```cpp
#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include <opencv2/opencv.hpp>

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

using namespace cv;

//courtesy of stack overflow :)
namespace patch { //std not registering as namespace for to_string(int)
    template <typename T> std::string to_string(const T& n) {
        std::ostringstream stm;
        stm << n;
        return stm.str();
    }
}

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

bool is_grayscale(Mat);

Mat get_gaussian_kernel(int, int, double, double);

void color2grey(Mat*);

void downsample(int, double, double, Mat*);

void quantize(int, Mat*);

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

int main(int argc, char** argv) {
    Mat image;
    image = imread(argv[1], 1);
    if(!image.data) {
        std::cout<< "No image data \n";
        std::cin.get(); //wait for key press
        return -1;
    }



    namedWindow("Lenna in Color", WINDOW_AUTOSIZE);
    imshow("Lenna in Color", image);
    waitKey(0);
```

```
    color2grey(&image);
    imwrite("Lenna_grayscale.jpg", image); //save grayscale to file

    namedWindow("Lenna in Grayscale", WINDOW_AUTOSIZE);
    imshow("Lenna in Grayscale", image);
    waitKey(0);

    //base of pyramid (not downsampled) is "Lenna_grayscale.png"
    Mat pyramid = image.clone();
    for(int i = 0; i < 3; i++) { //create image pyramid
        std::string file_name = "pyramid_" + patch::to_string(i+1) + ".png";
        std::cout << file_name << std::endl;

        int kernel_size = 3; //use n = 3 for 3x3 kernel
        double sigma = 6.0; //sigx = sigy = 9 are the standard deviations
        downsample(kernel_size, sigma, sigma, &pyramid);
        imwrite(file_name, pyramid); //save grayscale to file

        namedWindow("Pyramid" + patch::to_string(i+1), WINDOW_AUTOSIZE);
        imshow("Pyramid" + patch::to_string(i+1), pyramid);
        waitKey(0);
    }

    int quantization_factor = 4;
    quantize(quantization_factor, &image);
    imwrite("quantized.png", image); //save grayscale to file
    std::cout << "quantized.png" << std::endl;

    namedWindow("Quantized Lenna", WINDOW_AUTOSIZE);
    imshow("Quantized Lenna", image);
    waitKey(0);

    return 0;
}

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

bool is_grayscale(Mat image) {
    if(image.type() == CV_8UC1) { //CV_8UC1 is enumerated type for 8 bit single
channel unsigned matrix
        return true;
    } else {
        return false;
    }
}
```

```
Mat get_gaussian_kernel(int rows, int cols, double sigmax, double sigmay) {
   const int y_mid = rows / 2;
   const int x_mid = cols / 2;

   const double x_spread = 1. / (sigmax*sigmax*2);

   std::vector<float> gauss_x, gauss_y;

   gauss_x.reserve(cols);
   for(int i = 0;  i < cols;  ++i) {
      double x = i - x_mid;
      gauss_x.push_back(std::exp(-x*x*x_spread));
   }

   double n_rows=x_mid-y_mid;
   float sum=0;
   Mat kernel = Mat::zeros(rows, cols, CV_32FC1); //matrix of 32 bit floats
   for(int j = 0;  j < rows;  ++j) {
      float temp = gauss_x[n_rows+j];
      for (int i = 0;  i < cols;  ++i) {
         kernel.at<float>(j,i) = gauss_x[i] * temp;
         sum += kernel.at<float>(j,i);
      }
   }

   return kernel / sum;
}

void color2grey(Mat* image) { //using luminosity method
   Mat grayscale = Mat(image->rows, image->cols, CV_8UC1); //Mat constructor
   for(int r = 0; r < image->rows; r++) {
      for(int c = 0; c < image->cols; c++) { //Each pixel is an array of 3. We weight each
color based on human eye sensitivity.
         int tmp = (image->at<Vec3b>(r, c)[0] * .11) + (image->at<Vec3b>(r, c)[1] * .59)+
(image->at<Vec3b>(r, c)[2] * .33);
         grayscale.at<uchar>(r,c) = tmp;
      }
   }
   *image = grayscale;
}

void downsample(int n, double sigx, double sigy, Mat* image) { //outputs a greyscaled
and downsampled image
if(!is_grayscale(*image)) { //if grayscale
      color2grey(image);
   }
```

```
   Mat gaussian = get_gaussian_kernel(n, n, sigx, sigy);

   //filter2D(input, output, depth(neg = 0), kernel(Mat float), anchor((-1,-1) = center),
delta, border)
   Mat filtered;
   filter2D(*image, filtered, -1, gaussian, Point(-1,-1), 0, BORDER_DEFAULT); //
convolution

   //subsampling
   Mat downsampled(Size(image->rows/2, image->cols/2), CV_8UC1);
   for(int r = 0; r < image->rows/2; r++) { //using an averaging filter
      for(int c = 0; c < image->cols/2; c++) {
         downsampled.at<uchar>(r,c) = (filtered.at<uchar>(2*r,2*c) +
filtered.at<uchar>(2*r,2*c+1) + filtered.at<uchar>(2*r+1,2*c) +
filtered.at<uchar>(2*r+1,2*c+1))/4;
      }
   }

   *image = downsampled;
}

void quantize(int nlevels, Mat* image) { //integer division is easier bit kmeans produces
a better result
   if(!is_grayscale(*image)) { //if grayscale
      color2grey(image);
   }

   nlevels = 256/nlevels; //creates scale factor
   for(int r = 0; r < image->rows; r++) {
      for(int c = 0; c < image->cols; c++) { //formula is I = (I/nlevels)*nlevels
         image->at<uchar>(r,c) /= nlevels; //integer division rounds to nearest integer
         image->at<uchar>(r,c) *= nlevels;
      }
   }
}
```