# 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, which is commonly used in the Windows operating system. Most web browsers are also 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.
3. How to perform bit-wise operations.

## 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 code. 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 5**.

## 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 directive ensures that the header structure is really 54-byte long by using 1-byte alignment.

```
#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.

However, for simplicity, all the BMP files we have 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 wih uncompressed BMP format (compression field is 0).

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

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 pixel per row

- `height`: Number of row
- `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.

## 2.2 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 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.

We `require` you to assign value 0 to each padding byte. 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 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.

# 3 Bit operations

This exercise requires you to convert a 24-bit BMP file into a 16-bit BMP file, and vice versa.

For the 24-to-16-bit converion, you have to deal with bit-wise operations. Here are a few that could be useful: shift, bit-wise AND, and bit-wise OR. Assume that you have three variables `a`, `b`, and `k`, all of type `int`.

```
b = a >> k;  // This is to take the bits stored in a, and shift
             // them the right by k bits.  Bits shifted out of the
             // least significant position are dropped.
             // The most significant bit of a is replicated
             // and shifted right.
b = a << k;  // This is to take the bits stored in a, and shift
             // them the left by k bits.  Bits shifted out of the
             // most significant position are dropped.
             // 0's are shifted in from the least significant bit
```

If you are not dealing with an `unsigned int`, the left shift may turn a positive number into a negative number.

If you are dealing with an `unsigned int`, the right shift will shift 0's into the most significant bit.

My understanding of the ALU (Arithmetic-Logic Unit) design is that the operations are typically performed on a 32-bit register. So, unless variable `a` is a `long int`, only the least signicant 5 bits of `k` will be used to decide the number of positions to shift. If `a` is a `long int`, only the least significant 6 bits of `k` will be used to decide the number of positions to shift.

Also, if variable `a` is a `short` or `char`, and `b` is an `int` (32 bits), as the operation is carried out on a 32-bit register, `b` would be assigned the value as if variable `a` is an `int`. However, if `b` is a long int, the value of `b` can become unpredictble because the shifting is performed based on 32 bits.

To avoid unpredictable results, try to have variables `a` and `b` that are of the same type.

In this exercise, the 8 bits or 5 bits representing a color should be treated as an unsigned value. In other words, the 8 bits allow values 0 through 255 and the 5 bits allow values 0 through 31.

(Note that the following description is an example, not how we actually represent a pixel in a BMP file.)

Suppose you have an `int`, but only the 3 lower significant bytes are storing the color information of a pixel (for 24-bit representation). The format RGB means that the red color (R) occupies a more significant position and the blue color (B) occupies a less significant position. (Note that if I treat the `int` as an array of 4 `unsigned chars`, say `unsigned char *array`, `array[0]` is B, `array[1]` is G, and `array[2]` is R.)

To extract the value of RGB from the `int` (`pixel` is the variable name), we can use bit-wise AND operator:

```
int red_value = (pixel & 0xFF0000) >> 16;
int green_value = (pixel & 0xFF00) >> 8;
int blue_value = (pixel & 0xFF);
```

The operator `&` (which is different from logical AND operator `&&`) performs bit-wise AND operator at every corresponding bit position. The bit pattern `0xFF0000` has eight 1's at the second most significant byte (of an `int`). When you bit-wise AND `0xFF0000` with variable `pixel`, the result will match the contents in pixel at only the second most significant byte; the most significant byte, the second least significant byte, and the least significant byte of the result will be 0. Shifting the result by 16 bits will now store the correct value for the red color in the variable `red_value`. The pattern `0xFF0000` is called a bit mask.

The following shows where the values of the red, green, and blue colors of a pixel are stored if a pixel is represented by, for example, `short`, `unsigned short`, `int16_t`, `uint16_t`?

```
/* bit positions of RGB in a 16-bit representation */
/* R: 0111110000000000 */
/* G: 0000001111100000 */
/* B: 0000000000011111 */
```

What should be corresponding masks for the red, green, and blue colors of a pixel?

Bit-wise AND operation can be used to extract selected group of bits from an int (or any other types). Bit-wise OR operation `|`, which is different from logical OR operator `||`, can be used to put group of bits at selected locations into an `int` (or any other types).

Continuing with the 24-bit example, to assemble three bytes of RBG information into an int `pixel`, we can use

```
pixel = (red_value << 16) | (green_value << 8) | blue_value;
```

Of course, we do not have to use any of these operations for 24-bit representation, because each color is a byte, and therefore `unsigned char` can easily represent each color. You may have to use the shift, bit-wise AND, and bit-wise OR to handle 16-bit representation, where bits 0 through 14 are for the three colors. The most significant bit should be 0 (and not used in a meaningful way for this exercise).

# 4 Functions you have to define

## 4.1 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
//
int 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 pointer of `fptr`.

As the function has been partially written, you only have to fill in the part to check that the `size` and `imagesize` fields match up with the given `bits`, `width`, and `height` fields (see Section 2). The function returns 1 if the header is valid; otherwise, it returns 0. 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 16-bit image that is converted from a given
// 24-bit image
//
BMP_image *convert_24_to_16_BMP_image(BMP_image *image);
```

For the given image, you may assume that it is a valid 24-bit image (Otherwise, this function should not be called at all.)

You have to take the 24-bit representation and convert it into a 16-bit representation. Note that you should not modify the original image. Instead, the function should allocate new memory spaces to store the new header information and the new image data.

The header information of the new image should not be identical to that of the original image. In particular, you would have to compute the number of bytes in each row and the total image size, and therefore the file size.

You have to downscale 8 bits to 5 bits for each color. You have to scale 0 through 255 to 0 through 31. A division of a color value by 8 or a right shift of 3 position should achieve similar outcome because you are dealing with non-negative numbers. You also have to combine the three colors into two bytes (using perhaps bit-wise OR operation). Lastly, you have to store the two bytes into the data array of the new `BMP_image`.

It is important that the padding bytes, if required, are all assigned 0.

```
// Given a BMP_image, create a new 24-bit image that is converted from a given
// 16-bit image
//
BMP_image *convert_16_to_24_BMP_image(BMP_image *image);
```

This function performs the reverse of the preceding function. For the given image, you may assume that it is a valid 16-bit image (Otherwise, this function should not be called at all.)

You have to take the 16-bit representation and convert it into a 24-bit representation. Note that you should not modify the original image. Instead, the function should allocate new memory space to store the new image.

You cannot simply perform a left shift of 3 bit positions or a multiplication of 8 map 0 through 31 to 0 through 255. The reason is that in that case, you will not have the highest possible value for 24-bit representation, which is 255. Rather, you will have 248 as the highest value. To make sure that the highest intensity of 31 in a 16-bit representation is mapped to the highest intensity of 255 in a 24-bit representation, you should use the following formula:

$$((\text{5-bit value}) * 255)/31$$

This conversion formula is important because you will rely on this in PA04.

It is important that the padding bytes, if required, are all assigned 0.

## 4.2 `main` function in `pe07.c`

Your `main` function should expect `argv[1]` as the input file and `argv[2]` as the output file.

If `argv[1]` does not provide you a valid BMP file, you should not produce a new BMP file, and you should return `EXIT_FAILURE`. Based on the format of a valid BMP file, you should perform a 24-to-16 bit conversion or a 16-to-24 bit conversion, and save the new image in `argv[2]`.

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

Return `EXIT_SUCCESS` only if it is a successful conversion.

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

### 4.3 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.

## 5 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.

## 6 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. Some of the images are converted from the original format. For example, `car16.bmp` is converted from `car.bmp`, and `car24.bmp` is converted from `car16.bmp`. There are also images that are without the converted versions.

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`.

# 7 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.

# 8 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 15%. and the `write_BMP_image` function accounts for 10%. The `convert_24_to_16_BMP_image` function accounts for 40%. The `convert_16_to_24_BMP_image` function accounts for 30%. The `main` function accounts for 5%.

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

# 9 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.