OBJECT ORIENTED PROGRAMMING

LABORATORY 4

OBJECTIVES

With this laboratory we'll step into the realm of Object Oriented Programming (OOP). You will learn how to write simple classes in C++, how to use encapsulation and abstraction, the different types of constructors a class can have and how to overload operators.

PROPOSED PROBLEMS

1. Write a class to represent a complex number. The class should contain all the methods you implemented in the complex number module from the previous laboratory; so you could just refactor the code that you already have.
In addition, your class should contain at least:

- Default constructor and a constructor that initializes a complex number with the real and imaginary parts.
- Overloads for the equal to and not equal to operators.
- Overload for the stream insertion operator (>>) and stream extraction (<<) operators.
- Overloads for the arithmetic operators: addition (+), subtraction (-), multiplication(*) and division(/).

Declare and define multiple objects of the class Complex such that they are allocated on the stack and heap, respectively. Test all your methods and operators you wrote.

2. Transform the module you wrote for the dynamic array into a class. To start write the following :

- Default constructor;
- Destructor;
- Insert at element at the end of the array;
- Remove the last element of the array;
- Getters for the capacity and the length of the array;
- Overload for the stream insertion operator (<<);
- Overload for the indexing/subscript operator ([]). This should return a reference to the i^{th} element in the array.

Now, run the following code snippet (assuming that your class is called *DynamicArray* and that it stores integer values):

```
DynamicArray arr1;
arr1.insert(1);
arr1.insert(2);

DynamicArray arr2 = arr1;
DynamicArray arr3;
arr3 = arr2;

arr2[0] = 10;
arr3[1] = 7;

cout<<"The first array is: "<<arr1<<endl;
cout<<"The second array is: "<<arr2<<endl;
cout<<"The third array is: "<<arr3<<endl;
```

What do you notice? ☺ Why is that?

Now, implement the copy constructor and the assignment operator for the *DynamicArray* class.

Create **static** member variables that will tell you how many instances of *DynamicArray* were created, how many times the copy constructor was called, and how many times the assignment operator was called.

3. Load the data from the file complex_numers.txt into a dynamic array. This array contains a complex number per line, and, on each line, the real and imaginary parts are separated by a space.
Keep in mind that you don't know the size of the array in advance!
Check this tutorial on how to read data from a text file:
https://www.tutorialspoint.com/cplusplus/cpp_files_streams.htm.
Finally, compute the sum of all the numbers in that array.

Prove that your code doesn't have any memory leaks!

4. Write unit test for this class using Test Explorer:
https://docs.microsoft.com/en-us/visualstudio/test/writing-unit-tests-for-c-cpp?view=vs-2019

Exercises 1, 2, 3 and 4 from this lab are mandatory!

EXTRA CREDIT

For the second extra credit assignment you will develop a basic image processing application. Although it seems complicated at first, you will see that by the end of next week you will master (or you should) all the skills required to solve it.
With this assignment you will get a change to exercise pointers, classes, inheritance, namespaces, file input-output, as well as developing a pretty cool application ☺.

THE IMAGE CLASS

As we are dealing with an image processing application, let's start with its main building block: the abstract data type that represents an image. For this assignment, you will work with grayscale images. A grayscale image is represented by a 2D array of pixels, and each pixel can take a value between [0, 255].

The value 0 represents black, 255 is white, and values between represent different gray level intensities (the larger the value, the lighter the pixel is):

An example of how you could define an image class is presented below:

```
class Image
{
public:
   Image();
   Image(unsigned int w, unsigned int h);
   Image(const Image &other);

   ~Image();

   bool load(std::string imagePath);
   bool save(std::string imagePath);
```

```
    Image& operator=(const Image &other);

    Image operator+(const Image &i);
    Image operator-(const Image &i);
    Image operator*(const Image &i);

    bool getROI(Image &roiImg, Rectangle roiRect);
    bool getROI(Image &roiImg, unsigned int x, unsigned int y, unsigned int width, unsigned int
height);

    bool isEmpty() const;

    Size size() const;

    unsigned int width() const;
    unsigned int height() const;

    unsigned char& at(unsigned int x, unsigned int y);
    unsigned char& at(Point pt);

    unsigned char* row(int y);

    void release();

    friend std::ostream& operator<<(std::ostream& os, const Image& dt);

    static Image zeros(unsigned int width, unsigned int height);
    static Image ones(unsigned int width, unsigned int height);

private:
    unsigned char** m_data;
    unsigned int m_width;
    unsigned int m_height;
};
```

Of course, you can choose (and argue your decision) other ways of representing an image, but the class that you write should have at least the methods and operators:

- copy constructor;
- copy assignment operator;
- getters for the height and width of the image. You should also define a class to represent the size (width and height of an image) and have a getter that returns the size of the image in this format;
- two methods that load and save an image from/to a *.pgm* file. These methods should return a Boolean value that indicates if the operation was successful. More details about the *.pgm* file format can be found here: http://netpbm.sourceforge.net/doc/pgm.html ;
- overload for the stream insertion operator: operator<<; this would display the image pixels in a 2D grid. For a pretty print, you could use *setw*[1] to format the way you display these pixels (set *setw*(3) to display the pixels on 3 characters).
- overload for the +, -, and * arithmetic operators between two images. The precondition for this is that both images should have the same size. The operations will be applied element-wise!
- overload for the +, - and * arithmetic operator between an image and a scalar value;
- a method that returns whether the image is empty or not;
- a method that returns a reference to the pixel at a given location in the image. This method should have two overloads: one in which you specify the pixel's position with two values (x and y coordinates) and another one in which you specify the pixel's position with a Point class (that has two fields x and y);
- a method that returns a pointer to a row in the image;
- a method that releases the memory allocated for the image. You could call this method in the destructor of the class;
- in image processing, a region of interest (ROI) is a portion of the image that you want to apply to filter or apply some operation on. This is usually a rectangular region over the image. Write two methods that return a new Image that contains the pixels within a ROI specified as a parameter (you should perform a deep copy of the image pixels). Also, check that the ROI is valid (is within image bounds). You should use only pointer arithmetic for this (so don't use the indexing operator []);
- some static methods that create some "special" types of images: an image filled with zeros and an image filled with ones.

[1] http://www.cplusplus.com/reference/iomanip/setw/

IMAGE PROCESSING TASKS

For these tasks you will first start by writing a base class for image processing: *ImageProcessing,* The class will have a pure virtual method:

void process(const Image& src, Image& dst);

which will be implemented by all its subclasses. Then you will write a subclass for each image processing operation. You will have to implement the following operations:

1. **Brightness and contrast adjustment.**

This processing type operates at each pixel independently and alters the brightness and contrast of the image, as follows:

$$F(x, y) = \alpha \cdot I(x, y) + \beta$$

, where I(x, y) and F(x, y) represent the pixels at coordinates (x, y) from the input image and the filtered image respectively.

The parameters of this operation $\alpha>0$ (the gain) and β (the bias) can be considered to control the control *contrast* and *brightness* of the image. These parameters will be specified in the constructor of the class. The default constructor initializes $\alpha = 1$ and $\beta = 0$ (so, in this case, this operation is a no-op).
Basically, you should just iterate through each image pixel and apply the transformation to each pixel. Make sure that the result is in the range [0, 255]. If not, clip the result to this interval!

2. **Gamma correction.**

Gamma correction is used to correct the overall brightness of an input image with a non-linear transformation function:

$$F(x, y) = I(x, y)^{\gamma}$$

, where *F(x, y)* and *I(x, y)* are the values of the pixels at coordinates *(x, y)* in the filtered image and input image, respectively, and γ is the gamma encoding factor.
This operation is needed because humans perceive light in a non-linear manner; the human eye is more sensitive to changes in dark tones than to changes in lighter ones. So, gamma correction is used to optimize the usage of bytes when encoding an image, by taking into consideration this perception difference between the camera sensor and our eyes.

A value of γ less than 1 will darken the image, while a value larger than 1 will make the image appear brighter. If γ = 1 then the image will be left unchanged (no-op).

The value of should be passed as a parameter to the constructor of this class.

For further details about this operation, you can check the following websites: https://www.cambridgeincolour.com/tutorials/gamma-correction.htm https://en.wikipedia.org/wiki/Gamma_correction.

3. Image convolution

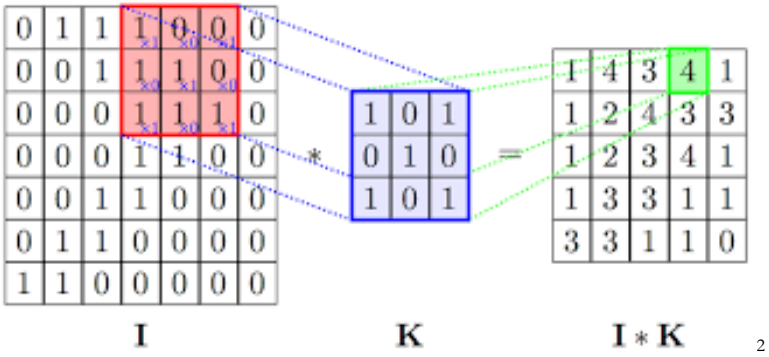
Convolutions are highly used in image processing and machine learning to extract some features from the input image. They involve the usage of a kernel (or filter) that is convolved over the input image as follows:

$$F = K * I$$

, where F is the output (filtered) image, K is the convolutional kernel and I is the input image. The convolutional kernel K has the shape (w, h), where w and h are usually odd numbers: w = 2k + 1.

At each image position (x, y), the convolutional filter is applied as follows:

$$F(x,y) = \sum_{u=0}^{w-1} \sum_{v=0}^{h-1} K(u,v) I(x-u+k, y-v+k)$$



[2]

Figure 1. Illustration of image convolution.

A nice animation for image convolution can be found here:
https://upload.wikimedia.org/wikipedia/commons/1/19/2D_Convolution_Animation.gif
When implementing this operation you should pay attention to the image bounds!

[2] Image source: https://blog.francium.tech/machine-learning-convolution-for-image-processing-42623c8dbec0

The constructor of this function should also take a parameter a pointer to a function that is used to scale the result of applying the convolution kernel to the range [0, 255].

For further details about convolution you can check this page:
https://en.wikipedia.org/wiki/Kernel_(image_processing)
Some well-known convolutional kernels are:

0	0	0	**Identity kernel**. No-op. Applying convolution with this kernel results in the same image.
0	1	0	
0	0	0	

1	1	1	**Mean blur kernel**. (for this kernel, the scaling function should just divide the convolution result to 1/9)
1	1	1	
1	1	1	

1	2	1	**3x3 Gaussian blur kernel**. Still a blurring kernel, but in this case, the central pixel of the kernel has a higher weight in the mean operation; for this kernel, the scaling function should just divide the convolution result to 1/16.0)
2	4	2	
1	2	1	

1	2	1	**Horizontal Sobel kernel.** Used to detect horizontal edges. In this case, the scaling function should be a linear mapping function that converts the range [-4*255, 4*255] to the range [0, 255]).
0	0	0	
-1	-2	-1	

-1	0	1	**Vertical Sobel kernel.** Used to detect vertical edges. In this case, the scaling function should be a linear mapping function that converts the range [-4*255, 4*255] to the range [0, 255]).
-2	0	2	
-1	0	1	

DRAWING MODULE

Next, you will implement a module to draw different shapes over an image.

Write a class to represent a 2D point (as you noticed, this class is also used in the *Image* class to get a reference to the value of a pixel in the image). This class should contain the x and y coordinates of a point (integer values). Overload the stream insertion and extraction operator for this class. Write a default constructor, that initializes the fields to 0, and a parameterized constructor.

Write a class to represent a Rectangle (as you noticed, this class is also used in the *Image* class to get a region of interest in the image). This class should contain the x and y coordinates of the top-left point, and the width and height of the rectangle. Overload the stream insertion and extraction operator for this class. Write a default constructor that initializes the fields to 0 and a parameterized constructor. Also, write a constructor that takes as parameters two 2D Points (the top left and bottom-right coordinates of the rectangle).

This class should also overload the following operators:

- Addition and subtraction operators: this will be used to translate the rectangle.
- & and | operators. The & operator will compute the intersection between two rectangles, while the | operator will compute the union of two rectangles.

Now that you have classes to represent basic shapes, let's use them in your image processing library. Create a *module* for drawing some shapes over images. You should have at least the following functions in this module[3]:

- drawCircle(Image& img, Point center, int radius, unsigned char color);
- drawLine(Image &img, Point p1, Point p2, unsigned char color);
- drawRectangle(Image &img, Rect r, unsigned char color);
- drawRectangle(Image &img, Point tl, Point br, unsigned char color);

REQUIREMENTS

- Draw the class diagram for your solution.
- Group all the classes and modules that you wrote into a **namespace**.
- You should use a consistent coding style throughout this project.
- Provide tests for all the code that you wrote. The code coverage should be at least 80%.
- Prove that your project doesn't have any memory leaks.
- Provide specifications for all your classes and functions and write the comments in *doxygen* format. Generate the documentation in html format.
- **Pay attention to the range of values that a pixel can have (when performing different operations on an image make sure that the result is between [0, 255]. If the value is less than 0 or larger than 255, you should clip it to this interval)!**
- For extra-extra-extra credit ☺, try to extend this solution to work with color images (in this case this) http://netpbm.sourceforge.net/doc/ppm.html

[3] In practice more complex rasterization algorithms are used to convert 2D shapes into a rasterized format, but for this simple application it is ok if you just use the basic formulas for the shapes to determine the position of the points that lie on the shape's boundary.

APPENDIX

This appendix shows the result of applying the image processing operations of the following image:

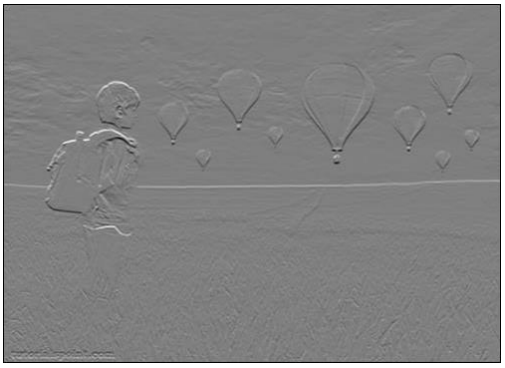
Brightenss alteration. The images above show the result of appling the *Brightenss contrast color adjustment* operation with the values for $\beta = -30$ (left) and $\beta = -70$ (right).

You can notice that negative β values make the resulting image darker. The lower β is, the darker the image is.

Brightenss alteration. The images above show the result of appling the *Brightenss contrast color adjustment* operation with the values for $\beta = +30$ (left) and $\beta = +70$ (right).
You can notice that positive β values make the resulting image brighter. The higher β is, the brighter the image becomes.

Contrast alteration. The images above show the result of appling the *Brightenss contrast color adjustment* operation with the values for $\alpha = 0.5$ (left) and $\alpha = 2$ (right).
You can notice that values of α less than 1 make the image darker, while values of α larger than 1 make the image brighter. As opposed to the previous operator (brightness alteration), you cano notice that the greylevel tones are uniformly scaled.

Convolution – mean blur and Gaussian blur. The images above show the result of appling the *convolution* operation with the kernels for mean blur and for Gaussian blur.



Convolution – horizontal and vertical Sobel kernels. The images above show the result of appling the *convolution* operation with the Sobel horizontal kernel (left) and Sobel vertical kernel (right).

These kernels emphasize on the edges that are present in the images. We already know that whiter tones represent bigger pixel values. If you look at the result of these images, you can notice that after applying the horizontal Soebel kernel, we get brighter pixels where there is a transition in the horizontal direction in the image. On the other hand, after applying the horizontal Soebel kernel, we get brighter pixels where there is a transition in the vertical direction in the image.