Hardware-Software Interface
Coursework 1 - Image Processing in C

Adam Sampson and Alistair McConnell
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 completing a C program that performs image processing operations on image files.

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 your local course leader.

Finding your tasks

This assessment is **personalised** for each student, so different students in the class are doing slightly different tasks. The tasks are chosen deterministically for each student based on a hash of their student number and the semester.

See the list on Canvas to find out your tasks. You will find an entry like this for your student number:

```
H12345678: HS8 MEDIAN CODE
```

Note this entry down in your `README.md` file before you start work. It is important that you check you are solving the correct tasks — if you do an incorrect task, we can't give you any points for it.

The problem

This coursework involves writing a simple RGB bitmap image processing program in C, called `process`. `process` is run from the command line with two arguments, an input image file (which must already exist) and an output image file (which will be created or overwritten). For example:

```
./process kitten.image kitten-processed.image
```

The image format

An RGB colour bitmap image consists of a grid of pixels, each of which has red, green and blue

colour values. Image files consist of a header, describing the properties of the image, followed by the RGB data for the pixels of the image.

The image format is personalised; check your list of tasks and read the appropriate section below.

Task HS8

An HS8 image has an ASCII text header:

```
HS8
width height
```

where:

- `HS8` — fixed code indicating HS8 format
- `width` — integer number of columns
- `height` — integer number of rows

The fields in the header are separated by one or more whitespace (space, tab, carriage return or newline) characters. The `height` field is followed by a single whitespace character.

The image data follows the header. Each pixel is stored as three unsigned 8-bit values as binary data (not text). Pixels are stored from left to right in the first row, then left to right in the second row, and so on until the end of the image.

Task HS16

An HS16 image has an ASCII text header:

```
HS16
width height
```

where:

- `HS16` — fixed code indicating HS16 format
- `width` — integer number of columns
- `height` — integer number of rows

The fields in the header are separated by one or more whitespace (space, tab, carriage return or newline) characters. The `height` field is followed by a single whitespace character.

The image data follows the header. Each pixel is stored as three unsigned 16-bit values as binary data in native byte order (not text). Pixels are stored from left to right in the first row, then left to right in the second row, and so on until the end of the image.

(**Native byte order** means that bytes within a number are stored in the same order that they would be stored in memory on the machine you're running the program on.)

Task HSDEC

An HSDEC image consists entirely of ASCII text. It starts with a header:

```
HSDEC
width height
```

where:

- `HSDEC` — fixed code indicating HSDEC format
- `width` — integer number of columns
- `height` — integer number of rows

The image data follows the header. Each pixel is stored as three unsigned 8-bit values as decimal numbers. Pixels are stored from left to right in the first row, then left to right in the second row, and so on until the end of the image.

All fields in the file are separated by one or more whitespace (space, tab, carriage return or newline) characters.

Task HSHEX

An HSHEX image consists entirely of ASCII text. It starts with a header:

```
HSHEX
width height
```

where:

- `HSHEX` — fixed code indicating HSHEX format
- `width` — integer number of columns
- `height` — integer number of rows

The image data follows the header. Each pixel is stored as three unsigned 16-bit values as hexadecimal numbers (digits A-F may be lower or upper case). Pixels are stored from left to right in the first row, then left to right in the second row, and so on until the end of the image.

All fields in the file are separated by one or more whitespace (space, tab, carriage return or newline) characters.

Getting started

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

- a Linux 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.

Setting up the project

Download `cwk1-c-template.zip` from **Coursework 1** on Canvas. This contains a template project containing an outline source file.

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

```
make
```

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

The template project also includes a Python program, `hsconvert`, that can convert standard PPM

images to and from the HS formats, or view an HS image using `gpicview`. Run it as `./hsconvert --help` for help.

A collection of image files for you to test your program with, in all the formats above, is also available from **Coursework 1** on Canvas.

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. You may write additional helper functions and/or type declarations if you like, but you should keep the functions that are provided in the template file in the existing order.

Work through the steps below in order.

Q1 - (2 points) Representing images

In `process.c`, complete the declaration of `struct Image`, which holds an image read from a file.

This will need to include the information from the image header (`width`, `height`), and a pointer to a dynamically-allocated array of `struct Pixel`s for the pixel data. Your program should be able to deal with image files with arbitrary numbers of rows and columns.

You should change the data types inside `struct Pixel` to match your image format.

Q2a - (1 point) Freeing image objects

Complete the `free_image` function, which should free a `struct Image` and its contents.

Q2b - (2 points) Loading images

Complete the `load_image` function, which allocates a new `struct Image` and reads the contents of an image file into it.

You will need to use the `fscanf` and/or `fread` functions to read data from the file. Once you know `width` and `height`, construct a dynamic array of the appropriate size. Be careful to handle errors — the input file might not contain valid contents.

Q2c - (2 points) Saving images

Complete the `save_image` function, which writes the contents of a `struct Image` out to an image file.

You should test this function by using it to write out an image loaded with `load_image`. You can modify `main` temporarily for this.

Q2d - (2 points) Duplicating image objects

To do Q3, you will need to allocate a new `struct Image` that is a copy of an existing image. Complete the `copy_image` function to do this. It will need to copy both the `struct Image` and the pixel data within it.

Q3 - (4 points) Personalised task 1

This task is personalised; check your list of tasks and read the appropriate section. All of these tasks should return a copy of the original image — they must not modify the original image.

Task BLUR

The `apply_BLUR` function should apply a 3x3 blur to an image. Each colour value in the output image should be computed as the mean of the 3x3 block of pixels surrounding it in the input image (that is, the corresponding input pixel, and the 8 pixels immediately adjacent to it).

Be careful to avoid accessing locations outside the bounds of the image.

Task MEDIAN

The `apply_MEDIAN` function should apply a median filter to an image. Each colour value in the output image should be computed as the median of the corresponding input pixel and the four pixels horizontally and vertically adjacent to it (above, below, left and right of it).

To compute the median of a set of numbers, sort them into order and take the middle one. For example, you could copy the values into a fixed-length array, then call the `qsort` function from the standard library to sort them.

Be careful to avoid accessing locations outside the bounds of the image.

Task MONO

The `apply_MONO` function should convert an image from colour to monochrome (greyscale). To convert a colour value to an equivalently bright grey value, you must compute a weighted sum of the red, green and blue components: `0.299R + 0.587G + 0.114B` (this is because the human eye is differently sensitive to different colours). You can then set the red, green and blue components in the output image to this grey value to produce greyscale output.

As this involves both integer and floating-point maths, you will need to be careful to avoid integer wraparound (too large or too small values) when computing the output values.

Task NOISE

The `apply_NOISE` function should add random noise to the red, green and blue components of the image, using the standard library's `rand()` random number generator. First, make it add random values from -5 to +5 to each component.

Then modify the `main` function so that the program takes an extra command-line argument at the end of the command line to indicate the noise strength. If you run the program as:

```
./process in.image out.image 10
```

it should add random values from -10 to +10. You can use the `atoi` standard library function to convert a string argument to an integer.

As you are adding random noise to numbers that may also be near the limits of the numbers you can represent, you should be careful to avoid integer wraparound (too large or too small values) when computing the output values.

Q4 - (4 points) Personalised task 2

This task is personalised; check your list of tasks and read the appropriate section. All of these functions should print their output to `stdout` (e.g. using `printf`).

Task NORM

The `apply_NORM` function should first scan through the image to find the smallest and largest values used in the red, green and blue values (all together; you don't need to track red/green/blue separately). It should print out a message like:

```
Minimum value: 10
Maximum value: 255
```

It should then normalise the image by scaling all the values so that the minimum value becomes 0, and the maximum value becomes the biggest possible value that can be stored. (Do this in two steps: subtract an offset to bring the minimum to 0, then multiply by a scaling factor to bring the maximum to the largest value.)

Because this means modifying the image, you will need to remove the `const` from the type of `apply_NORM`'s image argument.

Task HIST

The `apply_HIST` function should compute a histogram of the red, green and blue values used in the image (all together; you don't need to track red/green/blue separately). It should print a message like:

```
Value 0: 123 pixels
Value 1: 55 pixels
Value 2: 8729 pixels
...
Value 255: 17 pixels
```

(If you prefer, you could show the histogram as a chart; this isn't worth any more points, though.)

You will need to use a data structure such as an array to count how many times you've seen each value.

Task CODE

The `apply_CODE` function should print out the image as C source code that could be included in a program (for example, to use as an icon in a GUI application). It should print something like:

```
const int image_width = 320;
const int image_height = 256;
const struct Pixel image_data = {
    {0, 0, 0}, {255, 0, 255}, {255, 255, 127},
    {10, 255, 0}, ...
};
```

The output should be valid, correctly-indented C code, and you should limit the length of the lines that it produces to no more than 70 characters when printing the image data. You don't need to compile the output code yourself.

Task COMP

The `apply_COMP` function should take two `struct Image *` arguments, and it should count how many pixels are different between the two images. It should print something like:

```
Identical pixels: 1827841
Different pixels: 871285
```

You will need to modify the `main` function so that it takes a single extra image argument at the start of the command line to compare with, e.g.:

```
./process reference.image input.image output.image
```

Q5 - (2 points) Bonus task

For an additional 2 points, make the program work for an arbitrary number of input/output image filenames rather than just a single one. It must load all of the images into memory before doing any processing.

You will need to use a data structure such as a linked list or a dynamically-allocated array of `struct Image` pointers to hold the list of images you are working on. You could add a field for the output filename to `struct Image`.

Q6 - (1 point) README.md

The `README.md` is a short file in Markdown or plain text format, saying clearly how to compile your program, and how to run it from the command-line showing the arguments that it takes.

It is **not** a report: you should not restate the problem or write a description of how your program works here.

Submission

Submission is due by 15:30 on Thursday 29th February through **Coursework 1** on Canvas. (If you are studying at OUC, you have a different deadline; see Canvas.)

You must submit a `.zip` file containing the project files, including at least:

- `process.c` — the source code for your program
- `README.md` — file explaining how to compile and run your program

To reduce the submission size, please don't include the sample images or other large unnecessary files in your submission.

Demonstration

You must give a demonstration of your code to one of the teaching staff during one of the lab sessions. This lets us see that your code works, and lets us ask you questions about your code. Please bring your source code and have a compiled version of the program ready to demonstrate.

Times for demonstrations will be published on Canvas near the submission date.

If you don't give a demonstration, you will not receive a mark for this coursework. If you are unable to attend the lab for this (e.g. because of illness) then please contact your local course leader to make alternative arrangements.

Hints

- Your program should be as efficient as possible — particularly when processing image data. You should document any decisions you have made that improve your program's efficiency in the comments for each section of code.
- Your program should also be robust — for example, it shouldn't crash when given an incorrect

or corrupt input file, or if the computer runs out of memory. Remember to check the return values of the standard library functions you are using and do something sensible if they fail.

- You are working with data types that can represent a limited range of values (e.g. unsigned integers). When operating on pixel RGB values, be careful not to go out of the valid range of values — this would look bad in the resulting image.
- Be careful about memory leaks, array bounds violations and other types of memory error. You may like to use a dynamic analysis tool such as Address Sanitizer or Valgrind to check your program.
- The template code has various `TODO` comments in to indicate the things you need to change. Make sure you remove these as you work on the program — there shouldn't be any left in the submitted version.

Academic misconduct

Your attention is drawn to the university policy on academic misconduct and to the Contract Cheating Risk and Avoidance Guide (available on Canvas).

The code you submit must be your own work. If some text or code in the coursework has been taken from other sources (beyond the material provided for this course), you must give a clear reference to the source in comments in the code or the `README` file. We cannot give any marks for work that is not your own.

Failure to clearly reference work that has been obtained from other sources, or copying the words or code of another student, is plagiarism.

Asking a third party to complete coursework on your behalf, and then submitting it as your own work, is contract cheating.

You must never give a copy of your coursework writing or code to another student, and you must always refuse any request from another student for a copy of your coursework. Sharing your coursework with another student is collusion.

Suspected plagiarism, contract cheating or collusion will be referred to the School Discipline Committee for investigation. This will delay your feedback, and if you are found guilty of misconduct then it will result in a penalty such as voiding the course or being withdrawn from the university.

So please don't do any of the things above! If you have any concerns about potential academic misconduct, please ask your local course leader for guidance.