```
/* lab5.c
 * Christopher Brant
 * cbrant
 * ECE 2220, Fall 2016
 *
 * Purpose: Basic pixel transformations based on rotating 90 degrees
 *          or detecting edges in a file.
 *
 * Assumptions:
 *       The file must be 24-bit color, without compression, and without./
 *       a color map.
 *
 *       Some bmp files do not set the ImageSize field.  This code
 *       prints a warning but does not consider this an error since the
 *       size of the image can be calculated from other values.
 *
 * Command line argument
 *   command to detect edges or rotate, the input .bmp file,
 *   and the output .bmp file.
 *
 * Bugs:
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* WARNING: the header is 14 bytes, however on most systems
 * you will find that sizeof(struct Header) is 16 due to alignment
 * Thus trying to read the header with one fread may fail.  So,
 * read each member separately
 */
struct Header
{
    unsigned short int Type;                /* Magic identifier            */
    unsigned int Size;                      /* File size in bytes          */
    unsigned short int Reserved1, Reserved2;
    unsigned int Offset;                    /* Offset to image data, bytes */
};

struct InfoHeader
{
    unsigned int Size;               /* header size in bytes      */
    int Width, Height;               /* Width and height of image */
    unsigned short int Planes;       /* Number of colour planes   */
    unsigned short int Bits;         /* Bits per pixel            */
    unsigned int Compression;        /* Compression type          */
    unsigned int ImageSize;          /* Image size in bytes       */
    int xResolution,yResolution;     /* Pixels per meter          */
    unsigned int Colors;             /* Number of colors         */
    unsigned int ImportantColors;    /* Important colors         */
};

const char Matrix[3][3] =
{
    {  0, -1,  0 },
    { -1,  4, -1 },
    {  0, -1,  0 }
};

#define LINE 256

struct Pixel
{
    unsigned char Red, Green, Blue;
};

struct Pixel edge_trunc(struct Pixel **bitmap, int row, int col);
struct Pixel edge_mag(struct Pixel **bitmap, int row, int col);
void rotate_clock(struct Pixel **bitmap, struct Pixel **editbitmap, int row, int co
l);
void rotate_counter(struct Pixel **bitmap, struct Pixel **editbitmap, int row, int
col);
/*-------------------------------------------------------------*/

int main(int argc, char *argv[])
{
    char filein[LINE];
    char fileout[LINE];
    FILE *fpin;
    FILE *fpwrite;
    struct InfoHeader infoheader;
    struct Header header;
    int row, column;
    int pixel_cols, pixel_rows, pixel_count;
    int items_found;
    int rotateheight, rotatewidth;

    if (argc != 4)
    {
        printf("Incorrect number of inputs.\n");
        exit(1);
    }
    strcpy(filein, argv[2]);

    if ((fpin = fopen(filein, "rb")) == NULL)
    {
        printf("Cannot Open File. %s\n", filein);
        exit (1);
    }

    /* Read header */
    fread(&header.Type, sizeof(short int), 1, fpin);
    fread(&header.Size, sizeof(int), 1, fpin);
    fread(&header.Reserved1, sizeof(short int), 1, fpin);
    fread(&header.Reserved2, sizeof(short int), 1, fpin);
    fread(&header.Offset, sizeof(int), 1, fpin);

    if (header.Type != 0x4D42)
    {
        printf("This does not appear to be a bmp file: %s\n", filein);
        exit(1);
    }
    fread(&infoheader.Size, sizeof(int), 1, fpin);
    fread(&infoheader.Width, sizeof(int), 1, fpin);
    fread(&infoheader.Height, sizeof(int), 1, fpin);
    fread(&infoheader.Planes, sizeof(short int), 1, fpin);
    fread(&infoheader.Bits, sizeof(short int), 1, fpin);
    fread(&infoheader.Compression, sizeof(int), 1, fpin);
    fread(&infoheader.ImageSize, sizeof(int), 1, fpin);
    fread(&infoheader.xResolution, sizeof(int), 1, fpin);
    fread(&infoheader.yResolution, sizeof(int), 1, fpin);
    fread(&infoheader.Colors, sizeof(int), 1, fpin);
    fread(&infoheader.ImportantColors, sizeof(int), 1, fpin);
    pixel_rows = infoheader.Height;
    pixel_cols = infoheader.Width;
    pixel_count = 0;
    struct Pixel **editimage;
    struct Pixel **image = (struct Pixel**)malloc(pixel_rows*sizeof(struct Pixel*))
;

    // Mallocs for rotate functions
    if (strcmp(argv[1], "rotl") == 0 || strcmp(argv[1], "rotr") == 0)
    {
```

```
        editimage = (struct Pixel**)malloc(pixel_cols*sizeof(struct Pixel*));
        for (column = 0; column < pixel_cols; column++)
            editimage[column] = (struct Pixel*)malloc(pixel_rows*sizeof(struct Pixe
l));
        for (row = 0; row < pixel_rows; row++)
            image[row] = (struct Pixel*)malloc(pixel_cols*sizeof(struct Pixel));
    }

    // Mallocs for edge detection functions
    if (strcmp(argv[1], "edmag") == 0 || strcmp(argv[1], "edtrunc") == 0)
    {
        editimage = (struct Pixel**)malloc(pixel_rows*sizeof(struct Pixel*));
        for (row = 0; row < pixel_rows; row++)
        {
            image[row] = (struct Pixel*)malloc(pixel_cols*sizeof(struct Pixel));
            editimage[row] = (struct Pixel*)malloc(pixel_cols*sizeof(struct Pixel))
;
        }
    }

    // Read in the original image pixels
    for (row = 0; row < pixel_rows; row++)
    {
        for (column = 0; column < pixel_cols; column++)
        {
            items_found = fread(&image[row][column], 3, 1, fpin);
            if (items_found != 1)
            {
                printf("failed to read pixel %d at [%d][%d]\n",
                        pixel_count, row, column);
                exit(1);
            }
        }
    }
    fclose(fpin);

    // Edge detection using truncation
    if (strcmp(argv[1], "edtrunc") == 0)
    {
        for (row = 0, column = 0; row < pixel_rows; row++)
        {
            editimage[row][column] = image[row][column];
            editimage[row][pixel_cols - 1] = image[row][pixel_cols - 1];
        }

        for (column = 0, row = 0; column < pixel_cols; column++)
        {
            editimage[row][column] = image[row][column];
            editimage[pixel_rows - 1][column] = image[pixel_rows - 1][column];
        }

        for (row = 1; row < pixel_rows - 1; row++)
        {
            for (column = 1; column < pixel_cols - 1; column++)
                editimage[row][column] = edge_trunc(image, row, column);
        }
    }

    // Edge detection using magnitudes
    else if (strcmp(argv[1], "edmag") == 0)
    {
        for (row = 0, column = 0; row < pixel_rows; row++)
        {
            editimage[row][column] = image[row][column];
            editimage[row][pixel_cols - 1] = image[row][pixel_cols - 1];
        }

        for (column = 0, row = 0; column < pixel_cols; column++)
        {
            editimage[row][column] = image[row][column];
            editimage[pixel_rows - 1][column] = image[pixel_rows - 1][column];
        }

        for (row = 1; row < pixel_rows - 1; row++)
        {
            for (column = 1; column < pixel_cols - 1; column++)
                editimage[row][column] = edge_mag(image, row, column);
        }
    }

    // Rotation clockwise function
    else if (strcmp(argv[1], "rotr") == 0)
        rotate_clock(image, editimage, pixel_rows, pixel_cols);

    // Rotation counterclockwise function
    else if (strcmp(argv[1], "rotl") == 0)
        rotate_counter(image, editimage, pixel_rows, pixel_cols);

    // If no correct command, exit
    else
    {
        printf("Invalid command. Process eliminated\n");
        exit(1);
    }

    strcpy(fileout, argv[3]);

    // Open output file for binary writing
    if ((fpwrite = fopen(fileout, "wb")) == NULL)
    {
        printf("Cannot Open File. %s\n", fileout);
        exit (1);
    }


    // Write all header and info header information
    fwrite(&header.Type, sizeof(short int), 1, fpwrite);
    fwrite(&header.Size, sizeof(int), 1, fpwrite);
    fwrite(&header.Reserved1, sizeof(short int), 1, fpwrite);
    fwrite(&header.Reserved2, sizeof(short int), 1, fpwrite);
    fwrite(&header.Offset, sizeof(int), 1, fpwrite);
    fwrite(&infoheader.Size, sizeof(int), 1, fpwrite);

    // Conditional for the height and width for rotations
    if (strcmp(argv[1], "rotr") == 0 || strcmp(argv[1], "rotl") == 0)
    {
        rotateheight = infoheader.Width;
        rotatewidth = infoheader.Height;
        infoheader.Height = rotateheight;
        infoheader.Width = rotatewidth;
    }

    fwrite(&infoheader.Width, sizeof(int), 1, fpwrite);
    fwrite(&infoheader.Height, sizeof(int), 1, fpwrite);
    fwrite(&infoheader.Planes, sizeof(short int), 1, fpwrite);
    fwrite(&infoheader.Bits, sizeof(short int), 1, fpwrite);
    fwrite(&infoheader.Compression, sizeof(int), 1, fpwrite);
    fwrite(&infoheader.ImageSize, sizeof(int), 1, fpwrite);
    fwrite(&infoheader.xResolution, sizeof(int), 1, fpwrite);
    fwrite(&infoheader.yResolution, sizeof(int), 1, fpwrite);
    fwrite(&infoheader.Colors, sizeof(int), 1, fpwrite);
    fwrite(&infoheader.ImportantColors, sizeof(int), 1, fpwrite);

    // These conditionals with for loops write the edited pixels into the output fi
```

```
le
    if (strcmp(argv[1], "edtrunc") == 0 || strcmp(argv[1], "edmag") == 0)
    {
        for (row = 0; row < pixel_rows; row++)
        {
            for (column = 0; column < pixel_cols; column++)
                fwrite(&editimage[row][column], sizeof(struct Pixel), 1, fpwrite);
        }
        fclose(fpwrite);

        for (row = 0; row < pixel_rows; row++)
        {
            free(image[row]);
            free(editimage[row]);
        }
    }

    else
    {
        for (row = 0; row < pixel_cols; row++)
        {
            for (column = 0; column < pixel_rows; column++)
                fwrite(&editimage[row][column], sizeof(struct Pixel), 1, fpwrite);
        }
        fclose(fpwrite);

        for (row = 0; row < pixel_rows; row++)
            free(image[row]);

        for (column = 0; column < pixel_cols; column++)
        {
            free(editimage[column]);
        }
    }

    free(editimage);
    free(image);

    return 0;
}

// Edge detection with truncation function
struct Pixel edge_trunc(struct Pixel **bitmap, int row, int col)
{
    int i, Red, Green, Blue;
    struct Pixel new_pixel;
    int redSurround, greenSurround, blueSurround;
    redSurround = greenSurround = blueSurround = 0;

    for (i = -1; i < 2; i++)
    {
        if (i == 0)
            i = 1;

        redSurround += Matrix[1][i+1] * bitmap[row][col+i].Red;
        redSurround += Matrix[i+1][1] * bitmap[row+i][col].Red;
        greenSurround += Matrix[1][i+1] * bitmap[row][col+i].Green;
        greenSurround += Matrix[i+1][1] * bitmap[row+i][col].Green;
        blueSurround += Matrix[1][i+1] * bitmap[row][col+i].Blue;
        blueSurround += Matrix[i+1][1] * bitmap[row+i][col].Blue;
    }

    Red = (Matrix[1][1] * bitmap[row][col].Red) + redSurround;
    Green = (Matrix[1][1] * bitmap[row][col].Green) + greenSurround;
    Blue = (Matrix[1][1] * bitmap[row][col].Blue) + blueSurround;

    new_pixel.Red = Red;
    new_pixel.Green = Green;
    new_pixel.Blue = Blue;

    return new_pixel;
}

// Edge detection with magnitude function
struct Pixel edge_mag(struct Pixel **bitmap, int row, int col)
{
    int i, Red, Green, Blue;
    struct Pixel new_pixel;
    int redSurround, greenSurround, blueSurround;
    redSurround = greenSurround = blueSurround = 0;

    for (i = -1; i < 2; i++)
    {
        if (i == 0)
            i = 1;

        redSurround += Matrix[1][i+1] * bitmap[row][col+i].Red;
        redSurround += Matrix[i+1][1] * bitmap[row+i][col].Red;
        greenSurround += Matrix[1][i+1] * bitmap[row][col+i].Green;
        greenSurround += Matrix[i+1][1] * bitmap[row+i][col].Green;
        blueSurround += Matrix[1][i+1] * bitmap[row][col+i].Blue;
        blueSurround += Matrix[i+1][1] * bitmap[row+i][col].Blue;
    }

    Red = (Matrix[1][1] * bitmap[row][col].Red) + redSurround;
    Green = (Matrix[1][1] * bitmap[row][col].Green) + greenSurround;
    Blue = (Matrix[1][1] * bitmap[row][col].Blue) + blueSurround;

    Red = abs(Red);
    Green = abs(Green);
    Blue = abs(Blue);

    if (Red > 255)
        new_pixel.Red = 255;
    else
        new_pixel.Red = Red;
    if (Green > 255)
        new_pixel.Green = 255;
    else
        new_pixel.Green = Green;
    if (new_pixel.Blue > 255)
        Blue = 255;
    else
        new_pixel.Blue = Blue;

    return new_pixel;
}

// Rotation counterclockwise 90 degrees
void rotate_counter(struct Pixel **bitmap, struct Pixel **editbitmap, int row, int
col)
{
    int i, j;

    for (i = 0; i < col; i++)
    {
        for (j = 0; j < row; j++)
        {
            editbitmap[i][j] = bitmap[row-j-1][i];
        }
    }
}

// Rotation clockwise 90 degrees
```

```
void rotate_clock(struct Pixel **bitmap, struct Pixel **editbitmap, int row, int co
l)
{
    int i, j;

    for (i = 0; i < col; i++)
    {
        for (j = 0; j < row; j++)
        {
            editbitmap[i][j] = bitmap[j][col-i-1];
        }
    }
}
```