6: Steganography

CSCI 2212 Fall 2015

1 Goals

- To use a class that I wrote.
- To use a pointer to process an array.
- To use bit operators (`&`, `|`, `<<`, $\sim$).
- To use a binary input file and the `read()` function
- To use a binary output file and the `write()` function
- To learn about a strange and interesting kind of cryptography.

2 The PGM Data Structure and the Input FIles

PGM is a simple format for black-and-white photographs. My PGM class reads a photograph and stores it in the format below. The class provides several functions, as well.

```
typedef char pixel;
class PGM {
    char filetype[3];  // Should be "P5".  Please validate!
    int width;
    int height;
    int maxgrey;       // Should be 255, but no need to check it.
    pixel* image;      // Dynamically allocated array of width*height pixels.
};
```

2.1 The Hidden Messages

The two low-order bits of each pixel will be used to hold two bits of the hidden message. Thus, four pixels will produce one byte of message, and a hidden message cannot be longer than one quarter of the host photo. It could be much shorter. Each hidden message, therefore, must start with the length of the file that is embedded in the photo. This would be a long int (4 bytes) hidden in the first 16 pixels of the photo.

2.2 The Data Files

I will provide four data files in the .pgm format. Their names are:

- `poppy12.pgm`
- `suns.pgm`
- `midday.pgm`
- `nest.pgm`

The files provide four different kinds of tests:

- One has a complete book (old and well known) embedded in it.
- One has an .html file in it.
- One has an embedded photograph (pgm) that itself contains a complete .jpg photo.
- One is just an unmodified .pgm image with no hidden message.

2.3 What to Do and What to Turn In

Your job is to write a program to extract an embedded message from a .pgm file. Once extraction is done, you need to look at the text inside the extracted file and figure out what type of file it is. Look at the first line. If it starts with P5, it is another .pgm file. If it says "HTML", it is a .html file. If you can read the whole file, it is a .txt file. Rename the file to properly see the .html and .jpg files.

Hand in your extracted files, your .cpp and .hpp files, and screen shots from your test runs. The screen shots should tell me which file did not contain a hidden message.

3 Implementation

Start with the PGM class (.hpp and .cpp) that we looked at in class.

3.1 Add Membersto the PGM Class.

- Keep the PGM constructor and destructor.
- Keep or delete the functions `write`, `printHead` and `reverse`, whichever you please.
- Add a pixel* named `start`, which will be a cursor for processing the pixels.
- Add the functions `extract` and `extract4`, described below.

3.2 Do the following things in your main function:

- Ask the user to enter the name of a .pgm input file. Declare a PGM object and send the filename as a parameter to the `PGM` constructor.)
- Ask the user to enter the name of a text output file. Open a binary ofstream for this file.
- Call your `extract()` function with the open ofstream as the parameter. This will extract the hidden message and write the output.
- Print a termination comment.
- Look at the output file in a text editor. Change the file extension to .pgm or .html if that is appropriate.

3.3 `public void PGM::extract( ofsteam& )`

Write a function named `extract` in the PGM class, as follows:

- The number of pixels in the PGM photo can be calculated: `size = length*width`.
- Read 16 pixels of input. From this, extract `len`, the length of the OUTPUT file. This will be a long int, (4 bytes) stored two bits per pixel.
- This must be true: `len <= (size-16)/4` because it takes four pixels to make one byte of output. If this is NOT true, there is an error. Either there IS NO embedded message in this photo or the input file is corrupted. Abort with an appropriate message.
- If the expected output length is OK, set a pixel* to the 17th pixel.

- Now enter a loop that executes `len` times, once for each byte in the output message. Each time around the loop you will use the next four pixels in the pixel array and write one byte to the output file.
  - To extract a byte from the pixel array, call the `extract4()` function, described below. Its argument should be a pointer to the first pixel that has not yet been extracted.
  - During this process, your pixel* will be incremented by 4 pixels.
- When `extract4()` returns, write the return value to your output file. Your pixel should already be incremented by 4 pixels.
- Close the output file when extraction loop ends.

3.4 `private char PGM::extract4( pixel* start)`

At all times in this function, `start` points at the first unprocessed pixel.

- Declare a char variable named `result` and initialize it to the null character.
- Loop 4 times. Each time, do the following steps:
  - Shift `result` two bits leftward.
  - Use a mask and the `&` operator to extract the rightmost 2 bits of *start.
  - Use the || operator to move those two bits into the `result`.
  - Increment `start` and do it again.
- Return `result` from the function.