INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

DIGITAL IMAGE PROCESSING LABORATORY

A REPORT ON

EXPERIMENT 01

Image Read, Write and Basic Geometrical Transforms

Name :	Parakh Agarwal	Anand Jhunjhunwala
Roll. No.:	17EC35016	17EC35032

25.01.2021

Group No. 2

DEPT OF ELECTRONICS AND ELECTRICAL COMMUNICATION ENGINEERING

VISUAL INFORMATION AND EMBEDDED SYSTEMS

Table of Contents

Objective:

Write C/C++ modular functions to read, perform transformations (Grayscale conversion, Diagonal flip, Rotation by 90 and 45-degree, Scaling), and then write BMP image files.

Introduction:

BMP Image:
The BMP file format, also known as bitmap image file, device independent bitmap (DIB) file format and bitmap, is a raster graphics image file format used to store bitmap digital images, independently of the display device (such as a graphics adapter), especially on Microsoft Windows and OS/2 operating systems.

The BMP file format is capable of storing two-dimensional digital images both monochrome and color, in various color depths, and optionally with data compression, alpha channels, and color profiles.

The file format consists of the following structures:

Structure	**Corresponding Bytes**	**Description**
Header	0x00 - 0x0D	contains information about the type, size, and layout of a device-independent bitmap file
InfoHeader	0x0E - 0x35	specifies the dimensions, compression type, and color format for the bitmap
ColorTable	0x36 - variable	contains as many elements as there are colors in the bitmap, but is not present for bitmaps with more than 8 color bits
Pixel Data	variable	These are the actual image data, represented by consecutive rows, or "scan lines," of the bitmap. The system maps pixels beginning with the bottom scan line of the rectangular region and ending with the top scan line.

Below is a more detailed table of the contents of each of these structures.

Name		**Size**	**Offset**	**Description**
Header		**14 bytes**		**Windows Structure: BITMAPFILEHEADER**
	Signature	2 bytes	0000h	'BM'

	FileSize	4 bytes	0002h	File size in bytes
	reserved	4 bytes	0006h	unused (=0)
	DataOffset	4 bytes	000Ah	Offset from beginning of file to the beginning of the bitmap data
InfoHeader		**40 bytes**		**Windows Structure: BITMAPINFOHEADER**
	Size	4 bytes	000Eh	Size of InfoHeader =40
	Width	4 bytes	0012h	Horizontal width of bitmap in pixels
	Height	4 bytes	0016h	Vertical height of bitmap in pixels
	Planes	2 bytes	001Ah	Number of Planes (=1)
	Bits Per Pixel	2 bytes	001Ch	Bits per Pixel used to store palette entry information. This also identifies in an indirect way the number of possible colors. Possible values are: 1 = monochrome palette. NumColors = 1 4 = 4bit palletized. NumColors = 16 8 = 8bit palletized. NumColors = 256 16 = 16bit RGB. NumColors = 65536 24 = 24bit RGB. NumColors = 16M
	Compression	4 bytes	001Eh	Type of Compression 0 = BI_RGB no compression 1 = BI_RLE8 8bit RLE encoding 2 = BI_RLE4 4bit RLE encoding
	ImageSize	4 bytes	0022h	Size of Image It is valid to set this = 0 if Compression = 0
	XpixelsPerM	4 bytes	0026h	horizontal resolution: Pixels/meter
	YpixelsPerM	4 bytes	002Ah	vertical resolution: Pixels/meter
	Colors Used	4 bytes	002Eh	Number of actually used colors. For a 8-bit / pixel bitmap this will be 100h or 256.
	Important Colors	4 bytes	0032h	Number of important colors 0 = all

ColorTable	4 * NumColors bytes	0036h	present only if Info.BitsPerPixel less than or equal to 8 colors should be ordered by importance
Red	1 byte		Red intensity
Green	1 byte		Green intensity
Blue	1 byte		Blue intensity
reserved	1 byte		unused (=0)
repeated NumColors times			
Pixel Data	InfoHeader .ImageSize bytes		The image data

It is important to note that each scan line is zero padded to the nearest 4-byte boundary so if the image has a width that is not divisible by four, say, 21 bytes, there would be 3 bytes of padding at the end of every scan line. Also, the scan lines are stored bottom to top instead of top to bottom.
Also, it must be remembered that RGB values are stored backwards i.e. BGR.

Grayscale conversion:
Method of converting a colour RGB image to grayscale is known as grayscale conversion, there are mainly two methods to convert it:

- Average method
- Weighted method or luminosity method

Average method: In this method, grayscale value is just the average of all the 3 colour channels present in the colour image. i.e

Grayscale = (R + G + B / 3)

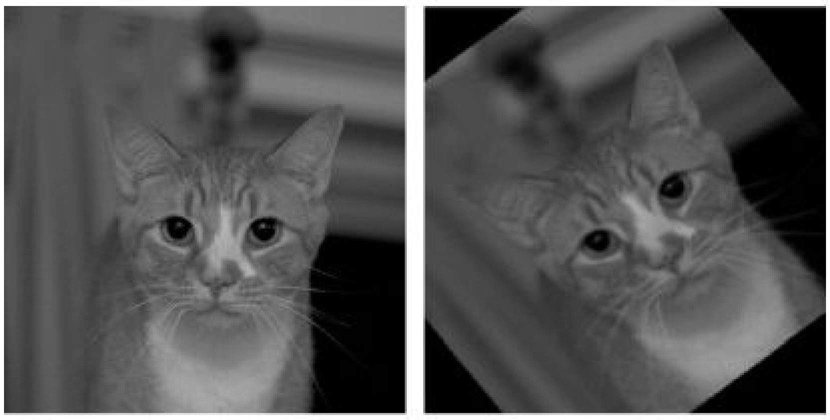
Problem: Since the three different colors have three different wavelengths and have their own contribution in the formation of image, so we have to take average according to their contribution, not by using the average method.

Weighted method: Since red color has more wavelength of all the three colors, and green is the color that has not only less wavelength then red color but also green is the color that gives more soothing effect to the eyes. So we have to decrease the contribution of red color, and increase the contribution of the green color, and put blue color contribution in between these two. This is done in this method and grayscale value is given by:

Grayscale = ((0.3 * R) + (0.59 * G) + (0.11 * B))

Diagonal Flip:
By Diagonal flip, we simply mean transpose of the image matrix. If the origin of the image is placed in the bottom left corner then this operation will flip the image w.r.t sub diagonal and if origin is placed at upper left corner then this operation will flip the image w.r.t main diagonal.
Image Rotation:
Image rotation is a common image processing routine with applications in matching, alignment, and other image-based algorithms. The input to an image rotation routine is an image, the rotation angle θ, and a point about which rotation is done, and the aim is to rotate the image by angle θ about a given point.

See Image below:



Original image After rotation of 45°

Mathematically:
The coordinates of a point (x1, y1) when rotated by an angle θ around (x0, y0) become (x2, y2), as shown by the following equation:

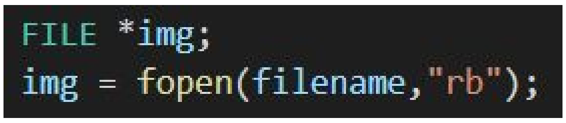
$$x2=\cos(\theta)*(x1-x0)+\sin(\theta)*(y1-y0)$$
$$y2=-\sin(\theta)*(x1-x0)+\cos(\theta)*(y1-y0)$$

Image Scaling:
By image scaling we refer to the resizing of a digital image. When we tend to increase the size of an image we need to fill new pixels with appropriate values without losing the original image context. For this we use interpolation which refers to the method of calculating unknown pixel values using the values of its known neighbours. In this assignment, we had used Nearest-neighbor interpolation, which is one of the simplest ways of increasing image size by replacing every pixel with the nearest pixel in the output, for upscaling this means multiple pixels of the same color will be present. This can preserve sharp details in pixel art, but also introduce jaggedness in previously smooth images.

Algorithm:

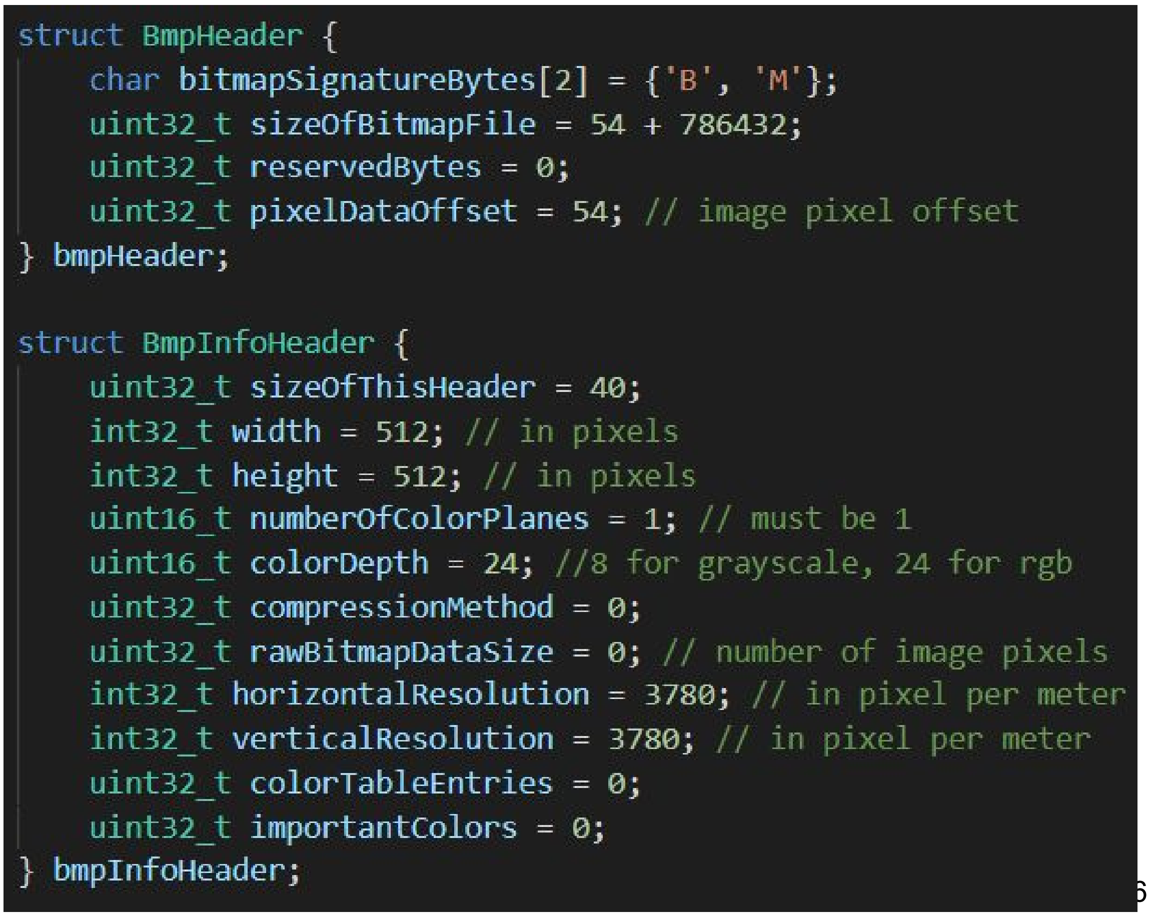
In this section we tried to explain our implementation of the different algorithms used for above transformations.
Note: We have used vector<vector<vector<int>>> to save the image array, the last dimension of this data structure tells us whether the image is grayscale or colour, i.e if last dimension is 1 then image is grayscale else colour.

1) **Reading BMP File:**
   We formulated the C++ functions from scratch for carrying out the reading tasks. We used the FILE object type to identify a stream and control the information needed to control it. It provides us with a pointer to the buffer of the file through which we can read the file character by character. We created the FILE object using fopen by reading the file in binary mode. The “rb” states we just want to read the binary file.



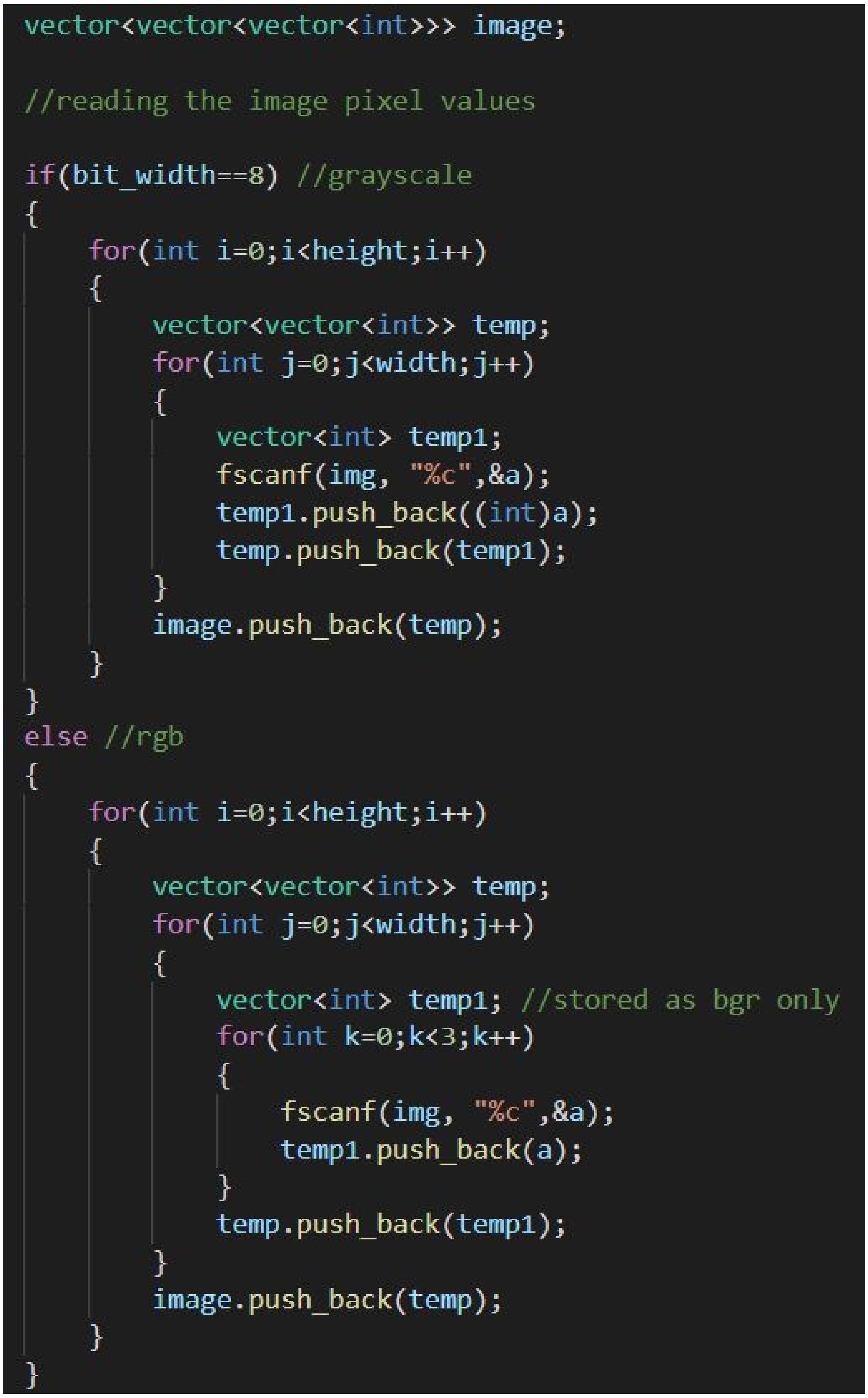
```
FILE *img;
img = fopen(filename,"rb");
```

   We want to read a single byte at once and hence we are using unsigned char to read the data from the file. First we check whether the file is a BMP file or not by using the first two bytes of the file. We further read the remaining data of the file byte by byte and store it in the structures defined for easy access.
   We identify the width and height of the image through this as well as the colorDepth is used to identify if the image is grayscale or is RGB.



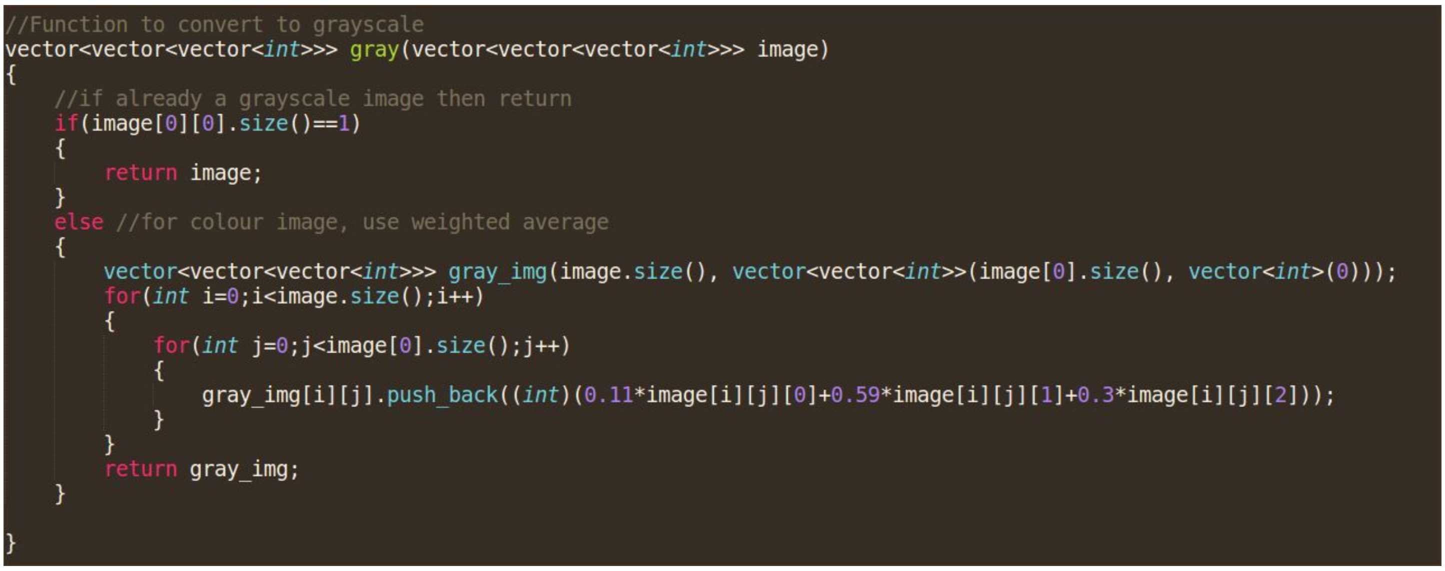
```
struct BmpHeader {
    char bitmapSignatureBytes[2] = {'B', 'M'};
    uint32_t sizeOfBitmapFile = 54 + 786432;
    uint32_t reservedBytes = 0;
    uint32_t pixelDataOffset = 54; // image pixel offset
} bmpHeader;

struct BmpInfoHeader {
    uint32_t sizeOfThisHeader = 40;
    int32_t width = 512; // in pixels
    int32_t height = 512; // in pixels
    uint16_t numberOfColorPlanes = 1; // must be 1
    uint16_t colorDepth = 24; //8 for grayscale, 24 for rgb
    uint32_t compressionMethod = 0;
    uint32_t rawBitmapDataSize = 0; // number of image pixels
    int32_t horizontalResolution = 3780; // in pixel per meter
    int32_t verticalResolution = 3780; // in pixel per meter
    uint32_t colorTableEntries = 0;
    uint32_t importantColors = 0;
} bmpInfoHeader;
```

Important to note that some fields require more than one byte, so the first encountered byte of the data is the least significant byte.



```
vector<vector<vector<int>>> image;

//reading the image pixel values

if(bit_width==8) //grayscale
{
    for(int i=0;i<height;i++)
    {
        vector<vector<int>> temp;
        for(int j=0;j<width;j++)
        {
            vector<int> temp1;
            fscanf(img, "%c",&a);
            temp1.push_back((int)a);
            temp.push_back(temp1);
        }
        image.push_back(temp);
    }
}
else //rgb
{
    for(int i=0;i<height;i++)
    {
        vector<vector<int>> temp;
        for(int j=0;j<width;j++)
        {
            vector<int> temp1; //stored as bgr only
            for(int k=0;k<3;k++)
            {
                fscanf(img, "%c",&a);
                temp1.push_back(a);
            }
            temp.push_back(temp1);
        }
        image.push_back(temp);
    }
}
```

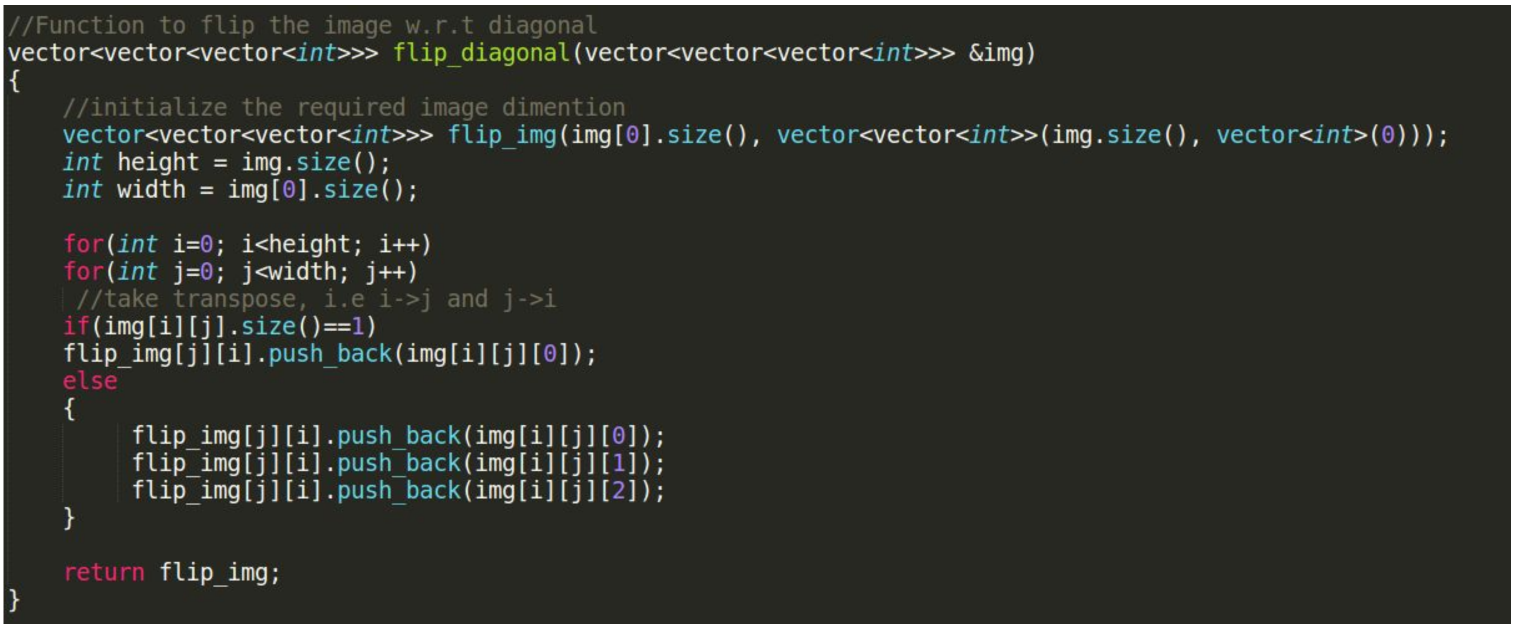
2) **Grayscale conversion:**
We have used the Weighted method to convert the image to grayscale, function first check if the image is coloured or not, and if it is coloured then it uses weighted averaging to convert it to grayscale.
See function snippet below.



```
//Function to convert to grayscale
vector<vector<vector<int>>> gray(vector<vector<vector<int>>> image)
{
    //if already a grayscale image then return
    if(image[0][0].size()==1)
    {
        return image;
    }
    else //for colour image, use weighted average
    {
        vector<vector<vector<int>>> gray_img(image.size(), vector<vector<int>>(image[0].size(), vector<int>(0)));
        for(int i=0;i<image.size();i++)
        {
            for(int j=0;j<image[0].size();j++)
            {
                gray_img[i][j].push_back((int)(0.11*image[i][j][0]+0.59*image[i][j][1]+0.3*image[i][j][2]));
            }
        }
        return gray_img;
    }

}
```

3) **Diagonal Flip:**

Since diagonal flip is equivalent to taking the transpose of the image array, this function simply takes the transpose of the matrix(image array) provided to it. Moreover since our origin is located at the bottom left corner, so it will flip the image about the sub diagonal.



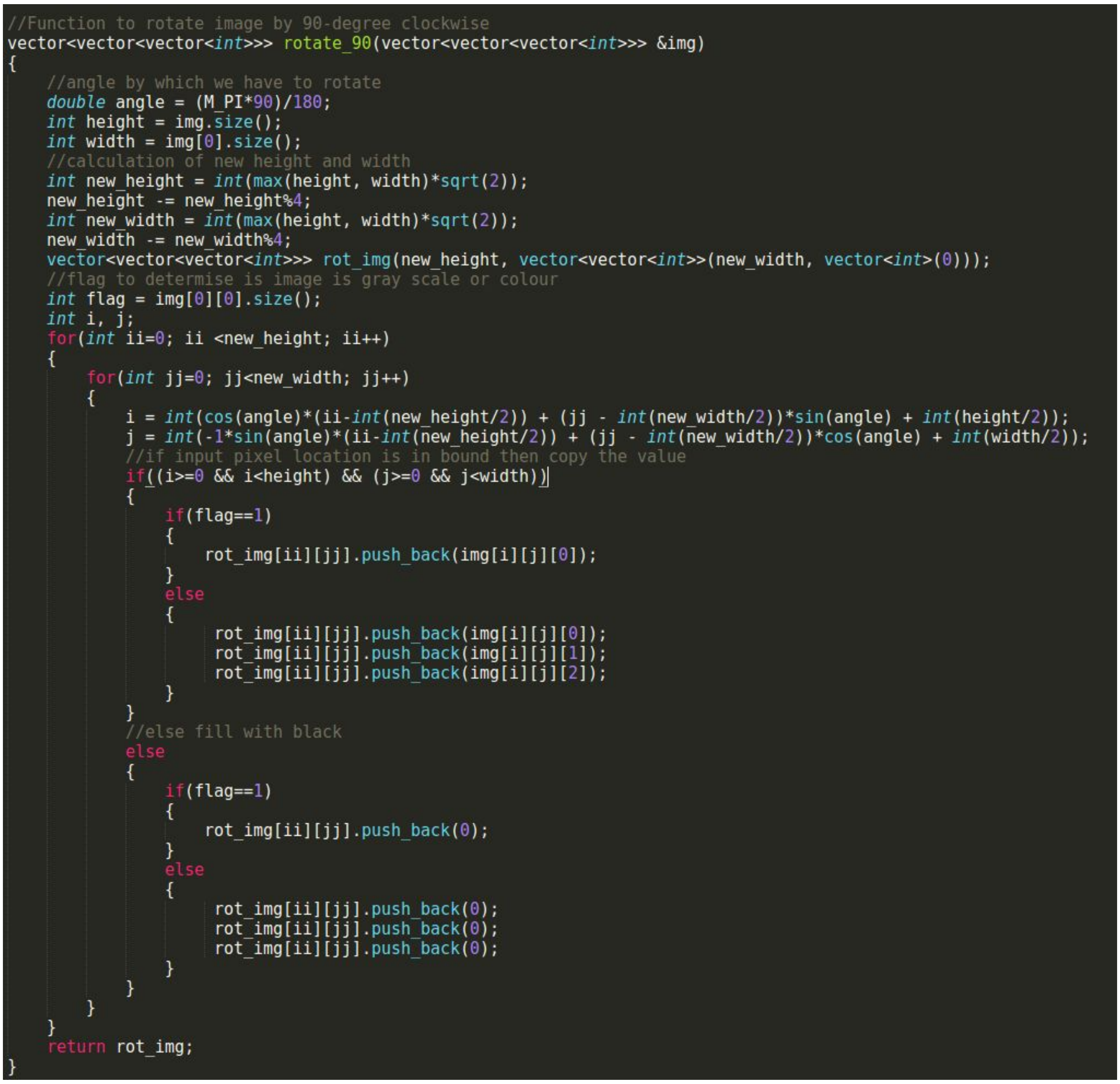
```
//Function to flip the image w.r.t diagonal
vector<vector<vector<int>>> flip_diagonal(vector<vector<vector<int>>> &img)
{
    //initialize the required image dimention
    vector<vector<vector<int>>> flip_img(img[0].size(), vector<vector<int>>(img.size(), vector<int>(0)));
    int height = img.size();
    int width = img[0].size();

    for(int i=0; i<height; i++)
    for(int j=0; j<width; j++)
     //take transpose, i.e i->j and j->i
    if(img[i][j].size()==1)
    flip_img[j][i].push_back(img[i][j][0]);
    else
    {
         flip_img[j][i].push_back(img[i][j][0]);
         flip_img[j][i].push_back(img[i][j][1]);
         flip_img[j][i].push_back(img[i][j][2]);
    }

    return flip_img;
}
```

4) Rotate Image

Due to rotation, some pixels of the original image map to pixels which are non-integers because of which we observe circular black patches in the resulting image. To fill those images we use interpolation, in this assignment we have used Nearest-neighbor interpolation. Moreover since rotation results in an increase of size of image (to preserve information), to accommodate that we increased the width and height of image by a factor of sqrt(2).

To implement Nearest-neighbor interpolation, we have used reverse mapping, we mapped each output image pixel to its nearest pixel in the input image and if that pixel lies outside the input image we fill it with colour black. The height/2 and width/2 factor is used to rotate the image w.r.t the center of the image.

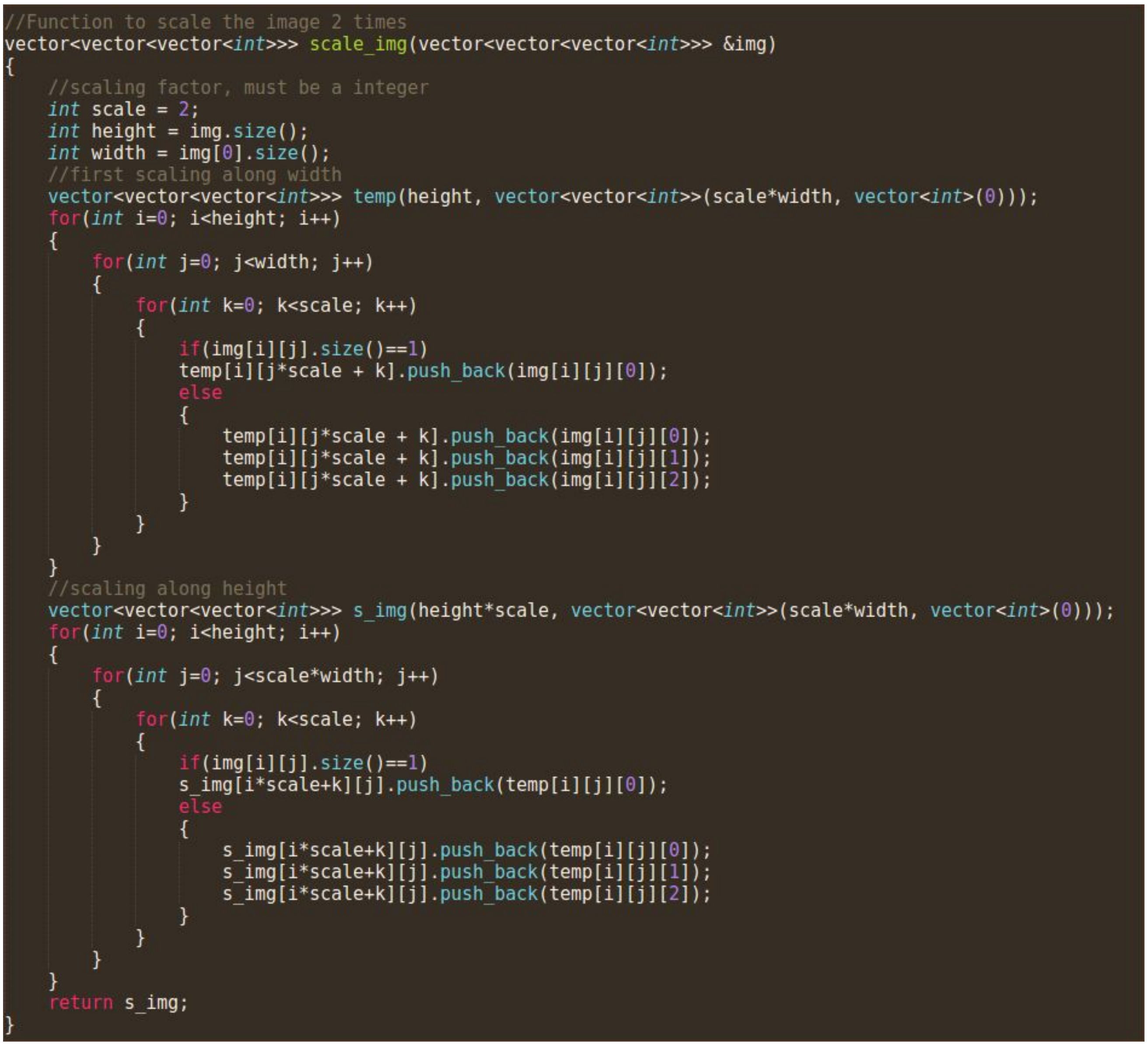
Below function shows the function for 90 degree rotation, same function is used for 45 degree rotation with only constant value changed.



```cpp
//Function to rotate image by 90-degree clockwise
vector<vector<vector<int>>> rotate_90(vector<vector<vector<int>>> &img)
{
    //angle by which we have to rotate
    double angle = (M_PI*90)/180;
    int height = img.size();
    int width = img[0].size();
    //calculation of new height and width
    int new_height = int(max(height, width)*sqrt(2));
    new_height -= new_height%4;
    int new_width = int(max(height, width)*sqrt(2));
    new_width -= new_width%4;
    vector<vector<vector<int>>> rot_img(new_height, vector<vector<int>>(new_width, vector<int>(0)));
    //flag to determise is image is gray scale or colour
    int flag = img[0][0].size();
    int i, j;
    for(int ii=0; ii <new_height; ii++)
    {
        for(int jj=0; jj<new_width; jj++)
        {
            i = int(cos(angle)*(ii-int(new_height/2)) + (jj - int(new_width/2))*sin(angle) + int(height/2));
            j = int(-1*sin(angle)*(ii-int(new_height/2)) + (jj - int(new_width/2))*cos(angle) + int(width/2));
            //if input pixel location is in bound then copy the value
            if((i>=0 && i<height) && (j>=0 && j<width))
            {
                if(flag==1)
                {
                    rot_img[ii][jj].push_back(img[i][j][0]);
                }
                else
                {
                     rot_img[ii][jj].push_back(img[i][j][0]);
                     rot_img[ii][jj].push_back(img[i][j][1]);
                     rot_img[ii][jj].push_back(img[i][j][2]);
                }
            }
            //else fill with black
            else
            {
                if(flag==1)
                {
                    rot_img[ii][jj].push_back(0);
                }
                else
                {
                     rot_img[ii][jj].push_back(0);
                     rot_img[ii][jj].push_back(0);
                     rot_img[ii][jj].push_back(0);
                }
            }
        }
    }
    return rot_img;
}
```

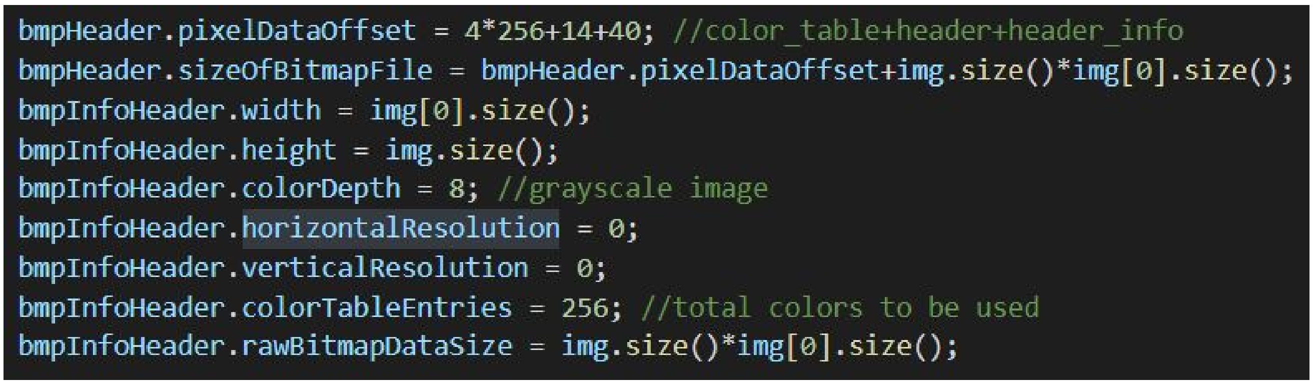
5) **Image Scaling**
   We have implemented image scaling using Nearest-neighbor interpolation, for doing so we first expand the input image along its width then use that image to expand about height which results in a final scaled image. To ensure Nearest-neighbor interpolation, we copied the pixel value directly from the input image pixel depending upon the scaling factor, which is 2 in our case.

   See function snippet below.



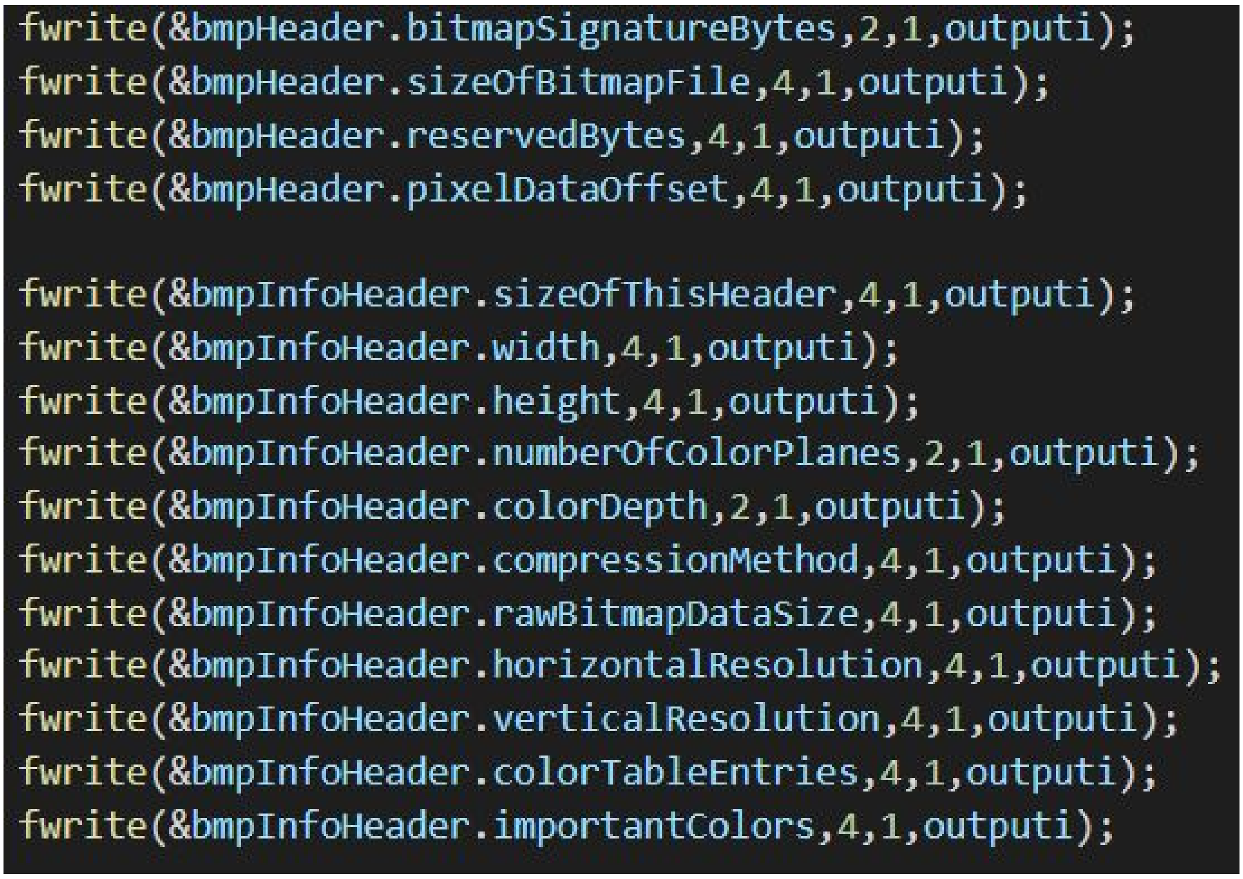
```
//Function to scale the image 2 times
vector<vector<vector<int>>> scale_img(vector<vector<vector<int>>> &img)
{
    //scaling factor, must be a integer
    int scale = 2;
    int height = img.size();
    int width = img[0].size();
    //first scaling along width
    vector<vector<vector<int>>> temp(height, vector<vector<int>>(scale*width, vector<int>(0)));
    for(int i=0; i<height; i++)
    {
        for(int j=0; j<width; j++)
        {
            for(int k=0; k<scale; k++)
            {
                if(img[i][j].size()==1)
                temp[i][j*scale + k].push_back(img[i][j][0]);
                else
                {
                    temp[i][j*scale + k].push_back(img[i][j][0]);
                    temp[i][j*scale + k].push_back(img[i][j][1]);
                    temp[i][j*scale + k].push_back(img[i][j][2]);
                }
            }
        }
    }
    //scaling along height
    vector<vector<vector<int>>> s_img(height*scale, vector<vector<int>>(scale*width, vector<int>(0)));
    for(int i=0; i<height; i++)
    {
        for(int j=0; j<scale*width; j++)
        {
            for(int k=0; k<scale; k++)
            {
                if(img[i][j].size()==1)
                s_img[i*scale+k][j].push_back(temp[i][j][0]);
                else
                {
                    s_img[i*scale+k][j].push_back(temp[i][j][0]);
                    s_img[i*scale+k][j].push_back(temp[i][j][1]);
                    s_img[i*scale+k][j].push_back(temp[i][j][2]);
                }
            }
        }
    }
    return s_img;
}
```

6) **Writing BMP File:**
   We open the BMP file to be written in the “wb” mode for writing the binary file. We are required to save a grayscale image only so we modified the required fields to save the image. The rest of the header fields remain the same and thus are fed directly from the structure specified.



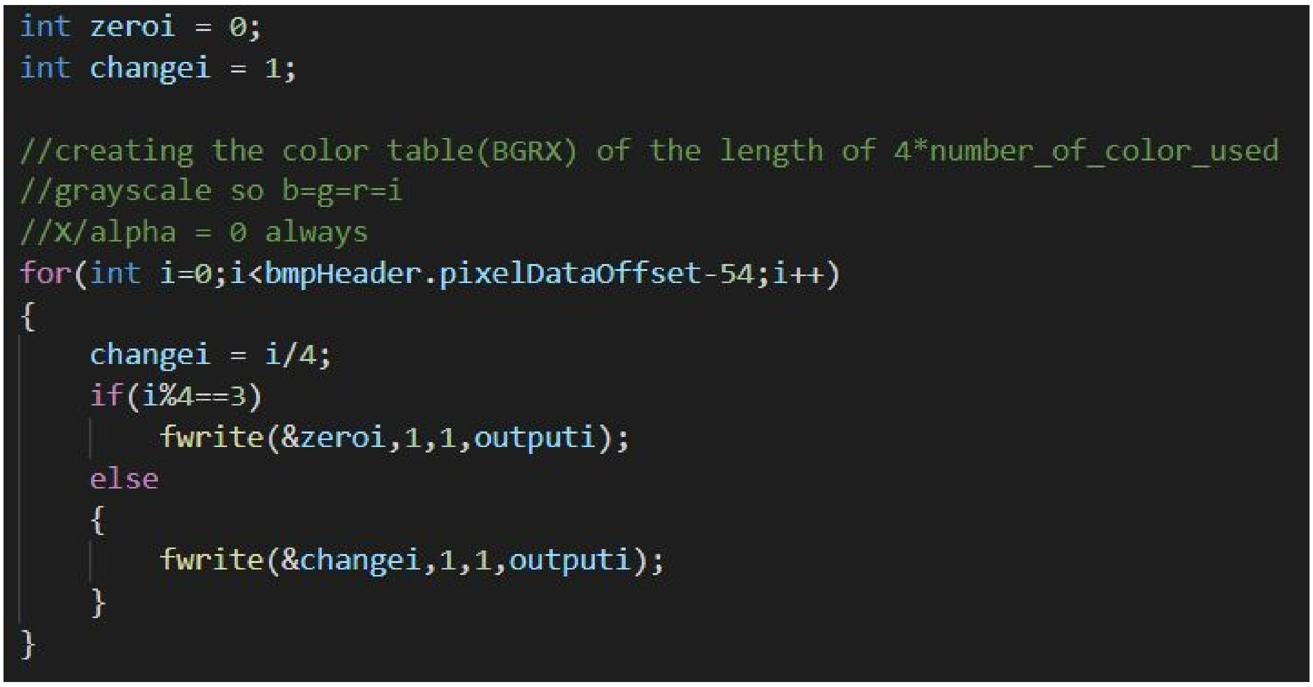
```
bmpHeader.pixelDataOffset = 4*256+14+40; //color_table+header+header_info
bmpHeader.sizeOfBitmapFile = bmpHeader.pixelDataOffset+img.size()*img[0].size();
bmpInfoHeader.width = img[0].size();
bmpInfoHeader.height = img.size();
bmpInfoHeader.colorDepth = 8; //grayscale image
bmpInfoHeader.horizontalResolution = 0;
bmpInfoHeader.verticalResolution = 0;
bmpInfoHeader.colorTableEntries = 256; //total colors to be used
bmpInfoHeader.rawBitmapDataSize = img.size()*img[0].size();
```

The bytes are written using the function fwrite where the input is the address to the data, number of bytes it will be using, number of such inputs and the FILE object.



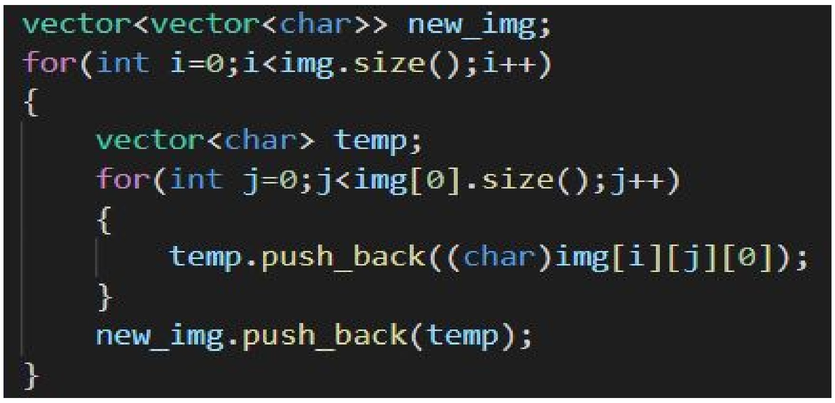
```
fwrite(&bmpHeader.bitmapSignatureBytes,2,1,outputi);
fwrite(&bmpHeader.sizeOfBitmapFile,4,1,outputi);
fwrite(&bmpHeader.reservedBytes,4,1,outputi);
fwrite(&bmpHeader.pixelDataOffset,4,1,outputi);

fwrite(&bmpInfoHeader.sizeOfThisHeader,4,1,outputi);
fwrite(&bmpInfoHeader.width,4,1,outputi);
fwrite(&bmpInfoHeader.height,4,1,outputi);
fwrite(&bmpInfoHeader.numberOfColorPlanes,2,1,outputi);
fwrite(&bmpInfoHeader.colorDepth,2,1,outputi);
fwrite(&bmpInfoHeader.compressionMethod,4,1,outputi);
fwrite(&bmpInfoHeader.rawBitmapDataSize,4,1,outputi);
fwrite(&bmpInfoHeader.horizontalResolution,4,1,outputi);
fwrite(&bmpInfoHeader.verticalResolution,4,1,outputi);
fwrite(&bmpInfoHeader.colorTableEntries,4,1,outputi);
fwrite(&bmpInfoHeader.importantColors,4,1,outputi);
```

We used a total of 256 colors, the maximum that could be used in a 8 byte image for saving it. Thus we will have to generate a color table for each of these 256 colors, i.e. occupying 256*4 = 1024 bytes of data.
We are generating the color table for a grayscale image so we mapped all the RGB values to the index. The alpha value is required to be set to zero for each color.

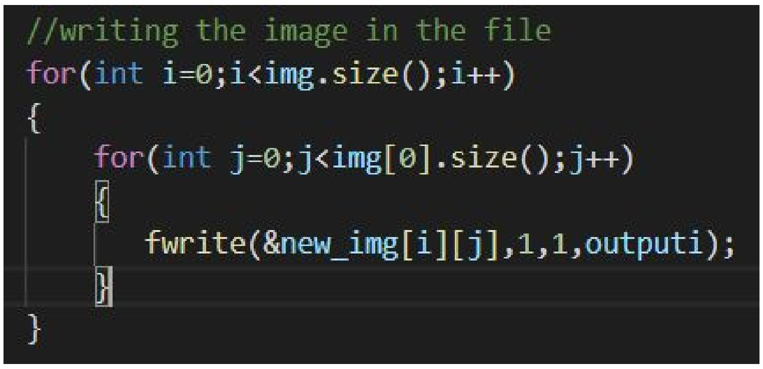


```
int zeroi = 0;
int changei = 1;

//creating the color table(BGRX) of the length of 4*number_of_color_used
//grayscale so b=g=r=i
//X/alpha = 0 always
for(int i=0;i<bmpHeader.pixelDataOffset-54;i++)
{
    changei = i/4;
    if(i%4==3)
        fwrite(&zeroi,1,1,outputi);
    else
    {
        fwrite(&changei,1,1,outputi);
    }
}
```

Finally we saved the image pixels in the file byte by byte using the below snippet. We first changed the int to char and then used it to save.



```
vector<vector<char>> new_img;
for(int i=0;i<img.size();i++)
{
    vector<char> temp;
    for(int j=0;j<img[0].size();j++)
    {
        temp.push_back((char)img[i][j][0]);
    }
    new_img.push_back(temp);
}
```



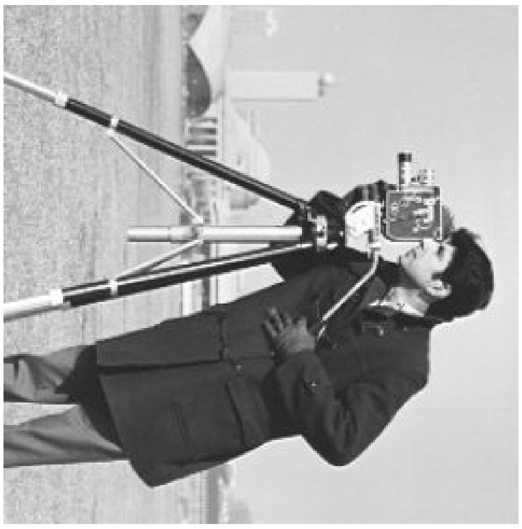
```
//writing the image in the file
for(int i=0;i<img.size();i++)
{
    for(int j=0;j<img[0].size();j++)
    {
        fwrite(&new_img[i][j],1,1,outputi);
    }
}
```

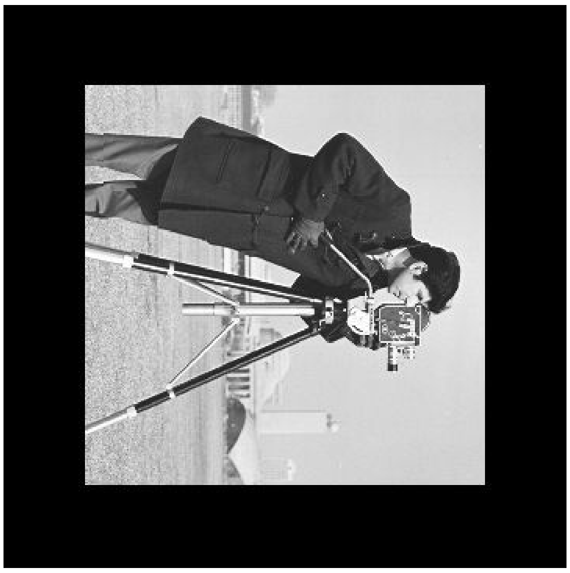
It is always required to close the FILE using fclose after the modifications are completed.

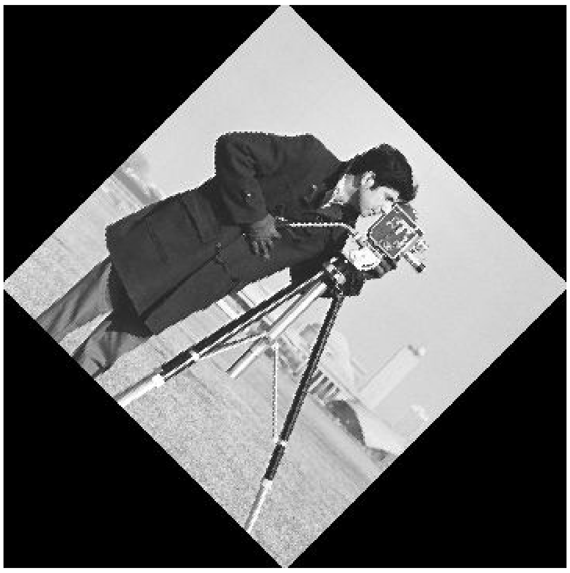
Results:

For Grayscale image:

Original Image


Diagonal Flipped

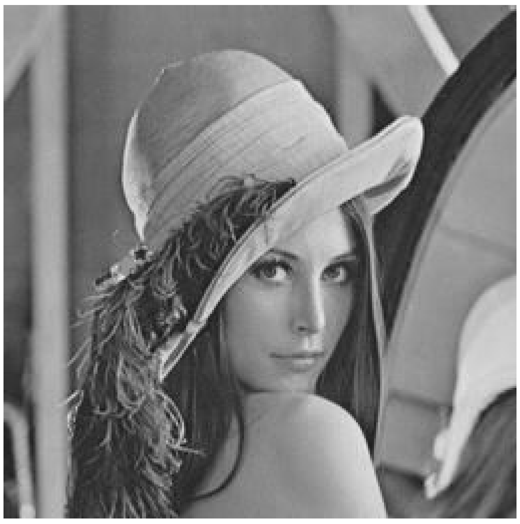

90 degree rotated

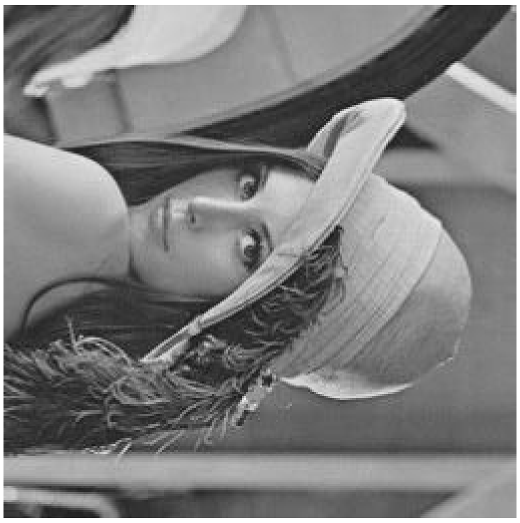

45 degree rotated

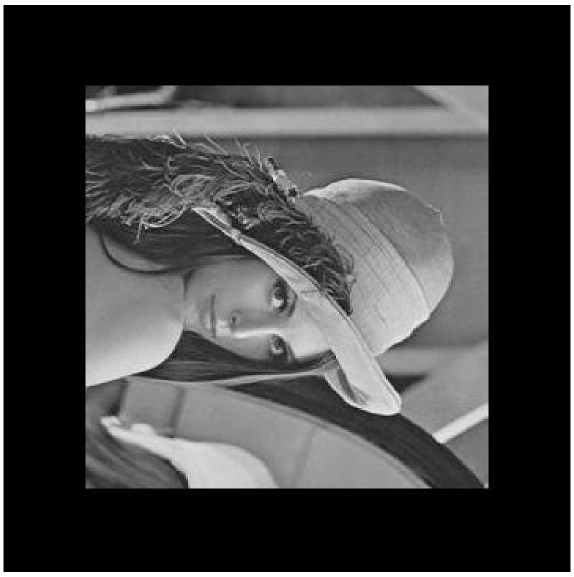
Scaled

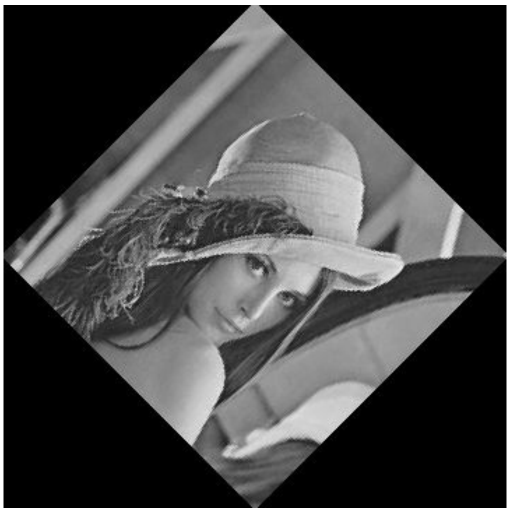
For Coloured Image:

Original Image


Grayscale converted


Diagonal Flipped


90-degree Rotated


45-degree Rotated

Scaled image

Analysis:

1. It is important to read the bmp images in binary mode for windows, however for unix systems, it does not matter.
2. The color table is important for images having less than or equal to 8 bits however for 24 bits, each pixel is represented by 24-bit red-green-blue (RGB) values in the actual bitmap data area.
3. It is important to note while reading the RGB files that R,G and B are stored opposite.
4. While scanning the images, if the width is not a multiple of 4, then the rows are padded. Similarly, it should be remembered while saving images.
5. Since the image origin is located at the bottom left corner, the diagonal flip function flipped the image about the sub diagonal and not about the main diagonal.
6. In BMP write function, width and height of the image should be in multiple of 4 in case of grayscale writing, so to ensure this we adjusted the width and height in the rotate_90 and rotate_45 function.
7. Due to above adjustment, when we rotate the image by 45 degrees, a small triangular patch gets clipped off from the output image.
8. Since we have used Nearest-neighbor interpolation in scaling, we observed certain blocky effects in the output, which can be improved using bilinear or bicubic interpolation.

References:

1. http://www.fysnet.net/bmpfile.htm
2. https://docs.fileformat.com/image/bmp/
3. https://www.codeproject.com/Articles/70442/C-RGB-to-Palette-Based-8-bit-Greyscale-Bitmap-Clas
4. https://dev.to/muiz6/c-how-to-write-a-bitmap-image-from-scratch-1k6m
5. http://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/2003_w/misc/bmp_file_format/bmp_file_format.htm