MP1: Image Transform

Object-Oriented Data Structures in C++

Introduction

By now you've had a chance to learn about basic program structure, classes, and functions in C++. You've also had a chance to try compiling a few programs and experimenting on the command line. Now it's time to try out a more free-form coding project in your IDE. At the University of Illinois, programming assignments are fondly referred to as *MPs*, although there's some debate about whether that stands for "machine-processed" or "machine problem." Perhaps a reminder of the days when students had to work out programs on paper and submit them on punch cards for batch processing by a mainframe in a separate building. You could say this is your MP1. Coincidentally, you might also prefer to use the Cloud9 IDE to work on your assignment.

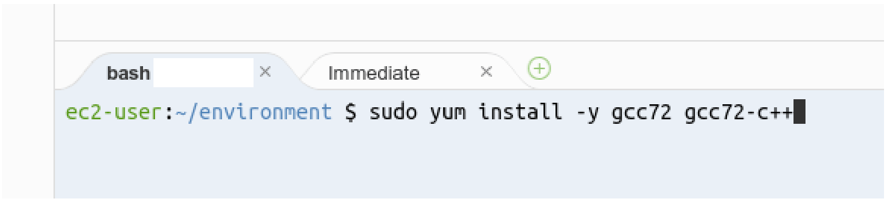
In this assignment, you will begin with foundational work that will be used for several future assignments in this course. Specifically, in Part 1 you will create a simple C++ class that represents a colored pixel; you are also provided with a PNG image class that encapsulates the loading and saving of image files. (You do not need to implement the PNG class yourself, but you'll find it enlightening to study the code provided.) In Part 2, you will use the pixel and PNG classes from Part 1 to transform an image in several different ways.

C++ Environment Setup

Provided with these instructions is a zip file, image-transform-given-files.zip. This file contains everything you need to get started. However, you need to have your IDE or compiler and terminal pre-configured to begin work. If you've been following the previous reading lessons, you should already have access to a Linux-compatible C++ work environment regardless of what your operating system is. If not, please go back and check out the readings!

Supposing that you are trying out the Cloud9 IDE, we'll show you how to load the files into the workspace. In the Cloud9 environment you were able to create following the Week 1 readings, make sure you've updated the compiler first, by typing this command in the terminal:

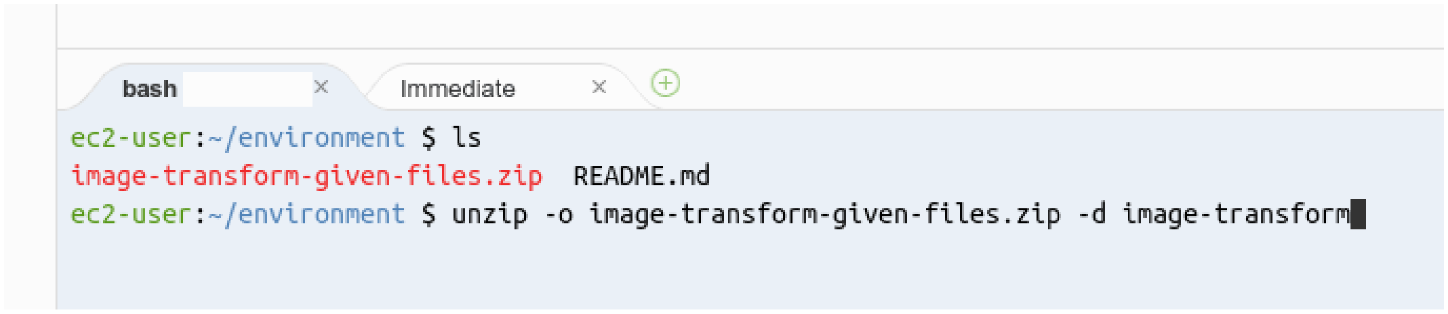
```
sudo yum install -y gcc72 gcc72-c++
```



```
ec2-user:~/environment $ sudo yum install -y gcc72 gcc72-c++
```

With that finished, you'll be ready to load the provided files for this project. In the Cloud9 interface, if you click **File > Upload Local Files**, you will see a popup appear that accepts uploads from your personal computer. You can use this to upload the image-transform-given-files.zip file to your Cloud9 workspace. After you do so, you'll see that the zip file appears in the environment file listing on the left of the screen.

Next, you'll want to extract the contents of the zip file to a new directory. Your terminal on Cloud 9 already has some commands for this, **zip** and **unzip**. First, type **ls** in the terminal and make sure you see the zip file in the same directory. If not, then as described in the earlier readings, you should use the **cd** command to change directory until you see the file in the same directory. (The file listing tree on the left side of the screen should make it easier to find.)

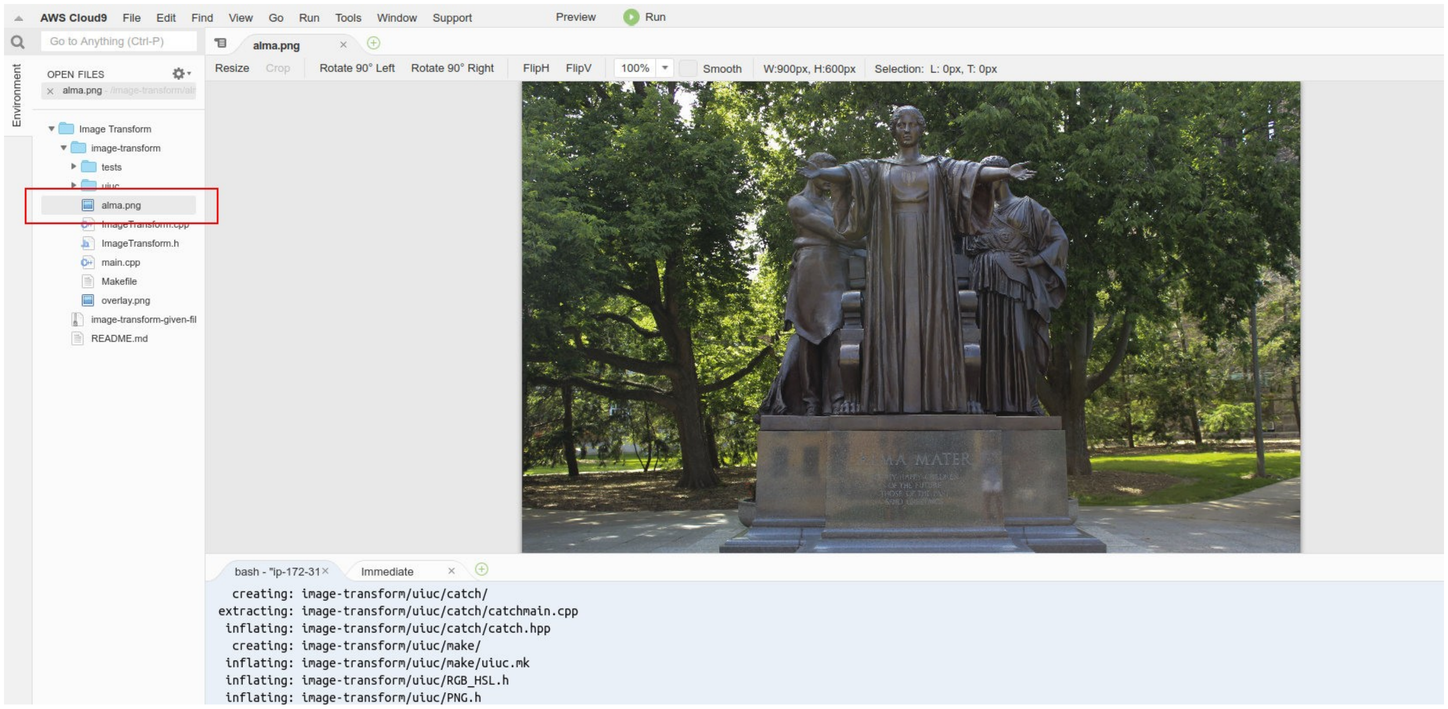


```
ec2-user:~/environment $ ls
image-transform-given-files.zip  README.md
ec2-user:~/environment $ unzip -o image-transform-given-files.zip -d image-transform
```

Once in the same directory as the zip file, you can enter this command to extract the contents of the file:

```
unzip -o image-transform-given-files.zip -d image-transform
```

This will unzip the files into an "image-transform" subdirectory, but be careful, because this will overwrite files in that subdirectory if it already exists! After you extract the code and image files, you'll see that you can double-click to view them directly in the Cloud9 workspace.



```
   creating: image-transform/uiuc/catch/
 extracting: image-transform/uiuc/catch/catchmain.cpp
  inflating: image-transform/uiuc/catch/catch.hpp
   creating: image-transform/uiuc/make/
  inflating: image-transform/uiuc/make/uiuc.mk
  inflating: image-transform/uiuc/RGB_HSL.h
  inflating: image-transform/uiuc/PNG.h
```

Now as you explore the files, there are several you should take note of. We are inviting you to edit these files in particular for the sake of the assignment; they are the only ones we will collect: **`ImageTransform.h`**, **`ImageTransform.cpp`**, **`uiuc/HSLAPixel.h`**, and **`uiuc/HSLAPixel.cpp`**. However, you don't necessarily need to make drastic edits to all of these files! We'll talk more about them below. You can also preview some unit tests used for grading, and take a look at the PNG library code.

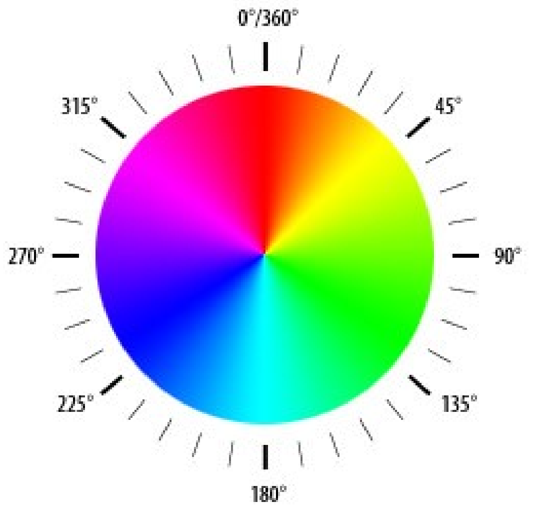
Part 1: HSLAPixel

Understanding Color Spaces

In this course, we will not be working with the physical properties of color that you may be familiar with from other sources (the “RGB color space” for red-green-blue channels.) Instead, we will be using an alternative color space that represents colors by human perception of color. The HSL color space uses the Hue, Saturation, and Luminance of the color. From the *Adobe Technical Guides* page on “The HSB/HLS Color Model”, Adobe explores these terms:

Hue

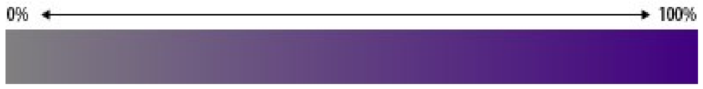
Hue (denoted as *h*) defines the color itself, for example, red in distinction to blue or yellow. The values for the hue axis run from 0–360° beginning and ending with red and running through green, blue and all intermediary colors like greenish-blue, orange, purple, etc.



There are two main hues that we'll use later in this assignment:

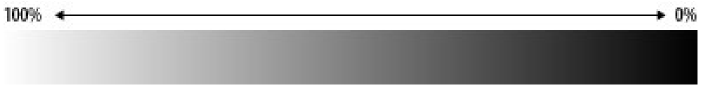
- "Illini Orange" has a hue of 11.
- "Illini Blue" has a hue of 216.

Saturation

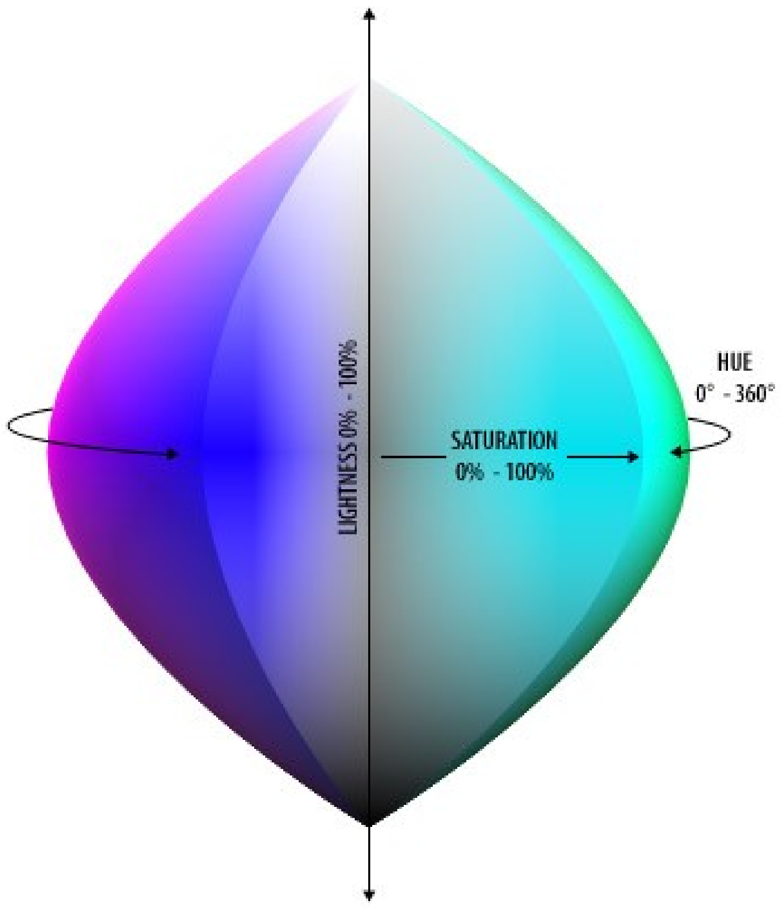


Saturation (denoted as *s*) indicates the degree to which the hue differs from a neutral gray. The values run from 0%, which is no color saturation, to 100%, which is the fullest saturation of a given hue at a given percentage of illumination.

Luminance



Luminance (denoted as *l*) indicates the level of illumination. The values run as percentages; 0% appears black (no light) while 100% is full illumination, which washes out the color (it appears white).



HSL Color Space

The **full HSL color space** is a three-dimensional space, but it is not a cube (nor exactly cylindrical). The area truncates towards the two ends of the luminance axis and is widest in the middle range. The ellipsoid reveals several properties of the HSL color space:

- At **l=0** or **l=1** (the top and bottom points of the ellipsoid), the 3D space is a single point (the color black and the color white). Hue and saturation values don't change the color.
- At **s=0** (the vertical core of the ellipsoid), the 3D space is a line (the grayscale colors, defined only by the luminance). The values of the hue do not change the color.
- At **s=1** (the outer shell of the ellipsoid), colors are vivid and dramatic!

Part 1 Programming Objectives

You need to create a class called **`HSLAPixel`** within the **`uiuc`** namespace in the file **`uiuc/HSLAPixel.h`**. Each pixel should contain four public member variables:

- **`h`**, storing the hue of the pixel in degrees between 0 and 360 using a **`double`**
- **`s`**, storing the saturation of the pixel as a decimal value between 0.0 and 1.0 using a **`double`**
- **`l`**, storing the luminance of the pixel as a decimal value between 0.0 and 1.0 using a **`double`**
- **`a`**, storing the alpha channel (blending opacity) as a decimal value between 0.0 and 1.0 using a **`double`**

That's it! Once you have this simple class, you are ready to compile and test Part 1 of this assignment.

Compiling Your Code

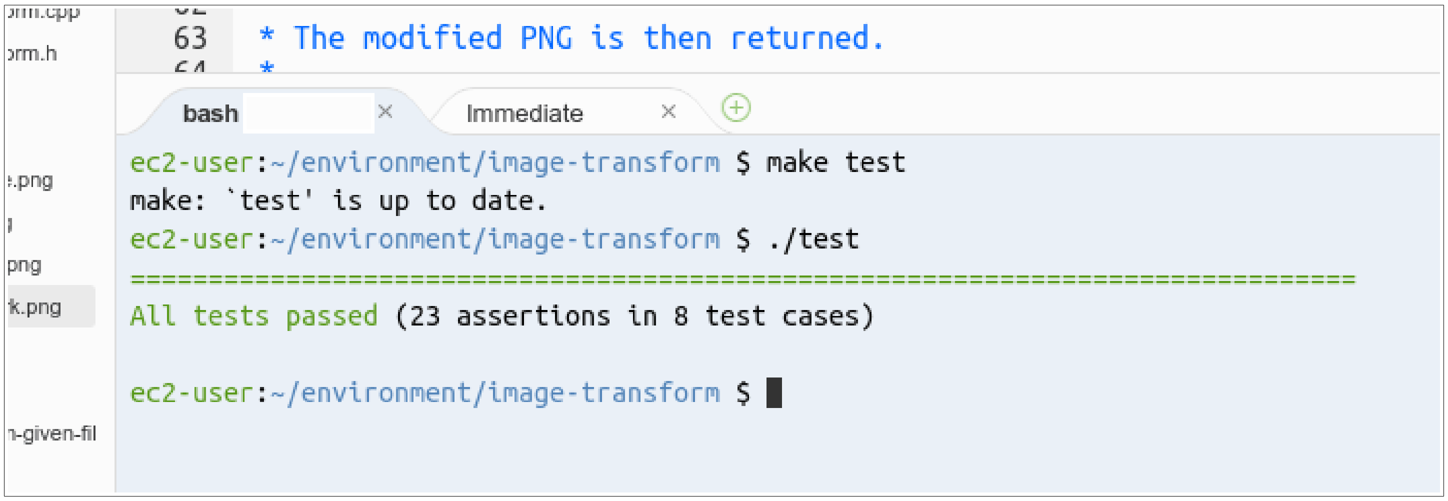
To compile the code, you must use the terminal to enter the same directory where the **`Makefile`** is stored; it's in the top directory next to **`main.cpp`**. As explained in the readings, use **`cd`** to change to the appropriate directory, use **`ls`** to view the file list in that directory, and then type **`make`** to automatically start the compilation. If you need to clear out the files you've previously built, type **`make clean`**. If you encounter any warnings or errors during compilation, study the messages in the terminal carefully to figure out what might be wrong with your code.

The primary compilation sequence, once successful for all parts of this MP, will build an executable file simply called **`ImageTransform`** in the top directory. You'll be able to run it by typing **`./ImageTransform`** in that directory to launch a sequence of image processing operations. However, first you will also want to compile the test suite and make sure you are doing Parts 1 and 2 correctly with unit tests. To compile the tests, type **`make test`** in the top directory. If successful, this will generate an executable called **`test`**, which you can run by typing **`./test`** in the same directory. We use the widely-used C++ testing framework Catch. You are always encouraged to write additional test cases; the executable Catch generates will run all of your tests and show you a report.

Working on your code is an iterative process. After doing Part 1, you will only be passing 4 of the tests. You should continue on to Part 2 to pass the rests of the tests. However, if any of the tests fail, you can always find out more by reading the message and peeking at the test source code files in the **`tests`** directory. The unit tests will usually point out a *contradiction* they have found by running your code: the test asserted that a certain result *should have* taken place, but the program did something else instead. Here is what you'll see when you've only finished Part 1:

```
tests/part2.cpp:109: FAILED:
  REQUIRE( png.getPixel(100, 15).l + 0.2 == result.getPixel(100, 15).l )
with expansion:
  0.7 == 0.5


===============================================================================
test cases:   8 | 4 passed | 4 failed
assertions: 15 | 7 passed | 8 failed
```

For example, this part of the unit test report is saying that the value it had expected (0.7, on the left of the equality) is not the same as the value it found with your program (0.5, on the right of the equality). The values *should* be equal (**==**), but they're not yet. But, there's no need to get too worried about unit tests for features you haven't gotten around to implementing yet. That's why the ideal unit test ensures that the smallest possible component is working correctly, in isolation; then, your large-scale engineering problem breaks down into a series of many tiny steps.



```
ec2-user:~/environment/image-transform $ make test
make: `test' is up to date.
ec2-user:~/environment/image-transform $ ./test
===============================================================================
All tests passed (23 assertions in 8 test cases)

ec2-user:~/environment/image-transform $ 
```

By time you've finished the next part of the assignment, you should be able to pass all of the tests, resulting in a clean report like this one.

Part 2: Using uiuc::PNG

Now that we have an **`HSLAPixel`**, it's time to manipulate some images! First, let's understand the PNG class that is provided for us! Note that PNG stands for "portable network graphics." It's a useful image file format for storing images without degradation (called losssless compression). A photograph saved as PNG will have a much larger filesize than a JPEG, but it will also be sharper and the colors won't bleed together as happens with JPEG compression. Apart from photographs, though, PNG offers relatively small filesizes for simple graphics with few colors, making it a popular choice for small details in a website user interface, for example.

Inside of your **`uiuc`** directory, you may have noticed **`PNG.h`** and **`PNG.cpp`**. We have provided an already complete PNG class that saves and loads PNG files and exposes a simple API for you to modify PNG files. (An API, or application programming interface, just means the documented, surface-level part of the code that you work with as someone who is making use of the library. You do *not* need to fully understand how the PNG class works underneath in order to use it. But, it's good for you to practice surveying library code, seeing how it is organized, and taking note of its intended usage.)

In C++, the scope resolution operator, **`::`**, denotes the fully-qualified name of a function or variable that is a member of a class (or namespace). So, you will see things like *class name*::*member name* in writing and in code, in some contexts. Note how this is different from the member selection operator, **`.`**, which appears in contexts like *class instance*.*member name*. This is because **::** shows a relationship to an entire *class* or to a namespace, whereas the **.** operator shows a relationship to a single instance of an object.

Note that some of the class members utilize pass-by-reference as shown by the **&** operator; this is fully explained in Week 3. It means that a direct *reference* to the memory is being passed, which uses the same concept of indirection that pointers offer, but with a simple syntax—you just use the variable as you would normally, not by dereferencing a memory address explicitly with the * operator as pointers do, and yet you are still able to *implicitly* manipulate the data located at the original memory location. This convenience feature was one of the benefits of C++ over the older C language.

For loading and saving PNG files:

- **`bool PNG::readFromFile(const std::string & fileName)`** loads an image based on the provided file name, a text string. The return value shows success or failure. The meaning **`&`** is discussed in Week 3 in the lecture about variable storage; it means a direct *reference* to the memory is being passed, similar to a pointer.
- **`bool PNG::writeToFile(const std::string & fileName)`** writes the current image to the provided file name (overwriting existing files).

For retrieving pixel and image information:

- **`unsigned int PNG::width() const`** returns the width of the image.
- **`unsigned int PNG::height() const`** returns the height of the image.
- **`HSLAPixel & getPixel(unsigned int x, unsigned int y)`** returns a direct reference to a pixel at the specified location.

Other methods:

- Methods for creating empty PNGs, resizing an image, etc. You can view **`uiuc/PNG.h`** and **`uiuc/PNG.cpp`** for complete details.

Sample Usage of PNG

Suppose we want to transform an image to grayscale. Earlier you learned that a pixel with a saturation set to 0% will be a gray pixel. For example, here's this transformation applied to the Alma Mater. The image on the left is the original, and the one on the right has been desaturated:

The source code that used the **`PNG`** class to create this grayscale transformation above is as follows (which is also provided to you in **`ImageTransform.cpp`**):

```
/**
 * Returns an image that has been transformed to grayscale.
 * The saturation of every pixel is set to 0, removing any color.
 * @return The grayscale image.
 */
PNG grayscale(PNG image) {
  /// This function is already written for you so you can see how to
  /// interact with our PNG class.
  for (unsigned x = 0; x < image.width(); x++) {
    for (unsigned y = 0; y < image.height(); y++) {

      HSLAPixel & pixel = image.getPixel(x, y);

      // `pixel` is a reference to the memory stored inside of the PNG `image`,
      // which means you're changing the image directly. No need to `set`
      // the pixel since you're directly changing the memory of the image.

      pixel.s = 0;
    }
  }
  return image;
}
```

Here the pixel itself is passed by reference, although for simplicity, the image manipulation functions themselves in **`ImageTransform.cpp`** simply pass the images by value, which means they pass an entire new copy of the image each time. In the future, in practice, you'll find that it's more efficient to *indirectly* refer to large memory objects using pointers and references instead of making extra copies just to pass between functions. Many scripting languages today use reference semantics for almost everything by default, but C++ gives you more control to choose when memory is copied or when it is altered in-place.

Part 2 Programming Objectives

For the rest of the assignment, you should finish implementing the functions in **`ImageTransform.cpp`**: **`illinify`**, **`spotlight`**, and **`watermark`**. You'll find that the grayscale function discussed above has already been completed for you. There are some comments on the function bodies with notes about their implementation. You may want to make use of some basic mathematical functions that are available because of the **`#include <cmath>`** in **`ImageTransform.h`**, but that's not required.

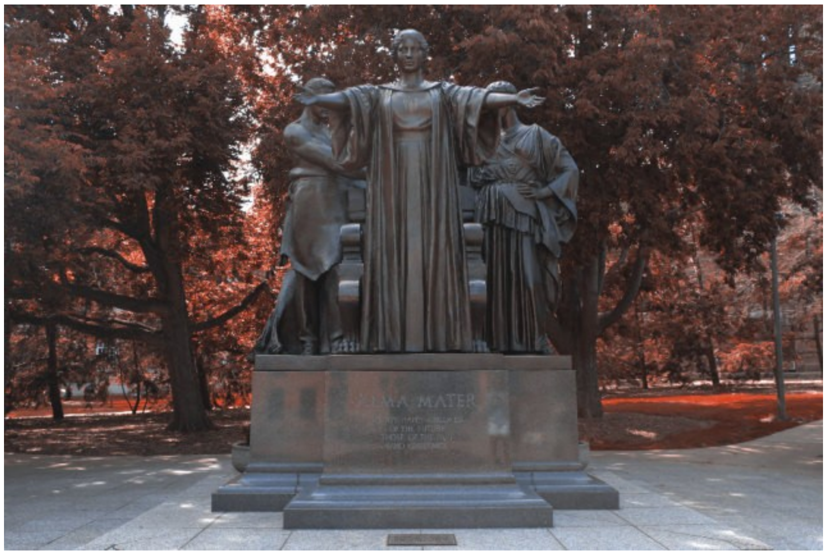
As you know, all C++ programs begin their execution with the main function, which is usually defined in main.cpp. You can find that a main function has been provided for you that:

- Loads in the image `alma.png`
- Calls each image modification function
- Saves the modified images named as `out-`*`modification`*`.png`, where *modification* shows the function being tested (e.g. `out-grayscale.png`)

Descriptions and examples of the three remaining functions are given below:

Function #1: illinify

To illinify an image is to transform the hue of every pixel to Illini Orange (11) or Illini Blue (216). The hue of every pixel should be set to one or the other of these two hue values, based on whether the pixel's original hue value is closer to Illini Orange or Illini Blue. Remember, hue values are arranged in a logical circle! If you keep increasing the hue value, for example, what should eventually happen?

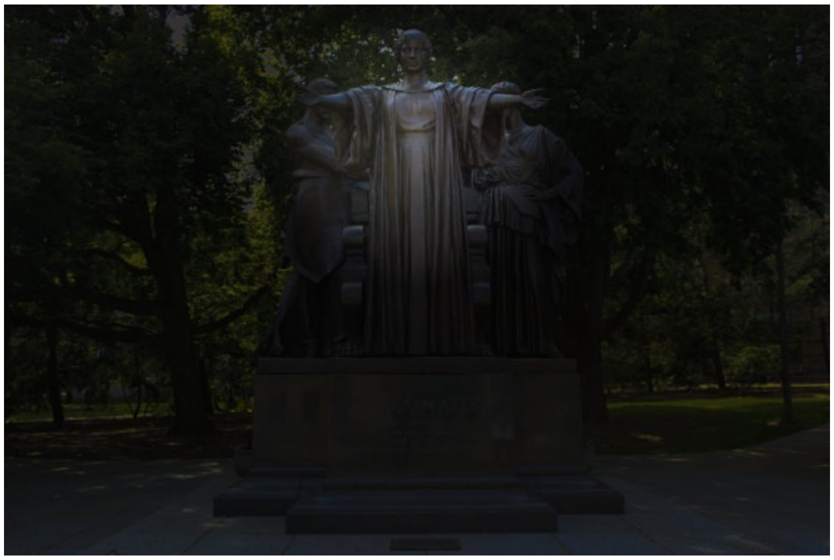


Function #2: spotlight

To spotlight an image is to create a spotlight pattern centered at a given point (**`centerX`**, **`centerY`**).

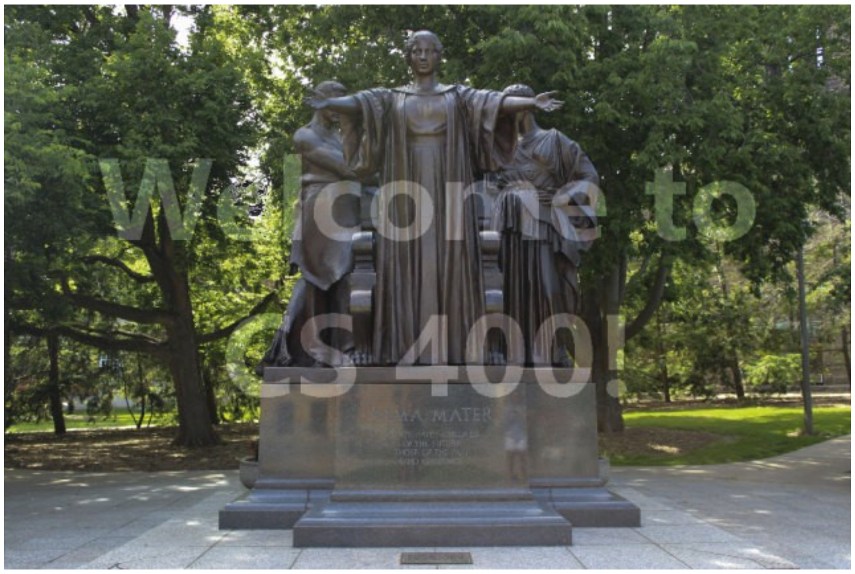
A spotlight adjusts the luminance of a pixel based on the distance between the the pixel and the designated center by decreasing the luminance by 0.5% per 1 pixel unit of Euclidean distance, up to an 80% decrease in luminance at most.

For example, a pixel 3 pixels above and 4 pixels to the right of the center is a total of **`sqrt(3*3 + 4*4) = sqrt(25) = 5`** pixels away and its luminance is decreased by 2.5% (0.975 its original value). At a distance over 160 pixels away, the luminance will always be decreased by 80% (0.2x its original value).



Function #3: watermark

To watermark an image is to lighten a region of it based on the contents of another image that acts as a stencil.



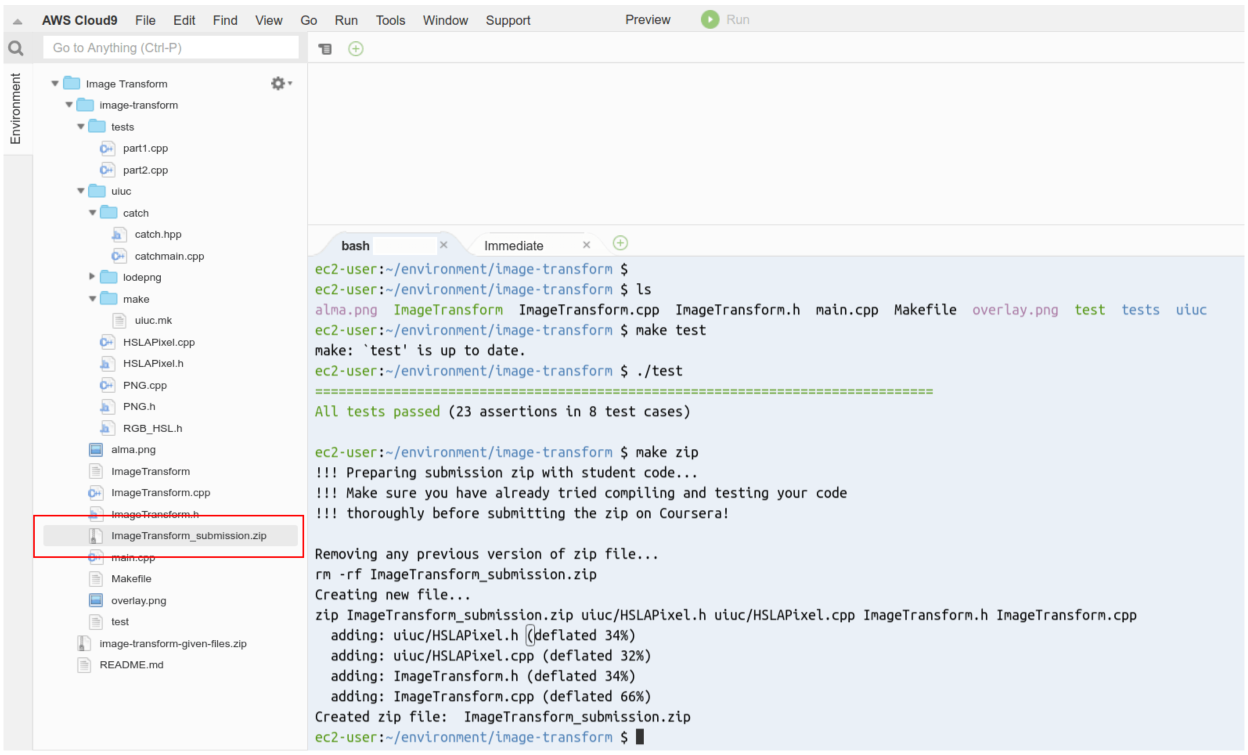
You should not assume anything about the size of the images. However, you need only consider the range of pixel coordinates that exist in both images; for simplicity, assume that the images are positioned with their upper-left corners overlapping at the same coordinates.

For every pixel that exists within the bounds of both base image and stencil, the luminance of the base image should be increased by +0.2 (absolute, but not to exceed 1.0) *if and only if* the luminance of the stencil at the same pixel position is at maximum (1.0).

Submitting Your Work

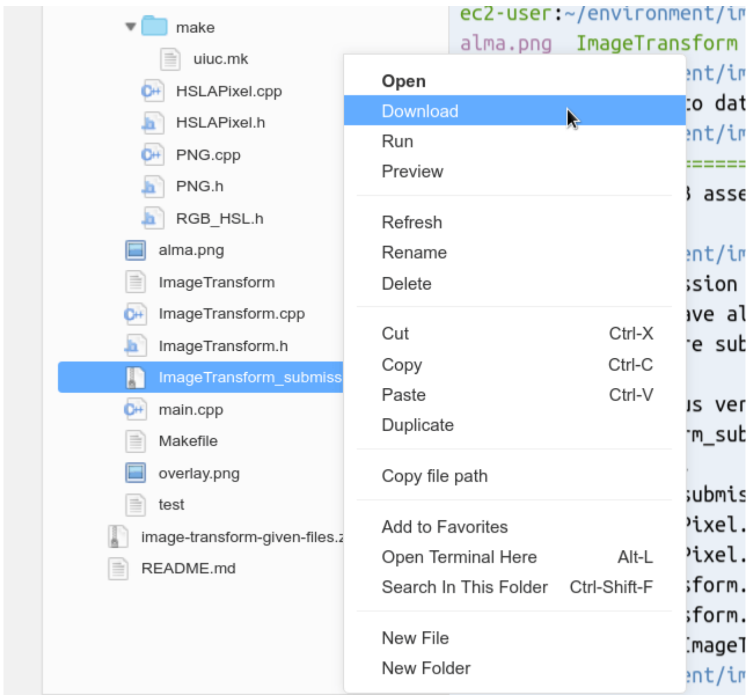
As described previously, you should extensively check that your code compiles in your own IDE before you prepare to submit it. Be sure to do **`make`** and run **`./ImageTransform`** to see that your images turn out as expected, and also do **`make test`** followed by **`./test`** to check that you aren't forgetting anything.

When you're ready to hand in your work, you should use **`cd`** to navigate to the directory where your **`main.cpp`** and **`Makefile`** are, and then run **`make zip`** to generate a submission file automatically, called



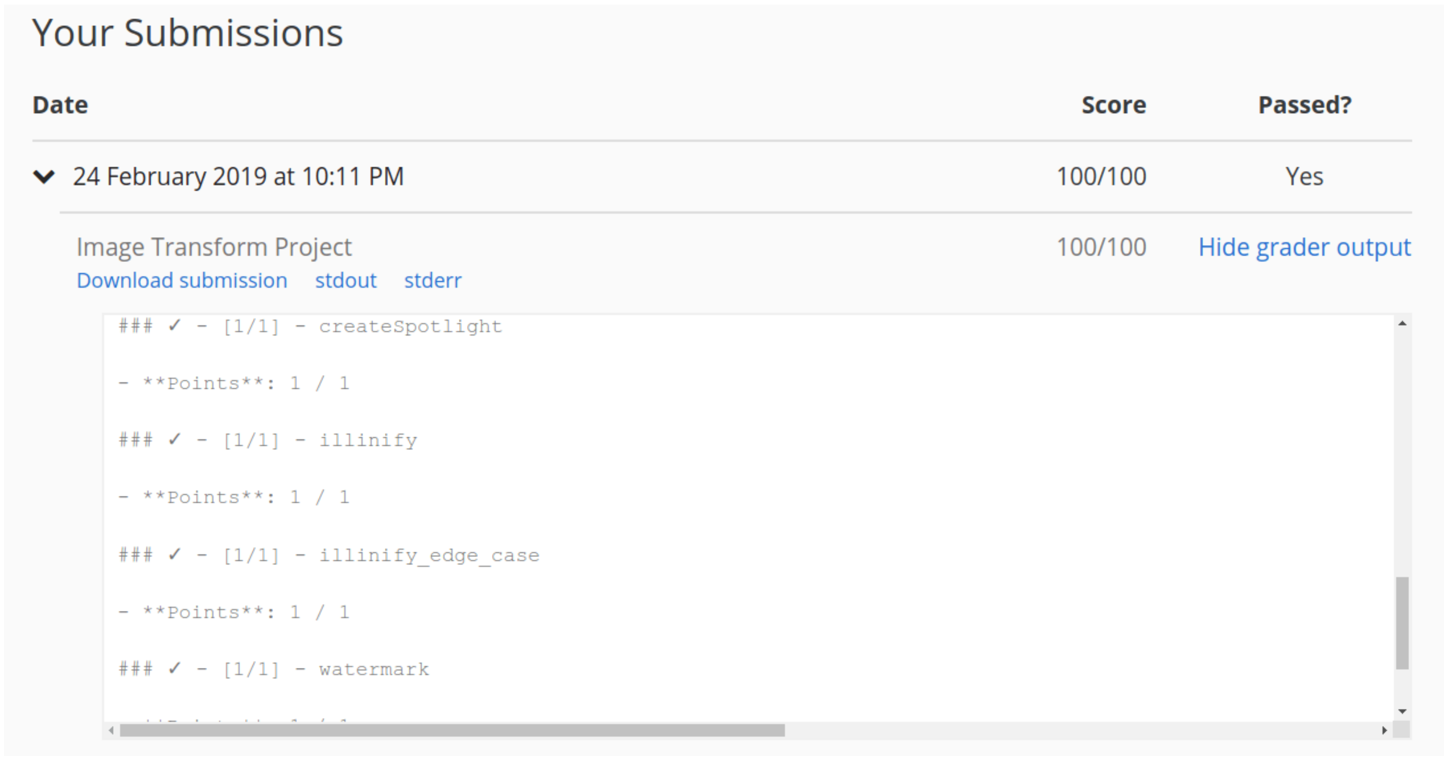
```
ec2-user:~/environment/image-transform $
ec2-user:~/environment/image-transform $ ls
alma.png  ImageTransform  ImageTransform.cpp  ImageTransform.h  main.cpp  Makefile  overlay.png  test  tests  uiuc
ec2-user:~/environment/image-transform $ make test
make: `test' is up to date.
ec2-user:~/environment/image-transform $ ./test
===============================================================================
All tests passed (23 assertions in 8 test cases)

ec2-user:~/environment/image-transform $ make zip
!!! Preparing submission zip with student code...
!!! Make sure you have already tried compiling and testing your code
!!! thoroughly before submitting the zip on Coursera!

Removing any previous version of zip file...
rm -rf ImageTransform_submission.zip
Creating new file...
zip ImageTransform_submission.zip uiuc/HSLAPixel.h uiuc/HSLAPixel.cpp ImageTransform.h ImageTransform.cpp
  adding: uiuc/HSLAPixel.h (deflated 34%)
  adding: uiuc/HSLAPixel.cpp (deflated 32%)
  adding: ImageTransform.h (deflated 34%)
  adding: ImageTransform.cpp (deflated 66%)
Created zip file:  ImageTransform_submission.zip
ec2-user:~/environment/image-transform $
```

`ImageTransform_submission.zip`. It will contain only the four code files being collected for grading. You can find the generated file in the file browser in the left pane. If you right-click on it, you'll see a Download option; save the zip file to your own computer. Now, if you go to the Week 4 programming assignment item on Coursera, you will be able to submit this zip file for autograding. The autograding procedure may take some minutes to show a result, but this time it should reflect what you expected based on your unit testing. In some projects there may be additional hidden edge cases tested in the final grading.



Please also note: If you see a link on the Week 4 assignment submission page mentioning "upgrading" your assignment to a new version, be sure to click it, so that Coursera will update the assignment page you've already signed into to connect to the newest version of the autograder.

If you have any questions, be sure to ask on the discussion forums!



Your Submissions

Date	Score	Passed?
24 February 2019 at 10:11 PM	100/100	Yes
Image Transform Project	100/100	Hide grader output

Download submission stdout stderr

```
### ✓ - [1/1] - createSpotlight

- **Points**: 1 / 1

### ✓ - [1/1] - illinify

- **Points**: 1 / 1

### ✓ - [1/1] - illinify_edge_case

- **Points**: 1 / 1

### ✓ - [1/1] - watermark
```