TP2 This exercise aims to focus on first steps in image and video manipulations exploiting OpenCV functions. You will find theoretical explanation and code to complete.

1. **Load, Modify and Save an Image**
   In Lesson 4 you studied how to load and visualise an image in OpenCV. If you don't remember please check again this part before following in. Now let's see a simple manipulation and how you can save a new image. *Read following explanation and then complete the code save_grey_students.cpp.*

   - You have at first to load an image using `cv::imread` , located in the path given by imageName. We assume you are loading a BGR image (as HappyFish.jpg you find in moodle).
   - To convert our image from BGR to Grayscale format, OpenCV has a really nice function: `cvtColor( image, gray_image, COLOR_BGR2GRAY );`
     As you can see, `cv::cvtColor` takes as arguments:
     - a source image (image);
     - a destination image (gray_image), in which we will save the converted image;
     - an additional parameter that indicates what kind of transformation will be performed. In this case we use `COLOR_BGR2GRAY` (because of `cv::imread` has BGR default channel order in case of color images).
   - So now we have our new *gray_image* and want to save it on disk (otherwise it will get lost after the program ends). To save it, we will use a function analogous to `cv::imread` that is `cv::imwrite (filename, image_object)`, which takes as arguments:
     - the name of the file with its exention;
     - the image you want to save.
   - Finally, let's check out the images by creating two windows (using namedWindow) and use them to show the original image as well as the new one (using imshow);
   - and add the waitKey(0) function call for the program to wait forever for an user key press.

2. **Cropping and resizing an image**

   In OpenCV, we can resize images. We can both shrink and enlarge images. Images are shrunk to perform certain operations on it and then once the operations are done they are enlarged back. Shrinking also helps to gain computational speed. We can also crop out images. This helps in selective computation as we do not need to process the entire image.
   Read following explanation and then complete the code cropAndResize_students.cpp.
   After reading the image, we define two scale values and then resize the image using the function `cv::resize(src, dst, dsize, fx, fy, interpolation method)`, which takes these arguments:

   - src is the source image
   - dst is the destination image
   - dsize is the output image size; if it equals zero, it is computed as: `dsize = Size(round(fx*src.cols), round(fy*src.rows))`. Either dsize or both fx and fy must be non-zero.
   - fx is scale factor along the horizontal axis; when it equals 0, it is computed as `(double)dsize.width/src.cols`
   - fy is scale factor along the vertical axis; when it equals 0, it is computed as `(double)dsize.height/src.rows`
   - it's possible to choose different interpolation method:
     - INTER_NEAREST a nearest-neighbor interpolation.

  - INTER_LINEAR a bilinear interpolation (used by default).
  - INTER_AREA resampling using pixel area relation. It may be a preferred method for image decimation, as it gives moirè-free results. But when the image is zoomed, it is similar to the INTER_NEAREST method.
  - INTER_CUBIC a bicubic interpolation over 4x4 pixel neighborhood.
  - INTER_LANCZOS4 a Lanczos interpolation over 8x8 pixel neighborhood.

We will use INTER_LINEAR as interpolation method.
Then, for cropping we use the `cv::Range((int_start, int_end)`, function of OpenCV and then first specify the rows and then the columns of the cropping region. The `cv::Range`is a template class which specify a continuous subsequence (slice) of a sequence and so in our case to define a row or a column span in a matrix (Mat) that is our image. Then we want to visualise our four images to conclude the program.

3. **Rotating an image**

In the *rotate_student.cpp* program we want now to rotate our *van_gogh.jpg* image.

In OpenCV, we can rotate images about any point (let's call it center) and by an angle (let's simply call this angle). A rotation is represented as a 2x3 Matrix because rotations belong to the special class of transforms called the Affine Transform.

The `getRotationMatrix2D(center, angle, scale)` function is used to get the matrix which will define the rotation. Its parameters are :

- center is the center of the rotation in the source image.
- angle is the rotation angle in degrees. Positive values mean counter-clockwise rotation (the coordinate origin is assumed to be the top-left corner).
- scale is the isotropic scale factor.

The getRotationMatrix2D returns the following matrix :

$$\begin{bmatrix} \alpha & \beta & (1-\alpha)\cdot \texttt{center.x} - \beta \cdot \texttt{center.y} \\ -\beta & \alpha & \beta \cdot \texttt{center.x} + (1-\alpha)\cdot \texttt{center.y} \end{bmatrix}$$

where

$$\alpha = \texttt{scale} \cdot \cos \texttt{angle}, \\ \beta = \texttt{scale} \cdot \sin \texttt{angle}$$

The matrix obtained using `getRotationMatrix2D` can be used with `cv::warpAffine( src, warp_dst, warp_mat, warp_dst.size())` to get the final rotated image. Remember rotation is a special case of the affine transform and therefore the function is called warpAffine. The `cv::warpAffine` has the following arguments:

- src is the input image;
- warp_dst is the output image;
- warp_mat is the affine transform, in this case our rotation matrix;
- warp_dst.size() is the desired size of the output image.

Complete the program rotate_students.cpp that has to show the original and the rotated image.

4. **Reading and Displaying a Video**

   In OpenCV, a video can be read either by using the feed from a camera connected to a computer or by reading a video file. The first step towards reading a video file is to create a `VideoCapture` object. Its argument can be either the device index or the name of the video file to be read. In most cases, only one camera is connected to the system. So, all we do is pass '0' and OpenCV uses the only camera attached to the computer. When more than one camera is connected to the computer, we can select the second camera by passing '1', the third camera by passing '2' and so on. After the VideoCapture object is created, we can capture the video frame by frame.

   A frame of a video is simply an image and we display each frame the same way we display images, i.e., we use the function `imshow()`. As in the case of an image, we use the `waitKey()` after `imshow()` function to pause each frame in the video. In the case of an image, we pass '0' to the `waitKey()` function, but for playing a video, we need to pass a number greater than '0' to the `waitKey()` function. This is because '0' would pause the frame in the video for an infinite amount of time and in a video we need each frame to be shown only for some finite interval of time, so we need to pass a number greater than '0' to the `waitKey()` function. This number is equal to the time in milliseconds we want each frame to be displayed.

   While reading the frames from a webcam, using `waitKey(1)` is appropriate because the display frame rate will be limited by the frame rate of the webcam even if we specify a delay of 1 ms in `waitKey()`.

   While reading frames from a video that you are processing, it may still be appropriate to set the time delay to 1 ms so that the thread is freed up to do the processing we want to do.

   In rare cases, when the playback needs to be at a certain frame rate, we may want the delay to be higher than 1 ms.

   Complete the program 'VideoRead_students.cpp' and run it. Then, at home, try it with a camera.

5. **Writing a video**

   After we are done with capturing and processing the video frame by frame, the next step we would want to do is to save the video from a camera. We have seen that for images, it is straightforward: we just need to use `cv::imwrite()`. But for videos, we need to toil a bit harder. We need to create a `cv::VideoWriter (filename, fourcc, fps, frameSize, isColor=true)` object. The `cv::VideoWriter` has the following parameters:

   - filename is the name of the output video file;
   - fourcc is -character code of codec used to compress the frames. For example, VideoWriter::fourcc ('P','I','M','1') is a MPEG-1 codec, VideoWriter::fourcc('M','J','P','G') is a motion-jpeg codec etc. List of codes can be obtained at Video Codecs by FOURCC page http://www.fourcc.org/codecs.php. FFMPEG backend with MP4 container natively uses other values as fourcc code: see ObjectType, so you may receive a warning message from OpenCV about fourcc code conversion;
   - fps is the frame rate of the created video stream;
   - frameSize is the size of the video frames;
   - If it is not zero, the encoder will expect and encode color frames, otherwise it will work with grayscale frames.

   At home, complete the code 'VideoWrite_students.cpp' and use your webcam when running it.