University of Nevada, Reno

CS 302 — Data Structures

Assignment 1

Students:
Joshua Gleason
Josiah Humphrey

Instructor:
Dr. George Bebis

February 16, 2010

Contents

1 Makefile 1
2 comp_curses.h 1
3 comp_curses.cpp 3
4 cubicSpline.h 9
5 cubicSpline.cpp 10
6 driver.cpp 14
7 image.h 47
8 image.cpp 48
9 imageIO.h 60
10 imageIO.cpp 60

1 Makefile

```
main.out: driver.o image.o cubicSpline.o imageIO.o comp_curses.o
	g++ -lncurses -g -o main.out driver.o image.o imageIO.o cubicSpline.o comp_curses.o

driver.o: driver.cpp image.h comp_curses.h cubicSpline.h imageIO.h
	g++ -c -lncurses -g driver.cpp

comp_curses.o: comp_curses.cpp comp_curses.h
	g++ -c -lncurses -g comp_curses.cpp

cubicSpline.o: cubicSpline.cpp cubicSpline.h
	g++ -c -g cubicSpline.cpp

imageIO.o: imageIO.h imageIO.cpp image.h image.cpp
	g++ -c -g imageIO.cpp image.cpp

image.o: image.h image.cpp
	g++ -c -g image.cpp

clean:
	rm *.o main.out

.PHONY: clean
```

2 comp_curses.h

```
/*****************************************************************************\
 Author: Joshua Gleason

 This header was created to compliment curses and allow user input in form of
 ints, doubles, and strings to be obtained.  A few other functions such as
```

```
 startCurses and setColor were defined here to make initialization and color
 changing easier to deal with.
\*****************************************************************************/

#ifndef COMP_CURSES
#define COMP_CURSES

#include <curses.h>
#include <cstring>
#include <cmath>

/*****************************************************************************\
                                    CONSTANTS
\*****************************************************************************/

        const int KEY_RETURN = 13;       // return key
        const int KEY_BS = 127;          // alternate backspace key value
        const int NUM_COLORS = 8;        // the number of colors in curses

        const int CURSOR_INVIS = 0; // the value indicating an invisible cursor
        const int CURSOR_VIS = 1;        // a visible cursor

        const int MAX_INT_LEN = 8;       // maximum allowed length of an integer
        const int MAX_DBL_LEN = 13; // maximum allowed length of a double

/*****************************************************************************\
                                FUNCTION PROTOTYPES
\*****************************************************************************/

        // name         : startCurses
        // input        : none
        // output       : starts curses and also sets all possible color pairs
        // assumptions : assumes curses hasn't been initialized
        void startCurses();

        // name         : endCurses
        // input        : none
        // output       : ends curses mode
        // assumptions : assumes that curses has been initialized
        void endCurses();

        // name         : setColor
        // input        : (optional WINDOW), foreground color and background colors
        // output       : changes the colors to be used at the given window
        // assumptions : if no WINDOW is passed then stdscr is assumed
        void setColor( WINDOW*, int, int );
        void setColor( int, int );

        // name         : screen<Width/Height>
        // input        : none
        // output       : returns the terminal width or height
        // assumptions : curses is started
        int screenWidth();
        int screenHeight();

        // name         : <hide/show>Cursor
```

```
        // input        : none
        // output       : set the cursor to visible or invisible
        // assumptions : curses is started
        void hideCursor();
        void showCursor();

        // name         : promptForInt
        // input        : a WINDOW, yLoc and xLoc and prompt string
        // output       : prompts user for an integer and returns that value
        // assumptions : curses is started and the prompt is a valid c string
        int promptForInt( WINDOW*, int, int, const char [] );

        // name         : promptForDouble
        // input        : a WINDOW, yLoc and xLoc and prompt string
        // output       : prompts user for a double and returns that value
        // assumptions : curses is started and the prompt is a valid c string
        double promptForDouble( WINDOW*, int, int, const char[] );

        // name         : promptForString
        // input        : a WINDOW, yLoc and xLoc, a prompt string, a string to store
        //                obtained value and the max length of the string
        // output       : prompt user for a string and return that value
        // assumptions : curses is started and the prompt is a valid c string
        void promptForString( WINDOW *, int, int, const char[], char[], int );

#endif
```

3 comp_curses.cpp

```
#include "comp_curses.h"

/*****************************************************************************\
 Start curses up with some useful settings, also set up all possible colors
\*****************************************************************************/
void startCurses()
{
        initscr();      // initialize curses
        cbreak();       // don't wait for enter between input
        noecho();       // don't echo characters typed to the window
        nonl();         // makes return key readable

        // initialize colors if possible
        if ( has_colors() )
        {
                // start colors mode
                start_color();

                // set all possible color pairs
                for ( int i = 0; i < NUM_COLORS; i++ ) // background
                        for ( int j = 0; j < NUM_COLORS; j++ ) // foreground
                                init_pair( j+(i*NUM_COLORS), j, i );
        }
}

/*****************************************************************************\
 Ends curses
```

```
\*****************************************************************************/
void endCurses()
{
        endwin();               // exit curses
}

/*****************************************************************************\
 Returns the width of the current terminal
\*****************************************************************************/
int screenWidth()
{
        return getmaxx( stdscr );
}

/*****************************************************************************\
 Returns the height of the current terminal
\*****************************************************************************/
int screenHeight()
{
        return getmaxy( stdscr );
}

/*****************************************************************************\
 Set the color in a certain window
\*****************************************************************************/
void setColor( WINDOW *somewin, int cf, int cb )
{
        // using the values created in startCurses set the color pair
        wattron( somewin, COLOR_PAIR( cf+cb*NUM_COLORS ) );
}

/*****************************************************************************\
 Set the color in the stdscr window
\*****************************************************************************/
void setColor( int cf, int cb )
{
        setColor( stdscr, cf, cb );
}

/*****************************************************************************\
 Hides the cursor from site until showCursor() is called
\*****************************************************************************/
void hideCursor()
{
        // set cursor to invisible
        curs_set( CURSOR_INVIS );
}

/*****************************************************************************\
 Make cursor visible to the user
\*****************************************************************************/
void showCursor()
{
        // set cursor to be visible
        curs_set( CURSOR_VIS );
}
```

```
/*****************************************************************************\
 Prompts the user for an integer value at the given location.  Only allow user
 to enter valid integers and return the value when the return key is pressed.
\*****************************************************************************/
int promptForInt( WINDOW *somewin, int y, int x, const char prompt[] )
{
        // holds the user input, current length of the string, and return value
        int input, length = 0, retVal = 0;

        // this is where all the users inputs are stored to be processed into an int
        // after the return key is pressed
        int intAry[MAX_INT_LEN];

        // set all the values in the array to 0
        for ( int i = 0; i < MAX_INT_LEN; i++ )
                intAry[i] = 0;

        // print the prompt
        mvwaddstr( somewin, y, x, prompt );

        // move the x value to the end
        x += strlen( prompt );

        // only echo integer values or negative sign
        do {
                input = wgetch( somewin );

                if ( input >= '0' && input <= '9' && length < MAX_INT_LEN )
                {
                        // if valid number is pressed
                        intAry[length] = input - (int)'0';
                        mvwaddch( somewin, y, x, input );
                        x++;
                        length++;
                }
                else if ( (input == KEY_BACKSPACE || input == KEY_BS) && length > 0 )
                {
                        // accounts for backspace
                        length--;
                        x--;
                        mvwaddch( somewin, y, x, ' ' );
                        intAry[length] = 0;

                        // actually move cursor back
                        move( y, x );
                }
                else if ( input == '-' && length == 0 )
                {
                        // if negative is pressed, only on first location
                        intAry[length] = 10;
                        mvwaddch( somewin, y, x, input );
                        x++;
                        length++;
                }
        // loop if return key is not pressed or
```

```
        //      if return is pressed but length is zero or
        //      if return is pressed but only negative sign exists
        } while ( input != KEY_RETURN ||
                ( input == KEY_RETURN && length == 0 ) ||
                        ( input == KEY_RETURN && length == 1 && intAry[0] == 10 ) );


        // process array into an integer
        for ( int i = length-1; i >= 0; i-- )
        {
                // process array into an integer
                if ( ( i == 0 && intAry[i] != 10 || i != 0 ) )
                        retVal += intAry[i] * pow(10,length-1-i);
        }

        // make it negative if nessessary
        if ( intAry[0] == 10 )
                retVal *= -1;

        // return integer value
        return retVal;
}

/*****************************************************************************\
 Prompts the user for a double value at the given location.  Only allow user
 to enter valid doubles and return the value when the return key is pressed.
\*****************************************************************************/
double promptForDouble( WINDOW *somewin, int y, int x, const char prompt[] )
{
        // user input, length of double, and decmial point location
        int input, length = 0, decimal = -1;

        // holds the user inputs for the double into an int array, to be processed
        // after return is pressed
        int intAry[MAX_DBL_LEN];

        // the return value
        double retVal = 0.0;

        // intitialize int array to zeros
        for ( int i = 0; i < MAX_DBL_LEN; i++ )
                intAry[i] = 0;

        // print the prompt
        mvwaddstr( somewin, y, x, prompt );

        // move the x value to the end
        x += strlen( prompt );

        // only echo integer values or negative sign
        do {
                input = wgetch( somewin );

                // if integer values are pressed thats okay
                if ( input >= '0' && input <= '9' && length < MAX_DBL_LEN )
                {
```

```
                        // if valid number is pressed
                        intAry[length] = input - (int)'0';
                        mvwaddch( somewin, y, x, input );
                        x++;
                        length++;
                }
                else if ( (input == KEY_BACKSPACE || input == KEY_BS) && length > 0 )
                {
                        // accounts for backspace
                        length--;
                        x--;
                        // reset decimal location if needed
                        if ( intAry[length] == 11 )
                                decimal = -1;

                        // move cursor back
                        mvwaddch( somewin, y, x, ' ' );
                        intAry[length] = 0;

                        // actually move cursor back
                        move( y, x );
                }
                else if ( input == '-' && length == 0 )
                {
                        // if negative is pressed, only on first location
                        intAry[length] = 10;
                        mvwaddch( somewin, y, x, input );
                        x++;
                        length++;
                }
                else if ( input == '.' && length < MAX_DBL_LEN-1 && decimal == -1 )
                {
                        // only allow a decimal point if one hasn't been placed
                        decimal = length;       // store location of decimal point
                        intAry[length] = 11;
                        mvwaddch( somewin, y, x, input );
                        x++;
                        length++;
                }
        // loop if return key is not pressed or
        //      if return is pressed but length is zero or
        //      if return is pressed but only negative sign or decimal exists
        //      if return is pressed but only a negative sign and decimal exists
        } while ( input != KEY_RETURN ||
                ( input == KEY_RETURN && length == 0 ) ||
                        ( input == KEY_RETURN && length == 1 && intAry[0] >= 10 ) ||
                        ( input == KEY_RETURN && length == 2 && intAry[0] >= 10 &&
                          intAry[1] >= 10 ) );

        // process into actual double value

        // if there is a decimal point
        if ( decimal >= 0 )
        {
                // if negative start at 1 otherwise start at 0
                for ( int i = (intAry[0]==10?1:0); i < decimal; i++ )
```

```
                        retVal += intAry[i] * pow(10,decimal-i-1);
                for ( int i = decimal+1; i < MAX_DBL_LEN; i++ )
                        retVal += intAry[i] / (double)pow(10,i-decimal);
        }

        // no decimal point is same as integer
        if ( decimal < 0 )
                for ( int i = length-1; i >= 0; i-- )
                {
                        if ( ( i == 0 && intAry[i] != 10 || i != 0 ) )
                                retVal += intAry[i] * pow(10,length-1-i);
                }

        // if negative return value
        if ( intAry[0] == 10 )
                retVal *= -1;

        // return the value as a double
        return retVal;
}

/*****************************************************************************\
 This is easiest of the prompts because it doesn't need any post input
 processing.  Simply prompt user for a string and return the string
\*****************************************************************************/
void promptForString( WINDOW *somewin, int y, int x, const char prompt[],
    char str[], int len )
{
        // user input and current length of string
        int input, length = 0;

        // print the prompt
        mvwaddstr( somewin, y, x, prompt );

        // move the x value to the end
        x += strlen( prompt );

        // only echo integer values or negative sign
        do {
                input = wgetch( somewin );

                // accounts for all valid string values
                if ( input >= ' ' && input <= '~' && length < len-1 )
                {
                        // if valid number is pressed
                        str[length] = input;
                        mvwaddch( somewin, y, x, input );
                        x++;
                        length++;
                }
                else if ( (input == KEY_BACKSPACE || input == KEY_BS) && length > 0 )
                {
                        // accounts for backspace
                        length--;
                        x--;
                        mvwaddch( somewin, y, x, ' ' );
```

```
                        move( y, x );
                }
        // loop if return key is not pressed or
        //      if return is pressed but length is zero or
        } while ( input != KEY_RETURN ||
                ( input == KEY_RETURN && length == 0 ) );

        // don't forget the null terminator
        str[length] = '\0';
}
```

4 cubicSpline.h

```
/*****************************************************************************\
 Author: Joshua Gleason

 This object is used to create cubic spline functions for the purpose of
 intermediate pixel approximation, this object is also capable of creating
 linear spline functions.

 Create a spline with...
 createCubic( int[], int ) passing an array of points and the array length
 or...
 create( int[], int ) the create the linear spline

 Obtain a value of the spline with...
 getCubicVal( double ) the spline is defined for all values of x but is defined
                       such that f(0)=firstPoint and f(100)=lastPoint
 or...
 getVal( double ) which returns the value of the linear spline at x, defined
                  such that f(0)=firstPoint and f(100)=lastPoint

 WARNING: the cubic spline must be created before calling getCubicVal as well
          as the linear spline must be defined before calling getVal, for
          example if you use createCubic then calling getVal will not give you
          the value of the linear spline(although getCubicVal will work).
\*****************************************************************************/

#ifndef CUBIC_SPL
#define CUBIC_SPL

class cubicSpline
{
public:
        // default constructor, intialize everything to 0 or NULL
        cubicSpline();

        // initialize the cubic spline using createCubic with the parameters
        cubicSpline( int[], int );

        // destructor de-allocates memory for all dynamically allocated memory
        ~cubicSpline();

        // builds the linear spline from a list of values
        void create( int[], int );
```

```
        // build the cubic spline from a list of values
        void createCubic(int[], int);

        // returns the value of the linear spline
        double getVal( double );

        // returns the value of cubic spline
        double getCubicVal( double );
private:

        // holds the coeffiencts for the linear splines
        double *coef_0;
        double *coef_1;

        // holds the values used to calculate the cubic splines
        double *a;
        double *y;

        int len;        // number of sub intervals for linear function
        int len2;       // number of sub intervals for cubic function
};

/* solves the matrix equation Ax=b for a tri-diagonal matrix, implementation
   for a better description, used in cubic spline function */
void solveTriDiag( double*, double*, double*, double*, double*, int );

#endif
```

5 cubicSpline.cpp

```
#include <iostream>
#include "cubicSpline.h"

/*****************************************************************************\
 default constructor, sets everything to zero or NULL
\*****************************************************************************/
cubicSpline::cubicSpline()
{
        // initialize all values to zero or NULL
        coef_0 = NULL;
        coef_1 = NULL;
        a = NULL;
        y = NULL;
        len = 0;
        len2 = 0;
}

/*****************************************************************************\
 paramaterized constructor, creates both a cubic spline and a linear spline
 based on the parameters
\*****************************************************************************/
cubicSpline::cubicSpline( int points[], int num )
{
        // initialize all the values to zero or NULL
        coef_0 = NULL;
        coef_1 = NULL;
```

```
        a = NULL;
        y = NULL;
        len = 0;
        len2 = 0;

        // creates both a cubic and linear spline based on points
        create( points, num );
        createCubic( points, num );
}

/*****************************************************************************\
 destructor makes sure all the memory is de-allocated before object is lost
\*****************************************************************************/
cubicSpline::~cubicSpline()
{
        if ( coef_0 != NULL )
                delete [] coef_0;
        if ( coef_1 != NULL )
                delete [] coef_1;
        if ( a != NULL )
                delete [] a;
        if ( y != NULL )
                delete [] y;
}

/*****************************************************************************\
 creates a CUBIC spline function for the given points, note that an equal
 distance between nodes is assumed, since this function is being used only
 for images this should be fine since pixels are evenly spaced

 NOTE: This creates a natural cubic spline
\*****************************************************************************/
void cubicSpline::createCubic( int points[], int num )
{
        // defines the step size
        double h = 100.0 / (num-1);

        // allocate memory for the variables if required
        if ( len2 != num-2 )
        {
                len2 = num-2;
                if ( a != NULL )
                        delete [] a;
                if ( y != NULL )
                        delete [] y;

                a = new double[num];
                y = new double[num];
        }

        // the vectors that will be used to calculate the tri-diagonal matrix
        double *diag_b = new double[len2];
        double *diag_d = new double[len2];
        double *diag_a = new double[len2];
        double *B = new double[len2];
```

```
        // set up the vectors based on the Lagrange spline method
        for ( int i = 0; i < len2; i++ )
        {
                diag_b[i] = h;
                diag_d[i] = 4*h;
                diag_a[i] = h;
                B[i] = 6/h*((points[i+2]-points[i+1])-(points[i+1]-points[i]));

                // y simply holds a copy of the points
                y[i] = points[i];
        }

        // get the last 2 y points since the previous loop went 2 to short
        y[len2] = points[len2];
        y[len2+1] = points[len2+1];

        // set the begining and end of a to zero like it should be
        a[0] = a[len2+1] = 0;

        // passing a+1 to solve for a[1]->a[len] since a[0] is already defines as 0
        solveTriDiag( diag_b, diag_d, diag_a, B, a+1, len2 );

        // now that a has been solved these are no longer required
        delete [] diag_a;
        delete [] diag_b;
        delete [] diag_d;
        delete [] B;

}

/*****************************************************************************\
 this creates a LINEAR spline function for the given points from 0-100
\*****************************************************************************/
void cubicSpline::create( int points[], int num )
{
        // define the step size
        double stepsize = 100.0 / (num-1);
        double x1, y1, x2, y2;

        // allocate memory if needed
        if ( len != num-1 )
        {
                if ( coef_0 != NULL )
                        delete [] coef_0;
                if ( coef_1 != NULL )
                        delete [] coef_1;

                len = num-1;
                coef_0 = new double[len];
                coef_1 = new double[len];
        }

        // set up the splines based on linear spline definition
        for ( int i = 0; i < len; i++ )
        {
                x1 = stepsize * i;
```

```
                y1 = points[i];
                x2 = stepsize * (i+1);
                y2 = points[i+1];

                // with x1, x2, y1 and y2 we can define the coefficients
                coef_0[i] = -x2*y1/(x1-x2) - x1*y2/(x2-x1);
                coef_1[i] = y1/(x1-x2) + y2/(x2-x1);
        }
}

/*****************************************************************************\
 depending on the value of x return a value using the linear spline coef.
\*****************************************************************************/
double cubicSpline::getVal( double x )
{
        if ( len == 0 ) return 0; // protects from divide by 0

        // define the step size again so we can use it in our calculations
        double stepsize = 100.0 / len;

        // the index is simply x / stepsize rounded down
        int index = (int)(x / stepsize);

        // if x is greater than 100 or less than 0 just use the closest line
        if ( index >= len )
                index = len-1;
        if ( index < 0 )
                index = 0;

        return coef_1[index] * x + coef_0[index];
}

/*****************************************************************************\
 Return the value of the cubic spline function at the given x value, the spline
 is defined from 0 to 100, so x should be around there, if not x will be the
 nearest spline value
\*****************************************************************************/
double cubicSpline::getCubicVal( double x )
{
        if ( len2 == 0 ) return 0;  // protects against divide by zero

        // define the step size to be used in the calculation
        double h = 100.0 / (len2+1);

        // define the index of a to be used
        int i = (int)(x/h);

        // if x is greater than 100 or less than 0 then just use the closest curve
        if ( i >= len2+1 )
                i = len2;
        if ( i < 0 )
                i = 0;

        // using the definition of Lagrange cubic interpolation
        if ( len2 != 0 )
                return (a[i]/(6*h)*((i+1)*h-x)*((i+1)*h-x)*((i+1)*h-x) +
```

```
                                        a[i+1]/(6*h)*(x-(i*h))*(x-(i*h))*(x-(i*h)) +
                                        (y[i]/h - a[i]*h/6)*((i+1)*h-x) +
                                        (y[i+1]/h-a[i+1]*h/6)*(x-i*h));
        else
                return 0;
}

/*****************************************************************************\
Solves the following tri-diagonal matrix Ax=b shown below
   [ d_0 a_0 0   0   0   ... 0     0     0     0    ]   [ x_0 ]   [ B_0 ]
   [ b_1 d_1 a_1 0   0   ... 0     0     0     0    ]   [ x_1 ]   [ B_1 ]
   [ 0   b_2 d_2 a_2 0   ... 0     0     0     0    ]   [ x_2 ]   [ B_2 ]
   [ 0   0   b_3 d_3 a_3 ... 0     0     0     0    ]   [ x_3 ]   [ B_3 ]
   [ :   :   :   :   :   ... :     :     :     :    ] * [  :  ] = [  :  ]
   [ :   :   :   :   :   ... :     :     :     :    ]   [  :  ]   [  :  ]
   [ 0   0   0   0       ... b_n-2 d_n-2 a_n-2 0    ]   [x_n-2]   [B_n-2]
   [ 0   0   0   0   0   ... 0     b_n-1 d_n-1 a_n-1]   [x_n-1]   [B_n-1]

This function assumes the matrix is not singular and changes a and B
but we don't really care in this case ^_^
\*****************************************************************************/
void solveTriDiag( double *b, double *d, double *a, double *B, double *x, int n )
{
        // step 1 is to define the vectors which is already done
        // step 2 calculate new a[0] and B[0];
        a[0] = a[0] / d[0];
        B[0] = B[0] / d[0];

        // step 3 for 1 < i < n calculate new a[i] and B[i]
        for ( int i = 1; i < n; i++ )
        {
                a[i] = a[i] / (d[i]-b[i]*a[i-1]);
                B[i] = (B[i] - B[i-1] * b[i]) / (d[i]-b[i]*a[i-1]);
        }

        // step 4 was to calculate B[n-2] but it was done in the prev loop
        // step 5 calculate x starting at x[n-1] and going to x[0]
        x[n-1] = B[n-1];
        for ( int i = n - 2; i >= 0; i-- )
                x[i] = B[i] - a[i] * x[i+1];
}
```

6 driver.cpp

```
/*****************************************************************************\
 Authors: Josiah Humphrey and Joshua Gleason

 Date Due For Review  : 02/16/2010

        This program is designed to be a driver for the ImageType objects.  The
 user interface attempts to allow the objects to be throughly tested in a
 robust, simple environment.

        The ImageType object (defined in image.h) is for manipulating grayscale
 images, it allows the user to easly enlarge, rotate, negate, etc... an image.
 The functions in imageIO.h are used to load and save images of type .pgm.
```

```
        We choose to use curses library to make a more visually pleasing main
 driver, it implements our scrolling menu system.

        The dirent.h library is used to scan for files in the appropriate location,
 in our case we only list .pgm files located in the local images folder.  This
 is better understood by examining the findLocalPGM function.

        comp_curses.h was written to make initializing curses easier, it also has
 some functions for obtaining user input as integers, doubles, and strings.
 Many ncurses library functions however are used directly in this program.
\*****************************************************************************/

#include <string>
#include <cstdlib>
#include <cstdio>
#include <dirent.h>
#include <cstring>
#include "comp_curses.h"
#include "imageIO.h"

#include "image.h"

using namespace std;

/*****************************************************************************\
                                  CONSTANTS
\*****************************************************************************/
        // sets cubic or linear interpolation for enlarge
        const bool CUBIC_INTER = true;

        // the folder with the images in it, (make it ./ for local)
        const char IMAGELOC[] = "./images/";

        const int REGS = 5;                                 // values 1-9
        const int MENU_OPTIONS = 16;     // number of main menu choices
        const int BAD_REG = REGS;                  // dont change this
        const int NAME_LEN = 50;                   // the max string length of names

        const int MSGBOX_WIDTH = 60;     // message box width (also input box)
        const int MSGBOX_HEIGHT = 4;     // message box height

        const int MENU_WIDTH = 40;                 // holds the menu width and height
        const int MENU_HEIGHT = MENU_OPTIONS*2+3;

        const int REGWIN_WIDTH = 36;     // holds the register window width
        const int REGWIN_HEIGHT = REGS*2+5;

        const int FILEWIN_WIDTH = 36;    // file window width and height
        const int FILEWIN_HEIGHT = MENU_HEIGHT-REGWIN_HEIGHT-1;

        const int MAX_IMG = 10000;                 // the max size you can enlarge to
        const int MIN_IMG = 4;                     // the min size you can reduce to

        const short BG_COLOR = COLOR_BLUE;         // doesnt matter
        const short FG_COLOR = COLOR_BLACK;        // background color
```

```
        const short MENU_BACKGROUND = COLOR_CYAN;       // window backgrounds
        const short MENU_FOREGROUND = COLOR_BLACK;      // window foregrounds


/*****************************************************************************\
                              FUNCTION PROTOTYPES
\*****************************************************************************/
        // name         : showMenu
        // input        : an un-initialized window pointer, a string to be the title,
        //                height, width, xLoc, yLoc of the window, list of c-style
        //                strings to be used in the menu, the number of menu options,
        //                and a bool value which says weather the last choice is
        //                left highlighted
        // output       : Display a window with menu options, let user choose and
        //                return the index of that choice
        // assumptions  : assumes that window is un-intialized and will be destructed
        //                by calling function
        int showMenu( WINDOW *&, const char[], int, int, int, int,
            char[][NAME_LEN], int, bool=false );

        // name         : showMenu
        // input        : same as above function except with dynamic string list
        // output       : same as above function
        // assumptions  : assumes the same about window as above, also assums the
        //                dynamic list of strings has been initialized to at least
        //                the number of window options.  The list of strings is not
        //                de-allocated by this function
        int showMenu( WINDOW *&, const char[], int, int, int, int,
            char*[], int, bool=false );

        // name         : showRegs
        // input        : an un-initialized window pointer, a title string, and a
        //                list of register names
        // output       : displays a window next to main of all the registers
        // assumptions  : allocates but doesn't delete the WINDOW object
        void showRegs( WINDOW *&, const bool[], const char[][NAME_LEN] );

        // name         : drawWindow
        // input        : an un-intialized window pointer, a title string, and then
        //                the window height, width, yLoc, and xLoc, plus the colors
        //                for the background and foreground which are defaulted
        // output       : displays a empty window with a border and title using the
        //                given parameters
        // assumptions  : allocates but doesn't delete the WINDOW object
        void drawWindow( WINDOW *&, const char[], int, int, int, int,
            short=MENU_BACKGROUND, short=MENU_FOREGROUND );

        // name         : deleteMenu
        // input        : a WINDOW pointer that is allocated
        // output       : de-allocate memory for the window pointer and refresh the
        //                main screen
        // assumptions  : assumes WINDOW object is intialized before calling
        void deleteMenu( WINDOW *& );

        // name         : processEntry
```

```
// input       : List of register images, list of register bools, list of
//               register names, and a value assumed to be choosen by user
// output      : depending on the value, call a function to do some image
//               manipulation
// assumptions : assumes value >= 0 and < MENU_OPTIONS, not that anything
//               will crash if its not true, but nothing will happen, also
//               assumes that names contain valid c strings
void processEntry( ImageType[], bool[], char[][NAME_LEN], int );

// name        : stdWindow
// input       : an un-initalized window and a title string
// output      : displays a window in the standard text box location with
//               the title and a border
// assumptions : the window object is initalized here but not deleted, this
//               is left up to the calling function
void stdWindow( WINDOW *&, const char[] );

// name        : promptForReg
// input       : list of register bools, list of regist names, a flag that
//               indicates if registers that have not been loaded can be
//               choosen, the yLoc, and xLoc of the menu
// output      : Display a menu with the registers in it, allowing user to
//               choose a register
// assumptions : assumes that names are already set to valid c strings
int promptForReg( bool[], char[][NAME_LEN], const bool = true,
    int=1, int=MENU_WIDTH+3 );

// name        : promptForFilename
// input       : title string, prompt string and char used to store user
//               input
// output      : sets the final parameter equal to the filename the user
//               chooses and returns the length
// assumptions : assumes first 2 parameters are valid c strings and that
//               the final parameter is a string of at least length 16 +
//               the length of the file path declared as a constant
int promptForFilename( const char[], const char[], char[] );

// name        : promptForLoc
// input       : prompt string, image object, and 2 integers passed by ref
// output      : sets two reference parameters equal to row and column of
//               users choice
// assumptions : assumes image is intialized and has a valid height/width
//               also that first parameter is a valid c string
void promptForLoc( const char[], ImageType&, int&, int& );

// name        : promptFor<Pix/Scale>Value
// input       : title string, prompt string and max input value
// output      : prompts user in message window and returns the value when
//               the user inputs a valid value.  -1 indicates cancel
// assumptions : for Pix the minimum value is 0 and for Scale its 2, thats
//               the only difference.  Also assumes that first 2 parameters
//               are valid c strings
int promptForPixValue( const char[], const char[], int );
int promptForScaleValue( const char[], const char[], int );

// name        : promptForMirror
```

```
// input        : title string, prompt string
// output       : prompts user for a H, V, or C and doesn't let them cont
//                until one is choosen, then returns input value to calling
//                function
// assumptions : both parameters are valid c strings
char promptForMirror( const char[], const char[] );

// name         : promptForAngle
// input        : title string, prompt string
// output       : prompts user for an angle 0-360 and returns the value when
//                a valid number is sent. -1 indicates cancel
// assumptions : both parameters are valid c strings
int promptForAngle( const char[], const char[] );

// name         : messageBox
// input        : title string and message string
// output       : displays a message box in the center of the screen with
//                the message displayed in it. Waits for user to press
//                return before continueing
// assumptions : assumes both parameters are valid c strings
void messageBox( const char[], const char[] );

// name         : fillRegs
// input        : list of images, bools, and c strings all of length REGS
//                also the number of arguments passed to main and the array
//                of strings passed to main
// output       : sets valid arguments to registers (loading images) and
//                clears the rest of the registers
// assumptions : assumes that char** is a valid list of strings with int
//                rows
void fillRegs( ImageType[], bool[], char[][NAME_LEN], int, char** );

// name         : Register manipulation functions
// input        : List of images of length REGS, list of bools of length
//                REGS, and list of c strings of length REGS.
// output       : Each function prompts user for information pertaining
//                to its manipulation function, these should be pretty
//                obvious looking at each functions name. All input is
//                bounds checked to make sure no bad input is passed to an
//                ImageType object
// assumptions : assumes all names in the c string list are valid c
//                c strings and bools coincide with wether image types are
//                loaded of the same index
void clearRegister( ImageType[], bool[], char[][NAME_LEN] );
void loadImage( ImageType[], bool[], char[][NAME_LEN] );
void saveImage( ImageType[], bool[], char[][NAME_LEN] );
void getImageInfo( ImageType[], bool[], char[][NAME_LEN] );
void setPixel( ImageType[], bool[], char[][NAME_LEN] );
void getPixel( ImageType[], bool[], char[][NAME_LEN] );
void extractSub( ImageType[], bool[], char[][NAME_LEN] );
void enlargeImg( ImageType[], bool[], char[][NAME_LEN] );
void shrinkImg( ImageType[], bool[], char[][NAME_LEN] );
void reflectImg( ImageType[], bool[], char[][NAME_LEN] );
void translateImg( ImageType[], bool[], char[][NAME_LEN] );
void rotateImg( ImageType[], bool[], char[][NAME_LEN] );
void sumImg( ImageType[], bool[], char[][NAME_LEN] );
```

```
        void subtractImg( ImageType[], bool[], char[][NAME_LEN] );
        void negateImg( ImageType[], bool[], char[][NAME_LEN] );

        // name         : findLocalPGM
        // input        : one un-intialized double pointer of chars
        // output       : allocates enough memory for a list of all the .pgm files
        //                in the local path specified by the FILELOC constant.  It
        //                then copys the file names to the array and returns the
        //                number of rows in the array.
        // assumptions : filenames is not initialized, but will be in the function
        //                this means it needs to be de-allocated before it goes out
        //                of scope
        int findLocalPGM( char **&filenames );

/*****************************************************************************\
                                    MAIN
\*****************************************************************************/

int main( int argc, char **argv )
{
        // main menu object pointer
        WINDOW *menu;

        // register window object pointer
        WINDOW *regWin;

        // users menu choice
        int choice;

        // holds the name of the image stored in the register
        char imgName[REGS][NAME_LEN];

        // bool array indicating if registers are loaded with images
        bool imgLoaded[REGS];

        // this is where all the registers are stored
        ImageType image[REGS];

        // create main menu strings
        char choices[MENU_OPTIONS][NAME_LEN] = {
                "  Read an image from a file",
                "  Save an image to a file",
                "  Get image info",
                "  Set the value of a pixel",
                "  Get the value of a pixel",
                "  Extract a subimage from an image",
                "  Enlarge image",
                "  Shrink image",
                "  Reflect image",
                "  Translate image",
                "  Rotate image",
                "  Sum two images",
                "  Subtract two images",
                "  Compute negative of an image",
                "  Clear a register",
                "  Exit" };
```

```
// start
startCurses();

// hide that pesky cursor
hideCursor();

// initialize the bool array
for ( int i = 0; i < REGS; i++ )
        imgLoaded[i] = false;

// read argument parameters
fillRegs( image, imgLoaded, imgName, argc, argv );

// set the colors
setColor( FG_COLOR, BG_COLOR );

// clear the screen
for ( int i = 0; i < screenWidth(); i++ )
        for ( int j = 0; j < screenHeight(); j++ )
                mvaddch(j, i, ' ');
refresh();

do {
        // display register window
        showRegs( regWin, imgLoaded, imgName );

        // show and get input from menu
        choice = showMenu( menu, "Main Menu", MENU_HEIGHT, MENU_WIDTH, 1, 1,
            choices, MENU_OPTIONS );

        try
        {
                // this is the main driving function that calls all others
                processEntry( image, imgLoaded, imgName, choice );
        }
        catch( string err )
        {
                // display the message string to user
                messageBox( "Error!", err.c_str() );
        }
        catch( ... )
        {
                // handle errors like this later
                return -1;
        }

        // makes sure everything is reset
        delwin( regWin );
        deleteMenu( menu );
} while ( choice != MENU_OPTIONS-1 );

// end curses
endCurses();

return 0;
```

```
}

/*****************************************************************************\
|                                FUNCTION IMPLEMENTATION                      |
\*****************************************************************************/

/*****************************************************************************\
 Overloaded showMenu that accepts static arrays of strings (length NAME_LEN)
 and then creates a dynamic version and passes it to the other showMenu,
 afterwards the memory is de-allocated
\*****************************************************************************/
int showMenu( WINDOW *& menu, const char title[], int height, int width,
    int locY, int locX, char menuStr[][NAME_LEN], int choices, bool erase )
{
        // holds the new dynamic array of strings
        char **menuStrPtr = new char*[choices];

        // will hold the return value from other menu call
        int retVal;

        // count through the static array, allocating memory for the dynamic one
        for ( int i = 0; i < choices; i++ )
        {
                menuStrPtr[i] = new char[NAME_LEN];

                // this is where the string value is copied
                strcpy( menuStrPtr[i], menuStr[i] );
        }

        // call the other menu function
        retVal = showMenu( menu, title, height, width, locY, locX,
            menuStrPtr, choices, erase );

        // de-allocate memory for temporary string array
        for ( int i = 0; i < choices; i++ )
                delete [] menuStrPtr[i];
        delete [] menuStrPtr;

        // return value obtained by other menu
        return retVal;
}

/*****************************************************************************\
 This is the function which builds the scrolling menu system, this simply
 creates a curses window and puts all the options stores in menuStr onto the
 window, it then waits for the user to press UP, DOWN, or RETURN before
 reacting.  The parameters allow menus to be different widths, heights, and
 locations.  A few constants can be changed to change the colors of the window.

 !!!Beware the WINDOW pointer 'menu' is intialized here but is NOT deleted.
 It is up to the calling function to take care of this object.
\*****************************************************************************/
int showMenu( WINDOW *& menu, const char title[], int height, int width,
    int locY, int locX, char *menuStr[], int choices, bool eraseHighlight )
{
        /* x, y variables hold location of cursor, choiceLoc is the current choice
```

```
    thats selected, menuLoc is the location on the menu, and input is the
    value of key input */
int x, y, choiceLoc, menuLoc, perScreen, input;

// formatStr is make sure strings are length 40 no matter what
char formatStr[10] = "%-";

// draw the window (intializing the WINDOW object as well)
drawWindow( menu, title, height, width, locY, locX );

// finish creating the format string
sprintf( formatStr, "%s%i.%is", formatStr, width-4, width-4 );

// initialize int values
x = 2, y = 2;
choiceLoc = input = menuLoc = 0;

// this is number of options displayed at a time
perScreen = (height - 3 ) / 2;

// turns keypad on for menu (for arrow keys)
keypad( menu, TRUE );

// loop for user input
do {

        // draw the visible menu options
        for ( int i = 0; i < perScreen && i < choices; i++ )
        {
                y = 2+i*2;
                mvwprintw( menu, y, x, formatStr, menuStr[choiceLoc-menuLoc+i] );
        }

        // set mode to highlight
        setColor( menu, MENU_BACKGROUND, MENU_FOREGROUND );
        y = 2 + menuLoc*2;

        // highlight current selection
        mvwprintw( menu, y, x, formatStr, menuStr[choiceLoc] );
        wrefresh( menu );

        // only react to KEY_UP, KEY_DOWN or KEY_RETURN
        do {
                input = wgetch( menu );
        } while ( input != KEY_UP &&
                  input != KEY_DOWN &&
                  input != KEY_RETURN );

        // dont highlight for now
        setColor( menu, MENU_FOREGROUND, MENU_BACKGROUND );

        // unhighlight current option
        if ( eraseHighlight )
                mvwprintw( menu, y, x, formatStr, menuStr[choiceLoc] );

        // redrawn menu with highlight off in case this is final loop
```

```
                wrefresh( menu );

                // test input value
                switch ( input )
                {
                        case KEY_UP:     // move up menu

                                // choice needs to roll to bottom
                                if ( --choiceLoc < 0 )
                                {
                                        choiceLoc = choices-1;

                                        // set menuLoc to bottom of screen or list
                                        if ( choices < perScreen )
                                                menuLoc = choiceLoc;
                                        else
                                                menuLoc = perScreen-1;
                                }
                                else if ( --menuLoc < 0 )
                                        // only decrement menuLoc if its not 0
                                        menuLoc++;
                                break;
                        case KEY_DOWN:   // move down menu

                                // choice needs to roll to top
                                if ( ++choiceLoc > choices-1 )
                                {
                                        choiceLoc = 0;
                                        menuLoc = 0;
                                }
                                else if ( ++menuLoc > perScreen-1 )
                                        // onlt increment menu if its not at the bottom
                                        menuLoc--;
                                break;
                }
        } while ( input != KEY_RETURN );

        // return the menu choice
        return choiceLoc;
}

/*****************************************************************************\
 Display a window of registers next to the main menu (or wherever the constants
 dictate)

 !!! Beware the WINDOW pointer 'regWin' is intialized here but is NOT deleted.
 It is up to the calling function to take care of this object.
\*****************************************************************************/
void showRegs( WINDOW *& regWin, const bool loaded[],
    const char names[][NAME_LEN] )
{
        // draw/initialize the window to display the
        drawWindow( regWin, "Registers", REGWIN_HEIGHT, REGWIN_WIDTH, 1,
            MENU_WIDTH+3 );

        // add the register names to the window
```

```
        for ( int i = 0; i < REGS; i++ )
                mvwprintw( regWin, i*2+2, 2, "%-32.32s", names[i] );

        // make sure the new characters are printed!
        wrefresh( regWin );
}

/*****************************************************************************\
 This basically clears the entire screen after deleting the window that is
 passed.
\*****************************************************************************/
void deleteMenu( WINDOW *& menu )
{
        // delete the menu object
        delwin( menu );

        // touch the screen to make sure it knows things have changed then refresh
        touchwin( stdscr );
        refresh();
}

/*****************************************************************************\
 This function simply draws an empty window with a given title, height, width,
 x, and y locations.  The colors have default values but can be changed if
 oddly colored windows are wanted.
\*****************************************************************************/
void drawWindow( WINDOW *& win, const char title[], int height, int width,
    int y, int x, short bgColor, short fgColor )
{
        // intialize window
        win = newwin( height, width, y, x );

        // set new colors
        setColor( win, fgColor, bgColor );

        // make text bold (brighter)
        wattron( win, A_BOLD );

        // draw border
        box( win, 0, 0 );

        // fill window with spaces
        for ( int i = 1; i < width-1; i++ )
                for ( int j = 1; j < height-1; j++ )
                        mvwaddch( win, j, i, ' ' );

        // add title
        mvwaddstr( win, 0, 2, title );
}

/*****************************************************************************\
 This is the function that decides where to go depending on the choice in the
 main menu.  The reason it has all the parameters is for passing to the
 subsequent functions that will be using them.
\*****************************************************************************/
void processEntry( ImageType img[], bool loaded[], char name[][NAME_LEN],
```

```
    int choice )
{
        // enter switch statement evaluating choice
        switch ( choice )
        {
                case 0: // read image
                        loadImage( img, loaded, name );
                        break;
                case 1: // write image
                        saveImage( img, loaded, name );
                        break;
                case 2: // image info
                        getImageInfo( img, loaded, name );
                        break;
                case 3: // set pixel val
                        setPixel( img, loaded, name );
                        break;
                case 4: // get pixel val
                        getPixel( img, loaded, name );
                        break;
                case 5: // extract subimage
                        extractSub( img, loaded, name );
                        break;
                case 6: // enlarge
                        enlargeImg( img, loaded, name );
                        break;
                case 7: // shrink
                        shrinkImg( img, loaded, name );
                        break;
                case 8: // reflect
                        reflectImg( img, loaded, name );
                        break;
                case 9: // translate
                        translateImg( img, loaded, name );
                        break;
                case 10:        // rotate
                        rotateImg( img, loaded, name );
                        break;
                case 11:        // sum
                       sumImg( img, loaded, name );
                        break;
                case 12:        // difference
                        subtractImg( img, loaded, name );
                        break;
                case 13:        // negative
                        negateImg( img, loaded, name );
                        break;
                case 14:        // clear register
                        clearRegister( img, loaded, name );
                        break;
                case 15:        // exit
                        // do nothing lol ^_^ maybe later add an exit screen
                        break;
        }
}
```

```
/*****************************************************************************\
 Prompt for a register that is filled and then clear it.
\*****************************************************************************/
void clearRegister( ImageType img[], bool loaded[], char name[][NAME_LEN] )
{
        int index;

        // prompt for register to reset
        index = promptForReg( loaded, name );

        // if back wasn't selected...
        if ( index != BAD_REG )
        {
                // set the register to empty
                loaded[index] = false;

                // reset the register name
                sprintf( name[index], "Register %i: Empty", index+1 );
        }
}

/*****************************************************************************\
 This is the function that reads the arguments passed to main by the terminal.
 It compiles an error message for every read/write exception that throws a
 string object.  It stores the values into the registers sequentially, if there
 are no arguments relating to the register it is set to empty
\*****************************************************************************/
void fillRegs( ImageType img[], bool loaded[], char name[][NAME_LEN], int argc,
    char **argv )
{
        char *msg = new char[1+(argc-1)*40];
        int i, j, k;
        bool f;

        // initalize string
        msg[0] = '\0';

        // set empty values
        for ( int index = 0; index < REGS; index++ )
        {
                sprintf( name[index], "Register %i: Empty", index+1 );
                loaded[index] = false;
        }

        // read arguments from main and attempt to load registers
        for ( int index = 1; index < argc && index <= REGS; index++ )
        {
                try
                {
                        // read image header
                        readImageHeader( argv[index], i, j, k, f );

                        // set image info to header value
                        img[index-1].setImageInfo( i, j, k );

                        // read the rest of the image
```

```
                        readImage( argv[index], img[index-1] );

                        // set the name of the register to the file path
                        sprintf( name[index-1], "Register %i: %s", index, argv[index] );

                        // make sure the program knows the register is in use
                        loaded[index-1] = true;
                }
                catch ( string s )
                {
                        // for every exception string caught compile a list of errors
                        strcat( msg, s.c_str() );
                        strcat( msg, "\n" );
                }
        }

        // display the message to the back screen (will be behind any windows)
        if ( strlen(msg) > 0 )
        {
                // just white on black
                setColor( COLOR_WHITE, COLOR_BLACK );

                // display the message
                mvaddstr( 0, 0, msg );

                // wait for input
                getch();

                // clear screen
                for ( int x = 0; x < screenWidth(); x++ )
                        for ( int y = 0; y < screenHeight(); y++ )
                                mvaddch( y, x, ' ' );
                refresh();
        }

        // de-allocate memory for message
        delete [] msg;
}

/*****************************************************************************\
 Prompt the user for a register to load to, then let them choose from a list
 of the .pgm files in the local images directory (defined as a constant)
\*****************************************************************************/
void loadImage( ImageType img[], bool loaded[], char name[][NAME_LEN] )
{
        // holds the file names, menuChoices is a copy with "Back" added at the end
        char **fileNames, **menuChoices;

        // the window that will hold the file menu
        WINDOW *fileMenu;

        // holds the image values N, M, and Q, the number of local image files, the
        // index of index of the register choosen and index of the file choosen
        int i, j, k, files, index, imageVal;

        // format of file (unused for now) but required for readImageHeader param
```

```
        bool f;

        // get a list of local files dynamically allocated
        files = findLocalPGM( fileNames );

        // add one more option to the menu
        menuChoices = new char*[files+1];

        // copy all the pointers from the filenames to the menuChoices
        for ( int a = 0; a < files; a++ )
                menuChoices[a] = fileNames[a];

        // delete the list of string pointers since menuChoices has them now
        delete [] fileNames;

        // create the final choice
        menuChoices[files] = new char[5];

        // make the final choice "Exit"
        strcpy( menuChoices[files], "Back" );

        // prompt for a register (false indicated it doesn't need to be full)
        index = promptForReg( loaded, name, false );

        // if exit wasn't choosen then prompt for a file
        if ( index != BAD_REG )
        {
                // prompt for file
                imageVal = showMenu( fileMenu, "Load Image", FILEWIN_HEIGHT,
                    FILEWIN_WIDTH, REGWIN_HEIGHT+2, MENU_WIDTH+3 , menuChoices,
                        files+1 );

                // if exit isn't choosen attempt to load image
                if ( imageVal != files )
                {
                        // read the image that was choosen
                        readImageHeader( menuChoices[imageVal], i, j, k, f );

                        // set up the image to store the correct data
                        img[index].setImageInfo( i, j, k );

                        // read and store image data
                        readImage( menuChoices[imageVal], img[index] );

                        // make sure that the register is read as full
                        loaded[index] = true;

                        // remove the file path from the front of the filename
                        // exampe: ./images/img.pgm -> img.pgm
                        sprintf( name[index], "Register %i: %s", index+1,
                            &(menuChoices[imageVal][strlen(IMAGELOC)]) );
                }

                // de-allocate fileMenu
                delwin( fileMenu );
        }
```

```
        // de-allocate list of menuChoices
        for ( int a = 0; a < files+1; a++ )
                delete [] menuChoices[a];
        delete [] menuChoices;

}

/*****************************************************************************\
 Save image from a register to the local images directory, prompting user for
 register and file name.
\*****************************************************************************/
void saveImage( ImageType img[], bool loaded[], char name[][NAMELEN] )
{
        // holds the file name
        char strInput[NAMELEN];

        // used to remove the file path from the front of the filename
        char imageLoc[NAMELEN];

        // holds the register that the user chooss
        int index;

        // prompt for register
        index = promptForReg( loaded, name );

        // if user doesn't choose 'Back'
        if ( index != BAD_REG )
        {
                // prompt the user for a file name
                promptForFilename( "Save Image", "Enter filename: ", strInput );

                // add .pgm to the filename if it wasnt already
                if ( strlen( strInput ) < 4 )
                        strcat( strInput, ".pgm" );
                else if ( strcmp( (strInput+strlen(strInput)-4), ".pgm" ) != 0 )
                        strcat( strInput, ".pgm" );

                // add the file path to the filename
                sprintf( imageLoc, "%s%s", IMAGELOC, strInput );

                // save the image to the given filename
                writeImage( imageLoc, img[index] );

                // set register name to match file name
                sprintf( name[index], "Register %i: %s", index+1, strInput );
        }
}

/*****************************************************************************\
 Simply retrieve image information and display to a window below the registers
 The data being displayed is the Register number, Image Height, Width, Q value,
 and average gray value.
\*****************************************************************************/
void getImageInfo( ImageType img[], bool loaded[], char name[][NAMELEN] )
{
```

```
        // hold image info
        int N, M, Q, index, y, x;

        // the window that holds all the info
        WINDOW *infoWin;

        // prompt for a register
        index = promptForReg( loaded, name );

        // if back isn't choosen
        if ( index != BAD_REG )
        {
                // retrieve image height, width, and color depth
                img[index].getImageInfo( N, M, Q );

                // draw/intialize the info window
                drawWindow( infoWin, name[index], 14, FILEWIN_WIDTH, REGWIN_HEIGHT+2,
                    MENU_WIDTH+3 );

                // set the starting x/y values
                x = 2;
                y = 2;

                // print all the information to the window with formating
                mvwprintw( infoWin, y, x, "%-20s%c %i", "Saved in Register",':',
                    index+1 );
                y+=2;
                mvwprintw( infoWin, y, x, "%-20s%c %i", "Image Width(pixels)",':',M );
                y+=2;
                mvwprintw( infoWin, y, x, "%-20s%c %i", "Image Height(pixels)",':',N );
                y+=2;
                mvwprintw( infoWin, y, x, "%-20s%c %i", "Color Depth",':',Q );
                y+=2;
                mvwprintw( infoWin, y, x, "%-20s%c %.2f", "Mean Gray Value",':',
                    img[index].meanGray() );
                y+=2;
                mvwprintw( infoWin, y, x, "Press Enter to continue..." );

                wrefresh( infoWin );

                // wait for input
                while ( wgetch( infoWin ) != KEY_RETURN );

                // de-allocate the window
                delwin( infoWin );
        }
}

/*****************************************************************************\
 Prompt user for a register then a pixel location (row, col) and then the pixel
 value to change that pixel to.
\*****************************************************************************/
void setPixel( ImageType img[], bool loaded[], char name[][NAME_LEN] )
{
        // holds various information about image
        int index, row, col, val;
```

```
        int N, M, Q;

        // prompt for register
        index = promptForReg( loaded, name );

        // if back isn't choosen
        if ( index != BAD_REG )
        {
                // prompt for a pixel location
                promptForLoc( "Set Pixel Value", img[index], row, col );

                // if back isn't choosen
                if ( row != -1 && col != -1 )
                {
                        // get image info (just for the Q)
                        img[index].getImageInfo( N, M, Q );

                        // prompt for the pixel with Q as the max value
                        promptForPixValue( "Set Pixel Value",
                            "Enter new pixel value(-1 to cancel): ", Q );

                        // if back isn't choosen
                        if ( val != -1 )
                        {
                                // change pixel value
                                img[index].setPixelVal( row, col, val );

                                // add modified to end of register name
                                if ( name[index][strlen(name[index])-1] != ')' )
                                        strcat( name[index], " (modified)" );
                        }
                }
        }
}

/*****************************************************************************\
 Return the value of a pixel in a selected image to the user.
\*****************************************************************************/
void getPixel( ImageType img[], bool loaded[], char name[][NAME_LEN] )
{
        // self describing variables
        int index, row = -1, col = -1, val;
        WINDOW *infoWin;

        // prompt for the register
        index = promptForReg( loaded, name );

        // if back isn't choosen
        if ( index != BAD_REG )
        {
                // prompt for pixel location
                promptForLoc( "Get Pixel Value", img[index], row, col );

                // if back isn't choosen
                if ( row != -1 && col != -1 )
                {
```

```
                        // create a message box window
                        stdWindow( infoWin, "Get Pixel Value" );

                        // put the pixel value message in the window
                        mvwprintw( infoWin, 1, 2, "The pixel Value at (%i,%i) is %i",
                                col, row, img[index].getPixelVal(row, col) );

                        // wait for input
                        while ( wgetch( infoWin ) != KEY_RETURN );

                        // de-allocate the message window
                        delwin( infoWin );

                }
        }
}

/*****************************************************************************\
 After getting the image to manipulate, prompt for two corners to make a
 subimage out of, if the lower right corner is above or left of the upper
 right corner re-prompt for valid points
\*****************************************************************************/
void extractSub( ImageType img[], bool loaded[], char name[][NAME_LEN] )
{
        // self documenting variables
        int index;
        int ULr, ULc, LRr = -1, LRc = -1;

        // temporary image to hold the subimage
        ImageType temp;

        // prompt for image register
        index = promptForReg( loaded, name );

        // if back isn't choosen
        if ( index != BAD_REG )
        {
                // prompt for the upper left corner
                promptForLoc( "Upper Left Corner", img[index], ULr, ULc );

                // if back isn't choosen prompt for lower right corner
                if ( ULr != -1 && ULc != -1 )
                        promptForLoc( "Lower Right Corner", img[index], LRr, LRc );

                // if invalid re-prompt for corners
                while ( ( LRr <= ULr || LRc <= ULc ) &&
                         ULr != -1 && ULc != -1 && LRr != -1 && LRc != -1 )
                {
                        // display message
                        messageBox( "Bad Coordinates",
                            "Lower Right Corner is Left or Above Upper Left" );

                        promptForLoc( "Upper Left Corner", img[index], ULr, ULc );

                        if ( ULr != -1 && ULc != -1 )
                                promptForLoc( "Lower Right Corner", img[index], LRr, LRc );
```

```
                }

                // if corners are good and back wasn't choosen
                if ( ULr != -1 && ULc != -1 && LRr != -1 && LRc != -1 )
                {
                        // extract sub image
                        temp.getSubImage( ULr, ULc, LRr, LRc, img[index] );

                        // set old image equal to subimage
                        img[index] = temp;

                        // adds modified to register name
                        if ( name[index][strlen(name[index])-1] != ')' )
                                strcat( name[index], " (modified)" );
                }
        }
}

/*****************************************************************************\
 This function prompts the user for a scale value to enlarge an image by, it
 makes sure the scale value does not make the image larger than MAX_IMG value
 because it may cause a stack overflow.
\*****************************************************************************/
void enlargeImg( ImageType img[], bool loaded[], char name[][NAME_LEN] )
{
        // holds image info and maxS value
        int index, N, M, Q, maxS;

        // scale value to be choosen by user
        int s;

        // temporary image used to store enlarged image
        ImageType temp;

        // prompt user for register value
        index = promptForReg( loaded, name );

        // if back isn't choosen
        if ( index != BAD_REG )
        {
                // get the basic image info used to determine maxS
                img[index].getImageInfo( N, M, Q );

                // calculate maxS
                maxS = (N > M ? MAX_IMG/N : MAX_IMG/M);

                // prompt for enlarge factor
                s = promptForScaleValue( "Enlarge Image By Factor",
                    "Enter enlargement multiplier(-1 to cancel): ", maxS );

                // if back isn't choosen
                if ( s != -1 )
                {
                        // enlarge image by factor s
                        temp.enlargeImage( s, img[index], CUBIC_INTER );
```

```
                        // set register image to the values of temp
                        img[index] = temp;

                        // adds modified to register name
                        if ( name[index][strlen(name[index])-1] != ')' )
                                strcat( name[index], " (modified)" );
                }
        }
}

/*****************************************************************************\
 The same as enlarge except it shrinks the image making sure it never gets
 smaller than MIN_IMG.  This is because some image viewers won't open images
 as small as 2x2 (xv for example)
\*****************************************************************************/
void shrinkImg( ImageType img[], bool loaded[], char name[][NAME_LEN] )
{
        // image info and max s value
        int index, N, M, Q, maxS;

        // scale factor to be reduced by
        int s;

        // holds the reduced image before transfering it to img[index]
        ImageType temp;

        // prompt for the register to be used
        index = promptForReg( loaded, name );

        // if quit isn't choosen
        if ( index != BAD_REG )
        {
                // get image info for calculating maxS
                img[index].getImageInfo( N, M, Q );

                // calculate maxS
                maxS = (N > M ? N/MIN_IMG : M/MIN_IMG);

                // prompt for the scale value
                s = promptForScaleValue( "Shrink Image By Factor",
                    "Enter reduction factor(-1 to cancel): ", maxS );

                // if quit isn't choosen
                if ( s != -1 )
                {
                        // shrink the image
                        temp.shrinkImage( s, img[index] );

                        // store back in the register image
                        img[index] = temp;

                        // adds modified to register name
                        if ( name[index][strlen(name[index])-1] != ')' )
                                strcat( name[index], " (modified)" );
                }
        }
```

```
}

/*****************************************************************************\
 Prompt user for a direction to reflect an image then reflect the image and
 store it back in the original register image.
\*****************************************************************************/
void reflectImg( ImageType img[], bool loaded[], char name[][NAME_LEN] )
{
        // holds the index of the register
        int index;

        // users direction choice
        char dir;

        // used to reflect the image before saving to image
        ImageType temp;

        // prompt for which image to use
        index = promptForReg( loaded, name );

        // if quit isn't choosen
        if ( index != BAD_REG )
        {
                // prompt for a valid reflect direction
                dir = promptForMirror("Reflect Image",
                    "Enter mirror direction (H(oriz), V(ert), C(ancel)): ");

                // if cancel isn't choosen
                if ( dir != 'c' && dir != 'C' )
                {
                        // reflect horizontally
                        if ( dir == 'h' || dir == 'H' ) // horizontal
                                temp.reflectImage( false, img[index] );
                        else                            // vertical
                                temp.reflectImage( true, img[index] );

                        // copy temp back to the register image
                        img[index] = temp;

                        // adds modified to register name
                        if ( name[index][strlen(name[index])-1] != ')' )
                                strcat( name[index], " (modified)" );
                }
        }
}

/*****************************************************************************\
 This prompts the user for how far to translate the image, then calls the
 translate function which moves the image down to the right 't' number of
 pixels.  Also Won't let user choose t value that would move image totaly off
 the screen.
\*****************************************************************************/
void translateImg( ImageType img[], bool loaded[], char name[][NAME_LEN] )
{
        // holds the image info and maximum t value
        int index, N, M, Q, maxT;
```

```
        // the translate value to be choosen by user
        int t;

        // temporary image used as a buffer to the register image
        ImageType temp;

        // get a valid image register
        index = promptForReg( loaded, name );

        // if back isn't choosen
        if ( index != BAD_REG )
        {
                // get the image info used to calculate maxT
                img[index].getImageInfo( N, M, Q );

                // calculate maxT value
                maxT = (N > M ? N-1 : M-1);

                // prompt for a valid T value (uses Pix because both pix or t can
                // be (0-max$)
                t = promptForPixValue( "Translate Image",
                    "Enter translation factor(-1 to cancel): ", maxT );

                // if cancel isn't choosen
                if ( t != -1 )
                {
                        // translate the image
                        temp.translateImage( t, img[index] );

                        // copy back to the image register
                        img[index] = temp;

                        // adds modified to register name
                        if ( name[index][strlen(name[index])-1] != ')' )
                                strcat( name[index], " (modified)" );
                }
        }
}

/*****************************************************************************\
 This prompts the user for an angle theta which will rotate the image counter
 clockwise by theta degrees.  The input is only valid from 0 to 360 which
 should cover all possibilities.
\*****************************************************************************/
void rotateImg( ImageType img[], bool loaded[], char name[][NAME_LEN] )
{
        // holds the register index and angle theta
        int index, theta;

        // temporary image used as a kind of buffer for register image
        ImageType temp;

        // prompt for a regiseter that is used
        index = promptForReg( loaded, name );
```

```
        // if quit isn't choosen
        if ( index != BAD_REG )
        {
                // prompt for valid angle
                theta = promptForAngle( "Rotate Image",
                    "Rotate counter-clockwise by angle(-1 to cancel):" );

                // if cancel isn't choosen
                if ( theta != -1 )
                {
                        // rotate the image by theta degrees clockwise
                        temp.rotateImage( theta, img[index] );

                        // set image register equal to buffer temporary image
                        img[index] = temp;

                        // adds modified to register name
                        if ( name[index][strlen(name[index])-1] != ')' )
                                strcat( name[index], " (modified)" );
                }
        }
}

/*****************************************************************************\
 Prompt for 2 images and attempt to sum them, there is no size checking because
 operator+ will throw a string which will be handeled by main if sizes of the
 two images are different.
\*****************************************************************************/
void sumImg( ImageType img[], bool loaded[], char name[][NAME_LEN] )
{
        // index of image 1 and 2
        int index1, index2;

        // 2 temporary images used to sum the images
        ImageType temp1, temp2;

        // prompt for first image
        index1 = promptForReg( loaded, name );

        // if quit isn't choosen
        if ( index1 != BAD_REG )
        {
                // set temp1 equal to the first image
                temp1 = img[index1];

                // prompt for second image but offset a little
                index2 = promptForReg( loaded, name, true, 3, MENU_WIDTH+5 );

                // if quit isn't choosen
                if ( index2 != BAD_REG )
                {
                        // set temp2 equal to the second image
                        temp2 = img[index2];

                        // sum the images
                        img[index1] = temp1 + temp2;
```

```
                        // adds modified to register name
                        if ( name[index1][strlen(name[index1])-1] != ')' )
                                strcat( name[index1], "_(modified)" );
                }
        }
}

/*****************************************************************************\
 Prompt for 2 images and attempt to calculate the difference, there's no size
 checking here for the same reason sumImg doesn't do size checking
\*****************************************************************************/
void subtractImg( ImageType img[], bool loaded[], char name[][NAME_LEN] )
{
        // index of image 1 and 2
        int index1, index2;

        // 2 temporary images used to subtract the images
        ImageType temp1, temp2;

        // prompt for first image
        index1 = promptForReg( loaded, name );

        // if quit isn't choosen
        if ( index1 != BAD_REG )
        {
                // set temp1 equal to the first image
                temp1 = img[index1];

                // prompt for second image but offset a little
                index2 = promptForReg( loaded, name, true, 3, MENU_WIDTH+5 );

                // if quit isn't choosen
                if ( index2 != BAD_REG )
                {
                        // set temp2 equal to the second image
                        temp2 = img[index2];

                        // subtract the images
                        img[index1] = temp1 - temp2;

                        // adds modified to register name
                        if ( name[index1][strlen(name[index1])-1] != ')' )
                                strcat( name[index1], "_(modified)" );
                }
        }
}

/*****************************************************************************\
 Prompt user for which image to negate and negate it, pretty simple function.
\*****************************************************************************/
void negateImg( ImageType img[], bool loaded[], char name[][NAME_LEN] )
{
        // index of register to use
        int index;
```

```
        // prompt for register number
        index = promptForReg( loaded, name );

        // if quit isn't choosen
        if ( index != BAD_REG )
        {
                // negate image
                img[index].negateImage();

                // adds modified to register name
                if ( name[index][strlen(name[index])-1] != ')' )
                        strcat( name[index], " (modified)" );
        }
}

/*****************************************************************************\
 This is the function that calls the menu for the register prompt, it can be
 called in different locations (like in addImg and subImg) but has a default
 defined by some global constants.  The function creates a list of registers
 and adds the "Back" option as the final option, this way the user has the
 option to cancel choosing a register.  Although in the program it looks like
 the register display and register choosing window are the same, this menu
 overlaps the other menu to make it seem like control is transfering to another
 window.

 The value check is true by default, if it is false it can return registers
 that have not been loaded.  This feature is needed in functions like
 loadImage.
\*****************************************************************************/
int promptForReg( bool loaded[], char name[][NAME_LEN], const bool check,
    int y, int x )
{
        // this is the WINDOW that the menu is stored in
        WINDOW *regMenu;

        // val is the index choice for the menu
        int val;

        // set the flag to continue looping until BAD_REG is true or a register
        // that is not set is choosen which check if true
        bool loop = true;

        // this is the array that will hold all the menu choices including "Back"
        char menuVals[REGS+1][NAME_LEN];

        // copy all the register names to the menu array
        for ( int i = 0; i < REGS; i++ )
                strcpy( menuVals[i], name[i] );

        // add exit to the list of commands
        strcpy( menuVals[REGS], "Back" );

        // prompt for the register
        do {
                val = showMenu( regMenu, "Registers", REGWIN_HEIGHT, REGWIN_WIDTH, y, x,
                    menuVals, REGS+1 );
```

```
                // set the loop flag depending on val
                if ( val == BAD_REG )
                        loop = false;
                else if ( ! loaded[val] && check )
                        loop = true;
                else
                        loop = false;
        } while ( loop );

        // delete the register menu window
        delwin( regMenu );

        // return the users choice
        return val;
}

/*****************************************************************************\
 This just builds the window used for message box, this function is just to
 simplify the plethora of other functions that use this.
\*****************************************************************************/
void stdWindow( WINDOW *&newWin, const char title[] )
{
        // simply draw the standard msg box window
        drawWindow( newWin, title, MSGBOX_HEIGHT, MSGBOX_WIDTH,
                        screenHeight()/2-MSGBOX_HEIGHT/2, screenWidth()/2-MSGBOX_WIDTH/2 );
}

/*****************************************************************************\
 Create a message box and prompt the user for a string value with given prompt
\*****************************************************************************/
int promptForFilename( const char title[], const char prompt[], char str[] )
{
        // holds the prompting window
        WINDOW *fileWin;

        // this is how long the filename can be (some arbitrary value right?...no)
        int len = 16;

        // draw the window
        stdWindow( fileWin, title );

        // prompt for the string
        promptForString( fileWin, 1, 2, prompt, str, len );

        // delete the window
        delwin( fileWin );

        // return the length of the string (not really used in this program)
        return strlen( str );
}

/*****************************************************************************\
 Prompt user for a valid angle using a message box, make sure input is between
 0 and 360, if not display a message box and then re-prompt.
\*****************************************************************************/
```

```
int promptForAngle( const char title[], const char prompt[] )
{
        // holds message box window
        WINDOW *pixWin;

        // user input value
        int val;

        // draw message window
        stdWindow( pixWin, title );

        // get user input
        val = promptForInt( pixWin, 1, 2, prompt );

        // check for valid input
        while ( val < -1 || val > 360 )
        {
                // display error
                messageBox( "Invalid Angle", "Please input an angle (0-360)" );

                // redraw window
                delwin( pixWin );
                stdWindow( pixWin, title );

                // re-prompt user
                val = promptForInt( pixWin, 1, 2, prompt );
        }

        // de-allocate window object
        delwin( pixWin );

        // return users choice
        return val;
}

/*****************************************************************************\
 Prompt for a pixel value which is from 0 to maxVal, if not display message
 box and re-prompt user until valid choice is made.
\*****************************************************************************/
int promptForPixValue( const char title[], const char prompt[], int maxVal )
{
        // message box window
        WINDOW *pixWin;

        // used in the error message
        char msg[NAME_LEN];

        // user input value
        int val;

        // draw message window
        stdWindow( pixWin, title );

        // get user input
        val = promptForInt( pixWin, 1, 2, prompt );
```

```
        // check for valid input
        while ( val < -1 || val > maxVal )
        {
                // display error
                sprintf( msg, "Please input a value (0-%i)", maxVal );
                messageBox( "Invalid Value", msg );

                // redraw window
                delwin( pixWin );
                stdWindow( pixWin, title );

                // re-prompt user
                val = promptForInt( pixWin, 1, 2, prompt );
        }

        // delete this dynamic memory
        delwin( pixWin );

        // return value to calling function
        return val;
}

/*****************************************************************************\
 Prompt the user for the characters h, v, or c (not case sensitive) and return
 the value as soon as one of the 3 is pressed.
\*****************************************************************************/
char promptForMirror( const char title[], const char prompt[] )
{
        // holds the prompting window
        WINDOW *pixWin;

        // holds user input
        char val;

        // draw message window
        stdWindow( pixWin, title );

        // display prompt message
        mvwaddstr( pixWin, 1, 2, prompt );

        //dont continue untill valid key is pressed
        do {
                // prompt user for character
                val = wgetch( pixWin );
        } while ( val != 'h' && val != 'H' &&
                  val != 'v' && val != 'V' &&
                  val != 'c' && val != 'C' );

        // delete dynamically allocated window
        delwin( pixWin );

        // return the users input
        return val;
}

/*****************************************************************************\
```

```
 This function prompts the user for a scale value and checks to make sure it
 is not greater than maxVal and not less than 2.  This is used in the enlarge
 and shrink functions.
\*****************************************************************************/
int promptForScaleValue( const char title[], const char prompt[], int maxVal )
{
        // points to the WINDOW that is our prompting window
        WINDOW *pixWin;

        // holds the error message (which needs some formating)
        char msg[NAME_LEN];

        // the users input value
        int val;

        // draw message window
        stdWindow( pixWin, title );

        // prompt user for an integer value
        val = promptForInt( pixWin, 1, 2, prompt );

        // if value is not valid display error message and re-prompt
        while ( val != -1 && ( val < 2 || val > maxVal ) )
        {
                // display error
                sprintf( msg, "Please input a value (2-%i)", maxVal );
                messageBox( "Invalid Value", msg );

                // redraw window
                delwin( pixWin );
                stdWindow( pixWin, title );

                // re-prompt user
                val = promptForInt( pixWin, 1, 2, prompt );
        }

        // delete dynamically allocated window
        delwin( pixWin );

        // return the user's input value
        return val;
}

/*****************************************************************************\
 This function prompts the user for a location (both row and column) and sets
 the valid points equal to row or col.  If -1 is returned in either location
 it means user choose to cancel the prompt.  The validity of the points is
 calculated by the image object it is passed.  The image properties are
 calculated and then used to determine the bounds of row and column.
\*****************************************************************************/
void promptForLoc( const char title[], ImageType& img, int& row, int& col )
{
        // holds various image info
        int N, M, Q;

        // holds the error messages
```

```
        char msg[NAME_LEN];

        // this is the WINDOW pointer that points to the prompting window
        WINDOW *pixWin;

        // set default values for row and column it exits early
        row = -1;
        col = -1;

        // retrieve image info
        img.getImageInfo( N, M, Q );

        // draw message window
        stdWindow( pixWin, title );

        // gets user input for row
        row = promptForInt( pixWin, 1, 2, "Enter pixel row (-1 to cancel): " );

        // if row input is not valid display error and re-prompt
        while ( row < -1 || row >= N )
        {
                // show message box
                sprintf( msg, "Invalid Row, must be (0-%i)", N-1 );
                messageBox( "Invalid Row", msg );

                // redraw the window
                delwin( pixWin );
                stdWindow( pixWin, title );

                // re-prompt user
                row = promptForInt( pixWin, 1, 2, "Enter pixel row (-1 to cancel): " );
        }

        // if user didn't choose to cancel
        if ( row != -1 )
        {
                // prompt for column
                col = promptForInt( pixWin, 2, 2,
                    "Enter pixel column (-1 to cancel): " );

                // if column input is not valid, display error and re-prompt
                while ( col < -1 || col >= M )
                {
                        // show message box warning
                        sprintf( msg, "Invalid Column, must be (0-%i)", M-1 );
                        messageBox( "Invalid Column", msg );

                        // redraw the window
                        delwin( pixWin );
                        stdWindow( pixWin, title );

                        // reprint the upper line
                        mvwprintw( pixWin, 1, 2, "Enter pixel row (-1 to cancel): %i",
                            row );

                        // re-prompt user
```

```
                        col = promptForInt( pixWin, 2, 2,
                            "Enter pixel column(-1 to cancel): " );
                }
        }

        // at this point row and column should be set or should be -1 (cancel)

        // de-allocate memory for WINDOW object
        delwin( pixWin );
}

/*****************************************************************************\
 This displays a simple message box to the screen with the given title and msg
 inside of it, it waits for the user to press RETURN before returning to
 calling function
\*****************************************************************************/
void messageBox( const char title[], const char msg[] )
{
        // message box window
        WINDOW *msgBox;

        // user
        int input;

        // draw/initialize window
        stdWindow( msgBox, title );

        // add msg value to window
        mvwaddstr( msgBox, 1, 2, msg );

        // wait for return to be pressed
        while ( wgetch( msgBox ) != KEY_RETURN );

        // delete message box window
        delwin( msgBox );
}

/*****************************************************************************\
 Couldn't find a good place to put this function, its a not so robust function
 that reads all the .pgm files from a local directory (defined as a constant)
 and places them into a dynamically allocated c style string array.

 !!!Note this function allocates memory for a 2D array and returns the number
 of rows.  This information is REQUIRED to properly de-allocate the memory in
 the calling function.
\*****************************************************************************/
int findLocalPGM( char **&filenames )
{
        // namelist holds a list of file names
        struct dirent **namelist;

        // n is the number of files total in the local directory
        int n;

        // holds the length of various strings
        int len;
```

```
// count keeps track of the number of .pgm files found
int count = 0;

// store the string values of the local files into namelist in alpha order
n = scandir( IMAGELOC, &namelist, 0, alphasort );

// if there are files...
if ( n > 0 )
{
        // go through checking for files ending in ".pgm"
        for ( int i = 0; i < n; i++ )
        {
                len = strlen( namelist[i]->d_name );
                if ( len > 5 )
                        // compare the last four characters to ".pgm"
                        if ( strcmp( ".pgm", &(namelist[i]->d_name[len-4]) ) == 0 )
                                // increase count
                                count++;
        }

        // if any .pgm files are found
        if ( count > 0 )
        {
                // allocate space for each one
                filenames = new char*[count];

                // j is used as a counter for the filenames
                int j = 0;

                // this loop does the same as the previous except it allocates
                // memory and copies names of .pgm files to filenames
                for ( int i = 0; i < n; i++ )
                {
                        // get the length of the filename
                        len = strlen( namelist[i]->d_name );
                        // compare the suffix to ".pgm" again
                        if ( len > 5 )
                                if ( strcmp( ".pgm", &(namelist[i]->d_name[len-4]) ) == 0
                                {
                                        // this time allocate memory to store the name
                                        filenames[j] = new char[strlen(namelist[i]->d_name
                                            1 + strlen(IMAGELOC)];
                                        // store the name with the file path added
                                        sprintf( filenames[j], "%s%s", IMAGELOC,
                                            namelist[i]->d_name );
                                        // increase counter
                                        j++;
                                }
                        // de-allocate name list
                        delete [] namelist[i];
                }
                // finish de-allocating name list
                delete [] namelist;
        }
}
```

```
        // return the number of rows in filenames
        return count;
}
```

7 image.h

```
/*****************************************************************************\
 Authors: Josiah Humphrey and Joshua Gleason

 Date Due for Review: 02/16/2010

 This object is used for storing and manipulating images, it allows the images
 to be processed and manipulated in multiple ways including, re-scaling,
 rotating, translating, adding, subtracting, negating, etc...
\*****************************************************************************/

#ifndef IMAGE_H
#define IMAGE_H

class ImageType {
public:

// CONSTRUCTORS AND DESTRUCTOR ////////////////////////////////////////////////
        // default construtor, sets everything to NULL and 0
        ImageType();

        // parameterized constructor sets up image to N, M and Q values
        ImageType(int, int, int);

        // copy allocates memory and copies info from the right hand side
        ImageType( const ImageType& );

        // same as copy except it de-allocates memory first if necessary
        ImageType& operator= ( const ImageType& );

        // destructor removes all dynamically allocated memory
        ~ImageType();

// IMAGE FUNCTIONS ////////////////////////////////////////////////////////////
        // returns the N, M and Q values to the calling function
        void getImageInfo(int&, int&, int&) const;

        // sets N, M and Q and also de-allocates and allocates memory if necessary
        void setImageInfo(int, int, int);

        // sets the value of a pixel at row, column to the 3rd parameters value
        void setPixelVal(int, int, int);

        // return the pixel value at the desired location
        int getPixelVal(int, int) const;

////// Josh's functions ////////////////////////////////////////////////////////

        // returns the mean value of all the pixels
        double meanGray() const;
```

```
        // enlarge the image by a factor is s, uses bi-cubic interpolation if the
        // bool is true, otherwise uses bi-linear interpolation which is more light
        // weight.  The version with an int value for s simply casts s to a double
        // and calls the version that takes a double
        void enlargeImage( double, const ImageType&, bool=true );
        void enlargeImage( int, const ImageType&, bool=true );

        // reflect the image either horiz. or vert. depending on the bool value
        void reflectImage( bool, const ImageType& );

        // subtract 2 images, make any pixels that differ by more than Q/6 white
        // and make the rest black.  This helps reduce some noise
        ImageType& operator- ( const ImageType& );

        // calculate the negative of every pixel (Q - current value)
        void negateImage();

////// Josiah's functions ////////////////////////////////////////////////////

        // calculate a sub image given the row,col for the upper left and lower
        // right corners
        void getSubImage( int, int, int, int, const ImageType& );

        // shrink image by a factor of s, find the average value for each 'block' of
        // pixels that is reduced and use that value.  This makes a smoother reduce
        // function
        void shrinkImage( int, const ImageType& );

        // translate the image down to the right by t pixels, effectively cutting
        // off the bottom and right side by the same number of pixels
        void translateImage( int, const ImageType& );

        // rotate image by theta degrees counter-clockwise
        void rotateImage( int, const ImageType& );

        // sum two images giving no particular bias to one or the other
        ImageType& operator+ ( const ImageType& );


private:
        int N; // # of rows
        int M; // # of cols
        int Q; // # of gray-level values

        // array of pixel values
        int **pixelValue;
};

#endif
```

8 image.cpp

```
#include <iostream>
#include <cstdlib>
#include <cmath>
```

```
#include "cubicSpline.h"
#include "image.h"

using namespace std;

// size of one grid on the background grid
const int BACKGRID = 25;

/*****************************************************************************\
 default constructor allocates no memory and sets the size to zero
\*****************************************************************************/
ImageType::ImageType()
{
        // start everything at zero
        N = 0;
        M = 0;
        Q = 0;

        pixelValue = NULL;
}

/*****************************************************************************\
 destructor wipes any memory that was dynamically allocated
\*****************************************************************************/
ImageType::~ImageType()
{
        if ( pixelValue != NULL )
        {
                // delete all the memory for the array
                for ( int i = 0; i < N; i++ )
                        delete [] pixelValue[i];
                delete [] pixelValue;
        }
}

/*****************************************************************************\
 change the dimensions of the image, delete and re-allocate memory if required
\*****************************************************************************/
ImageType::ImageType(int tmpN, int tmpM, int tmpQ)
{
        // set the new values of N, M and Q
        setImageInfo( tmpN, tmpM, tmpQ );
}

/*****************************************************************************\
 copy contructor, copys data from rhs to the current object
\*****************************************************************************/
ImageType::ImageType( const ImageType& rhs )
{
        // set the info to the new image data
        setImageInfo( rhs.N, rhs.M, rhs.Q );

        for ( int i = 0; i < N; i++ )
                for ( int j = 0; j < M; j++ )
                        pixelValue[i][j] = rhs.pixelValue[i][j];
}
```

```
/*****************************************************************************\
 equal operator overload, this is basically the same as the copy constructor
 except it will likely have to de-allocate memory before copying values, all
 this is decided in setImageInfo however
\*****************************************************************************/
ImageType& ImageType::operator= ( const ImageType& rhs )
{
        if(this != &rhs){
                setImageInfo( rhs.N, rhs.M, rhs.Q );

                // copy pixel values
                for ( int i = 0; i < N; i++ )
                        for ( int j = 0; j < M; j++ )
                                pixelValue[i][j] = rhs.pixelValue[i][j];
        }

        return *this;
}

/*****************************************************************************\
 returns the width height and color depth to reference variables
\*****************************************************************************/
void ImageType::getImageInfo(int& rows, int& cols, int& levels) const
{
        rows = N;
        cols = M;
        levels = Q;
}

/*****************************************************************************\
 sets the image info, deleting and allocating memory as required, also create
 background grid
\*****************************************************************************/
void ImageType::setImageInfo(int rows, int cols, int levels)
{
        // re-allocate the integer array if the size changes
        if ( N != rows && M != cols )
        {
                // delete memory if not NULL
                if ( pixelValue != NULL )
                {
                        for ( int i = 0; i < N; i++ )
                                delete [] pixelValue[i];
                        delete [] pixelValue;
                }

                // sets N and M
                N = rows;
                M = cols;

                // allocate the rows of pixel value
                pixelValue = new int* [N];

                // allocate the columns of pixel value
                for ( int i = 0; i < N; i++ )
```

```
                        pixelValue[i] = new int[M];
        }

        // set Q equal to the levels
        Q = levels;

        // make a checkered background
        for ( int i = 0; i < N; i++ )
                for ( int j = 0; j < M; j++ )
                {
                        if ( ( (i%(BACKGRID*2)+1.0) / (BACKGRID*2.0) > 0.5 &&
                                (j%(BACKGRID*2)+1.0) / (BACKGRID*2.0) <= 0.5 ) ||
                             ( (i%(BACKGRID*2)+1.0) / (BACKGRID*2.0) <= 0.5 &&
                                (j%(BACKGRID*2)+1.0) / (BACKGRID*2.0) > 0.5 ) )
                                pixelValue[i][j] = Q/2;
                        else
                                pixelValue[i][j] = Q/3;
                }
}

/*****************************************************************************\
 sets the value of a pixel
\*****************************************************************************/
void ImageType::setPixelVal(int i, int j, int val)
{
        pixelValue[i][j] = val;
}

/*****************************************************************************\
 returns the value of a pixel
\*****************************************************************************/
int ImageType::getPixelVal(int i, int j) const
{
        return pixelValue[i][j];
}

/*************************Josh's functions*********************************/

/*****************************************************************************\
 this calculates the average gray value in the picture, this is done by adding
 all of the pixels and dividing by the total number of pixels
\*****************************************************************************/
double ImageType::meanGray() const
{
        double gray = 0;

        // sum the gray values
        for ( int i = 0; i < N; i++ )
                for ( int j = 0; j < M; j++ )
                        gray += pixelValue[i][j];

        // return 0 if there are no pixels, otherwise divide gray by total pixels
        return ( M*N == 0 ? 0 : gray/(M*N) );
}

/*****************************************************************************\
```

```
 This simply calls the double version of enlargeImage
\*****************************************************************************/
void ImageType::enlargeImage( int S, const ImageType& old, bool cubic )
{
                // call double version of enlarge
                enlargeImage( (double)S, old, cubic );
}

/*****************************************************************************\
 This function enlarges an image by a magnitude of s, so for example if the
 original function was 100x100 and s is 10, then the new image is 1000x1000

 The method I choose to use was bicubic/linear interpolation which creates
 splines for each column(cubic or linear), then using those splines create an
 image which is a stretched version of the original image.  The way I achieved
 this was to stretch the entire image only vertically, and then stretch that
 image horizontally.  I did the same thing except reversed (stretched image
 horizontally first) and then summed the two images together.  This gives an
 average value between both methods.  Although it can handle S values less
 than 1, the shrinkImage function works better for this.

 if cubic = true then use cubic interpolation
 if cubic = false then use linear interpolation
\*****************************************************************************/
void ImageType::enlargeImage( double S, const ImageType& old, bool cubic )
{
        // check parameters
        if ( old.M < 4 || old.N < 4 && old.M != 0 && old.N != 0 )
                // force linear if less than 4 height or width
                cubic = false;
        else if ( old.M == 0 || old.N == 0 )
                throw (string)"Image file is empty!";

        // slightly modify S to make it a better value for spline
        double Shoriz = (double)((int)(old.M*S))/old.M;
        double Svert = (double)((int)(old.N*S))/old.N;

        int colorVal;   // used to hold the calculated color value
        double splineX; // the x value passed to the spline function

        int *horizVals = new int[old.M];        // holds points for spline
        int *vertVals = new int[old.N];         // holds points for spline

        // temporary images used to stretch before 2nd interpolation
        ImageType horiz, vert, half1, half2;

        // this is the spline object used to store the spline values
        cubicSpline spline;

        // set the new image to the
        setImageInfo( old.N * S, old.M * S, old.Q );

        // set temp to a stretched horizontally only
        horiz.setImageInfo( old.N, M, Q );
        vert.setImageInfo( N, old.M, Q );
        half1.setImageInfo( N, M, Q );
```

```
        half2.setImageInfo( N, M, Q );

        // stretch old image vertiacally and store in vert
        for ( int col = 0; col < old.M; col++ )
        {
                // get the values used to create the spline for the column
                for ( int row = 0; row < old.N; row++ )
                        vertVals[row] = old.pixelValue[row][col];

                // actually create the spline (assumed the pixels are equally spaced)
                if ( cubic )
                        spline.createCubic( vertVals, old.N );
                else
                        spline.create( vertVals, old.N );

                // using the spline set the values of vert
                for ( int row = 0; row < N; row++ )
                {
                        // value to pull from spline for current row
                        splineX = (row-Svert/2.0)/(N-Svert-1.0) * 100.0;
                        colorVal;

                        if ( cubic )
                                colorVal = spline.getCubicVal(splineX);
                        else
                                colorVal = spline.getVal(splineX);

                        // NOTE: don't clip values yet, it causes problems...

                        vert.pixelValue[row][col] = colorVal;
                }
        }

        // now stretch vert horizontally and store in half1
        for ( int row = 0; row < N; row++ )
        {
                // get the values used to create the spline for the row
                for ( int col = 0; col < old.M; col++ )
                        horizVals[col] = vert.pixelValue[row][col];

                // actually create the spline (cubic if parameter is true)
                if ( cubic )
                        spline.createCubic( horizVals, old.M );
                else
                        spline.create( horizVals, old.M );

                // using the spline set the values
                for ( int col = 0 ; col < M; col++ )
                {
                        splineX = (col-Shoriz/2.0)/(M-Shoriz-1.0) * 100.0;
                        colorVal;

                        // obtain new color from spline value
                        if ( cubic )
                                colorVal = spline.getCubicVal(splineX);
                        else
```

```
                        colorVal = spline.getVal(splineX);

                    // clip the color if it goes out of bounds
                    if ( colorVal < 0 ) colorVal = 0;
                    if ( colorVal > Q ) colorVal = Q;

                    // set the pixel value for half1
                    half1.pixelValue[row][col] = colorVal;
            }
    }

    // stretch old image horizontally and store in horiz
    for ( int row = 0; row < old.N; row++ )
    {
            // get the values used to create the spline for the row
            for ( int col = 0; col < old.M; col++ )
                    horizVals[col] = old.pixelValue[row][col];

            // actually create the spline (assumed the pixels are equally spaced)
            if ( cubic )
                    spline.createCubic( horizVals, old.M );
            else
                    spline.create( horizVals, old.M );

            // using the spline set the values of temp
            for ( int col = 0; col < M; col++ )
            {
                    // value to pull from spline for current col
                    splineX = (col-Shoriz/2.0)/(M-Shoriz-1.0) * 100.0;
                    colorVal;

                    if ( cubic )
                            colorVal = spline.getCubicVal(splineX);
                    else
                            colorVal = spline.getVal(splineX);

                    // NOTE: don't clip values yet, it causes problems...

                    horiz.pixelValue[row][col] = colorVal;
            }
    }

    // now stretch horiz vertically and store in half2
    for ( int col = 0; col < M; col++ )
    {
            // get the values used to create the spline for the column
            for ( int row = 0; row < old.N; row++ )
                    vertVals[row] = horiz.pixelValue[row][col];

            // actually create the spline (cubic if parameter is true)
            if ( cubic )
                    spline.createCubic( vertVals, old.N );
            else
                    spline.create( vertVals, old.N );

            // using the spline set the values
```

```
                for ( int row = 0 ; row < N; row++ )
                {
                        splineX = (row-Svert/2.0)/(N-Svert-1.0) * 100.0;
                        colorVal;

                        // obtain new color from spline value
                        if ( cubic )
                                colorVal = spline.getCubicVal(splineX);
                        else
                                colorVal = spline.getVal(splineX);

                        // clip the color if it goes out of bounds
                        if ( colorVal < 0 ) colorVal = 0;
                        if ( colorVal > Q ) colorVal = Q;

                        // set the value of half2
                        half2.pixelValue[row][col] = colorVal;
                }
        }

        // sum half1 and half2 to get the average value everywhere
        *this = half1 + half2;

        // de-allocate all that memory
        delete [] horizVals;
        delete [] vertVals;
}

/*****************************************************************************\
 reflects image by moving the pixel to N or M minus the current row or column
 depending on the value of the flag (true being a horizontal reflection and
 false being a vertical reflection)
\*****************************************************************************/
void ImageType::reflectImage( bool flag, const ImageType& old )
{
        // set image info same as old's
        setImageInfo( old.N, old.M, old.Q );

        // if flag is set copy opposite row, if not copy opposite column
        for ( int i = 0; i < N; i++ )
                for ( int j = 0; j < M; j++ )
                        if ( flag )
                                pixelValue[i][j] = old.pixelValue[i][M-j-1];
                        else
                                pixelValue[i][j] = old.pixelValue[N-i-1][j];
}

/*****************************************************************************\
 subtract two images from eachother to see the differences, if the magnitude of
 the difference is less then Q/6 then the pixel is replaced with black,
 otherwise white is used.  This seems to help reduce the amount of noise in the
 pictures
\*****************************************************************************/
ImageType& ImageType::operator- ( const ImageType& rhs )
{
        // throw exception if images don't match
```

```
        if ( N != rhs.N || M != rhs.M || Q != rhs.Q )
                throw (string)"Images do not have the same dimensions!";

        for ( int i = 0; i < N; i++ )
                for ( int j = 0; j < M; j++ )
                {
                        // calculate subtracted value
                        pixelValue[i][j] = abs(pixelValue[i][j] - rhs.pixelValue[i][j]);

                        // if pixels are less than Q/6 different then make them black
                        // this helps prevent noise
                        if ( pixelValue[i][j] < Q/6 )
                                pixelValue[i][j] = 0;
                        else
                                pixelValue[i][j] = Q;
                }
        return *this;   // return current object
}

/*****************************************************************************\
 This function simply subtracts the current pixel value from the max value of
 the pixel, thus negating the image
\*****************************************************************************/
void ImageType::negateImage()
{
        // calculate the negative of each pixel
        for ( int i = 0; i < N; i++ )
                for ( int j = 0; j < M; j++ )
                        pixelValue[i][j] = Q - pixelValue[i][j];
}

/*****************************Josiah's functions*********************************/

/*****************************************************************************\
 Obtain a sub-image from old.  Uses the coordinates of the upper left corner
 and lower right corner to obtain image.
\*****************************************************************************/
void ImageType::getSubImage( int ULr, int ULc, int LRr, int LRc,
    const ImageType& old )
{
        int width, height;
        // get sub image height
        height = abs(ULr - LRr);

        // get sub image width
        width = abs(ULc - LRc);

        // make a new array for the exact size of the new subimage
        setImageInfo(height, width, old.Q);

        // copy over the old stuff into the new subimage array
        for(int i = 0; i < N; i++)
                for(int j = 0; j < M; j++)
                        pixelValue[i][j] = old.pixelValue[ULr+i][ULc+j];
}
```

```
/*****************************************************************************\
 Shrink image, average all the values in the block to make the new pixel, this
 makes the shrink much less jagged looking in the end
\*****************************************************************************/
void ImageType::shrinkImage( int s, const ImageType& old )
{
        // used to find average pixel value
        int total, num;

    // make new array with correct size
        setImageInfo(old.N / s, old.M / s, old.Q);

        // copy over every s pixel
        for(int i = 0; i < N; i++)
                for(int j = 0; j < M; j++) {
                        // reset values for averaging
                        total = num = 0;

                        // sum the total value of pixels in the row/col
                        for ( int row = i*s; row < (i+1)*s; row++ )
                                for ( int col = j*s; col < (j+1)*s; col++ ) {
                                        total += old.pixelValue[row][col];
                                        num++;
                                }

                        // set the new pixel value
                        pixelValue[i][j] = total/num;
                }
}

/*****************************************************************************\
 Translate the image down to the right, any part that goes out of the screen is
 not calculated.  Checkered background from setImageInfo is retained.
\*****************************************************************************/
void ImageType::translateImage( int t, const ImageType& old )
{
        //make this image's image array
        setImageInfo(old.N, old.M, old.Q);

        // count backwards through to the newly defined upper left corner copying
        // from everywhere before
        for(int i = N - 1; i >= 0 + t; i--)
                for(int j = M - 1; j >= 0 + t; j--)
                        pixelValue[i][j] = old.pixelValue[i-t][j-t];
}

/*****************************************************************************\
 Rotate the image counter-clockwise using bilinear interpolation, basically
 traversing the entire image going from the destination to the source by using
 the equation in reverse (which is why the angle is reversed).  Once a location
 is determined the surrounding pixels are used to calculate intermediate values
 between the pixels, this gives a pretty smooth rotate.

 - Originally written by Josiah, modified with Joshua's help
\*****************************************************************************/
void ImageType::rotateImage( int theta, const ImageType& old )
```

```
{
        // reverse theta to make a counter-clockwise rotation
        theta *= -1;

        // set image to correct size
        setImageInfo(old.N, old.M, old.Q);
        int final;      // holds final color value for given location

        // holds various color values
        int UL, UR, LL, LR, U, D, L, R, Hval, Vval;

        // holds the slopes between different points
        int USlope, DSlope, LSlope, RSlope, HSlope, VSlope;

        // 4 * atan(1) = pi
        float rad = theta * 4 * atan(1.0)/180;

        // row, column, middle row, middle column
        double r, c, r_0, c_0;

        // set middle values
        r_0 = N/2.0;
        c_0 = M/2.0;

        // loops through entire picture, going from dest, to source to prev holes
        for(int i = 0; i < N; i++){
                for(int j = 0; j < M; j++){
                        // calculate where the original value should be
                        r = r_0 + (i-r_0)*cos(rad) - (j-c_0)*sin(rad);
                        c = c_0 + (i-r_0)*sin(rad) + (j-c_0)*cos(rad);

                        // only draw a pixel if source value is in range
                        if ( r > 0 && ceil(r) < N && c > 0 && ceil(c) < M ) {

                                // get four pixel value which surround the desired value
                                UL = old.pixelValue[(int)r][(int)c];
                                UR = old.pixelValue[(int)r][(int)ceil(c)];
                                LL = old.pixelValue[(int)ceil(r)][(int)c];
                                LR = old.pixelValue[(int)ceil(r)][(int)ceil(c)];

                                // find the slope of the line between all four corners
                                USlope = UR - UL;
                                DSlope = LR - LL;
                                LSlope = LL - UL;
                                RSlope = LR - UR;

                                // get the intermediate value corresponding with desired r/c val
                                U = UL + USlope*(c - (int)c);
                                D = LL + DSlope*(c - (int)c);
                                L = UL + LSlope*(r - (int)r);
                                R = UR + RSlope*(r - (int)r);

                                // get the slop between intermediate values
                                HSlope = D-U;
                                VSlope = R-L;
```

```
                                // find 2 different color estimations of the desired pixel
                                Hval = U + HSlope*(r - (int)r);
                                Vval = L + VSlope*(c - (int)c);

                                // average the estimations
                                final = (Hval + Vval) / 2;
                        }
                        else if ( r > 0 && ceil(r) < N && c > 0 && c < M ) { // right edge
                                // get upper and lower value
                                UL = old.pixelValue[(int)r][(int)c];
                                LL = old.pixelValue[(int)ceil(r)][(int)c];

                                // find slope between two values
                                LSlope = LL - UL;

                                // get value of final point
                                final = UL + LSlope*(r - (int)r);
                        }
                        else if ( r > 0 && r < N && c > 0 && ceil(c) < M ) { // bottom edge
                                // get left and right values
                                UL = old.pixelValue[(int)r][(int)c];
                                UR = old.pixelValue[(int)r][(int)ceil(c)];

                                // find slope between two values
                                USlope = UR - UL;

                                // get value of final point
                                final = UL + USlope*(c - (int)c);
                        }
                        else if ( r > 0 && r < N && c > 0 && c < M ) { // lower right
                                // no slopes, just set value
                                final = old.pixelValue[(int)r][(int)c];
                        }
                        else { // no value here
                                final = pixelValue[i][j]; // retain background
                        }

                        // make sure final value is not out of color bounds
                        if ( final < 0 ) final = 0;
                        if ( final > Q ) final = Q;

                        // set final pixel value
                        pixelValue[i][j] = final;
                }
        }

}

/*****************************************************************************\
 Sum two images together, basically just finding the average pixel value of
 every pixel between two images.  Throws an exception if dimesions of both
 images don't match
\*****************************************************************************/
ImageType& ImageType::operator+ ( const ImageType& rhs )
{
        // throw error if images don't match
```

```
        if ( N != rhs.N || M != rhs.M || Q != rhs.Q )
                throw (string)"Images do not have the same dimensions!";

        // this is the value that determines the weight of each image
        // large a gives more weight to first image
        // small a gives more weight to second image
        // a must be 0 >= a <= 1
        float a = 0.5;

        //the general formula is aI1(r,c)+(1-a)I2(r,c)
        for(int i = 0; i < N; i++)
                for(int j = 0; j < M; j++)
                        // set new pixel value
                        pixelValue[i][j] = pixelValue[i][j]*a+(1.0-a)*rhs.pixelValue[i][j];

        return *this;   // return current object
}
```

9 imageIO.h

```
#ifndef IMAGE_IO
#define IMAGE_IO

#include "image.h"

// These are the functions used to read and write images to .pgm files I just
// added them to this header file to make linking easier to understand
//    -Josh

        // name : readImageHeader
        // input : cstring of filename, int N, M, Q and bool value to hold image data
        // output : sets values of image header to last 4 paramters
        // dependencies : image.h
        void readImageHeader( const char[], int&, int&, int&, bool& );

        // name : readImage
        // input : cstring of filename, and ImageType object to hold image data
        // output : set image info to the ImageType object
        // dependencies : image.h
        void readImage( const char[], ImageType& );

        // name : writeImage
        // input : cstring of filename, and ImageType object to be store in file
        // output : writes a pgm type file with ImageType stored as a RAW form
        // dependencies : image.h
        void writeImage( const char[], ImageType& );

#endif
```

10 imageIO.cpp

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <stdio.h>
```

```
using namespace std;

#include "image.h"

void readImage(const char fname[], ImageType& image)
{
        int i, j;
        int N, M, Q;
        unsigned char *charImage;
        char header [100], *ptr;
        string msg;
        ifstream ifp;

        ifp.open( fname, ios::in | ios::binary );

        if ( !ifp )
        {
                msg = "Can't read image: ";
                msg += fname;
                throw msg;
        }

        // read header

        ifp.getline( header, 100, '\n' );
        if ( ( header[0] != 80 ) ||     /* 'P' */
             ( header[1] != 53 ) ) {    /* '5' */
                msg = "Image ";
                msg += fname;
                msg += " is not PGM";
                throw msg;
        }

        ifp.getline( header, 100, '\n' );

        while( header[0] == '#' )
                ifp.getline( header, 100, '\n');

        M = strtol( header, &ptr, 0 );
        N = atoi( ptr );

        ifp.getline( header, 100, '\n' );
        Q = strtol( header, &ptr, 0 );

        charImage = (unsigned char *) new unsigned char [M*N];

        ifp.read( reinterpret_cast<char *>(charImage), (M*N)*sizeof(unsigned char));

        if ( ifp.fail() )
        {
                msg = "Image ";
                msg += fname;
                msg += " has wrong size";
                throw msg;
        }
```

```
        ifp.close();

        //
        // Convert the unsigned characters to integers
        //

        int val;

        for(i=0; i<N; i++)
                for(j=0; j<M; j++)
                {
                        val = (int)charImage[i*M+j];
                        image.setPixelVal(i, j, val);
                }

        delete [] charImage;
}

void readImageHeader(const char fname[], int& N, int& M, int& Q, bool& type)
{
        int i, j;
        unsigned char *charImage;
        char header [100], *ptr;
        string msg;
        ifstream ifp;

        ifp.open( fname, ios::in | ios::binary );

        if ( !ifp )
        {
                msg = "Can't read image: ";
                msg += fname;
                throw msg;
        }

        // read header

        type = false; // PGM

        ifp.getline( header, 100, '\n' );
        if ( (header[0] == 80) &&  /* 'P' */
             (header[1] == 53) )   /* '5' */
        {
                type = false;
        }
        else if ( (header[0] == 80) &&  /* 'P' */
                  (header[1] == 54) )   /* '6' */
        {
                type = true;
        }
        else
        {
                msg = "Image ";
                msg += fname;
                msg += " is not PGM or PPM";
```

```
                throw msg;
        }

        ifp.getline( header, 100, '\n');
        while( header[0] == '#' )
                ifp.getline( header, 100, '\n' );

        M = strtol( header, &ptr, 0 );
        N = atoi( ptr );

        ifp.getline( header, 100, '\n' );

        Q = strtol( header, &ptr, 0 );

        ifp.close();

}

void writeImage(const char fname[], ImageType& image)
{
        int i, j;
        int N, M, Q;
        unsigned char *charImage;
        string msg;
        ofstream ofp;

        image.getImageInfo( N, M, Q );

        charImage = (unsigned char *) new unsigned char [M*N];

        // convert the integer values to unsigned char

        int val;

        for( i=0; i<N; i++ )
                for( j=0; j<M; j++ )
                {
                        val = image.getPixelVal(i, j);
                        charImage[i*M+j] = (unsigned char)val;
                }

        ofp.open( fname, ios::out | ios::binary );

        if ( !ofp )
        {
                msg = "Can't open file: ";
                msg += fname;
                throw msg;
        }

        ofp << "P5" << endl;
        ofp << M << " " << N << endl;
        ofp << Q << endl;

        ofp.write( reinterpret_cast<char *>(charImage), (M*N)*sizeof(unsigned char) );
```

```
        if ( ofp.fail() )
        {
                msg = "Can't write image ";
                msg += fname;
                throw msg;
        }

        ofp.close();

        delete [] charImage;
}
```