CSC 411: Implementation of a Two-Dimensional, Polymorphic Arrays Design

for our assigntment

We want to implement a 2 dimensional array which is a structure,

- use a struct:
- struct Array2

We want the structure to polymorphic

- use a polymorphic struct
- struct Array2<T>

In our 2D structure

We want to use a single vector to represent our 2 dimensional structure

- use single dimensional vector of type T:
- arr: Vec<T>,

it should also contain a width, a height

- use tuple for them as a pair of type usize:
- width_height: (usize, usize)

We want to be able to keep track of a column index and a row index for an element that will change each time we call the next function on it in the implementation of our iterator

- use another tuple of usize elements
- col_row: (usize, usize),

We want a path vector to store the track after each next() call

- xyi_path: Vec<(usize, usize, usize)>,

For our 2D structure

We want to implement an iterator

- impl<T> Iterator for Array2<T>

For our iterator

We want to iterate over coordinates (x, y) with x the index of the column and y the index of the row

- define an item of type tuple with 2 elements, each of type usize
- type Item = (usize, usize);

We want to go from one coordinate to another until there is no more coordinate to advance

- we make a next function that will return an option. Either an element of the type of our item defined previously or None if there is no more coordinate of that type to advance to
- fn next(&mut self) -> Option<Self::Item>

For our next function

We want an algorithm to advance from a coordinate to another in our 2D array either in row major order or in column major order

- we want to save our current coordinate just in case we need it later for our algorithm so we initialize a current tuple of two usize elements and give it the value of our tracker col_row in our 2D (that tracker should start at (0,0) for the top left corner of our 2D array
    - let mut current = self.col_row;

By Yann Youbi
Professor Dr. Noah Daniels

- **_for row major order_**

- we want to go from one column to the other, and each time we are at the last column, go back to the first column down one row
    - we increment the column index by one, each time the column index is equal to the width minus one *(index of the last column as indexes start by 0 and the width is a count of the number of rows starting at 1 for the one column and not at 0 for one column)*
    - self.col_row.0 = (self.col_row.0 + 1) % self.width_height.0;
- we want to go down by one row each time we come back to the first column and similarly each time we reach the last row we go back to the first row
    - if self.col_row.0 == 0 {
        self.col_row.1 = (self.col_row.1 + 1) % self.width_height.1;}

(our coordinates never goes of of our 2D array dimensions)

- **_for column major order_** (*similar w/ save = row, row = column and column = save*)

- we want to go from one row to the other, and each time we are at the last row, go back to the first row down one column
    - we increment the row index by one, each time the row index is equal to the height minus one.
    - self.col_row.1 = (self.col_row.1 + 0) % self.width_height.1;
- we want to go down by one row each time we come back to the first column and similarly each time we reach the last row we go back to the first row
    - if self.col_row.1 == 0 {
        self.col_row.0 = (self.col_row.0 + 1) % self.width_height.0;}

our coordinates never goes of of our 2D array dimensions,)

- _for each major order

for our next() function

we want if the advanced coordinate is the first element of our 2D array (0,0) to return the option none (we *might want to give our tracker its original coordinates (width - 1, height -1)before)*

```
- if self.col_row.1 == 0 {
    // self.col_row = current;
    return None
  }
```

else we want you to return the option of some type which is the type of our current after taking the coordinates of our tracer at the end of our algorithm

- current = self.col_row;
- Some(current)

for our 2D structure

we want a function taking a mutable reference to a structure of type Array2 returning an iterator over a vector of tuples of usize elements (x, y, i):

with x a column index, y a row index and i is the one dimensional coordinate corresponding to the 2 coordinates x,y .

By Yann Youbi
Professor Dr. Noah Daniels

- fn iter_row_major(&mut self) -> impl Iterator <Item = &(usize, usize, usize)>
-

_for our iter_row_major function_ OR _for our iter_col_major function_

we want to call next function as many times as there are elements in our 2 dimensional array and store our tracker's coordinates col_row at each call inside of a vector of tuples with their 1d coordinates index

- for _i in 0.._len{
  let index = self.width_height.0 * self.col_row.1 + self.col_row.0;
  self.xyi_path.push((self.col_row.0, self.col_row.1, index));
  self.next();
  }
- we get the index by multiplying the width by the row index and adding the row index
  - let index = self.width_height.0 * self.col_row.1 + self.col_row.0;

we want to return an iterator over a vector of tuples of usize elements (x, y, i):

- we use the trait iter() to return an iterator over our vector xyi_path
- return self.xyi_path.iter();

We want a constructor to initialize and define a desired 2D structure with a desired vector, width and height

- impl<T> Array2<T> {
- fn new(v: Vec<T>, w: usize, h: usize) -> Self {
- Array2 {
- arr: v,
- width_height: (w, h),
- col_row: (0, 0),
- saved: (0,0),
- counter: 0,
-
- xyi_path: vec![],
- }
- }
- }

(we might want to make sure the given vector v lengths is equal to the given width multiplied by the giving height in our constructor)

Notice we our track, path and counter at an initial state

we want to make sure that initial state is maintained at the end or the beginning of each function implemented for our 2D structure

- we can set them at the beginning of each function manually or using a constructor taking a mutable reference to itself
- self.col_row = (0, 0);

We might want a constructor takin a value val of unknown type _ and making our 2D array with

- fn new(v: Vec<T>, w: usize, h: usize) -> Self {
  Array2 { arr: vec![val; width * height];

By Yann Youbi
Professor Dr. Noah Daniels

Tests

To test my program I would construct multiple 2D arrays of different width, height and vectors with different values and types then call my functions iter_row_major and/or iter_col_major on it use collect() on it with a print line and a macro {:?}

```
-       fn test_iter_col_major() {
-           let mut random = Array2::new(vec![1, 2, 3, 4, 5, 6, 7, 8], 4, 2);
-           println!("{:?}", random.iter_col_major().collect::<Vec<(usize, usize, usize)>>());
```

access an element

We might want to access an element in the our array based on an coordinates or the index itself

- *we get the index by multiplying the width by the row index and adding the row index*
- we access with something like
    - my_arr2.arr[index]
- *We might want to loop over our vector of tuple collected previously access the element each time using that macro*

We might use a next function implemented for row major to make our function trait returning an iterator in column major order

In such cases recursion might be needed as well.

- *We would use a for loop to call next based on the width and we might recurse to repeat that action based on the width with the initial coordinate advanced by one each time.*

By Yann Youbi
Professor Dr. Noah Daniels