Hardware-Software Interface
Coursework 1- Image Steganography in C

Alistair McConnell

Based on Coursework by Adam Sampson

School of Mathematical and Computer Sciences, Heriot-Watt University

Overview

In this coursework, you will demonstrate your ability with the C programming language and its standard library by writing a steganography program which encodes and decodes secret messages in bitmap images.
This coursework contributes to the following learning outcomes of the course:

- Ability to develop efficient, resource-conscious code.
- Practical skills in low-level, systems programming, with effective resource management.
- Ability to articulate system-level operations and to identify performance implications of given systems.

This is an **individual** assessment. The code and documentation you submit must be entirely your own work.
If you have any questions, please ask me **<alistair.mcconnell@hw.ac.uk>.**

The problem

This coursework involves writing a simple steganography program in C, called `steg`. `steg` can operate in two modes — encode and decode — selected by the first command-line argument being e or d.
An RGB colour bitmap image consists of a grid of pixels, each of which has red, green and blue colour values. To **encode** text inside an image, your program will replace the red value in successive random pixels in the image with characters from the text, outputting a new image.
When encoding, your program will be invoked as:

```
./steg e old.ppm >new.ppm
```

It must prompt for a message to encode, and output the new image to **`stdout`** (in this case we have redirected it to a file).
To **decode** the text, your program will compare the new image with the old image, and extract characters from the new one where it differs from the old one.

When decoding, your program will be invoked as:

```
./steg d old.ppm new.ppm
```

It must decode the message and output the hidden text to `stdout`.

The PPM image format

You will work with **Plain PPM** format images. This is one of a family of simple open source image formats, designed to be read and written easily by C programs. See the PPM specification for full details of PPM and Plain PPM.

A Plain PPM file consists of ASCII text:

```
P3
# comment1
# ...
# commentN
width height
max
r1 g1 b1
r2 g2 b2
r3 g3 b3
...
```

where:

- `P3` - code indicating Plain PPM format
- `comment` - arbitrary optional comment text
- `width` - integer number of columns
- `height` - integer number of rows
- `max` - integer maximum colour value - usually 255
- `ri gi bi` - integers between **0** and **max** for pixel i's red, green and blue values

All integers are in decimal.

(From now on, I will refer to Plain PPM just as PPM.)

Getting started

You will need a Linux C development environment, like we have been using in the practical exercises. This might be:

- a MACS lab machine
- a MACS machine remotely, using x2go
- the MACS Linux Virtual Machine on your own computer
- a regular Linux installation on your own computer

You may like to refer to Hans-Wolfgang Loidl's Linux Introduction.

Cloning the project

You are provided with a template project containing an outline source file. You must use Git and the MACS GitLab Student server to download the template project and to submit the assessment.

First, fork this project.

Then clone your fork on your Linux system, e.g. (replacing username with your **username**):

```
git clone git@gitlab-student.macs.hw.ac.uk:username/f28hs-2022-23-cwk1-
c.git
```

Make sure that you can compile the starter code before you start making changes to the `steg.c` file. Change into the project directory, and run:

```
Make
```

(This may print some warnings because the code is incomplete.)

As you progress through the coursework requirements, push your work to GitLab. Remember to create small incremental improvements with Git commits — you may like to refer to Rob Stewart's Introduction to Git video for the lifecycle of files in a Git repository. When you push your commits, GitLab will check that your file can be compiled.

Test dataset

A collection of PPM image files for you to test your steganography program with is available from Canvas, under **Course Information > Assessment.**

You may notice some PPM files in this collection have a new line after every colour value, whilst others have the R, G and B values for a pixel on one row. Either is fine — the PPM specification allows any kind of whitespace between numbers.

Requirements

This project as a whole is marked out of 20 points, and makes up 50% of your final mark for the course. The program overall should be valid C99 (or later), and should be robust against error.

Work through the steps below in order.

1 - (2 points) Storing PPM images

In `steg.c`, complete the declaration of struct PPM, which holds the image read from a PPM file. This will need to include the information from the PPM header (`width`, `height`, `max`), and a pointer to a dynamically-allocated array containing the pixel data. Your program should be able to deal with PPM files with arbitrary numbers of rows and columns.

2 - PPM functions

Implement in `steg.c` the following functions.
You may also write additional helper functions or type declarations if you like. The template includes some already, e.g. `struct Pixel` and `readPPM`, but you can rename or remove these provided you implement `struct PPM`, `getPPM`, `showPPM`, `encode` and `decode`.

2a - (2 points) Reading PPM files

```
struct PPM *getPPM(FILE *f);
```

To return a new `struct` PPM, containing the PPM image read from open file `f`.
Use the `fscanf` function to read numbers from the file. Once you know `width` and `height`, construct a dynamic array of the appropriate size. Don't worry about handling comments in the PPM file for now.

2b - (2 points) Writing PPM files

```
void showPPM(const struct PPM *img);
```

To output the PPM image `img` to `stdout` (e.g. by using `printf`), following the syntax in the PPM specification.
You should test this function by using it to write out an image loaded with `getPPM`. The template code's `main` function has an extra `t` mode for this.

2c - (2 points) Encode data

```
struct PPM *encode(const char *text, const struct PPM *img);
```

To return a copy of PPM image `img` with message `text` hidden in its red pixel values.
You will need to make it replace successive random red pixels with the characters from the message. How you select pixels is up to you, but make sure that the pixels are chosen in a consistent order (e.g. left to right, top to bottom) and enough pixels are selected to encode all of the message.

2d - (2 points) Decode data

```
char *decode(const struct PPM *oldimg, const struct PPM *newimg);
```

to return a new string containing the message hidden in the red pixel values of PPM image `newimg`, by comparing it with the red pixel values of PPM image `oldimg`.
The length of the string that it allocates and returns will depend on the length of the message. You may impose a limit on the maximum length of a decoded message if this makes the implementation easier.

3 - (6 points) Steganography program

Complete the main behaviour of the `steg` program, which encodes or decodes images based on its command-line arguments as described above.

4 - (2 points) Additional PPM features

For an additional 2 points, implement one of the following two features:

- Make `getPPM` and `showPPM`read and write comments, by storing the comments as a linked list of strings in `struct PPM`. You can assume that any comment lines will always be immediately before width (although the PPM specification says they're allowed to occur anywhere in the file). You will need to rework how you read `width` — if the first character you read is a **#,** then you have a comment, else it must be a digit and should be counted as part of `width`.
- Make `getPPM` handle normal PPM files as well as Plain PPM. Normal PPM files start with P6 instead of P3, and have the image data stored directly as bytes (one or two bytes per colour, depending on `max`) rather than as decimal numbers — see the PPM specification for more details. You can read data like this using the `fread` function.
-

5 - (2 points) Report

In `Report.md`, you should give a **brief** description — no more than a couple of pages of text — of:

- Your program design
- Your choice of data structures and algorithms

Write Report.md using Markdown syntax, which will allow GitLab to render it nicely as HTML. See the Markdown Guide, or the help within GitLab, for more details.

Submission

Submission is due by 15:00 on Friday 24nd February through the MACS GitLab Student server.

To submit coursework 1, you must **push** your code to your fork of `f28hs-2021-22-cwk1-c` on the MACS GitLab Student server.

Your project on GitLab must include:

- `steg.c` - your implementation of image steganography.
- `Report.md` - your report describing your implementation.

Plagiarism Policy

This coursework is individual work. It is not group work, and you are assessed individually. Therefore, the code and report you submit for this lab must be entirely your own. You must not share your code with other students, and you must not copy code solutions from others. The university plagiarism policy is clear:

"Plagiarism involves the act of taking the ideas, writings or inventions of another person and using these as if they were one's own, whether intentionally or not." Definition 2.1.

We run plagiarism detection software against **all** source code submissions to identify any code similarity across multiple submissions. The minimum disciplinary action for plagiarism is an award of an F grade (fail) for the course, and serious instances of plagiarism will result in compulsory withdrawal from the university.

Hints

- The PPM specification is usually available as a manual page on Linux — try `man 5 ppm`.
- You should check that the output image looks reasonable. Many image viewers and editors are capable of reading PPM files, e.g. shotwell, eog, gimp, feh, xli...
- You can use od to look at the raw bytes inside a PPM file. For example: `od -x -a ape.ppm | more`
- C's `<stdlib.h>` includes a basic random number generator. The `srand(seed)` function initialises the random number generator with a given unsigned integer, then subsequent calls to `rand()` will return a number in the range `0` to `RAND_MAX`.. You can get a value in the range 0 to N - 1 with `rand() % N`.
- Clue: what happens if a letter you're going to hide has the same value as the pixel component it's going to replace?