Lab 9

Published

Tools needed: Part A: Hydra machine, Part B: Arduino

Part A

[Overview]

In this lab, you will be reading, manipulating, and writing binary, bitmap files.

This part of the lab requires usage of the Terminal. If you are on Windows, use MobaXTerm. If you are on Mac or Linux, your Operating System comes with a terminal that you can use.

Clara Nguyen (CS130 TA) has put together **a guide for how to use the Terminal (http://utk.claranguyen.me/guide.php?id=terminal_intro)** .

[Bitmap File Structure]

A 24-bit bitmap contains two headers that describe significant details about a picture, such as where to locate the pixel data, its width and height, and so forth. Each pixel contains three color values and is stored using 24-bits. Each color is stored in 1 byte (8 bits). They are stored in the order: blue, green, red and may contain values from 0 (no color) to 255 (full color). For example B=255, G=255, R=255 would be white, B=0, G=0, R=0 would be black, and B=10,G=10,R=255 would be a mostly red color.

File header (starts at offset 0)

The very top of a bitmap is what is called a file header. This header contains five fields and is 14 bytes total:

2 bytes - type (the type of the file, this will be Windows bitmap BM)
4 bytes - size (the entire size of the file, including headers)
2 bytes - reserved (reserved for future expansion)
2 bytes - reserved (reserved for future expansion too)
4 bytes - offset (this tells you the offset from the top of the file where the pixel data is stored)

Info header (starts at offset 14)

Right after the file header is the "info" header which describes the picture itself. It is 40 bytes total:

4 bytes - size (the size of the info header [should be 40])
4 bytes - width (the width of the picture in pixels)
4 bytes - height (the height of the picture in pixels)

2 bytes - planes (the number of layers, this should always be 1 for the test pictures)
2 bytes - bitcount (The number of bits per pixel, should be 24)
4 bytes - compression (the compression used, there is none, so this should be zero)
4 bytes - imagesize (the size of the image. May be zero for you to figure out!)
4 bytes - x_pixels_per_meter (the number of pixels per meter horizontally)
4 bytes - y_pixels_per_meter (the number of pixels per meter vertically)
4 bytes - color_used (the color index used, this should be 0 since we're not using indexed color)
4 bytes - color_important (again, should be 0)

Pixel data (starts at offset specified by file header [typically 54])

The pixel data starts at file_header->offset, but it is usually directly after the info header at offset 54 (14+40). The pixels are arranged with three channels BGR (blue, green, and red). Each color is 1 byte (8 bits).

There are width * height pixels, but there is something weird about bitmaps. Each row is "padded" to a multiple of 4 bytes. That means the number of bytes per row % 4 must be 0. For example, if I had a width of 3 and each pixel is 3 bytes, then my row takes 9 bytes. However, 9 % 4 = 1. Therefore, I must add 0s, 1 byte at a time, to the end of the row until the x % 4 = 0. In the previous example, 9 % 4 = 1, so lets add a "padded zero" which gives us 10 % 4 = 2, add another, 11 % 4 = 3, and another, 12 % 4 = 0. Ah, there we go. So, for a width of 3 using 3 bytes per pixel, each row is 12 bytes long. The pixel data itself only takes 9 bytes, then there are 3 zeros to give us a total of 12 bytes for the row. **NOTE:** It is extremely inefficient to keep calculating the amount of padding per row. However, if you think about it, you know the size of a pixel and the width of a row, so you will only calculate the amount of padding once! Furthermore, you may notice that the padding is either 0, 1, 2, or 3 bytes. Therefore, it is possible for a pixel row to already be a multiple of 4. If that's the case, no padding is added.

Lastly, pixel rows are stored in inverted order (upside down). This won't matter for this lab because you will be flipping around the Y axis (horizontally), but it is nice to know.

[Step 1: Reading / Writing binary files]

1. Download the test files: **lab9.bmp** (3 MB) and **lab9ut.bmp** (112 KB) and create a file called bmp.cpp. You will write all of your code in bmp.cpp

2. In this step, you will check to make sure you understand how to read a bitmap file. Write a program whose first command line argument is the input bitmap file and the second command line argument is the output bitmap file.

3. Create a structure called "Pixel" that stores the pixel color channels. **[See bitmap file structure for details on how to construct a pixel]**

4. Pixel is the ONLY structure you may use in this lab, and it must have a size of 3 bytes.

5. Using ifstream, read the offset (from file header), width, and height (from info header) from the file by **seeking** to the appropriate location.

6. Create a pixel array on the heap to store all of the pixels in memory. You must use a pointer of data type Pixel *.

7. Read in all of the pixels into memory using your pixel pointer. DO NOT FORGET to skip the padding after each row. This means you must read row-by-row rather than the entire pixel array all at once. You must use file **seeking** to skip the padding and move the file location to the next row. You must determine the amount of padding using the row size.

8. Now, using an ofstream, write the exact same file and info header to the output file.

9. Write the pixel data. Do not forget to include the extra padding after each row (if applicable).

10. Compile your code by using: **g++ -o bmp bmp.cpp**

11. Test your code by typing: **./bmp lab9.bmp test.bmp**

12. Check test.bmp to see if it is exactly the same as lab9.bmp (it should be!). Note: it is not enough to check the images using only visual inspection. The images may be different but it might not be easily apparent. To test if image files are the same, use the following command: **diff lab9.bmp test.bmp**

If the files are the same, you will not get any output. If the files differ, diff will say so.

[Step 2: Inverting]

1. Now you will need to add a step before you write the pixels to invert them in place. Remember that there are width * height pixels. Inversions mean that you take the maximum color intensity per channel (for blue, green, and red) and subtract from that the actual color intensity.

2. Re-compile and re-test your code.

3. The colors should now be inverted (blues should be reddish/brown and reds should be blackish/blue).

[Step 3: Flipping]

1. Now, either before or after step 2 (inverting the colors), add code that flips the image horizontally (around the Y axis) in place.

2. Since the pointer is a large array of pixels, you cannot directly index a row and a column (it isn't a 2D array). We know that pixel_array[0] is the bottom, left pixel and pixel_array[1] is the bottom, left+1 pixel. Therefore, pixel_array[width-1] would describe the rightmost pixel on the bottom array. Use this to determine how to index a row and column.

3. Re-compile and re-test your code.

4. Not only should the colors be inverted still, the image should be flipped around. That is, it should look like a mirror image.

[Step 4: Verifying]

1. If your input file is lab9.bmp, the output should look like **lab9_final.bmp**. If your input file is lab9ut.bmp, the output should look like **lab9ut_final.bmp**. You should use the diff command to check your output matches these images.

2. You must have a Pixel structure and use it for your pixel array. No other structures are allowed. Instead, you must seek using the appropriate methods to ifstream.

Part B

[Overview]

You will be implementing a cache simulator using the Arduino.

[Step 1: Understanding the code]

1. Download **cache.ino**

2. Do not edit any of the existing code in this file. You will be adding the member function definitions in this file where it says to put them (see the comments).

3. You will notice that there are three data structures: a) the Entry structure, b) the EvictPolicy enumeration, and c) the Cache class.

4. The Entry structure describes a single cache entry and has three fields: (1) epdata, (2) address, and (3) value. The (1) epdata variable is used to keep track of eviction policy data. You may use this as you see fit. For example, for the LRU eviction policy, I would use the epdata as the last time that the entry was get or set. The (2) address is the memory address contained in cache, and finally, the (3) value is the actual value at that memory address. **NOTE:** In actual cache the address in memory would be a "tag", which means only the part of the address that isn't already specified in the set. This is more space-efficient, but it is not used in this lab. Instead, you will be storing the entire address into the (2) address variable.

5. The EvictPolicy enumeration contains three policies: EP_FIFO - the first-in, first-out eviction policy, EP_LRU - the least-recently used policy, and EP_LFU - the least frequently used policy.

6. The Cache class contains five member fields: mSets - the number of cache sets, mMask - the bitmask used to determine which set a particular address maps to, mWays - the number of entries that each set contains, mPolicy - the eviction policy (one of EP_FIFO, EP_LRU, or EP_LFU), and mEntries - a pointer to all of the entries. mEntries must store all the sets with each set containing its own ways! So, for example, if I have 2 sets and 3 ways, there must be 2*3 = 6 entries.

[Step 2: Constructing]

1. Implement the Cache() constructor. This requires three parameters: an integer _bits, an integer _ways, and finally the eviction policy _p. The _bits parameter is a number from [0, 32]. This is the number of bits that will be used for the set mapping. These bits will be obtained starting with the least significant bit of the address. For example, if _bits is 2, then 010**11** and 001**11** will map to the exact same set because their rightmost bits are the same.

Notice there is no number of sets parameter. You will use _bits to determine how many sets you have to make. You will also use _bits to set mMask [NOTE: _bits and mMask are NOT the same thing!].

The integer _ways tell you the numbers of ways that the cache should have. The eviction policy _p directly relates to mPolicy. For all of these member fields, you must use the initializer constructor because they are constant.

Finally, inside the body of the constructor, make sure that you set the cache entries properly. For example, how will you determine if a way inside of a set is empty? I would recommend using the epdata, but if you do this, the epdata variable must contain a value that tells you it is empty.

Restrictions: You may ONLY use a single bit-shift operator to determine the number of sets based on the "_bits" parameter. You may ONLY use a single arithmetic operation to determine the mask.

2. Implement the ~Cache() destructor. This will simply free the memory from the heap that you created in the constructor. Make sure you use the appropriate keyword!

3. Implement the FindSet() member function. This takes an address and returns which set that address will map to. **NOTE:** The book makes use of the modulo operator (%) when determining which set an address maps to. This is inefficient, so you must use a logical operator and your mask to accomplish this task.

4. Implement GetNumSets, GetNumWays, and GetPolicy. These will simply return mSets, mWays, and mPolicy, respectively. These will be used by the setup code, so they must return appropriate variables, otherwise, it may crash your Arduino.

5. Implement the GetEntry function. This function takes two parameters: a set and a way. Its job is to return the entry at the given set and given way. For example, GetEntry(1, 3) will return a pointer to the entry in fourth way of the second set.

6. Implement the Set function. The Set function takes an address and a value. Its purpose is to find the appropriate location in cache (set and way) and store the epdata/address/value entry.

7. Implement the Get function. The Get function takes an address and returns the value at that address. If the address is not located in cache, return the sentinel value -99887766.

NOTE: For time on the Arduino, you may use the millis() function. It returns the number of milliseconds since the Arduino restarted. It has a 50-day cycle, so that will give you plenty of time without repeating.

[Step 3: Testing]

1. Test your code by first compiling, fixing the errors, and then uploading your code.

2. Type 'h' to get a list of commands.

3. Do a thorough job of setting values, adding more values than ways, and make sure your cache is properly evicting depending on the policy.

4. You will notice the LEDs will cascade. Since it is possible to crash your Arduino, the "cascading" tells you if your board is still running. If the LEDs stop, you will need to restart your Arduino by pressing the reset button on the board or re-uploading your code via the Arduino IDE.

5. Try creating a new cache (using the c command) with 0 as the number of bits (fully associative cache). If you did your logic correctly in the constructor, your code will handle this.

6. Try creating a new cache with the number of ways being 1 (direct-mapped cache).

[You are finished with this lab!]

1. Rename cache.ino to cache.txt.

2. Submit bmp.cpp and cache.txt.

Points	150
Submitting	a file upload
File Types	txt and cpp

Due	For	Available from	Until
Oct 25, 2017	Everyone else	Oct 20, 2017 at 8am	Oct 25, 2017 at 11:59pm
Nov 20, 2017 at 7:45pm	1 student	-	Nov 20, 2017 at 7:45pm

Lab 9

Criteria	Ratings					Pts
PartA: Reading bitmap file -Reads all pixels in a row using a single .read() -Finds the offset, width, and height by appropriately seeking -Skips padding by seeking	**10.0 pts Full Marks**	**7.0 pts Minor**	**5.0 pts Some**	**1.0 pts Major**	**0.0 pts No Marks**	10.0 pts
PartA: Writing bitmap file -Writes all pixels in a row using a single .write() -Adds padding -Padding is calculated ONLY ONCE!	**10.0 pts Full Marks**	**7.0 pts Minor**	**4.0 pts Some**	**1.0 pts Major**	**0.0 pts No Marks**	10.0 pts
PartA: Dynamic memory allocation (pixels) -Uses the "new" operator w/ array notation -Dynamic memory is deleted w/ array notation when no longer needed	**5.0 pts Full Marks**				**0.0 pts No Marks**	5.0 pts
PartA: Pixel inversion	**20.0 pts Full Marks**	**16.0 pts Minor**	**10.0 pts Some**	**5.0 pts Major**	**0.0 pts No Marks**	20.0 pts
PartA: Horizontal flip	**25.0 pts Full Marks**	**20.0 pts Minor**	**12.0 pts Some**	**8.0 pts Major**	**0.0 pts No Marks**	25.0 pts
PartB: Creating cache (Constructor) -Determining # of sets must only use logical operators! No external function calls (such as pow, etc) -Determining bitmask must only use logical operators! -mEntries must be a singly allocated array. It stores both sets and ways, so it is much like the bitmap file in Part A.	**25.0 pts Full Marks**	**17.0 pts Minor**	**13.0 pts Some**	**7.0 pts Major**	**0.0 pts No Marks**	25.0 pts
PartB: Setting a cache value -Eviction policy must be correct -Must scan for address in cache first -Cannot have two of the same addresses in the same set!	**25.0 pts Full Marks**	**20.0 pts Minor**	**15.0 pts Some**	**8.0 pts Major**	**0.0 pts No Marks**	25.0 pts
PartB: Getting a cache value	**25.0 pts Full Marks**	**20.0 pts Minor**	**14.0 pts Some**	**5.0 pts Major**	**0.0 pts No Marks**	25.0 pts
PartB: Freeing cache (Destructor) -Memory must be PROPERLY freed in the destructor	**15.0 pts Full Marks**		**5.0 pts Major**		**0.0 pts No Marks**	15.0 pts

Criteria	Ratings	Pts
		Total Points: 160.0