CSC 230 Project 2

Image Processing Pipeline

For this project, you're going to write three simple programs that perform basic image-processing procedures. We'll have a program named **brighten** that can brighten an image, another named **border** that can add a little border around the edge of any image, and we'll have one named **blur** that can apply a blur operation to an image.

These programs will all read image data from standard input and write the output image to standard output. We can use I/O redirection to get them to read and write to files instead of actually reading and writing in the terminal.

Here's a small image we use to test our programs.

Sample input image of a jack-o-lantern

In the starter for this project, this image is stored in a file named **image-5.ppm**. If you run the **brighten** program as follows, it will brighten this image a little bit and write the result to a file named **output.ppm**. The result should look like the following.

```
$ ./brighten < image-5.ppm > output.ppm
```

Brightened version of the jack-o-lantern image.

The **border** program will add a extra rows and columns of black pixels on all four sides of the image (giving it a little border). If you run **border** as follows, you should get an output image that looks like the following.

```
$ ./border < image-5.ppm > output.ppm
```

Jack-o-lantern image with a little border around the edge.

The **blur** program applies a little Gaussian blur to the image, hiding some details and giving it a softer appearance. If you run **blur** as follows, you should get an output image that looks like the following.

```
$ ./blur < image-5.ppm > output.ppm
```

Blurry version of the jack-o-lantern image.

Since all three of our programs read and write on standard input and standard output, we can use a feature of the shell to apply multiple image operations at once. On a Unix machine, you can use what's called a pipe to get the output of one program to go directly as input to some other program. You can connect up any number of programs like this, with the output of each program automatically being sent as input to the next one. Using this, we could run our programs as follows (the vertical bars tell the shell to make a pipe):

```
$ ./brighten < image-5.ppm | ./blur | ./border > output.ppm
```

Here we're running the **brighten** program with input from the **image-5.ppm** file. Then, we use a pipe to send the output of **brighten** as input to **blur**. This should give us a slightly brighter, blurry version of the original image. Finally, we send the output from **blur** as input to **border**. This should put a border around the image. The output from **border** is sent to the file, **output.ppm**. If we view this output image, it should look like:

Jack-o-lantern with all three operations applied.

To help get you started, we're providing you with a few partial implementation files, a test script, and several test input files and expected output files. See the Getting Started section for instructions on how to set up your development environment so that you can be sure to submit everything needed when you're done.

This project supports a number of our course objectives. See the Learning Outcomes section for a list.

Rules for Project 2

You get to complete this project individually. If you're unsure of what's permitted, have a look at the academic integrity guidelines in the course syllabus.

In the design section, you'll see some instructions for how your implementation is expected to work. Be sure you follow these rules. It's not enough to just turn in a working program; your program has to follow the design constraints we've asked you to follow. This helps to make sure you get some practice with the parts of the language we want to make sure you've seen.

Requirements

Requirements are a way of describing what a program is supposed to be able to do. In software development, writing down and discussing requirements is a way for developers and customers to agree on the details of a system's capabilities, often before coding has even begun. Here, we're trying to demonstrate good software development practice by writing down requirements for our program, before we start talking about how we're going to implement it.

Input / Output Image Format

Our programs will read and write images using standard input and standard output. We haven't learned how to do file I/O yet, but we can use I/O redirection to get our program to write to any file we want. We'll run them like the following, getting help from the shell to get them to read and write to files instead.

```
$ ./brighten < image-5.ppm > output.ppm
```

Our programs will read and write images in a simple, uncompressed, text-based format called PPM. This format isn't as popular as some other image formats (particularly because it isn't very space efficient), but there are a few programs that support it, and it's really easy to read and write.

The text PPM image format starts with a 3-line header like the following. The `P3` on the first line is something called a magic number (not the same idea a magic number in your source code). It identifies the type of file this is. Here, the P3 says that this file is an image in plain (text) PPM format, the next line gives the size of the image, 60 pixels wide and 45 pixels tall for this example. The third line gives the maximum intensity for the red, green and blue color components of the image's pixels. All the images we work with will be at least 1 pixel wide and 1 pixels tall, and they'll all have 255 as their maximum intensity.

```
P3
60 45
255
```

A PPM image gives the color values for all the pixels after the header. This starts with the color for all the pixels on the top row of the image, left-to-right. This is followed by color values for the next row of the image, and so on. Each pixel's color is given as three integer values, a red intensity ranging from 0 (no red) to 255 (maximum red), then a green intensity, then blue (also between 0 and 255). For example, the following text:

```
P3
4 4
255
255 255   0 255 255   0 255 255   0 255 255   0
255 255   0 255   0   0 128 128 128 255 255   0
255 255   0   0 255   0   0   0 255 255 255   0
255 255   0 255 255   0 255 255   0 255 255   0
```

.. is a PPM description of the following image (enlarged considerably in this picture to show the individual pixels).

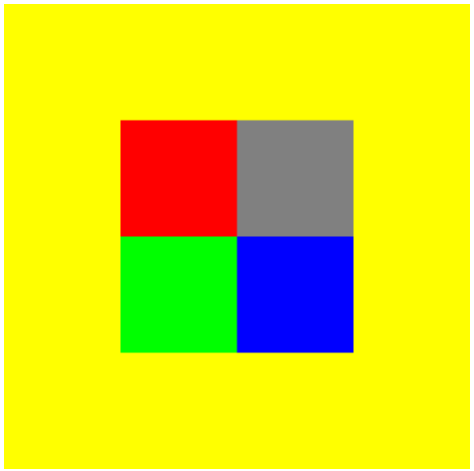


Image represented by the PPM example given above.

In this example, all the pixels around the edge of the 4 X 4 image have a value of "255 255 0". This represents red and green at maximum intensity with no blue, so the image has a yellow border. Inside this border, the upper left pixel (the second group of three values on the second line of pixel data) has a value of "255 0 0", so we get a red pixel. Just to the right, we have a pixel with all colors at half intensity, so we get gray. The middle two pixels on the next row represent a green pixel " 0 255 0" and a blue pixel " 0 0 255".

The text PPM format permits any amount of space between the integers describing pixel colors. In the example above, I put the each row of the image on a separate line of text, and added extra spaces between some values to make the columns line up.

When you're reading in an image, your program should be able to tolerate any amount of whitespace before any of the numeric values (but the P3 will always be right at the start of the input). Being able to tolerate any amount of space actually makes the image parsing code easier to write, since scanf() automatically skips whitespace for most conversion specifiers.

For output images, we're going to use a simpler, more strict organization for the image. An output image should have the header printed on 3 lines, like the example above. The magic number should be on the first line, then the width and height with one space between them. Then, the maximum pixel intensity should be on the next line. This value should always be 255.

For the pixel data, we'll print the values for each row on its on line. Values should have one space between them (and no extra space at the end of each line).

So, for our programs, the sample PPM file shown above would look like the following if one of our programs printed it as output.

```
P3
4 4
255
255 255 0 255 255 0 255 255 0 255 255 0
255 255 0 255 0 0 128 128 128 255 255 0
255 255 0 0 255 0 0 0 255 255 255 0
255 255 0 255 255 0 255 255 0 255 255 0
```

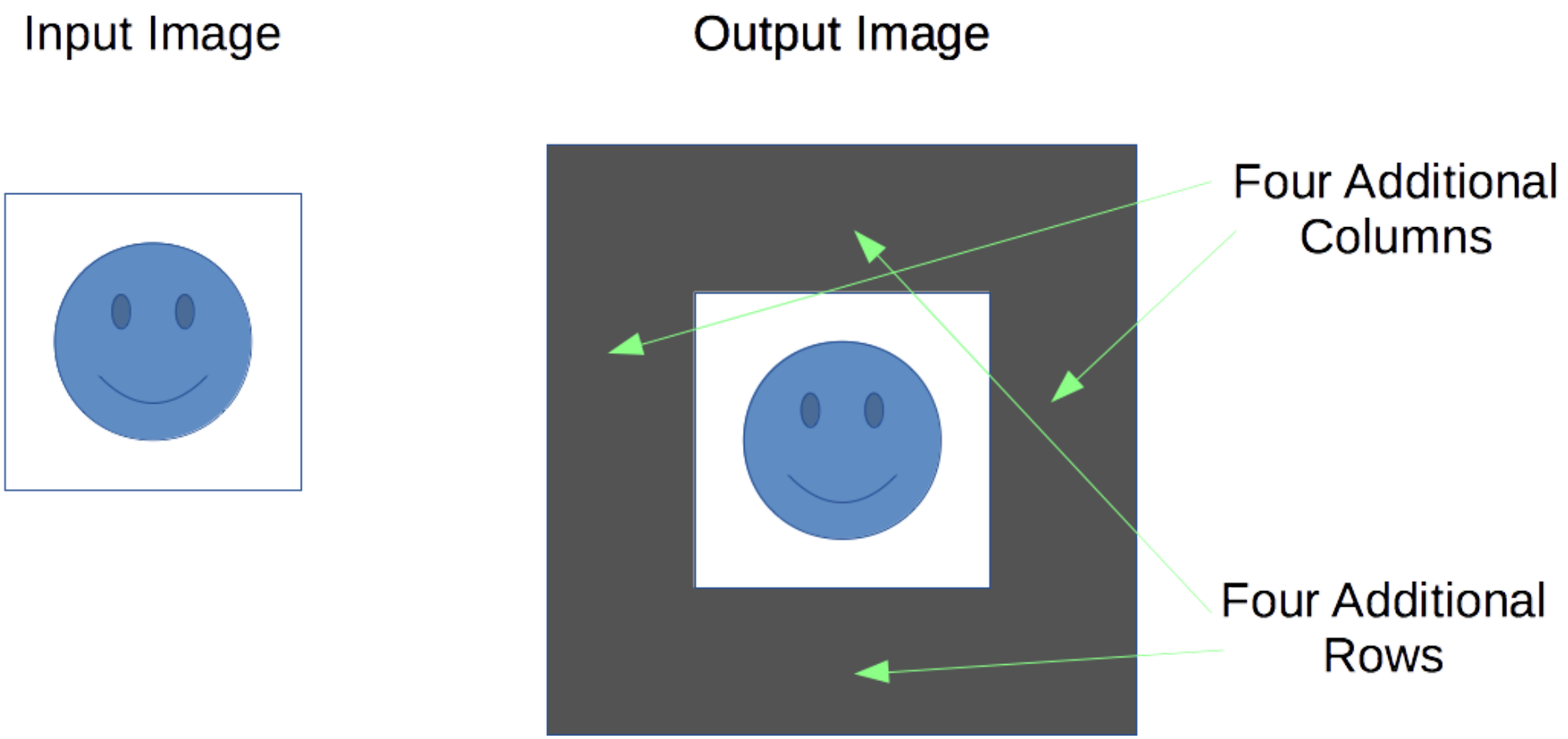
Brighten Program

The **brighten** program is probably the easiest. To brighten an image, we'll just add 32 to the intensity values (the red, green and blue values) for every pixel. For some colors, this could give us a value that's too large (greater than 255). If this happens, we'll just use 255 for that value (so, all the intensity values max out at 255).

After modifying the pixel data, the **brighten** program will write out the result to standard output.

Border Program

The **border** program is probably the one at medium difficulty. Its job is to add a 4-pixel border around the entire image. It will make a new image with four new columns of pixels on the left and right and four new rows of pixels at the top and bottom. It will copy the original image to the middle of this larger image and color all the pixels outside black.



Adding a border around an image

After making a new, larger image with a 4-pixel border, the **border** program will write out the result to standard output.

Blur Program

The **blur** program is probably the most difficulty one. It's job is to apply a 3 x 3 Gaussian blur to all the pixels of the input image. There's a Wikipedia page about this and related operations, if you'd like some more references: Wikipedia Page on Convolution

This program will apply a blur by building a new image where the intensity values at each pixel computed as a weighted sum of the original intensity values of that pixel and its 8 neighbors. We'll do this separately for red, green and blue. So, for some pixel at row *r* and column *c*, the blurred pixel will have a red component computed as a weighted sum of the red components of the input pixel at row *r*, column *c* and 8 surrounding pixels. It will have a green component computed as a weighted sum of the green components of the input pixel at row *r*, column *c* and 8 surrounding pixels. Similarly for its blue component.

The following figure shows the weights we'll use. If we're computing the red intensity for some pixel at row *r* and column *c*, the 4 in the middle gives the weight we use for that pixel itself. We multiply its red intensity by 4 and add it to a sum. The 2 above that is the weight we use for the pixel above it, so we multiply that pixel's red intensity by 2 and add it to the sum. The 1 in the upper left corner is the weight we use for the pixel up and to the left by 1. We multiply that pixel's red intensity by 1 and add it to the sum. Once we've done this for all 9 of the pixels in this little 3x3 neighborhood, we divide the sum by 16 and that gives us the red intensity for pixel at row *r*, column *c* in the blurred output image.

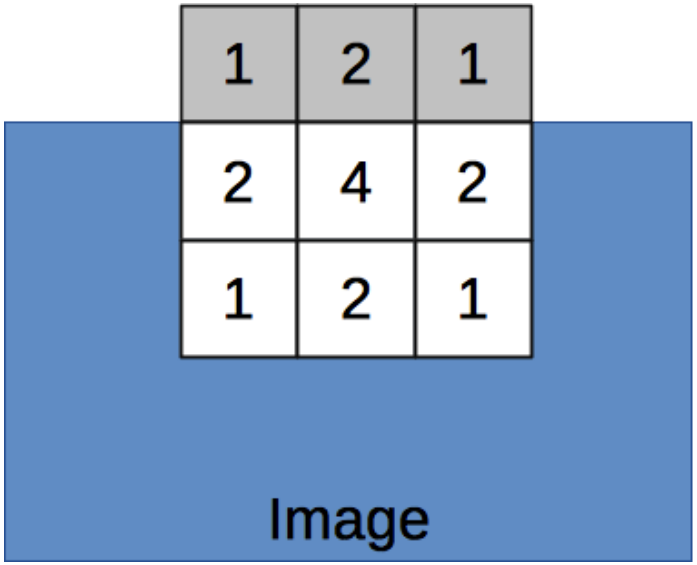
$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

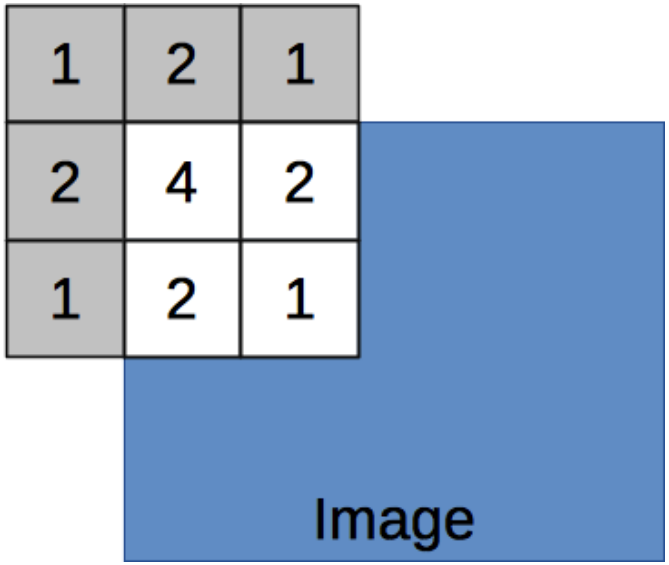
Weights for a neighborhood around a pixel

We just have to go through this procedure for all 3 color components in every pixel of the input image to get a blurred version. You can see from this procedure why its important to store the input and output images separately in memory. Otherwise, if you just modified the input image in place, you would be changing some pixel data before you were done using. it.

The procedure we're doing here is an example of convolution. It's a common component in lots of image processing procedures.

During the blur operation, we're going to have to give special treatment for pixels at the edge of the image. Like the next two pictures show, pixels at the edge or corner of an image may not have 8 neighbors.





Neighborhoods for pixels near the edge of the image

We're going to handle cases like this by just including neighbors that actually exist. So, in the figure on the left, when we're computing a weighted sum for the pixel in the middle, it's missing its three neighbors above. On the right, the pixel in the middle doesn't have any neighbors above or to the left. In these cases, we'll just add up a weighted sum for the pixels that are part of the image, the ones colored white in the picture.

But, if we leave some pixels out of the sum, then dividing by 16 at the end wouldn't be correct. This divide-by-sixteen is done so that all the weights sum to 1 (after the division). Instead, we'll just divide by the total weight for which there were pixels to add. For example, in the picture on the left, we're missing the top three neighbors, with a total weight of 4. That leaves six remaining pixels with a total weight of 12. That's what we should divide when we compute the weighted average. For the picture on the right the pixels included in the weighted sum would get a total weight of 9. For this case, that's what we should divide by.

When you're computed the weighted sums for the blur operation, do this with integer arithmetic rather than using a floating point type. This means we'll get some truncation when we do the division operation (rounding would probably be better). The reason for using integer math is to try to make sure your output matches the expected output. When you're using a floating-point type like double, small differences in how you compute a value can affect the rounding behavior. This could cause some of the numbers in your output image to disagree with the ones I got. I don't think we'll get this as much with integer math.

Invalid Input

If the given input image isn't valid, your program will exit immediately (without printing any output). We'll use the exit status of the program to indicate what was wrong.

If there's something wrong with the header of the input image, your programs will terminate with an exit status of 100. Errors in the header could include a bad magic number at the start, one of the other fields not being a valid integer or the maximum intensity value not being given as 255.

If there's something wrong with the pixel data of the input image, your programs will terminate with an exit status of 101. Errors in the pixel data could include numbers that are too small or too large, or text that wasn't a valid integer.

Design

The program will be implemented in four components.

- image.h / image.c
  This component provides functions for reading and writing PPM images and making sure they are in the the right format. It is used by all three of the other components.

- brighten.c
  This is the implementation of the **brighten** program. It contains the main() function for **brighten**, along with any other utility functions you need to implement.

- border.c
  This is the implementation of the **border** program. It contains the main() function for **border**, along with any other utility functions you need to implement.

- blur.c
  This is the implementation of the **blur** program. It contains the main() function for **blur**, along with any other utility functions you need to implement.

Image Representation

We will represent the pixel data in an image using a 3-dimensional array of unsigned char. Three dimensions is a lot, but it maps well to the image data. We'll use the first dimension to index through all the rows of the image. The second dimension will be for all the columns in the image, and the last dimension will be for the three color components (red, green and blue). So, the first two dimensions will depend on the size of the image, but the 3rd dimension will always jut give us three values per pixel.

The unsigned char type is a good match for storing the pixel data; its range of 0 .. 255 is exactly what we need. As you're doing math operations on a pixel, you can temporarily copy these unsigned int values to a wider type like int, then just assign them back to the unsigned char pixel data when you're done.

Required Functions

You can use as many functions as you want to solve this problem, but you'll need to implement and use at least the following ones. These are all provided by the image component, but they will be used by the other three components.

- `void checkType()`
  This function is used to read the magic number at the start of an input image. It checks to make sure the value is correct and exits (with the right exit status) if it's not. This is probably the first function you will call as you start reading an image.

- `int readDimension()`
  This function is used to read the width and height in the image header. It's called once for each and returns the value it reads. It also error checks these values before returning.

- `void checkRange()`
  This is for reading the maximum intensity value at the end of the image header. It checks to make sure it is 255 and exits appropriately if not.

- `void readPixels( int height, int width, unsigned char pix[ height ][ width ][ DEPTH ] )`
  This is for reading all the pixel data for an image. These values are stored in the given array (so, the caller has to allocate this array before calling this function).

- `void writeImage( int height, int width, unsigned char pix[ height ][ width ][ DEPTH ] )`
  This function is for writing the output image to standard output. It writes out the header for the PPM image, then writes out all the pixel data in the format described by the requirements section.

Globals and Magic Numbers

You should be able to complete this project without creating any global variables. The function parameters give each function everything it needs.

Be sure to avoid magic numbers in your source code. Use the preprocessor to give a meaningful name to all the important, non-obvious values you need to use.

You don't have to define named constants for all the weights used in the blur operation. You can just build a little 2D array to hold the weights and fill in all of them using array initialization syntax.

Extra Credit

PPM files normally support comments. To make them easier to parse, we're using files that don't contain any comments. For up to 8 points of **extra credit**, you can add support for comments in the header. You don't have to do anything with the comments (and you don't have to copy them into the output); you just have to skip past them to get to the parts of the image data you need.

In a PPM file, a comment starts with a pound sign. Then, everything up to the end of that line is considered a comment.

We'll say a comment can show up anywhere in the image header between the magic number at the start and up to the maximum intensity (255) at the end. There won't be comments inside numbers like the width or the height, but comments could show up between any of these values. You could even have multiple comments in a row. So, for example, the following should be a valid input file if you implement support for comments.

```
P3
# Here's a comment.
2 2
255
  0   0   0 255   0   0
  0 255   0   0   0 255
```

The starter includes a couple of test cases you can try if you do the extra credit. Here's how you can try them out.

```
# The first test uses the brighten program
# It should run successfully and output should
# match the first ec-output file.
$ ./brighten < ec-image-1.ppm > output.ppm
$ echo $?
$ diff output.ppm ec-expected-1.ppm

# The next test uses the blur program
# It should run successfully and output should
# match the second ec-output file.
$ ./blur < ec-image-2.ppm > output.ppm
$ echo $?
$ diff output.ppm ec-expected-2.ppm
```

Build Automation

You get to implement your own Makefile for this project (called `Makefile` with a capital 'M', no filename extension). Since we're building three programs for this project, your Makefile will need to have rules to build each of the programs. When you run make, you can give it a command-line option to tell it which target to build (that's what the test script does when it tries out your programs). For example, if you run make as follows it, you're asking it to use the rules you've written to build the target named **blur**.

```
$ make blur
```

Your make file should be smart enough to separately compile each of your source files into an object file, then link the right objects together to make each of the executables you need. It should correctly describe the project's dependencies, so targets can be rebuilt selectively, based on what parts of the project have changed. The Makefile should compile each source file with the -Wall, -std=c99, and -g options. This will be useful when you try to use `gdb` to help debug the program.

The Makefile should also have a rule with the word `clean` as the target. This will let the user easily delete any temporary files that can be rebuilt the next time make is run (e.g., the object files, executables and any temporary output files).

A clean rule can look like the following. It doesn't have any prerequisites, so it's only run when it's explicitly specified on the command-line as a target. It doesn't actually build anything; it just runs a sequence of shell commands to clean up the project workspace. Your clean rule will look like the following (we used the <tab> notation to remind you where the hard tabs need to go). In your clean rule, replace things like all-your-object-files with a list of the object files that get built as part of your project.

```
clean:
<tab>rm -f all-your-object-files
<tab>rm -f your-executable-program
<tab>rm -f any-temporary-output-files
<tab>rm -f anything-else-that-doesn't-need-to-go-in-your-repo
```

To use your `clean` target, type the following at the command line, ***but first be sure you have committed all of the files that you need to your github repo***, as the -f flag forces the removal without asking you if it's OK.

```
$ make clean
```

Testing

The starter includes a test script, along with test input files and expected outputs for the program. When we grade your programs, we'll test it with this script, along with a few other test inputs we're not giving you.

To run the automated test script, you should be able to enter the following:

```
$ chmod +x test.sh # probably just need to do this once
$ ./test.sh
```

This will automatically build all three of your programs using your Makefile and see how they behaves on all the provided test inputs.

You probably won't pass all the tests the first time. In fact, until you have a working Makefile, you won't be able to use the test script at all.

If you want to compile one of your programs by hand, the following command should do the job.

```
$ gcc -Wall -std=c99 -g image.c brighten.c -o brighten
$ gcc -Wall -std=c99 -g image.c border.c -o border
$ gcc -Wall -std=c99 -g image.c blur.c -o blur
```

To run your program, you can do something like the following. The test script prints out how it's running your program for each test case, so this should make it easy to check on individual test cases you're having trouble with. The following commands run the border program with input read from the `image-4.ppm` and with output written to the file, `output.ppm`. Then, we check the exit status to make sure the program exited successfully (it should for this particular test case). Finally, we use diff to make sure the output we got looks exactly like the expected output.

```
$ ./border < image-4.ppm > output.ppm
$ echo $?
0
$ diff output.ppm expected-06.txt
```

If your program generated the correct output, diff shouldn't report anything. If your output isn't exactly right, diff will tell you where it sees differences. For the non-trivial input files, this could be a lot of output. There's an ImageComp program described below that can help with this. Also, keep in mind that diff may complain about differences in the output that you can't see, like differences in spacing or a space at the end of a line.

Examining your Output

For some of the test cases, your output files will be so large it will be difficult to see where your program isn't exactly right. To help, we wrote a simple Java program called ImageComp.java. If you run this program with just one PPM file as a command-line argument, it will display just that image, magnified 10X so you can see individual pixels better. If you run it like the following, with two PPM files on the command line, it will let you look at each of the two files individually, along with a difference image, where any pixels that don't match between the images are colored in pink.

```
$ java ImageComp expected-06.ppm output.ppm
```

I don't think the ImageComp program is smart enough to skip comments in the PPM header. So, if you do the extra credit, you'll have to use a different viewing program to look at the test inputs. Your outputs won't contain comments, so ImageComp should work fine on those.

The PPM image format is a little uncommon, but there are some image viewer programs that support it. Installing one of these can give you another way of looking at your program's output.

If you are working on a Linux desktop machine, you may already have some programs you can use. Gimp and gthumb will display PPM images. Since gimp is available for lots of platforms, that's an option even if you're not on a Linux machine. It also looks like LibreOffice will open PPM files in the drawing program. On a Windows system, IrfanView is a small program that can view files in this format. We can add more viewing programs to this list if people report others they've had success with.

Sample Input Files

With the starter, we're providing a number of ppm image files that are used as test inputs. Here's what each test input is like.

1. A tiny, 1x1 image with one black pixel (so, the red, green and blue intensities all zero).
2. A tiny, 2x2 image containing a black and white checkerboard (upper-left and lower-right pixels white, the other two black).
3. A tiny, 2x2 image containing a black pixel, a red one, a green one and a blue one.
4. A small, 21x16 image containing a little grid of different colors.
5. A small, 200x198 image containing a picture of a jack-o-lantern.
6. A bad image file with an invalid width in the header.
7. A bad image file with an invalid magic number on the first line.
8. A bad image file with an invalid RGB value in the image pixel data.
9. A bad image file with an RGB value that's too large.

Using these inputs, the **test.sh** script runs your programs through several tests. Here's what each test does. This can help you think about and examine what's going wrong if your program's behavior isn't exactly what the test script is expecting.

1. This test runs the brighten program on image-1.ppm
2. This test runs the brighten program on image-2.ppm
3. This test uses a pipe to run the brighten program 3 times on image-4.ppm (making it even brighter)
4. This a test for error handling. It runs the brighten program on image-6.ppm.
5. This test runs the border program on image-3.ppm
6. This test runs the border program on image-4.ppm
7. This test uses a pipe to run the border program 3 times on image-4.ppm (giving it an even wider border).
8. This a test for error handling. It runs the border program on image-7.ppm.
9. This test runs the blur program on image-4.ppm
10. This test runs the blur program on image-5.ppm
11. This test uses a pipe to run the blur program 3 times on image-4.ppm (making it even more blurry)
12. This a test for error handling. It runs the blur program on image-8.ppm.

13. This a test for error handling. It runs the blur program on image-9.ppm.
14. This test uses a pipe to run image-5.ppm through the brighten program, then the blur program, then the border program.
15. This test is like the previous one, but for this test, we apply the 3 image operations in reverse.

Keep in mind, when we're grading your programs, we'll test them on these tests and on a few other tests that we're not providing. This is a good reason to think about possible tests or inputs that aren't done by the **test.sh** program.

Grading

The grade for your programs will depend mostly on how well they function. We'll also expect your code to compile cleanly, we'll expect it to to follow the style guide and the expected design, and we'll expect a working Makefile.

- Working Makefile, including dependencies and a make clean rule: **8 points**
- Compiling cleanly on the common platform: **10 points**
- Correct behavior on all test cases: **80 points**
  This represents 20 points for each of the programs individually, and 20 more points for passing tests that involve two or more of your programs together.
- Source code follows the style guide: **20 points**
- Support for comments in the header: **8 extra credit points**
- Deductions
  - Up to **-50 percent** for not following the required design.
  - Up to **-30 percent** for failing to submit required files, submitting files with the wrong name or having extraneous files in the repo.
  - Up to **-20 percent** penalty for late submission.

Getting Started

To get started on this project, you'll need to clone your NCSU github repo and unpack the given starter into the p2 directory of your repo. You'll submit by committing files to your repo and pushing the changes back up to the NCSU github.

Cloning your Repository

Everyone in CSC 230 has been assigned their own NCSU GitHub repository to be used during the semester. It already has subdirectories (mostly empty) for working on each of the remaining projects. How do do you figure out what repo you've been assigned? Use a web browser to visit `github.ncsu.edu`. After authenticating, you should see a drop-down menu over on the left, probably with your unity ID on it. Select "engr-csc230-fall2020" from this drop-down and you should see a repo named something like "engr-csc230-fall2020/unity-id" in the box labeled Repositories over on the left. This is your repo for submitting projects.

I've had some students in the past who do not see the organization name, "engr-csc230-fall2020" in their drop-down menu. I'm not sure why that happened, but when they chose "Manage organizations" near the bottom of this drop-down, it took them to a list of all their repos, and that list did include "engr-csc230-fall2020". Clicking on the organization name from there took them to a page that listed their repo. If you're having this problem, try the "Manage organizations" link.

You will need to start by cloning this repository to somewhere in your local AFS file space using the following command (where unity-id is your unity ID, just like it appears in your repo name):

```
$ git clone https://unity-id@github.ncsu.edu/engr-csc230-fall2020/unity-id.git
```

This will create a directory with your repo's name. If you `cd` into the directory, you should see directories for each of the projects for the class. You'll want to do your development for this assignment right under the `p2` directory. That's where we'll expect to find your project files when we're grading.

Unpack the starter into your cloned repo

You will need to copy and unpack the project 2 starter. We're providing this file as a compressed tar archive, starter2.tgz. You can get a copy of the starter by using the link in this document, or you can use the following curl command to download it from your shell prompt.

```
$ curl -O https://www.csc2.ncsu.edu/courses/csc230/proj/p2/starter2.tgz
```

Temporarily, put your copy of the starter in the p2 directory of your cloned repo. Then, you should be able to unpack it with the following command:

```
$ tar xzvpf starter2.tgz
```

Once you start working on the project, be sure you don't accidentally commit the starter archive to your repo (that would be an example of an extraneous file that doesn't need to be there). After you've successfully unpacked it, you may want to delete the starter from your p2 directory, or move it out of your repo.

```
$ rm starter2.tgz
```

Instructions for Submission

If you've set up your repository properly, pushing your changes to your assigned CSC230 repository should be all that's required for submission. When you're done, we're expecting your repo to contain the following files. You can use the web interface on github.ncsu.edu to confirm that the right versions of all your files made it.

- `image.h / image.c` : Image component, completed by you.
- `brighten.c` : Brighten program, completed by you.
- `border.c` : Border program, completed by you.
- `blur.c` : Blur program, created by you.
- `Makefile` : Makefile for efficiently building your program, created by you.
- `image-*.ppm` : Test image input files, provided with the starter.
- `expected-*.ppm` : Expected image output files, provided with the starter.
- `test.sh` : test script, provided with the starter.
- `.gitignore` : a file for this project, telling git some files to *not* commit to the repo.

Pushing your Changes

To submit your solution, you'll need to first commit your changes to your local, cloned copy of your repository. First, you need to add any new files you've created to the index. Running the following command will stage the current versions of a file in the index. Just replace the `some-file-name` with the name of the new file you want commit. You only need to do this once for each new file. The `-am` option used with the `commit` below will tell git to automatically commit modified files that are already being tracked.

```
$ git add some-file-name
```

When you're adding new files to your repo, you can use the shell wildcard character to match multiple, similar filenames. For example, the following will add all the input files to your repo.

```
$ git add input-*.txt
```

When you start your project, don't forget to add the `.gitignore` file to your repo. Since its name starts with a period, it's considered a hidden file. Commands like `ls` won't show this file automatically, so it might be easy to forget.

```
$ git add .gitignore
```

Before you commit, you may want to run the git status command. This will report on any files you are about to commit, along with other modified files that haven't been added to the index yet.

```
$ git status
```

Once you're ready to commit, run the following command to commit changes to your local repository. The `-am` option tells git to automatically commit any tracked files that have been modified (that's the `a` part of the option) and that you want to give a commit message right on the command line instead of starting up a text editor to write it (that's the `m` part of the option).

```
$ git commit -am "<a meaningful message about what you're committing>"
```

Beware, you haven't actually submitted anything for grading yet. You've just put these changes in your local git repo. To push changes up to your repo on github.ncsu.edu, you need to use the push command:

```
$ git push
```

Feel free to commit and push as often as you want. Whenever you've made a set of changes you're happy with, you can run the following to update your submission.

```
$ git add any-new-files
$ git status
$ git commit -am "<a meaningful message about what you're committing>"
$ git push
```

Keeping your repo clean

Be careful not to commit files that don't need to be part of your repo. Temporary files or files that can be easily re-generated will just take up space and obscure what's really changing as you modify your source code. And, the NCSU github site puts a file size limit on what you can push to your repo. Adding files you don't really need could create a problem for you later.

The `.gitignore` file helps with this, but it's always a good idea to check with `git status` before you commit, to make sure you're getting what you expect.

Checking Jenkins Feedback

We have created a Jenkins build job for you. Jenkins is a continuous integration server that is used by industry to automatically build and test applications as they are being developed. We'll be doing the same thing with your project as you push changes to the NCSU github. This will provide early feedback on the quality of your work and your progress toward completing the assignment.

The Jenkins job associated with your GitHub repository will poll GitHub every two minutes for changes. After you have pushed code to GitHub, Jenkins will notice the change and automatically start a build process on your code. The following actions will occur:

- Code will be pulled from your GitHub repository
- A testing script will be run to make sure you submitted all of the required files, to check your code against parts of the course style guidelines and to try out your solution on each of our provided test cases

Jenkins will record the results of each execution. To obtain your Jenkins feedback, do the following tasks (remember, after a push, you may have to wait a couple of minutes for the latest results to appear):

- Go to Jenkins for CSC230. Your web browser will probably complain that this site doesn't have a valid certificate. That's OK. Tell your browser to make an exception for this site and it should let you continue to the site.
- You'll need to authenticate with your unity ID and password.
- Click the project named p2-*unityid*
- There will be a table called *Build History* in the lower left, click the link for the latest build
- Click the *Console Output* link in the left menu (4th item)
- The console output provides the feedback from compiling your program and executing the provided test cases. If the bottom of the output says you failed some tests, there should be one or more lines earlier in the output that starts with four stars. These should say something about what parts of the test your program failed on.

Succeeding on Project 2

Be sure to follow the style guidelines and make sure your program compiles cleanly on the common platform with the required compile options. If you have your program producing the right output, it should be easy to clean up little problems with style or warnings from the compiler.

Be sure your files are named correctly, including capitalization. We'll have to charge you a few points if you submit something with the wrong name, and we have to rename it to evaluate your work.

We'll test your submission with a few extra test cases when we're grading it. Maybe try to think about sneaky test cases we might try, like behaviors that are described in this write-up but not actually tested in any of the provided test cases. You might be able to write up a simple test case to try these out yourself.

There is a 24 hour window for **late submissions**. Use this if you need to, but try to keep all your points if you can. Getting started early can help you avoid this penalty.

Learning Outcomes

The syllabus lists a number of learning outcomes for this course. This assignment is intended to support several of theses:

- Write small to medium C programs having several separately-compiled modules
- Correctly identify error messages and warnings from the preprocessor, compiler, and linker, and avoid them.
- Interpret and explain data types, conversions between data types, and the possibility of overflow and underflow
- Use the C preprocessor to control tracing of programs, compilation for different systems, and write simple macros.
- Write, debug, and modify programs using library utilities, including, but not limited to assert, the math library, the string library, random number generation, variable number of parameters, standard I/O, and file I/O

- Use simple command-line tools to design, document, debug, and maintain their programs.
- Use an automatic packaging tool, such as make or ant, to distribute and maintain software that has multiple compilation units.
- Use a version control tools, such as subversion (svn) or git, to track changes and do parallel development of software.
- Distinguish key elements of the syntax (what's legal), semantics (what does it do), and pragmatics (how is it used) of a programming language.