HACETTEPE UNIVERSITY - COMPUTER ENGINEERING

Programming Assignment 1

DEADLINE: FRIDAY, 25/10/2024 AT 23:59:59

BBM203 Software Practicum I - Fall 2024

CONFIDENTIAL

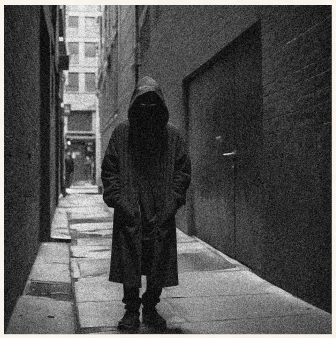
Topics: Arrays, Matrices, Dynamic Memory Allocation, Operator Overloading, File I/O

Course Instructors: Assoc. Prof. Dr. Adnan ÖZSOY, Asst. Prof. Dr. Engin DEMİR, Assoc. Prof. Dr. Hacer YALIM KELEŞ
TAs: M. Aslı Taşgetiren, S. Meryem Taşyürek, **Asst. Prof. Dr. Selma DİLEK***
Programming Language: C++11 - **You MUST USE this starter code**
Due Date: **Friday, 25/10/2024 (23:59:59)** over **https://submit.cs.hacettepe.edu.tr**

ClearVision

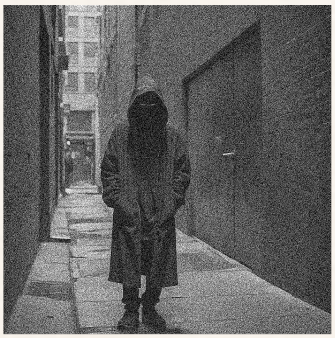
Vision Beyond the Visible: Image Processing for Defense

In a world where information is power, and national security hinges on the protection and analysis of visual data, the Turkish defense industry has initiated a classified mission in collaboration with Hacettepe University. This mission involves the development of an advanced application known as `ClearVision Defense`, a top-secret program designed to enhance and safeguard critical imagery used in defense operations. The government has entrusted this highly sensitive task to the brightest minds in the Computer Engineering department, with one goal: to build robust image filters that can improve image clarity and security, while embedding and decrypting confidential messages hidden within the images.


(a) Original image


(b) Mean filtered


c) Gaussian smoothed


d) Unsharp filtered

In the world of defense, every pixel holds the potential to reveal crucial information. The ability to extract hidden details from noisy or obscured images can mean the difference between victory and failure in covert operations. Whether its analyzing satellite reconnaissance, improving low-light surveillance, or decoding classified messages embedded within seemingly harmless images, `ClearVision Defense` must be the ultimate tool in Turkeys arsenal for image enhancement and encryption.

As top-tier engineering students, you have been specially selected for this classified operation. Your mission is to develop powerful denoising filters that can make sense of noisy data and create an impenetrable method for hiding and retrieving encrypted messages in images. The success of this project depends on your expertise.

Good luck!

CONFIDENTIAL

1 Reading the Input Data and Initializing the Image(s)

In this section, we explain how to initialize the program with the provided input data. You will interact with the program through a series of command-line arguments, where you specify the operation to be performed (filtering, image comparison, or message encryption/decryption) as well as the corresponding input files and parameters. **Pay close attention to the required dynamic memory allocation when handling the image data**.

1.1 Input Files and Command Line Arguments

The input file(s) containing the grayscale image(s) will be provided in `PNG` or `JPG` format as the command-line arguments. You will use the `stb_image.h` library to load the image data into a **2D array of grayscale pixel values** within the `GrayscaleImage` class.

The pixel values of the image are stored in a matrix form, where each element represents a grayscale intensity between 0 (black) and 255 (white). You will implement filters that modify the pixel intensities based on surrounding values, and potentially decrypt or encrypt hidden information in the least significant bits of these values.

0	30	63	128
86	128	172	221
172	221	240	255

1.2 Using Command-Line Arguments

Your program will be invoked via command-line arguments, which determine the operation to be performed. Below is a description of the available operations and the associated arguments:

Operation Command-Line Arguments	Description
`mean <img> <kernel_size>`	Applies a Mean filter to the input image using a kernel of size `kernel_size`.
`gauss <img> <kernel_size> <sigma>`	Applies Gaussian smoothing to the input image with the specified kernel size and `sigma` value.
`unsharp <img> <kernel_size> <amount>`	Applies an Unsharp Mask filter to the input image using the given kernel size and amount of sharpening.
`add <img1> <img2>`	Adds the pixel values of two images together.
`sub <img1> <img2>`	Subtracts the pixel values of `img2` from `img1`.
`equals <img1> <img2>`	Compares two images to check if they are identical. Prints the result to the console.
`disguise <img>`	Generates a secret image from the given grayscale image and saves it to a .dat file.
`reveal <dat>`	Reads the given .dat file into a secret image, reconstructs it into a grayscale image, and saves it.
`enc <img> <message>`	Embeds the provided secret message into the image by modifying the least significant bits (LSBs) of the pixel values.
`dec <img> <msg_len>`	Extracts and decrypts the hidden message from the image, assuming the message is of length `msg_len` characters.

Table 1: Command-line operations for ClearVision application

Below are a few examples of how a sample run command might look in the terminal:

```
./clearvision mean input_image.png 3
./clearvision gauss input_image.png 3 1.0
./clearvision enc secret_image.png "Kurt sürüsü harekete geçti. Bölgeyi kontrol
↪  edin."
```

1.3 Using `stb_image.h` to Read and Save Image Data

You will use the `stb_image.h` library to load and save grayscale images. Here are the steps involved in loading an image and storing it in the `GrayscaleImage` class:

1. Load the image file using the `stbi_load()` function. This function reads the image from the file and stores its pixel values into an array.

```
int width, height, channels;
unsigned char* image_data = stbi_load("input_image.png", &width, &height,
↪  &channels, STBI_grey);
```

2. Once loaded, the image is represented as an instance of the `GrayscaleImage` class. You will initialize the object with the image data:

```
GrayscaleImage img(image_data, width, height);
```

3. The filters or operations (such as secret message encryption or decryption) can then be applied to the instance(s) of the `GrayscaleImage` class.

4. After processing, you can save the resulting image using the `saveToFile()` function from the `GrayscaleImage` class, which uses the `stbi_write_png()` function internally to write the modified pixel data to a new image file.

```
img.saveToFile("output_image.png");
```

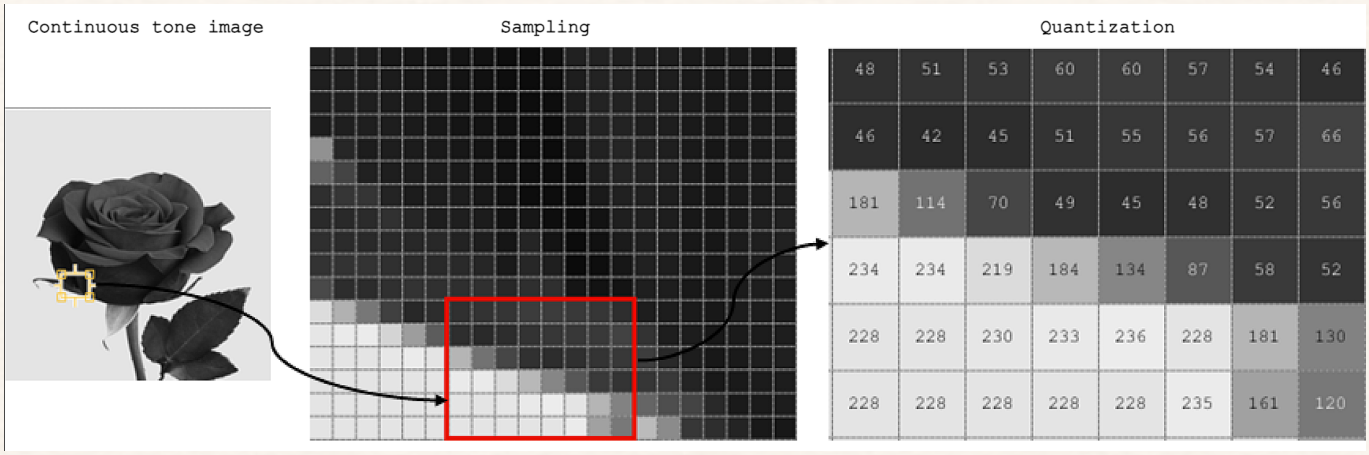


Figure 1: Grayscale image representation and pixel manipulation using `stb_image.h`

1.4 Applications of ClearVision Defense

- **Satellite and Drone Surveillance**: Enhanced images from satellites and drones will be used to detect enemy movements or potential threats. `ClearVision Defense` will remove noise from surveillance images and sharpen critical details.
- **Night Vision Operations**: In low-light environments, the noise in night vision footage can obscure important features. With your filters, the footage will be clearer and help identify objects or persons of interest during defense operations.
- **Classified Intelligence**: Secret messages can be hidden in seemingly harmless images, embedded in the least significant bits of the pixel data. This system will be used by intelligence agencies to securely send and receive encrypted messages through images.

CONFIDENTIAL

2 Part 1: Implementing Image Filters

You will implement two filters to smooth-en and clean up noisy images, and one filter to sharpen a given image. These filters are crucial for enhancing image quality in various defense-related applications, such as satellite imaging, drone footage, and night vision.

2.1 Adding, Subtracting, and Comparing Images

Before implementing filters, you should implement addition, subtraction, and comparison operations on images. These operations are necessary for certain filtering techniques, such as unsharp masking, which require manipulating the pixel intensities between two images. For simplicity, you may assume that the input images will always have the same dimensions. To ensure proper handling of pixel values in addition and subtraction, **you must clamp the resulting pixel values to the range [0, 255], which represents the valid intensity range for grayscale images, to prevent overflow or underflow of pixel intensities**.

Addition: The addition of two images is performed by summing the pixel values at corresponding positions in the two images. The result is clamped so that no pixel value exceeds 255, ensuring it remains within the valid range. You must implement this by **overloading the '+' operator in your** `GrayscaleImage` **class**.

Subtraction: The subtraction of two images is done by subtracting the pixel values at corresponding positions in the two images. As with addition, the result is clamped to ensure it remains within the valid grayscale intensity range (0-255). You must implement this by **overloading the '-' operator in your** `GrayscaleImage` **class**.

Equality Check: The equality check between two images is performed by comparing the pixel values at corresponding positions in both images. Two images are considered equal if they have the same dimensions and all pixel values at corresponding positions are identical. You must implement this by **overloading the == operator in your** `GrayscaleImage` **class**.

Before and After Example: The following figure shows a visual example of adding and subtracting two images.

(a) Original Image 1 (b) Original Image 2 (c) Addition of Images (d) Subtraction of Images

Figure 2: Visual examples of picture arithmetic.

2.2 Mean Filter

- **Purpose**: The mean filter is a basic smoothing technique used to reduce noise in images by replacing each pixel with the average of its surrounding pixels.
- **Common Names:** Mean Filtering, Smoothing, Averaging, Box Filtering
- **Application**: Ideal for surveillance footage or drone imagery, where noise can make it difficult to identify key details, or medical imaging, where it can be used to reduce high-frequency noise in radiological images.

- **Short Explanation**: The mean filter works by calculating the average value of a pixel and its neighbors, making the image appear smoother but potentially blurring fine details.

Description: Mean filtering is a simple and intuitive technique for reducing noise in images by smoothing the intensity variations between neighboring pixels. The key idea behind the mean filter is to replace each pixel value in the image with the mean (or average) value of its neighbors, including itself. This process helps in reducing the prominence of pixel values that are unrepresentative of their surroundings, thus removing noise while preserving the general structure of the image.

Mean filtering is often implemented as a *convolution* operation using a kernel. The kernel defines the neighborhood of pixels that are sampled to compute the mean value. A common choice for the kernel is a 3×3 square kernel, although larger kernels such as 5×5 can be used for stronger smoothing.

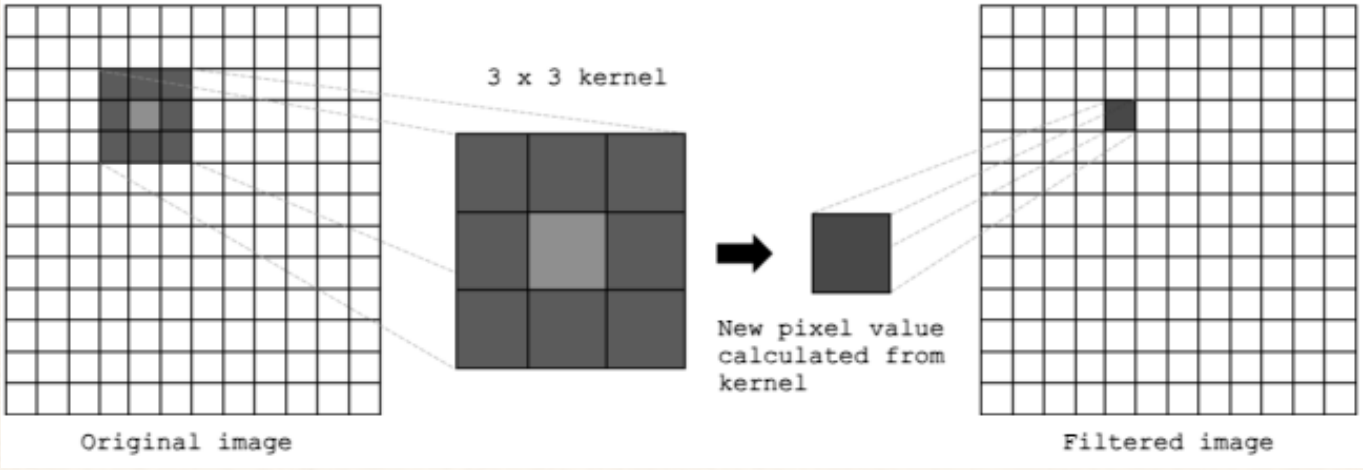


Figure 3: Noise reduction using filtering.

How It Works: The process of mean filtering involves sliding the kernel over the image and computing the average of the pixel values in the neighborhood defined by the kernel. The new pixel value is then assigned as the average of these neighboring pixels. This process is repeated for all pixels in the image. For a 3×3 kernel, the kernel matrix given on the right is used to compute the average of a pixel and its surrounding neighbors. This kernel averages the values of the pixel in question and its eight immediate neighbors. The result is a smoothed version of the original image, with reduced noise.

$$K = \frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Implementation Instructions: You are required to implement the Mean Filter in C++ by completing `apply_mean_filter()` function in `Filter` class using the following steps:

1. For each pixel in the image, extract its neighborhood using a $n \times n$ kernel, where n is the kernel size given as the function parameter.
2. Compute the average of the pixel values within this kernel.
3. Replace the central pixel with the computed average value (you should cast the double values to integer or floor them).
4. Repeat this process for all pixels in the image, ensuring proper handling of edge pixels (by ignoring the edges). Padding is another type of proper handling of edge pixels, but it is out of the scope of this assignment.

You may start by implementing the filter with a 3×3 kernel, and then experiment with larger kernel sizes to see how it affects the smoothing process. **We will only use odd kernel sizes**.

Example: Consider the 5×5 image matrix given on the right. Applying a 3×3 mean filter to the central pixel (position $[2, 2]$) involves computing the average of its neighboring pixels:

$$I = \begin{bmatrix} 120 & 125 & 130 & 135 & 140 \\ 110 & 115 & 125 & 255 & 130 \\ 100 & 105 & 110 & 115 & 120 \\ 90 & 95 & 100 & 105 & 110 \\ 80 & 85 & 90 & 95 & 100 \end{bmatrix}$$

$$\frac{1}{9} \times \begin{bmatrix} 115 & 125 & 255 \\ 105 & 110 & 115 \\ 95 & 100 & 105 \end{bmatrix} = \frac{1}{9} \times (115+125+255+105+110+115+95+100+105) = \frac{1125}{9} = 125$$

Thus, the value of the central pixel $I[2, 2]$ will be updated from 110 to 125.

Before and After Example: You can visualize the before and after effects of applying the Mean Filter as follows:

(a) Original Image (b) Mean Filter (3x3) (c) Mean Filter (11x11) (d) Mean Filter (19x19)

Figure 4: Comparing effects of different kernel sizes for applying Mean Filter on an image.

2.3 Gaussian Smoothing Filter

- **Purpose**: The Gaussian smoothing filter reduces image noise and blurs details using a weighted average around each pixel, where nearby pixels have more influence than those farther away.
- **Common Names:** Gaussian Smoothing, Gaussian Blur
- **Application**: Used for satellite imaging and aerial surveillance, where noise from atmospheric conditions can distort the image, and in medical imaging to enhance important regions while removing high-frequency noise.
- **Short Explanation**: The Gaussian filter applies a bell-shaped weight distribution (Gaussian function) to the neighboring pixels, smoothing the image while preserving more detail compared to the mean filter.

Description: The Gaussian smoothing filter is a more advanced method of reducing noise in an image compared to the mean filter. Instead of treating all neighboring pixels equally, the Gaussian filter assigns a higher weight to pixels closer to the center and less weight to pixels farther away. This weighted average is based on the **Gaussian distribution** (bell curve), which ensures that nearby pixels have a stronger influence on the central pixel than distant ones. The mathematical form of a 2D Gaussian function is given by:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

where:

- x and y are the pixel distances from the center of the kernel.
- σ (sigma) is the standard deviation, controlling the amount of smoothing (larger σ means more smoothing).

- $G(x, y)$ is the weight assigned to the pixel located at (x, y) in the kernel.

The kernel is a matrix that holds the weights for the neighboring pixels. Larger values of σ lead to larger kernels, and this kernel is applied to each pixel in the image through convolution, much like the mean filter.

Implementation Instructions: You are required to implement the Gaussian Smoothing Filter in C++ by completing the `apply_gaussian_smoothing()` function in the `Filter` class using the following steps:

1. Create a Gaussian kernel of size $n \times n$, where n is the kernel size provided as an input argument to the function. The kernel values are computed using the Gaussian formula, which depends on the distance from the center of the kernel and the standard deviation σ (provided as another function argument).
2. Normalize the kernel by dividing each element by the sum of all elements, so that the weights sum to 1. This ensures the smoothing effect maintains the overall brightness of the image.
3. Slide the kernel over each pixel in the image, computing the weighted average of the neighboring pixels. For each pixel, multiply the neighboring pixels by the corresponding kernel values, sum the result, and replace the pixel with this weighted sum (you should cast the double values to integer or floor them).
4. Handle boundary pixels by ignoring neighbors outside the image bounds. Only valid pixels within the image boundaries should contribute to the weighted sum.
5. Apply this process to all pixels in the image (note: padding is out of scope for this assignment, so edge pixels should be treated with boundary checking.)

Example: Consider a simple 5×5 image matrix given below on the left. We will apply a 3×3 Gaussian kernel with $\sigma = 1.0$ to the pixel at position $[2, 2]$ (marked in red). The corresponding Gaussian kernel matrix is shown on the right:

$$I = \begin{bmatrix} 10 & 20 & 30 & 40 & 50 \\ 60 & 70 & 80 & 90 & 100 \\ 110 & 120 & 130 & 140 & 150 \\ 160 & 170 & 180 & 190 & 200 \\ 210 & 220 & 230 & 240 & 250 \end{bmatrix} \qquad K = \frac{1}{16} \times \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

To compute the smoothed value of pixel $I[2, 2]$ (originally 130), we multiply each pixel in the 3×3 neighborhood by the corresponding value in the Gaussian kernel and then sum the results:

$$\frac{1}{16} \times (1 \cdot 70 + 2 \cdot 80 + 1 \cdot 90 + 2 \cdot 120 + 4 \cdot 130 + 2 \cdot 140 + 1 \cdot 170 + 2 \cdot 180 + 1 \cdot 190)$$
$$= \frac{1}{16} \times (70 + 160 + 90 + 240 + 520 + 280 + 170 + 360 + 190) = \frac{2080}{16} = 130$$

Thus, the pixel value $I[2, 2]$ remains 130, showing how the Gaussian filter smooths the pixel values while preserving important details like edges.

Before and After Example: You can visualize the effects of applying the Gaussian filter as shown in the figure below, where different kernel sizes (k) and σ values result in varying degrees of smoothing.

(a) Original Image (b) Gaussian Filter ($k = 21,\ \sigma = 2.0$) (c) Gaussian Filter ($k = 41,\ \sigma = 2.0$) (d) Gaussian Filter ($k = 21,\ \sigma = 4.0$) (e) Gaussian Filter ($k = 41,\ \sigma = 4.0$)

Figure 5: Effects of varying σ values on Gaussian Smoothing.

2.4 Unsharp Masking Filter

- **Purpose**: This filter sharpens the image by emphasizing the edges, which helps make important features more visible while preserving the clarity of key details.
- **Application**: Useful in target detection and recognition systems, where its critical to enhance features like edges of objects or faces in low-light conditions.
- **Short Explanation**: The unsharp masking filter subtracts a blurred version of the image from the original to create a sharper result. It highlights edges and details.

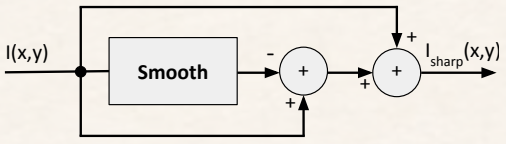
Description: The Unsharp Masking filter enhances image details by sharpening the image. The main idea is to create a sharper image by subtracting a blurred (or unsharp) version of the original image from the image itself. The process is often referred to as **highpass filtering** because it essentially removes low-frequency components of the image, which correspond to smooth areas, and enhances high-frequency components like edges and details.

The Unsharp Masking process involves three main steps:

1. **Blurring (Smoothing) the Original Image**: Apply a Gaussian smoothing filter to the image to create a blurred (unsharp) version of the image.
2. **Subtracting the Blurred (Smoothed) Image**: Subtract the blurred image from the original image to isolate the high-frequency components (edges).
3. **Enhancing the Edges**: Add the weighted high-frequency components back to the original image to sharpen the details. A scaling factor k is applied to control the level of sharpening.

Formula: The equation for the Unsharp Masking filter is:

$$I_{\text{sharp}}(x, y) = I(x, y) + k \cdot (I(x, y) - I_{\text{smooth}}(x, y))$$



Where:

- I_{sharp}: The resulting sharpened image.
- I: The original image.

- I_{smooth}: The smoothed version of the original image, typically obtained using a Gaussian smoothing filter.
- k: A scaling constant that controls the amount of sharpening. Typical values for k range from 0.2 to 0.7.

The term $(I - I_{\text{smooth}})$ is often called the "edge" or "high-frequency" component of the image. By adding this term back to the original image, we enhance the edges and sharpen the image.

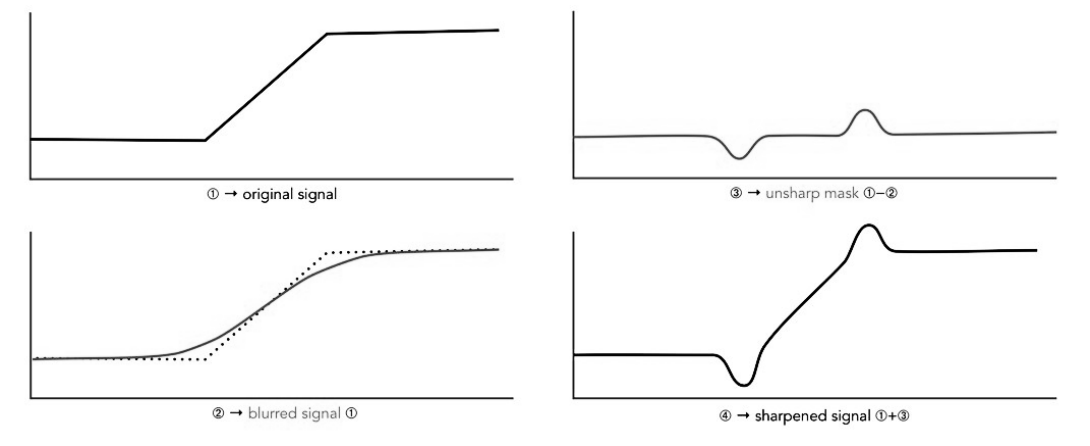


Figure 6: Visual representation of the Unsharp Masking process.

Implementation Instructions: You are required to implement the Unsharp Masking filter in C++ by completing the `apply_unsharp_mask()` function in the `Filter` class using the following steps:

1. **Apply Gaussian Smoothing**: Blur the original image using the Gaussian Smoothing filter. The kernel size will be provided as the input argument and the standard deviation should be taken as $\sigma = 1.0$.
2. **Calculate the Difference Image (Edge Image)**: Subtract the blurred image from the original image to calculate the "edge image" or high-frequency component.
3. **Enhance the Edges**: Multiply the edge image by the scaling factor k provided as the input argument, which controls the amount of sharpening.
4. **Add Back the Edge Image**: Add the scaled edge image back to the original image to enhance the edges and produce a sharpened result.

Before and After Example: The following figure shows a visual example of applying the Unsharp Masking filter on an image, with varying amounts of sharpening and kernel size of 9x9.

(a) Original Image (b) Unsharp Mask (k=1) (c) Unsharp Mask (k=5) (d) Unsharp Mask (k=10)

Figure 7: Visual example of Unsharp Masking with varying sharpening constants.

CONFIDENTIAL

3 Part 2: Secret Image Handling

In addition to enhancing images, `ClearVision Defense` requires you to securely store and process sensitive information. This part of the project involves splitting images into upper and lower triangular matrices for secure storage, as well as encrypting and decrypting secret messages hidden within the image's pixels.

3.1 What are Upper and Lower Triangular Matrices?

A triangular matrix is a square matrix where all the elements below (or above) the main diagonal are zero. The main diagonal runs from the top-left to the bottom-right of the matrix. These matrices are commonly used in numerical analysis and encryption because they can efficiently store and manipulate data.

Upper Triangular Matrix: This matrix has all zero elements below the main diagonal. Only the elements on and above the diagonal are stored.

Lower Triangular Matrix: This matrix has all zero elements above the main diagonal. Only the elements on and below the diagonal are stored.

Storage as 1D Arrays: To store these matrices efficiently, we flatten the upper and lower triangular matrices into 1D arrays. This saves memory by avoiding the storage of zero elements. For example, a 5×5 upper triangular matrix has 15 non-zero elements, so it will be stored as a 1D array of length 15. Similarly, the lower triangular matrix will be stored as a 1D array, excluding the diagonal, which is stored only once in the upper triangular matrix.

Example: Consider the following 5×5 image matrix:

$$I = \begin{bmatrix} 10 & 20 & 30 & 40 & 50 \\ 60 & 70 & 80 & 90 & 100 \\ 110 & 120 & 130 & 140 & 150 \\ 160 & 170 & 180 & 190 & 200 \\ 210 & 220 & 230 & 240 & 250 \end{bmatrix}$$

From this matrix, we can extract the upper and lower triangular matrices:

$$\text{Upper Triangular} = \begin{bmatrix} 10 & 20 & 30 & 40 & 50 \\ 0 & 70 & 80 & 90 & 100 \\ 0 & 0 & 130 & 140 & 150 \\ 0 & 0 & 0 & 190 & 200 \\ 0 & 0 & 0 & 0 & 250 \end{bmatrix} \quad \text{Lower Triangular} = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 \\ 60 & 70 & 0 & 0 & 0 \\ 110 & 120 & 130 & 0 & 0 \\ 160 & 170 & 180 & 190 & 0 \\ 210 & 220 & 230 & 240 & 250 \end{bmatrix}$$

The flattened 1D array for the upper triangular matrix would store the following values:

$$\text{Upper Triangular (1D array)} = [10, 20, 30, 40, 50, 70, 80, 90, 100, 130, 140, 150, 190, 200, 250]$$

And the lower triangular matrix (excluding the diagonal) would be stored as:

$$\text{Lower Triangular (1D array)} = [60, 110, 120, 160, 170, 180, 210, 220, 230, 240]$$

Purpose in `ClearVision Defense`: For security reasons, images are split into these triangular matrices and stored separately. The **upper triangular matrix** stores all the pixel values on and above the main diagonal, while the **lower triangular matrix** stores only the pixel values strictly below the main diagonal. **The diagonal itself is stored only once in the upper triangular matrix**, avoiding redundancy.

CONFIDENTIAL

Reconstructing the full image requires combining the two matrices, which are stored securely in the encrypted `SecretImage` class. This ensures that sensitive data is protected and can only be reconstructed when necessary. For simplicity, you may assume that all provided images will be square.

3.2 Task 1: Secret Image Storage

In this task, you will work with the `SecretImage` class, which splits a grayscale image into upper and lower triangular matrices for secure storage. Your job is to:

- **Splitting the image:** When constructing a `SecretImage`, the constructor will take an instance of `GrayscaleImage` as input, and the given image will be split into upper and lower triangular matrices. Each part will be stored separately in dynamically allocated 1D array memory.
- **Reconstructing the image:** When needed, the original `GrayscaleImage` image must be reconstructed by combining the upper and lower triangular matrices. The `reconstruct()` function will reconstruct and return the full `GrayscaleImage` image by combining both triangular matrices.
- **Saving the filtered image:** Filtering operations on a `SecretImage` object involve reconstructing the image and then applying the filters. Once filtering operations are performed on the reconstructed image, the new image must be saved back into the `SecretImage` format with triangular arrays. The `save_back()` function will save the changes done to the `GrayscaleImage` image back to the triangular matrices, ensuring the data is kept secure.
- **Saving the secret image to a file:** Saving operations on a `SecretImage` object require writing the dimensions of the picture and the upper and lower triangular matrices to a text file. The `save_to_file()` function handles this by first writing the image dimensions to the first line of the file, with the width and height separated by a space. This is followed by the contents of the upper triangular array on the second line, where the pixel values should be separated by spaces, and then by the lower triangular array on the third line in a similar manner. The size of the `upper_triangular` array is calculated as $\frac{\text{width} \times (\text{width}+1)}{2}$, and the `lower_triangular` array is calculated as $\frac{\text{width} \times (\text{width}-1)}{2}$.

3.3 Task 2: Embedding and Extracting Secret Messages

`ClearVision Defense` requires embedding and extracting secret messages embedded and hidden within the least significant bits (LSBs) of an image's pixels. These messages could contain important commands or intelligence.

Embedding Secret Messages: Secret messages will be converted into bits and stored in the least significant bits (LSBs) of the image pixels. The process involves the following steps:

1. Convert the ASCII characters of the message into a binary bitstream, where each character is represented by 7 bits.
2. Embed the bitstream into the least significant bits of the image pixels, starting from the pixel calculated such that the last bit of the secret message is embedded at the very last pixel of the image (i.e., the pixel at the bottom-right corner). The embedding process should proceed bit by bit, moving from left to right across each row, and continuing down to the next row.
3. Once the message is fully embedded, save the modified image back into the triangular matrices (upper and lower triangular matrices as described earlier).

CONFIDENTIAL

Example: Consider a 4×4 grayscale image matrix where each pixel value is stored as an 8-bit number:

$$I = \begin{bmatrix} 100 & 120 & 140 & 160 \\ 180 & 200 & 220 & 240 \\ 255 & 210 & 190 & 170 \\ 150 & 130 & 110 & 90 \end{bmatrix}$$

Let the secret message be "AB", represented in ASCII and binary (7 bits each) as shown below, with the resulting bitstream:

$$\text{A} = 65 = (1000001)_2, \quad \text{B} = 66 = (1000010)_2, \quad \text{Bitstream} = 10000011000010$$

We will embed this bitstream into the least significant bits of the image starting from pixel $I[0, 2] = 140$ and moving row by row, ending at the bottom-right pixel $I[3, 3]$. The embedding process replaces the LSB of each pixel value with the next bit from the bitstream.

After embedding the secret message, the image matrix becomes:

$$I_{\text{LSB-embedded}} = \begin{bmatrix} 100 & 120 & 141 & 160 \\ 180 & 200 & 220 & 240 \\ 255 & 211 & 190 & 170 \\ 150 & 130 & 111 & 90 \end{bmatrix}$$

Explanation of changes (we give the 8-bit binary representation for the sake of simplicity):

- Pixel $I[0, 2]$: Original value was 140, binary is 10001100, LSB replaced by '1' (first bit from the bitstream), new value is $10001101 = 141$. The same operation happened with pixels $I[2, 1]$ and $I[3, 2]$. Pixel $I[2, 0]$ LSB was also assigned '1', however, it did not cause any change as its original value was $11111111 = 255$, and thus, its LSB was already '1'.
- Pixel $I[0, 3]$: Original value was 160, binary is 10100000, LSB replaced by '0' (second bit), remains $10100000 = 160$. The same operation was performed on other pixels that got '0' as their LSB.

Thus, the message "AB" has been successfully embedded into the image, starting from pixel $I[0, 2]$ and ending at pixel $I[3, 3]$.

Extracting Secret Messages: To retrieve hidden messages from the image:

1. Calculate the starting pixel based on the length of the secret message. Since each character is represented by 7 bits, multiply the message length by 7 to determine the total number of bits to be extracted.
2. Extract the least significant bits of the image pixels and reconstruct the bitstream.
3. Convert the extracted bits back into ASCII characters to retrieve the secret message.

Implementation Instructions: The `Crypto` class provides functions to handle message encryption and decryption:

- The `extract_LSBits()` function extracts the LSBs from the image's pixel values. It calculates the starting pixel based on the provided message length, extracts the required number of bits, and reconstructs the bitstream.
- The `decrypt_message()` function converts the extracted bitstream into readable ASCII characters and returns the decoded message.
- The `encrypt_message()` function converts a given string into its binary representation (LSBs), and returns a bit array that can be embedded into the image.
- The `embed_LSBits()` function embeds the provided bit array into the image by replacing the least significant bits of the pixel values with the message bits.

CONFIDENTIAL

4 Assignment Implementation Tasks and Requirements

In this section, we outline the classes and functions you are required to implement. You must follow the provided starter code, ensuring clarity, proper encapsulation, and adherence to the predefined class structures. Do not change the names or signatures of functions provided in the templates. You are free to introduce additional functions or variables as needed to improve functionality or readability.

4.1 `GrayscaleImage` Class

This class represents a grayscale image object, storing pixel values in a 2D matrix.

- **Constructors**:

  `GrayscaleImage(const char* filename)`

  - This constructor reads the image file (e.g., PNG) and loads the pixel data into the image matrix.

  `GrayscaleImage(int** inputData, int h, int w)`

  - This constructor initializes an object from a 2D data matrix.

  `GrayscaleImage(int w, int h)`

  - This constructor creates a blank image of given width and height.
- **Copy Constructor**:

  `GrayscaleImage(const GrayscaleImage& other)`

  - This constructor creates a copy of the given object.
- **Operator Overloading**:
  - '==' operator compares two images for equality based on their size and pixel data.
  - '+' operator adds pixel values of two images together.
  - '−' operator subtracts the pixel values of the second image from the first.
- **Function**:

  `void print_image() const`

  - Prints the image matrix to the console.
- **Function**:

  `void save_to_file(const char* filename) const`

  - Save the image back to an image file after applying filters or modifications.
- **Destructor**:

  `~GrayscaleImage()`

  - Clean up dynamically allocated memory for the image data.

4.2 `SecretImage` Class

This class is responsible for securely storing and processing grayscale images by splitting them into upper and lower triangular matrices.

- **Constructor**:

  `SecretImage(const GrayscaleImage& image)`

  - Initializes the image by splitting it into upper and lower triangular matrices. Stores the diagonal elements only in the upper triangular matrix.

- **Function**:
  `GrayscaleImage reconstruct() const`
  - Reconstructs the full grayscale image from the two triangular matrices.
- **Function**:
  `void save_back(const GrayscaleImage& image)`
  - Updates the upper and lower triangular matrices with the modified pixel data after filtering or embedding a message.
- **Function**:
  `void save_to_file(const string& filename)`
  - Saves a secret image into the given text file. The file should store width and height values in the first line, upper triangular matrix on the second and lower triangular matrix in the third line.
- **Function**:
  `SecretImage load_from_file(const string& filename)`
  - Reads the given text file and instantiates and returns a SecretImage.
- **Destructor**:
  `~SecretImage()`
  - Clean up dynamically allocated memory for the triangular matrices.

4.3 `Filter` Class

This class provides functions for applying different filters to grayscale images.

- **Function**:
  `void apply_mean_filter(GrayscaleImage& image, int kernelSize)`
  - Applies a mean filter (box filter) to the grayscale image using the specified kernel size.
- **Function**:
  `void apply_gaussian_smoothing(GrayscaleImage& image, int kernelSize, double sigma)`
  - Applies Gaussian smoothing to the grayscale image using the specified kernel size and sigma.
- **Function**:
  `void apply_unsharp_mask(GrayscaleImage& image, int kernelSize, double amount)`
  - Applies the unsharp mask filter to the image, emphasizing edges for a sharpening effect.

4.4 `Crypto` Class

This class handles the embedding and extraction of secret messages within the image pixels' least significant bits (LSBs).

- **Function**:
  `std::vector<int> extract_LSBits(SecretImage& secret_image, int message_length)`
  - Extracts the least significant bits (LSBs) from the image starting from the calculated pixel position and retrieves the message.

- **Function**:
  `std::string decrypt_message(const std::vector<int>& LSB_array)`
  - Converts the extracted LSB bitstream into readable ASCII characters to retrieve the hidden message.
- **Function**:
  `std::vector<int> encrypt_message(const std::string& message)`
  - Converts an ASCII message into its binary representation and prepares it for embedding.
- **Function**:
  `SecretImage embed_LSBits(GrayscaleImage& image, const std::vector<int>& LSB_array)`
  - Embeds the encrypted message bitstream into the least significant bits of the image and returns a `SecretImage`.

Must-Use Starter Codes

You MUST use **this starter (template) code**. All headers and classes should be placed directly inside your **zip** archive.

Grading Policy

- No memory leaks and errors: 10%
  - No memory leaks: 5%
  - No memory errors: 5%
- Implementation of the game: 80%
  - Correct image operations (operator overloading, constructors, destructors): 20%
  - Correct application of filters (matrix operations): 30%
  - Correctly revealing and disguising images (array operations): 10%
  - Correctly encrypting and decrypting images: 20%
- Output tests: 10%

The tests on the images will be performed by matching your solutions to the expected solutions pixel by pixel.

Important Notes

- Do not miss the deadline: **Friday, 25.10.2024 (23:59:59)**.
- Save all your work until the assignment is graded.
- The assignment solution you submit must be your original, individual work. Duplicate or similar assignments are both going to be considered as cheating.
- You can ask your questions via Piazza (https://piazza.com/hacettepe.edu.tr/fall2024/bbm203), and you are supposed to be aware of everything discussed on Piazza.
- You must test your code via **Tur^{6}Bo Grader** https://test-grader.cs.hacettepe.edu.tr/ (**does not count as submission!**).

- You must submit your work via https://submit.cs.hacettepe.edu.tr/ with the file hierarchy given below:
  - **b<studentID>.zip**
    - Crypto.cpp <FILE>
    - Crypto.h <FILE>
    - Filter.cpp <FILE>
    - Filter.h <FILE>
    - GrayscaleImage.cpp <FILE>
    - GrayscaleImage.h <FILE>
    - SecretImage.cpp <FILE>
    - SecretImage.h <FILE>
- **You MUST use this starter code**. All classes should be placed directly in your **zip** archive.
- This file hierarchy must be zipped before submitted (not .rar, only .zip files are supported).

Run Configuration

Here is an example of how your code will be compiled (note that instead of `main.cpp` we will use our test files):

```
$ g++ -g -std=c++11 -o clearvision main.cpp SecretImage.cpp GrayscaleImage.cpp
↪  Filter.cpp Crypto.cpp
```

Or, you can use the provided `Makefile` or `CMakeLists.txt` within the sample input to compile your code:

```
$ make
```

or

```
$ mkdir clearvision_build
$ cmake -S . -B clearvision_build/
$ make -C clearvision_build/
```

After compilation, you can run the program as follows:

```
$ ./clearvision mean puppy.png 3
```

Academic Integrity Policy

All work on assignments **must be done individually**. You are encouraged to discuss the given assignments with your classmates, but these discussions should be carried out in an abstract way. That is, discussions related to a particular solution to a specific problem (either in actual code or in pseudocode) **will not be tolerated**. In short, turning in someone else's work (including work available on the internet), in whole or in part, as your own will be considered as **a violation of academic integrity**. Please note that the former condition also holds for the material found on the web as everything on the web has been written by someone else.

The submissions will be subjected to a similarity check. Any submissions that fail the similarity check will not be graded and will be reported to the ethics committee as a case of academic integrity violation, which may result in the suspension of the involved students.