IMAGE PROCESSING IN C

CSB 102 – DATA STRUCTURES

S. BHAVYESH

191210045

INTRODUCTION

This project deals with the basic functions of image processing, written in C language and using no external libraries. The images used in this project are 24-bit depth coloured BMP/DIB images with no padding and height and width multiples of 4. There will be instances where DIB and BMP will be interchangeably used, but both formats are very similar (after all, DIB stands for Device Independent Bitmap).

In this project, ***all of the basic functions, which are 10 in number have been implemented***. They are as follows:

1. **Read image**
2. **Write image**
3. **Getting image info**
4. **Getting pixel data**
5. **Changing pixel data**
6. **Mean grey value**
7. **RGB to Gray conversion**
8. **Gray/RGB to binary**
9. **Reflecting an image**
10. **Cropping an image**

Apart from the basic functions***, all of the extra credit functions, which are 8 in number have been implemented***. They are as follows:

1. **Enhancing image quality**
2. **Shrinking an image**
3. **Enlarging an image**
4. **Translating an image**
5. **Rotating image by any arbitrary angle**
6. **Adding two images**
7. **Subtracting two images and comparing**
8. **Negative of an image**

THE BMP FORMAT

The bitmap (BMP) format is a way of storing images, which consists of two, or sometimes three components, which are header, colour table(if bit-depth <=8) and pixel data.

Header

The format for header is:

offset	size	description
0	2	signature, must be 4D42 hex
2	4	size of BMP file in bytes (unreliable)
6	2	reserved, must be zero
8	2	reserved, must be zero
10	4	offset to start of image data in bytes
14	4	size of BITMAPINFOHEADER structure, must be 40
18	4	image width in pixels
22	4	image height in pixels
26	2	number of planes in the image, must be 1
28	2	number of bits per pixel (1, 4, ,8 or 24) (bitDepth)
30	4	compression type (0=none, 1=RLE-8, 2=RLE-4)
34	4	size of image data in bytes (including padding)
38	4	horizontal resolution in pixels per meter (unreliable)
42	4	vertical resolution in pixels per meter (unreliable)
46	4	number of colors in image, or zero
50	4	number of important colors, or zero

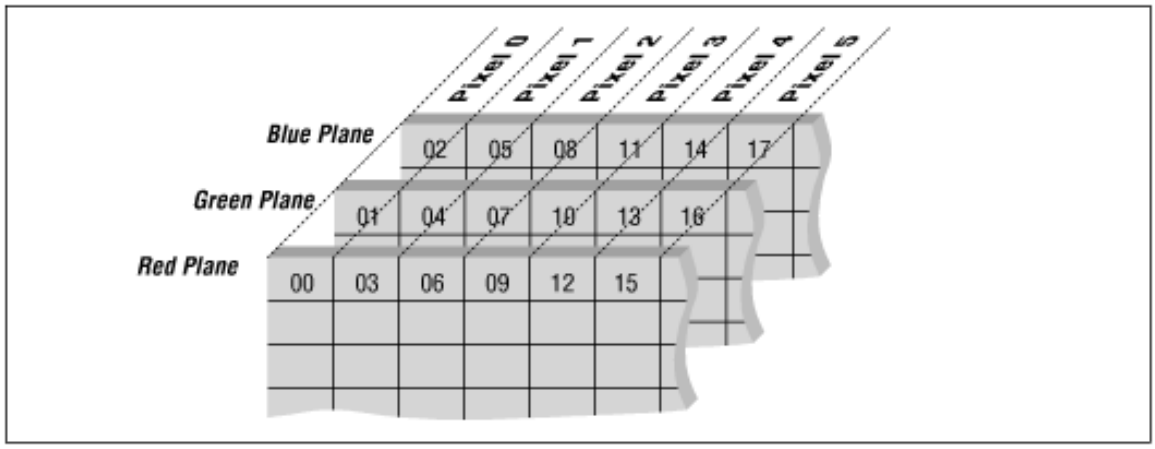
As we can see, the image header uses 54 bytes to store information. Out of this information, we will mainly require the height data, width data, and bit-depth. Bit depth refers to the number of bits used to represent every pixel. In this project, we will be dealing with 24-bit depth BMPs, so we won't dabble with this much.

Colour Table

The color table is a block of bytes (a table) listing the colors used by the image. Each pixel in an indexed color image is described by a number of bits (1, 4, or 8) which is an index of a single color described by this table. Color table is useful for images with bit depth less than 16. Since bit depth of images used in this project is 24, there is no color table for such images.

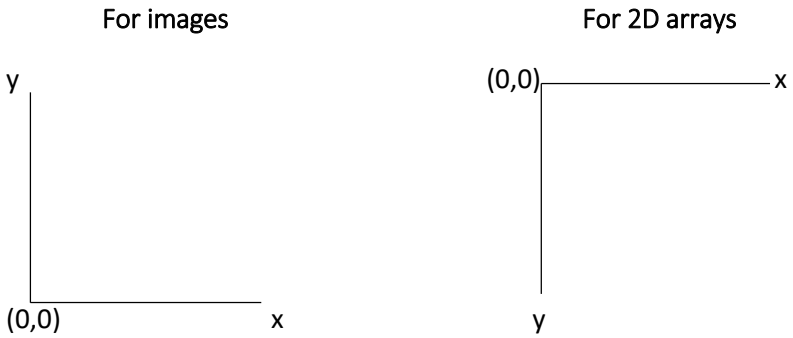
Pixel Array

The pixel contains pixel data of the image. The data is stored in repeating sequences of BGR, BGR, BGR…(Run Length Encoding or RLE) with the values of each component varying. The diagram below shows a representation of pixel data.



A visualization of pixel data

Usually pixels are stored "bottom-up", starting in the lower left corner, going from left to right, and then row by row from the bottom to the top of the image. Due to this bottom-up approach, the origin (0,0) is at the bottom left of the image, as opposed to top left of image, which is the usual case with 2D arrays.



Site used for cropping JPG images to suitable size for execution: https://resizeimage.net/

Site used for converting JPG to BMP: https://online-converting.com/image/convert2bmp/

STRUCTURES

The first structure that is used is ***pixel.*** The definition for the structure is given below:

```
typedef struct pixel    // pixel structure
{
      // ordering is done this way as pixel data is
      //stored in RLE of BGR, BGR, BGR....

      unsigned char blue;
      unsigned char green;
      unsigned char red;

}pixel;
```

In order to simplify operations, it is good to have the RGB values of a pixel in one place. Also, it aids in visualizing the pixel map of the image.

The second structure that is used (which is the main structure of the program) is ***image.*** The definition is given below:

```
typedef struct image    // image data type
{
      char path[50]; // image path address in the system
      unsigned char header[54]; // image header
      pixel** buffer; // pixel data
      int height;
      int width;
      int bitdepth;

}image;
```

The members height, width, bit-depth are introduced for quick access of these variables, instead of looking into the header.

PART 1
BASIC FUNCTIONS

The readImage(FILE f, image* img) function*

As it known, an image is just another file for C. So, we begin with the most important function of the package, which allows us to read images. The code snippet for the function is given below:

```
void readImage(FILE* f,image* img)
{
	f=fopen(img->path,"rb");

	if(!f) printf("FILE DNE");
//Checking the existence of image file
	else
	{
		fread(img->header,sizeof(unsigned char),54,f);
//Reading the header
		img->width=*(int*)&(img->header[18]);
		img->height=*(int*)&(img->header[22]);
		img->bitdepth=*(int*)&(img->header[28]);

		img->buffer=malloc(img->height*sizeof(pixel*));

		for(int i=0;i<img->height;i++)
			img->buffer[i]=malloc(img->width*sizeof(pixel));

		for(int i=0;i<img->height;i++)
			fread(img->buffer[i],sizeof(pixel),img->width,f);
	//Reading pixel data into buffer
	}
	fclose(f);
}
```

The parameters of the function are the file pointer (f) and the image pointer (img). First, the file is opened in binary mode, and then we check if the file exists. If it exists, first we read the header data of the image into the header part of the image. Then, the values of height, width and bit-depth are assigned for quick access. Then, memory is allocated to the buffer of image structure and row-by-row, pixel data is put inside the buffer.

The writeImage(FILE f, image* img) function*

Now that the image is read, after performing some function, it must be read back to see the result of the manipulation. The writeImage function helps us to do so. The code for the function is given here:

```
void writeImage(FILE* f,image* img)
{
	f=fopen(img->path,"wb");

	fwrite(img->header,sizeof(unsigned char),54,f);

	for(int i=0;i<img->height;i++)
	fwrite(img->buffer[i],sizeof(pixel),img->width,f);

	fclose(f);
}
```

The code is self-explanatory. A file is opened and the header data of the image along with the pixel data is read into it.

The output is a copy of the image:

Copy.bmp

The getInfo(image img) function*

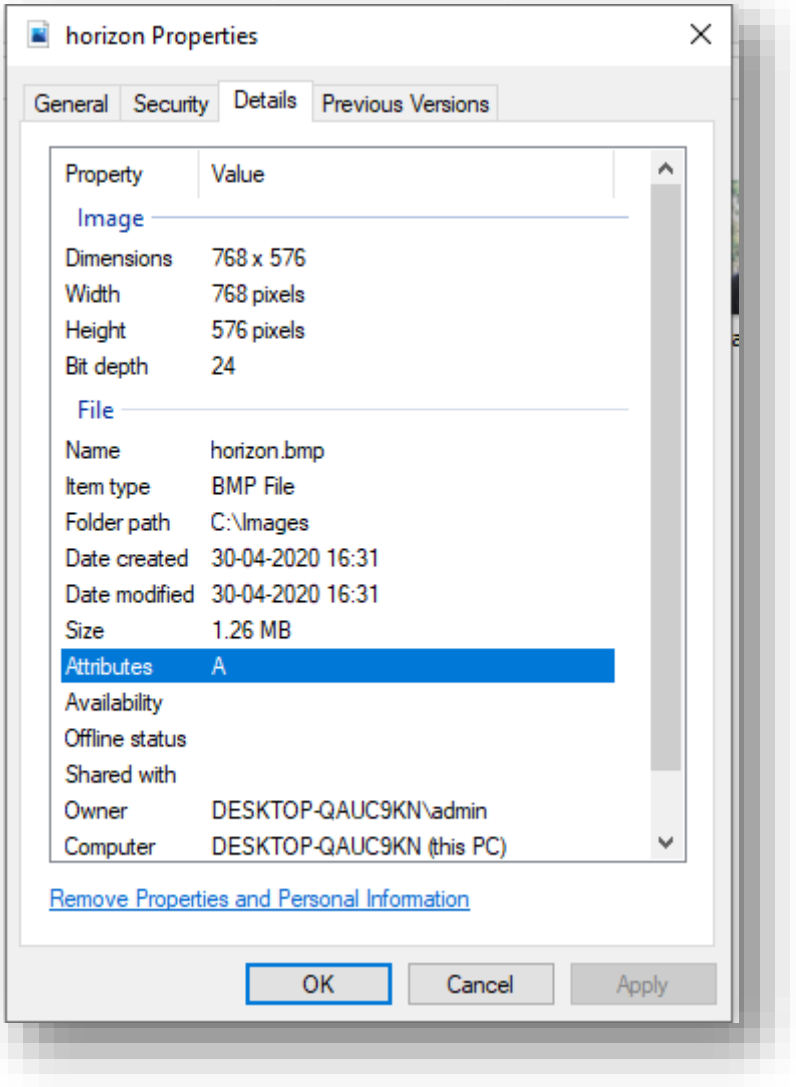
This function is used for displaying data related to the image, mainly the image path address, height of the image, width of the image and its bit-depth. The code is given below:

```
void getInfo(image* img)
{
	printf("\n image path: %s",img->path);
	printf("\n image height: %d",img->height);
	printf("\n image width: %d",img->width);
	printf("\n image bit depth: %d",img->bitdepth);
}
```

The output in Eclipse IDE console is:

```
image path: C:/Images/horizon.bmp
image height: 576
image width: 768
image bit depth: 24
```

And the properties of image verify this:



Properties of the concerned image

The getPixelval(image img, int r, int c) function*

This function allows us to see the RGB values of a pixel where ‘r’ represent the elevation from the origin and ‘c’ represents the distance from the origin. Here is the code:

```
void getPixelval(image* img,int r,int c)
{
	printf("\n\n red value:%d\n",img->buffer[r][c].red);
	printf(" green value:%d\n",img->buffer[r][c].green);
	printf(" blue value:%d\n",img->buffer[r][c].blue);
}
```

The output in Eclipse IDE when coordinates are (100,100) is:

```
red value:254
green value:254
blue value:253
```

Which makes sense since 100,100 lies in the region given below:

Getpixelval.bmp

The setPixelval(image img, int r, int c) function*

This function allows us to set the RGB values of a pixel where ‘r’ represent the elevation from the origin and ‘c’ represents the distance from the origin. Here is the code:

```
void setPixelval(image* img,int r,int c)
{
	image img_temp=*img;

	int temp;

	printf("\n\n set red value:");
	scanf("%d",&temp);
	img_temp.buffer[r][c].red=temp;

	printf("\n\n set green value:");
	scanf("%d",&temp);
	img_temp.buffer[r][c].green=temp;

	printf("\n\n set red value:");
	scanf("%d",&temp);
	img_temp.buffer[r][c].blue=temp;

	FILE* f=NULL;
	strcpy(img_temp.path,"C:/Images/pixel_change.bmp");
	writeImage(f,&img_temp);
}
```

The output when coordinates are (30,200) and red=0,green=0,blue=0 is give on the next page.

Original

Pixel change at (30,200) (black dot since red=green=blue=0)

The meanGray(image img) function*

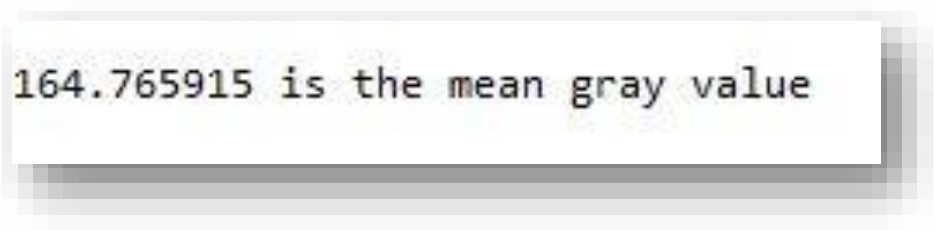
This function displays the mean gray value of the image. Mean gray value refers to the brightness level of a pixel. So, the mean gray value of the image is the average brightness or the luminance of the image. Here, we use the following formula for gray value of a pixel:

Gray value= 0.11 * blue + 0.59 * green + 0.3 * red

```
void meanGray(image* img)
{
	float sum=0;
	int i=0;

	for(i=0;i<img->height;i++)
	for(int j=0;j<img->width;j++)
					sum+=0.11*img->buffer[i][j].blue +
0.59*img->buffer[i][j].green + 0.29*img->buffer[i][j].red;
//Using the formula for luminance/gray value

	sum/=(img->height*img->width);

	printf("\n%f is the mean gray value\n",sum);
}
```

A double for loop is run to traverse the pixel map of the image. Then, the gray value is displayed.

The output on console window of Eclipse IDE is:



The rgb_to_gray(image img) function*

This function is used to convert an RGB coloured image into a gray image. The code for the function is given below:

```
void rgb_to_gray(image* img)
{
	image *img_temp=malloc(sizeof(image));
	*img_temp=*img;

	for(int i=0;i<img_temp->height;i++)
	{
		for(int j=0;j<img_temp->width;j++)
		{

			int y=0.11*img->buffer[i][j].blue
+ 0.59*img->buffer[i][j].green + 0.29*img->buffer[i][j].red;
//Using the formula for luminance

img_temp->buffer[i][j].blue=img_temp->buffer[i][j].green
=img_temp->buffer[i][j].red=y;

		}
	}

	FILE* f=NULL;
	strcpy(img_temp->path,"C:/Images/rgb_to_gray.bmp");
	writeImage(f,img_temp);
	free(img_temp);
}
```

We use the same formula as in the previous function (gray value). The gray value of each pixel is found and the RGB components of the pixel are updated with the gray value.

The output is given on the next page.

Original

Rgb_to_gray.bmp

The gray_to_binary(image img) function*

This function is used to convert a gray image to a binary image with each pixel either black or white, although coloured images can be converted to binary too. We again use the gray value formula to find the luminance of each pixel. We assign a threshold value of luminance of 128, and pixel with gray value greater than or equal to 128, that pixel's RGB components are assigned the colour white(255). The pixels with luminance less than 128 are converted to black (0). The code is given below:

```
void gray_to_binary(image* img)
{
	image *img_temp=malloc(sizeof(image));
	*img_temp=*img;

	for(int i=0;i<img_temp->height;i++)
	{
		for(int j=0;j<img_temp->width;j++)
		{
			int y=0.11*img_temp->buffer[i][j].blue+
0.59*img_temp->buffer[i][j].green+0.29*img_temp->buffer[i][j].red;
//Using the formula for luminance

			if(y>=128) img_temp->buffer[i][j].blue=
img_temp->buffer[i][j].green=img_temp->buffer[i][j].red=255;


			else img_temp->buffer[i][j].blue=
img_temp->buffer[i][j].green=img_temp->buffer[i][j].red=0;
//Assigning value to each component of pixel

		}
	}

	FILE* f=NULL;
	strcpy(img_temp->path,"C:/Images/gray_to_binary.bmp");
	writeImage(f,img_temp);
}
```

The output is on the next page.

Original

Binary (with threshold 128)

Gray image

Binary of gray image

The reflectImage(image img, int flag)*

This function allows us to obtain the reflection of the image about the horizontal or vertical axis, depending on the flag value: if the flag is 1, the reflection is done about the horizontal axis. If the flag is zero, reflection is done along the vertical axis. The code is given below:

```
void reflectImage(image* img,int flag)
{
	image* img_temp=malloc(sizeof(image));
	*img_temp=*img;

	if(flag) // Vertical reflection
	{
		for(int i=0;i<img_temp->height/2;i++)
//Swapping pixels about the horizontal axis
		{
			for(int j=0;j<img_temp->width;j++)
			{
				pixel temp=img_temp->buffer[i][j];

	img_temp->buffer[i][j]=img_temp->buffer[img->height-i-1][j];

				img_temp->buffer[img->height-i-1][j]=temp;
			}
		}
	}

	else //Horizontal reflection
	{
		for(int i=0;i<img->height;i++)
		{
			for(int j=0;j<img->width/2;j++)
//Swapping pixels about the vertical axis
			{
				pixel temp=img_temp->buffer[i][j];
		img_temp->buffer[i][j]=img_temp->buffer[i][img->width-j-1];
			img_temp->buffer[i][img->width-j-1]=temp;
					}
			}
	}

		FILE* f=NULL;
		strcpy(img_temp->path,"C:/Images/reflect.bmp");
		writeImage(f,img_temp);
		free(img_temp);
}
```

The outputs are here:

Original

Vertical reflection

Horizontal reflection

The cropImage(image img, int down, int up ,int left, int right) function*

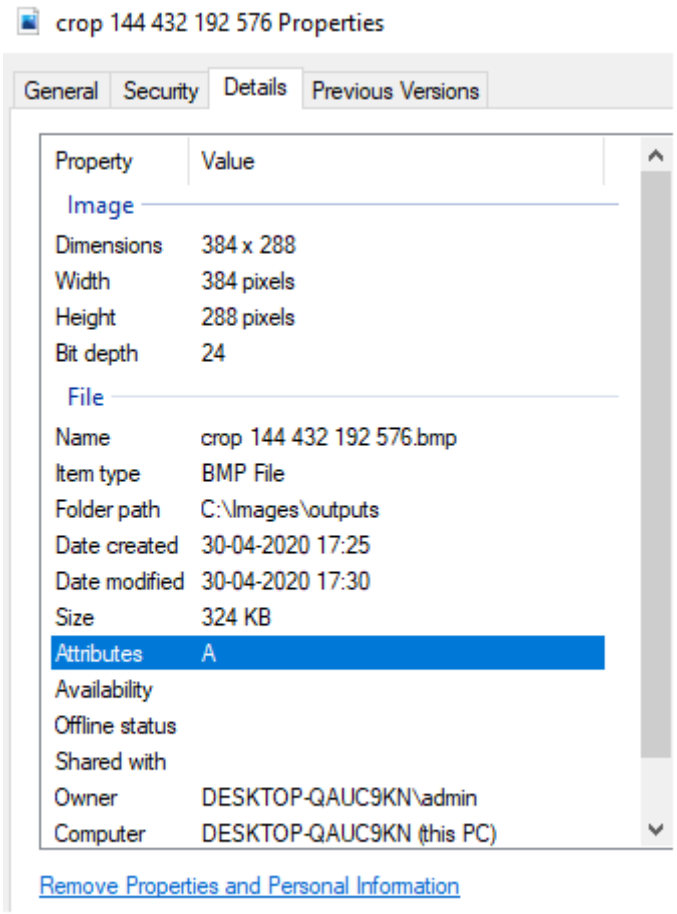
This function is used to crop a sub image from a given image. The main idea here is to reallocate the buffer of the image with the new dimensions: new height and new width. The size is automatically changed. Then, the pixels that are within the desired region are read into the reallocated buffer. And finally, the new image is read into a file. Obviously, it is assumed that the dimensions are 'sensible': as in the **down** parameter should be less than **up** and similarly for **left** and **right**, and these dimensions must be less than the **height** and **width** respectively. The code is given below:

```
void cropImage(image* img,int down,int up,int left,int right)     // obtain
a sub-image
{

            image* img_temp=malloc(sizeof(image));
            *img_temp=*img;
            int width=right-left;
            int height=up-down;

            //allocation of buffer

            pixel**     temp=malloc(height*sizeof(pixel*));
            for(int i=0;i<height;i++) temp[i]=malloc(width*sizeof(pixel));

            for(int i=down;i<up;i++)
            for(int j=left;j<right;j++)
                  temp[i-down][j-left]=img->buffer[i][j];
            //Pixels within the desired space are read into buffer

            *(int*)&img_temp->header[22]=img_temp->height=height;
            *(int*)&img_temp->header[18]=img_temp->width=width;
            //Header data is modified accordingly

            printf("width %d height %d",
*(int*)&img_temp->header[18],*(int*)&img_temp->header[22]);
            printf("width %d height %d",
*(int*)&img->header[18],*(int*)&img->header[22]);

            //Buffer data is written into file
            FILE* fo=fopen("C:/Images/crop.bmp","wb+");
            fwrite(img_temp->header,sizeof(unsigned char),54,fo);
                        for(int i=0;i<height;i++)
            fwrite(temp[i],sizeof(pixel),width,fo);

            for(int i=0;i<height;i++) free(temp[i]);
            free(temp);
```

```
    fclose(fo);
    free(img_temp);
}
```

Original

Cropping area (144 to 432)(along height) , (192,576)(along width)



Properties of cropped part

PART 2
EXTRA FUNCTIONS

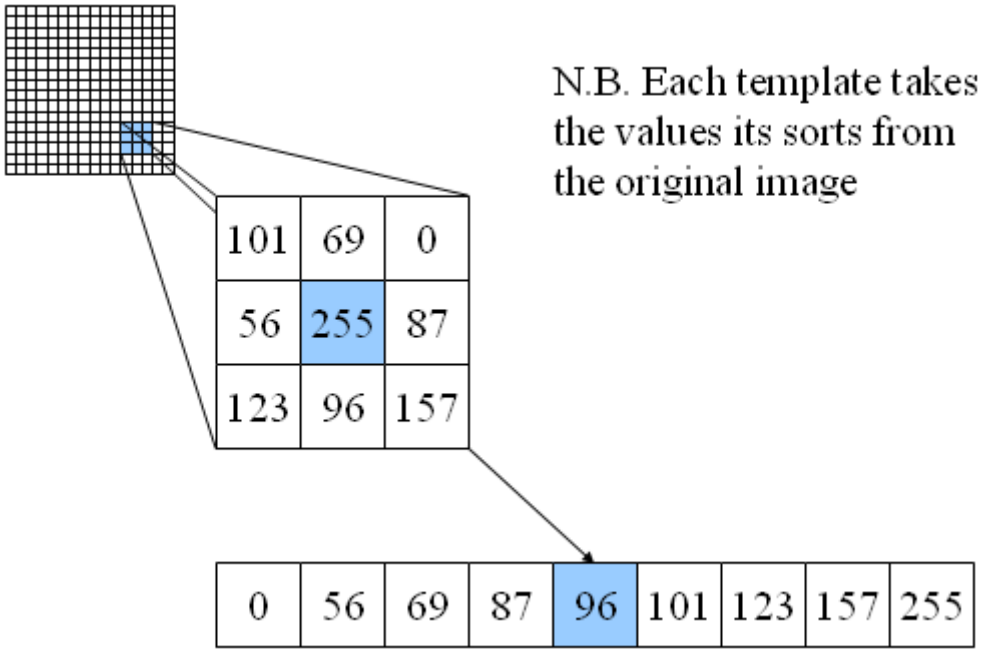
The Enhance(image img,int median,int edge,int cont,int bright) function*

This function is used to enhance the quality of the image. The function enhances the image by the following four mechanisms:

1. **Median filtering -** to smoothen the image and remove noise from it.
2. **Sobel filter/operator and image sharpening –** detecting edges and highlighting them.
3. **Contrast generation** – to increase the contrast of the image.
4. **Brightness increase** – to increase brightness of image.

Median Filtering

The basic idea of median filtering is that the given pixel after getting median filtered will hold a colour that is the *median* of its neighbours. The given diagram shows how:



As we can see, the resultant number is the median of all the numbers in the 3x3 block. Here, we will individually take the median of the three components (RGB).

The code for median filtering part is given below:

```
for(int i=1;i<img->height-1;i++) // ignoring edge cases
{
	for(int j=1;j<img->width-1;j++) // ignoring edge cases
	{

int red[9],blue[9],green[9]; // array of RGB components in the 3x3 matrix

	for(int k=-1;k<=1;k++)
	for(int l=-1;l<=1;l++) // looping around neighbors
	{

		red[3*(k+1)+l+1]=img_temp->buffer[i+k][j+l].red;

		green[3*(k+1)+l+1]=img_temp->buffer[i+k][j+l].green;

		blue[3*(k+1)+l+1]=img_temp->buffer[i+k][j+l].blue;
	}

	for(int i=0;i<9;i++)
	{
		for(int j=0;j<8;j++)
		{
			//sorting the arrays

		if(red[j]>red[j+1]) swap(red+j,red+j+1);

		if(green[j]>green[j+1]) swap(green+j,green+j+1);

		if(blue[j]>blue[j+1]) swap(blue+j,blue+j+1);
		}
	}

	// assigning median values to the concerned pixel
	img_temp->buffer[i][j].red=red[4];
	img_temp->buffer[i][j].green=green[4];
	img_temp->buffer[i][j].blue=blue[4];
	}
}
}
```

The code for swapping function is given below:

```
void swap(int *a,int *b)	// to swap elements
{
	int temp=*a;
	*a=*b;
	*b=temp;
}
```

The output due to median filtering is given here:

Original

Median only (note that image is now smooth)

Image with salt-and-pepper noise

Median filter removes salt-and-pepper noise

Sobel filter for edge detection

There are many edge detection algorithms such as Prewitt mask, Kirsch mask etc. but here we will be using Sobel mask for edge detection.

The Sobel mask detects two types of edges in an image:

1. Horizontal edges
2. Vertical edges

The kernels used for detection of edges are:

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

But here, we will be defining our kernel X as: [{-0.25,0,0.25},{-0.5,0,0.5},{-0.25,0,0.25}]

And our kernel Y as: [{0.25,0.5,0.25},{0,0,0},{-0.25,-0.5,-0.25}]

The first kernel is used for finding gradients in the X-direction, and the second kernel finds gradients in the Y-direction. Therefore, for every pixel, there will be two gradients: horizontal and vertical. To ensure that the values obtained will be between 0 and 255, we will call the roundoff(int n) function multiple times in the course of this operation.

The formula to find the magnitude of gradient is:

$$Gradient_{effective}=sqrt(Horizontal^2+Vertical^2).$$

To implement the filter, we apply the kernels over each pixel of the image, ignoring the edge cases. Also, to implement the filter, we need to first convert the coloured image to grayscale.

The code is given below:

```
// declaring a temporary buffer
pixel** temp_buffer=malloc(img->height*sizeof(pixel*));
      for(int i=0;i<img->height;i++)
temp_buffer[i]=malloc(img->width*sizeof(pixel));


// declaring a buffer of horizontal gradient
pixel** temp_x=malloc(img->height*sizeof(pixel*));
      for(int i=0;i<img->height;i++)
temp_x[i]=malloc(img->width*sizeof(pixel));

// declaring a buffer of vertical gradient
pixel** temp_y=malloc(img->height*sizeof(pixel*));
            for(int i=0;i<img->height;i++)
      temp_y[i]=malloc(img->width*sizeof(pixel));

//grayscale conversion
for(int i=0;i<img->height;i++)
{
      for(int j=0;j<img->width;j++)
      {
int y=0.11*img->buffer[i][j].blue + 0.59*img->buffer[i][j].green +
0.29*img->buffer[i][j].red;

temp_buffer[i][j].blue=temp_buffer[i][j].green=temp_buffer[i][j].red=y;
      }
}


      // kernels for horizontal and vertical gradients
      float kernel_x[3][3]={{-0.25,0,0.25},{-0.5,0,0.5},{-0.25,0,0.25}};
      float kernel_y[3][3]={{0.25,0.5,0.25},{0,0,0},{-0.25,-0.5,-0.25}};

// computing horizontal and vertical gradients of each pixel
      for(int i=1;i<img->height-1;i++)
      {
            for(int j=1;j<img->width-1;j++)
            {
                  float sum1=0,sum2=0;
                  for(int k=-1;k<=1;k++)
                  {
                        for(int l=-1;l<=1;l++)
                        {

            sum1+=temp_buffer[i+k][j+l].red*kernel_x[k+1][l+1];
//since image is 24 bit grayscale all components RGB will have same value
            sum2+=temp_buffer[i+k][j+l].red*kernel_y[k+1][l+1];
                        }
                  }

temp_x[i][j].red=temp_x[i][j].green=temp_x[i][j].blue=roundoff((int)sum1);

temp_y[i][j].red=temp_y[i][j].green=temp_y[i][j].blue=roundoff((int)sum2);
```

```
                }
            }


// effective gradient computation
for(int i=0;i<img->height;i++)
{
      for(int j=0;j<img->width;j++)
      {
int temp=(int)(sqrt(pow(temp_x[i][j].red,2)+pow(temp_y[i][j].red,2)));

temp_buffer[i][j].green=temp_buffer[i][j].blue=temp_buffer[i][j].red=temp;
      }
}

// saving a copy of Sobel edge detection
                  FILE* fo=fopen("C:/Images/sobel.bmp","wb+");
                  fwrite(img->header,sizeof(unsigned char),54,fo);
      for(int i=0;i<img->height;i++)
fwrite(temp_buffer[i],sizeof(pixel),img->width,fo);
            fclose(fo);


// adding the colored version and Sobel version
      for(int i=0;i<img->height;i++)
      {
            for(int j=0;j<img->width;j++)
            {
img_temp->buffer[i][j].red=roundoff(temp_buffer[i][j].red+
img->buffer[i][j].red);

img_temp->buffer[i][j].green=roundoff(temp_buffer[i][j].green+
img->buffer[i][j].green);

img_temp->buffer[i][j].blue=roundoff(temp_buffer[i][j].blue+
img->buffer[i][j].blue);
            }
      }

for(int i=0;i<img->height;i++) free(temp_buffer[i]);
      free(temp_buffer);

for(int i=0;i<img->height;i++) free(temp_x[i]);
      (temp_x);

for(int i=0;i<img->height;i++) free(temp_y[i]);
      free(temp_y);



}
```

The result due to Sobel filtering is given here:

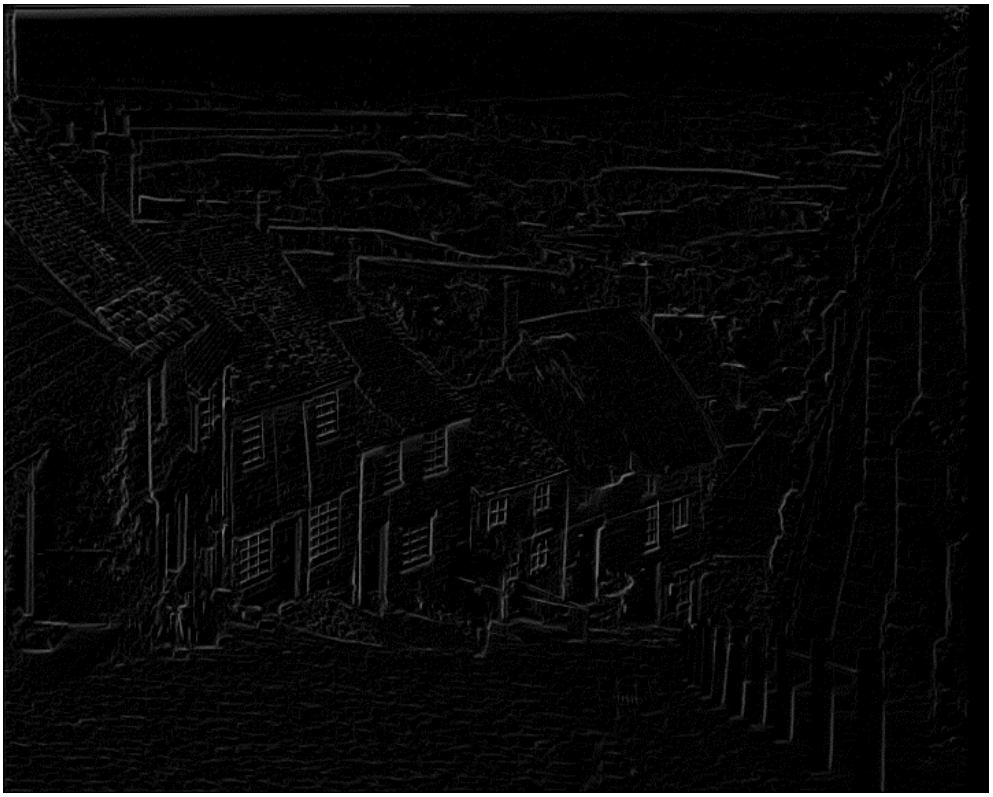
Original

Result of Sobel filtering highlights the edges

Sobel + original now highlights the boundaries(image sharpening)

(notice the face and the flowers on the right)

Original



Sobel filter

Sobel filter added to original

(windows and background are more highlighted)

Contrast Generation

Here, we are trying to increase the contrast by P(or here, 'con') . But, we need a different expression for the contrast factor in order to make the desired changes. The formula used to find the factor of contrast is:

Contrast Factor = (259 x (255+P)) / (255 x (259-P))

Source: https://www.dfstudios.co.uk/

And, we use the following formula to find the new value for each RGB component:

New Value of <colour> = Round off (factor * (<colour> - 128) + 128)

Source: https://www.dfstudios.co.uk/

In the above formula, replace <colour> with R or G or B.

Also, we the auxiliary function for rounding off the value obtained, to ensure that we stay within the bounds of 0 and 255 for pixel values:

```
int roundoff(int n)
// auxiliary function to stay within the limits of 0-255 for pixel
components
{
	if(n>=255) return 255;
	else if(n<=0) return 0;
	else return n;
}
```

The snippet dealing with contrast is given below:

```
float contra=(259*(255+con))/(255*(259-con));
// contrast adjustment of 'con' units.

for(int i=0;i<img->height;i++)
{
	for(int j=0;j<img->width;j++)
	{
		// new pixel value.
```

```
img->buffer[i][j].red=roundoff((int)(contra*(img->buffer[i][j].red-128)+128));

img->buffer[i][j].green=roundoff((int)(contra*(img->buffer[i][j].green-
128)+128));

img->buffer[i][j].blue=roundoff((int)(contra*(img->buffer[i][j].blue-
128)+128));
                                    }
                               }
```

Original

Contrast by 90 units

Brightness Increase

Here, we are trying to increase the brightness of image by a factor of 'bright', by simply adding this amount to the RGB values, because a higher value of pixel automatically tends to 255, the colour white. Again, we use the **roundoff (int n)** function to stay within the bounds. The code is given below:

```
for(int i=0;i<img->height;i++)
{
	for(int j=0;j<img->width;j++)
	{
	// simple addition by 'bright' to each component
	img->buffer[i][j].red=roundoff(img->buffer[i][j].red+bright);
	img->buffer[i][j].green=roundoff(img->buffer[i][j].green+bright);
	img->buffer[i][j].blue=roundoff(img->buffer[i][j].blue+bright);
	}
}
```

Original

Brightness increase by 60 units

The complete code for the The Enhance(image* img,int median,int edge,int cont,int bright) is given below:

```
void Enhance(image* img,int median,int edge,int con,int bright)
// enhance image quality
{
	image* img_temp=malloc(sizeof(image));
	*img_temp=*img;

	// Median Filtering begins

	if(median)
	{

		for(int i=1;i<img->height-1;i++) // ignoring edge cases
		{
			for(int j=1;j<img->width-1;j++) // ignoring edge cases
			{
int red[9],blue[9],green[9]; // array of RGB components in the 3x3 matrix

	for(int k=-1;k<=1;k++)
	for(int l=-1;l<=1;l++) // looping around neighbors
```

```
	{

	red[3*(k+1)+l+1]=img_temp->buffer[i+k][j+l].red;

	green[3*(k+1)+l+1]=img_temp->buffer[i+k][j+l].green;

	blue[3*(k+1)+l+1]=img_temp->buffer[i+k][j+l].blue;
	}

	for(int i=0;i<9;i++)
	{
		for(int j=0;j<8;j++)
		{
		//sorting the arrays


	if(red[j]>red[j+1]) swap(red+j,red+j+1);

	if(green[j]>green[j+1]) swap(green+j,green+j+1);

	if(blue[j]>blue[j+1]) swap(blue+j,blue+j+1);
		}
	}
// assigning median values to the concerned pixel

	img_temp->buffer[i][j].red=red[4];
	img_temp->buffer[i][j].green=green[4];
	img_temp->buffer[i][j].blue=blue[4];
		}
	}
}


// Median filtering ends


// Image sharpening with Sobel edge detection begins

if(edge)
{
	// declaring a temporary buffer
	pixel** temp_buffer=malloc(img->height*sizeof(pixel*));
for(int i=0;i<img->height;i++)
temp_buffer[i]=malloc(img->width*sizeof(pixel));


// declaring a buffer of horizontal gradient
pixel** temp_x=malloc(img->height*sizeof(pixel*));
for(int i=0;i<img->height;i++)
temp_x[i]=malloc(img->width*sizeof(pixel));

// declaring a buffer of vertical gradient
pixel** temp_y=malloc(img->height*sizeof(pixel*));
	for(int i=0;i<img->height;i++)
```

```
temp_y[i]=malloc(img->width*sizeof(pixel));

//grayscale conversion
for(int i=0;i<img->height;i++)
{
	for(int j=0;j<img->width;j++)
	{
int y=0.11*img->buffer[i][j].blue + 0.59*img->buffer[i][j].green +
0.29*img->buffer[i][j].red;

temp_buffer[i][j].blue=temp_buffer[i][j].green=temp_buffer[i][j].red=y;
	}
}


// kernels for horizontal and vertical gradients
	float kernel_x[3][3]={{-0.5,0,0.5},{-1,0,1},{-0.5,0,0.5}};
	float kernel_y[3][3]={{0.5,1,0.5},{0,0,0},{-0.5,-1,-0.5}};

// computing horizontal and vertical gradients of each pixel
	for(int i=1;i<img->height-1;i++)
	{
		for(int j=1;j<img->width-1;j++)
		{
			float sum1=0,sum2=0;
			for(int k=-1;k<=1;k++)
			{
				for(int l=-1;l<=1;l++)
				{

	sum1+=temp_buffer[i+k][j+l].red*kernel_x[k+1][l+1];
//since image is 24 bit grayscale all components RGB will have same value

	sum2+=temp_buffer[i+k][j+l].red*kernel_y[k+1][l+1];
				}
			}
temp_x[i][j].red=temp_x[i][j].green=temp_x[i][j].blue=roundoff((int)sum1);
temp_y[i][j].red=temp_y[i][j].green=temp_y[i][j].blue=roundoff((int)sum2);
		}
	}


// effective gradient computation
for(int i=0;i<img->height;i++)
{
	for(int j=0;j<img->width;j++)
	{

int temp=(int)(sqrt(pow(temp_x[i][j].red,2)+pow(temp_y[i][j].red,2)));
temp_buffer[i][j].green=temp_buffer[i][j].blue=temp_buffer[i][j].red=temp;
	}
}

// saving a copy of Sobel edge detection
```

```
FILE* fo=fopen("C:/Images/Sobel.bmp","wb+");
            fwrite(img->header,sizeof(unsigned char),54,fo);
                                    for(int i=0;i<img->height;i++)
      fwrite(temp_buffer[i],sizeof(pixel),img->width,fo);
      fclose(fo);

      // adding the colored version and Sobel version
      for(int i=0;i<img->height;i++)
      {
            for(int j=0;j<img->width;j++)
            {
img_temp->buffer[i][j].red=roundoff(temp_buffer[i][j].red+
img->buffer[i][j].red);

img_temp->buffer[i][j].green=roundoff(temp_buffer[i][j].green+
img->buffer[i][j].green);

img_temp->buffer[i][j].blue=roundoff(temp_buffer[i][j].blue+
img->buffer[i][j].blue);
            }
      }
}


// Image sharpening ends



// Increasing contrast begins

float contra=(259*(255+con))/(255*(259-con));
// contrast adjustment of 'con' units.


for(int i=0;i<img->height;i++)
{
      for(int j=0;j<img->width;j++)
      {
            // new pixel value.
img_temp->buffer[i][j].red=roundoff((int)(contra*
(img_temp->buffer[i][j].red-128)+128));

img_temp->buffer[i][j].green=roundoff((int)(contra*
(img_temp->buffer[i][j].green-128)+128));

img_temp->buffer[i][j].blue=roundoff((int)(contra*
(img_temp->buffer[i][j].blue-128)+128));
      }
}

// Increasing contrast ends
```

```
// Brightness increase begins

for(int i=0;i<img->height;i++)
{
      for(int j=0;j<img->width;j++)
      {
            // simple addition by ‘bright’ to each component


img_temp->buffer[i][j].red=roundoff(img_temp->buffer[i][j].red+bright);

img_temp->buffer[i][j].green=roundoff(
img_temp->buffer[i][j].green+bright);

img_temp->buffer[i][j].blue=roundoff(img_temp->buffer[i][j].blue+bright);
      }
}

// Brightness increase ends

// writing into a file

FILE* f=NULL;
strcpy(img_temp->path,"C:/Images/enhance.bmp");
writeImage(f,img_temp);
free(img_temp);

}
```

The Shrink(image img, float s) function*

The basic idea for shrinking is this: suppose we move 1 step in the shrunken image, then it is equivalent to moving **s** steps in the original, where s is the shrinking factor.

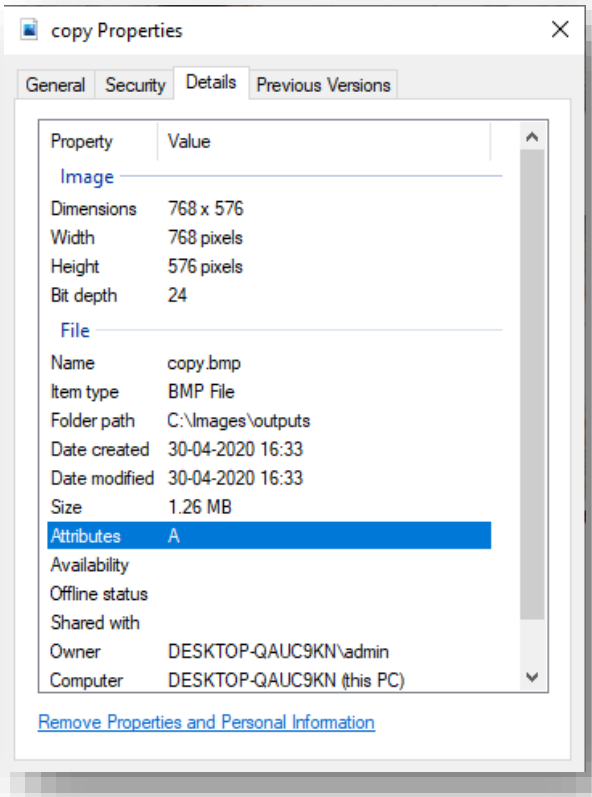
We create a temporary buffer of the required height and width which is obtained by dividing the height and width of original image by the shrinking factor. Since padding is not accounted for, if the shrunken image doesn't have height or width as a multiple of 4, the function does not work.

Then, the new height and width values are updated and finally, the new image is written into a file. The code is given below:

```
void Shrink(image* img,float s)
{
	image* img_temp=malloc(sizeof(image));
	*img_temp=*img;

	// modified values of height and width for new image
	int width=(int)(img_temp->width/s);
	int height=(int)(img_temp->height/s);

	// Declaring a 1D buffer
	pixel* temp=malloc(height*width*sizeof(pixel));

	for(int i=0;i<height;i++)
	for(int j=0;j<width;j++)
temp[i*width+j]=img->buffer[(int)(i*s)][(int)(j*s)];
	// implementing the basic idea

	*(int*)&img_temp->header[22]=height;
	*(int*)&img_temp->header[18]=width;
	// making changes in header of image

printf("width %d height %d",*(int*)&img_temp->header[18],
*(int*)&img_temp->header[22]);
	//printing height, width for confirmation


	// writing into a file
	FILE* fo=fopen("C:/Images/shrink.bmp","wb");
	fwrite(img_temp->header,sizeof(unsigned char),54,fo);
	fwrite(temp,sizeof(pixel),height*width,fo);


	fclose(fo);
	free(img_temp);
	free(temp);
}
```

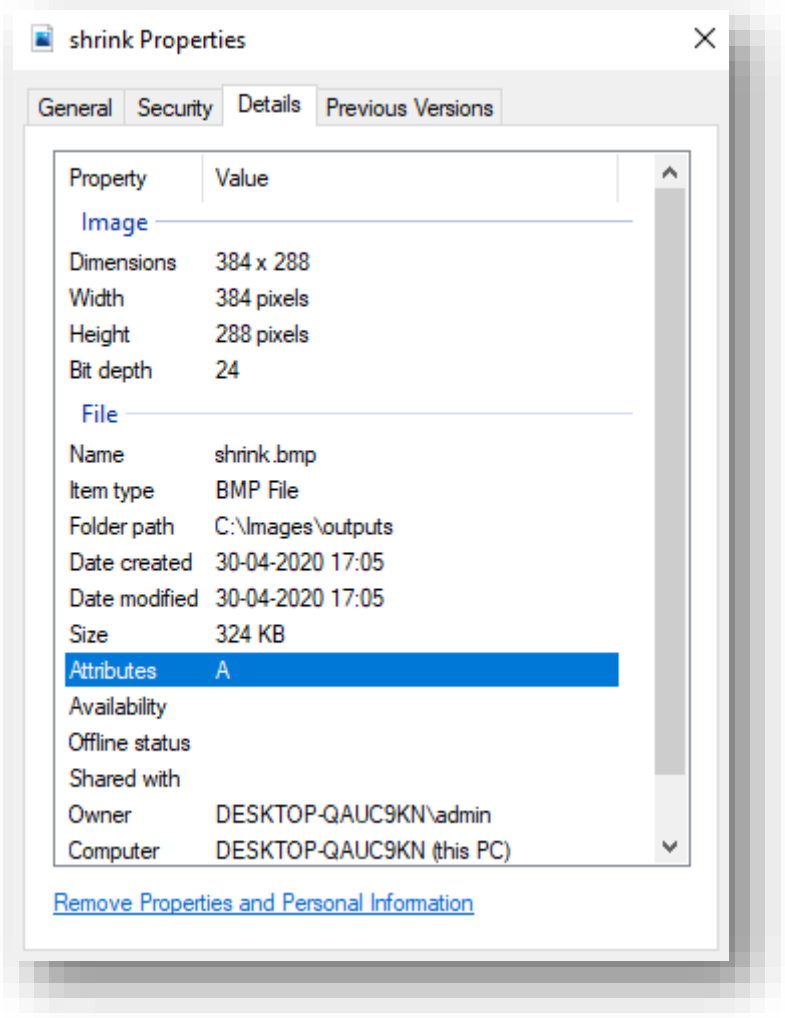
We could have used a 2D buffer but 1D buffer also can do the job.

Original



Properties of original

Shrunken image by factor of 2



Properties of shrunken version (compare the lengths: dimensions are halved)

The Enlarge(image img, float s) function*

Here, the idea is same as in the previous function of shrinking. Suppose we move **s** steps in the enlarged image, then it is equivalent to moving **1** step in the original, where s is the enalrging factor.

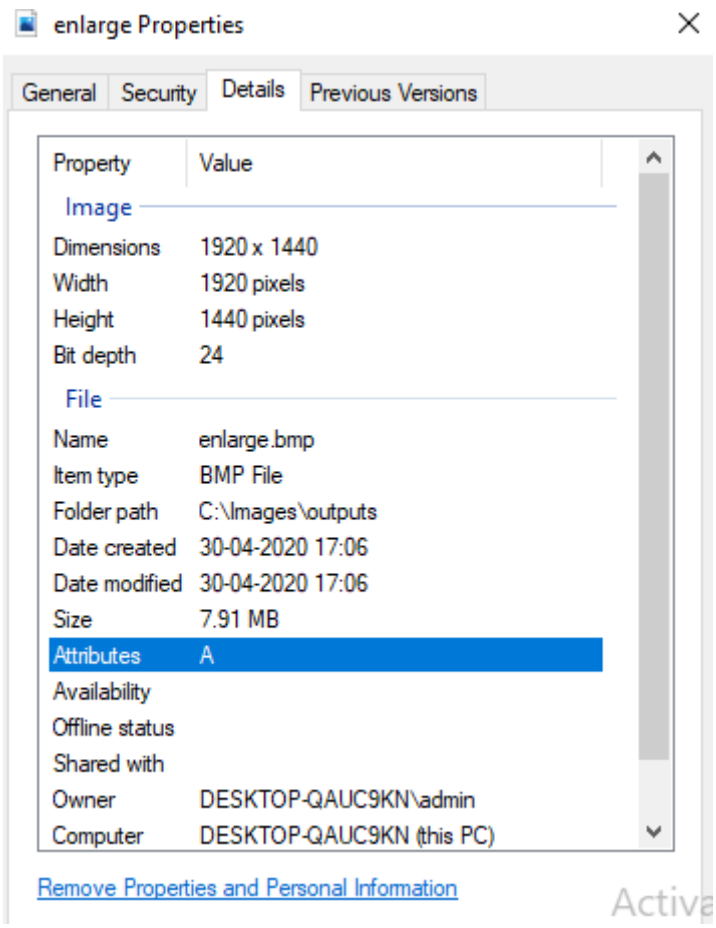
We create a temporary buffer of the required height and width which is obtained by multiplying the height and width of original image by the enlarging factor.

Then, the new height and width values are updated and finally, the new image is written into a file. The code is given below:

```
void Enlarge(image* img,float s)
{
	image* img_temp=malloc(sizeof(image));
	*img_temp=*img;

	// modified values of height and width for new image

	int width=(int)(img_temp->width*s);
	int height=(int)(img_temp->height*s);

	// Declaring a 1D buffer

	pixel* temp=malloc(height*width*sizeof(pixel));

	for(int i=0;i<height;i++)
	for(int j=0;j<width;j++)
temp[i*width+j]=img->buffer[(int)(i/s)][(int)(j/s)];
	// implementing the basic idea


	*(int*)&img_temp->header[22]=height;
	*(int*)&img_temp->header[18]=width;
	// making changes in header of image


printf("width %d height %d",*(int*)&img_temp->header[18],
*(int*)&img_temp->header[22]);
	//printing height, width for confirmation


	// writing into a file
	FILE* fo=fopen("C:/Images/enlarge.bmp","wb+");
	fwrite(img_temp->header,sizeof(unsigned char),54,fo);
	fwrite(temp,sizeof(pixel),height*width,fo);

	fclose(fo);
	free(img_temp);
	free(temp);
}
```

As a matter of fact, since enlarging and shrinking use the same principles, we could have created a single function for both. But, since different functions were asked, it had to be done.

Enlarged by 2.5



Properties of enlarged image(dimensions are multiplied by 2.5)

The Translate(image img, int t) function*

This function involves translating the image by **t** pixels in the four directions, depending on the choice of the user.

The basic idea is to shift all the pixels in the original image by **t** pixels. Then, in the region, we convert all the pixels to black. The code is given below:

```
void Translate(image* img,int t,int dirn)
{
		image* img_temp=malloc(sizeof(image));
		*img_temp=*img;

		switch(dirn)
		{
			case 1: // towards right

			{
				for(int i=0;i<img_temp->height;i++)
					for(int j=img_temp->width-1;j>=t;j--)
				// shifting pixels to the right
			img_temp->buffer[i][j]=img_temp->buffer[i][j-t];

				for(int i=0;i<img->height;i++)
					for(int j=0;j<t;j++)
			// colouring the required area black
img_temp->buffer[i][j].red=img_temp->buffer[i][j].blue=
img_temp->buffer[i][j].green=0;

			}break;

			case 2:		// towards left
			{
				for(int i=0;i<img_temp->height;i++)
					for(int j=t;j<img->width;j++)
			img_temp->buffer[i][j-t]=img_temp->buffer[i][j];
				// shifting pixels to the left

				for(int i=0;i<img->height;i++)
					for(int j=img->width-t;j<img->width;j++)
					// colouring the required area black
img_temp->buffer[i][j].red=img_temp->buffer[i][j].blue=
img_temp->buffer[i][j].green=0;

			}break;

			case 3:		// upwards
			{
```

```
                        for(int j=0;j<img_temp->width;j++)
                              for(int i=img->height-1;i>=t;i--)
                        img_temp->buffer[i][j]=img_temp->buffer[i-t][j];
                              // shifting pixels upwards

                        for(int j=0;j<img_temp->width;j++)
                              for(int i=0;i<t;i++)

img_temp->buffer[i][j].red=img_temp->buffer[i][j].blue=
img_temp->buffer[i][j].green=0;
                              // colouring the required area black
                  }break;

                  case 4:     // downwards
                  {
                        for(int j=0;j<img_temp->width;j++)
                              for(int i=t;i<img->height;i++)
                  img_temp->buffer[i-t][j]=img_temp->buffer[i][j];
                              // shifting pixels downwards

                        for(int j=0;j<img_temp->width;j++)
                        for(int i=img->height-1;i>=img->height-t;i--)
                              // colouring required area black
img_temp->buffer[i][j].red=img_temp->buffer[i][j].blue=
img_temp->buffer[i][j].green=0;

                  }break;

                  default: break;
            }

                  // writing into a file

                  FILE* f=NULL;
                  strcpy(img_temp->path,"C:/Images/translate.bmp");
                  writeImage(f,img_temp);
                  free(img_temp);
}
```

Translated up by 100 pixels

Translated down by 100 pixels

Translated left by 100 pixels

Translated right by 100 pixels

The Rotate(image img, int theta) function*

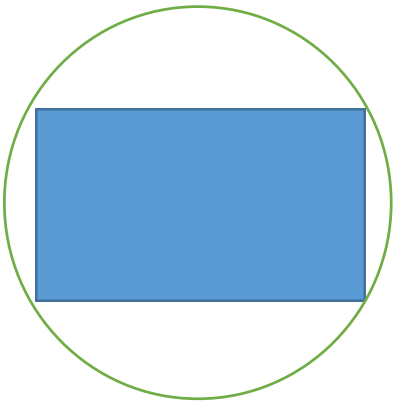
This function deals with rotating an image through any arbitrary angle, if theta being positive, in the clockwise direction, and counter clockwise if theta is negative. Firstly, we declare a buffer representing a square with side given by:

$$\textbf{Side (square buffer) = root (height}^2\textbf{+width}^2\textbf{)}$$

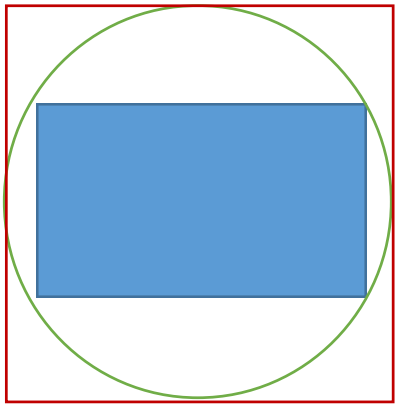
Where height, width represent dimensions of source image.

Consider an image:

It will rotate within a circle that circumscribes it,



This circle is further circumscribed by a square:



This outer square is the buffer on which pixels from original image will fall on. As rotation is done about the centre of the original image, using the centre of the image as the reference and by a little bit of trigonometry, the transformation formula from source image to destination is:

new_y = cos (theta)*(i-y_centre)-sin (theta)*(j-x_centre) + y_centre+(side-height)/2

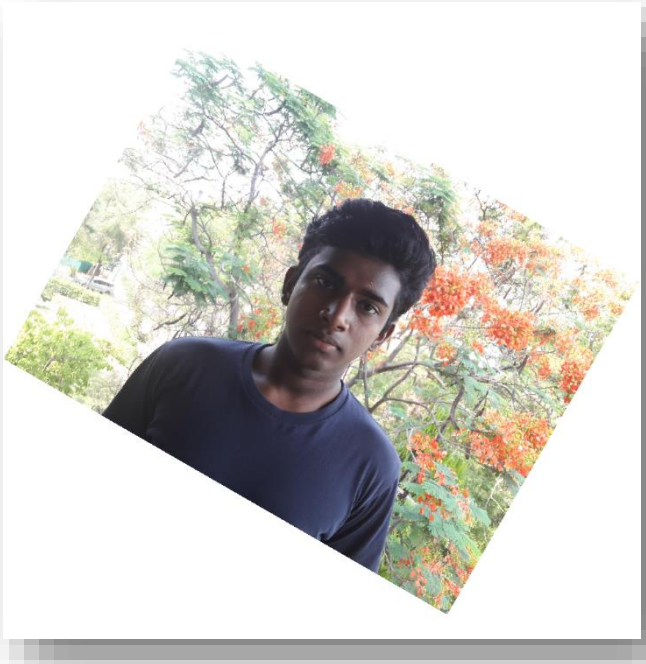
new_x=sin (theta)*(i-y_centre)+cos (theta)*(j-x_centre) + x_centre+(side-width)/2

Where x_centre, y_centre represent the x, y coordinates of the centre of source image, and side is as defined earlier.

After the mapping, interpolation by nearest neighbour algorithm is done in the neighbourhood of the pixel with new coordinates, to ensure that the resultant image is devoid of any random white spots. The full code of the function is given below:

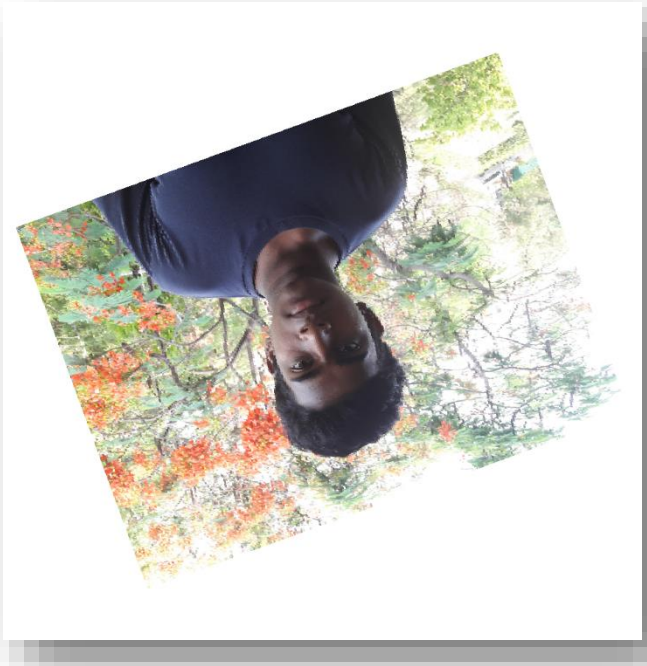
```
void Rotate(image* img,int theta)
{
	image* img_temp=malloc(sizeof(image));
	*img_temp=*img;

	// converting from degrees to radians

	float rad=theta*(float)3.141592/180;

	// defining centre of source image

	int x_center=img_temp->width/2;
	int y_center=img_temp->height/2;

	printf("width %d height %d",x_center,y_center);

	// computing length of square buffer

	int len=sqrt(pow(img->width,2)+pow(img->height,2));
	if(len%4!=0) len+=4-len%4;

	printf("len %d",len);

	// buffer declaration
	pixel** temp=malloc(len*sizeof(pixel*));
	for(int i=0;i<len;i++) temp[i]=malloc(len*sizeof(pixel));

	// assigning every pixel the colour white in temp buffer

	for(int i=0;i<len;i++)
		for(int j=0;j<len;j++)
		temp[i][j].red=temp[i][j].green=temp[i][j].blue=255;
```

```
for(int i=0;i<img->height;i++)
      {
            for(int j=0;j<img->width;j++)
            {
              // assigning new coordinates to the pixel

                  int new_vert=(int)(cos(rad)*(i-y_center)-
sin(rad)*(j-x_center)+y_center+(len-img->height)/2);

                  int new_hrzn=(int)(sin(rad)*(i-y_center)+
cos(rad)*(j-x_center)+x_center+(len-img->width)/2);

// checking if interpolation will result in going 'out-of bounds' of the
buffer.

            if(new_vert - 1 >=0 && new_vert + 1 < len &&
            new_hrzn - 1 >=0 && new_hrzn + 1 <len)

// if okay, then interpolation is done to cover up stray white points

                  for(int k=-1;k<=1;k++) for(int l=-1;l<=1;l++)
            temp[new_vert+k][new_hrzn+l]=img_temp->buffer[i][j];


            }
      }

// updating header values

                        *(int*)&img_temp->header[18]=len;
                        *(int*)&img_temp->header[22]=len;

// writing into a file

            FILE* f=fopen("C:/Images/rot.bmp","wb");
            fwrite(img_temp->header,sizeof(unsigned char),54,f);

                        for(int i=0;i<len;i++)
                         fwrite(temp[i],sizeof(pixel),len,f);

                        fclose(f);

                        for(int i=0;i<len;i++) free(temp[i]);
                        free(temp);
}
```



Rotation by 30 clockwise

Rotation by 45 clockwise



Rotation by 200 anti clockwise

Rotation by 90 clockwise

Morph(image img, image* img2, float f) function*

This function is used to morph two images by weighted addition of colours. The parameter **f** is the merging factor. More the value of f or the merging factor, the characteristics of the first image will be more prevalent that those of the second. The code is given below:

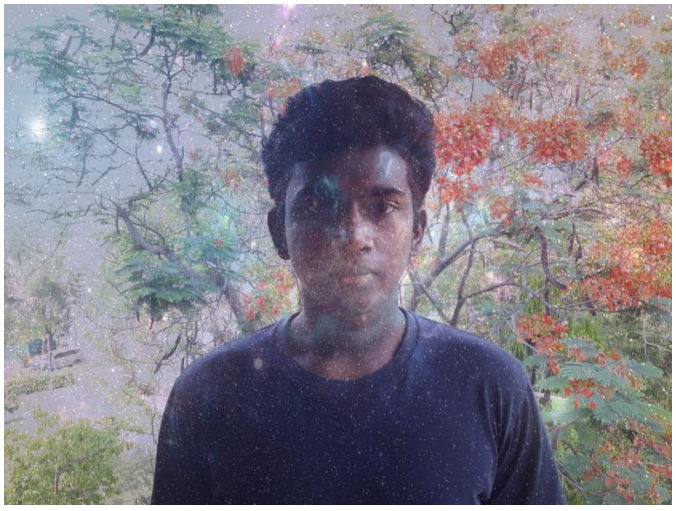
```
void Morph(image* img,image* img2,float f)
{
	image *img3=malloc(sizeof(image));
	*img3=*img;

	if(img->height!=img2->height || img->width!=img2->width ||
img->bitdepth!=img2->bitdepth) {}
// checking if image characteristics are same

	else
	{
		for(int i=0;i<img3->height;i++)
		for(int j=0;j<img3->width;j++)
		{

// weighted addition of red components
img3->buffer[i][j].red=(int)((float)img->buffer[i][j].red*(f)+
(float)img2->buffer[i][j].red*(1-f));

// weighted addition of green components
img3->buffer[i][j].green=(int)((float)img->buffer[i][j].green*(f)+
(float)img2->buffer[i][j].green*(1-f));

// weighted addition of blue components
img3->buffer[i][j].blue=(int)((float)img->buffer[i][j].blue*(f)+
(float)img2->buffer[i][j].blue*(1-f));

		}
	}

	// writing into a file

	FILE* fo=NULL;
	strcpy(img3->path,"C:/Images/morph.bmp");
	writeImage(fo,img3);
	free(img3);
}
```

Original

Second Image

0.35 morph

0.5 morph



0.65 morph

0.85 morph

The Compare(image img, image* img2)function*

This function is used to compare two images and generate their absolute difference. The gist of this function is that it takes two pointers to the images under comparison and takes the absolute difference of each of the RGB components of the corresponding pixels. A counter variable is incremented with the absolute difference. Then, the percentage difference is displayed, along with the generation of an image that can be seen as the absolute difference. The code is given below:

```
void Compare(image* img,image* img2)
{
	image* img3=malloc(sizeof(image));
	*img3=*img;
	if(img->height!=img2->height || img->width!=img2->width ||
 img->bitdepth!=img2->bitdepth) {}

// checking if image characteristics are same
	else
	{
		float sum=0;
		for(int i=0;i<img->height;i++)
		{
			for(int j=0;j<img->width;j++)
			{

			img3->buffer[i][j].red=
			abs(img->buffer[i][j].red-img2->buffer[i][j].red);
// absolute difference of red components
				sum+=img3->buffer[i][j].red;

			img3->buffer[i][j].green=
			abs(img->buffer[i][j].green-img2->buffer[i][j].green);
// absolute difference of green components
				sum+=img3->buffer[i][j].green;


			img3->buffer[i][j].blue=
			abs(img->buffer[i][j].blue-img2->buffer[i][j].blue);
// absolute difference of blue components
				sum+=img3->buffer[i][j].blue;
			}
		}

		// calculation and display of percentage difference
		sum/=img->height*img->width*3*255;
		sum*=100;
		printf("%f percent difference",sum);

		// writing into file
		FILE* f=NULL;
		strcpy(img3->path,"C:/Images/compare.bmp");
		writeImage(f,img3);
```

```
            free(img3);
        }
}
```

Image 1

Image 2

```
:24.814980 percent difference
```

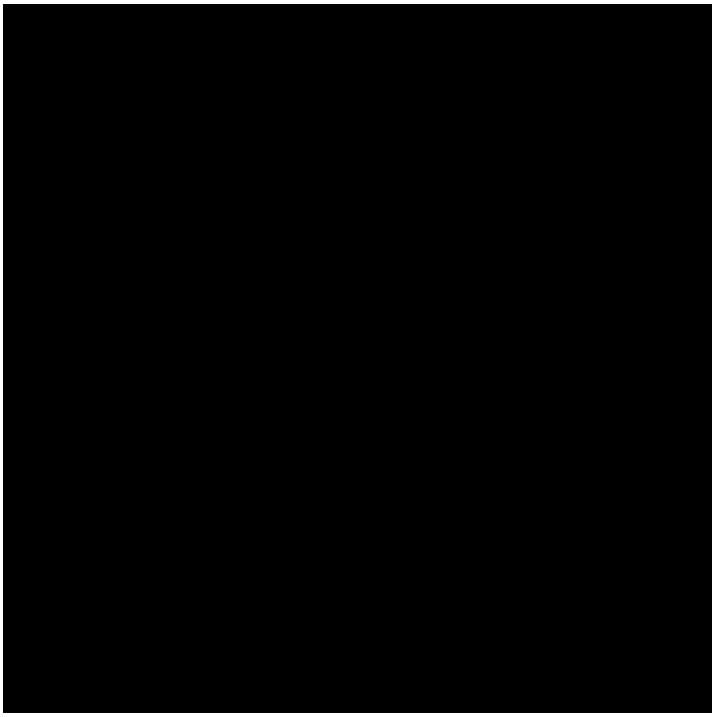
Console output after comparison

Result of absolute difference (more dark regions imply more similarity)

Image 2(same as image 1)

```
0.000000 percent difference
```

Console output are comparison with itself



Absolute difference of itself (images are same)

The Negative(image img) function*

This function allows us to generate the negative of an image. For generating the negative, we use the following formula:

Negative of colour = 255 – colour

The code is given below:

```
void Negative(image* img)
{
	image *img_temp=malloc(sizeof(image));
	*img_temp=*img;

	for(int i=0;i<img_temp->height;i++)
	{
		for(int j=0;j<img_temp->width;j++)
		{
		  // computing the negative of each component
img_temp->buffer[i][j].red=255-img_temp->buffer[i][j].red;

img_temp->buffer[i][j].green=255-img_temp->buffer[i][j].green;

img_temp->buffer[i][j].blue=255-img_temp->buffer[i][j].blue;
		}
	}

		  // writing into a file

			FILE* fo=fopen("C:/Images/negative.bmp","wb+");
			fwrite(img_temp->header,sizeof(unsigned char),54,fo);

			for(int i=0;i<img_temp->height;i++)
fwrite(img_temp->buffer[i],sizeof(pixel),img_temp->width,fo);

			fclose(fo);
			free(img_temp);
}
```

Original

Negative

REFERENCES

- https://github.com/abhijitnathwani/image-processing
- https://github.com/abhijitnathwani/image-processing/blob/master/image_rotate.c
- https://stackoverflow.com/questions/40677838/c-bmp-rotate-image?noredirect=1&lq=1
- https://stackoverflow.com/questions/14184700/how-to-rotate-image-x-degrees-in-c
- https://stackoverflow.com/questions/23917887/rotate-an-image-by-n-degrees?noredirect=1&lq=1
- https://stackoverflow.com/questions/2654480/writing-bmp-image-in-pure-c-c-without-other-libraries
- https://stackoverflow.com/questions/9296059/read-pixel-value-in-bmp-file/43140660
- https://stackoverflow.com/questions/17615963/standard-rgb-to-grayscale-conversion
- https://etc.usf.edu/techease/win/images/what-is-bit-depth/
- https://stackoverflow.com/questions/5359047/reading-the-bgr-colors-from-a-bitmap-in-c
- https://www.stemmer-imaging.com/en/knowledge-base/grey-level-grey-value/
- https://en.wikipedia.org/wiki/Grayscale
- https://stackoverflow.com/questions/32507061/convert-gray-image-to-binary-image-0-1-image-in-c
- https://abhijitnathwani.github.io/blog/2018/01/08/RGB-Image-Basics
- https://stackoverflow.com/questions/32310989/bmp-char-array-to-file-c-c?noredirect=1&lq=1
- https://stackoverflow.com/questions/44961084/scaling-down-a-bmp-image-in-c
- https://www.canva.com/learn/how-to-use-photo-filters-to-enhance-your-images/
- http://ndevilla.free.fr/median/median.pdf
- https://lodev.org/cgtutor/filtering.html
- http://homepages.inf.ed.ac.uk/rbf/HIPR2/mean.htm
- https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm
- https://github.com/jrahim/CPP-Sobel-Edge-Detection/blob/master/sobelEdgeDetection.cpp