

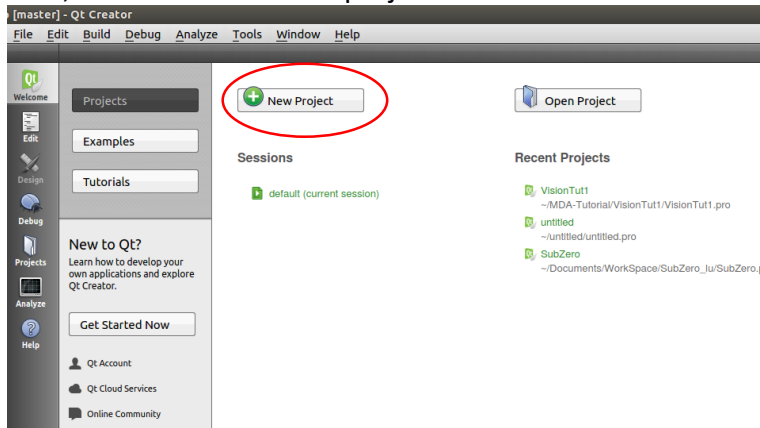
# MDA Vision Tutorial 1: Brightness and Contrast Control

This tutorial will teach you how to use OpenCV functions to change the brightness and contrast of an image. This is based off of OpenCV's tutorial

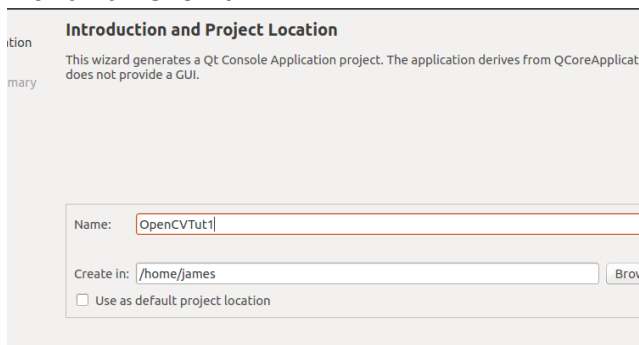
[http://docs.opencv.org/2.4/doc/tutorials/core/basic\\_linear\\_transform/basic\\_linear\\_transform.html](http://docs.opencv.org/2.4/doc/tutorials/core/basic_linear_transform/basic_linear_transform.html)

## Part 0: Setting up the project

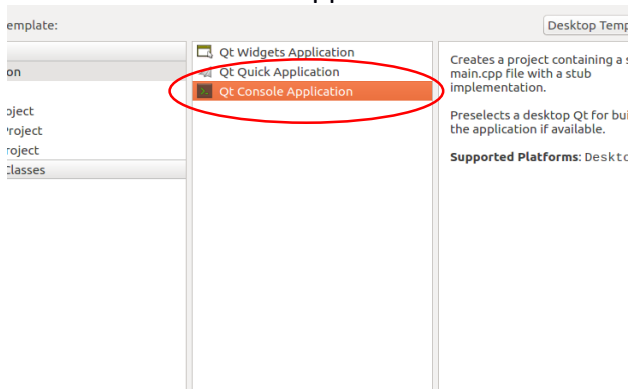
First, let's make a new Qt project



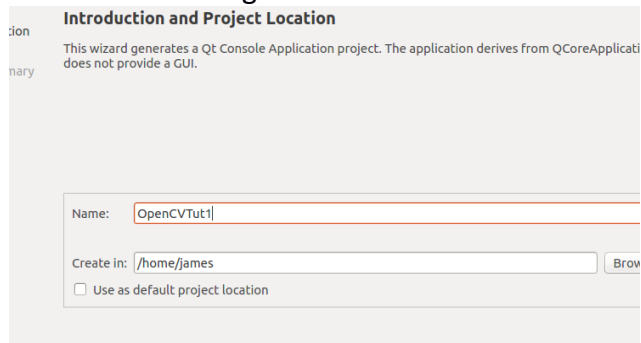
## Pick a name for it



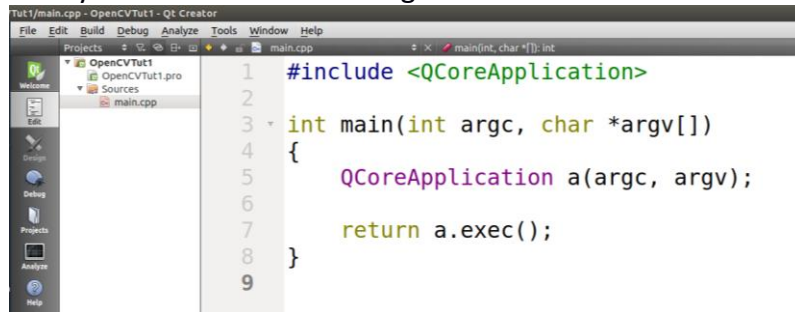
## Select the Qt Console Application



## Name it something

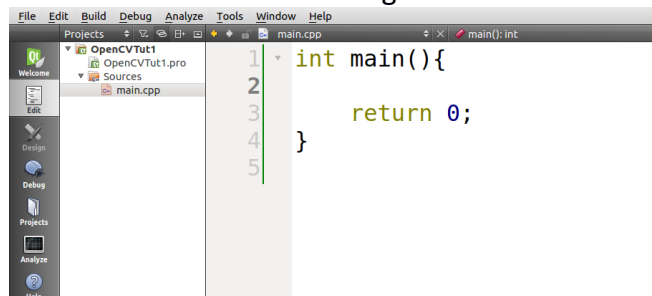


Now you should see something like this.

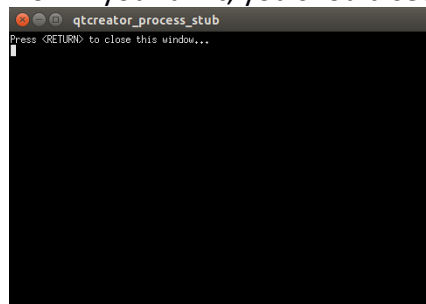


Since we are not using any Qt functions in this tutorial, you can delete all the pre-generated code

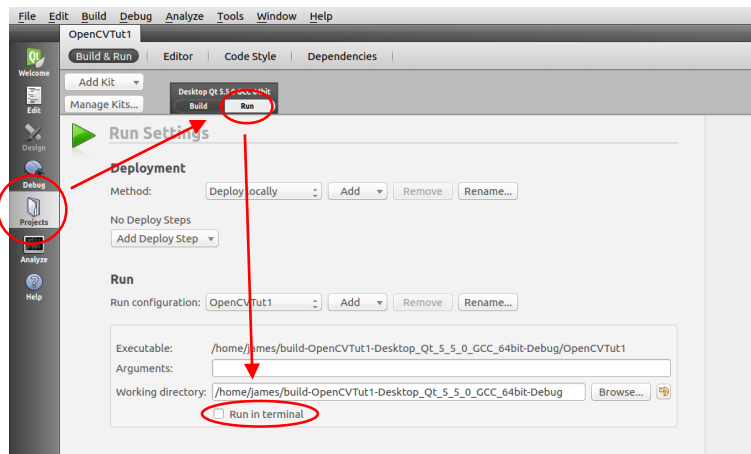
Now it should look something like this. I also added the return 0.



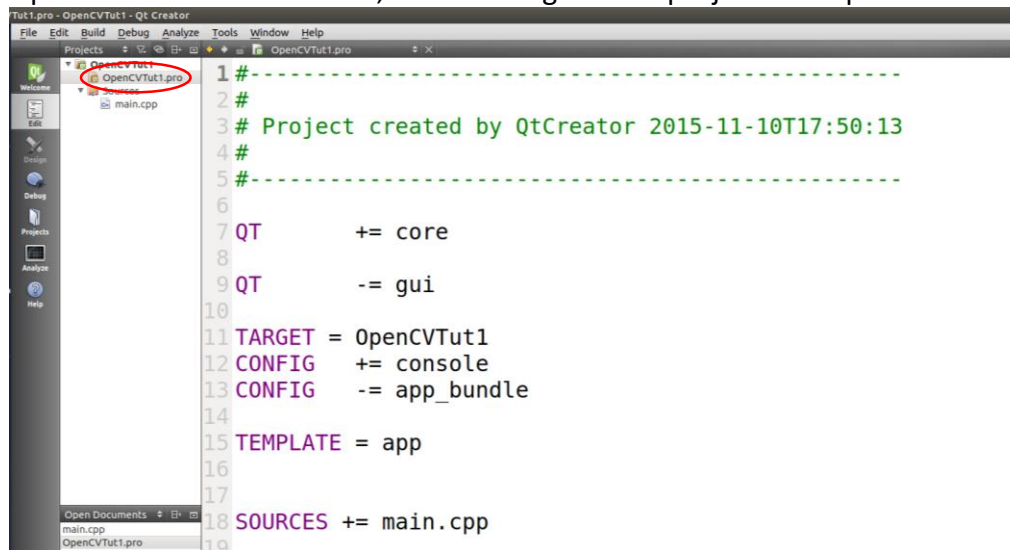
Now if you run it, you should see something like this



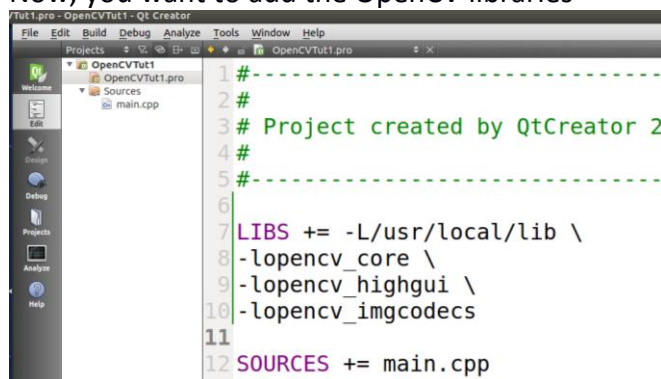
This step is optional, but since we are not going to use the console, we can go disable it. Go to Projects, and then click Run. Then you should uncheck this box.



Now, since we are going to use OpenCV functions, we want to tell the IDE (Qt in this case) to include OpenCV libraries. To do that, we have to go to the projectName.pro file. It will look like this.



Now, you want to add the OpenCV libraries



I've deleted a lot of the code here that Qt has automatically generated, because it's not necessary for now. If you get some errors at this step, then leave the pre-generated code as it is.

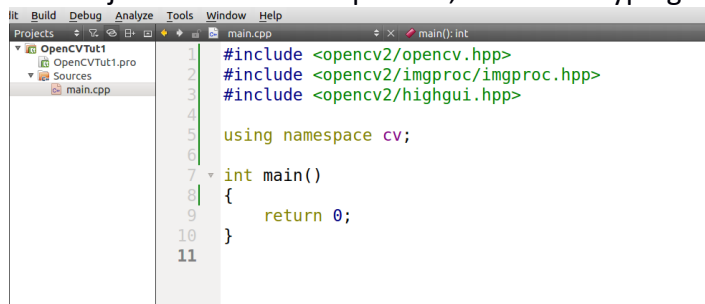
Now we are done setting up the project

## Part 1: Displaying an image

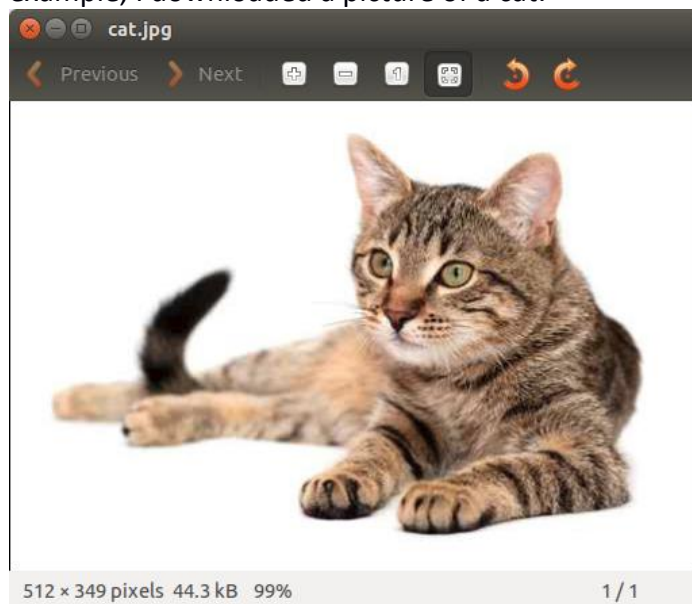
To use OpenCV functions, we need to tell the compiler where to find those functions. Since we have already added the OpenCV libraries to the project, we can just import it into our file. To do that, we use the `#include` keyword. In this example, these are the files we need to include

```
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui.hpp>
```

OpenCV functions use the namespace `cv`. This means that whenever you want to use their functions, you need to use the syntax `cv::function()`. Now since we are going to use their functions a lot, we can just use the namespace `cv`, instead of typing out `cv::function()` every time



Now, we would want to display an image. To do that, let's download an image from Google. In this example, I downloaded a picture of a cat.



In OpenCV, they store images in something called *Mats*.

“The class `Mat` represents an n-dimensional dense numerical single-channel or multi-channel array”  
[http://docs.opencv.org/2.4/modules/core/doc/basic\\_structures.html#mat](http://docs.opencv.org/2.4/modules/core/doc/basic_structures.html#mat)

So now we make a `Mat` called `image`

```
Mat image;
```

To load the image, OpenCV has a function called `imread()`, which takes in a string of the file name.

It returns a `Mat`. We store the returned `Mat` in the image variable we created earlier.

```
Mat image = imread("cat.jpg");
```

Now, if you try to run the code now it would crash. We will explain why below

To display the image onto the screen, we need to make a window to show the image. We do that using the function `namedWindow()`. It takes in two parameters, the first being a string of the name of the window. The second is the type of window it is. For our case, we want to call the window *Image*, and tell it to auto size itself.

```
namedWindow("Image", WINDOW_AUTOSIZE);
```

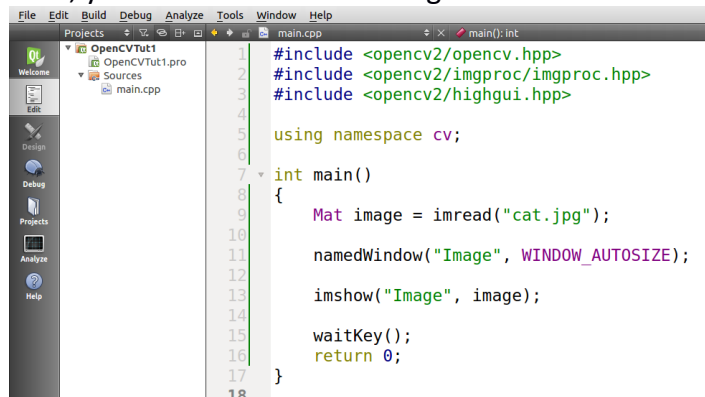
Now we show the image on the window we created. To do that, we use the `imshow()` function. The parameters are the name of the window that the image will be shown in and the image to show.

```
imshow("Image", image);
```

Now if you were to run it, it would execute the code and then exit the program (actually it would just say it cannot find the image/crash, but lets ignore that for now). This is because we don't tell the computer to wait after it shows the image. To the computer, it shows the image, and then shuts down immediately after.

To fix this, we need to tell the computer to wait. The easiest and most convenient way is to use the `waitKey()` function. This function pauses the program and waits for a keypress from the user.

Now, you should have something like this.

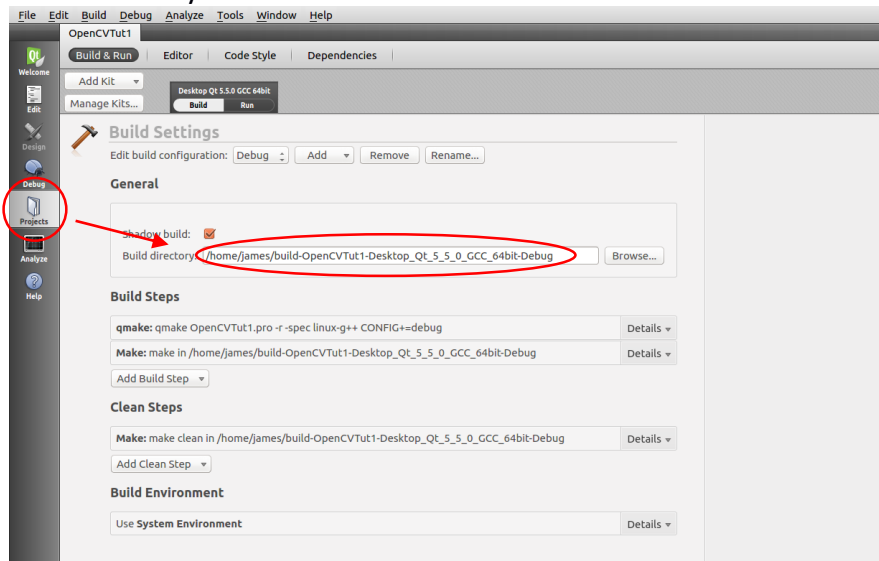


This will be all the code needed to show an image on the screen. However, if you run it, this happens.

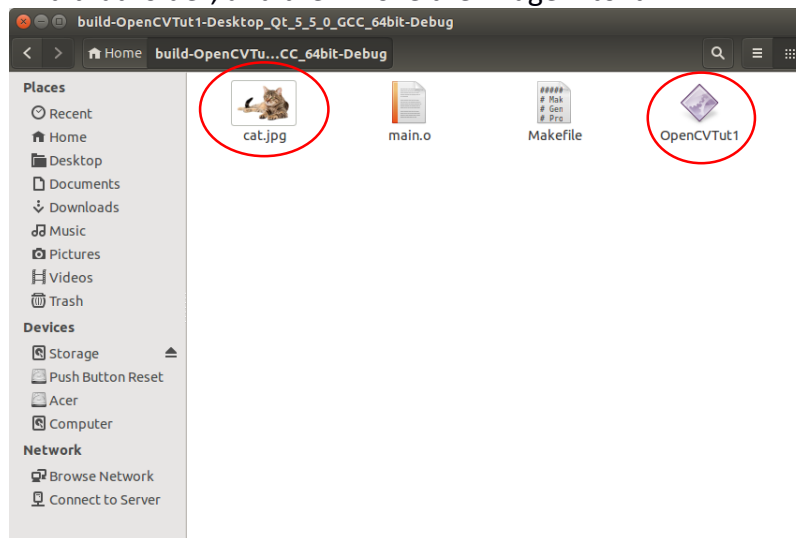


This is because the program cannot find the image you want. To fix this, we need to move the image

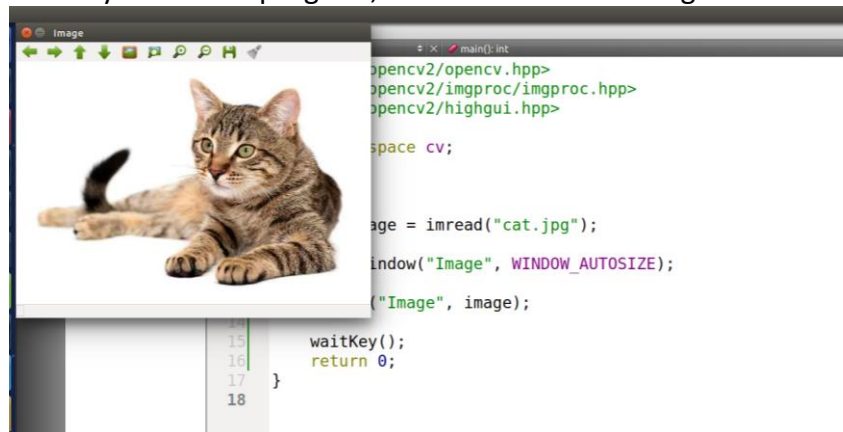
file to the same folder as the executable. To find where your executable is, go to projects and look for build directory



Find that folder, and then move the image into it.



Now if you run the program, it should show the image like so.



## Part 2: Editing the Image

In this section, we will edit the image by modifying the RGB values for every pixel.

First, we will need a new Mat to store the modified image

```
Mat modifiedImage = Mat::zeros(image.size(), image.type());
```

We use the `Mat::zeros()` function to make a black image (R, G, B = 0). We tell it to have the same size and type as the original image. The *size* is the width and height of the image, while the *type* is the depth of the image (8 bit, 16 bit, 32 bit image, etc).

We make a second window to show the image

```
namedWindow("Modified Image", WINDOW_AUTOSIZE);
```

Now, on the OpenCV website's tutorial, they use the formula

$$g(i,j) = \alpha \cdot f(i,j) + \beta$$

to change the brightness and contrast of an image. Where  $\alpha$  is the *gain* parameter, controlling contrast ( $0 < \alpha < 3$ ).  $\beta$  is the *bias* parameter, controlling brightness ( $\beta \in \mathbb{R}$ ).  $f(i,j)$  is the pixel at position  $(i,j)$ , and  $g(i,j)$  is the output pixel at position  $(i,j)$ . We need to make the alpha and beta parameters. Lets set alpha = 2 and beta = 50 for now.

```
float alpha = 2;  
int beta = 50;
```

We will have to edit each pixel, so we would need two for-loops. But since we are dealing with an RGB image, we also have to deal with the three channels individually. Thus, we need another for-loop

```
for (int y = 0; y < image.rows; y++){  
    for (int x = 0; x < image.cols; x++){  
        for (int c = 0; c < 3; c++){  
  
        }  
    }  
}
```

To get the pixel at position  $(y, x)$ , we use the syntax

```
modifiedImage.at<Vec3b>(y, x) [c]
```

where  $y$  is the row,  $x$  is the column, and  $c$  is the channel (R = 0, G = 1, B = 2).

Now we apply the formula

```
modifiedImage.at<Vec3b>(y, x) [c] = alpha*( image.at<Vec3b>(y, x) [c] ) +  
beta;
```

Since the formula we use can result in values out of range, we use OpenCV's `saturate_cast` function.

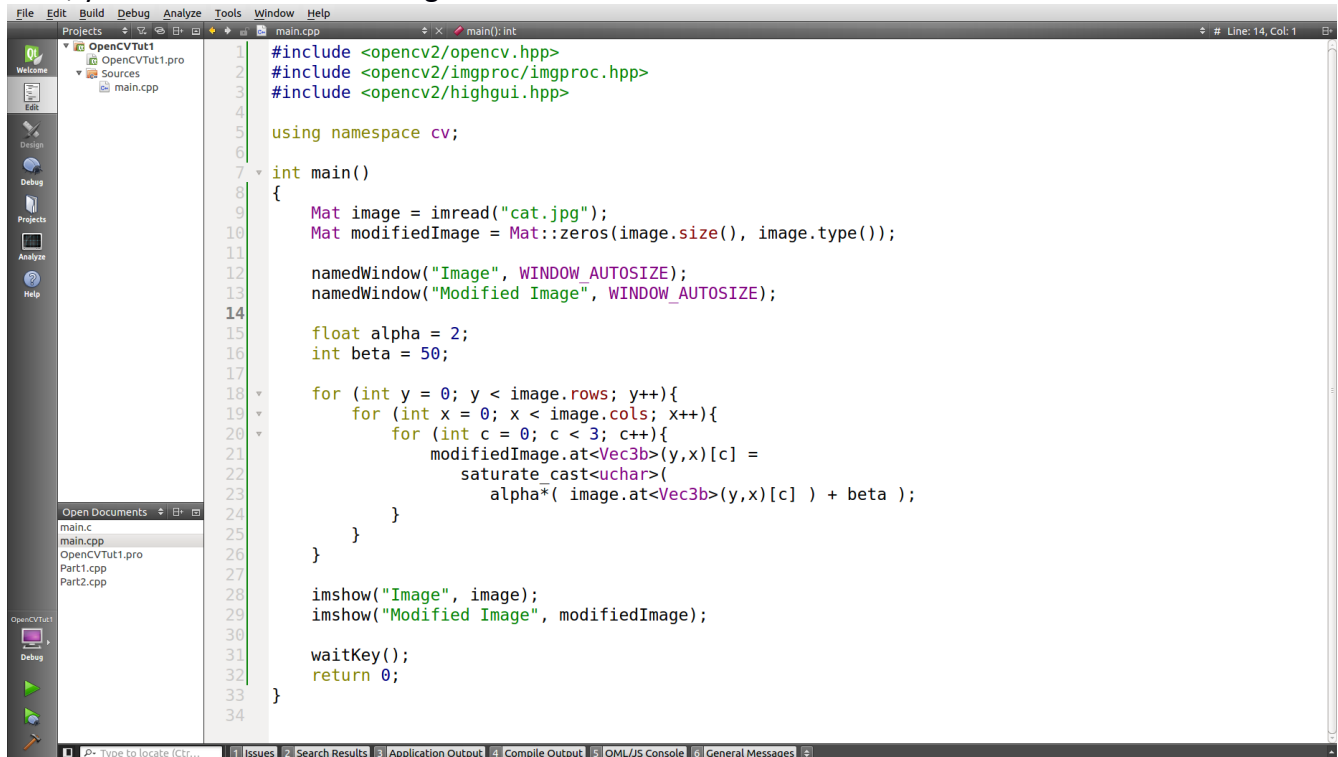
What `saturate_cast` does is that it converts one type into another, just like any ordinary cast.

However, it also cuts off values out of range. For example, 2,147,483,647 is the maximum value of an int. If you use `saturate_cast<int>(x)`, where  $x > 2,147,483,647$  (ie 9,999,999,999), it will return 2,147,483,647.

```
modifiedImage.at<Vec3b>(y,x)[c] =
    saturate_cast<uchar>( alpha*( image.at<Vec3b>(y,x)[c] ) + beta);
```

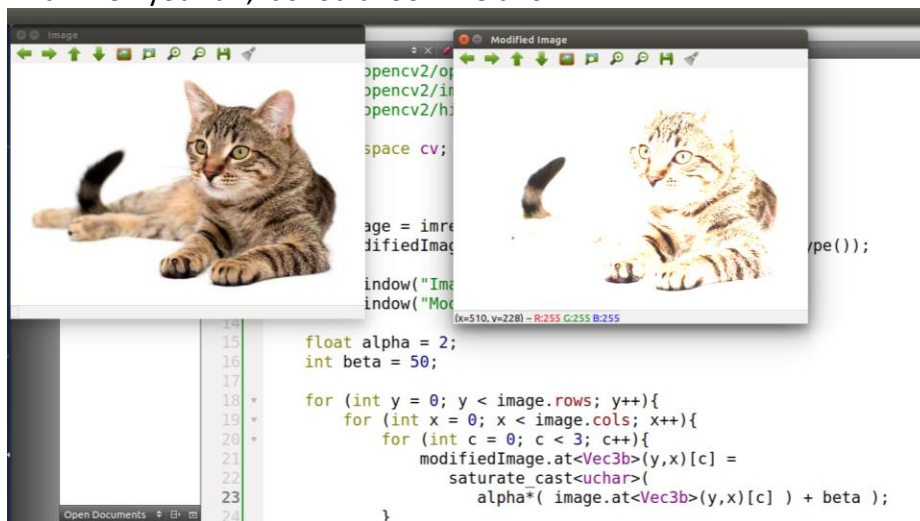
We use imshow again to show the modified image  
 imshow("Modified Image", modifiedImage);

Now, you should have something like this



```
File Edit Build Debug Analyze Tools Window Help
Projects Sources main.cpp
1 #include <opencv2/opencv.hpp>
2 #include <opencv2/imgproc/imgproc.hpp>
3 #include <opencv2/highgui.hpp>
4
5 using namespace cv;
6
7 int main()
8 {
9     Mat image = imread("cat.jpg");
10    Mat modifiedImage = Mat::zeros(image.size(), image.type());
11
12    namedWindow("Image", WINDOW_AUTOSIZE);
13    namedWindow("Modified Image", WINDOW_AUTOSIZE);
14
15    float alpha = 2;
16    int beta = 50;
17
18    for (int y = 0; y < image.rows; y++){
19        for (int x = 0; x < image.cols; x++){
20            for (int c = 0; c < 3; c++){
21                modifiedImage.at<Vec3b>(y,x)[c] =
22                    saturate_cast<uchar>(
23                        alpha*( image.at<Vec3b>(y,x)[c] ) + beta );
24            }
25        }
26    }
27
28    imshow("Image", image);
29    imshow("Modified Image", modifiedImage);
30
31    waitKey();
32    return 0;
33 }
34
```

And when you run, it should look like this





Part 3: Adding slider bars  
TO BE COMPLETED

Part 4: Using the webcam  
TO BE COMPLETED