Dan Glorioso & Brandon Dionisio (dglori02 & bdioni01)

Part C: Implementation Plan

ppmtrans

1. Implement the file opening aspect of the program. If no other recognizable flags are provided and there are more command line arguments to read, open the file (at the end of the if else command line loop in ppmtrans main function).
    a. Test to see if the program can open a valid file with helper open_or_die function.
    b. Test with an invalid file to see if the program provides a message on stderr and exits with code EXIT_FAILURE.
2. After the file is open, use the pnm reader to read in the file once we found that it is valid and store the data using the A2Methods_new function. The specific method used has already been set by previous flag arguments in the SET_METHODS macros at the top. If no method is provided, default to A2plain methods. This will store the inputted ppm image as an A2 array2.
    a. Verify the functionality on our own small, valid binary ppm formatted file.
    b. Pnm.h will deal with improperly formatted files and raising errors
    c. Test to see if the reader catches and exits the program with a nonzero exit code upon reading an incorrectly formatted file (using `echo$?` command).
    d. Test to see if the file can be read in with prior flag arguments. At this point, our test command arguments will not contain "-rotation", "-row/col/block major" or "-time" flags. These will be implemented below.
3. Create a print function that prints the values of the specified array. This will be used for testing purposes and for printing the final array.
    a. Test with a small file that the array was properly read in by calling the print function on the original file array.
4. Change the command line to accept the valid rotational commands ("-rotate 0", "-rotate 90", "-rotate 180"). This will set the "rotation" int at the start of main to the rotation value. Implementing the extra credit commands will come later on. For now, have the extra credit flags print an error to stderr and exit with a nonzero exit code.
    a. With each -rotation flag command, print to check to see if the rotation variable in ppmtrans.c gets set to the right integer value, only when the value is 0, 90, 180, or 270.
    b. Test to see if the rotation variable remains 0 if the rotation flag does not get called, or if it does get called and the values are not 0, 90, 180, or 270.
5. Implement the conditional statement for the time command ("-time <timing_file>") that saves the timing file name to char *time_file_name at the start of the command line loop.
    a. Test to see if time_file_name is no longer NULL upon calling the program with the "-time" flag and a filename by printing the value of this char pointer.
    b. Test to see if time_file_name remains NULL if the "-time" flag is not used by calling a program with no specified timing_file name and a print statement at the end of the loop for the char pointer holding the time_file_name.
6. After the command line loop is finished, check to see if a file is open. If a file is not open, assume we are using standard input and read in the ppm image from standard input.
7. At the end of the command loop after errors have been checked, all flags will be caught and appropriate variables (i.e. int rotation, char *time_file_name) have been saved. At

this point the file reading described at the top or the contents from standard input would be executed to be stored into an A2 array2, as specified in the SET_METHODS macro. This array will now be used to perform the appropriate transformations.

8. Before moving on, test to see if all commands that are expected to run to completion produce a zero exit code and that all commands that are expected to fail to run to completion produce a nonzero exit code.
9. Call a rotation helper function that completes the rotation commands (the methods for completing these commands are mentioned below). The function will take in an integer for the degrees of rotation and the appropriate-typed A2 array. Later, this function will take in the time file for time testing. If no rotation is specified from above, the rotation int will still be set to 0 and the rotation function will not perform a rotation on the array.
    a. Before implementing the functionality for reach rotation, print out a statement for each valid rotation degree to signify that we've reached the intended command. (e.g., row major + rotation 90, default + rotate 270, etc.). Using just the command line arguments, we will check that each is reachable.
    b. Test the edge cases such as when the program is run with 0 arguments (0° rotation), when just a method is provided, or when just a rotation command is provided.
    c. Next, call the intended apply functions that will be passed through the mapping function to facilitate the transformation. Although these functions will not be implemented yet, we can test to see if they were reached with print statements.
10. Finally, the program will print the transformed array image to standard output and afterward, will deallocate the A2 array2 storing the image.

Command Implementation

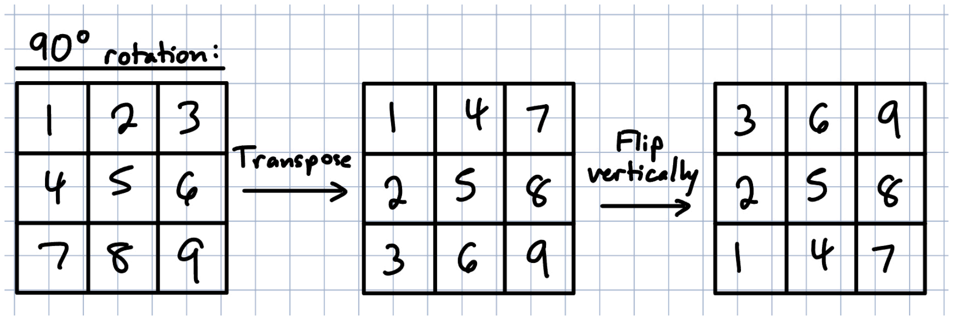
For our command implementation, we will first construct a function that will execute the valid rotational commands ("-rotate 0", "-rotate 90", "-rotate 180"). Using this single function to perform all rotations will increase modularity, as we will not need to create separate functions for each rotation degree. The parameters of this function are an int of the degrees of rotation and a pointer to the A2 array read in previously. Using if else, we will call mapping functions with apply functions for the necessary transformations to perform that rotation. The flip_horizontal, flip_vertical, and transpose functions are defined below afterward. The mapping method (row/col/block) has already been defined in the SET_METHODS macros, so map will call the appropriate mapping function.

- If the rotation is 90° (clockwise):
    - Since the array will need to be transposed, initialize a new A2 array of dimensions (height x width), which is swapped dimensions compared to the original array read in from the file.
        - Assert that the memory was properly allocated.
        - Test that the dimensions of the newly created array are correct by using print statements that print width and height. We will know the dimensions of the original array for reference.
    - First, the A2 Methods map function will be called with the parameters: the original A2 array, a function pointer to transpose as the apply function parameter, and the

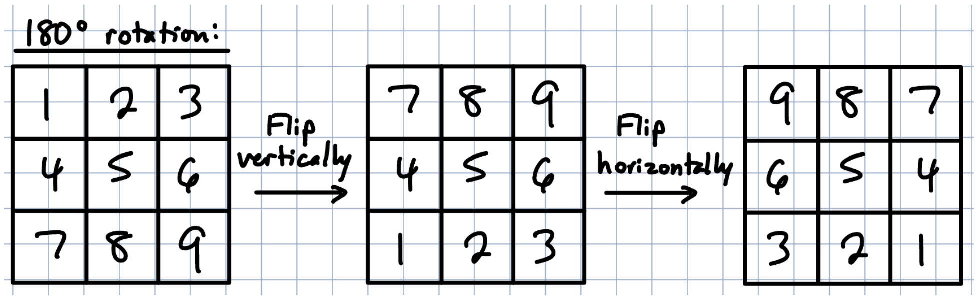
new initialized A2 (height x width) array as the closure pointer. This will return the newly transposed array as the closure pointer.

    - Test that all the passed in parameters are given as expected by printing out each of their contents using the print function declared earlier. We will start by using small testing files so that we know the expected output and can verify the transposition was as performed as intended.

  - Using the transposed array from the closure pointer, add and call the map function again with the closure pointer as the A2 array, the flip_vertical function as the apply function, and a NULL closure pointer. The flip_vertical function will flip the passed-in instance of the array.
    - Again, use the print function with a small file to verify the array was flipped properly.
  - Since a new array was created, deallocate the old original array and save the newly create array as the original array pointer
  - Run this function using the correct command line arguments with valgrind to ensure the old array was removed completely from memory



- If the rotation is 180°:
  - Using the A2 Methods, call map with the original array pointer, a function pointer to flip_vertical as the apply function, and no closure pointer.
    - Use the print function with the original array to confirm the elements were properly swapped for a vertical flip.
  - Then, call the A2 Methods map function again with a pointer to the original array (which has already been flipped vertically), a function pointer to flip_horizontal as the apply function, and no closure pointer.
    - Use the print function with the original array to confirm the elements were properly swapped for a horizontal flip.



- If the rotation is 270° (or 90° rotation counterclockwise):

        - Since the array will need to be transposed, initialize a new A2 array of dimensions (height x width), which is swapped dimensions compared to the original array read in from the file.
            - Verify the memory was properly allocated
        - First, the A2 Methods map function will be called with the parameters: the original A2 array, a function pointer to transpose as the apply function parameter, and the new initialized A2 (height x width) array as the closure pointer. This will return the newly transposed array as the closure pointer.
            - Use similar testing techniques described above to verify the transformation
        - Using the transposed array from the closure pointer, call the map function again with the closure pointer as the A2 array, the flip_horizontal function as the apply function, and a NULL closure pointer. The flip_horizontal function will flip the passed-in instance of the array.
- Else, since the rotation has already been check in the command line loop that it is valid, the degree rotation is 0°:
    - Do not perform any transformation functions on the array. The final array will be the same as the original array.
        - Use the print function to verify this occurs properly
- Later, this function will be modified to take the filename as a parameter and start the time at the beginning of the function and stop at the end of the function and save the calculated time divided by the number of cells in the array to the specified time file.

To execute each rotation, we will create three apply functions that can be used together to perform all rotation operations: transpose, flip vertically, and flip horizontally. This will increase modularity as the three functions can be used to perform the rotation commands, as well as the flip horizontal, flip vertical, and transpose flags.

- Transpose(int row, int col, T array2, void *elem, void *cl):
    - Before this function is called, a new array is always initialized of swapped dimensions of the original array since this function changes the dimensions of the final array. This newly allocated array is passed through as the closure variable.
    - The original instance of the array is passed as A2 array2, and using the A2 method mapping, the element of index (row, col) of the original array becomes index (col, row) of the newly initialized array.
        - Test by calling a mapping function with this apply function on its own to see if the A2 array2 gets transposed correctly.
        - Run valgrind to check if the memory associated with the original array gets freed properly.
- Flip_vertical(int row, int col, T array2, void *elem, void *cl):
    - This function expects cl to be passed in as NULL and does not use cl or elem. The function takes all elements of which i is less than or equal to (length/2) and swaps elements (row, col) with (length - row, col) in array2 using a temporary variable.
        - Test by calling a mapping function with this apply function on its own to see if the A2 array2 gets flipped vertically correctly.

- Flip_horizontal(int row, int col, T array2, void *elem, void *cl):
    - This function expects cl to be passed in as NULL and does not use cl or elem. The function takes all elements of which j is less than or equal to (width/2) and swaps elements (row, col) with (row, width - col) in array2 using a temporary variable.
        - Test by calling a mapping function with this apply function on its own to see if the A2 array2 gets flipped horizontally correctly.

<u>Part D (experimental): Analyze locality and predict performance</u>

	Row-major access (UArray2)	Column-major access (UArray2)	Blocked access (UArray2b)
90-degree rotation	5	5	4
180-degree rotation	2	2	1

180-degrees blocked: This operation is predicted to have the best hit rate due to its locality, lack of variable reads, and temp variable usage. Our implementation for this operation involves flipping an A2 array2 horizontally and then vertically. In both of these transformations, we are maintaining the same dimensions for the array2 (within the same array) and only parsing through half of the array to swap with the other half (e.g., (row, col) ↔ (row, width - col) for horizontal and (row, col) ↔ (length - row, col) for vertical). These swaps involve a temporary swapping variable to hold one of the values being swapped which will always be hit in the cache. Additionally, the implementation for this operation involves parsing through a blocked array which will have high locality for accessing elements; this, in turn, increases the likelihood that data accessed by a program will already be present in the cache (increasing hit rate) and will run the fastest.

180-degrees row-major & 180-degrees column-major: These operations are predicted to have the second best hit rate due to the lack of variable reads and temp variables used, but lack of locality. Our implementation for these operations involve flipping A2 array2s horizontally and then vertically. In both of these transformations, we are maintaining the same dimensions for the array2s and only parsing through half of the array to swap with the other half (e.g., (row, col) ↔ (row, width - col) for horizontal and (row, col) ↔ (length - row, col) for vertical). These swaps involve a temporary swapping variable to hold one of the values being swapped which will always be hit in the cache. The implementation for this operation involves parsing through an unblocked array, so through accessing elements in the plain array2, there is a greater chance that the data will not already be present in the cache (decreasing hit rate). We believe that the hit rate of this operation is better than that of the **90-degrees blocked** because having to perform a transpose outweighs the loss of the hit rate than having low locality.

90-degree blocked: This operation is predicted to have the fourth best hit rate due to its locality, high concentration of variable reads, and no temp variable usage. Our implementation for this operation involves taking the transpose of an A2 array2 and then flipping it vertically. In the transpose transformation, we are initializing an entirely new array2 with the opposite dimensions as the original array and then copying over all the inverse indices into that array. Although this transformation has good locality due to the blocked implementation, we have to allocate more memory and access twice as many non-repeating variables compared to a flip. This, in turn, decreases the likelihood that data accessed by a program will already be present in the cache (decreasing hit rate).

90-degree row-major & 90-degree column-major: These operations are predicted to have the worst hit rate due to their lack of locality, high concentration of variable reads, and no temp variable usage. Our implementations for these operations involve taking the transpose of A2 array2s and then flipping them vertically. In the transpose transformation, we are initializing an entirely new array2 with the opposite dimensions as the original array and then copying over all the inverse indices into that array. We have to allocate more memory and access twice as many non-repeating variables compared to a flip. In doing so, we do not utilize a temporary variable as we are just copying every single element of one array2 into a different array2. On top of the bad hit rate due to the transpose transformation, this operation also lacks locality as we are parsing through an unblocked implementation of the array2. This, in turn, decreases the likelihood that data accessed by a program will already be present in the cache (decreasing hit rate.