

# MP 2 Image Manipulation II

**Extra credit: Tuesday, September 8 at 11:59 PM**

**Due: Tuesday, September 15 at 11:59 PM**


[Doxygen for MP 2](#)

In this MP (machine problem) you will:

- do cool stuff with images
- use existing C++ functions and classes
- write your own classes
- use inheritance
- write constructors and the Big Three
- make use of `const` correctness and reference variables
- create a `Makefile`

For this MP you will write a subclass of the `PNG` class that provides additional functions for manipulating the image (MP 2.1). Then you will write a `Scene` class that stores multiple images and their coordinates, and can output an image of the whole scene (MP 2.2). Finally, in order to compile your code, you will write a `Makefile`. We strongly recommend implementing, compiling, and testing the files in 2.1 before starting 2.2. We say this because you cannot compile or test the code for the `Scene` class until you implement the `Image` class.

In addition to the usual (incomplete) testing instructions provided on the MP 2.1 and MP 2.2 instructions, we have uploaded the same, intentionally insufficient, test cases into the monad framework on SVN. Although these are the same tests included in the `mp2` folder, running them through monad may help you catch grading-related errors before they occur (such as incorrect file names). To run these, see [MP Grading](#).

 We will not be testing your `Makefile` in the provided test cases but it will be tested in the final grading runs.

## Video FAQ

- [What's the deal with `operator\(\)`?](#)
- [Pointers on Pointers](#)
- [Makefile Dependencies](#)

## Checking Out the Code

To check out the provided code simply run

from your `cs225` directory.

This should update your directory to contain a new directory called `mp2`. These files are used for both parts of the MP: MP 2.1 and MP 2.2.

## Requirements

These are strict requirements that apply to **both** parts of the MP. Failure to follow these requirements may result in a failing grade on the MP.

- You must name all files, public functions, public member variables (if any exist), and executables **exactly** as specified in the Doxygen.
- Your public function signatures must match ours **exactly** for full credit.
- Your code must produce the **exact** output that we specify: nothing more, nothing less. Output includes files such as `Images`.
- Your code must compile on the EWS machines using **clang++**. Being able to compile on a different machine is **not** sufficient.
- Your code must be submitted correctly by the **due date and time**. Late work is not accepted.
- Your code must not have any memory errors or leaks for full credit.

For part one of MP 2, you will be implementing the `Image` class.

## The Makefile

In this MP, you will practice what you learned from MP 1 about `Makefiles` by creating your own. You are allowed to use any of the `Makefiles` we have given you as a starting point for creating the `Makefile` for MP 2. Be sure to use the correct file dependencies and file names. Your `Makefile` must use `clang++` for the compiler and linker.

We recommend using the following compiler flags:

Flag Name	Description
<code>-c</code>	compile (will link without this)
<code>-g</code>	include debugging information
<code>-O0</code>	disable optimizations
<code>-Wall</code>	enable all warnings

You'll also need some flags from MP 1 and `lab_intro`:

Flag Name	Description
-----------	-------------

<code>-std=c++1y</code>	use C++14
<code>-stdlib=libc++</code>	use libc++ as the standard library
<code>-lc++abi</code>	(linker (LDFLAGS) only) link with the c++abi library (needed to use libc++)

Your `Makefile` should initially compile the EasyPNG library (RGBAPixel and PNG), the `Image` class, and `testimage.cpp`. Then link all their object files to form an executable named `testimage`.

If you do this correctly, you should be able to compile (after implementing the `Image` class) using the following command:

```
make testimage
```

TERMINAL

We also strongly recommend creating a target named `clean` that removes all the object files (`*.o`) in the working directory and the `testimage` executable. Be careful not to remove any of your source code. If you add this functionality, the command `make clean` should automate this task.

## MP 2.1: The Image Class

An `Image` object is a subclass of the `PNG` class. This means it inherits all the member functions of the `PNG` class; so anything you could do with a `PNG`, you can also do with an `Image`. You will be writing additional functions for the `Image` class as documented in the [Doxygen](#). Make `Image` a subclass of `PNG` by writing:

```
class Image : public PNG
{
    // the function definitions from the Doxygen go here
};
```

C++

To see all the required functions, read the [Doxygen](#)!

To create the `Image` class, you should create a header file (`image.h`) that contains a declaration of the `Image` class and a source file (`image.cpp`) that contains the implementation of the `Image` class. Make sure to name your files exactly as specified. The file names should not be capitalized.



### Double Inclusion Guards

Do not forget to use double inclusion guards!!! A common error in the past has been a compilation failure due to doubly included header files.

Double inclusion guards look like the following (included in a `myfile.h` file):

```
#ifndef MYFILE_H
#define MYFILE_H
// ... code goes here
#endif
```

C++

This is general for any `*.h` file you create — for instance, you should have these in your `image.h` file (make sure to change `MYFILE_H` out with something more relevant!).

## MP 2.1: Testing

After compiling, you can run the Image tests with the following command:

```
./testimage
```

TERMINAL

If execution goes smoothly, images named `brightened.png`, `flipped.png`, and `inverted.png` will be created in your working directory. These files can be `diff`d with their respective solution images: `soln_brightened.png`, `soln_flipped.png`, and `soln_inverted.png`. Do not assume that if two images look similar that they match perfectly. Use a utility such as `diff` to check for correctness.

These tests are **deliberately insufficient**. We **strongly** recommend augmenting these tests with your own.

If the `diff` utility reports differences in your image, there is also a helpful utility available on the EWS machines which compares two image files with more detailed output. The command to run this utility is:

```
compare image1 image2 outputImage
```

TERMINAL

In the command above, `compare` is the utility name, `image1` and `image2` represent the names of the two image files you want to compare, and `outputImage` will be the name of the image file created by the utility to illustrate the differences between the input files. The following is an example of running the above command to compare the files `outHalf.png` and `soln_outHalf.png`:

```
compare outHalf.png soln_outHalf.png diff.png
```

TERMINAL

This example will produce a new file called `diff.png` which will illustrate the differences between the two input files by laying one image over the other image and applying a red tint to any pixels which are different.

To test your code using Monad, just go into your `monad` directory, and then type

## MP 2.1: Extra Credit Submission

For a few bonus points, you can submit the code you have implemented and tested for part one of MP 2. You must submit your work before the extra credit deadline (given above). Although this is optional, we encourage everyone to do this for a couple reasons:

- if you get a sufficient grade on the submission, you will receive bonus points to improve your grade.
- regardless of the quality of your work, you will get feedback that can be used to improve your grade on the required submission of MP 2.

To facilitate anonymous grading, **do not** include any personally-identifiable information (like your name, your UIN, or your NetID) in any of your source files. Instead, before you hand in this assignment, create a file called `partners.txt` that contains only the NetIDs of people in your collaboration group (if it exists), one per line. If you worked alone, include only your own NetID in this file. We will be automatically processing this information, so do not include anything else in the file. (If we must manually correct your submission, you may lose points.) As always, if you're working in a group, each group member must hand in the assignment. (Failure to cite collaborators violates our academic integrity policy; we will be aggressively pursuing such violators.)

We will be using Subversion as our hand-in system. Our grading system will checkout your most recent (**pre-deadline**) commit for grading. Therefore, to hand in your code, all you have to do is commit it to your Subversion repository.

Be sure your working directory is the `mp2` directory that was created when you checked out the code. To hand in your code, you first need to add the new files you created to the working copy of your repository with the following commands:

```
svn add Makefile
svn add image.cpp
svn add image.h
svn add partners.txt
```

TERMINAL

To commit your changes to the repository type:

```
svn ci -m "mp2.1 submission"
```

TERMINAL

## Grading Information

The following files will be used to grade MP 2.1:

- `Makefile`
- `image.cpp`

- `image.h`
- `partners.txt`

All other files including any testing files you have added will not be used for grading.

## MP 2.2: The Scene Class

For part two of MP 2, you will be modifying the `Makefile` you created in MP 2.1 and implementing the `Scene` class. The `Scene` class depends on the `Image` class that was written in part one. Be aware that completing the first part is required to get any credit for MP 2.

Your goal in this part of the MP is to make a scene composed of a collection of images. To do so, you will create a class `Scene` that will maintain an array of pointers to `Image` objects. Each `Image` in the `Scene` will have an index, an  $x$ -coordinate, and a  $y$ -coordinate. The member functions described below will support creating, modifying, and drawing the collection of `Images` in the `Scene`. The user can request an `Image` object in which all the `Images` in the scene are drawn, in order from index 0 to the maximum index, at their coordinates. Only one `Image` can occupy each index, because internally the images should be stored as an array of pointers to `Image` objects with `NULL` pointers at the vacant indices.

To implement the `Scene` class, you will write a header file that contains a declaration of the `Scene` class (`scene.h`) and a source file that contains the implementation of the `Scene` class (`scene.cpp`).

To see all the required functions, check out the [Doxygen for MP2](#).

The destructor, copy constructor, and assignment operator are known as the Big Three. The latter two must make totally independent copies of the source. That is, after the function returns, changing any thing in the source `Scene` should not result in any change of this `Scene` and vice versa. Remember the `Image` class inherited a copy constructor from the `PNG` class.

## MP 2.2: Modifying the `Makefile`

You will be using the same `Makefile` you created in MP 2.1, but you need to extend its functionality.

Namely, you need to add rules to compile the `Scene` class and link the EasyPNG Library, the `Image` class, the `Scene` class, and `testscene.cpp` to form an executable named `testscene`.

If you do this correctly, you should be able to compile (after implementing the `Scene` class) using the following command:

```
make testscene
```

TERMINAL

If you created the `clean` rule, don't forget to add the `testscene` executable to the list of files to remove.

In addition, we recommend adding a target `all`, such that the command `make all` would construct both `testimage` and `testscene`. This is actually really easy to do. More than likely



you have defined some macro for the executable names. Replace the `testimage` and `testscene` with their corresponding macros in the following and add it to your `Makefile`:

```
all: testimage testscene
```

## MP 2.2: Testing

After compiling, you can run the Scene tests with the following command:

```
./testscene
```

TERMINAL

If execution goes smoothly, images named `scene.png`, `scene2.png`, `scene3.png`, `scene4.png`, `scene5.png`, `scene6.png`, `scene7.png`, and `scene8.png` will be created in your working directory. These files can be diffed with their respective solution images: `soln_scene.png`, `soln_scene2.png`, `soln_scene3.png`, `soln_scene4.png`, `soln_scene5.png`, `soln_scene6.png`, `soln_scene7.png`, and `soln_scene8.png`. Do not assume that if two images look similar that they match perfectly. Use a utility such as `diff` to check for correctness.

These tests are **deliberately insufficient**. We strongly recommend augmenting these tests with your own.

If the `diff` utility reports differences in your image, there is also a helpful utility available on the EWS machines which compares two image files with more detailed output. The command to run this utility is:

```
compare image1 image2 outputImage
```

TERMINAL

In the command above, `compare` is the utility name, `image1` and `image2` represent the names of the two image files you want to compare, and `outputImage` will be the name of the image file created by the utility to illustrate the differences between the input files. The following is an example of running the above command to compare the files `outHalf.png` and `soln_outHalf.png`:

```
compare outHalf.png soln_outHalf.png diff.png
```

TERMINAL

This example will produce a new file called `diff.png` which will illustrate the differences between the two input files by laying one image over the other image and applying a red tint to any pixels which are different.

## Handing in your code

To facilitate anonymous grading, **do not** include any personally-identifiable information (like your name, your UIN, or your NetID) in any of your source files. Instead, before you hand in this

assignment, create a file called `partners.txt` that contains only the NetIDs of people in your collaboration group (if it exists), one per line. If you worked alone, include only your own NetID in this file. We will be automatically processing this information, so do not include anything else in the file. (If we must manually correct your submission, you may lose points.) As always, if you're working in a group, each group member must hand in the assignment. (Failure to cite collaborators violates our academic integrity policy; we will be aggressively pursuing such violators.)

We will be using Subversion as our hand-in system this semester. Our grading system will checkout your most recent (**pre-deadline**) commit for grading. Therefore, to hand in your code, all you have to do is commit it to your Subversion repository.

Be sure your working directory is the `mp2` directory that was created when you checked out the code. To hand in your code, you first need to add the new files you created to the working copy of your repository. You only need to add files to the working copy once. Therefore, if you added some of these files during the extra credit submission, then you don't need to add them again. However, you would still need to add `scene.h` and `scene.cpp` to the working copy.

The following commands add the files to your `mp2` working copy:

```
svn add image.cpp
svn add image.h
svn add partners.txt
svn add scene.cpp
svn add scene.h
svn add Makefile
```

TERMINAL

To commit your changes to the repository type:

```
svn commit -m "mp2 submission"
```

TERMINAL

## Grading Information

The following files are used to grade `mp2`:

- `image.cpp`
- `image.h`
- `partners.txt`
- `scene.cpp`
- `scene.h`
- `Makefile`

All other files including any testing files you have added will not be used for grading.

## Good Luck!



