Amoses Holton

Katie Hoskins

Assignment 2 Design Doc

Part A

1.**[*]** What is the abstract thing you are trying to represent? Often the answer will be in terms of sets, sequences, and finite maps.

We are trying to represent a two dimensional array by using a one-dimensional array implementation.

2.What functions will you offer, and what are the contracts of that those functions must meet?

We will offer the following functions:

- extern T UArray2_new(int height, int width, int size)
  - This function will create a new two dimensional array for the user with dimensions: length x width. Each index of the array will be able to store elements of size “size”.
- extern void UArray2_free(UArray2_T *array)
  - This function will free the memory used by the 2D array given as an argument.
- extern void * UArray2_at(UArray2_T array, int col, int row)
  - This function takes a 2D array and the coordinates of a desired element [row, col]. It returns a pointer to the element at that location.
- extern int UArray2_width(UArray2_T array)
  - This function returns the width of the 2D array.
- extern int UArray2_height(UArray2_T array)
  - This function returns the length of the 2D array.
- Extern int UArray2_size(UArray2_T array)
  - This function returns the size of the elements stored in the array.
- extern void UArray2_map_row_major(UArray2_T array, void *apply(int col, int row, UArray2_T array, void *element, void *cl), void *cl)
  - Function traverses the array row by row, applying apply function for each element.
- extern void UArray2_map_col_major(UArray2_T array, void *apply(int col, int row, UArray2_T array, void *element, void *cl), void *cl)
  - Function traverses each column of the array, applying apply function to each element.

3.What examples do you have of what the functions are supposed to do?

- If a user wanted to map a 8x8 chess that stores a 4-bit chess piece, they would create the board by calling UArray2_New(12, 12, 1).
- After the checker game the user will call UArray2_free to free all memory used by the board.
- If the user wanted to know what piece was in front of their queen at position [row,col], they would call UArray2_at(array, row+1, col).
- If the user wanted to know the width of the board they would call UArray2_width(array). Same goes for if they wanted to find the length using UArray2_length.
- If the user wanted to print the current board to the screen, they would use UArray2_map_row_major(array, &apply, &cl) where apply is a function that prints the current piece.
- If the player wanted to count the number of pieces at any point, they would use Uarray_map_col_major(array, &apply, &cl), where apply is what?

4.What representation will you use, and what invariants will it satisfy? (This question is especially important to answer precisely.)

We will use a single one-dimensional array built using the UArray interface provided to represent the 2D array. Since we are taking this approach, it should be true that for any element at [x, y] in the 2D array, the index i in the 1D array where the element is actually stored will be

$$i = (x * width) + y$$

Lastly, an obvious invariant would be that at any given element at location [col, row] the previous element's row value will always be less than or equal to row.

5.When a representation satisfies all invariants, what abstract thing from step [<-] does it represent? That is, what is the inverse mapping from any particular representation to the abstraction that it represents. This question is also especially important to answer precisely.)

When the one-dimensional array satisfies all invariants listed above, it should represent a 2D array.

6.What test cases have you devised?

- Use provided useuarray2.c program to test that uarray2_new creates a 2D array with the proper dimensions and element size.
- We'll also use that program to test the UArray2_height, UArray2_width and UArray2_size to make sure they return the correct information of the 2D array.
- This program will also be used to test the UArray2_map_row_major and UArray2_map_col_major functions with a simple apply function that just examines the elements.
- We will also read and write a graymap using the UArray2_map_col_major to read it and the UArray2_map_row_major to write it.
    - Doing so should transpose the image.
- We will run valgrind tests to confirm that all memory is freed.
- We will try to create an array with negative dimensions to make sure CRE's are used correctly.
- We will also try to access an out of bounds element to make sure this also raises a CRE.

7.What programming idioms will you need?

- We will be getting rid of "unused variable" warnings using void in order to test the apply function in UArray2_map_row_major and UArray2_map_col_major
- We will be using the Ramsey approach to accessing elements in unboxed arrays, initializing array elements and storing values into an unboxed array.
- We will also be allocating memory using a single point of truth.

Part B

1. **[*]** What is the abstract thing you are trying to represent? Often the answer will be in terms of sets, sequences, and finite maps.

We are trying to represent a black and white image as a two dimensional array of bits.

2.What functions will you offer, and what are the contracts of that those functions must meet?

- void Bit2_free(Bit2_T *array)
  - Clears the memory used in the array of bits.
- void *Bit2_new(int width, int height)
  - Creates a new 2D bit array of the given length and width.
- int Bit_get(Bit2_T array, int col, int row)
  - Returns the value of the bit located at [row, col] in the 2D array.
- int Bit2_put(Bit2_T array, int col, int row, int bit)
  - Places “bit” in position [row, col] of the 2D array.
- int Bit2_width(Bit2_T array)
  - Returns the width of the 2D bit array.
- int Bit2_height(Bit2_T array)
  - Returns the height of the 2D bit array.
- void Bit2_map_row_major(Bit2_T array, void *apply(int col, int row, Bit2_T array, void *element, void *cl), void *cl)
  - Function traverses the array row by row, applying apply function for each element.
- extern void Bit2_map_col_major(Bit2_T array, void *apply(int col, int row, Bit2_T array, void *element, void *cl), void *cl)
  - Function traverses each column of the array, applying apply function to each element.

3.What examples do you have of what the functions are supposed to do?

- If someone wanted to store a black and white image of a tree, they would call Bit2_new(width, width) using the width and height of the image desired.
- In order to get read all of the pixels in the image, the person would call Bit2_map_col_major(array, read, cl) using a simple read function to read the next bit from the image and store it in the array.

- If the person wanted to transpose the image, they would read the image using Bit2_map_col_major(array, read, cl) and write the image using Bit2_map_row_major(array, print, cl).
- If the person wanted to tell if a particular bit is black or white they would use Bit2_get(array, col, row) where [col, row] is the location of that bit in 2D array. They could also replace this bit in the image by using Bit2_put(array, col, row, bit) where [col, row] is the location and bit is what's being placed at that location.

4.What representation will you use, and what invariants will it satisfy? (This question is especially important to answer precisely.)

We will use a single one-dimensional array, built using the UArray interface provided to build a 2D array of bits. Since we are using a single UArray to represent a 2D array, there is an invariant that should be true about the index of any element i, given location [x, y]. The index of i should be

$$i = (x * width) + y$$

In addition, for any given element at location [col, row] the previous element's location row will always be less than or equal to row.

5.When a representation satisfies all invariants, what abstract thing from step [<-] does it represent? That is, what is the inverse mapping from any particular representation to the abstraction that it represents. This question is also especially important to answer precisely.)

When the one-dimensional array of bits satisfies the above invariants then it should successfully represent a 2D array.

6.What test cases have you devised?

- Using the provided program usebit2.c, we will use it to test if it can create a new array of bits successfully.
- We will also use the program mentioned above to test that Bit2_height and Bit2_width return the correct numbers.

- We will create a 5x5 bit array of all zeros
    - Using that array we will use Bit2_map_row_major to change all elements to “1”.
    - Using the same array, we would use Bit2_map_col_major to check that all elements are “1”
    - We would then use Bit2_put to place 0's into every other row of the array.
    - Using Bit2_map_row_major and an apply function that prints the current element, we will be able to verify that we are correctly traversing by row if the test prints 5 “1's” followed by 5 “0's”.
    - Using Bit2_map_col_major and the same apply function to print the current element, we will be able to verify that we are correctly traversing by column if it alternates printing 1's and 0's.
- We would also continuously run valgrind tests to confirm that memory is freed at the end of use.

7.What programming idioms will you need?

We will be using three main programming idioms. We will be getting rid of “unused variables” when testing by voiding those variables. In addition, when allocating memory, we will only use a single point of truth. Lastly, when handling unboxed arrays, we will make sure to use Ramsey's approach of accessing elements, initializing the elements and storing values.