~ Created By Kip ~

Key Features

Ultimatley, we are taking a 8 bit 32 color palette image and duplicating or creating similar palettes to fill out the 256 color range.

With this, we have a toal of 8 similar or duplicate palettes.

Embedding:

Using the pixel data, whichever palette is being pointed to will indicate 3 bits of a hidden message (in MSB).

For example: If a pixel = 5A (01011010),
010|11010 -> the hidden message bit is 2 (010), and the color being referenced in the 26th color in any palette (11010).

Logic used:

- Read the origional cover pixel,
- Clear out the 3 MSB to prepare to embed 3 message bits
- Create and dummy byte and move message bits to MSB
  Add the dummy byte and prepared pixel data together
- Place in correct pixel index
  Iterate through message bits

(extracting is the same idea in reverse)

Code:

```
for (size_t i = 0; i < bitArraySize; i++) {
        // masking og data
        maskedByte = coverData[1078 + i] & 0x1F;
        // assigning hidden message data
        hiddenMD = (BYTE)bitArray[i];
        // shifting hidden message data to 3 MSB
        xorMD = hiddenMD << 5;
        // assigning hidden message data in MSB of pixel data
        maskedByte = maskedByte ^ xorMD;
        //printf("%02x ", maskedByte);
        coverData[1078 + i] = maskedByte;
```

Duplucate/Similar Palettes AND Palette Randomization:

We duplicate the original palette by first saving it to an array of RGBQUAD values.

```
    // Seek to the start of palette data
    fseek(fptr, 0x36, SEEK_SET);

    // Create a new palette with only 32 unique colors
    RGBQUAD newPalette[256];
    for (int i = 0; i < 32; i++) {
        newPalette[i] = ptrPalette[i];
    }
```

Then, to create the other 7 duplitcates- we step through the array and assign ith value to the i%32 value. (This is how we ensure palettes are in groupings of 32)

If we want to create 7 similar palettes (NOT duplicates), this is where the rnd and B value make a difference. g_mask[B] will create a bit mask over the rnd value, this bit mask affects how many bits are randomized in the similar palates (where B determines 0-8 bits randomized).
When B=0, the default, the palettes are only duplicates.

The bitmasked rnd value is XOR'd with the original palette data to replace B number of LSB's with random bits.

```
    for (int i = 32; i < 256; i++) {

        newPalette[i] = newPalette[i % 32];

        rnd = rand();
        //if(err != 0) {printf("error in rnd\n");}

        rnd = rnd & g_mask[B];

        newPalette[i].rgbRed = newPalette[i].rgbRed ^ rnd;
        rnd = rand();
        rnd = rnd & g_mask[B];
        newPalette[i].rgbGreen = newPalette[i].rgbGreen ^ rnd;
        rnd = rand();
```

```
        rnd = rnd & g_mask[B];
        newPalette[i].rgbBlue = newPalette[i].rgbBlue ^ rnd;
```

Reading in and separating bits to sets of three:

To hide, to do the bit manipulation we needed the message binary to be split into sets of 3. To accomplish this we first converted the whole message binary to a bit array.

Logic Used:

- While the binaryString isn't ended with a null terminator
    - Set the current char variable the index of binaryStr - '0' (i dont understand what this does personally)
    - Shift the msb of the current byte by one and or it with current bit (works like adding but taking into account what I think of a 'magnitude')
-
    - Once three bits are stored in an index, increase index
    - Repeat until null terminator reached

```
void binaryToBitArray(const char *binaryStr, char *bitArray) {
    size_t bitArrayIndex = 0;
    char currentByte = 0;
    int bitCount = 0;

    for (size_t i = 0; binaryStr[i] != '\0'; i++) {
        if (binaryStr[i] == ' ') {
            continue;
        }

        char bit = binaryStr[i] - '0';
        currentByte = (currentByte << 1) | bit;
        bitCount++;

        if (bitCount == 3) {
            bitArray[bitArrayIndex++] = currentByte;
            currentByte = 0;
            bitCount = 0;
        }
    }

    // If there are remaining bits, pad them with zeros on the right
    if (bitCount != 0) {
        currentByte <<= (3 - bitCount);
```

```
        bitArray[bitArrayIndex++] = currentByte;
    }
```
To extract, we utilized the bitArrayToBinary function.

```
void bitArrayToBinary(const char *bitArray, int bitArraySize, unsigned
char *binaryStr) {
    int binaryStrIndex = 0;

    for (int i = 0; i < bitArraySize; i++) {
        char currentByte = bitArray[i];

        // Extract each bit from the 3-bit group
        for (int j = 2; j >= 0; j--) {
            char bit = (currentByte & (1 << j)) ? '1' : '0';
            binaryStr[binaryStrIndex++] = bit;
        }
    }
    binaryStr[binaryStrIndex - 1] = '\0';
}
```

Tests To Run

There are a lot of different files provided for both hiding in and hiding. Here are the key things we want to showcase:

- Message > Cover Imgae Capcity
- Message < Capacity (may want to showcase different percentages)
- Hiding in Files with different types of backgrounds (i.e. white, plain,complex)
    - This can be analyzed for perceptability
    - Analyzing how using the -b option (1-8) may change perceptability visually in the image vs visually within hex analysis
    - Analyzing the perceptability of the embedded length depending on the background, size of the hidden message, and size of the cover image.

- Ways to analyze:
    - Visual Analysis

    - Histogram Analysis
    - File Compare binary from the command line

- Questions You May Want Ask Youself:
    - Are the hidden files ever degraded?
    - When extracting a file above image capacity, are there any issues when opening it? Which ones?
    - Which types of images are most affected by the hidden data, the least?
    - Does the type of file being hidden have an effect on the above question?
    - Are there any patterns that occur when randomizing larger amounts of the palette data?
    - How does randomixing the similar palette data visually affect the cover images. What situations make it the most notable? The least?
    - When in the embedded message length most percievable? The least?

Questions and tests suggestions MAY NOT BE comprehensive, this would be my approach if I were to test and analyze this program.