The University of Hong Kong
ENGG1111 Computer Programming and Applications

Assignment 2 Deadline: 17:00 Nov 16, 2015.

ENGG1111Shop: Enhancing photographic images

Many photo manipulation applications such as Photoshop are written in C++. Thanks to your hard work over the past weeks you now have all the programming skills needed to create your own photo manipulation software. In this assignment you will write functions to enhance greyscale images in several different ways. In the Appendix you'll find links to some photos to experiment with.

Introduction: Greyscale bitmaps

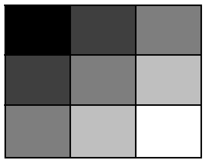
A bitmap (or, more correctly, pixmap) is a common way of representing digital images. A rectangular image, in the form of a grid of pixels, is represented by an array in which each element of the array stores colour information for a single pixel. The images you'll work with in this assignment are greyscale images in which the colour of a pixel is a shade of grey ranging from pure black to pure white.

For convenience we will use a 2D-array of *int* to represent an image. Each element of the array represents the shade of a single pixel in the image. In image files in secondary memory, however, the shade of each pixel is stored as a single byte (8 bits). That means we can represent a maximum of 2^8 or 256 different shades of grey, with values ranging from 0 to 255. By convention, 0 maps to pure black and 255 maps to pure white. Values in between 0 and 255 map to progressively lighter shades.

Example:

0	63	127
63	127	191
127	191	255

encodes the image



Working with 8-bit greyscale values is very straightforward: in general, increasing the value of a pixel will make it lighter, and decreasing its value will make it darker. You just need to take care at the extremes of the range to avoid decreasing a value below 0 or increasing a value beyond 255. After each manipulation, you must clip any result less than zero to zero and any result greater than 255 to 255. This follows the simple observation that no shade can be darker than pure black nor lighter than pure white.

Our image files will be in standard BMP format. Almost every photo viewer can open and display BMP files. Double-clicking the files should display them so you can see the effect of your manipulations.

Supplied code

A partially completed program, *ENGG1111Shop.cpp*, is already written for you. It declares several global constants related to the bitmap images you'll work with:

- HEADER_SIZE – size of the BMP file header block in bytes:1078
- IMAGE_HEIGHT – vertical dimension of the image in pixels: 250
- IMAGE_WIDTH – horizontal dimension of the image in pixels: 200
- AVG_REGION_DIM – dimension of the sides of the region used in blurring: 3 (*see Task 4*)
- PURE_BLACK – value representing the darkest possible shade: 0
- PURE_WHITE – value representing the lightest possible shade: 255

All code to read and write BMP files (with suffix .bmp) is provided. You won't manipulate header blocks and so you don't need to use HEADER_SIZE in your own functions. You will, however, make use of the other constants to complete this assignment.

IMAGE_HEIGHT and IMAGE_WIDTH are used to set dimensions of the image arrays that your functions will operate on. The sample files referenced in the Appendix all contain 250 x 200 pixel images. When testing and debugging your individual functions you'll find it convenient to work with much smaller arrays and, for example, test to see whether you can duplicate the example provided for each function. To do this you can write test programs that declare smaller values of IMAGE_HEIGHT and IMAGE_WIDTH and call your functions from within those programs.

Your only responsibility in this assignment is to implement image enhancement functions. The supplied code takes care of all other details including the user interface. You do not need to study that code.

Tasks

The principal tasks in this assignment are to write functions to:

- adjust the brightness of an image: *adjust_brightness()*
- convert an image to monochrome (pure black and white): *convert_to_bw()*
- blur an image: *blur()*
- sharpen an image: *sharpen()*

The declarations of these functions are provided; you are required to write their definitions. They will be called from the user interface in response to user requests. Following our design approach of "divide-and-conquer", some functions are further decomposed into component functions which you must also write. Again, the declarations of those functions are provided: *clip()*, *copy_border()*, and *average()* .

Each task is described below. If you are interested in knowing more about a function and its role in image enhancement, you can find such information in the document *ENGG1111Shop: Background*.

Task 1 [5%]: Clip out-of-range values

(a) [5%] `int clip(int value);`

This is a simple utility function that implements the clipping described in the Introduction. It returns PURE_BLACK for arguments less than PURE_BLACK and PURE_WHITE for arguments greater than PURE_WHITE. Otherwise it returns the argument value. Example:

```
const int PURE_BLACK = 0;
const int PURE_WHITE = 255;

// ...
int tests[3] = {-10, 242, 279};

for (int i = 0; i < 3; ++i)
{
   int result = clip(tests[i]);
   cout << result << ' ';
}
```

Executing this code produces the output:

```
0 242 255
```

Task 1 Deliverable

(a) A file named *clip.cpp* containing only the definition of the `clip()` function.

For the remaining functions, the first two arguments are arrays of pixel values representing *source* and *result* images with dimensions IMAGE_HEIGHT x IMAGE_WIDTH. The *source* image is not changed as a result of executing any of the functions.

Task 2 [10%]: Adjust the brightness of an image

(a) [10%]

```
void adjust_brightness(int source[][IMAGE_WIDTH],
                       int result[][IMAGE_WIDTH],
                       int adjustment);
```

This function applies the adjustment passed as the third argument. The adjustment value is added to each pixel value in the *source* image to calculate the value of the equivalent pixel in the *result* image. Each resulting value should be clipped to lie in the range PURE_BLACK to PURE_WHITE by calling the `clip()` function you wrote in Task 1. Example:

```
const int IMAGE_HEIGHT = 3;
const int IMAGE_WIDTH = 3;

// ...
   int source[IMAGE_HEIGHT][IMAGE_WIDTH] =
   {
      {0, 63, 127},
      {63, 127, 191},
      {127, 191, 255}
   };
   int result[IMAGE_HEIGHT][IMAGE_WIDTH] = {};
   int adjustment = -100;

   adjust_brightness(source, result, adjustment);
```

After the call to *adjust_brightness()*, the array *result* will contain the following values:

```
0   0   27
0   27  91
27  91  155
```

Task 2 Deliverable

(a) A file named *adjust_brightness.cpp* containing only the definition of the `adjust_brightness()` function.

Task 3 [10%]: Convert an image to monochrome (pure black and white)

(a) [10%]

```
void convert_to_bw(int source[][IMAGE_WIDTH],
                   int result[][IMAGE_WIDTH]);
```

This function generates the *result* image by taking each pixel value of the *source* image and converting it to whichever is the closest of PURE_BLACK and PURE_WHITE. You can assume that the size of the range of values, PURE_BLACK to PURE_WHITE, is an even number. Example:

```
const int IMAGE_HEIGHT = 3;
const int IMAGE_WIDTH = 3;

// ...
   int source[IMAGE_HEIGHT][IMAGE_WIDTH] =
   {
      {0, 63, 127},
      {63, 127, 191},
      {127, 191, 255}
   };
   int result[IMAGE_HEIGHT][IMAGE_WIDTH] = {};

   convert_to_bw(source, result);
```

After the call to *convert_to_bw()*, the array *result* will contain the following values:

```
0   0   0
0   0   255
0   255 255
```

Task 3 Deliverable

(a) A file named *convert_to_bw.cpp* containing only the definition of the `convert_to_bw()` function.

Task 4 [55%]: Blur an image

A simple technique for smoothing a data set is to replace each data point with the average of a region of adjacent points. The effect of applying this to an image is to smooth or blur the image. Your *blur()* function will use this technique; it will generate *result* pixel values by averaging the values of the 3x3 neighbourhood of pixels centred on the equivalent pixel in the *source*.

The technique is straightforward to apply for pixels that have 8 immediate neighbours and, hence, a full 3x3 neighbourhood. We sum the values of the 9 pixels in the neighbourhood and divide by 9 to obtain the average. For pixels in the outermost 1-pixel border of the image, however, there is a problem: those pixels do not have 8 neighbours. Since we can't calculate a meaningful neighbourhood average for pixels in the border we'll simply copy their values unchanged from *source* to *result*.

Thus there are two distinct sub-operations that make up the blurring operation. You will implement each as a function and call them in turn from the body of *blur()*:

a) copy all border pixels from the *source* image directly to the *result* image.
b) for each non-border pixel of the *source* image, calculate the average of its 3x3 neighbourhood values to obtain the value of the equivalent pixel in the *result* image.

(a) [15%]

```
void copy_border(int source[][IMAGE_WIDTH],
                 int result[][IMAGE_WIDTH]);
```

This function copies values from the 1-pixel border of the *source* image to the *result* image. Thus the border values of the *result* will be identical to those of the *source* and other pixel values of *result* will be unchanged. Example:

```
const int IMAGE_HEIGHT = 5;
const int IMAGE_WIDTH = 6;

// ...
   int source[IMAGE_HEIGHT][IMAGE_WIDTH] =
   {
      {255, 255, 255, 255, 255, 255},
      {255, 0, 255, 255, 255, 255},
      {255, 255, 255, 255, 100, 150},
      {255, 255, 60, 255, 255, 255},
      {150, 255, 255, 255, 255, 255}
   };
   int result[IMAGE_HEIGHT][IMAGE_WIDTH] = {};

   copy_border(source, result);
```

After the call to *copy_border()*, the array *result* will contain the following values:

```
255 255 255 255 255 255
255 0   0   0   0   255
255 0   0   0   0   150
255 0   0   0   0   255
150 255 255 255 255 255
```

(b) [35%]
```
void average(int source[][IMAGE_WIDTH],
             int result[][IMAGE_WIDTH]);
```

This function uses *source* pixel neighbourhood-averaging as described above to generate the values of all *result* array pixels except those in its 1-pixel border. You may assume that the neighbourhood dimension is 3 as given in the constant AVG_REGION_DIM and that both IMAGE_HEIGHT and IMAGE_WIDTH are at least 3. Your implementation will step through each non-border pixel of *source*, sum the 9 values in the pixel's 3 x 3 neighbourhood and divide the sum by 9. Use integer division (you can discard any fractional part of the result) and, finally, apply the `clip()` function to obtain a value for the equivalent pixel in the *result* image. An example is shown below for *blur()*.

(c) [5%]
```
void blur(int source[][IMAGE_WIDTH],
          int result[][IMAGE_WIDTH]);
```

This function simply calls the previous two functions to apply them, as a pair, to a *source* image array to generate a *result* image array. The required steps are:

- call `copy_border()` to copy a 1-pixel border from the *source* to the *result* image array.
- call `average()`, supplying, as the second argument, the *result* array as modified by `copy_border()`.

Example:

```
const int IMAGE_HEIGHT = 5;
const int IMAGE_WIDTH = 6;

// ...
   int source[IMAGE_HEIGHT][IMAGE_WIDTH] =
   {
      {255, 255, 255, 255, 255, 255},
      {255, 0, 255, 255, 255, 255},
      {255, 255, 255, 255, 100, 150},
      {255, 255, 60, 255, 255, 255},
      {150, 255, 255, 255, 255, 255}
   };
   int result[IMAGE_HEIGHT][IMAGE_WIDTH] = {};

   blur(source, result);
```

After the call to *blur()*, the array *result* will contain the following values:

```
255 255 255 255 255 255
255 226 226 237 226 255
255 205 205 216 226 150
255 221 233 216 226 255
150 255 255 255 255 255
```

Task 4 Deliverables

(a) A file named *copy_border.cpp* containing only the definition of the `copy_border()` function.
(b) A file named *average.cpp* containing only the definition of the `average()` function.
(c) A file named *blur.cpp* containing only the definition of the `blur()` function.

Task 5 [20%]: Sharpen an image

The technique you'll use to sharpen an image is known as *unsharp masking*. The edges of features and objects depicted in the image are emphasized by first subtracting a blurred version of the image from the original. This produces an image consisting predominantly of those edges (if you are interested in how this works, there is more detail in *ENGG1111Shop: Background)*. That image is called an *unsharp mask* because of the way it is generated.

A percentage of the unsharp mask is then added back to the original *source* to obtain a *result* in which the edges are now more distinct.   The user will specify the percentage of the unsharp mask to be added.

(a) [20%]

```
void sharpen(int source[][IMAGE_WIDTH],
             int result[][IMAGE_WIDTH],
             int amount);
```

This function applies unsharp masking to the *source* image array to generate a sharpened version as the *result* array. The argument *amount* represents the percentage of the unsharp mask that will be added to the *source* image to generate the result. The required steps are:

- declare an array to contain a blurred version of the *source* image.
- call the `blur()` function from Task 4 to generate the pixel values of the blurred image.
- for each pixel value in *source*:
    - subtract the value of the equivalent pixel in the blurred image to obtain the value of that pixel in the unsharp mask,
    - calculate the pixel value in the *result* image by scaling the unsharp mask value by (*amount / 100)* and adding it to the *source* pixel value as outlined above. Use integer division, discarding any fractional part of the result.

      Thus if *source(i, j)* is a pixel value in the original image and *blurred(i, j)* is the equivalent value in the blurred image, then the value *result(i, j)* is calculated as:

      $$\begin{aligned} result\ (i, j) &= source(i, j) + unsharp(i, j) * amount / 100 \\ &= source(i, j) + (source(i, j) - blurred(i, j)) * amount / 100 \end{aligned}$$

    - finally, as always, apply the clip() function to the resulting value.

Example:

```
const int IMAGE_HEIGHT = 5;
const int IMAGE_WIDTH = 6;

// ...
   int source[IMAGE_HEIGHT][IMAGE_WIDTH] =
   {
      {94, 94, 94, 160, 160, 160},
      {94, 94, 94, 160, 160, 160},
      {94, 94, 94, 160, 160, 160},
      {94, 94, 94, 160, 160, 160},
      {94, 94, 94, 160, 160, 160}
   };
   int result[IMAGE_HEIGHT][IMAGE_WIDTH] = {};
   int amount = 200;

   sharpen(source, result, amount);
```

After the call to *sharpen()*, the array *result* will contain the following values:

```
94 94 94 160 160 160
94 94 50 204 160 160
94 94 50 204 160 160
94 94 50 204 160 160
94 94 94 160 160 160
```

Task 5 Deliverable

(a) A file named *sharpen.cpp* containing only the definition of the `sharpen()` function.

Summary of Deliverables

IMPORTANT: Be sure to name your files exactly as listed here. Any deviation from the expected filenames will prevent the auto-marking software from compiling and executing your functions and you will lose marks.

Please hand in through Moodle a total of 7 files:

- `clip.cpp`
- `adjust_brightness.cpp`
- `convert_to_bw.cpp`
- `copy_border.cpp`
- `average.cpp`
- `blur.cpp`
- `sharpen.cpp`

You are *not* required to hand in the completed *ENGG1111Shop* program.

If your functions work correctly within the *ENGG1111Shop.cpp* implementation as supplied, then they will work correctly in our test environment. For this reason it is essential that you *do not* alter the supplied code in any way in order to get your functions to work.

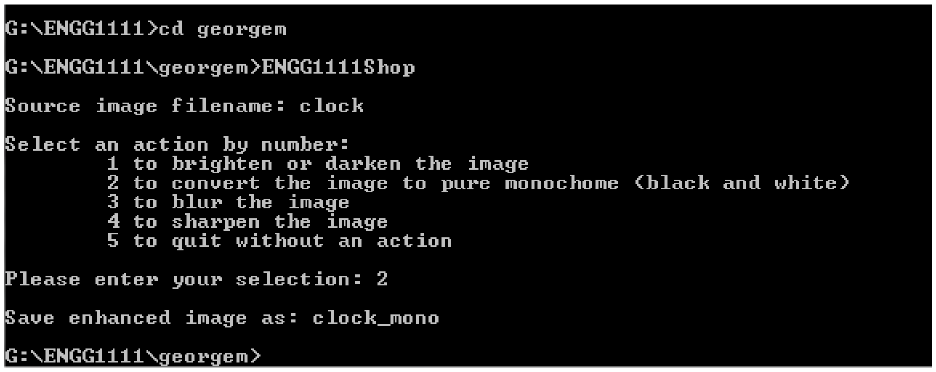
As an example of delivering a function as a file, consider the *get_filename()* function in the supplied code. If you were required to hand in that function as a file, you would write the function, test it and then copy the function definition, and only the function definition, into the file. You would hand in the file, named *get_filename.cpp*, and it would contain only the following text:

```
string get_filename()
{
    string filename;
    cin >> filename;
    return filename + ".bmp";
}
```

Appendix

Here are some images to experiment with. To open an image in ENGG1111Shop, make sure the file is in the current directory. When the program prompts for a source filename, input the name of the image file without its .bmp suffix.

After performing an enhancement, the program will prompt for a filename to save the enhanced image. Don’t use the filename of the original image unless you want to overwrite that file. Provide a new name and, again, there is no need to add the .bmp suffix.   Example:



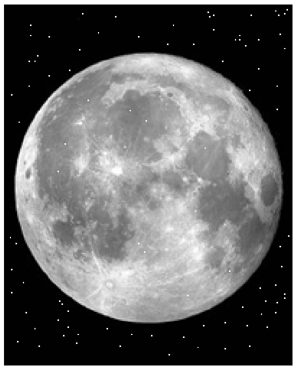
```
G:\ENGG1111>cd georgem

G:\ENGG1111\georgem>ENGG1111Shop

Source image filename: clock

Select an action by number:
        1 to brighten or darken the image
        2 to convert the image to pure monochome (black and white)
        3 to blur the image
        4 to sharpen the image
        5 to quit without an action

Please enter your selection: 2

Save enhanced image as: clock_mono

G:\ENGG1111\georgem>
```

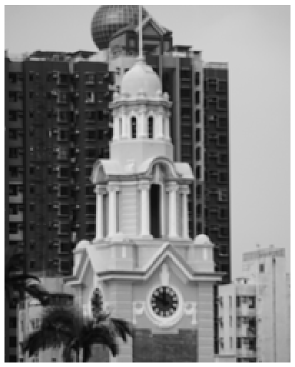
Here the file *clock.bmp* is opened, the image is enhanced by converting it to pure black and white, and a new file, *clock_mono.bmp* is created containing the enhanced image.

Try working with the following files available from Moodle:

kitten.bmp

clock.bmp


moon.bmp


oof.bmp

- Try brightening and darkening *kitten.bmp*. Try blurring and sharpening.
- For *clock.bmp*, try converting to monochrome to emphasize the clock face against its immediate background.
- The white dots in *moon.bmp* are noise (see *Reference*). Try blurring to smooth them into the background. This will also blur the details of the moon itself but should reduce the visible noise that is scattered across the photo.
- The last image, *oof.bmp*, is out of focus. This is a common defect. Try sharpening with an *amount* between 100% and 200% to see if you can improve the photo.