also want a `from_col_major` constructor.
  - You might want a "blank slate" constructor that allows for a single value to be copied to each element. Note that since this is a polymorphic datatype, you don't know what this element will be. Having it be the integer value `0` does not make sense. Consider the following macro usage: `let data = vec![val; width * height];`
- You do **not** need to worry about providing element access using square bracket notation
  - you probably **do** want to provide a function that allows access to an element by a pair of coordinates
- You will want to create this project using cargo's lib mode:
  - `cargo new --lib array2`

`move` and Box:

You may need to use the `move` keyword to implement your iterators. This keyword is used when a function needs to take ownership of a variable that was originally defined outside of the scope of that function.

Let's work through an example:

```rust
pub fn is_too_big_builder(threshold: u32) -> Box<dyn Fn(u32) -> bool> {
    Box::new(move |value: u32| value > threshold)
}
```

At first glance, this seems like a complicated function. Let's break down the signature:

- The function `is_too_big_builder` takes a single argument, `threshold`.
- It returns a `Box` of a new function.
- That returned function takes a `u32` and returns a `bool`.
- The returned function lives on the heap, and we get a `Box`, i.e. a pointer, to it.

Let's move on to the body:

- `Box::new(...)` creates a new pointer to something living on the heap. Whatever lives on the heap has to be passed to `Box::new`.
- `|value: u32| value > threshold` is a closure, i.e. an anonymous function, that takes one `value` of type `u32` and returns whether or not `value` is greater than `threshold` (a boolean value).
- Note the `move` keyword:

  - `threshold` was originally defined in the signature for `is_too_big_builder`, i.e. before the closure was created.
  - The closure uses `threshold` directly, i.e. by value instead of by reference.
  - Therefore, the closure needs to *take ownership* of `threshold`.
  - In other words, the ownership of `threshold` needs to be `moved` from the builder to the closure.

Here's an example of the builder being used:

```
#[test]
fn test_builder() {
    let is_over_nine_thousand = is_too_big_builder(9_000);

    assert!(is_over_nine_thousand(12_000));  // True
    assert!(is_over_nine_thousand(6_000));   // False
}
```

How to construct an `Array2`

- Unlike in C, you cannot create a data structure with allocated but uninitialized elements
  - Ok, this is a bit of a lie
  - You can use `MaybeUninit` in unsafe Rust: https://doc.rust-lang.org/nomicon/unchecked-uninit.html
  - But this is too advanced for what we want to do right now
- At a minimum, you want a `from_row_major()` constructor that takes a vector of elements in row-major order, and a set of dimensions

There will undoubtedly be more hints along the way!

Part B: Using the `Array2` abstraction to identify Sudoku solutions

Write the test program `sudoku`. It takes as input a single portable graymap file, which may be named on the command line or may be given on standard input. Your program **must not print anything**, but if the graymap file represents a solved sudoku puzzle, your program should call exit(0); otherwise it should call exit(1). A solved sudoku puzzle is a nine-by-nine graymap with these properties:

- The maximum pixel intensity (aka the denominator for scaled integers) is nine.
- No pixel has zero intensity.
- In each row, no two pixels have the same intensity.
- In each column, no two pixels have the same intensity.
- When the nine-by-nine graymap is divided into nine three-by-three submaps (like a tic-tac-toe board), in each three-by-three submap, no two pixels have the same intensity.

Here's an example (which you can also view as an image in `/csc/411/images/sudoku/sudoku.pgm`):

```
P2
9 9
# portable graymap representing a sudoku solution
9
1 2 3   4 5 6   7 8 9
4 5 6   7 8 9   1 2 3
7 8 9   1 2 3   4 5 6

2 3 4   5 6 7   8 9 1
5 6 7   8 9 1   2 3 4
8 9 1   2 3 4   5 6 7

3 4 5   6 7 8   9 1 2
6 7 8   9 1 2   3 4 5
9 1 2   3 4 5   6 7 8
```

- In the above example, we have added spaces between three-by-three submaps purely for cosmetic reasons. The actual `.pgm` files given to you will not have such spaces.

If a user uses `sudoku` in a way that violates its specification, you should terminate with a checked run-time error (any will do). Read the specification carefully!

Note that for testing what return value your program exits with for a given input, you can examine the special shell variable `$?` which shows the return value of the last command. Thus, immediately after running sudoku, you can run:

```
echo $?
```

Just as in the Intro assignment, you can (and should) use the crate `csc411_image` which we wrote for you.

In particular, it provides a `read` function that takes an `Option<&str>` which is intended to be a filename. If the value of the `Option<&str>` is `None`, it will instead read from `stdin`. Thus, all you need to do is determine whether the command-line arguments include an argument (i.e. if `env::args()` has more than one element) and if so, pass that as an argument to `read()`; otherwise, pass `None` to `read()`.

In particular, a `GrayImage` looks like this:

```
pub struct GrayImage{
    pub pixels: Vec<Gray>,
    pub width: u32,
    pub height: u32,
    pub denominator: u16
}
```

The `pixels` are stored in row-major order, so should be easy to read into your `Array2`

- You'll want to create this project using cargo's "bin" mode:
  - `cargo new --bin sudoku`

Dependencies

Your `sudoku` program will need two dependencies: the `csc411_image` crate we provide, and your own `array2` crate.

Inside your `sudoku` project, your `Cargo.toml` should have something like this:

```
[dependencies]
csc411_image = "0.3.4"
array2 = { path = "../array2" }
```

Part C: Programming technique

Meet with your partner and identify *one* programming technique that meets *either* of these two criteria:

- One of you has incorporated it into your programming practice since the start of the semester.
- One of you would like to incorporate it into your programming practice by the end of this term.

Your assignment is to **describe the technique in enough detail that a student halfway through CSC 212 could put it into practice**. Use at most one page.

You should submit your description in a plain text file called `technique.txt`, or if you prefer, a PDF file called `technique.pdf`.

Expectations for your solutions

Your course instructor has thought of several ways to solve part A. *Representation is the essence of programming!* Your major design decision will be how to represent an `Array2`. Two obvious alternatives, both of which acceptable and each of which has variations, are

- To represent a `Array2` as a nested `Vec< Vec<T> >`.
- To represent a `Array2` as a single `Vec<T>`.

What is not acceptable is to clone and modify the Vec implementation or any existing 2D array implementation. Your new code should be a *client* of the standard library, and you should rely on `Vec` to do the heavy lifting.

> *If you're concerned about performance, don't worry—we'll make a careful study of it throughout the term, and you'll have a chance to revisit and improve your implementation.*

Common mistakes

Avoid these common mistakes:

- The indices into a two-dimensional array, whether they are called `x` and `y` or `i` and `j` or `r` and `c` or `row` and `column`, are always both integers. The only way to distinguish them is which one appears first. A common mistake is to use different orders in different parts of your code. Choose one order or the other and put it first **consistently** in all your code.
- This is a case where ***good names matter***

Organizing and submitting your solutions

Design

For the design document, submit a `lib.rs` and a design.pdf file which describes your design:

```
lib.rs
design.pdf
```

Your `lib.rs` does NOT need to compile, and we will not attempt to compile it. However, it should indicate the fields of your `Array2` struct, as well as the function signatures (not bodies) for the essential functions in your interface.

For example, suppose your `Array2` had just two fields, `width` and `height`, and there was a single function for creating an Array2 with specified dimensions (obviously, this would be an incomplete design!). Then your `lib.rs` might look like:

```rust
/// Elements contained must implement `Clone`.
#[derive(Debug, Clone, Eq, PartialEq)]
pub struct Array2<T: Clone> {
    width: usize,
    height: usize,
}

impl<T: Clone> Array2<T> {
    /// Creates a new `Array2`.
    ///
    /// # Arguments
    ///
    /// * `width`: the width of the `Array2`.
    /// * `height`: the height of the `Array2`.
    pub fn new(width: usize, height: usize,) -> Self {

    }
}
```

Your design file should contain a design checklists (see design-adt.pdf on EdStem) for the Array2 abstraction. **We are especially interested in items 1, 2, and 4** from the checklist. If you are confident of your design, *you can skip the other items.*

Upload your design to the iii-design assignment on Gradescope.

Implementation

Your best bet is to first create a `iii` directory with the `mkdir` command:

- `mkdir iii`

And then within that, create the two cargo projects:

- `cargo new --bin sudoku`
- `cargo new --lib array2`
- In your final submission, don't forget to include a README file which
  - Identifies you and your programming partner by name
  - Acknowledges help you may have received from or collaborative work you may have undertaken with classmates, programming partners, course staff, or others (ask me sometime about Otis the debugging dog)
  - Identifies what has been correctly implemented and what has not
  - Contains the critical parts of your design checklists
  - Says **approximately how many hours you have spent** completing the assignment
- Your submission should include at least these files:

```
iii/
  README.{md/txt/pdf}
  sudoku/
    Cargo.toml
    src/
      main.rs
  array2/
    Cargo.toml
    src/
      lib.rs
  technique.{txt/pdf}
```

  - You may have more `.rs` files in the `src` directories to help you better modularize your code. If you do have such files, you will need `use` statements referring to them in `lib.rs`/`main.rs`.

A reasonable zip command will be something like (while sitting in the parent directory of `iii`):

```
zip -r iii.zip iii -x "iii/*/Cargo.lock" "iii/*/target/**" "iii/*/.git/**"
```

Upload your zip archive to Gradescope under the assignment `iii`