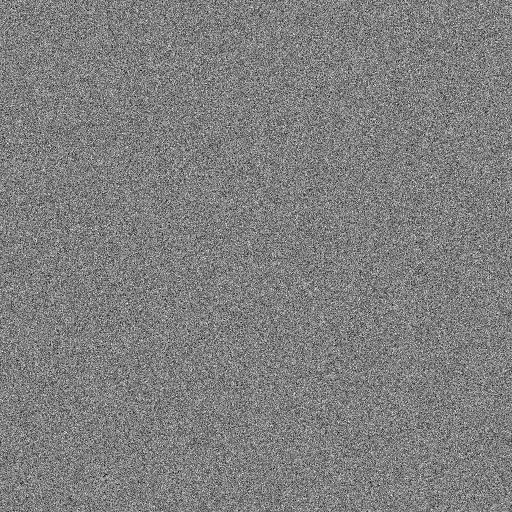
*1. Test log:*

Tests of the Image Class’ functions were carried out with the following images:  
  
  
Lena.pgm, 512x512px



Noise.pgm, Sizes: 512x512, 412x412 (small), 612x612 (large)



Ramp.pgm , Sizes: 512x512, 412x412 (small), 612x612 (large)  
  
  
Test structure: Tests were carried out using the above images. The expected results were determined by the operations carried out with ImageJ. Actual results were gathered with the Image class implemented in C++.

The tests have the following layout:  
- Type of operators  
- operator  
- Result comparison

A critical evaluation is provided on page 25. The appendix with the code begins on page 26.

*Arithmetic operators using another image:*

Image operator+(const Image& anImage);  
This functions adds two images and returns a new composite of both.  
  
Parameters used:lena.pgm and noise.pgm (same size)  
Expected Result:  
  
Actual result:  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Parameters used:  
lena.pgm and noise.pgm (noise.pgm = small)  
Expected result:  
  
Actual result:   




Parameters used:  
lena.pgm and noise.pgm (noise.pgm = big)  
Expected result:  
  
Actual result:  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Image operator-(const Image& anImage);  
Subtracts an image from another one and returns the composite of both in a new image.  
  
Parameters used:  
  
lena.pgm, noise.pgm (equal size)  
  
Expected result:

  
Actual result:  


Lena.pgm, noise.pgm (smaller)  
Expected result:  
  
Actual result:  


lena.pgm, noise.pgm (bigger)  
Expected result:  
  
  
Actual result:

  
  
  
  
  
  
  
  
*Arithmetic assignment operators using another image*  
Image& operator+=(const Image& anImage);  
Adds an image (pixelwise) to the current image and returns a reference to the current image.  
  
Parameters used:  
lena.pgm, ramp.pgm (equal size)  
  
Expected result:  
  
Actual result:  
  
  
Pass/fail: pass



Lena.pgm, ramp.pgm(bigger)  
Expected result:

Actual result:  
  
  
Pass/fail: pass



lena.pgm, ramp.pgm(smaller)  
Expected result:

Actual result:  




Pass/fail: pass

Image& operator-=(const Image& anImage);  
Subtracts an image (pixelwise) from the current image and returns a reference to the current image.  
  
Parameters used:  
lena.pgm, ramp.pgm (equal size)  
  
Expected result:  
  
  
Actual result:  
  




Pass/fail: pass

lena.pgm, ramp.pgm (smaller)  
  
  
Expected result:  
  
  
Actual result:



Pass/fail: pass



Lena.pgm, ramp.pgm(bigger)

Expected result:

Actual result:



Pass/fail: pass

Arithmetic operators using a floating point parameter:

Image operator+(float aValue);  
Addition operator, add a value to each pixel.  
  
Parameters used:  
lena.pgm, aFloat = 50  
  
Expected Result:  
  
  
  
Actual result:  
  
  
  
  
Pass/fail: pass  
  
  
  
  
  
  
  
Image operator-(float aValue);  
Subtraction operator, subtracts a value from each pixel.  
  
Parameters used:  
lena.pgm, aFloat = 50  
  
Expected result:  
  
  
  
Actual result:



Pass/fail: pass

Image operator\*(float aValue);  
Multiplication operator, multiplies every pixel of the image by floating point value  
  
Parameters used:  
lena.pgm, aFloat = 2.5  
  
Expected result:  
  
  
  
  
Actual result:  
  
  
  
  
  
Pass/fail: pass



Image operator/(float aValue);  
Division operator, divides each pixel of the image by aFloat.  
  
Parameters used:  
lena.pgm, aFloat = 2.5  
  
Expected result:  
  
  
Actual result:



Comment: this operator did not yield the expected result.

Arithmetic assignment operators using a floating point parameter:

Image operator+=(float aValue);  
Addition operator, add a value to each pixel of the current image.  
  
Parameters used:  
lena.pgm, aFloat = 50  
  
Expected Result:  
  
  
  
Actual result:  
  
  
  
  
  
  
  
  
  
  
  
  
Image operator-=(float aValue);  
Subtraction operator, subtracts a value from each pixel.  
  
Parameters used:  
lena.pgm, aFloat = 50  
  
Expected result:  
  
  
  
Actual result:  


Image operator\*=(float aValue);  
Multiplication operator, multiplies every pixel of the image by floating point value  
  
Parameters used:  
lena.pgm, aFloat = 2.5  
  
Expected result:  
  
  
  
  
Actual result:  
  


Image operator/=(float aValue);  
Division operator, divides each pixel of the image by aFloat.  
  
Parameters used:  
lena.pgm, aFloat = 2.5  
  
Expected result:  
  
  
Actual result:  


Comment: this operator did not yield the expected result.

Image Image::operator!()  
Negation operator  
  
Actual result:



Setters and getters:  
  
void setPixel(unsigned int i, unsigned int j, float aValue);  
  
float getPixel(unsigned int i, unsigned int j) const;

The test of these two functions was carried out using the lenga.pgm image.

Test procedure:  
- getPixel(25, 25)  
- setPixel(25,25, 125) , sets pixel at 25, 25 to 125  
- getPixel(25,25)  
  
Initial state: 163  
Expected value after call to setPixel: 125  
Actual result: 125

unsigned int getHeight() const; unsigned int getWidth() const;

This test was carried out using the lenga.pgm image, size 512x512

imgsize  
Top parameter height, bottom parameter width.

float getAspectRatio();  
Tested with an image size 800x600.   
Expected result 1.34  
Actual result:   
  
  
*2. Critical Evaluation:*



Overall, I found this assignment relatively straightforward and I didn’t encounter any major problems with the implementation of the overloaded operators, save for the division operator. I implemented the operators that used images as parameters in a way that allowed for the images used to be of different size to the original image. As for the division operator, I was not able to locate the reason as to why it wasn’t and still is not working as expected.  
  
I could’ve done a more coherent job with testing and actually provided Test.cpp with a command line interface that would have allowed me to call the functions directly using a switch statement, in order to simplify the process. Instead I called each operator discretely, resulting in a largely unsanitary file which made keeping track of the testing process rather difficult.

Unfortunately, I did not have the time to implement the ASCII read and write method, which the program would have benefitted from.

*3. Code:*

#ifndef IMAGE\_H

#define IMAGE\_H

/\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* @file Image.h

\*

\* @brief File contains function headers for the image class.

\*

\* @version 1.0

\*

\* @todo Implement the default constructor; x

\* @todo Implement the copy constructor; x

\* @todo Implement the destructor x

\* (don't forget to release the memory if needed);

\* @todo Add the assignment operator; x

\* @todo Imaplement the code to read an image from an ASCII file

\* as follows:

\* - Each line of the ASCII file corresponds to a line of

\* the image;

\* - Each pixel is the ASCII file corresponds to

\* a floating point number;

\* - Two successive pixels in the image are separated

\* by a space character in the ASCII file

\* the image.

\* @todo Implement the code to save an image in an ASCII file;

\* @todo Implement the arithmetic operators x

\* (with a float as a parameter);

\* @todo Implement the arithmetic operators x

\* (with an image as a parameter);

\* @todo Implement the corresponding arithmetic/assignment operators; x

\* @todo Implement the negation operator to return the negative image; x

\* @todo Add an accessor on the width of the image; x

\* @todo Add an accessor on the height of the image; x

\* @todo Add a method returning the aspect ratio;

\* @todo Add a method to set the value of a given pixel; x

\* @todo Add a method returning the value of a given pixel; x

\* @todo Add a method returning the smallest value in the image; x

\* @todo Add a method returning the highest value in the image. x

\*

\* @date 25/10/2016

\*

\* @author Dorian B. Dressler (eeu436@bangor@ac.uk)

\*

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Include

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <string>

//==============================================================================

/\*\*

\* @class Image

\* @brief Image is a class to manage a greyscale image.

\*/

//==============================================================================

class Image

//------------------------------------------------------------------------------

{

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public:

//--------------------------------------------------------------------------

/// Default constructor.

//--------------------------------------------------------------------------

Image();

//------------------------------------------------------------------------

/// Copy constructor.

/\*\*

\* @param anImage: the image to copy

\*/

//------------------------------------------------------------------------

Image(const Image& anImage);

//------------------------------------------------------------------------

/// Destructor.

//------------------------------------------------------------------------

~Image();

//------------------------------------------------------------------------

/// Assignment operator (also called copy operator).

/\*\*

\* @param anImage: the image to copy

\* @return the updated version of the current image

\*/

//------------------------------------------------------------------------

Image& operator=(const Image& anImage);

//------------------------------------------------------------------------

/// Release the memory.

//------------------------------------------------------------------------

void destroy();

//------------------------------------------------------------------------

/// Addition operator. Add anImage

/\*\*

\* @param anImage: the image to add

\* @return the resulting image

\*/

//------------------------------------------------------------------------

Image operator+(const Image& anImage);

//------------------------------------------------------------------------

/// Subtraction operator. Add anImage

/\*\*

\* @param anImage: the image to subtract

\* @return the resulting image

\*/

//------------------------------------------------------------------------

Image operator-(const Image& anImage);

//------------------------------------------------------------------------

/// Addition assignment operator. Add anImage

/\*\*

\* @param anImage: the image to add

\* @return the updated version of the current image

\*/

//------------------------------------------------------------------------

Image& operator+=(const Image& anImage);

//------------------------------------------------------------------------

/// Subraction assignment operator. Add anImage

/\*\*

\* @param anImage: the image to subtract

\* @return the updated version of the current image

\*/

//------------------------------------------------------------------------

Image& operator-=(const Image& anImage);

//------------------------------------------------------------------------

/// Addition operator. Add aValue to every pixel of the image

/\*\*

\* @param aValue: the value to add

\* @return the resulting image

\*/

//------------------------------------------------------------------------

Image operator+(float aValue);

//------------------------------------------------------------------------

/// Subtraction operator. Subtract aValue to every pixel of the image

/\*\*

\* @param aValue: the value to subtract

\* @return the resulting image

\*/

//------------------------------------------------------------------------

Image operator-(float aValue);

//------------------------------------------------------------------------

/// Multiplication operator. Multiply every pixel of the image by aValue

/\*\*

\* @param aValue: the value for the multiplication

\* @return the resulting image

\*/

//------------------------------------------------------------------------

Image operator\*(float aValue);

//------------------------------------------------------------------------

/// Division operator. Divide every pixel of the image by aValue

/\*\*

\* @param aValue: the value for the division

\* @return the resulting image

\*/

//------------------------------------------------------------------------

Image operator/(float aValue);

//------------------------------------------------------------------------

/// Addition operator. Add aValue to every pixel of the image

/\*\*

\* @param aValue: the value to add

\* @return the updated version of the current image

\*/

//------------------------------------------------------------------------

Image& operator+=(float aValue);

//------------------------------------------------------------------------

/// Subtraction operator. Subtract aValue to every pixel of the image

/\*\*

\* @param aValue: the value to subtract

\* @return the updated version of the current image

\*/

//------------------------------------------------------------------------

Image& operator-=(float aValue);

//------------------------------------------------------------------------

/// Multiplication operator. Multiply every pixel of the image by aValue

/\*\*

\* @param aValue: the value for the multiplication

\* @return the updated version of the current image

\*/

//------------------------------------------------------------------------

Image& operator\*=(float aValue);

//------------------------------------------------------------------------

/// Division operator. Divide every pixel of the image by aValue

/\*\*

\* @param aValue: the value for the division

\* @return the updated version of the current image

\*/

//------------------------------------------------------------------------

Image& operator/=(float aValue);

//------------------------------------------------------------------------

/// Negation operator. Compute the negative of the current image.

/\*\*

\* @return the negative image

\*/

//------------------------------------------------------------------------

Image operator!();

//------------------------------------------------------------------------

/// Compute the maximum pixel value in the image

/\*\*

\* @return the maximum pixel

\*/

//------------------------------------------------------------------------

float getMaxValue() const;

//------------------------------------------------------------------------

/// Add aShiftValue to every pixel, then multiply every pixel

/// by aScaleValue

/\*\*

\* @param aShiftValue: the shift parameter of the filter

\* @param aScaleValue: the scale parameter of the filter

\*/

//------------------------------------------------------------------------

void shiftScaleFilter(float aShiftValue, float aScaleValue);

//------------------------------------------------------------------------

/// Load an image from a PGM file

/\*\*

\* @param aFileName: the name of the file to load

\*/

//------------------------------------------------------------------------

void loadPGM(const char\* aFileName);

//------------------------------------------------------------------------

/// Load an image from a PGM file

/\*\*

\* @param aFileName: the name of the file to load

\*/

//------------------------------------------------------------------------

void loadPGM(const std::string& aFileName);

//------------------------------------------------------------------------

/// Save the image in a PGM file

/\*\*

\* @param aFileName: the name of the file to write

\*/

//------------------------------------------------------------------------

void savePGM(const char\* aFileName);

//------------------------------------------------------------------------

/// Save the image in a PGM file

/\*\*

\* @param aFileName: the name of the file to write

\*/

//------------------------------------------------------------------------

void savePGM(const std::string& aFileName);

//------------------------------------------------------------------------

/// Change a pixel in the PGM file

/\*\*

\* @param i: the row of the pixel to set

\* @param i: the column of the pixel to set

\* @param aValue: the value of the pixel to set

\*/

//------------------------------------------------------------------------

void setPixel(unsigned int i, unsigned int j, float aValue);

//------------------------------------------------------------------------

/// Change a pixel in the PGM file

/\*\*

\* @param i: the row of the pixel to get

\* @param i: the column of the pixel to get

\* @return aValue: the value of the pixel to get

\*/

float getPixel(unsigned int i, unsigned int j) const;

//------------------------------------------------------------------------

/// Gets the width of an image.

/\*\*

\* @return m\_width: the width of the image

\*/

unsigned int getWidth() const;

//------------------------------------------------------------------------

/// Gets the height of an image.

/\*\*

\* @return m\_height: the height of the image

\*/

unsigned int getHeight() const;

//------------------------------------------------------------------------

/// Gets the aspect ratio of an image.

/\*\*

\* @return the image's aspect ratio

\*/

float getAspectRatio();

//------------------------------------------------------------------------

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

private:

/// Number of pixel along the horizontal axis

unsigned int m\_width;

/// Number of pixel along the vertical axis

unsigned int m\_height;

/// The pixel data

float\* m\_p\_image;

};

#endif

/\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* @file Image.cpp

\*

\* @brief Constructor file for the Image class.

\*

\* @version 1.0

\*

\* @date 25/10/2016

\*

\* @author Dorian B. Dressler (eeu436@bangor@ac.uk)

\*

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Define

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#define LINE\_SIZE 2048

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Include

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <sstream> // Head file for stringstream

#include <fstream> // Head file for filestream

#include <algorithm>

#include <iostream>

#include "Image.h"

//Constructor

//------------------

Image::Image():

//------------------

m\_width(0),

m\_height(0),

m\_p\_image(0)

//------------------

{ // NOTHING TO DO HERE

}

//Copy constructor

//--------------------------------

Image::Image(const Image& anImage):

//--------------------------------

m\_width(anImage.m\_width),

m\_height(anImage.m\_height),

m\_p\_image(new float[m\_height\*m\_width])

//--------------------------------

{

// IT IS THE CONSTRUCTOR, USE AN INITIALISATION LIST

// Copies data from one float array to another float array

std::memcpy(m\_p\_image, anImage.m\_p\_image, (m\_height\*m\_width\*sizeof(float)));

}

//Destructor

//-------------

Image::~Image()

//-------------

{

delete[] m\_p\_image;

// ADD CODE HERE TO RELEASE THE MEMORY

}

//Destructor

//-------------------

void Image::destroy()

//-------------------

{

// Memory has been dynamically allocated

if (m\_p\_image)

{

// Release the memory

delete [] m\_p\_image;

// Make sure the pointer is reset to NULL

m\_p\_image = 0;

}

// There is no pixel in the image

m\_width = 0;

m\_height = 0;

}

//Copies the data of an image

//-------------------------------------------

Image& Image::operator=(const Image& anImage)

//-------------------------------------------

{

// CODE HERE TO COPY THE DATA

destroy();

m\_width = anImage.m\_width;

m\_height = anImage.m\_height;

m\_p\_image = new float[m\_height\*m\_width];

std::memcpy(m\_p\_image, anImage.m\_p\_image, (m\_height\*m\_width\*sizeof(float)));

return \*(this);

}

//Adds two images to each other and returns resulting image

//------------------------------------------

Image Image::operator+(const Image& anImage)

//------------------------------------------

{

Image tempImage;

// if original image is smaller than added image

if (m\_width < anImage.m\_width) {

//std::cout << "added image bigger" << std::endl;

tempImage.m\_width = m\_width;

tempImage.m\_height = m\_height;

tempImage.m\_p\_image = new float[m\_width\*m\_height];

//copy smaller image into temp

std::memcpy(tempImage.m\_p\_image, m\_p\_image, (m\_width\*m\_height \* sizeof(float)));

//itereate over each pixel up to size of smaller image

for (unsigned int j = 0; j < m\_height; ++j) {

for (unsigned int i = 0; i < m\_width; ++i) {

//add to values in master image' i,j, the values at anImage's i,j and store in temp

tempImage.m\_p\_image[j \* m\_width + i] = m\_p\_image[j \* m\_width + i] + anImage.m\_p\_image[j \* anImage.m\_width + i];

}

}

}

else { //if subtracted image equal or smaller

//std::cout << "added image smaller or equal" << std::endl;

tempImage.m\_width = anImage.m\_width;

tempImage.m\_height = anImage.m\_height;

tempImage.m\_p\_image = new float[anImage.m\_width\*anImage.m\_height];

//copy image pixel by pixel

//copy original image into temporary image

std::memcpy(tempImage.m\_p\_image, anImage.m\_p\_image, (anImage.m\_width\*anImage.m\_height \* sizeof(float)));

for (unsigned int j = 0; j < anImage.m\_height; ++j) {

for (unsigned int i = 0; i < anImage.m\_width; ++i) {

//still some issues here

tempImage.m\_p\_image[j\* anImage.m\_width + i] = m\_p\_image[j \* m\_width + i] + anImage.m\_p\_image[j\* anImage.m\_width + i];

//use couts

}

}

}

return tempImage;

}

//SUBTRACTS TWO IMAGES FROM EACH OTHER AND RETURNS RESULTING IMAGE

//------------------------------------------

Image Image::operator-(const Image& anImage)

//------------------------------------------

{

Image tempImage;

// if original image is smaller than added image

if (m\_width < anImage.m\_width) {

std::cout << "subtracted image bigger" << std::endl;

tempImage.m\_width = m\_width;

tempImage.m\_height = m\_height;

tempImage.m\_p\_image = new float[m\_width\*m\_height];

//copy smaller image into temp

std::memcpy(tempImage.m\_p\_image, m\_p\_image, (m\_width\*m\_height\*sizeof(float)));

//itereate over each pixel up to size of smaller image

for (unsigned int j = 0; j < m\_height; ++j) {

for (unsigned int i = 0; i < m\_width; ++i) {

//add to values in master image' i,j, the values at anImage's i,j and store in temp

tempImage.m\_p\_image[j \* m\_width + i] = m\_p\_image[j \* m\_width + i] - anImage.m\_p\_image[j \* anImage.m\_width + i];

}

}

}

else { //if subtracted image equal or smaller

std::cout << "subtracted image smaller or equal" << std::endl;

tempImage.m\_width = anImage.m\_width;

tempImage.m\_height = anImage.m\_height;

tempImage.m\_p\_image = new float[anImage.m\_width\*anImage.m\_height];

//copy image pixel by pixel

//copy original image into temporary image

std::memcpy(tempImage.m\_p\_image, anImage.m\_p\_image, (anImage.m\_width\*anImage.m\_height\*sizeof(float)));

for (unsigned int j = 0; j < anImage.m\_height; ++j) {

for (unsigned int i = 0; i < anImage.m\_width; ++i) {

tempImage.m\_p\_image[j\* anImage.m\_width + i] = m\_p\_image[j \* m\_width + i] - anImage.m\_p\_image[j\* anImage.m\_width + i];

}

}

}

return tempImage;

}

//Adds two images and returns a reference to the created image

//--------------------------------------------

Image& Image::operator+=(const Image& anImage)

//--------------------------------------------

{

unsigned int imageWidth, imageHeight;

//if anImage is bigger than original image

if (m\_width < anImage.m\_width || m\_height < anImage.m\_height) {

//set image width and height of smaller image original

imageWidth = m\_width;

imageHeight = m\_height;

} else { //if anImage is smaller or of equal size than original image

//set image width and height of added image

imageWidth = anImage.m\_width;

imageHeight = anImage.m\_height;

}

//iterate rows and columns of pixel array

for (unsigned int j = 0; j < imageHeight; ++j) {

for (unsigned int i = 0; i < imageWidth; ++i) {

//add pixels from anImage to image and store in the image

m\_p\_image[j\* m\_width + i] = m\_p\_image[j \* m\_width + i] + anImage.m\_p\_image[j\* anImage.m\_width + i];

}

}

//return the reference

return \*(this);

}

//Subtracts two images and returns a reference to the image.

//--------------------------------------------

Image& Image::operator-=(const Image& anImage)

//--------------------------------------------

{

unsigned int imageWidth, imageHeight;

//set image width and height of smaller image original

if (m\_width < anImage.m\_width || m\_height < anImage.m\_height) {

//set image width and height to that of original image

imageWidth = m\_width;

imageHeight = m\_height;

}

else { //if anImage is smaller or of equal size than original image

//set image width and height of added image

imageWidth = anImage.m\_width;

imageHeight = anImage.m\_height;

}

//iterate rows and columns of pixel array

for (unsigned int j = 0; j < imageHeight; ++j) {

for (unsigned int i = 0; i < imageWidth; ++i) {

//subtract pixels from anImage to image and store in the image

m\_p\_image[j\* m\_width + i] = m\_p\_image[j \* m\_width + i] - anImage.m\_p\_image[j\* anImage.m\_width + i];

}

}

//return the reference

return \*(this);

}

//Adds aValue to each pixel of the current image.

//----------------------------------

Image Image::operator+(float aValue)

//----------------------------------

{

unsigned int pixelsInImage(m\_height\*m\_width);

Image tempImage;

tempImage.m\_height = m\_height;

tempImage.m\_width = m\_width;

tempImage.m\_p\_image = new float[pixelsInImage];

//iterate through pixel array

for (unsigned int i = 0; i < pixelsInImage; ++i) {

//add value to each value already in array and store at in new image

tempImage.m\_p\_image[i] = m\_p\_image[i] + aValue;

}

//returns the new image

return tempImage;

}

//Subtracts avalue from each pixel of the current image.

//----------------------------------

Image Image::operator-(float aValue)

//----------------------------------

{

unsigned int pixelsInImage(m\_height\*m\_width);

Image tempImage;

tempImage.m\_height = m\_height;

tempImage.m\_width = m\_width;

tempImage.m\_p\_image = new float[pixelsInImage];

//iterate through pixel array

for (unsigned int i = 0; i < pixelsInImage; ++i) {

//subtracts a value from each value in the original imge and store at in new image

tempImage.m\_p\_image[i] = m\_p\_image[i] - aValue;

}

//return the new iamge

return tempImage;

}

//Multiplies each pixel in image by aValue

//----------------------------------

Image Image::operator\*(float aValue)

//----------------------------------

{

unsigned int pixelsInImage(m\_height\*m\_width);

Image tempImage; //(new float[pixelsInImage]);

tempImage.m\_height = m\_height;

tempImage.m\_width = m\_width;

tempImage.m\_p\_image = new float[pixelsInImage];

//iterate through pixel array

for (unsigned int i = 0; i < pixelsInImage; ++i) {

//multiplies each value in the original with a floating point value and stores it in a new image

tempImage.m\_p\_image[i] = m\_p\_image[i] \* aValue;

}

//returns the new image

return tempImage;

}

//Divides each pixel by aValue and returns resulting image

//----------------------------------

Image Image::operator/(float aValue)

//----------------------------------

{

// ADD CODE HERE TO DIVIDE EACH PIXEL OF THE CURRENT IMAGE BY aValue,

// AND RETURN THE RESULTING IMAGE

unsigned int pixelsInImage(m\_height\*m\_width);

Image tempImage;

tempImage.m\_height = m\_height;

tempImage.m\_width = m\_width;

tempImage.m\_p\_image = new float[pixelsInImage];

for (unsigned int i = 0; i < pixelsInImage; ++i) {

tempImage.m\_p\_image[i] = m\_p\_image[i] / aValue;

}

return tempImage;

}

//Arithmetic assignment operator for addition

//-----------------------------------

Image& Image::operator+=(float aValue)

//-----------------------------------

{

unsigned int pixelsInImage(m\_height\*m\_width);

//iterates through the pixel array and adds a value to each pixel

for (unsigned int i = 0; i < pixelsInImage; ++i) {

m\_p\_image[i] = m\_p\_image[i] + aValue;

}

//returns the current image

return \*(this);

}

//Assignment operator for subtraction

//------------------------------------

Image& Image::operator-=(float aValue)

//------------------------------------

{

unsigned int pixelsInImage(m\_height\*m\_width);

//iterates through each pixel in the array and subtracts a value from it

for (unsigned int i = 0; i < pixelsInImage; ++i) {

m\_p\_image[i] = m\_p\_image[i] - aValue;

}

//returns a reference ot the image

return \*(this);

}

//Assignment operator for multiplication

//------------------------------------

Image& Image::operator\*=(float aValue)

//------------------------------------

{

unsigned int pixelsInImage(m\_height\*m\_width);

//iterates through each pixel in the array and multiplies it with a value

for (unsigned int i = 0; i < pixelsInImage; ++i) {

m\_p\_image[i] = m\_p\_image[i] \* aValue;

}

return \*(this);

}

//Assignment operator for division

//------------------------------------

Image& Image::operator/=(float aValue)

//------------------------------------

{

unsigned int pixelsInImage(m\_height\*m\_width);

//iterates through each pixel in the array and divides it by a value

for (unsigned int i = 0; i < pixelsInImage; ++i) {

m\_p\_image[i] = m\_p\_image[i] / aValue;

}

return \*(this);

}

//Implementation of negation operator

//----------------------

Image Image::operator!()

//----------------------

{

unsigned int pixelsInImage(m\_height\*m\_width);

const int MAX\_COLOR = 255;

Image tempImage;

tempImage.m\_height = m\_height;

tempImage.m\_width = m\_width;

tempImage.m\_p\_image = new float[pixelsInImage];

//iterates through each pixel in the array and inverts its

for (unsigned int i = 0; i < pixelsInImage; ++i) {

tempImage.m\_p\_image[i] = MAX\_COLOR - m\_p\_image[i];

}

return tempImage;

}

//Sets pixel value

//------------------------------

void Image::setPixel(unsigned int i, unsigned int j, float aValue)

//------------------------------

{

m\_p\_image[j \* m\_width + i] = aValue;

}

//Gets pixel value

//------------------------------

float Image::getPixel(unsigned int i, unsigned int j) const

{

return m\_p\_image[j \* m\_width + i];

}

//------------------------------

//Returns max value

//------------------------------

float Image::getMaxValue() const

//------------------------------

{

return (\*std::max\_element(&m\_p\_image[0], &m\_p\_image[m\_width \* m\_height]));

}

//Returns height

//------------------------------

unsigned int Image::getHeight() const

//------------------------------

{

return m\_height;

}

//Returns width

//------------------------------

unsigned int Image::getWidth() const

//------------------------------

{

return m\_width;

}

//Displays the aspect ratio

//------------------------------

float Image::getAspectRatio()

{

float image\_ratio = ((float)m\_width / (float)m\_height);

return image\_ratio;

}

//----------------------------------------------------------------

void Image::shiftScaleFilter(float aShiftValue, float aScaleValue)

//----------------------------------------------------------------

{

// Process every pixel of the image

for (unsigned int i = 0; i < m\_width \* m\_height; ++i)

{

// Apply the shilft/scale filter

m\_p\_image[i] = (m\_p\_image[i] + aShiftValue) \* aScaleValue;

}

}

//----------------------------------------

void Image::loadPGM(const char\* aFileName)

//----------------------------------------

{

// Open the file

std::ifstream input\_file(aFileName, std::ifstream::binary);

// The file does not exist

if (!input\_file.is\_open())

{

// Build the error message

std::stringstream error\_message;

error\_message << "Cannot open the file \"" << aFileName << "\". It does not exist";

// Throw an error

throw (error\_message.str());

}

// The file is open

else

{

// Release the memory if necessary

destroy();

// Variable to store a line

char p\_line\_data[LINE\_SIZE];

// Get the first line

input\_file.getline(p\_line\_data, LINE\_SIZE);

// Get the image type

std::string image\_type(p\_line\_data);

// Valid ASCII format

if (image\_type == "P2")

{

// Variable to save the max value

int max\_value(-1);

// There is data to read

unsigned int pixel\_count(0);

while (input\_file.good())

{

// Get the new line

input\_file.getline(p\_line\_data, LINE\_SIZE);

// It is not a comment

if (p\_line\_data[0] != '#')

{

// Store the line in a stream

std::stringstream stream\_line;

stream\_line << std::string(p\_line\_data);

// The memory is not allocated

if (!m\_p\_image && !m\_width && !m\_height)

{

// Load the image size

stream\_line >> m\_width >> m\_height;

// Alocate the memory

m\_p\_image = new float[m\_width \* m\_height];

// Out of memory

if (!m\_p\_image)

{

throw ("Out of memory");

}

}

// The max value is not set

else if (max\_value < 0)

{

// Get the max value;

stream\_line >> max\_value;

}

// Read the pixel data

else

{

// Process all the pixels of the line

while (stream\_line.good())

{

// Get the pixel value

int pixel\_value(-1);

stream\_line >> pixel\_value;

// The pixel exists

if (pixel\_count < m\_width \* m\_height)

{

m\_p\_image[pixel\_count++] = pixel\_value;

}

}

}

}

}

}

// Valid binary format

else if (image\_type == "P5")

{

// Variable to save the max value

int max\_value(-1);

// There is data to read

unsigned int pixel\_count(0);

while (input\_file.good() && !pixel\_count)

{

// Process as an ASCII file

if (!m\_width || !m\_height || max\_value < 0)

{

// Get the new line

input\_file.getline(p\_line\_data, LINE\_SIZE);

// It is not a comment

if (p\_line\_data[0] != '#')

{

// Store the line in a stream

std::stringstream stream\_line;

stream\_line << std::string(p\_line\_data);

// The memory is not allocated

if (!m\_p\_image && !m\_width && !m\_height)

{

// Load the image size

stream\_line >> m\_width >> m\_height;

// Allocate the memory

m\_p\_image = new float[m\_width \* m\_height];

// Out of memory

if (!m\_p\_image)

{

throw ("Out of memory");

}

}

// The max value is not set

else

{

// Get the max value;

stream\_line >> max\_value;

}

}

}

// Read the pixel data

else

{

unsigned char\* p\_temp(new unsigned char[m\_width \* m\_height]);

// Out of memory

if (!p\_temp)

{

throw ("Out of memory");

}

input\_file.read(reinterpret\_cast<char\*>(p\_temp), m\_width \* m\_height);

for (unsigned int i(0); i < m\_width \* m\_height; ++i)

{

++pixel\_count;

m\_p\_image[i] = p\_temp[i];

}

delete [] p\_temp;

}

}

}

// Invalid format

else

{

// Build the error message

std::stringstream error\_message;

error\_message << "Invalid file (\"" << aFileName << "\")";

// Throw an error

throw (error\_message.str());

}

}

}

//-----------------------------------------------

void Image::loadPGM(const std::string& aFileName)

//-----------------------------------------------

{

loadPGM(aFileName.data());

}

//----------------------------------------

void Image::savePGM(const char\* aFileName)

//----------------------------------------

{

// Open the file

std::ofstream output\_file(aFileName);

// The file does not exist

if (!output\_file.is\_open())

{

// Build the error message

std::stringstream error\_message;

error\_message << "Cannot create the file \"" << aFileName << "\"";

// Throw an error

throw (error\_message.str());

}

// The file is open

else

{

// Set the image type

output\_file << "P2" << std::endl;

// Print a comment

output\_file << "# ICP3038 -- Assignment 1 -- 2015/2016" << std::endl;

// The image size

output\_file << m\_width << " " << m\_height << std::endl;

// The get the max value

output\_file << std::min(255, std::max(0, int(getMaxValue()))) << std::endl;

// Process every line

for (unsigned int j = 0; j < m\_height; ++j)

{

// Process every column

for (unsigned int i = 0; i < m\_width; ++i)

{

// Process the pixel

int pixel\_value(m\_p\_image[j \* m\_width + i]);

pixel\_value = std::max(0, pixel\_value);

pixel\_value = std::min(255, pixel\_value);

output\_file << pixel\_value;

// It is not the last pixel of the line

if (i < (m\_width - 1))

{

output\_file << " ";

}

}

// It is not the last line of the image

if (j < (m\_height - 1))

{

output\_file << std::endl;

}

}

}

}

//-----------------------------------------------

void Image::savePGM(const std::string& aFileName)

//-----------------------------------------------

{

savePGM(aFileName.data());

}

/\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* @file test.cpp

\*

\* @brief TESTER FOR IMAGE CLASS

\*

\* @version 1.0

\*

\* @date 25/10/2016

\*

\* @author Dorian B. Dressler (eeu436@bangor@ac.uk)

\*

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Include

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <sstream>

#include <iostream>

#include <exception>

#include "Image.h"

//-----------------------------

int main(int argc, char\*\* argv)

//-----------------------------

{

// Return code

int error\_code(0);

// Catch exceptions

try

{

// Good number of arguments

if (argc == 3)//prev 3

{

// Load an image

Image test\_image;

Image test\_image2;

//Image test\_image2;

test\_image.loadPGM(argv[1]);

//test\_image2.loadPGM(argv[2]);

// Save the image

char\* fileName = (argv[2]);

//test\_image.savePGM(argv[3]);

Image test\_image3;

std::cout << test\_image.getAspectRatio() << std::endl;

//std::cout << test\_image.getHeight() << std::endl;

//std::cout << test\_image.getWidth() << std::endl;

//std::cout << test\_image.getPixel(25, 25) << std::endl;

//test\_image.setPixel(25, 25, 125);

//test\_image.savePGM(fileName);

//std::cout << test\_image.getPixel(25, 25) << std::endl;

//std::cout << test\_image.getAspectRatio() << std::endl;

//test\_image3 = test\_image / (2.5);

//test\_image3.savePGM(fileName);

//(!(test\_image)).savePGM(fileName);

//test\_image3.savePGM(fileName);

//(!(test\_image)).savePGM("test.pgm");

//Create second image to perform tests on

//Image test\_image3(test\_image);

/\*std::cout << test\_image3.getPixel(25, 25) << std::endl;

test\_image3.setPixel(25, 25, 100);

std::cout << test\_image3.getPixel(25, 25) << std::endl;

test\_image3.savePGM("Result.pgm");\*/

//(test\_image -= (test\_image2)).savePGM("Result.pgm");

//(!(test\_image)).savePGM("result.pgm");

//test\_image2 = test\_image + (100.0);

//test\_image3.savePGM("ResultThree.pgm");

//Image test\_image3;

//test\_image3 = test\_image+(test\_image2);

//test\_image3.savePGM("result.pgm");

//----------------------

//testing plus operator again

//test\_image3 = test\_image / (100.0);

//test\_image3 = test\_image /= (50.0);

//-----------------------

//testing + operator

//test\_img2 = test\_image + (100);

//test\_img2.savePGM("+operator\_test.pgm");

//(!(test\_image)).savePGM("result.pgm");

//test\_image3.savePGM("result.pgm");

//Image test\_img3;

//test\_img2 = test\_image - (100);

//test\_img3 = test\_img2 -= (test\_image);

//Image = test\_image += (test\_image);

//test\_img2 -= (100);

//-------------------------

//Testing ! operator

//(!test\_image).savePGM("test3.pgm");

//test\_img2 = !(test\_image);

//test\_img2.savePGM("test4.pgm");

}

// Wrong number of argument

else

{

// Build the error message

std::stringstream error\_message;

error\_message << "Wrong number of arguments, usage:" << std::endl;

error\_message << "\t" << argv[0] << " input\_file\_name.pgm output\_file\_name.pgm" << std::endl;

// Throw the error

throw (error\_message.str());

}

}

// An error occured

catch (const std::exception& error)

{

error\_code = 1;

std::cerr << error.what() << std::endl;

}

catch (const std::string& error)

{

error\_code = 1;

std::cerr << error << std::endl;

}

catch (const char\* error)

{

error\_code = 1;

std::cerr << error << std::endl;

}

catch (...)

{

error\_code = 1;

std::cerr << "Unknown error" << std::endl;

}

return (error\_code);

}