TIMI DAYO-KAYODE
FAVOUR OKEREKE-MBA
02/19/2018
HW4 DESIGN CHECKLIST
COMP 40 SPRING 2018

PART A: UARRAY2B_T

- **What is the abstract thing you are trying to represent?**

  We are representing an unboxed polymorphic blocked 2D array. It will be capable of storing anything as long as the size of a single cell in a block of the 2D array is less than or equal to 64KB.

- **What functions will you offer, and what are the contracts those functions must meet?**
  - **UArray2b_new (int width, int height, int size, int blocksize):** This function takes in four integers; The first two represent the number of columns and rows of the 2D blocked array to be created, the third represents the size (in bytes) of a single cell, and the last represents the number of cells on one side of a block. It returns a pointer to the 2D block array.
  - **UArray2b_new64K_block(int width, int height, int size):** Similar to the UArray2b_new() function, the UArray2b_new64K_block() function creates a 2D blocked array and returns a pointer to it. It however determines the the block-size based on the size of a single cell in a block. It sets the blocksize to 1 if the total number of bytes in a block exceeds 64KB.
  - **UArray2b_free (T *array2b):** This function takes in a pointer to the address of the pointer that points to the 2D blocked array. It frees all allocated memory that is associated with the 2D array and returns nothing.
  - **UArray2b_width (T array2b):** Takes in a pointer to a 2D blocked array and returns the number of columns in the array.
  - **UArray2b_height (T array2b):** Takes in a pointer to a 2D blocked array and returns the number of rows in the array.
  - **UArray2b_size (T array2b):** Takes in a pointer to a 2D blocked array and returns the size of a single cell in the array.
  - **UArray2b_blocksize (T array2b):** Takes in a pointer to a 2D blocked array and returns the number of cells on one side of a block.
  - **UArray2b_at (T array2b, int column, int row):** Takes in a pointer to a 2D blocked array, and two integers that represent the column and row of a single cell in the array and returns a pointer to the element in that cell.
  - **UArray2b_map (T array2b, void apply(int col, int row, T array2b, void *elem, void *cl), void *cl):** This function takes in a pointer to a 2D blocked array and a function pointer as well as an optional pointer a parameter that will be passed into the function. It maps over the 2D array block wise and applies the function to each cell in a block.

    - **NOTE: IT IS A CHECKED RUN-TIME ERROR TO PASS A NULL T TO ANY FUNCTION IN THE UARRAY2B_T INTERFACE.**
- **What examples do you have of what the functions are supposed to do?**
    - UArray2b_T myArray = UArray2b_new(width, len, size, block_size) //len has value 10, and width has value 10, size has value 4, and block_size has value 4.
    - int *myVal = UArray2b_at(myArray, loc1, loc2) //loc1 has value 6 and loc2 has value 4
    - int mySize = UArray2b_size(myArray)
    - Check that sizeof(*myVal) == mySize
    - UArray2b_free(&myArray)

- **What representation will you use, and what invariants will it satisfy?**
    - The 2D blocked array will be represented as a UArray2_T 2D array of blocks, where each block will be represented as a single 1D UArray_T. The length of each UArray_T will be equivalent to the number of cells in the block it represents.
    - The invariants that our representation must satisfy are as follows:
        - The length of the 1D arrays must be equal to the number of cells in the block they represent.
        - The size of the data stored in each cell must equal the initial size specified by the user when the UArray2b_T was initialized.
        - Finally and most importantly, with row = i, col = j, and block size = A, the index (j, i) must be located at:
            - Index (j / A , i / A) in the 2D array
            - and index (A * (i % A)) + j % A)
- **How does an object in your representation correspond to an object in the world of ideas?**
    - As stated above, for any particular index (col,row) in the representation, it would be mapped to its location in our 2D blocked array using the functions:
        - Index (col / A , row / A) in the 2D array //this would map the location to the right block in our representation.
        - and index (A * (row % A)) + col % A), where A = blocksize. //this would then map to the right location WITHIN the block.
- **What test cases have you devised?**
    - Our first tests will involve inserting integer values into the UArray2b using the UArray2b_at function, and then printing out with both a nested for loop and our mapping function. The results should be the same.
    - Passing any non-positive number such as a value less than 1 to the UArray2b_new function should end the program with a Checked Runtime Error.
    - If the pointer passed to the UArray2b_free, UArray2b_width, UArray2b_height, or UArray2b_size function is NULL, or the address it points to is NULL, the program should end with a Checked Runtime Error.

    - We will populate a 2D blocked array and print out the values of each block on a newline using the map function to make sure the map function handles all the cells in one block before moving onto a different block.
    - We will try to access cells in blocks that may have unused cells. This should not be possible and should result in a checked runtime error.
- **What programming idioms will you need?**
    - The idiom for creating an abstract type using incomplete structures(Hanson Style)

PART C: PPMTRANS

- **Components And Their Interfaces:** The ppmtrans program has 4 stages to go through before completing the transformation specified by the user. Each stage will be treated as a different component of the program. The stages and their components are defined below:
    - **Command Line Arguments:** We will have a function to handle this stage. The function will go through the command line arguments deciding if the number of command line arguments provided fit the number required to run the program and from this decides what method to use and the transformation to perform.
    - **Reading Stage**: Using the interface defined in pnm.h we will read the image file into the 2D array of the type specified by the method. We will trust the pnm.h interface to throw exceptions in the case of a bad image format and expect it to return the valid 2D array type specified by the method. All this will be done within a function.
    - **Transformation Stage:** Using the the 2D array returned by the read function in the pnm interface, we will map through the array using the specified copy method and write this to a destination 2D array. This function will call other functions based on the angle of rotation specified by the user.
    - **Writing stage:** After the transformation is done, we will use the the write function in the pnm interface to write the rotated image to stdout. This will also be done within a function.

- **Architecture:** The general architecture of our program is not too complicated. We simply have a function to handle every stage (component) of the image transformation. The function that handles the command line arguments will decide what methods we are using and what rotation is being performed. With this information, the function handling the reading stage reads the provided image and returns a valid 2D array which will serve as a source image for the transformation function. The transformation function copies pixels from this source image, and then calls the required rotation function that places the pixels in the required position in the destination image as specified by the angle of rotation. This destination image is then passed to the function that handles writing to stdout.

- **Invariants:**
    - For the ppmtrans program to run correctly, we must be given a file that contains a valid image in ppm format for a rotation to some degree to be performed on.
    - For all rotations, for an image of original size w * h, for a 90 degree rotation, a pixel (x, y) is transposed to location (h - y - 1, x) in the rotated image.
    - For all rotations, for an image of original size w * h, for a 180 degree rotation, a pixel (x, y) is transposed to location (w - x - 1, h - y - 1) in the rotated image.

- **Testing Plan:**
    - We will test the ppmtrans program on non-ppm images;
        - The program should end with a checked runtime error.
    - We will test the functions on valid ppm image files
        - We know what the images look like before being rotated by x degrees and know what an x degree rotation of the image should look like, and so we will compare our expected outcome of an x degree rotation with the outcome of running our functions on these images to see if we have the desired outcome, and also check that the program then ends with an exit code of EXIT_SUCCESS.
    - We will also test the program on unimplemented options
        - The program should end with an error message written to stderr and a nonzero exit code.

- **Explanation of why it works:**
    - The program works because to rotate an image, all that is necessary is to change the location of its pixels. No matter the rotation of an image, it maintains the same number of pixels as its original orientation; no more, no less. With our invariants in place and satisfied, we ENSURE that no pixels are lost or gained in the process of transposition of the image, and only the locations of the images' pixels are changed.