CS 451/551 Project 1: Steganography (version 2020.01)

February 6, 2020

1 Introduction

This document first provides the aims of this project, followed by a discussion of its background. It then lists the requirements as explicitly as possible. This is followed by an example which should help understand the requirements. Finally, it provides some hints as to how those requirements can be met.

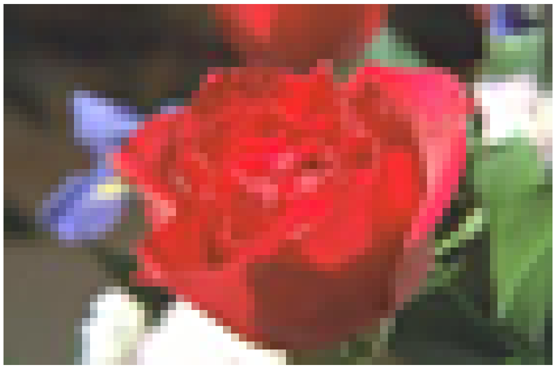
1.1 Aims

The aims of this project are as follows:

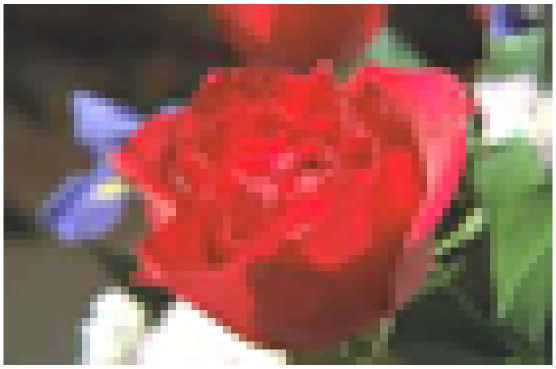
- To get you to write a simple but non-trivial Rust program.
- To make you understand the representation of unsigned integers/bytes and the use of bit twiddling.
- To allow you to familiarize yourself with the tools you will be using in this course.
- To introduce you to steganography.

1.2 Background

Ultimately, all data in computers is represented using binary numbers. Various mappings (known as encoding schemes) are used to map high-level data like a string, number or instance of an object class to the binary numbers within a computer.



(a) An unmanipulated picture of a rose.



(b) A picture of a rose that has a secret message embedded in it.

In C, strings are merely an abstraction of a sequence of bytes in memory with the last byte in the sequence having value ‘0x00‘. The values of the non-zero bytes represent characters using some encoding like ASCII or UTF-8.

For example, using ASCII the 5-character C string “hello” represents the sequence of the following 6 bytes:

```
0x68 'h', 0x65 'e', 0x6c 'l', 0x6c 'l', 0x6f 'o', 0x00 '\0',
```

Although Rust has a more advanced String type than C, it is important to note that ultimately, all data is just a sequence of bytes. A particular sequence of bytes like those above, when associated with an encoding like ASCII can be interpreted as a textual string. OTOH, the same sequence of bytes could be interpreted as the binary data bytes of an image or the instructions of a program. So the meaning of a sequence of data bytes depends on how they are being interpreted in the current context.

Steganography (`http://en.wikipedia.org/wiki/Steganography`) is the art or practice of concealing a message, image, or file within another message, image, or file. A simple example of steganography in the physical world is using “invisible ink” to conceal a secret message in a normal-looking letter. A historical use was by a US POW (`http://en.wikipedia.org/wiki/Jeremiah_Denton`) “blinking out” a secret message using Morse code when forced to participate in a propaganda video.

In this project, we will use Rust’s bit-twiddling (`http://en.wikipedia.org/wiki/Bit_manipulation`) to conceal a ASCII string message within a PPM image file (`http://en.wikipedia.org/wiki/Netpbm_format`). The inefficient PPM format was chosen over more popular and practical formats like GIF or PNG as it is extremely simple to understand and manipulate.

The PPM format allows easy steganography as random changing of less-significant bits do not have any easily visible effect on the displayed image. For example, see Figures 1a and 1b.

2 Requirements

Create a new cargo project called “project01”, and change the name of the outputted binary to be “steg.” Submit a **gzipped** version of your project directory, such that `cargo run` in the decompressed file will build and execute your program. **Make sure that you run `cargo clean` before compressing your directory and submitting!**

It should be possible to invoke the ‘steg‘ program with upto two arguments:

- When invoked with zero arguments, the ‘steg‘ program should simply output a usage message on standard error.
- When invoked with one argument, the ‘steg‘ program should unhide the message concealed in the PPM file named by its first argument and print it on standard output followed by a newline character.
- When invoked with two arguments, the ‘steg‘ program should hide the message specified by its second argument in the PPM image file named by its first argument and write the contents of the resulting image on standard ouput.

The program should detect errors in its inputs, including things like trying to hide a message that is too big to fit into the destination file, input files not being found, etc. It should also terminate with an error message if it runs out of memory.

3 Understanding Hidden Message in an Image

The image contained in ‘aux/out/rose-hello.ppm‘ contains the hidden message “hello”. Let’s dump out the contents of the initial portion of this file and try to understand how the message is concealed within it:

```
$ od -N 128 -t x1 -c ~/cs551/projects/prj1/aux/out/rose-hello.ppm
0000000  50  36  0a  37  30  20  34  36  0a  32  35  35  0a  30  2f  2d
          P   6  \n   7   0       4   6  \n   2   5   5  \n   0   /   -
0000020  33  30  2e  37  33  2e  39  33  2f  3a  33  2c  38  32  2d  39
          3   0   .   7   3   .   9   3   /   :   3   ,   8   2   -   9
0000040  30  2c  39  31  2f  38  31  2d  38  31  2c  36  2e  2c  35  2d
          0   ,   9   1   /   8   1   -   8   1   ,   6   .   ,   5   -
0000060  2a  34  2d  28  35  2c  2b  35  2c  2b  31  2c  26  30  2f  27
          *   4   -   (   5   ,   +   5   ,   +   1   ,   &   0   /   '
0000100  34  31  2b  36  34  2c  39  37  2e  3f  3b  2f  47  3e  32  4a
          4   1   +   6   4   ,   9   7   .   ?   ;   /   G   >   2   J
0000120  42  34  4c  40  32  4e  42  32  55  41  32  74  44  34  9a  43
          B   4   L   @   2   N   B   2   U   A   2   t   D   4 232   C
0000140  33  b4  41  35  c5  45  3d  e0  44  47  ed  43  46  f6  3d  42
```

```
          3 264    A    5 305    E    = 340    D    G 355    C    F 366    =    B
0000160  f1  3c  40  d6  40  3b  b8  3f  2f  b2  3f  2d  ad  40  2d  a3
        361   <   @ 326   @   ; 270   ?   / 262   ?   - 255   @   - 243
0000200
$
```

The dump contains the first 128 bytes of the file, 16 bytes per line. Each byte is dumped out in both hex as well as characters (when it corresponds to a ASCII character).

Looking at the characters, we clearly see the image header:

```
P6
70 46
255
```

The pixel data start with the byte with value ‘0x30‘ after the ‘ n‘ after the 255.

Extracting the message stored in the pixel bytes is easy: starting with the first pixel byte, segment the stream of data bytes into groups of 8 and then extract the message bits from the LSB of each byte; if the value of the byte is even, the the message bit is 0; if it is odd then it is 1.

So we have:

```
-------------------------------------------------------------------
          Pixel Bytes              |    Binary   |   Hex      Char
-----------------------------------+-------------+---------+--------
  30  2f  2d  33  30  2e  37  33   |  0111_0011  |   0x73  |  's'
  2e  39  33  2f  3a  33  2c  38   |  0111_0100  |   0x74  |  't'
  32  2d  39  30  2c  39  31  2f   |  0110_0111  |   0x67  |  'g'
  38  31  2d  38  31  2c  36  2e   |  0110_1000  |   0x68  |  'h'
  2c  35  2d  2a  34  2d  28  35   |  0110_0101  |   0x65  |  'e'
  2c  2b  35  2c  2b  31  2c  26   |  0110_1100  |   0x6c  |  'l'
  30  2f  27  34  31  2b  36  34   |  0110_1100  |   0x6c  |  'l'
  2c  39  37  2e  3f  3b  2f  47   |  0110_1111  |   0x6f  |  'o'
  3e  32  4a  42  34  4c  40  32   |  0000_0000  |   0x00  |  '\0'
-----------------------------------+-------------+---------+--------
```

Hence the image contains the string ‘”stghello”‘ followed by a ‘NUL‘-terminator.

4 Hints

There are several ways to accomplish this project, and so you are free to take or ignore the following advice at will.

- The `aux.zip` file expands to an `aux` directory, which has subdirectories of sample input images as well as a sample output with “stghello\” as the hidden message.

- Remember that in Rust, `char`s are *not* 8-bit bytes, they are actually 4 bytes wide to handle unicode. It is probably a good idea to think of things not in terms of characters, but in terms of bytes. What primitive Rust type is equivalent to a byte?

- Making use of pattern matching here and there might make your life a bit easier.

- Remember that we can use the `?` operator on a `Result` to save us some time with respect to handline errors.

- My solution is approximately 200 lines, including white space and comments.

- Although dealing with binary data might look daunting, PPM is relatively easy to handle.

- Remember you can print to standard err with the `eprintln!` macro, but be careful about using it and `println!` to print out bytes; you might get unexpected output because of Unicode.