CSC209 Image Filtering

Learning Objectives

At the end of this assignment students will be able to

- write a program that creates new processes
- manage communication between processes using pipes
- read, write, and manipulate binary data
- read and add to a medium-sized C program

Introduction

In this assignment, you'll build a simple pipeline for manipulating bitmap image files by applying a set of common *filters* on the images, which will allow the user to do things like turn the image to grayscale, blur the image, and increase the image size.

Each image filter will be a separate program that reads in a bitmap image from its standard input, calculates some transformation on the image's pixels and possibly its dimensions, and then writes a transformed bitmap image to its standard output.

Because we want to provide a convenient interface to the user to run several filters on a single image, you will also create a "master" program that spawns a separate process for each filter the user specifies. This program will also perform some basic process management, waiting for all of its child processes to complete, and reporting any errors that occur.

Preparation

Do a `git pull` in your repository to find the starter code for assignment 3.

Also, we strongly suggest first completing Lab 5 if you haven't yet done so. This will make sure you are familiar with the bitmap file format we are using for this assignment.

Starter code overview

Here is a brief description of the files found in the starter code for this assignment. *Please note that unlike previous assignments, you will be creating brand-new files, not just adding to existing ones!*

- `bitmap.h` and `bitmap.c`: definitions of the main types and functions used in this assignment. Look for `TODO`s here to fill in the required functionality for processing bitmap files.

  NOTE: there's a small typo in `bitmap.h` that doesn't affect compilation, but you should change to make the code clearer. The prototype of `run_filter` should be

  ```
  c void run_filter(void (*filter)(Bitmap *), int scale_factor);
  ```

- `copy.c`: a skeleton file for the simplest image filter you're writing. You'll be completing this file, and adding four more similar ones.
- `image_filter.c`: a very incomplete "master" program that manages a pipeline of image filters. You don't have to worry about this file until Part 2.
- `Makefile`: an *incomplete* makefile that you (and we) will use to compile the different executables in this assignment. We've also provided a basic `test` target that you should extend to automate some basic tests of your filters as you work through this assignment.
- `images/`: a directory containing a sample bitmap image file you can use for testing. Feel free to add your own!

General C code guidelines

We have the following expectations for all source code you submit for this assignment:

1. Perform error-checking on system and library calls and use perror to print good error messages.
2. Explicitly free all dynamically-allocated memory before a program terminates.
3. Explicitly close all unused pipe ends, for every process.
4. Don't modify/delete any of the provided functions/structs in the starter code unless we say that you can.

Part 1: Image filters

Your first task is to complete a set of image filter programs, described below. Each filter will have the same base structure, but will differ in the actual pixel value calculations they perform, and how many pixels they need to buffer for their calculations.

Bitmap file format

We are using the same standard for bitmap images as Lab 5: 24-bit bitmap files whose width and height are divisible by 4. From the bitmap header section, you will again need to access the header size and image width and height. In addition to these three integers, you will also need to access the image *file size*, which is an integer stored at offset 2 in the bitmap header.

Your **first task** should be to complete the `read_header` function in `bitmap.c`. This reads in the bitmap metadata from standard input, which is necessary to complete all of the filters for this part of the assignment.

Image filter structure

Unlike Lab 5, in which you read in a bitmap image from a file, here you will read in the data from standard input. And because each program must write out a complete bitmap file, you must write out all of the header data as well.

Because we expect these programs to be chained together with pipes, it would be rather inefficient to always read in the *entire* image file (including all its pixel data), and then perform some computations on the pixels, and the write out the *entire* output file. All of the filters you'll be implementing are local filters, meaning each computed pixel value will be based on at most a few surrounding pixels, and will not require reading in more than a few pixel rows of the image before being able to start computing. To take advantage of this, you'll use implement simple buffering approach for each filter, which is loosely described as follows:

1. Read in all of the header data, and extract the necessary image metadata described above.
2. Write out all of the header data for the transformed image file. In most cases, the output header will be exactly the same as the input header; for the scaling transformations, the file size and

image dimensions will need to be updated.
3. Read in a number of pixels (specified for each filter below).
4. Compute the values for the transformed pixel(s), and write out these new pixels.
5. Repeat steps 3-4 until the entire input image has been processed, and all the new pixels written out.

Of course, the key questions are:

- How do we compute the values of the transformed pixels?
- How many pixels do we need to read in step 3 before computing the transformed pixel values?

These questions depend on the exact filter being used, and we'll describe each one separately. Note that we specify the *exact executable name* in each case; for all of them besides `copy`, you should create and submit a new C source file that compiles to the required executable, and add the compilation recipe to your Makefile.

Individual pixel filters: `copy` and `greyscale`

The two simplest filters operate on one pixel at a time. Your loop in this case can read in and write out exactly one pixel in each iteration.

The `copy` filter leaves the pixels unchanged; this is a good one to get started, because you simply need to read in one pixel and then immediately write it back out. Running


```
$ mkdir images  # Create a folder for generated images to avoid clutter.
$ ./copy < dog.bmp > images/dog_copy.bmp
```

should product an exact copy of the original bitmap file.

The `greyscale` filter turns the image into a black-and-white version by transforming each pixel `p` into a new pixel `q`, where the blue, green, and red values of the new pixel are all equal to the average of the blue, green, and red values of the original pixel, rounded down to the nearest integer. You should do normal integer arithmetic on the individual pixel fields to compute the average: `(p.blue + p.green + p.red) / 3` works just fine.

Basic image convolutions: `gaussian_blur` and `edge_detection`

The next two filters compute a transformation on a pixel `p` from the pixel values of `p` *and* the eight pixels surrounding `p`, i.e., the 3-by-3 grid of pixels centred on `p`. For example, the pixel at position (2, 3) in the image is transformed based on the following nine pixel values:

```
(1, 2) (1, 3) (1, 4)
(2, 2) (2, 3) (2, 4)
(3, 2) (3, 3) (3, 4)
```

But because the pixels in a bitmap file are stored in rows, we require a few rows of pixel data to be read in before any transformation can begin. The main loop for these two filters reads in pixels by *row*: at each iteration, your program should be store exactly three rows of pixels from the original image, and using these pixels to calculate and print the transformed values for every pixel in the middle row.

We have given you the code that computes the pixel transformations for you; they involve some technical calculations that are beyond the scope of this course, but you can learn about in a course like CSC320 (Introduction to Visual Computing). Instead, your main job here will be to manage the buffering of the pixel data, and figuring out how to invoke the provided functions to compute the correct pixel transformations. Read about what these functions expect in `bitmap.h`.

Note about boundaries: because each of these filters compute transformations based on each pixel's neighbours, the pixels at the very edge of the image pose a problem (they don't have neighbours on one or more sides). For simplicity on this assignment, resolve this simply by making every "edge" pixel have the exact same transformed value as its "inner" neighbour. For example, the pixel at position (0, 5) should have the same transformed value as the pixel (1, 5), and the pixel at position (0, 0) should have the same transformed value as the pixel (1, 1). If the image width is 200, the pixel at position (199, 3) should have the same transformed value as the pixel (198, 3). You may find the provided `min` and `max` macros helpful for handling these boundary pixels elegantly.

If you're interested in reading more about image convolutions, try the Wikipedia page.

Scaling

The last filter is used to scale an existing image. It takes one command-line argument `scale_factor`, which must be an integer greater than 0. The image produced by this filter should be the same as the original image, except its width and height are multiplied by the provided scale factor. You may assume this command-line argument is always provided, and is always valid (no error-checking is required for this).

The pixel at position `(i, j)` in the scaled image should equal the pixel at position `(i / scale_factor, j / scale_factor)` in the original image. Note that this is integer division, which rounds down. (This is the simplest form of scaling, and it doesn't produce very smooth-looking images. There are more complex algorithms for scaling you'll learn about in courses like CSC320.)

In order for the output bitmap to be valid, you'll need to update three parts of the bitmap header before writing it to stdout: the image width, image height, and the total file size. Note that the header's size does not change with this transformation, and so you can compute the new file size by adding the header size to the total space required to store all the pixels in the image.

Note: due to the structure of the provided `run_filter` function, you'll need some way to access the input scale factor when calculating and writing the transformed pixels. There are a few different ways to do this (feel free to choose from the following):

1. Save this value in a global variable in your `scale.c` source file.
2. Modify the `Bitmap` struct to store this data.
3. Do not use `run_filter` at all, and instead use your own modified version that makes the scale factor more accessible to the filter itself. This might require updating `bitmap.h` as well.

Checkpoint!

We strongly encourage you to test each image filter separately as you complete it. At the bottom of the starter `Makefile`, we've provided a target `test` with a simple example command of one of the filters. If you add more commands to this rule, you'll have a convenient way to run simple tests on different executables, just by running `make test`!

You should even be able to run multiple filters in a pipeline using standard shell syntax, e.g.

```
$ ./gaussian_blur < dog.bmp | ./gaussian_blur | ./gaussian_blur | ./greyscale | scale 2 > images/dog_piped.bmp
```

which should produce a big, blurry, grey dog.

Part 2: Managing multiple filter processes

The second part of this assignment is to complete the program `image_filter`, which takes at least two

command-line arguments:

1. Its first two command-line arguments specify filename for the input and output images, respectively.
2. The remaining arguments each specify the one of the five image filter programs you developed in Part 1.

   If the user wants to run the scaling filter, this must be specified in a command-line argument that contains “scale “ and a scale factor, e.g. `"scale 2"`.

   If no arguments are provided, a single `copy` filter is performed.

`image_filter` is responsible for doing the following:

1. Create one new process for each filter specified by the command-line arguments.
2. Connect the processes properly to each other (and to the input and output files) using pipes. Remember that each filter process can *only* communicate using its stdin/stdout, meaning that `dup2` must be used to redirect each of their stdins and stdouts.

   The filters must process the image in the same order the command-line arguments are specified by the user.

3. Wait for all processes to complete, and print a success message to the user. If one or more of the processes failed (exit with non-zero status), print a warning message to the user. We’ve provided string constants for these messages for you in the starter code.

For example, the command


```
$ ./gaussian_blur < dog.bmp | ./gaussian_blur | ./gaussian_blur | ./greyscale |
./scale 2 > images/dog_piped.bmp
```


should result in the same behaviour (except the output message) as

```
$ ./image_filter dog.bmp images/dog_piped.bmp ./gaussian_blur ./gaussian_blur
./greyscale "./scale 2"
```

Submission

Commit all your code to your repository. Running `make` should produce six executables: the five image filters `copy`, `greyscale`, `gaussian_blur`, `edge_detection`, and `scale` (check their names carefully!) and the master `image_filter` executable. You are welcome to commit sample test files that you used.

Note: it is generally not good practice to commit the .o or executable files to your repository, as these files should be automatically generated from your source code. If you accidentally committed such files, please remove them from your repo (`git rm`) before your final submission.

Back to top