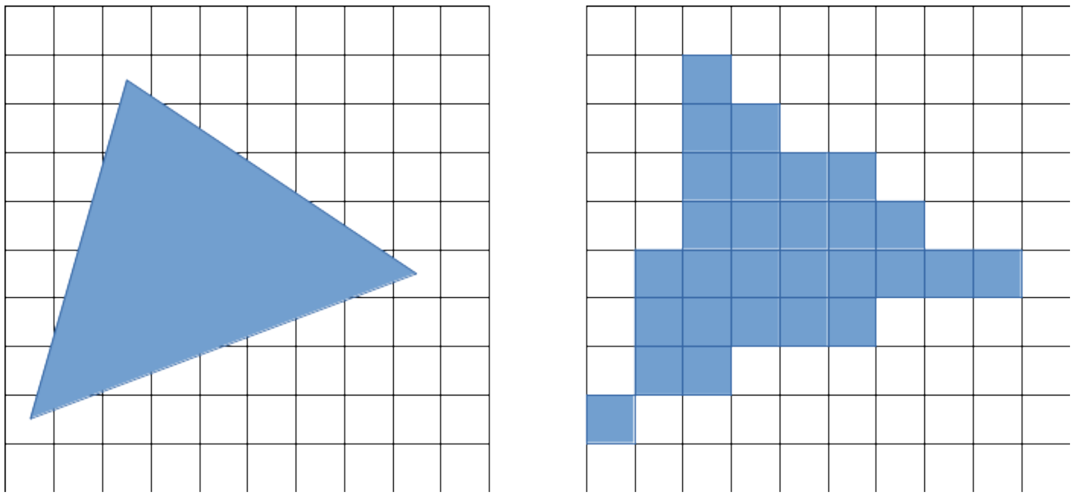
CSC 230 Project 2

Triangle-Drawing Program

For this project, you're going to write two versions of a simple, image-drawing program. Both versions create an image with a single triangle drawn on it, like the figure below. The picture on the left shows a 10 X 10 pixel image, with a blue triangle you're expected to draw. The picture on the right shows how your program will draw this triangle, by filling in pixels with the triangle's color.



Drawing a blue triangle by filling in pixels that fall inside it.

The two versions of your program will create the output image in slightly different versions of the PPM image format. This is a very simple image format, with no compression. One of your programs will output the text version of this format, where pixel colors are written out as text. The other version will output pixel colors in binary instead of text. This will reduce the image size, but still not nearly as much as you'd expect for a real, compressed image format.

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, you can 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.

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 a 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.

Program Input

The `triangle` program will take several integer and floating point values as input, read from standard input (i.e., from the terminal). First, it will expect two positive integers giving the width and the height of the desired output image (measured in pixels).

After the image size, the program will expect six floating point values (that should be parsed using the `double` type), *x1 y1 x2 y2 x3 y3*. These give the locations of the triangle's vertices in counter-clockwise order. The first two values, *x1 y1*, are the location of the first vertex, the next two, *x2 y2*, give the location of the second vertex, and the last two, *x3 y3* give the location of the third vertex.

Finally, the program should expect integers giving the Red, Green and Blue (RGB) components for the color the triangle should be drawn in. Each color value should be between 0 and 255, inclusive.

The following sample input describes the picture given at the start of the assignment. We're asked to generate a 10 X 10 image containing a triangle with vertices at (2.5, 1.5), (0.5, 8.5) and (8.5, 5.5). The triangle should be drawn by filling in pixels with a red value of 114, a green value of 159 and a blue value of 207 (so, a light-blue triangle).

```
10 10
2.5 1.5 0.5 8.5 8.5 5.5
114 159 207
```

The program is expected to detect invalid input, input values that are out of range or that don't parse as integer or floating point values as expected. If there's something wrong with the input, the program should terminate immediately with an exit status of `EXIT_FAILURE`.

Text Image Output

The `triangle` program will write its output image to 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 it like the following to trick it into doing file I/O instead.

```
$ ./triangle < input.txt > output.ppm
```

The `triangle` program will write images in a simple, uncompressed 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 generate. We'll place some additional constants on our output images, to make it easier to check the correctness of the output.

The text PPM image format starts with a 3-line header like the following. The `P3` on the first line 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).

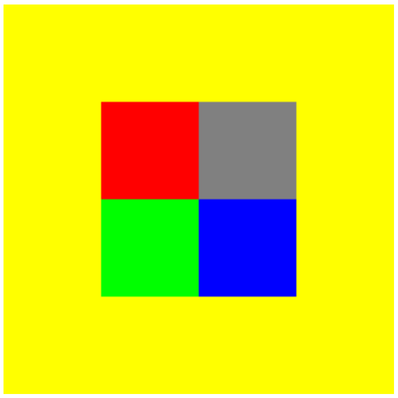


Image represented by the PPM example given above.

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.

For our output images, we're going to use a simpler organization for the pixel data. The PPM format requires that we print the pixel data from top row of the image to the bottom row, with each row of pixels given left to right, but we don't have to organize these values with each row of the image on its own output line. Instead, we're just going to fit as many output RGB components as we can on each output line, as long as each output line is no more than 70 characters long. On each output line values will be separated by a single space (so, no extra space at the end of the line). The last output line will end with a line terminator (`\n`), just like we'd normally expect for a text file.

In our output format, the small image shown above would get written like the following. The first output line contains the first row of pixel data, along with some of the second row. The next line finishes up the second row, then gives the whole third row and the start of the last row. The last line gives the last few values for the fourth row of the image.

```
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
```

Binary Image Output

One version of our program will output images using the slightly more compressed binary PPM format. The header for this format is almost the same as the text format, except that it starts with "P6" instead of "P3". The main difference is in how the RGB values are encoded. In the binary format, each color component is written out as a single byte, starting right after the newline character at the end of the header. Pixel colors are still given in the same order as the text format, but with each component written out as a single byte and without the need for delimiters (spaces) between values, this format yields a much smaller file size than the text format (but still not nearly as small as a real, compressed format would give you).

In the binary PPM format, the sample image above would look more like the following:

```
P6
4 4
255
     binary pixel data
one byte per color component
```

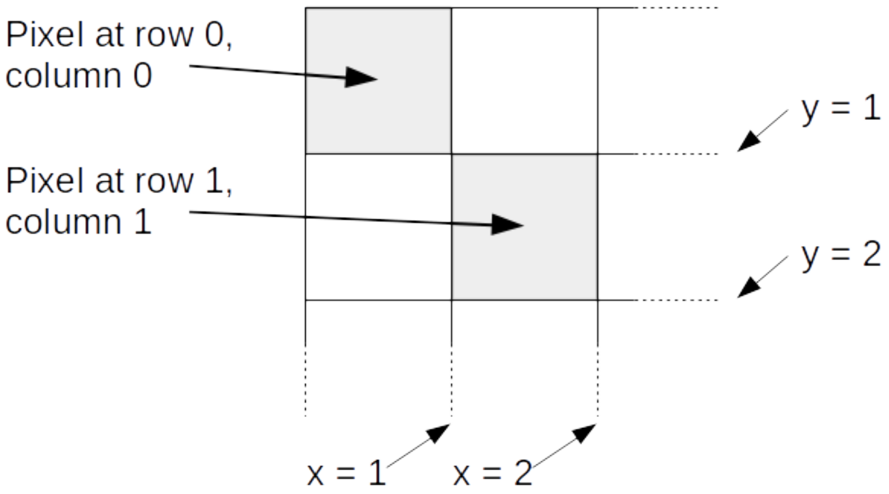
Or, if you wanted to see what the binary encoding for the pixel data looks like, you could look at this image file using `hexdump -C`. You'd get something like the following. You can see the header is in plain text, but after the 255 and the newline character at the end of the header, there are 4 X 4 X 3 bytes containing the color components of the image.

```
00000000  50 36 0a 34 20 34 0a 32  35 35 0a ff ff 00 ff ff  |P6.4 4.255......|
00000010  00 ff ff 00 ff ff 00 ff  ff 00 ff 00 00 80 80 80  |................|
00000020  ff ff 00 ff ff 00 00 ff  00 00 00 ff ff ff 00 ff  |................|
00000030  ff 00 ff ff 00 ff ff 00  ff ff 00                 |...........|
```

Triangle Drawing

We'll draw the output image in black (0 for each of the the RGB values), except for pixels that are in the triangle. Those will get the color specified by the last three values from the input.

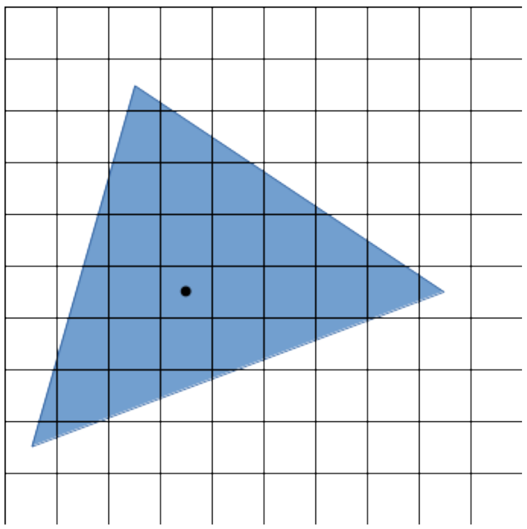
We'll think of each pixel as a little 1 X 1 square. The left edge of the leftmost column of pixels will be at x = 0. The right edge of this column will be at x = 1, right at the left edge of the next column.



Geometry of pixels in the image.

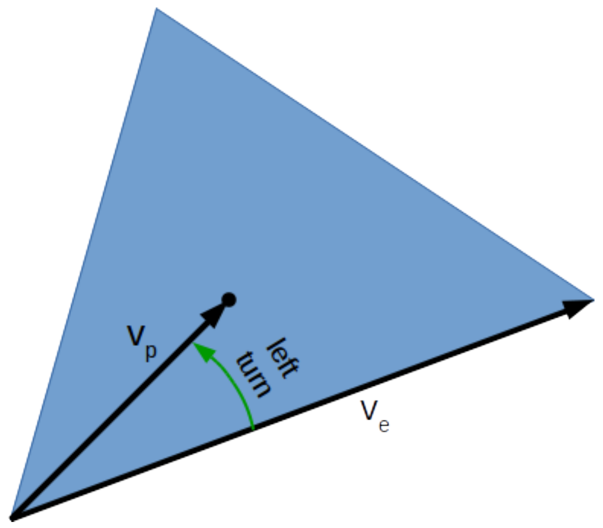
Vertically, we'll think of the Y axis as pointing down (a typical convention for talking about images). The top edge of the first row of pixels will be at y = 0, and bottom edge of this row will be at y = 1, right at the top edge of the next row.

We'll use the center of each pixel to decide whether that pixel is inside the triangle. In the following figure, for example, the pixel in row 5, column 3 (both counting from zero) has its center at coordinates x = 3.5, y = 5.5.



Center point used to decide whether a pixel is inside the triangle

The following figure shows how we'll decide whether a pixel is inside the triangle. We'll compute a vector between two consecutive vertices on the triangle (taken in counter clockwise order, labeled V_e in the figure). We'll also compute a vector between the first of these vertices and the center of some pixel (labeled V_p in the figure). The sign of the cross product of V_e and V_p will tell us whether the vector to the pixel is a left turn compared to the vector along the edge of the triangle.



Left turn test, for checking if a pixel is inside the triangle.

If you don't remember how to compute the cross product, it's pretty easy. Normally, it's defined for three-dimensional vectors, but it simplifies down to two dimensions if we just assume everything is in the XY plane. Let's say vector V_e has X and Y components *(x_e, y_e)* and vector V_p is defined as (x_p, y_p). The cross product of these vectors can be calculated as $x_e\, y_p - y_e\, x_p$. If this quantity is less than zero, then V_p is a left turn from V_e. This may be backward from what you expect (if you're familiar with the cross product), but it's because our Y axis is pointing down instead of up.

We'll apply this left turn test for each edge of the triangle. If they all report that the vector to the pixel is to the left of the vector along the edge of the triangle, we'll say the pixel is inside the triangle. Otherwise, we'll say it's not. Pixels right on an edge of the triangle will yield a cross product of zero. We'll say these pixels are also inside the triangle.

Design

The `triangle` program will be implemented using two header files and four implementation files. The two header files are mostly written for you, but you get to fill in some of the comments.

- geometry.h
  This is the header file for the geometry component. It contains prototypes for functions that help decide whether a given point is inside the triangle being drawn.
- geometry.c
  This is the implementation file for the geometry component. It contains definitions for functions that help decide whether a given point is inside the triangle.
- encoding.h
  This is the header file for the encoding component. It contains prototypes for functions that let us write out the image in either the text format or the binary format.
- text.c
  This file is a little unusual. It's one of two implementation files that goes with the encoding.h header. Normally, an implementation and its header would share the same base name, but here we're using the encoding.h header to describe the interface for the encoding functions, with either text.c or binary.c being linked in to supply the definitions of these functions. This file contains implementations of the encoding functions to write out the image in the text format.
- binary.c
  This file is like text.c, but it provides implementations of the encoding functions that write out an image in our binary

format.

- triangle.c
  This is the top-level, main component. It contains the main() function and is responsible for reading the input and lookking for errors in the input. Then, it generates the output image, with help from the other two components.

Encoding Selection

You can see we have two different implementations of the encoding component. We'll select the encoding we want by linking together an executable that has just the one of these two implementations (either text.c or binary.c, but not both). If we link with the text implementation, we'll get a text encoding. If we link with the binary implementation, we'll get binary. The testing section below has some simple gcc commands for compiling this project either way.

Image Generation

We haven't learned about arrays yet, so we don't have a good way to store the whole image in memory. Instead, we'll color in the triangle as we output the image, testing each pixel right before we print it to see if it's inside the triangle.

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 four.

- `bool leftOf( double xa, double ya, double xb, double yb, double x, double y );`
  This function helps to determine whether a pixel is inside the triangle. It implements the left-turn test described in the Requirements. The first four parameters give it the location of two consecutive vertices of the triangle (in counter-clockwise order), and the last two parameters give the location of the center of a pixel. This function is part of the `geometry` component.

- `bool inside( double x1, double y1, double x2, double y2, double x3, double y3, double x, double y );`
  This function uses leftOf() to determine whether a pixel (the last two parameters) is inside the triangle. It's part of the `geometry` component.

- `void printHeader( int width, int height );`
  This function is part of the `encoding` component. It's implemented in the `text.c` to print a header for a text encoding and in the `binary.c` file, to print a header for a binary encoding. The parameters give the size of the image (which is required for the header).

- `void printValue( unsigned char c );`
  This function is part of the `encoding` component. It prints one of the RGB components (given via the parameter). It's implemented in the `text.c` to print a value in a text encoding and in the `binary.c` file, to print a value for a binary encoding.

The printValue() function in `binary.c` can be really simple. It just has to print out the given value as a single char (you know a couple of functions from the standard library that can do this for you). The version of printValue() in `text.c` is going to need to be a little smarter. It has to print out the given value in text (which is easy), but it has to make sure it doesn't exceed the maximum line length of 70 characters. To do this, it will need some persistent state. This is a good job for a static local variable. Use one to keep up keep up with how many characters long the current output line is. Then, before you print the next value, you can decide if it's going to make the current output line too long, and start printing on a new output line if you need to. You'll need to fiure out how many digits it takes to print the value you're about to print (so you can tell if it will fit on the output line). You'll have to write a little code to figure this out. If it helps, remember that all the RGB values must be between 0 and 255.

Floating Point Representation

Use values of type double for any floating point calculations you need to do.

Globals and Magic Numbers

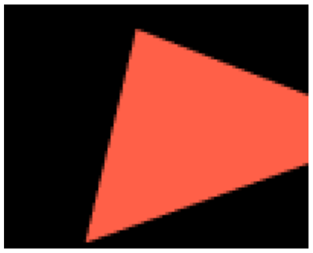
You should be able to complete this project without using any global variables. The function parameters give each function everything it needs.

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 (e.g., the values used for your program's exit status).

Extra Credit

For up to 8 points of **extra credit**, you can implement a compile-time option for anti-aliasing the output image, to hide the stair-steps along the edge of the triangle. We can do this with super-sampling, testing multiple locations inside each pixel to determine how much of the pixel is actually inside the triangle.

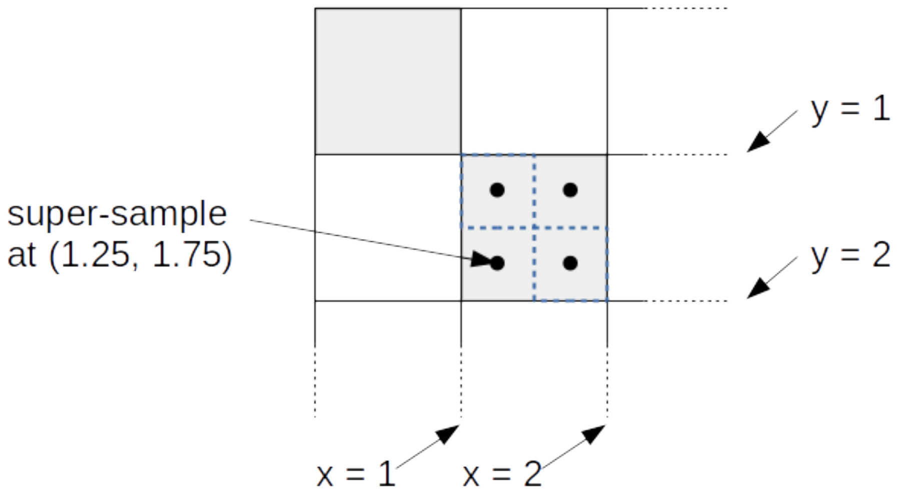
The following (enlarged) image shows the effects of anti-aliasing. Pixels along the edge of the triangle are shaded to hide the stair-step from one pixel to the next.



Anti-aliasing to smooth triangle edges.

In the starter, you'll see that there's a preprocessor constant called `SSAMP` defined in the geometry header. It's set to one, but it's possible to set it to larger values using a compiler option. Compiling with the `-DSSAMP=3` flag will override its value from the header file, setting it to 3 instead. If you do the extra credit, use this constant to determine how many horizontal and vertical samples to use for each pixel.

For super-sampling, we'll think of each pixel as consiting of SSAMP X SSAMP smaller, square regions. We'll test the center of each of these to see if it's inside the triangle. Then, we'll shade the pixel based on how many of these samples fall inside the triangle. That way, pixels on the edge of the triangle will be shaded with an intermediate color between the background color (black) and the color of the triangle.



Illustraton of 2x2 super-sampling

The picture above shows 2x2 supersampling, with each pixel bronken into 4 smaller regions and the center of each tested against the triangle. If SSAMP was 3, you'd sample each pixel using the centers of 3 X 3 = 9 smaller squares, shading it based on how many of these samples were inside the triangle. To choose the right shade, compute a weighted average of the background color (black) and the triangle color given in the input. The weight for the background will be the number of samples that fall outside the triangle (divided by SSAMP^2), and the weight for the triangle color will be the number of samples that fall inside the triangle (divided by SSAMP^2). This weighted average yields a floating point intensity for each

RGB value. Use the round() function to round it to the nearest integer intensity for each RGB value. The round() function is part of the math library, so you'll need to include the math.h header and link with the math library (-lm) if you do the extra credit.

If you do the extra credit, we have two sample input input files and two expected output files you can compare your output against. The input files are slightly modified versions of input-3.txt and input-7.txt, modified to avoid cases where rounding error might affect your output. The sample inputs are input-ec1.txt, with an expected output of expected-ec1.ppm. There's also input-ec2.txt with an expected output of expected-ec2.ppm. Both of these expected outputs use the text encoding and a SSAMP value of 3.

In the original version of this assignment, I just used input-3.txt and input-7.txt as examples for the extra credit. However, these contained some cases where the output you get might depend on rounding behavior for floating point types (I had some code to detect this, but, apparently, I either hadn't re-compiled with this code or I hadn't re-run it on the examples, since I missed these cases).

Testing

The starter includes a test script, along with test input files and expected outputs for both versions of your program. When we grade your program, 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 each version of your program and see how they behave on all the provided test inputs.

You probably won't pass all the tests the first time. The test script reports how it's running your program for each test. This should help you see how you can run a particular test yourself, to try to figure out what's going wrong.

If you want to compile the text-encoding version by hand, the following command should do the job. The -lm option tells the compiler to link with the math library, which you probably only need if you did the extra credit.

```
$ gcc -Wall -std=c99 -g triangle.c geometry.c text.c -o triangle -lm
```

If you want to compile the binary-encoding version, the following should work. With the binary version, we're using a slightly different name for the executable, so we can keep both versions of the program around at the same time.

```
$ gcc -Wall -std=c99 -g triangle.c geometry.c binary.c -o btriangle -lm
```

To run the text-encoding version, you can do something like the following. These commands run the program with input read from the `input-3.txt` test case 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.

```
$ ./triangle < input-3.txt > output.ppm
$ echo $?
0
$ diff expected-t3.ppm output.ppm
```

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. 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. Also, for the larger output images, it may be difficult to tell what part of the image diff is complaining about. The Image Comparison program described below may help with this.

To run the binary-encoding version of your program, you can do something like the following. Like the previous example, we're running against one of the test inputs and checking the exit status and the contents of the output after the program finishes.

```
$ ./btriangle < input-6.txt > output.ppm
$ echo $?
0
$ diff expected-b6.ppm output.ppm
```

Since these output files are binary, diff probably won't be able to show you where your output isn't exactly right. Using the Image Comparison program could help here, or looking at your actual and expected output using `hexdump -C`.

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're providing 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 2X. 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-t3.ppm output.ppm
```

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 common platform desktop machine (or probably any other Linux machine), you 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

Both the text-encoding and binary-encoding programs can be tested against the same input files. The starter includes the following test inputs, all named like input-x.txt.

1. A tiny little 4 X 4 image, where the triangle overlaps the two columns of pixels on the left.
2. A tiny little 4 X 4 image, where the edge of the triangle goes down the diagonal.
3. The same 10 x 10 image shown earlier in this assignment.
4. A 10 x 10 image where the entire image is inside a big triangle.
5. A 10 x 10 image with the entire triangle is off the edge of the image.
6. A 100 x 100 image with a triangle that fits entirely on the image.
7. A 100 x 80 image with a triangle that falls partially off the image.
8. An invalid input, where one of the triangle vertices isn't a valid floating-point number.
9. An invalid input, where the width of the image is negative.

Grading

The grade for your program will depend mostly on how well it functions. We'll also expect your code to compile cleanly, to follow the style guide and to follow the expected design.

- Compiling cleanly on the common platform: **10 points**
- Both versions of the program behave correctly on all tests: **60 points**
- Source code follows the style guide: **20 points**
- Deductions
  - Up to **-50 percent** for not following the required design.
  - Up to **-30 percent** for failing to submit required files or submitting files with the wrong name.
  - 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 checking files into 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. 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-spring2018" from this drop-down and you should see a repo named something like "engr-csc230-spring2018/csc230-???-??" in the box labeled Repositories over on the right. The part at the end like "csc230-???-??" is your *repo_name* for this class.

You will need to start by cloning this repository to somewhere in your your local AFS file space using the following commands, where `unity_id` is your unity ID, and `repo_name` is the repo you've been assigned.

```
$ unset SSH_ASKPASS
$ git clone https://unity_id@github.ncsu.edu/engr-csc230-spring2018/repo_name.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

Make sure you're in `p2` directory in 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. After you download this file, you can unpack its contents into your `p2` directory. You can do this like you unpacked the starter from project 1. If you are logged in on one of the common platform systems, you can save yourself a few steps by unpacking the starter directly from our official copy in AFS.

```
$ tar xzvpf /afs/eos.ncsu.edu/courses/csc/csc230/common/www/proj/p2/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.

- `triangle.c` : Source file, created by you.
- `encoding.h` : Header file, provided with the starter and modified by you.
- `text.c` : Source file, created by you.
- `binary.c` : Source file, created by you.
- `geometry.h` : Header file, provided with the starter and modified by you.
- `geometry.c` : Source file, created by you.
- `input-*.txt` : Test input files, provided with the starter.
- `expected-t*.txt` : Expected text-encoding output files, provided with the starter.
- `expected-b*.txt` : Expected binary-encoding output files, provided with the starter.
- `test.sh` : test script, provided with the starter.
- `.gitignore` : a file for this project, telling git what 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. As with the clone command above, you may want to unset the `SSH_ASKPASS` variable before you run the push command. You should only need to do this once for each login session. 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

It will take me a few days to get our Jenkins systems working. Once they're set up, I'll send out a note on Piazza, and these instructions should work.

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 application 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.

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.

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.