1 Introduction
In this lab you will use recursion to plot a Sierpinski triangle, illustrated below.

![Sierpinski triangle of order 9](image)

2 A Recursive Design
The Sierpinski triangle is another example of a fractal pattern like the H-tree pattern from the course notes. The Polish mathematician Waclaw Sierpiński described the pattern in 1915, but it has appeared in Italian art since the 13th century. Though the Sierpinski triangle looks complex, it can be generated with a short recursive program. Your task is to write a program `sierpinski.cpp` with a recursive function `sierpinski()` and a `main()` function that calls the recursive function once, and plots the result using standard drawing.

Think recursively: `sierpinski()` should draw one filled equilateral triangle (pointed downwards) and then call itself recursively 3 times (with an appropriate stopping condition). When writing your program, exercise modular design: include a (non-recursive) function `filled_triangle()` that draws a filled equilateral triangle of a specified size at a specified location.
Your program must be organized according to the following API:

```c
// draw shaded equilateral triangle with bottom vertex
// at (x, y), side length s
void filled_triangle(double x, double y, double s)

// draw one triangle, bottom vertex (x, y), side s;
// recursively call itself three times to generate the next
// order Sierpinski triangles above, left and right of
// current triangle. n is the order
void sierpinski(int n, double x, double y, double s)

// read order of recursion N as a command-line argument;
// draw a outline triangle with endpoints (0, 0), (1, 0), and
// (1/2, \sqrt{3}/2); generate an order-N Sierpinski triangle inside
// the outline. the outline should be a color other than black
void main(int argc, char* argv[])
```

Your program will take a command-line argument N, the depth of the recursion. The drawing should fit snugly inside the equilateral triangle with endpoints (0, 0), (1, 0), and (1/2, \sqrt{3}/2). **Do NOT change the range of the drawing window.** Here are some sample runs showing what your program should display:

```
./sierpinski 1
./sierpinski 2
./sierpinski 3
./sierpinski 4
./sierpinski 5
./sierpinski 6
```
This week, your code is a single file dependent on just the draw library, so we use that library instead of “starter files”:

```
mkdir lab-fractal # or whatever folder you like
cd lab-fractal
wget ftp://bits.usc.edu/cs103/draw/draw.zip
unzip draw.zip
```

Call your file `sierpinski.cpp` and compile it with `make sierpinski`

3 Designing your Recursion (Optional Walkthrough)

Here is the approach we suggest for creating this program. These are purely suggestions for how you might make progress. You do not have to follow these steps.

Note that your final program should not be very long (similar to `htree.cpp`, not including comments and blank lines).

1. Review the textbook and lecture notes on recursion, paying attention to the `htree.cpp` example and the `nestedcircles.cpp` example from the draw library.

2. Write a (nonrecursive) function `filled_triangle()` that takes three real-valued arguments (`x`, `y`, and `size`), and draws a filled equilateral triangle (pointed downward) with bottom vertex at `(x, y)` and side length `size`. To debug and test your function, write `main()` so that it calls `filled_triangle()` a few times, with different parameters. To actually draw the triangles, you’ll need to use the correct function from the draw library API, which is documented here: [http://bits.usc.edu/cs103/draw/](http://bits.usc.edu/cs103/draw/)

3. Write a recursive function `sierpinski()` that takes 4 arguments (`n`, `x`, `y`, and `size`) and plots a Sierpinski triangle of order `n`, whose largest triangle has side length `size` and bottom vertex `(x, y)`. To do this...

   a. Write a recursive function `sierpinski()` that takes one argument `n`, prints the value `n`, and then calls itself three times with the value `n-1`. The recursion should stop when `n` becomes 0. To test this function out, write `main()` so that it reads one integer `N` from the command line and calls `sierpinski(N)`. Excepting the spacing, when you call `sierpinski()` with `N` ranging from 0 to 5, you should get the output shown at [http://bits.usc.edu/files/cs103/sierpinski/a.txt](http://bits.usc.edu/files/cs103/sierpinski/a.txt) Make sure you understand how this function works, and why it prints the numbers in the order it does.
b. Modify `sierpinski()` so that in addition to printing \( n \), it also prints the size of the triangle to be plotted. Your function should now take two arguments: \( n \) and size. The initial call from `main()` should be to `sierpinski(N, 0.5)` since the largest triangle has side length 0.5. Each successive level of recursion halves the length. The output should match [http://bits.usc.edu/files/cs103/sierpinski/b.txt](http://bits.usc.edu/files/cs103/sierpinski/b.txt).

c. Modify `sierpinski()` so that it takes 4 arguments (\( n, x, y, \) and size) and plots a Sierpinski triangle of order \( n \), whose largest triangle has side length size and bottom vertex \((x, y)\). Start by drawing Sierpinski triangles with pencil and paper. Use the pictures on the previous page to help you. Focus on getting the outline and the first triangle correct (`./sierpinski 1`) which has nothing to do with recursion, then get the next three triangles working (`./sierpinski 2`). Then try the other values of \( N \).

d. Remove all print statements before submitting.

### 4 Geometry and Style
To help get started on the geometry, here is what the first 4 triangles and the outline should look like:
Notice that the top vertex has y-coordinate $\sqrt{3}/2$. This is a consequence of the Pythagorean theorem (and the fact that the altitude bisects the opposite side). Actually, you will have quite a few places in your code using $\sqrt{3}/2$.

Stylistically, you should try to help the reader realize this “magic number” comes up again and again. The general standard is to define a named constant. For instance, in a program about pentagons, we might define

```cpp
const double GOLDEN_RATIO = (1+sqrt(5))/2;
```

We would define this once at the top of your program outside of all the functions, and then use that variable throughout to guide the reader. Follow this approach for this assignment. There is more than one way to do it. What is essential is to avoid that the reader sees 1.73205080757 or 0.86602540378 all over your program, since it’s impossible to know what that really means.

Note: defining variables outside of all the functions creates a “global” variable. It is generally bad style to create a global variable because it adds hidden complexity to your functions; any one can affect any other function through that variable. But, a global constant is fine because such side-effects are not possible. The `const` makes the compiler enforce that this “variable” can never change.

5 Fractal Dimension (Optional Reading)

In grade school, we learn that the dimension of a line segment is one, the dimension of a square is two, and the dimension of a cube is three. But you probably didn’t learn what is really meant by dimension. How can we express what it means mathematically or computationally? Formally, we can define the **Hausdorff dimension** or **similarity dimension** of a self-similar figure by partitioning the figure into a number of self-similar pieces of smaller size. We define the dimension to be the $\log$ (# self similar pieces) / $\log$ (scaling factor in each spatial direction). For example, we can decompose the unit square into 4 smaller squares, each of side length 1/2; or we can decompose it into 25 squares, each of side length 1/5. Here, the number of self-similar pieces is 4 (or 25) and the scaling factor is 2 (or 5). Thus, the dimension of a square is 2 since $\log (4) / \log(2) = \log (25) / \log (5) = 2$. We can decompose the unit cube into 8 cubes, each of side length 1/2; or we can decompose it into 125 cubes, each of side length 1/5. Therefore, the dimension of a cube is $\log(8) / \log (2) = \log(125) / \log(5) = 3$.

We can also apply this definition directly to the (set of white points in) Sierpinski triangle. We can decompose the unit Sierpinski triangle into 3 Sierpinski triangles, each of side length 1/2. Thus, the dimension of a Sierpinski triangle is $\log (3) / \log (2) = 1.585$. Its dimension is fractional—more than a line segment, but less than a square! With Euclidean geometry, the dimension is always an integer; with fractal
geometry, it can be something in between. Fractals are similar to many physical objects; for example, the coastline of Britain resembles a fractal, and its fractal dimension has been measured to be approximately 1.25; see http://en.wikipedia.org/wiki/List_of_fractals_by_Hausdorff_dimension#Random_and_natural_fractals