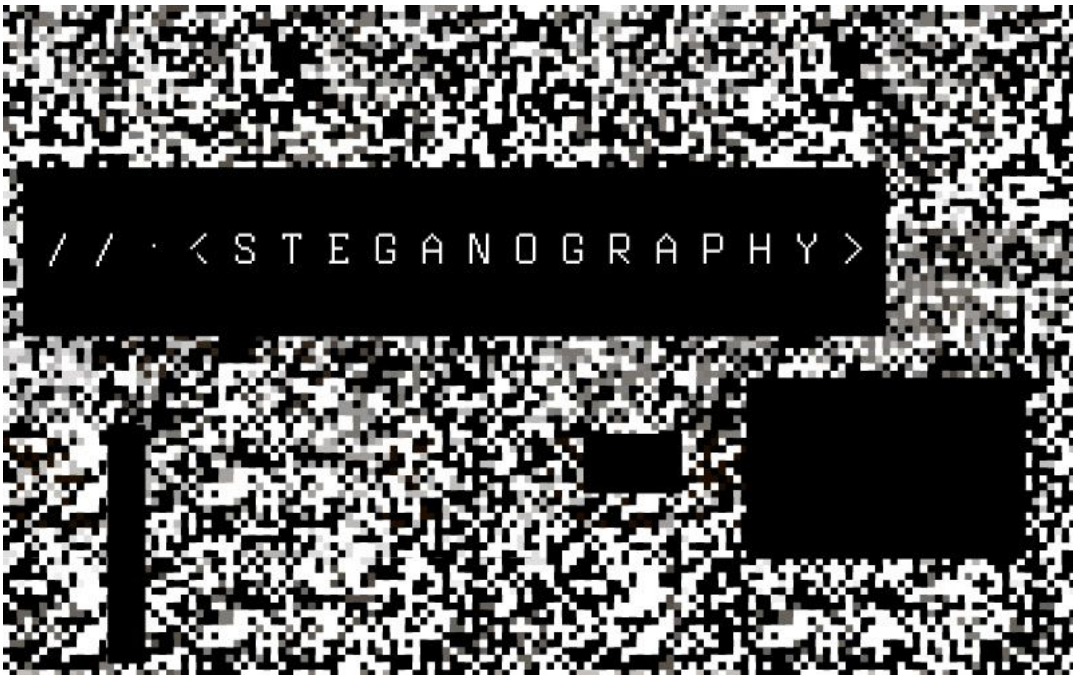
STEGANOGRAPHY

-

REPORT



컴퓨터보안 [CSE4308-001]

담당교수: 이문규

컴퓨터공학과, Inha Univ.

학번: 12180152

끼서폿 - KY SOPHOT

sophotk@gmail.com

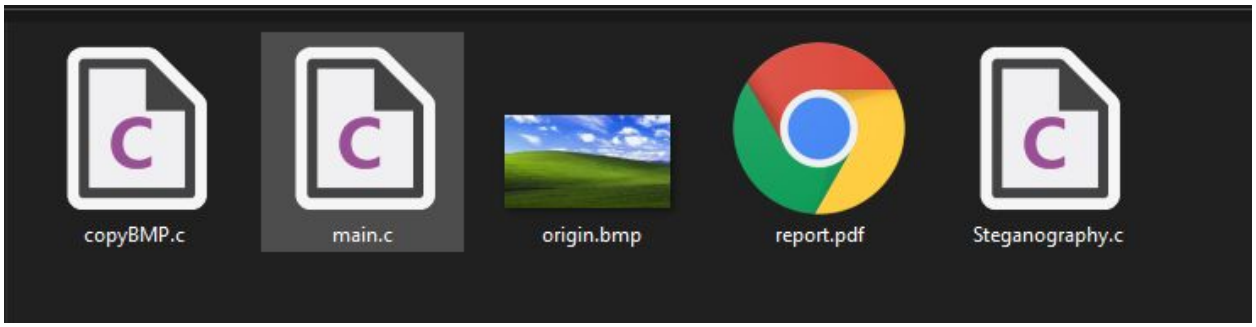
Contents

I. Introduction

In computer security, steganography[1] is the practice of concealing a file, message, image, or video within another file, message, image, or video. Generally, the hidden messages appear to be part of something else. In this assignment we are required to encode input text from the terminal into our BMP file, decode and output back onto the terminal. There are many approaches to steganography. Later in the report, I will explain the half-byte approach which I find simpler than the LSB approach for this particular assignment. Also, I will elaborate on the program structure as well as encoding and decoding algorithms to manipulate the bmp file.

A. Getting Started

Initially, we have the "origin.bmp" file which will be used as reference to make a "stego.bmp" that has encoded content in it. The core components are "main.c", "Steganography.c" and "copyBMP.c", which contain all source codes for the assignment. And the "report.pdf" which you are currently reading right now.



B. Demonstration

Compile: **gcc main.o -c stego**
Execute: **./stego.exe e** or **./stego.exe d** (other than 'e' or 'd' will not work)

In 'e' mode, type anything in to encode and press enter after finished. "stego.bmp" file will be created with the content of the input earlier.
In 'd' mode, if stego.bmp doesn't exist an error message will be printed, else the encoded message will be displayed.

[1] "Steganography - Wikipedia." https://en.wikipedia.org/wiki/Steganography

```
Phot@DESKTOP-TD2OLE3
$ gcc main.c -o stego

Phot@DESKTOP-TD2OLE3
$ ./stego.exe e
H

Phot@DESKTOP-TD2OLE3
$ ./stego.exe d
H
Decoded 1 characters
```

```
$ ./stego.exe e
Hello World

Phot@DESKTOP-TD2OLE3
$ ./stego.exe d
Hello World
Decoded 11 characters
```

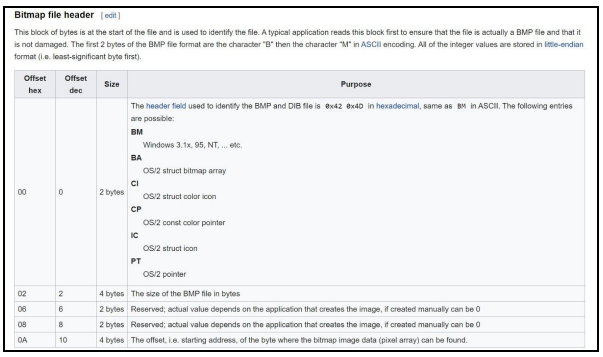
```
$ ./stego.exe e
sth wrong

Phot@DESKTOP-TD2OLE3 MINGW6
$ ./stego.exe d
Error opening "stego.bmp".
```

```
$ ./stego.exe e
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy
text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has
 survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popu
larized in the 1960s with the release of Letrasetsheets containing Lorem Ipsum passages, and more recently with desktop publis
hing software like Aldus PageMaker including versions of Lorem Ipsum.

Phot@DESKTOP-TD2OLE3 MINGW64 /D/UNIV/Year 3/Security/Assignment 1 (master)
$ ./stego.exe d
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy
text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has
 survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popu
larized in the 1960s with the release of Letrasetsheets containing Lorem Ipsum passages, and more recently with desktop publis
hing software like Aldus PageMaker including versions of Lorem Ipsum.
Decoded 573 characters
```

```
$ ./stego.exe e                                                                                                    Lorem Ip
sum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since
the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five
centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularized in the 1960s with the re
lease of Letrasetsheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker inclu
ding versions of Lorem Ipsum.Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the indus
try's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen
 book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was
 popularized in the 1960s with the release of Letrasetsheets containing Lorem Ipsum passages, and more recently with desktop publishin
g software like Aldus PageMaker including versions of Lorem Ipsum.Lorem Ipsum is simply dummy text of the printing and typesetting ind
ustry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and
 scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, re
maining essentially unchanged. It was popularized in the 1960s with the release of Letrasetsheets containing Lorem Ipsum passages, and
 more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.Lorem Ipsum is simply dummy tex
t of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unkn
own printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the
 leap into electronic typesetting, remaining essentially unchanged. It was popularized in the 1960s with the release of Letrasetsheets
 containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem
Ipsum.Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy te
xt ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived
not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularized in the 196
0s with the release of Letrasetsheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus P
ageMaker including versions of Lorem Ipsum.

Phot@DESKTOP-TD2OLE3 MINGW64 /D/UNIV/Year 3/Security/Assignment 1 (master)
$ ./stego.exe d
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text eve
r since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not on
ly five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularized in the 1960s wit
h the release of Letrasetsheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMak
er including versions of Lorem Ipsum.Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been t
he industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type
specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged
. It was popularized in the 1960s with the release of Letrasetsheets containing Lorem Ipsum passages, and more recently with desktop p
ublishing software like Aldus PageMaker including versions of Lorem Ipsum.Lorem Ipsum is simply dummy text of the printing and typeset
ting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of
type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typeset
ting, remaining essentially unchanged. It was popularized in the 1960s with the release of Letrasetsheets containing Lorem Ipsum passa
ges, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.Lorem Ipsum is simply d
ummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when
 an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but
also the leap into electronic typesetting, remaining essentially unchanged. It was popularized in the 1960s with the release of Letras
etsheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions o
f Lorem Ipsum.Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard
dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has s
urvived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularized in
 the 1960s with the release of Letrasetsheets containing Lorem Ipsum passages, and more recently with desktop publishing software like
 Aldus PageMaker including versions of Lorem Ipsum.
Decoded 2865 characters
```

C. BMP File Format

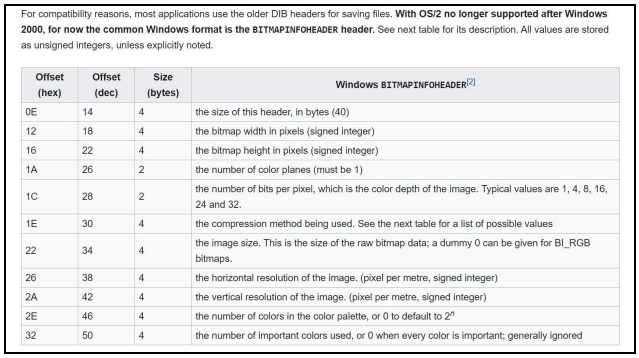
It is beneficial to understand about BMP file format[2] before we go to the code.



Bitmap file header [edit]

This block of bytes is at the start of the file and is used to identify the file. A typical application reads this block first to ensure that the file is actually a BMP file and that it is not damaged. The first 2 bytes of the BMP file format are the character "B" then the character "M" in ASCII encoding. All of the integer values are stored in little-endian format (i.e. least-significant byte first).

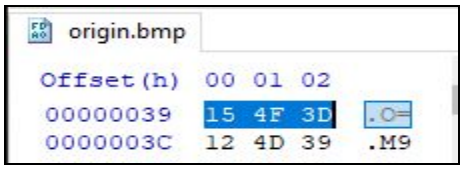
Offset hex	Offset dec	Size	Purpose
00	0	2 bytes	The header field used to identify the BMP and DIB file is `0x42 0x4D` in hexadecimal, same as `BM` in ASCII. The following entries are possible: **BM** Windows 3.1x, 95, NT, ... etc. **BA** OS/2 struct bitmap array **CI** OS/2 struct color icon **CP** OS/2 const color pointer **IC** OS/2 struct icon **PT** OS/2 pointer
02	2	4 bytes	The size of the BMP file in bytes
06	6	2 bytes	Reserved; actual value depends on the application that creates the image, if created manually can be 0
08	8	2 bytes	Reserved; actual value depends on the application that creates the image, if created manually can be 0
0A	10	4 bytes	The offset, i.e. starting address, of the byte where the bitmap image data (pixel array) can be found.



For compatibility reasons, most applications use the older DIB headers for saving files. **With OS/2 no longer supported after Windows 2000, for now the common Windows format is the `BITMAPINFOHEADER` header.** See next table for its description. All values are stored as unsigned integers, unless explicitly noted.

Offset (hex)	Offset (dec)	Size (bytes)	Windows `BITMAPINFOHEADER`[2]
0E	14	4	the size of this header, in bytes (40)
12	18	4	the bitmap width in pixels (signed integer)
16	22	4	the bitmap height in pixels (signed integer)
1A	26	2	the number of color planes (must be 1)
1C	28	2	the number of bits per pixel, which is the color depth of the image. Typical values are 1, 4, 8, 16, 24 and 32.
1E	30	4	the compression method being used. See the next table for a list of possible values
22	34	4	the image size. This is the size of the raw bitmap data; a dummy 0 can be given for BI_RGB bitmaps.
26	38	4	the horizontal resolution of the image. (pixel per metre, signed integer)
2A	42	4	the vertical resolution of the image. (pixel per metre, signed integer)
2E	46	4	the number of colors in the color palette, or 0 to default to 2^n
32	50	4	the number of important colors used, or 0 when every color is important; generally ignored

From the left and right figures the file header acquires 14 bytes and 40 bytes respectively. Total of 54 bytes.



The 3 highlighted hexadecimal numbers represent a pixel and RGB color of #3D4F15 ■ of origin.bmp file.

II. Source Code Explained

The source file that is needed to compile is “main.c”. So we’ll look at it first.

```c
#include <stdlib.h> //exit()
#include "Steganography.c"

#define MAX_LENGTH 65940

int main(int argc, char *argv[])
{
    //RECEIVES 2 INPUT ARGUMENTS ONLY FROM TERMINAL (#1 -> .exe file, #2 -> 'e' or 'd')
    if (argc != 2)
    {
        printf("Arguments do not match!\nRun again with './mystego.exe e' or './mystego.exe d'");
        exit(1);
    }
    //ASSIGN 2nd ARGUMENT TO MODE
    char *mode = argv[1];
    //CHECK IF ARGUMENT's(MODE) LENGTH IS NOT 1 CHARACTER
    if (sizeOfInputMode(mode) != 1)
    {
        printf("Argument is too long!\nRun again with './mystego.exe e' or './mystego.exe d'");
        exit(1);
    }
```

[2] "BMP file format - Wikipedia." https://en.wikipedia.org/wiki/BMP_file_format

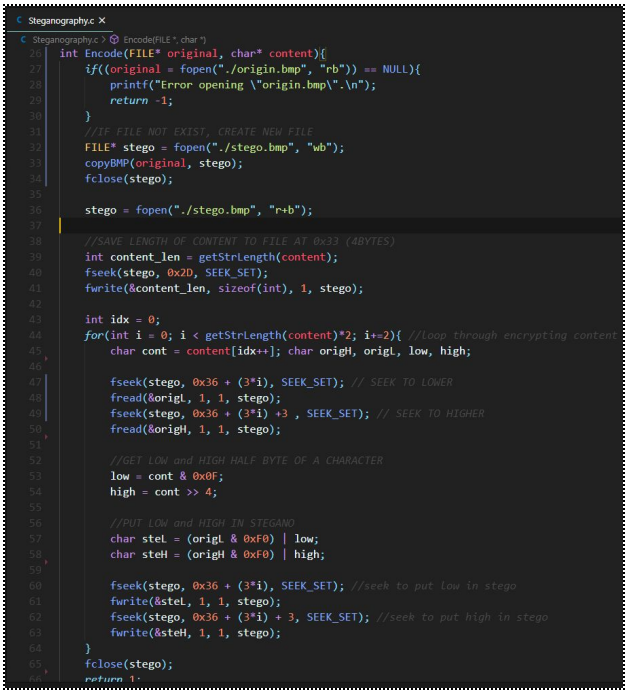
Line 9 - 13: I'm making sure that the input arguments received will always be 2.
Line 15: set mode to the 2nd argument. Because the 1st argument is the executable file.
Line 17 - 21: make sure that the 2nd argument mentioned earlier has size of 1. ('e' or 'd')

Line 24 - 40: Encode. Content to be encoded will be required to be input in the terminal. If nothing is input, "sth wrong" will be printed out and the program terminates. After getting input, a File is created. The file and content are passed as arguments to Encode() function.

Line 44 - 49: Decode. A file is created and passed as a parameter to Decode() function.

P.S: Implementation of Encode() and Decode() can be found on "Steganograpy.c"

Below is the **Encode()** function. It returns 1 on success and -1 if there's error.
Parameters: original file as reference to copy, and content to encode.



Line 27-30: attempt to open original file for reading.
Line 32-34: Create a file name (if it doesn't exist) stego.bmp and call copyBMP() to copy original.bmp to stego.bmp, and close the file.
Line 36: open stego.bmp again this time to write encoding content to the file.
Line 39-41: Calculate the size of content to encode and save it in stego.bmp at location 0x2D (I chose this location because as I check the addresses around it is not used. I.e hex 00 00 00).

Line 43-65: Looping through the content that will be encoded one character at a time and write its hexadecimal representation to stego.bmp and close the file.

Let's look at the **Decode()** function. It returns 1 on success and -1 if there's an error.

```c
int Decode(FILE* stego){
    if((stego = fopen("./stego.bmp", "rb")) == NULL){
        printf("Error opening \"stego.bmp\". MAKE SURE THE FILE EXIST!\n");
        return -1;
    }
    int content_len;

    //READ LENGTH OF CONTENT TO DECRYPT
    fseek(stego, 0x2D, SEEK_SET);
    fread(&content_len, sizeof(int), 1, stego);

    for(int i = 0; i < content_len*2; i+=2){

        char low;
        char high;
        fseek(stego, 0x36 + (3*i), SEEK_SET);
        fread(&low, sizeof(char), 1, stego);
        fseek(stego, 0x36 + (3*i) + 3, SEEK_SET);
        fread(&high, sizeof(char), 1, stego);


        high = high << 4;
        low = low & 0x0F;
        high = high | low;
        printf("%c", high);
    }
    printf("Decoded %d characters\n", content_len);
    fclose(stego);
    return 1;
}
```

Line 69-72: Is checking if the stego.bmp file exists, meaning has Encode() been called before or not. If we haven't called it then stego.bmp doesn't exist. Error message is printed.

Line 73-77: Read from stego.bmp at 0x2D, where we saved the content's size when we called Encode() earlier, and assign the content's size to 'content_len' variable.

Line 79-93: Loop 'content_len' times to read one character at a time and output to the terminal.

Now, let's explore copyBMP.c.

```c
typedef struct
{
    WORD h0;
    DWORD h1;
    WORD h2;
    WORD h3;
    DWORD h4;
    DWORD h5;
    LONG h6;
    LONG h7;
    WORD h8;
    WORD h9;
    DWORD h10;
    DWORD h11;
    LONG h12;
    LONG h13;
    DWORD h14;
    DWORD h15;
}BITMAPHEADER;

typedef struct
{
    BYTE red;
    BYTE green;
    BYTE blue;
}PIXEL;
```

I defined 2 structures, BITMAPHEADER and PIXEL, to easily indicate the size to read and write to file.
Recall from BMP file format, the header costs 54 bytes.

In the BITMAPHEADER structure, a WORD, DWORD and LONG take 2, 4, 4 bytes respectively. And we have 5 WORDs (10 bytes), 7 DWORDs (28 bytes), and 4 LONGs (16 bytes), altogether 54 bytes.
* Note: calculate *sizeof(BITMAPHEADER) = 56*. Because although *h0* is declared 2 bytes, it's actually reserved 4 bytes by GCC Compiler.[3]

In the PIXEL structure, there are 3 bytes of RGB.

[3] https://en.wikipedia.org/wiki/Data_structure_alignment

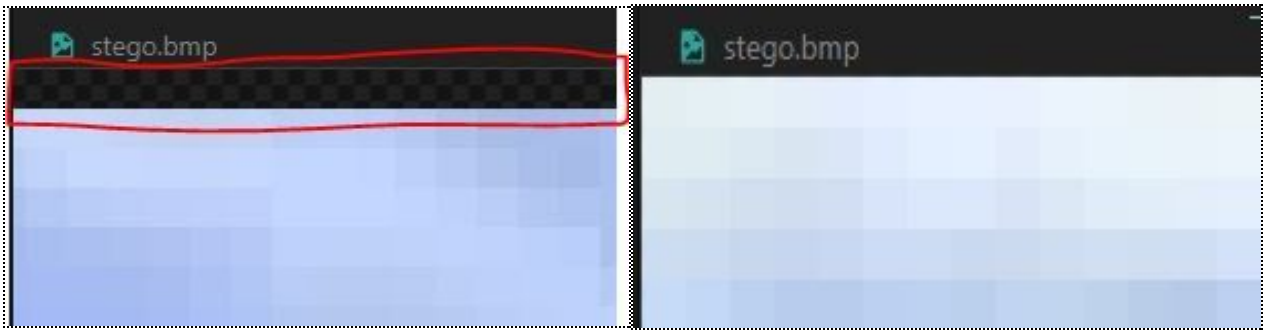
```c
void copyBMP(FILE* original, FILE* stego){

    // FILE *original, *stego;
    // original = fopen("./origin.bmp", "rb");
    // stego = fopen("./stego.bmp", "wb");
    // printf("%lu", sizeof(BITMAPFILEHEADER));

    BITMAPHEADER bf;
    fread(&bf, sizeof(BITMAPHEADER), 1, original);
    fwrite(&bf, sizeof(BITMAPHEADER), 1, stego);

    PIXEL pixel;

    while(1){
        //COPYING ORGINAL CONTENTS TO STEGO --RESULT IS CLOSE
        if ( fread(&pixel, sizeof(PIXEL), 1, original) < 1 )
            break;

        fwrite(&pixel, sizeof(PIXEL), 1, stego);
    }

    // fclose(stego);

    /*TO COMPENSATE THE LAST BYTE*/
    char nul = 0x00;
    // stego = fopen("./stego.bmp", "r+b");
    fseek(stego, 0x00, SEEK_END);
    fwrite(&nul, 1, 1, stego);
}
```

"copyBMP()" function takes in 2 file pointer arguments, first is the original file and second is the file that will be the duplicate of the first file.

Line 44-46: Copy BMP file header of original file, 56 Bytes, to stego file.

Line 48-56: Reading datas from original file and writing them to stego file in pixel (3 bytes each loop). Because we copied the header 2 bytes longer than expected it breaks off a pixel and 1 byte is leftover. The loop breaks when attempting to copy that last byte.

Line 61-64: Copy the remaining last byte to stego.



(Without the last byte) (With the last byte)

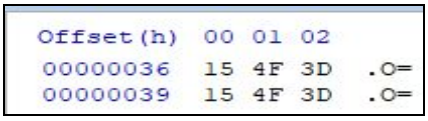
At first I thought the last byte was not really that impactful to the image but as I looked closely, I spotted a big difference in the top row.

III. Encode

As I played around with the RGB colors I notice that varying the last 4 bits, a nibble, doesn't differentiate much of that color. Therefore I decided to try the half byte approach for this assignment.

A. Half-Byte Approach

A character has 1 Byte or 2 hexadecimals. The idea is to split a full byte to 2 half-bytes. For example 'H' is presented by 0x48, and we want to split it to 0x04 (high) and 0x08 (low). Then overwrite the stego file with our splitted nibbles.



```
Offset(h)  00 01 02
00000036   15 4F 3D   .O=
00000039   15 4F 3D   .O=
```

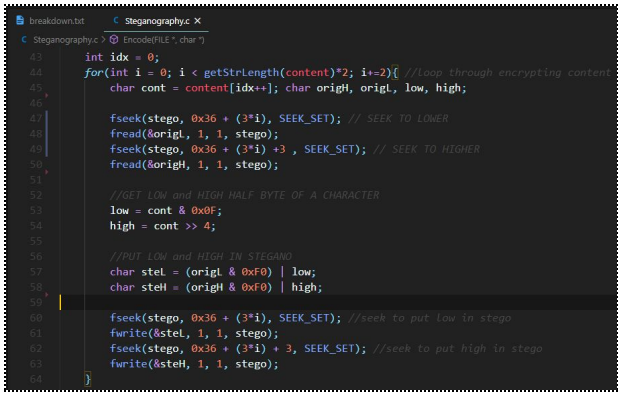
binary form of 'origin.bmp'.

At offset 0x36 and 0x39's pixel color are #3D4F15 ■
If we change 0x36 to #3D4F18 ■ (8 from low)
0x39 to #3D4F14 ■ (4 from high)
Well, the 3 colors look exactly the same to our eyes.
(Note the 4 left most bytes are identical to original)

We've encoded character 'H' to stego.bmp by overwriting 2 pixels. Now let's see how we got it to work. First, in the example, we need to get *high=0x04*, and *low=0x08* from 'H'=0x48.

- To get *low,* boolean operation **&** will give us the result. **0x48 & 0x0F = 0x08**.
- To get *high,* we simply shift 0x48 left four times. **0x48 >> 4 = 0x04.**
- To get *0x18*, first we **0x15 & 0xF0 = 0x10 | 0x08** (*low*) **= 0x18.**
- To get *0x14*, we only need to operate | (*or*) the result from **&** (*and*) with *high*.

B. A Deeper Look With Code

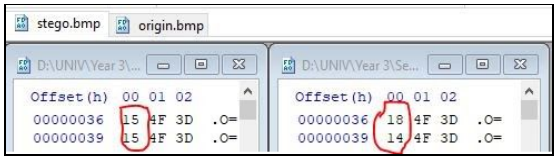


```
43      int idx = 0;
44      for(int i = 0; i < getStrLength(content)*2; i+=2){ //Loop through encrypting content
45          char cont = content[idx++]; char origH, origL, low, high;
46
47          fseek(stego, 0x36 + (3*i), SEEK_SET); // SEEK TO LOWER
48          fread(&origL, 1, 1, stego);
49          fseek(stego, 0x36 + (3*i) +3 , SEEK_SET); // SEEK TO HIGHER
50          fread(&origH, 1, 1, stego);
51
52          //GET LOW and HIGH HALF BYTE OF A CHARACTER
53          low = cont & 0x0F;
54          high = cont >> 4;
55
56          //PUT LOW and HIGH IN STEGANO
57          char steL = (origL & 0xF0) | low;
58          char steH = (origH & 0xF0) | high;
59
60          fseek(stego, 0x36 + (3*i), SEEK_SET); //seek to put low in stego
61          fwrite(&steL, 1, 1, stego);
62          fseek(stego, 0x36 + (3*i) + 3, SEEK_SET); //seek to put high in stego
63          fwrite(&steH, 1, 1, stego);
64      }
```

Looping each character of the encoding content and read the first byte of 2 pixels, line 47-50.

If *'H'* is to be encoded, then *cont = 0x48.*
origL = 0x15 (first byte of offset 0x36) and *origH = 0x15* (first byte of offset 0x39).
Line 53-58: Apply the logics described above.

Line 60-63: write the encoded data back to their positions.



The two files after encoding ‘H’. (*left: original.bmp, right: stego.bmp*)

IV. Decode

To decode, we just have to reverse engineering the logics of encode. First, we read the first byte of the 2 pixels starting at offset 0x36 and 0x39 from stego.bmp.
In the example would be, *low=0x18* and *high=0x14*.

- If we shift *high* left four times, *high=0x40*.
- Next mask *low = 0x18 & 0x0F to get 0x08*.
- Last step: (or) *high | low, 0x40 | 0x08,* to get *0x48* which is equivalent to ‘H’

V. Conclusion

From doing this assignment, I have clearly understood one of the concepts of steganography by ciphering text into a bmp file, and deciphering back. I also learned about how BMP files are formatted, a better understanding of C FILE, and the reading of binary data of an image. Also, this assignment triggers my curiosity in steganography with audios, or videos and motivation in computer security.