```cpp
/*****************************************************************************************
*                                    EXERCISE 3                                          *
*                                   ------------                                         *
*                                    FILTERING                                           *
*****************************************************************************************/

#include <opencv2/opencv.hpp>
#include "Ex3-CV-header.h"
#include <iostream>
#include <string>
#include <stdlib.h>
#include <math.h>

using namespace cv;
using namespace std;



/****************************************************************************************
* Filter an image using the function cvFilter2D and the kernel specified to emphasize   *
* horizontal gradients.                                                                 *
****************************************************************************************/

void xGradientFilter(IplImage * original , IplImage * x_filtered) {
    /*float x_mat [3][3] = { { -1 , 0, 1},
                             { -2 , 0, 2},
                             { -1 , 0, 1}  }; */
    CvMat x_kernel = cvMat(3 , 3 , CV_32F , x_mat);
    cvFilter2D(original, x_filtered , &x_kernel , cvPoint(-1,-1));
}



/****************************************************************************************
* Filter an image using the function cvFilter2D and the kernel specified to emphasize   *
* vertical gradients.                                                                   *
****************************************************************************************/

void yGradientFilter(IplImage * original , IplImage * y_filtered) {
    /*float y_mat [3][3] = { { -1, -2, -1},
                             {  0,  0,  0},
                             {  1,  2,  1}  }; */

    CvMat y_kernel = cvMat(3 , 3 , CV_32F , y_mat);
    cvFilter2D(original , y_filtered , &y_kernel , cvPoint(-1,-1));
}
```

```cpp
/****************************************************************************************
* Streams in 2 different windows the input from a webcam. One window shows the stream   *
* from the camera directly. The second window shows the processed stream.               *
* The kernel of the filter to be applied is passed as a pointer to  a CvMat so different*
* types of filtering are possible.                                                      *
* The function returns 0 if everything goes well, -1 otherwise.                         *
****************************************************************************************/

int doubleStream(CvMat * kernel) {
    VideoCapture cap(1);

    Mat frame, tframe, frame_grey;

       if (!cap.isOpened()) {
        std::cout << "Cam could not be accessed" << std::endl;
        return -1;
       }

       namedWindow("Cam");
       cvNamedWindow("LG_Cam", 0);

       while(cap.read( frame )) {
        GaussianBlur(frame , tframe , Size(3 , 3) , 0 , 0 , BORDER_DEFAULT);
        cvtColor(tframe , frame_grey , CV_BGR2GRAY);
        cvtColor(frame  , frame_grey , CV_BGR2GRAY);
        IplImage * LG_im_cam = new IplImage(frame_grey);
        cvFilter2D(LG_im_cam , LG_im_cam , kernel , cvPoint(-1,-1) );
        imshow("Cam" , frame);
        cvShowImage("LG_Cam" , LG_im_cam);

        if (waitKey(10) >= 0) {
            imwrite("Cam.png" , frame);
            cvSaveImage("LG_Cam.png" , LG_im_cam);
            break;
        }

        if(frame.empty()) {
            std::cout << "End of stream" << std::endl;
            break;
        }
       }
       cvDestroyAllWindows();

       return 0;
}
```

```
/****************************************************************************************
* Starting from an image this function applies a percentile filter.                     *
* The percentage is passed as a parameter. A 3x3 window is assumed.                     *
****************************************************************************************/

/// prototypes of the supporting functions used by the following

void MtoV(char *, int, uchar []);
int cmpfunc(const void *, const void *);

void fractileFilter(IplImage * src , IplImage * dest , int p) {
    uchar v[9];
    int percentile = 9 * p / 100;

    for(int i = 0 ; i < dest->height ; i++) {

        char * psrc  = src->imageData  + i * src->widthStep;
        char * pdest = dest->imageData + i * dest->widthStep;

        for(int j = 0 ; j < dest->width ; j++) {
            MtoV(psrc , src->widthStep , v);                // pixels within window in an array
            qsort(v , 9 , sizeof(uchar) , cmpfunc);         // Quicksort
            *pdest = v[percentile];
            psrc++;
            pdest++;
        }
    }
}



/****************************************************************************************
* Comparing function used in the call to 'qsort' in 'fractileFilter'.                   *
****************************************************************************************/

int cmpfunc(const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}




/****************************************************************************************
* starting from a 3x3 ROI of a IplImage (so of type char), this function creates a      *
* 9-elements array.                                                                     *
*                                                                                       *
* NB. In this context the order of the elements is not relevant.                        *
****************************************************************************************/

void MtoV(char * src , int step , uchar v[]) {
    for(int i = 0 ; i < 3 ; i++) {

        char * psrc = src + i * step;

        for(int j = 0 ; j < 3 ; j++) {
            v[3 * i + j] = (uchar)(*psrc);
            psrc++;
        }
    }
}
```

```cpp
/******************************************************************************************
* Receives pointers to the source and destination images and produces the filtered image  *
* in the destination.                                                                     *
*                                                                                         *
* It uses a "frame" images to deal with the border of the image in a not too complicated  *
* way. The frame is a black image whose dimensions are such that there is a 1-pixel frame *
* around the source image.                                                                *
*                                                                                         *
* When filtering one of the 4 corner pixels, 5 pixels out of 9 from the Kernel stick out  *
* of the image and go over the "frame". These will not contribute to the filtering, but   *
* the divisor must be changed to 4 (useful pixels in the Kernel) instead of 9. There is   *
* no need to know which specific corner pixel is considered, it works in general!         *
*                                                                                         *
* Similarly, for border pixels (not in the corners), 3 pixels in the Kernel stick out of  *
* the image and 6 must be used as divider for filtering.                                  *
*                                                                                         *
* IMPORTANT: Most of the pixels are of course in the inner part of the image, so that     *
* condition must be checked first and it will avoid other conditions to be checked all    *
* the time for nothing. This improves the                                                 *
* For the same reason, it would be better to check for the borders before checking for    *
* the corners, but the condition for the borders is very long and hard to read.           *
******************************************************************************************/

#define INNER  9
#define CORNER 4
#define SIDE   6

void lowPass(IplImage * src , IplImage * dest) {
    /// Container set to black (Frame)
    IplImage * temp = cvCreateImage(cvSize(src->width + 2, src->height + 2), IPL_DEPTH_8U, 1);
    cvSet(temp, 0);

    /// Image "framed"
    cvSetImageROI(temp , cvRect(1, 1, src->width , src->height) );
    cvCopy(src , temp);
    cvResetImageROI(temp);

    int N = dest->height;
    int M = dest->width;

    for(int i = 0 ; i < N ; i++) {
        char * pdest = dest->imageData + i * dest->widthStep;
        char * ptemp = temp->imageData + i * temp->widthStep;
        for(int j = 0 ; j < M ; j++) {
            if ((i > 0 && i < N - 1) && (j > 0 && j < M - 1))             // Inner Image
                func(ptemp, pdest, temp->widthStep, INNER);
            else if ((i == 0 || i == N - 1) && (j == 0 || j == M - 1))    // Corners
                func(ptemp, pdest, temp->widthStep, CORNER);
            else                                                           // Borders
                func(ptemp, pdest, temp->widthStep, SIDE);
            pdest++;
            ptemp++;
        }
    }
    cvReleaseImage(&temp);
}
```

```cpp
/****************************************************************************************
* Support function for 'lowPass'. It does the dirty job!!!                              *
* Takes 2 pointers to the source (framed) and destination images, the 'widthStep' value *
* for the framed image to scan it and a divider 'dk' that takes 3 possible values:      *
* 9 (inner pixels), 6 (border pixels), and 4 (corner pixels).                           *
* It produces the value of a single pixel of the final image, pointed to by 'pd', by    *
* averaging over the Kernel and taking into account the rim.                            *
****************************************************************************************/

void func(char * pf , char * pd , int step , short dk) {
    float k = 1.0 / dk;
    *pd = 0;                                              // pixel in the final image.
    for(int i = 0 ; i < 3 ; i++) {

        char * lpf = pf + i * step;                       // Local pointer to framed image

        for(int j = 0 ; j < 3 ; j++) {
            /// After many tests I found that this sequence of casting keeps the correct
            /// result at the end...
            *pd += (uchar)(k * (float)((uchar)(*lpf)));
            lpf++;
        }
    }
}





//////////////////////////////////////////////////////////////////////////////////////////




/*****************************************************************************************
* The following group of functions do exactly the same thing as the combination          *
* 'lowPass' + 'func' above.                                                              *
* It was intended to be more efficient since every function does only what is strictly   *
* needed for a specific task, avoiding unnecessary computation. It's not as easy to read *
* and it turned out to be just as efficient as above in terms of speed.                  *
* The nice thing, and the main reason why I'm leaving it here is that it's more efficient*
* in terms of memory requirements since it does not need a framed intermediate image.    *
*****************************************************************************************/
/*
void lowPass(IplImage * src , IplImage * dest) {
    int N = dest->height;
    int M = dest->width;
    for(int i = 0 ; i < N ; i++) {
        char * p  = dest->imageData + i * dest->widthStep;
        char * pp = src->imageData  + i * src->widthStep;
        for(int j = 0 ; j < M ; j++) {
            if ((i > 0 && i < N - 1) && (j > 0 && j < M - 1))            // Inner image
                func(pp, p, src->widthStep);
            else if (j == 0 && i > 0 && i < N - 1)                       // left border
                func_3((pp - src->widthStep), p, src->widthStep);
            else if (j == M - 1 && i > 0 && i < N - 1)                   // right border
                func_3((pp - src->widthStep - 1), p, src->widthStep);
            else if (i == 0 && j > 0 && j < M - 1)                       // top border
                func_2((pp - 1), p, src->widthStep);
            else if (i == N - 1 && j > 0 && j < M- 1)                    // bottom border
                func_2((pp - src->widthStep - 1), p, src->widthStep);
            else if (i == 0 && j == 0)                                   // top-left
                func_1(pp, p, src->widthStep);
            else if (i == 0 && j == M - 1)                               // top-right
                func_1((pp - 1), p, src->widthStep);
            else if (i == N - 1 && j == 0)                               // bottom-left
                func_1((pp -  src->widthStep), p, src->widthStep);
            else if (i == N - 1 && j == M - 1)                           // bottom-right
                func_1((pp - src->widthStep - 1), p, src->widthStep);
            p++;
            pp++;
        }
    }
}



void func(char * ps, char * pd, int step) {
    float k = 1.0 / 9;
    *pd = 0;
    char * lps = ps - step - 1;
    for(int i = 0 ; i < 3 ; i++) {
        lps += i * step;
        for(int j = 0 ; j < 3 ; j++) {
            *pd += (uchar)(k * (float)((uchar)(*lps)));
            lps++;
        }
    }
}


void func_1(char * pp, char * p, int step) {
    *p = 0;
    *p += (uchar)((((float)((uchar)(*pp))) + ((float)((uchar)(*(pp + 1)))) +
          ((float)((uchar)(*(pp + step)))) + ((float)((uchar)(*(pp + step + 1))))) / 4);
}


void func_2(char * pp, char * p, int step) {
    *p = 0;
    *p += (uchar)(((float)((uchar)(*pp)) + (float)((uchar)(*(pp + 1))) +
          (float)((uchar)(*(p + 2)))) / 6);
    pp += step;
    *p += (uchar)(((float)((uchar)(*pp)) + (float)((uchar)(*(pp + 1))) +
          (float)((uchar)(*(p + 2)))) / 6);
}


void func_3(char * pp, char * p, int step) {
    *p = 0;
    for(int i = 0 ; i < 3 ; i++) {
        *p += (uchar)(((float)((uchar)(*pp)) + (float)((uchar)(*(pp + 1)))) / 6);
        p += i*step;
    }
}
//*/
```