ECE26400 Programming Exercise #7

This exercise is related to PA04. This exercise will familiarize you with structures, dynamic memory allocation, and file operations. In particular, we will deal with an image format called BMP. Most systems and web browsers are able to read and display BMP files. It is a modification of an exercise that Prof. Yung-Hsiang Lu used in his class.

The main learning goals are:

1. How to read and write a structure into a file.
2. How to manipulate arrays.

1 Getting started

You should unzip on eceprog.ecn.purdue.edu the zip file `pe07_files.zip` using the following command:

```
unzip pe07_files.zip
```

The zip file `pe07_files.zip` contains a folder, named `PE07`, and two files within the folder and two subfolders containing some sample files:

1. `answer07.h`: This is a "header" file and it declares the functions you will be writing for this exercise.
2. `answer07.c`: One of the functions in `answer07.h` has been partially coded. You have to complete that function. You have to define all other functions declared in `answer07.h`.
3. `images`: This is a subfolder with valid BMP files.
4. `corrupted`: This is a subfolder containing invalid BMP files.

In this exercise, you have to define in `answer07.c` all functions that have been declared in `answer07.h` and create a file called `pe07.c` for the `main` function.

Please note that there are changes to the `gcc` command used for the compilation of your program. **Please read Section 4**.

2 Image file format

For the purpose of this exercise (and PA04), the BMP files we deal with have the following format:

```
/*
 * BMP files are laid out in the following fashion:
 *   --------------------------
 *   |          Header        |   54 bytes
 *   |-------------------------
 *   |       Image Data       |   file size - 54 bytes
 *   --------------------------
 */
```

The header has 54 bytes, which are divided into the following fields. Note that the #pragma directives ensure that the header structure is really 54-byte long by using 1-byte alignment. The first #pragma directive pushes the current alignment scheme on a stack. The second #pragma directive informs the compiler to use 1-byte alignment. Consequently, each field in the structure is packed closely together, with no padding bytes between adjacent fields. The third #pragma pops from the top most alignment scheme on the stack and compiler reverts to the original alignment scheme.

```
#pragma pack(push)
#pragma pack(1)

/**
 * BMP header (54 bytes).
 */

typedef struct _BMP_header {
    uint16_t type;                    // Magic identifier
    uint32_t size;                    // File size in bytes
    uint16_t reserved1;               // Not used
    uint16_t reserved2;               // Not used
    uint32_t offset;                  // Offset to image data in bytes
                                      // from beginning of file (54 bytes)
    uint32_t DIB_header_size;         // DIB header size in bytes (40 bytes)
    int32_t  width;                   // Width of the image
    int32_t  height;                  // Height of image
    uint16_t planes;                  // Number of color planes
    uint16_t bits;                    // Bits per pixel
    uint32_t compression;             // Compression type
    uint32_t imagesize;               // Image size in bytes
    int32_t  xresolution;             // Pixels per meter
    int32_t  yresolution;             // Pixels per meter
    uint32_t ncolours;                // Number of colors
    uint32_t importantcolours;        // Important colors
} BMP_header;

#pragma pack(pop)
```

The fields used here have types `uint16_t`, `uint32_t`, and `int32_t`. The number in each of these types indicates the number of bits in the type. Therefore, the number of bytes each field occupies can be obtained by dividing the number 16 or 32 by 8. For example, the `type` field occupies 2 bytes. These fields are all treated as integers. An `uint` means unsigned, and `int` means signed. For example the `width` and `height` fields are signed integers. When we add up the sizes of all fields in the structure, you should have 54 bytes for the `BMP_header` structure.

For simplicity, all the BMP files for PE07 and PA04 will contain only non-negative integers. In other words, you may assume that the most significant bit of any field that is of signed type is 0 in your code. Also, we are dealing with uncompressed BMP format (compression field is 0).

Because of the packing specified in the `answer07.h` file, you should be able to use a single `fread` statement to read in the first 54 bytes of a BMP file and store 54 bytes in a `BMP_header` structure. Similarly, you should be able to use a single `fwrite` statement to write the 54 bytes in a `BMP_header` structure to a file.

Among all these fields in the `BMP_header` structure, you have to pay attention to the following fields:

- `bits`: Number of bits per pixel
- `width`: Number of pixels per row
- `height`: Number of rows
- `size`: File size
- `imagesize`: The size of image data, which is (`size` − `sizeof(BMP_header)`), with `sizeof(BMP_header)` being 54.

We will further explain the `bits`, `width`, `height`, and `imagesize` fields later. You should use the following structure to store a BMP file, the `header` for the first 54 bytes of a given BMP file, and `data` should contain an address that points to a location that is big enough (of imagesize) to store the image data (color information of each pixel).

```
typedef struct _BMP_image {
    BMP_header header;
    unsigned char *data;
} BMP_image;
```

Effectively, the `BMP_image` structure stores the entire BMP file.

2.1 The `bits` field

The `bits` field records the number of bits used to represent a pixel. For this exercise (and PA04), we are dealing with BMP files with only 24 bits per pixel or 16 bits per pixel. For 24-bit representation, 8 bits (1 byte) for RED, 8 bits for GREEN, and 8 bits for BLUE. For 16-bit representation, each color is represented using 5 bits and the most significant bit is not used.

For this exercise, we will use only 24-bit and 16-bit BMP files to test your functions. Note that the header format is actually more complicated for 16-bit format. However, for this exercise and PA04, we will use the same header format for both 24-bit and 16-bit BMP files for simplicity. **So yes, we are abusing the format!**

RGB means that R occupies a more significant position and B occupies a less significant position in both 24-bit and 16-bit representations. Also note that the most significant bit of the 16-bit representation is not used for PE07 and PA04.

2.1.1 Structures for 24-bit and 16-bit representations

Do you know how to define a structure for the 24-bit representation such that each field in the structure corresponds to one of the three colors? Similarly, do you know how to define a structure for the 16-bit representation such that each field in the structure corresponds to one of the three colors? How should you order the fields in the structures such that R occupies a more significant position and B occupies a less significant position in both structures?

2.2 The `width` and `height` fields

The `width` field gives you the number of pixels per row. The `height` field gives you the number of rows. **For the purpose of this exercise and PA04, both `width` and `height` should have positive values (>0).** Row 0 is the bottom of the image. The file is organized such that the bottom row follows the header, and the top row is stored at the end of the file. Within each row, the left most pixel has a lower index. Therefore, the first byte at the location pointed to by `data` in the `BMP_image` structure belongs to the bottom left pixel.

2.3 The `imagesize` field

The `imagesize` field records the total number of bytes representing the image.

The total number of (informative) bytes required to represent a row of pixel for a 24-bit representation is `width` $\times$ 3 and a 16-bit representation is `width` $\times$ 2.

However, the BMP format requires **each row to be padded at the end such that each row is represented by multiples of 4 bytes of data**. For a 24-bit representation, for example, if there is only one pixel in each row, we need an additional byte to pad a row. If there are two pixels per row, 2 additional bytes. If there are three pixels per row, 3 additional bytes. If there are four pixels per row, we do not have to perform padding.

While an image file provided as an input may have indeterminate values stored in the padding bytes, if there are, **all new images created by the `reflect_BMP_image` function in this exercise are required to have value 0 assigned to each padding byte.** (See details of the function in Section 3.1.)

The `imagesize` field should store a value that is equal to

$$(\texttt{height} \times \text{amount of data per row}).$$

Note that the amount of date per row includes padding at the end of each row.

The image data stored after the 54 bytes of header file in the file is actually one-dimensional. We also expect you to store the image data in a one-dimensional array of `unsigned char`, and the address of the array is stored in the `data` field of the `BMP_image` structure.

You can visualize the one-dimensional image data as a three-dimensional array, which is organized as rows of pixels, with each pixel represented by 3 bytes of colors (24-bit representation) or 2 bytes of colors (16-bit representation). However, because of padding, you cannot easily typecast the one-dimensional data as a 3-dimensional array.

Instead, you can first typecast it as a two dimensional array, rows of pixels (or rows of [amount of data per row]).

For each row of data, you can typecast it as a two-dimensional array, where the first dimension captures pixels from left to right, the second dimension is the color of each pixel (3 bytes or 2 bytes), i.e., pixels of 3 bytes or pixels of 2 bytes. Alternatively, if you have defined appropriate structures to represent the RGB colors of each pixel, you can think of each row of data as a row of pixel (structure) or a one-dimensional array of structures.

3 Functions you have to define

3.1 Functions in `answer07.c`

```
// Read BMP_image from a given file
//
```

```
BMP_image *read_BMP_image(char *filename);
```

This function reads from a file that contains a BMP image, and stores the contents of the file in a `BMP_image` structure. The name of the file is provided through the first parameter `filename`. The function returns the address of the `BMP_image` structure in which the file contents are stored. As we want this `BMP_image` structure to be available throughout the entire (almost) lifetime of the executable, your function should allocate the memory for the `BMP_image` and the memory for the image data. The location of the memory for the image data should be stored in the `data` field of the `BMP_image`.

If the given file contains an invalid BMP file (i.e., the BMP header contains incorrect information about the file), the function should return `NULL`. If there are issues allocating memory or reading the file, the function should also return `NULL`.

```
// Check the validity of the header with the file from which the header is read
//
bool is_BMP_header_valid(BMP_header *bmp_hdr, FILE *fptr);
```

The function has been partially written and provided in `answer07.c`. We assume that `*bmp_hdr` already contains the header information read from `fptr`. Now, you have to check whether the header information in `*bmp_hdr` is valid. You may not assume anything about the file position index of `fptr`.

As the function has been partially written, you only have to fill in the part to check that the `size` field matches up with the actual file size, and the `size` and `imagesize` fields match up with the given `bits`, `width`, and `height` fields (see Section 2). The function returns `true` if the header is valid; otherwise, it returns `false`. This function should be called in the `read_BMP_image` function. It will be tested together with the `read_BMP_image` function. If that function does not work properly, this function would be deemed non-functional.

```
// Write BMP_image to a given file
//
int write_BMP_image(char *filename, BMP_image *image);
```

Given the filename of an output file, and an address to a valid `BMP_image` structure, write the image to the output file using the BMP format we introduced in Section 2. The function returns 1 if the writing to the output file is successful; otherwise, it returns 0.

```
// Free memory in a given image
//
void free_BMP_image(BMP_image *image);
```

This function frees the memory that is pointed by the address stored in `image`. You also have to free the memory used to store the image data.

```
// Given a BMP_image, create a new image that is a vertical and/or horizontal
// reflection of the given image
// All padding bytes must be assigned 0
// The new image may be similar to the given image, except the padding bytes
// It may be a horizontal reflection (with the vertical mirror being placed
// at the center of the image)
// It may be a vertical reflection (with the horizontal mirror being placed
// at the center of the image)
// It may be a horizontal reflection followed by a vertical reflection (or
// equivalently, a vertical reflection followed by horizontal reflection).
// hrefl == true implies that a horizontal reflection should take place
// hrefl == false implies that a horizontal reflection should not take place
// vrefl == true implies that a vertical reflection should take place
// vrefl == false implies that a vertical reflection should not take place
//
BMP_image *reflect_BMP_image(BMP_image *image, bool hrefl, bool vrefl);
```

For the given image, you may assume that it is a valid image; otherwise, this function should not be called at all. The image could be in a 24-bit representation or a 16-bit representation.

`hrefl` is a flag to indicate whether a horizontal reflection should take place. A horizontal reflection is a reflection that assumes a vertical axis in the middle of the image and reflect the left of the given image to be the right of the new image and the right of the given image to be the left of the new image. If `hrefl` is `true`, horizontal reflection should take place. If `hrefl` is `false`, horizontal reflection should not take place.

`vrefl` is a flag to indicate whether a vertical reflection should take place. A vertical reflection is a reflection that assumes a horizontal axis in the middle of the image and reflect the top of the given image to be the bottom of the new image and the bottom of the given image to be the top of the new image. If `vrefl` is `true`, vertical reflection should take place. If `vrefl` is `false`, vertical reflection should not take place.

You may assume that `hrefl` and `vrefl` will be supplied with `false` or `true`. If both hrefl and vrefl are `true`, it does not matter which reflection is performed first. If your implementation is correct, you will get the same new image regardless of the order in which you perform the vertical and horizontal reflections.

If both hrefl and verfl are `false`, the function should still allocate an appropriate amount of memory for the new image.

The input image should not be modified at all.

It is critical that you include an appropriate number of padding bytes for each row in the new image, and that you assign 0 to these padding bytes.

The returned value should an address that points to a (newly allocated) `BMP_image` structure. The header of the new `BMP_image` structure should be valid, such that if the returned address is passed to the `write_BMP_image` function, a valid BMP file will be stored. The reflected image should be stored as data in the new `BMP_image` structure.

If there is a problem with memory allocation, the function returns NULL.

3.2 `main` function in `pe07.c`

Your main function should expect an arbitrary number of `"-v"` or `"-h"` options, followed by an input BMP filename, and an output BMP filename. The `"-v"` option indicates a vertical reflection and the `"-h"` option indicates a horizontal reflection.

For example, if we compile `pe07.c` and `answer07.c` into executable `pe07` (see Section 4) and run the executable as follows:

```
./pe07 -v -h -v -v inputfile.bmp outputfile.bmp
```

The new image to be written into `outputfile.bmp` should be obtained by performing vertical (`argv[1]`), horizontal (`argv[2]`), vertical (`argv[3]`), and vertical (`argv[4]`) reflections on the original image stored in `inputfile.bmp`. The reflections are performed in the order in which the options ”-v” and ”-h” appears in the command line. **Of course, you have to decide how the `reflect_BMP_image` function should be called in order to perform any sequence of reflections. How many times do you have to call the function?**

It is also valid to run the executable as follows:

```
./pe07 inputfile.bmp outputfile.bmp
```

Although you do not perform any vertical or horizontal reflections on the input image, the output image may differ from the input image if the padding bytes of the input image have indeterminate values (and the padding bytes of the output image are assigned 0). In other words, a new image should still be created and be saved into `outputfile.bmp`.

If you encounter an invalid option, you should print an error message to stderr and return `EXIT_FAILURE`.

After you have exhaustively processed all valid options, you should be left with 2 arguments in the command line. Otherwise, you should print an error message to stderr and return `EXIT_FAILURE`.

The second-to-last argument should contain a valid BMP file. Otherwise, you should not produce a new BMP file, and you should return `EXIT_FAILURE`.

The image from the second-to-last argument should be reflected (according to the `"-v"` or `"-h"` options, and the new image should be output to the last argument. If the call to the `reflect_BMP_image` returns a valid image, and the writing to the output file is successful, you should return `EXIT_SUCCESS`. Otherwise, you should return `EXIT_FAILURE`.

If for whatever reasons (insufficient arguments, memory allocation problem, file opening issue, format issue), the reflection(s) cannot be performed, the output should not be printed and you should return `EXIT_FAILURE`.

Whenever you encounter errors and you want to print an error message, use `stderr` for printing, not `stdout`. In such a case, always return `EXIT_FAILURE`.

Return `EXIT_SUCCESS` only if image reflection and writing are successful.

You are responsible for opening and closing files, allocating and deallocating memory.

3.3 Printing, helper functions and macros

All debugging, error, or log statements in `answer07.c` and `pe07.c` should be printed to `stderr`.

You may define your own helper functions in `pe07.c` and `answer07.c`. All these functions should be declared and defined as static functions. In other words, the scope of these functions are only within the files that they are found. Declaring and defining these functions as static eliminate the conflicts in function names.

You should not use macros that start with `T_` (Upper case T and underscore) in their names. We will use such macros in the evaluation of your submissions. If you use such macros, we may not be able to evaluate your submission properly.

4 Compiling your program

As you have advanced to this stage, we believe that you have the necessary experience to decide what warnings are allowed and what warnings should be eliminated. We will now remove the `-Werror` flag from the `gcc` command.

```
gcc -std=c99 -Wall -Wshadow -Wvla -g -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe07.c answer07.c -o pe07
```

When we evaluate your program, we also use the optimization flag `-O3` (uppercase letter `O` and number three) instead of the `-g` flag as follows:

```
gcc -std=c99 -Wall -Wshadow -Wvla -O3 -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe07.c answer07.c -o pe07
```

You are recommended to copy the `Makefile` from PE01 and modify it appropriately for PE07.

In fact, we will remove the `-Werror` flag for all exercises and assignments that come after PE07. However, for the re-submissions of PA01, PA02, and PA03, we will continue to use the `gcc` command as mentioned in the respective PDF files for those assignments.

5 Writing, running and testing your program

We provide a few test cases for you. The `images` subfolder contains some images that are of the correct format. Each reflected output file is named using the options used in its production: if we use `"-v -h"`, the output file should have a name like `filename_vh.bmp`, where `filename.bmp` is the name of the input filename.

You can use the `diff` command to compare your own output files and the corresponding files in the `images` subfolder.

The `corrupted` subfolder contains some corrupted BMP files. Note that most image viewers are very robust and could still display these "corrupted" images properly. However, for the purpose of this exercise, they are considered to be corrupted. If you run your executable on the files in the `corrupted` folder, the conversion should not take place.

You should also run `./pe07` with appropriate arguments under `valgrind`.

Please see PE01 description about `valgrind`.

6 Submission

You must submit a zip file called `PE07.zip`, which contains two files:

1. `answer07.c`
2. `pe07.c`

Assuming that you are in the folder that contains `answer07.c` and `pe07.c`, use the following command to zip your files:

```
zip PE07.zip answer07.c pe07.c
```

Make sure that you name the zip file as `PE07.zip`. Moreover, the zip file should not contain any folders. Submit `PE07.zip` through Brightspace. You may submit as many versions as you like. Brightspace will keep only the most recent submission, and we will grade only the final submission.

7 Grading

All debugging messages should be printed to `stderr`. It is important that if the instructor has a working version of `pe07.c`, it should be compilable with your `answer07.c` to produce an executable. For evaluation purpose, we will use different combinations of your submitted `.c` files and our `.c` files to generate different executables. If a particular combination does not allow an executable to be generated, you do not get any credit for the function(s) that the executable is supposed to evaluate. We use both `-g` and `-O3` flags (separately) to compile your submission. Your submission is evaluated only when compilation using each of the flags is successful.

It is important to note that the `.c` files from the instructor do not assume the presence of global variables that are declared by the students. Of course, the `.c` files from the instructor may use global variables that are in C libraries, such as `errno`, `stdout`, `stderr`, and so on. If your submission contains global variables that are declared by you, it is unlikely that your executable will work correctly.

Be aware that we set a time-limit for each test case based on the size of the test case. If your program does not complete its execution before the time limit for a test case, it is deemed to have failed the test case. We will not announce the time-limits that we will use. They will be very reasonable.

The `read_BMP_image` function accounts for 30%. The `write_BMP_image` function accounts for 10%. The `reflect_BMP_image` function accounts for 40%. The `main` function accounts for 20%.

The occurrence of any memory issues (memory errors or memory leaks flagged in a `valgrind` report) will result in 50-point penalty.

8 A few points to remember

All debugging messages should be printed to `stderr`. If the program creates an output file when it should not, or the output file is correct, you get 0 for that test case.

You can declare and define additional static functions that you have to use in `pe07.c` and `answer07.c`.

You should not use macros that start with `T_`.

Grading of programming exercises and assignments is performed on machines with similar setup as eceprog.ecn.purdue.edu. You should perform testing of your work on eceprog.ecn.purdue.edu before submission. Correct output on your computer does not translate into correct output on eceprog.ecn.purdue.edu.