CSCI160 Assignment 6 Two Dimensional Arrays, File I/O, Pointers, and References

Objectives

Upon completion of this assignment, you need to be able to:

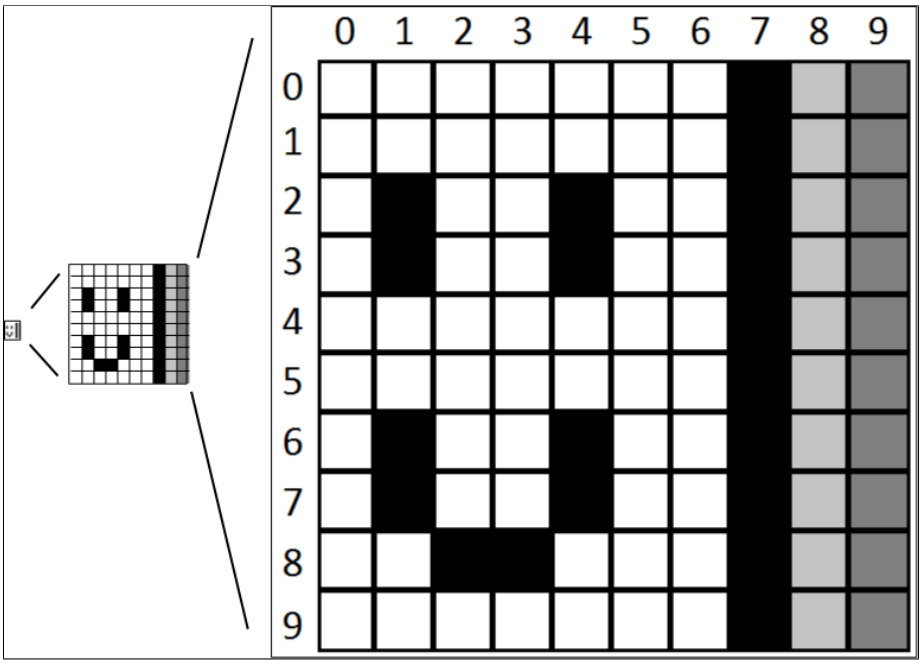
- Pass `parameters` and `return` values using *references*.
- Pass arguments from the command-line to the `String[] args` array.
- Handle input and output text files.
- Implement C++ code that handles *two-dimensional* (`2D`) arrays.
- Have some practice dealing with objects from a class description.
- Extend previously written code.
- Continue building good programming skills.

Introduction

One well-known set of tasks for computers is the manipulation of images. Your work for this assignment is to add additional functionality to the C++ source code called `imageManipulations.cpp`. Each execution of the program will perform one of the following conversions on an image:

- An `ASCII`-art transformation,
- The *scaling* of an image either enlarging or reducing its size,
- Two *reflections*, one about the `x`-axis, the other about the `y`-axis,
- An *inversion*, which creates the negative of an image, and
- A *rotation*, which rotates an image by $90°$, in a clockwise direction.

To illustrate how the data in a bitmap image file is represented in a 2D array of integers, consider the very small image below that is 10 pixels wide by 10 pixels high to the far left of the following image:



If we really zoom in on this image, we can see that the image is a matrix of pixels, each representing some shade of gray.

A 2D array of integers with values ranging from 0 to 255 to represent the above image is used. Suppose this image was in a file called `happyFace.pgm`*. In a text editor, we would see the *magic number* which is **P2**, indicating that the pixels are represented by integers separated by white space. In the second line, we have numbers for the width and height of the image (in pixels) as well as the grayscale range, which is generally $2^8 = 256$ numbers between 0 and 255. This is the range of values that are given each pixel. Any pixel that has the number 255 will indicate a white spot (pixel) and any pixel with 0 with indicate a black spot.

Quick Start

(1) Download this pdf file and store it on your computer.
(2) Create a directory / folder on your computer specific to this assignment. for example, `CSCI160assn6` is a good name.

*A Portable Gray Map (pgm) file's content can be easily viewable when opened with a text editor, or when opened with image viewing software.

(3) Save the file called `source.zip` in your directory. Unzip the file, by double clicking on it on a Windows machine or typing `unzip source.zip` on the command line. Inside the assignment directory you created, you will find the following files:

`imageConversions.cpp`: This is the file that you will use to complete the source code for this assignment. The shell has been provided for you.

`ImageType.hpp`: This is a declaration of a `class` in C++. It lists the private member variables and one private member function. The public member functions are accessible by any initialized object of this class. Note that not ALL the functions are declarations: a couple of them are definitions. This is acceptable for functions that have a very small set of simple statements within the body. YOU MUST NOT ALTER THIS FILE!

`ImageType.cpp`: This finishes the definitions of the functions declared in the `ImageType.hpp` file. It defines the member functions that you can use for any ImageType object. We will be discussing this example in lectures when we talk about *Object-oriented* programming. YOU MUST NOT ALTER THIS FILE!

`makefile`: All of the above mentioned files need to be compiled and linked in the same directory for you to create, run and test the `ImageConversions.cpp` source code. The `makefile` is a handy tool to do the compilation and linking. You do not need to know *how* it works, nor are you required to use it. It is a unix/linux file that tells the system the ordering and compilation of the files. For some helpful information on separate compilation and the `makefile`, see this information document.

(4) Save the file called `images.zip` in your directory and unzip it as well. You will find three image files (`*.pgm` files), that you can use as input to test your code as you write it. They are each viewable within a text editor, although that will only show you the pixel values of the image. If you double-click on the file, and you do not see an actual image come up, then check out the information here that provides a list of applications that will open a pgm file (as an image).

(5) Have a look at the ImageType specification document, to get a sense of how to use the created `ImageType` class. You do not need to understand *how* it works, just *what* it does. You can read a little more details about this class in the next section.

(6) Look at the `imageConversions.cpp` file. The basic shell has been provided and is compilable but does not do anything yet. Currently, the program compiles (with lots of warnings about unused variables) and runs, but only works with the request to "`invert`", which creates a new image that is exactly the same as the original image. As you progress, you will fill in each function and check the results.

(7) Compile and run the program. If you would like to try the `makefile` (and are using a Mac or Linux machine), you can type:

```
make go
```

or if you are not using this file:

```
g++ -go -Wall ImageType.cpp imageConversions.cpp
```

Note that you get more warnings when using the make file. That's because I added a lot of extra warning flags. If you wish, you can add them in the regular command as well, but they require a bit of typing.
To run the program, copy the following to the console:

```
./go images/mona_lisa.pgm images/mona_inverted.pgm invert
```

When finished, there will be a new file in the images directory, called `mona_invert.pgm`. It will look exactly the same as the original. Once you complete the `invert` function, it should look more like the image you see in this document.

(8) Start with the easiest methods first. Once you get going, you will find that the functions have very similar structure, and you are mostly playing with the values within the 2-D array. See the detailed instructions in the next section for some tips.

Detailed Instructions

The `ImageType` class

This class allows you to create an object that represents a very basic pgm image. The ImageType class is useful because it contains, as one of its member variables, a 2-dimensional array with `height` rows and `width` colums. Each entry in the array is a number between 0 and `grayscale`-1, and corresponds to the balance of black and white levels that create a grayscale colour.

Note that the variables `height`, `width`, `grayscale` and `pixels` are all private, and are therefore not directly accessible. However, there are member functions (often called *methods*) that allow you access to them indirectly (remember the `length()` function for any `string` object?).

To create an object of type `ImageType`, we call a *constructor*, which will initialize an object for us. There are two constructors in the class: One that takes values for the height, the width and the grayscale value and one that takes the name of a file that has the required information to be extracted.

In both cases, a double dimensional array is reserved in memory. Note that this array does not disappear when the function is finished. The memory for this array is dynamically allocated, then attached to the newly created object. When the object is no longer in use by

any part of the program, the system will make a call to the function called the *destructor* to delete any dynamically allocated memory that was created. (The destructor is the method that has the tilde '~' as its first character.)

The single statement in `imageConversions` `main`: `ImageType image(infile);` calls the constructor in `ImageType` that takes a filename as its input parameter, creates an object, initializes the pixel array, and stores the information in the variable `image`.

We also make use of the *copy constructor*, which allows us to make a copy of an `ImageType` object. When no dynamically allocated memory is used to create an object, the system can easily copy the object. However, for the `ImageType` object, the copy will have a pointer to the original 2D array. The side effect is that any changes to one object's array will affect the other's. In other words, the dynamically allocated member variables of the original object are not actually copied. If we want separate copies, then we need to make sure that happens in the copy constructor we write.

We will be discussing how to use a class definition to create an object and access the functions in lectures. This is not completely new, though. We have already been using the `string`, `istream`, and the `ostream` classes when we access a string `length()` or concatenate strings, or whenever we use the variables `cin` and `cout`.

imageConversions.cpp

The imageConversions specifications document provides details on *what each function does*. In each of the following sections, we also provide information on *how it each function works*. Note that `main` has been started for you.

main

Note that we are using the originally designed C `main` function that takes very specific input parameters. All of the information needed by the program is passed in through the `argv` array (yes, it is actually an array of arrays of characters) when the program is executed. Remember the run command in the last section:

```
./go images/mona_lisa.pgm images/mona_inverted.pgm invert
```

The `main` function extracts all the words and stores it in the `argv` array. The `int` variable called `argc` is the *argument count* and tells us how many string arguments (including `"./go"`) are stored in the `argv` array. In the above example, `argc` is 4 and `argv` is an array that contains {"./go", "mona_lisa.pgm", "mona_inverted.pgm", "invert"}. The `main` function perfoms the following tasks:

(1) Checks to make sure that the `argc` value is at least 4, because we need at least 3 values to run the program. (We don't want to use `argv[0]` for anything.)
(2) Sets up the necessary variables: `argv[1]`,`argv[2]`,`argv[3]` as the input filename, the output filename, and which method to call (in this case `invert`).[†]
(3) Initializes `image` with a call to the `ImageType` constructor that takes the input filename as input. After this is called, `image` has access to the `height`, `width`, `grayscale`, and `pixels`: the array of values.
(4) Calls the appropriate function, based on what the user gives as the input parameter corresponding the the specific manipulation. In the example used to run the program, the `invert` function would be the one called. Note that the function is *called* with the `image` variable as its input.
    - Note that in the actual function descriptions, the input parameter has `const` and and ampersand (`&`) added.
    - We will discuss this in detail during lectures. In short, any object should be passed *by reference*, which means that a local copy is not made, nor is a pointer passed to the function. The function simply is allowed access to the original object. However, when passing an object to a function by reference, it is difficult to guarantee that the original object will not be altered *unless* the qualifier `const` is added to the description.
    - If any attempt is made to alter the original object, the compiler will complain.
(5) Lastly, for every task except `makeAscii`, `main` writes the information for the converted image to a new file (with the output filename).

Double-click on the new file (inside the image directory) to see the results. For now, it's only a copy of the original.

invert

FIGURE 1 illustrates what happens when `invert` creates an inverted version of the `mona_list.pgm` file.

Programming notes:

To produce the *inverse* of a grayscale value, you subtract the original value from the *grayscale range*, which is generally 256, but can be any value. For example, if the value in a particular cell is 36, and the `grayscale` value is 256, then the new value of the cell will be $256 - 36 = 220$. You can see that the darkest shades are now much lighter in the example figure.

[†]Be very careful never to access `argv[0]`, which gives you access to "`go`". It is one of the dangerous things about C and C++ that one can access the very executable file that is currently running the program.

(A) before (B) after

FIGURE 1. Inverting the Mona Lisa

verticalFlip

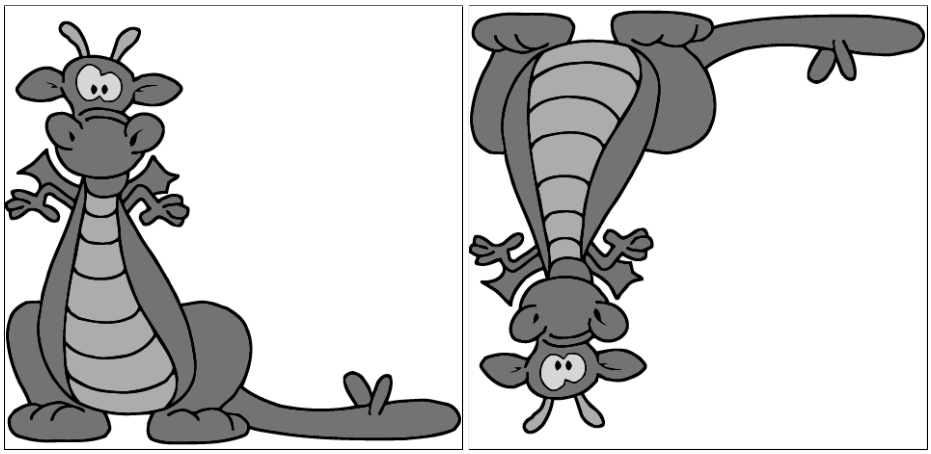
The `verticalFlip` function creates a flipped image. Its left to right pixels do not change, but the up-down pixels are swapped. The command:

```
./go images/dragon.pgm images/dragonVflip.pgm verticalFlip
```

performs the action illustrated in FIGURE 2:

Programming notes:

- Take advantage of the fact that a 2-D array is actually an array of arrays, or in this case, each row contains an array of integers that do not change their position within the row.
- When you think in these terms, you really only need to swap the rows.



(A) before (B) after

FIGURE 2. Vertical flip of the dragon

horizontalFlip

The `horizontallFlip` function creates a flipped image. Its up-down pixels do not change, but its left-right pixels are swapped. The command:

```
./go images/dragon.pgm images/dragonHflip.pgm horizontalFlip
```

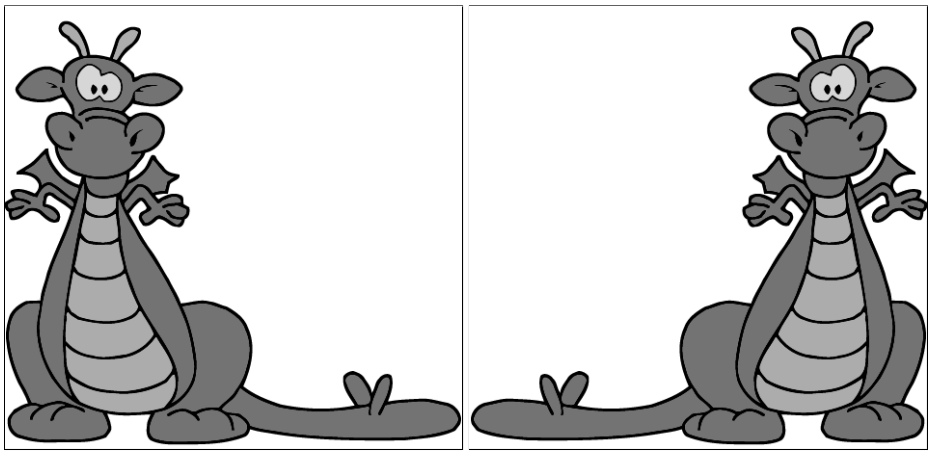
performs the action illustrated in FIGURE 3:

Programming notes:

- This method swaps values in each row in a 2D array. Because each row is really a 1-dimensional array, we are just swapping values in a single array.
- For each swap action, only the column indices are changed.

makeAscii

The `makeAscii` function converts an image to an ASCII image, an image similar to what could be created using a common typewriter. The text is written directly to a text file.



(A) before (B) after

FIGURE 3. Horizontal flip of the dragon

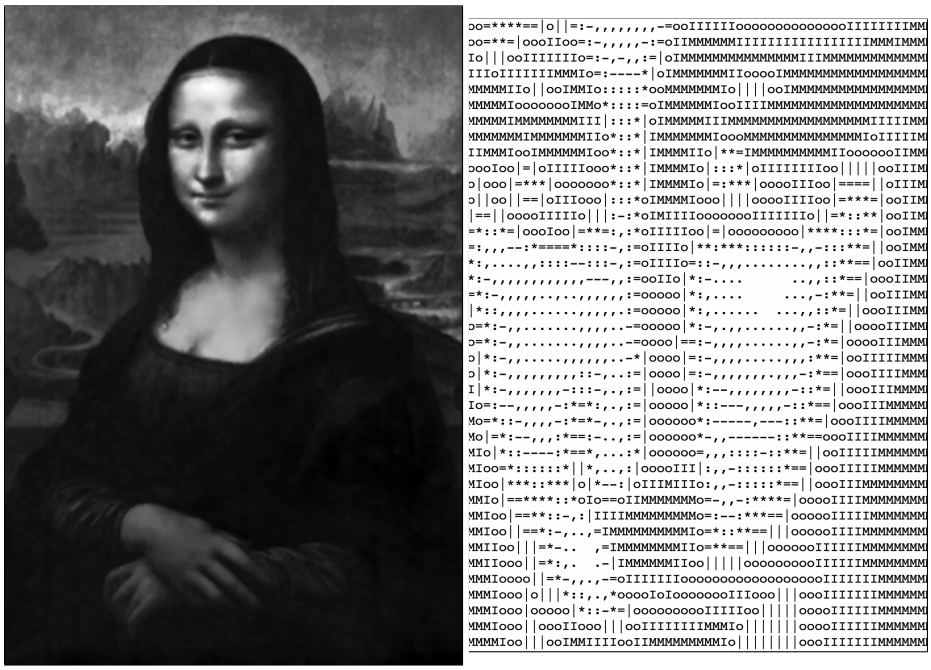
Value	Character
0 to 20	`M`
21 to 40	`L`
41 to 60	`I`
61 to 80	`o`
81 to 100	`|`
101 to 120	`=`
121 to 140	`*`
141 to 160	`:`
161 to 180	`-`
181 to 200	`,`
201 to 220	`.`
221 to 255	`[space]`

TABLE 1. Converting the grayscale values to single characters

The command:

```
./go images/mona_lisa.pgm images/mona_ascii.txt makeAscii
```

performs the action illustrated in FIGURE 4. Because of the size, only a small section of the contents of mona_ascii.txt is shown.



(A) before (B) after

FIGURE 4. Part of the ascii file created from Mona Lisa

Programming notes:

- This is the only function that does not create a new `ImageType` object, nor is there a resulting `pgm` file.
- The output file is a `txt` file filled with characters that line up correctly to create an ascii version, with no whitespace between the characters.

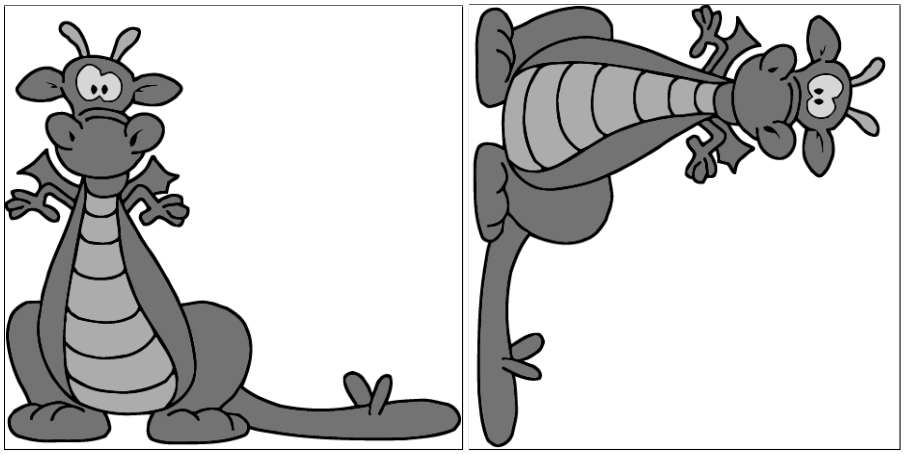
See TABLE 1 on the previous page for the conversion information. If the pixel value is between the numbers on the left, then the character printed to the file is the one on the right.

rotate

The `rotate` method *rotates* the image in a clockwise direction $90°$. The command:

```
./go images/dragon.pgm images/dragonRotate.pgm rotate
```

performs the action illustrated in FIGURE 5:



(A) before (B) after

FIGURE 5. Rotation of the dragon

Programming notes:

If the image is not square, the output image's height is equal to the original image's width, and the width equal to the original image's height.

Use some graph-lined paper and draw the values, mapping where they will go in a clockwise rotation. Try to find the pattern and discover the bit of arithmetic needed for the array indices in the conversion.

scale

The `scale` function enlarges or reduces the image by a scaling factor, provided as an additional argument in the `argv`. The command:

```
./go images/mona_lisa.pgm images/mona_small.pgm scale 0.5
```

performs the action illustrated in FIGURE 6.

Programming notes:

The output image should be as close to a replica of the original image as possible. For scale factors larger than 1, an image will not look as smooth, as more pixels with the same shade of gray create a boxy look. For scale factors < 1, the new image takes the average of the shades of gray from the original pixels into a single pixel, producing a bit of blur.

(A) before (B) after

FIGURE 6. Scaling the Mona List by a factor of 0.5: Right image has been enlarged to show the change in resolution

When scaling down, some rows and columns do not divide evenly. However, because we are casting the scaled rows and cols to an `int`, we may end up with missing rows or columns at the edge of the new image. This is fine, and likely will not be detectable by the human eye. The most important thing is that we will not fall into the trap of trying to access an *out-of-range* pixel from the original array.

This task requires that the user add another item to the `go` command, so you will need to check for that. Note that the `argv` array contains `string` objects, so picking up the double value of `argv` requires that we convert this to a `double`. There are some ways to do this, but the safest way is to create a `StringStream` object, which acts similarly to the input streams from either a file or the console. That way, you can take advantage of the natural conversion when you get a numerical value. The following code sample shows you how:

```
#include <stringstream>
   using std:istringstream

   int main() {
```

```
    string valueAsString = "3.7";
    double number;
    istringstream(valueAsString) >> number;
    cout << (2*number) << end;

    return 0;
}
```

Submission

Submit the following completed file to the Assignment folder on VIULearn.

- `imageConversions.cpp`

A note about academic integrity

It is OK to talk about your assignment with your classmates, and you are encouraged to design solutions together, but each student must implement their own solution.

Grading

Marks are allocated for . . .

- No errors during compilation of the source code.
- The function headers must be exactly as specified and they must perform as specified. Be sure to read the **imageConversion Details** in the specification document.
- The `main` method handles the extraction of the information in the `args` array and calls the required methods.
- Some user mistakes are handled with helpful information. For example, if the user does not execute the program with the correct number of arguments, or the input file does not exist, then the program stops after a helpful message is printed to the console.
- Style of the source code meets the requirements outlined in the Coding conventions document available in the General Resources fold on the course site.