

# Assignment 1

## Steganography

**Due: 09:00am, Monday 27 March, 2017**

The first two assignments for FIT3042 this year form a single project with two phases. In the first phase, you will write two programs: one to hide a secret message in an image, and one to retrieve a secret message from an image. In the second phase, you will add extra functionality to this system, including the ability to display before-and-after images to allow the user to see the impact of the hidden message on how the image looks.

### Requirements

For Assignment 1, you must write two programs:

- **hide** must take two *command line parameters*. The first is the name of an input file, which must be in PPM format. The second is the name of an output file. This program should open the input file, read it, encode a message into it using the steganographic protocol described below, and output the resulting image into the output file.

The message should be read from the standard input (`stdin`). If the message is too long for the chosen file, the program should output a sensible error message and terminate.

- **unhide** must take one command line parameter: the name of a file that contains a PPM image with a hidden message that uses the same steganographic protocol as **hide**. It should open the file, extract the message, and display it on the screen (i.e. to `stdout`).

We won't tell you exactly how to implement this, although you should find that it's easy to figure out. There's a fairly comprehensive example below that will help you understand how to hide a string in an image.

### Things to consider

There are several possible error conditions, and you will need to handle them all gracefully. Make sure that your program never crashes – that is, if it terminates before finishing what the user expected it to do, it should always output meaningful information to `stderr` or `stdout`. For example, there's a limit to how much data you can hide in an image. Don't forget to give a meaningful error message if the user provides a message that is too long.

Please bear in mind that you will be marked on the efficiency of your solution; that is, the quality of your algorithm and the amount of memory it uses. Your marker will also be checking that you have broken your program up into well-designed and well-documented functions, that these functions have been grouped sensibly into files, and that you are using `.c` and `.h` files appropriately.

As well as your source code, you will also be expected to submit a **Makefile** and a **README** explaining how to build and use your system.

### Bonus marks

For **bonus marks**, you may also get your program to support Windows BMP (`.bmp`) files. You may use an optional command line switch to tell the program to expect a BMP, or you may decide whether you've got a BMP or a PPM by inspecting the header – it's up to you. Let your marker know that you have implemented this functionality by describing it in your **README**.

There's information in this assignment sheet about working with PPMs, but if you want to implement the bonus functionality, you'll need to read up on it independently. We suggest




that you start with the Wikipedia page on the BMP format, which you'll find at [https://en.wikipedia.org/wiki/BMP\\_file\\_format](https://en.wikipedia.org/wiki/BMP_file_format).

Please note that in order to get any marks for BMPs, you'll also need to have your program work with PPMs. We suggest that you make sure that the basic functionality is working well before you start working on the bonus – including ensuring that it has no memory leaks and can handle error conditions gracefully.

### Background: pixel maps and the PPM format

As you probably know, images displayed on a computer screen are made up of a collection of coloured dots or pixels. To represent these digitally, we use a sequence of numbers representing the intensity of red, green, and blue colouration of each pixel - by convention, that order is typically used. For most image files, 8-bit integers are used to represent colour intensity, where 0 (the smallest value that can be represented as an 8-bit unsigned integer) represents the lowest intensity, and 255 the highest. We use 8 bits for each of the three colours, so we end up using 24 bits per pixel. *8 bits x 3 colors*

This chart shows some example combinations of red, green, and blue, and the colours they create:

Color values	Colour
(0,0,0)	 (black)
(0, 255,0)	 (green)
(128, 128, 0)	 (vomit)
(255, 255, 255)	(white)

To represent an entire image in a file, all we need to do is store a sequence of colour values in a defined order, and supply a little bit of extra information about the image size and shape (either explicitly or implicitly). One very simple format for doing so is the Portable PixMap format, or PPM, used in the widely-distributed netpbm image processing tools. There are two versions of PPM - one encodes the numbers directly, the second writes the numbers as ASCII text and is known as *plain PPM*. A very small example plain PPM is below. (You'll be using standard PPM, using integers rather than ASCII strings, but this example is in plain PPM so that you can read it.)

The first line is a so-called *magic number* that identifies the file as a plain PPM file, the second line contains a comment, the third line records the width and height of the image in pixels, and the last line contains the maximum value for a channel (note that this example uses 15 rather than the more common 255).

Pixels are represented in the file left-to-right, then top-to-bottom order.

```
P3
# feep.ppm
4 4
15
0 0 0 | 0 0 0 | 0 0 0 | 15 0 15
0 0 0 | 0 15 7 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 15 7 | 0 0 0
15 0 15 | 0 0 0 | 0 0 0 | 0 0 0
```

In standard PPM, the header information is stored as an ASCII string, but the data following the last line of the header is binary rather than text. That is, instead of representing a pink pixel using the string "255 102 78", which would take up ten bytes of space, the numbers are stored in binary. This uses a lot less space – read on if you'd like to see why.

It's easiest to see how this works if you start with hexadecimal numbers rather than decimal, because each hex digit represents exactly four bits.<sup>1</sup> The hex representation of 255 is 0xFF; 102 is 0x66; and 78 is 0xB2. If we pad this to fit a thirty-two bit word size, that means our pixel is represented by the number 0x00FF66B2. In decimal notation, that's 16737970. That's only going to take up two bytes – the text version will take up five times as much space!

You will be using standard PPM, so your program will have to deal with binary data. You will need to choose your file-reading functions appropriately and figure out how to extract the bits representing the red, green, and blue colour values of each pixel.

### Background: Steganography

*Steganography* literally means 'hidden writing'. It is a method for passing messages by hiding them, rather than encoding them – in an image, sound clip, video, or a piece of other writing. After the message has been hidden, it can be sent to co-conspirators disguised as a picture of a kitten or something similarly innocent-looking.

In this assignment, we'll hide our messages in 24-bit colour PPM image files. As we've seen, these files use 24 bits to represent each pixel: 8 bits per colour. We'll hide our messages in the *least significant bit* of each colour – the rightmost bit. That will make the colour only one shade more intense (or less intense). That's a difference likely to be very hard to see with the human eye. We will be able to hide three bits of message in each pixel.

Here's an example to show you how this works. Suppose we want to transmit the message Hello, and we want to hide it in `feep.ppm` as described in the previous section. The message itself is encoded in ASCII and null-terminated, as is normal for strings in C. If we look at the bit pattern, we'll see:

letter	H	e	l	l	o	\0
ASCII	72	101	108	108	111	0
binary	01000010	01100101	01101100	01101100	01101111	00000000

The message, in binary, consists of the bits shown on the bottom row of this table. It is  $(5+1)*8 = 48$  bits long. To hide this in an image, we'll need at least  $48/3 = 16$  pixels. Fortunately, `feep.ppm` has exactly 16 pixels, so we are in luck. Its colour values only go up to 15, though. For the purposes of this explanation, we'll only show 4 bits per colour (so 12 bits per pixel) – using fewer bits will make it easier to see what's going on. Remember, though: *your* images will have 24 bits per pixel!

This table shows how the first four pixels can be made to store the first 12 bits of the message:

	pixel 1			pixel 2			pixel 3			pixel 4		
colour	red	green	blue	red	green	blue	red	green	blue	red	green	blue
original	0	0	0	0	0	0	0	0	0	15	0	15
binary	0000	0000	0000	0000	0000	0000	0000	0000	0000	1111	0000	1111
message bit	0	1	0	0	0	0	1	0	0	1	1	0
new binary	0000	0001	0000	0000	0000	0000	0001	0000	0000	1111	0001	1110
new decimal	0	1	0	0	0	0	1	0	0	15	1	14

<sup>1</sup>If you're an experienced web developer, you might have used hex codes to represent colours in the past – the hex code representing our pink pixel is #FF66B2.

## Notes

It is important that you get the basic functionality working as this will allow you to use your program in Assignment 2. You won't need BMP support for Assignment 2.

## Submission instructions

You must submit a single archive file in gzipped tar format (`.tar.gz`) through Moodle. See `info tar` (or consult the internet) for information how to create a gzipped tar archive.

You must include a working `Makefile` with your submission. It should be possible for the marker to build all the executables by changing to the root directory of your submission and running `make` at the command line. The use of `make` and `Makefiles` will be covered in lectures in Week 4.

You should include a `README` file, in text format, that describes (at a minimum):

- your name and student ID number
- how to compile and run the program
- what functionality is and is not supported
- and any known bugs or limitations.

This information is to help your marker build and run your program. That means that it is to your advantage if it is accurate: don't claim that you have implemented something that's not actually there. You'll lose marks for not following the submission instructions. Remember, you want your marker to feel happy and generous!

## Marks breakdown

Marks for this assignment are broken down as follows:

- Basic functionality:
  - `hide/unhide` work: **5 marks**
  - I/O and command line parameters: **3 marks**
  - Submission guidelines followed, i.e. `tarfile` extracts properly and `Makefile` builds: **2 marks**
- Code quality:
  - error handling: **4 marks**
  - memory management, including avoiding overruns and memory leaks: **3 marks**
  - readability/usability (modularity, use of `.c` and `.h` files, quality of `README`, etc.): **3 marks**

This assignment is out of **20 marks**. An additional **2 marks** are available for extending `hide` and `unhide` to allow them to hide messages in BMP files as well as PPMs.