CSCI 2212: Intermediate Programming / C
Program 6: Steganography

Alice E. Fischer

November 6, 2015

Steganography

- Reading the input file.
- Extracting the Hidden Message

The Problem

In program 6, you will decode a file encrypted by a technique called "steganography".

- The data files for program 6 were, originally, photographs. I converted them from full color to grayscale and wrote them in the .pgm format.
- Then I modified the images using steganography, a form of concealed writing that could be used by spies.
- A message is hidden within the pixels of the picture in such a way that the human eye cannot tell that the photo has been modified, but the message can be easily extracted by a program.
- In our implementation, we will replace the least important two bits of every pixel by two bits from the concealed message.

The .pgm image format

The input files are in .pgm format, which has an ascii header followed by the pixels of the image, as a binary file. The file structure is:

```
typedef unsigned char pixel;
typedef struct {
    char filetype[3];
    int width;
    int height;
    int maxgrey;
    pixel* image;
} PgmType;
```

Reading a .pgm file in C

Reading a mixed text/binary file will be easier in C than in C++.

- Open the file as a text file and read the header as normal text, using `fscanf( instream, "%2s%d%d%d", ...);`
- Verify that the file starts with the code "P5", indicating a .pgm file.
- Calculate the number of pixels using the width and height of the picture.
- Then skip trailing whitespace up to the newline using `while (fgetc(instream) != '\n');`
- Allocate the right number of bytes for the image.
- On a Unix system, read all the pixels into your allocated area using a single `fread()`.
- For Windows, see the next slide.

Reading a .pgm file in C

In Windows (not necessary in Unix or Mac), after reading the header, do this:

- Declare `fpos_t filepos;`
- Find out how many bytes of header text have been processed by calling `fgetpos( instream, &filepos );`
- Use `fclose(instream)` and `fopen(instream, "rb")` to close and reopen the file in binary mode.
- Use `fsetpos( instream, &filepos )` to skip over the bytes of the header and position the file pointer at the first pixel of the image.
- Read all the pixels into your allocated area using a single `fread()`

Decoding the Secret: `getbits()`

Now the bytes of the image can be processed one at a time to pull out the secret message, if any.

Write a function `int getbits( pixel p );`

- Mask off and return the low order two bits of the current pixel. They will become the next two bits of the reconstructed message.

Implement this as an inline function in C++.

Decoding the Secret: `extract()`

Write a function `char extract ( pixel* image );`

- Declare a static local int variable to remember the current position in the pixel array (initially 0).
- Get the current pixel and postincrement the position variable.
- Declare a local variable curChar and set it to `'\0'`.
- Repeat this 4 times:
  - Shift the curChar 2 bits leftward to make spaces for the next two bits.
  - Call `getbits()` to get the next 2 bits.
  - Bitwise-OR the bits into the curChar .
- Now you have a char that can be returned.

This function needs a static variable to remember, from call to call, the position of the pixel it is working on. If it were not static, the position would be reinitialized for each function call.

Decoding the Secret: `decode()`

Write a function `void decode( pixel* image );`

- Set the message length to 0 and open an output file for the message.
- Decode the 32-bit integer that tells you how long the message is. Set the length initially to 0 and repeat this 16 times:
  - Shift the length 2 bits leftward to make spaces for the next two bits.
  - Call `getbits()` to get the next 2 bits.
  - OR the bits into the length.
- Now you know how many bytes are in the hidden message. Do the extraction process that many times.
- After each extraction, write a byte to the output file.