ECE 252: Programming Methodology I
Department of Electrical and Computer Engineering
Prof. Jorge Ortiz
Rutgers University
RUTGERS
School of Engineering
ELECTRICAL AND COMPUTER ENGINEERING

ECE 252: Programming Methodology (Spring 2024) – Project 2

Due: May 1, 2024 (11:59:59 PM)

Rationale and learning expectations: This project reinforces all the constructs, data types, and operations supported in C++. This includes data types, pointers, loops, operators, classes, inheritance, overloading, and compilation. The goal is to combine these into a comprehensive program that transforms images passed as input to your program using object-oriented programming.

For this project, your task is to refactor the code from Project 1 into an object-oriented program implemented in C++. In Project 1, you developed an image processing library capable of processing images using matrix operations. It's important to understand that all images can be represented as a matrix of single-byte values. Each pixel at matrix position **`(i, j)`** is denoted by a numerical value ranging from 0 to 255. Additionally, a color image consists of three separate channels: a **`red (R) channel, a green (G) channel, and a blue (B) channel`**. The image data includes three corresponding matrices, each sharing the same dimensions regarding the number of rows (height) and columns (width). Each of these matrices represents one of the three color channels. These matrices are then collectively passed to an image-writing library to generate an output file and facilitate visualization.

In this assignment, you will construct several classes to build the library. The process begins with the creation of a vector class, followed by the development of a matrix class that is composed of vectors. Lastly, you will extend the matrix class to create an image class.

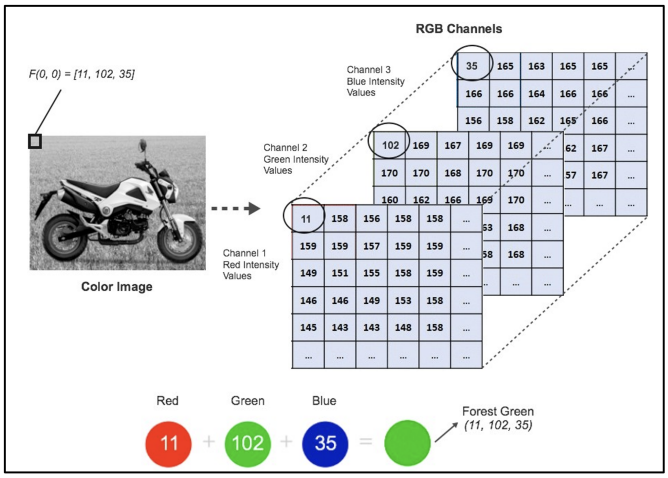

Figure 1: This illustrates how a computer 'sees' an image. A color image is a set of three matrices that represent the pixel intensity in (R)ed, (G)reen, (B)lue. In this project, we will manipulate the image by performing arithmetic operations on individual pixel values and observing the outs.

To complete the project, you must write various functions that bridge the general image library and the RGB matrix view. These functions will use the matrix view to create new images, including image addition, subtraction, scalar

and multiplicative scaling. By updating the individual pixel values in the image's matrix, you can manipulate the image in various ways to achieve the desired result. We will provide you with the starting code and required files for the project.

Environment Setup

Code link: https://rutgers.box.com/s/mox3yuqqe96q1lpv14naum9ihn7ixtmz

1. Set up visual studio code for C/C++: Visual Studio Code C/C++ setup.
2. Download the code and set up your environment.
3. Rename the file **`FIRSTNAME_LASTNAME_NETID`** with your first name, last name, and NETID.
4. Test by running make. To learn more about **`make`**, read this Makefile tutorial.

Part I (35 points)

Create a templated class named Vector that serves as a fundamental component for implementing matrices in the image-processing library (from Project 1) that performs matrix operations on images. The Vector class should be capable of storing a collection of homogeneous data of any data type. Implement the class using either an array or a linked list, based on your preference. The class should provide the following functionalities:

1. **`int getsize()`**: A member function that returns the size of the vector, i.e., the number of elements it contains.
2. Copy constructor: A constructor that creates a new vector as a copy of an existing one.
3. Assignment operator (**`operator=`**): An operator that allows the assignment of one vector to another.
4. Destructor: A destructor that cleans up any resources the vector uses.
5. Input and output stream operators (**`operator>>`** and **`operator<<`**): Operators that allow reading from and writing to input and output streams, respectively.
6. Arithmetic operators (**`operator+, operator-, and operator*`**): Operators that allow addition, subtraction, and multiplication of vectors.
7. Subscript operator (**`operator[]`**): An operator that provides access to individual elements of the vector based on their index.
8. Additionally, implement any member functions as virtual if you believe they should be overrideable in derived classes. This class will later be used to create the **`Matrix`** class, which will be an essential part of the image-processing library.
9. Constructors also need to be defined for the Vector object to be usable.

Part II (35 points)

Create a new class named **Matrix** that represents a matrix of elements of type **uint8_t**. The **Matrix** class should utilize the **Vector** class through composition to implement the matrix structure, where each row or column of the matrix is represented as an instance of the **Vector** class. The class should provide implementations for the following operators and member functions:

1. Copy constructor
2. Assignment operator (**operator=**)
3. Destructor
4. Input and output stream operators (**operator>>** and **operator<<**)
5. Arithmetic operators (**operator+**, **operator-**, and **operator***)
6. Subscript operator (**operator[]**)

Additionally, implement the following new member functions in the **Matrix** class:

1. **int getrows()**: A member function that returns the number of rows in the matrix.
2. **int getcols()**: A member function that returns the number of columns in the matrix.

The **Matrix** class will serve as an essential component in the image-processing library (from Project 1) that performs matrix operations on images. The composition of **Vector** instances allows the **Matrix** class to leverage the functionalities of the **Vector** class while maintaining the semantic distinction between vectors and matrices.

Part III (30 points)

Create a new **Image** class that inherits from the **Matrix** class and implements all the functions described in Project 1. The **Image** class should have the following properties:

- `filePath`: The file path for the image.
- `numChannels`: The number of color channels (e.g., 3 for RGB images).
- `width`: The width of the image in pixels.
- `height`: The height of the image in pixels.

The **Image** class should be capable of performing the following operations on images:

1. Scaling an image
2. Adding two images
3. Subtracting two images
4. Multiplying two images

The operations for scaling, adding, subtracting, and multiplying images should be implemented as overloaded operators (**+**, **-**, and *****).

Part IV (30 points) EXTRA CREDIT

- Add a function called **`transpose()`** to the **`Matrix`** class. This function should generate the transpose of the original matrix, such that each element in the transposed matrix M^T is given by the formula $M^T_{ij} = M_{ji}$, where M_{ji} is the element in the j-th row and i-th column of the original matrix M. The resulting matrix M^T should have dimensions that are the transpose of the original matrix's dimensions, i.e., if the original matrix M has dimensions $m \times n$, the transposed matrix M^T should have dimensions $n \times m$.
- Additionally, create a function called **`resize()`** in the **`Image`** class that uses the `stb_image_resize` library to resize an image and populates the **`Image`** object accordingly. The **`resize()`** function should take the original image of dimensions $m \times n$ (height m and width n) and produce a resized image of dimensions $p \times q$ (new height p and new width q), where p and q are the target dimensions for resizing. Use the functions provided by the `stb_image_resize` library to perform the resizing operation, preserving the aspect ratio and visual content of the image while adjusting its size. The resized image should be stored in the **`Image`** object.

You may want to write a **`main.cpp`** file that tests the **`Image`** class with real images, executing all the operations implemented in the previous parts of this project.

Submission

Please zip your code and submit it via Canvas. You must include the file with your name and NETID (as specified in Section and your modified headers.

An illustration of the file structure is shown below:

The file organizational structure for this project is as follows:

- FIRSTNAME_LASTNAME_NETID
- Makefile
- src
  - Image.cpp

  - Image.h
  - Matrix.cpp
  - Matrix.h
  - Vector.h
  - main.cpp
- stb_image
  - stb_image.h
  - stb_image_resize.h
  - stb_image_write.h

Assignment Rubric

1. **Compilation and Execution:** (30 points)
   This criterion evaluates the code's ability to compile and execute correctly. Points are assigned based on the code's compilation success and its ability to run without any runtime errors or crashes.
   Point Distribution:
   - Code compiles successfully: 10 points
   - Code executes without runtime errors or crashes: 20 points

2. **Unit Test Results:** (40 points)
   This element assesses the results of the unit tests for each function. Points are assigned based on the number of tests passed and the accuracy of the output.
   Point Distribution:
   - All unit tests pass: 40 points
   - Some unit tests pass: 20-30 points
   - A mix of passing and failing unit tests: 10-20 points
   - Most unit tests pass: 30-35 points

3. **Partial Credit for Correctness:** (20 points)
   This criterion awards points for achieving partial correctness in the functions. Points are assigned based on the level of correctness and the number of cases in which the function produces partial correct output.
   Point Distribution:
   - No partial credit awarded: 0 points
   - Minimal partial credit for partially correct output in some cases: 5-10 points
   - Moderate partial credit for partial correctness in multiple cases: 10-15 points
   - Significant partial credit for substantial partial correctness in most cases: 15-18 points
   - Full credit for achieving partial correctness in all relevant cases: 20 points

4. **Handling of Edge Cases and Corner Cases:** (10 points)
   This element evaluates the code's ability to handle edge cases and corner cases effectively. Points are assigned based on the code's accuracy in handling these special scenarios.
   Point Distribution:
   - Does not handle any edge or corner cases correctly: 0 points
   - Handles a few basic edge cases but fails to handle most corner cases: 2-4 points
   - Handles some common edge and corner cases, but may miss a few: 4-7 points
   - Handles most edge and corner cases effectively but may have minor issues: 7-9 points
   - Handles a wide range of edge and corner cases accurately and comprehensively: 10 points