COMP 206 Winter 2018 – Assignment 2

VERSION A – Feb 22, 2018

Objectives:

Build up "systems programming" aspect of C. Work with binary files, read and understand a simple file-format specification and write C code to interact with binary data. Experience with visual image data.

Instructions:

The assignment folder holds a tar archive with 4 .c files and 1 .h file inside. Extract it with the command "$ tar xzf COMP206_A2_provided.tar.gz", and you should end up with a file called A2_provided with the code and sample BMPs inside. These provided files define the structure of several programs which are meant to interact with BMP files, but the low-level details are missing. Your job is to fill in these details, by completing each function in the file "A2_bmp_helpers.c".

Each question text describes what you must code and how to test it. Also, the header file, "A2_bmp_helpers.h" has code-level descriptions of each variable involved, so you should read it carefully. Finally, the provided "main" programs, bmp_info.c, bmp_mask.c, and bmp_collage.c are provided so you see how your code will be connected to the user's command-line.

Handing In:

Upon completion, hand-in only the A2_bmp_helpers.c file to My Courses. This file is the only place you should write any new code (i.e., do not change any other .c or .h file provided).

Deadlines:

1. For full marks: Thursday March 8th
2. Final deadline (with 10% penalty): Sunday March 11th

Intro: What you'll need to know about BMP files

BMP (bitmap) files are a traditional way to store image that's still used a little, but has been mostly overtaken by jpg, png, and other more complex file formats. We'll use BMPs because they are simple enough for us to do everything needed with raw C more easily.

A BMP is made to store an image in binary format. It is not a text file, so if you open it with a text editor, it looks garbled (except the intial "BM" characters"). A BMP file starts with a header of a fixed length, where several of the key pieces of information are stored as 4-byte unsigned ints at fixed (zero-indexed) offsets:

1. The total **data size** of the entire file, including header and pixel data, at byte 2
2. The zero-indexed **offset** from the beginning of the file to the start of pixel data, at bytes 10
3. The **width** of each image row, in # of pixels, at bytes 18
4. The **height** of each image column, in # of pixels, at byte 22
5. The number of **bits per pixel**, which is related to the number of image colors, at byte 28

After the header, at the indicated **offset,** the pixel data is stored in sequential order:

1. Starting from the bottom left of the image
2. Progressing first along rows, from left to right
3. Progressing next along columns, from bottom to top
4. Within each pixel, the color order is first blue, then green, then red

Each row is stored in an even multiple of 4 bytes. If the natural size of the row is not an even multiple, then extra padding bytes must be added (typically filled with zero).

A full reference to the file format is here, with helpful visual example #1 and example #2.

Question #1 – BMP Info (30 marks)

The C file "bmp_info.c" contains a main function that calls the "bmp_open" and "bmp_close" functions and prints the file's information to the terminal if successful. You can compile the program with:

$ gcc bmp_info.c A2_bmp_helpers.c -o bmp_info

And run the program with:

$ ./bmp_info mario_16_bit.bmp

Initially, you should see the program printing 0's everywhere rather than the file's true information. Complete the function "bmp_open" within the file "A2_bmp_helpers.c" so that the bmp_info program always outputs the correct data.

Steps:

1. Read the first few bytes of the file, ensuring you see the "BM" characters and data size.
2. Read all of the data into heap memory (use malloc), and store the address in img_data.
    a. fread function recommended
3. Read each of the information fields using their byte offsets, and store them in the pointers provided
4. Compute the padding as the next multiple of four bytes that can store width*bpp bits

Testing:

1. Run the bmp_info program and ensure it matches the examples:

```
$ ./bmp_info mario_16_bit.bmp
bmp_info for file mario_16_bit.bmp:
  width         = 236
  height        = 253
  bpp           = 24
  padding       = 0
  data_offset   = 54
  The middle pixel has (X,Y)=(126,126) and
  color (R,G,B)=(0,0,9).
$
```

```
$ ./bmp_info metro_mid_left.bmp
bmp_info for file metro_mid_left.bmp:
  width         = 147
  height        = 303
  bpp           = 24
  padding       = 3
  data_offset   = 122
  The middle pixel has (X,Y)=(151,151) and
  color (R,G,B)=(10,10,10).
$
```

Question #2 – BMP Mask (30 marks)

The C file "bmp_mask.c" contains a main function that calls the "bmp_mask" function, which is supposed to open a BMP file, modify the image data so that a rectangular region is set to a fixed color, and then write the content to different BMP file. You can compile the program with:

$ gcc bmp_mask.c A2_bmp_helpers.c -o bmp_mask

And run the program with:

$ ./bmp_mask mario_16_bit.bmp mario_masked.bmp 150 150 200 200 0 0 0

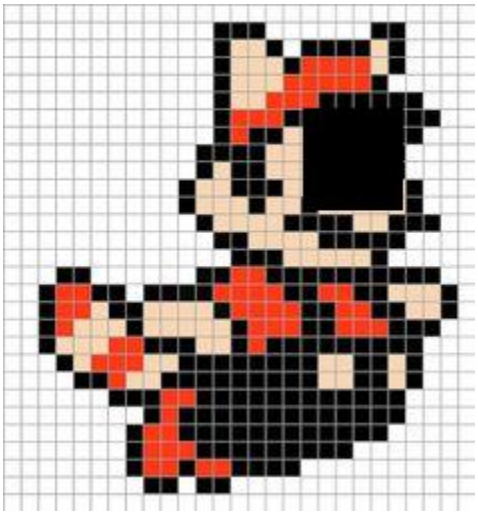
Initially, the program will do nothing and only tell you to complete the code. You must complete the "bmp_mask" function within the file "A2_bmp_helpers.c" so that the output image is produced. Details for exactly what is meant by the mask parameters are given in the "A2_bmp_helpers.h" header file.

Steps:

1. Call your own implementation of bmp_open. Use the main of bmp_info.c as a starter.
2. Allocate memory for your new image (either stack or heap array work). Use data_size.
3. Copy the contents of the input into the new image. Check memcpy for this.
4. Write a loop to iterate over the pixels in the mask regiøn. Set their color.
5. Save the result in a new BMP file (recommend fwrite, but other methods can work).

Testing:

1. Ensure your output matches the example given in the assignment archive. Run with "$ ./bmp_mask mario_16_bit.bmp mario_masked.bmp 150 150 200 200 0 0 0" to get:



Question #3 – BMP Collage (40 marks)

The C file "bmp_collage.c" contains a main function that calls the "bmp_collage" function, which is supposed to open two BMP files, and combine their contents into a (potentially) larger output collage. You can compile the program with:

$ gcc bmp_collage.c A2_bmp_helpers.c -o bmp_collage

And run the program with:

$ ./bmp_collage metro_top_left.bmp metro_top_right.bmp metro_collage.bmp 65 135

Initially, the program will just tell you the code is not complete. You must complete the "bmp_collage" function within the file "A2_bmp_helpers.c" so that the output image is produced with the correct contents. Some info about the details of collaging is in the "A2_bmp_helpers.h" header. Examples will be on the next page.

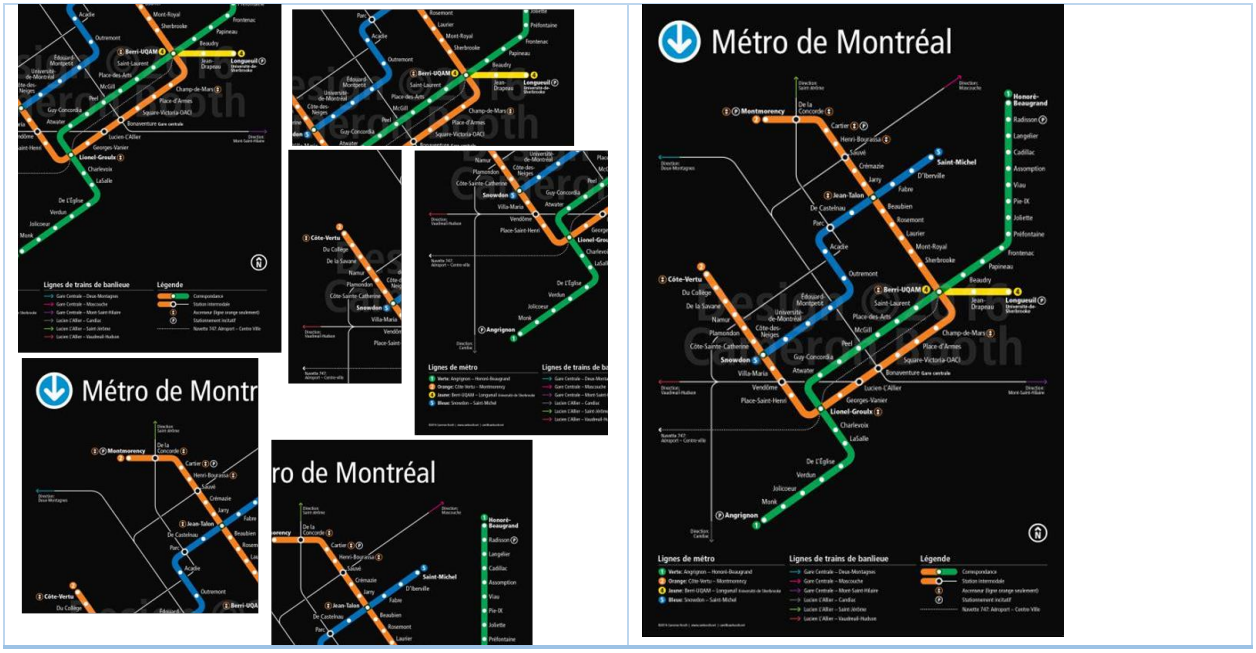
Steps:

1. The provided code already opens both images for you, and holds all of their information in variables. Leave that part alone and start on the next line.
2. First, compute the size of the new region and its boundaries with respect to each of the images.
3. Use the new size to allocate memory for the new image (either stack or heap array work). Use the new size you have computed, and don't forget to make space for the new header
4. Initialize the header of your new file to be identical to one of the input bmps (use memcpy). Then, change just the fields you need (recommend: length, width, overall size)
5. Write a loop that iterates over the pixels of the inputs and/or output, determining the correct logic and indexing to apply to each pixel to achieve the collage.
6. Write your result to the output BMP file (again, fwrite is handy, but you have options).

Testing:

1. We have provided a bash script called "create_metro_collage.bash". Run it as "$ bash create_metro_collage.bash". If all is successful, you will see that Montreal's combined metro map built up from the small pieces provided, as shown on the next page.
2. Until the full process works, you may like to copy/paste only one of the lines from the bash file and run them manually, perhaps with gdb to help you work through the details.

On the left we have shuffled the individual pieces of the metro map, which are provided to you in the archive as "metro_top_left.bmp" etc. On the right is the output you could get with your bmp_collage program run multiple times with the create_metro_collage.bash script provided.



To show what collaging "on top" means. Check that "$ ./bmp_collage utah.bmp mario_16_bit.bmp mixed_collage.bmp 100 50" gives you this output: