https://prod.liveshare.vsengsaas.visualstudio.com/join?C644236A8C2BF7AE4AB79EA8F81F4E628F45

Implementation Plan

Note: The numbers listed next to each step are the order in which we plan to implement/test each function (i.e. readPPMFile and WritePPMfile are both “step 1”)

Note: We believe that we have thought through our implementation and testing plan thoroughly but as we begin coding we expect to run it new test cases and intend to implement them in our unit_test. We also intend to test with Valgrind (-leak-check=full and -s) to ensure that all blocks are freed and to catch any errors we encounter in coding.

Note: We also intend to implement each end of compression and decompression hand-in-hand so we are able to thoroughly test each step before moving on to the next (as per the spec)

Compression		**Decompression**
1. ReadPPMFile input: - ppm image file purpose: ● Reads in ppm image file into a pixmap using pnm_ppmread testing: ● After implementation we will read a file and use the methods suite to then print it out and ensure that the image was read in correctly information loss: - we assume no information will be lost at this point as the PNM reader was implemented for us output: - pnm_ppm pixmap (unsigned r/g/b values)	**Joint Testing:** - pass a ppm image file to the compression func - pass the output of the compression func to the decompression function - ppmdiff the original image file with the output from the decompression func	**1. WritePPMFile** output: - ppm image file purpose: ● Use pnm_ppmwrite to write the input pixmap to stdout testing: ● After implementation we will create an instance of a Pnm_ppm pixmap (unrelated to our compression function to ensure that we are solely testing the write portion) in our unit_test file information loss: ● we assume no information will be lost in this step input: ● pnm_ppm pixmap (unsigned r/g/b)
2. ConvertPixels (US → FP) input: - pnm_ppm pixmap (unsigned r/g/b values) purpose: - convert pixels from unsigned ints to	**Joint Testing:** - pass an unsigned pixmap to the compression function - pass the output from the compression function to the	**2. ConvertPixels (FP → US)** output: - pnm_ppm pixmap (unsigned r/g/b values) purpose: ● builds an instance of a pnm_ppm pixmap,

floating point values
● creates an instance of a new **blocked 2D array** (with trimmed dimensions if needed and block size 2).
● uses an apply function to populate the new array with the input array's pixel values, casted from unsigned ints to floating points
● In using an apply function we are able to pull out values and apply our casting and conversion
● since the new 2D array will hold floats instead of unsigned ints, we will outline a new struct for each element in the array:
typedef struct **RGB_float** {
Float red
Float green
Float blue
}
information loss:
● We assume some information is lost because floating point is less precise when compared to unsigned
testing:
● Pass a pixmap with unsigned pixels and check the size of the (r,g,b) values of each pixels and assert that the are all size of a floating point (32 bits)
output:
- 2D blocked array with floating point (r/g/b) pixels
	decompression func
	- ppmdiff the input of the compression func with the output of the decompression func (also an unsigned pixmap)
		then populates the 2D array that will be assigned to it
		● creates an instance of a 2D array with the same dimensions as the input array. each pixel will contain an instance of the Pnm_rgb struct
		● uses an apply function to convert each pixel's RGB values from floating point to unsigned, and stores each converted pixel in the new 2D array
		information loss:
		- no information is lost in this step specifically
		input:
		- 2D blocked array with floating point (r/g/b) pixels
		
		testing:
		- pass the function a pixmap with floating point pixels and ensure that the size of the (r,g,b) values within each pixel is the size of an unsigned int (9 bits)

3. ConvertColorSpace (RGB→CV) input:	**Joint Testing:** - pass the compression	**3. ConvertColorSpace (CV→RGB)** output:

- 2D blocked array with floating point (r/g/b) pixel pixels
purpose:
- convert pixels from RBG colourspace to Component Video colourspace
● create a new **blocked** 2d array with the same dimensions as the input 2D array.
● use an apply function to convert the RGB values in each pixel to Y/Pb/Pr values, which we will store in the following struct and copy over to the new 2D array:
typedef struct **CV_Float** {
float y, pb, pr
}
information loss:
- we assume no information is lost in this step, as it is simply a unit conversion, not of data type
testing:
● we will go through and make sure the dimensions are the same and all the pixel got copied over in the apply function
● we will write an assertion function in the unit test to look at each copied pixel and see if the values at each point in the new 2d **blocked** array are y, pb, pr and not r, g, b
output:
- 2D blocked array with floating point (y/Pb/Pr) pixels
	func a 2D blocked array with RGB_float structs at each pixel
	- pass the output of the compression func (2D array with CV_float structs) to the decompression func
	- compare the input to the compression func with the output of the decompression func (using mapping funcs to print out the values)
		- 2D blocked array with floating point (r/g/b) pixel values
		purpose:
		- convert pixels from component video color space (Y, Pb, Pr) to RGB color space
		- build a new 2D array where each pixel is represented by floating point R/G/B values, which will be contained within the structure:
		typedef **struct RGB_float** {
		float r, g, b
		}
		- use an apply function to perform conversions from Y/Pb/Pr (input) to R/G/B (output) and populate the new 2D array with the converted pixels
		information loss:
		- no information lost in this step
		
		testing:
		● ensure the dimensions are preserved (none of the pixels were lost/not copied over in the apply func)
		● we will write an assertion function in the unit test to look at each copied pixel and see if the values at each point in the new 2d array are r, g, b and not y, pb, pr
		input:
		- 2D blocked array with floating point (y/Pb/Pr) pixels

4. shrinkToTransform (4 pixels → 1 pixel)	**Joint Testing:**	**4. growToTransform (1 pixel → 4 pixels)**

input:
- 2D blocked array with floating point (y/Pb/Pr) pixels
purpose:
- Using a for loop we will go through the input array and pull information out of each **4-pixel block** in the array (since we declared block size of 2) and use provided functions to compute the float values of **pb, pr, a, b, c,** and **d** for one compressed pixel
● We will store the six float values (calculated from the 4 pixels) in a struct and then place a collection of these structs in a (plain or blocked) 2D array
typedef struct **PbPrabcd_float** {
float pb, pr, a, b, c, d
}
information loss:
- information will be lost in this step, as we are compressing the data from four pixels into just one pixel
testing:
● We will go through and make sure that our array is getting populated with values by using the at function to go to each spot in memory and asserting that something has been allocated there
● ensure that we are grouping and
	- pass the compression func a 2D blocked array with YPbPr_float structs at each pixel
	- pass the output of the compression func (2D array with PbPrabcd_float structs) to the decompression func
	- compare the input to the compression func with the output of the decompression func (using mapping funcs to print out the values)
		output:
		- 2D blocked array of floating point y/Pb/Pr pixels
		purpose:
		- create a new 2D blocked array with block size 2 and four times the number of elements as the input array
		- read through the 2D array of Pb/Pr/a/b/c/d pixels and use inverse functions to **reproduce** the Y/Pb/Pr values for the **four pixels** that were compressed to create this pixel
		- for every pixel in the input array, produce and populate four pixels in the output array, each of which will contain the struct:
		typedef struct **CV_Float** {
		float y, pb, pr
		}
		information loss:
		- no information is lost in this step, although the four restored pixels will be less precise than the original image
		testing:
		- ensure that the output array has 4 times as many pixels as the input array
		- ensure that all pixels from the input array are being “returned” to the correct spot (the proper 2x2 block)
		- run our decompression

compressing pixels in 2x2 blocks ● ensure the output array has ¼ the number of elements as the input array output: - 2D plain array of compressed pixels, each containing floating point (pb/pr/a/b/c/d) values		function on a small 2D array of compressed pixels, with easy/simple numbers input: - 2D plain array of compressed pixels, each containing floating point (pb/pr/a/b/c/d) values

5. ConvertCompactPix
 	
input:
- 2D plain array of compressed pixels, each containing floating point (pb/pr/a/b/c/d) values
purpose:
- convert the values within in each pixel from floating point to their respective signed/unsigned representation
- use bitpack.c functions to perform shifts and bit transformations from floating point to ints
- scale the integers accordingly using the denominator of the original image
- create a new 2D array with the same dimensions as the input array, except each element will store one of the following structs:
typedef struct **codeword** {
unsigned int a
signed int b, c, d
unsigned int pb, pr
}
information loss:
	Joint Testing
	- pass the compression func a 2D blocked array with **PbPrabcd_float** structs at each pixel
	- pass the output of the compression func (2D array with **codeword** structs at each pixel) to the decompression func
	- compare the input to the compression func with the output of the decompression func (using mapping funcs to print out the contents of each array)
		5. ConvertCompactPix
		
		output:
		- 2D plain array of compressed pixels, each containing floating point (pb/pr/a/b/c/d) values
		purpose:
		- create a new 2D array with the same dimensions as the input array, except each element will store one of the following structs:
		typedef struct **PbPrabcd_float** {
		float pb, pr, a, b, c, d
		}
		- use bitpack.c functions to perform shifts and bit transformations from signed/unsigned ints to floating points
		- use the denominator of the image to de-scale (?) the integers
		information loss:
		- information is lost in this step, as we are converting from unsigned/signed ints to floating points, which are much less

- no information is lost in this step testing: - convert one codeword, and output the integer values → ensure they are each the correct number of bits, and that they are the numbers we expected, scaled as expected - ensure the output array has the same number of elements/pixels as the input array output: - 2D plain array of compressed pixels, each containing signed/unsigned (pb/pr/a/b/c/d) values → each pixel is a 32-bit codeword!		precise testing: - process one codeword, and output the float values → ensure the numbers are as expected, scaled as expected - ensure the output array has the same number of elements/pixels as the input array input: - 2D plain array of "codewords" → each element containing signed/unsigned (pb/pr/a/b/c/d) values

6. WriteCodewords
input:
- 2D plain array of compressed pixels, each containing signed/unsigned (pb/pr/a/b/c/d) values → each pixel is a 32-bit codeword!
purpose:
- go through the array and write each 32-bit codeword to stdout in row-major order, using putchar() to write the bytes in big-endian form
- write a properly formatted header, including the width and height of the original image (which will have to be saved somewhere in an
	Joint Testing
	- pass the compression func a 2D blocked array with **codeword** structs at each pixel
	- pass the output of the compression func (2D array with codewords) to the decompression func
	- compare the input to the compression func with the output of the decompression func (using ppmdiff unit_test)
	- Also we are going to pass the output of the compression function atp (a binary image) into the readcode words function on the decompression function to ensure
		7. ReadCodewords
		output:
		- 2D plain array of "codewords" → each element containing signed/unsigned (pb/pr/a/b/c/d) values
		purpose:
		- read through the input pbm file using fscanf() and from each "codeword," extract the following values and place into their corresponding positions in the struct:
		typedef struct **codeword** {
		unsigned int a
		signed int b, c, d
		unsigned int pb, pr
		}
		- since the codewords were written in big endian form, i assume

earlier step) information loss: - no information lost at this step testing: - write one codeword, ensure that it was written in big-endian form - ensure each codeword takes up two rows and two columns (maybe try writing one entire row/column) output: - binary image file containing the final, compressed image	that we have finished out the loop	we will have to take this into account - use the width and the height found in the header to create a 2D plain array that will store the collection of these codeword structs information loss: - no information lost at this step testing: - read one codeword, ensure that it properly populates each field in the struct - ensure that all elements of the 2D array are populated input: - binary image file containing the final, compressed image

---

the compressor

---

1. reads a ppm image from the file specified on the command line (using pnm_ppmread, so the image is read into a pnm_ppm image object).
2. trims the last row and/or column of the image so that the **width and height are even numbers.** (i.e. if the width is 49, it will cut down to 48). this seems like some information loss (
3. changes the numbers from an **unsigned to a floating point** representation (i.e. the red, green and blue values within the pgm_rgb struct) and transforms each pixel from RGB color space to **component video color space (Y, Pb, Pr).** Use these formulas to convert from from (R, G, B) to (Y, Pb, Pr) for each pixel: →

```
y  =  0.299    * r + 0.587    * g + 0.114    * b;
pb = -0.168736 * r - 0.331264 * g + 0.5      * b;
pr =  0.5      * r - 0.418688 * g - 0.081312 * b;
```

4. packs each 2 by 2 block into a 32 bit word by doing the following (remember that a 2x2 block contains 4 pixels from the original ppm image):
    - note: use a blocked array to represent the four pixels
    - to compute the y, pb and pr values for each block

pixel 1 - float Y - float Pb - float Pr	pixel 2 - float Y - float Pb - float Pr
pixel 3 - float Y - float Pb - float Pr	pixel 4 - float Y - float Pb - float Pr

- build 2d array to represent a block of four pixels
- call the mapping function of this array, with an apply function that:
    - converts the current pixel's r, g, b values to pb, pr, and y.
    - adds the current pixel's converted fields to the running sum of the final pixel (which is the closure parameter)
    - again, accepts the resulting pixel as the closure parameter and updates the final pixel's fields as it reads in each of the four pixels in the block
- after the mapping function is done running through the four pixels in the 2x2 block, we will have a final pixel with three fields:
    - a container of all four Y's (to perform DCT later)

    - sum of all Pb's
    - sum of all Pr's
- it is now our responsibility to apply the proper calculations to get the final pixels' Y, Pb and Pr values.
- **Pb and Pr** components are computed by taking the average value of the four pixels (using each of their Pb and Pr values) in the block.
- use the provided function to convert Pb and Pr elements from floating point to unsigned 4-bit values. this **function** returns a **4-bit quantized representation of a chroma (Pb or Pr) value,** assuming x is a chroma value between -0.5 and +0.5

```
unsigned Arith40_index_of_chroma(float x);
```

- **Y component** of the final pixel is computed using a discrete cosine transformation, which transforms the four pixels' Y values into cosine coefficients **a, b, c** and **d**
- converts the cosine coefficients b, c, and d into **five-bit signed values,** assuming they are between -0.3 and 0.3.
- packs a, b, c, d (all Y component), and Pb and Pr into a 32-bit word as follows:

Value	Type	Width	LSB
a	Unsigned scaled integer	9 bits	23
b	Signed scaled integer	5 bits	18
c	Signed scaled integer	5 bits	13
d	Signed scaled integer	5 bits	8
$index(\overline{P_B})$	Unsigned index	4 bits	4
$index(\overline{P_R})$	Unsigned index	4 bits	0

- note: don't worry about the index operation, it is implemented by artih40.
- note: you will need to develop an implementation of the bitpack.h interface in order to pack the codeword.

5. writes the compressed binary image to stdout in the following format:
    - header:

```
printf("COMP40 Compressed image format 2\n%u %u", width, height);
```

    - new line
    - sequence of 32-bit code words, one for each 2x2 block of pixels. width/height are the dimensions of the original (decompressed) image after trimming
        - 32 bit code words are written in big-endian (the block contains 4 bytes, put the most significant byte first). use **putchar()** to put each byte.
        - code words are written in row-major orders. since they are two-by-two blocks, each block will take up two columns/two rows.

---

the decompressor

- your decompressor will do the exact opposite of your compressor. as the inverse, your decompressor will take a compressed binary image file as input and do the following:
    1. read the header, calling **fscanf()** with the exact same string used to write the header. fscanf() will pull out the width and height (as **unsigned**) and store them in the variables you declared. the integer that fscanf() returns is the number of input items successfully matched and assigned (hence the assert == 2). then use **getc()** from the file stream that fscanf() was read into to ensure the string wasn't empty (i.e. the first character gotten wasn't a \n).
    2. create an instance of a Pnm_ppm struct to represent the pixmap, and populate its fields (i.e. unsigned width, height, denominator, pixels array, methods).
        - build the 2D pixels array using the width and height we got, and for the size parameter do the size of a colored pixel (which is?)
        - methods → plain methods, since
    3. "read" the 32-bit code words in order, remembering that each word is stored in big endian order. I assume this will be done using some sort of mapping function? remember each code word is compressed into one, 32-bit chunk that was previously four separate 32 bit chunks.
    - we will need to simultaneously have access to both the ppm image file and the binary image file. the closure parameter is the thing we want to write to (ppm image) and the thing we're mapping through is what we want to read from (binary image file)

      produce a **checked runtime error** if:
        - the number of codewords does not match the number of pixels in the array (specifically, if # codewords is **less than** the # pixels in the array) → to know the number of pixels, we would do (height * width)/4 mod 32, as the binary image file will have ¼ the number of pixels as the original ppm image)
        - the last codeword is incomplete (missing bits?) → i.e. last codeword has 30 bits instead of 32 or something
    4. from each codeword, you will need to **extract** the values a, b, c, d, Pb and Pr and store them in local variables (possibly a struct). this is called "unpacking" the compressed 32-bit sequence. → possibly a helper function within the apply function that performs the arithmetic to reproduce the Y, Pb, and Pr
    5. Pb and Pr are currently represented as 4-bit unsigned values. convert them back to **floating point numbers** using this formula (i think this is chroma quantization?): → i believe all four pixels that are written back will have the same, quantized Pb and Pr values (i.e. the "average" Pb and Pr)

       ```
       float Arith40_chroma_of_index(unsigned n);
       ```

    6. compute the four pixels' Y components (Y1, Y2, Y3, Y4) by using the **inverse discrete cosine transform** on a, b, c, and d. recall how we combined the four Y values to get a, b, c, and d in the first place!

7. now, for each of the four pixels in the original 2 x 2 block, we have gotten back their **original (Y, Pb, and Pr) values**. We now need to transform them from component video color to RGB color format, which is kind of tricky, because:
    - the values need to be “quantized” from floating point back to unsigned integers in a range 0 to xxx. **We are responsible for choosing an xxx (upper threshold) that is appropriate for the image denominator.**

   after converting each pixel back to a pnm_rgb struct, copy values back into pixmap->pixels (i.e. the pixels field of whatever you called you Pnm_ppm struct)
8. After putting all the pixels back into the pixmap, print it to standard output using Pnm_ppmwrite