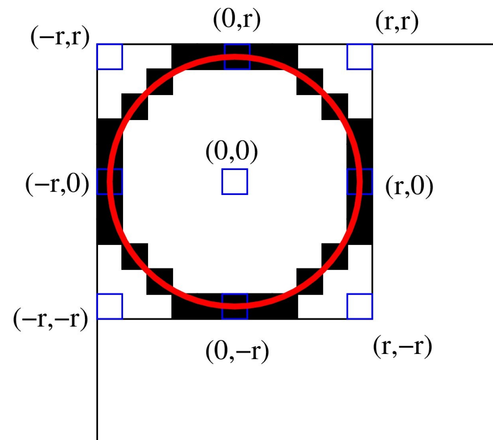
CS19001/CS19002 PROGRAMMING AND DATA STRUCTURES LABORATORY

Assignment No: 7

Last Date of Submission: 23–March–2015

A computer (or TV) screen consists of a two-dimensional array of dots, called *pixels*. For example, a 16:9 widescreen HDTV screen has a 1920 × 1080 array of pixels. A color screen actually consists of three such pixel arrays corresponding to the fundamental color components Red, Green and Blue (RGB). In this assignment, you deal with two-color (black and white) images only. Moreover, you simulate the screen in the text mode so that you can print the screen output to your terminal.

Declare a two-dimensional array (say, of size MAXSIZE × MAXSIZE) to stand for the screen. Your task is to draw a circle of radius r at the top left corner. The points on the circle do not in general have integer coordinates. Here, you deal with pixels which have integer coordinates. Therefore, you need to approximate the circle to a set of integer-valued points on the screen. The adjacent figure demonstrates this idea. The red circle is the *actual* circle. When projected to the pixel array, we have only the blackened pixels that approximately stand for the circle. If you count the array indices from the top left corner, the center of the circle is at the array element at index (r, r) (the blue-bordered pixel marked (0,0)). The entire circle fits inside the $(2r + 1) \times (2r + 1)$ segment of your screen array. The equation of the circle is $x^2 + y^2 = r^2$. This is symmetric about the x- and y-coordinates. You can draw thicker circles too using a strategy described in Part 2 below.



Part 1

Initialize the top left $(2r + 1) \times (2r + 1)$ part of your two-dimensional array as unmarked. As you discover pixels on the circle, mark those pixels. After all the pixels on the circle are marked, print the $(2r + 1) \times (2r + 1)$ part of your array.

Here follows a strategy how you can identify the black pixels. Run h from 0 to r. For each h, get the rounded value k of $\sqrt{(r^2 - h^2)}$. Mark the eight pixels $(\pm h, \pm k)$ and $(\pm k, \pm h)$ relative to the center of the circle.

Part 2

In this part, you draw a circle of thickness t pixels, where $1 \leq t \leq r$. Use the same strategy as described in Part 1. Instead of drawing only one circle with radius r, now draw t circles of radii $r, r - 1, r - 2, \ldots, r - t + 1$. Print the thick circle.

For the first two parts, write a *single* function which marks a circle of radius ρ at the center (0,0) of the circle of radius r. In part 1, call the function with $\rho = r$ only. In Part 2, make additional $t - 1$ calls with $\rho = r, r - 1, r - 2, \ldots, r - t + 1$.

Part 3

Your printing in Part 2 often reveals that there are *holes* in the thick circle. We call a pixel a hole if it is unmarked and all its four adjacent pixels are marked. Identify all the holes, and mark them. Print the refined thick circle again. Write a function for this part.

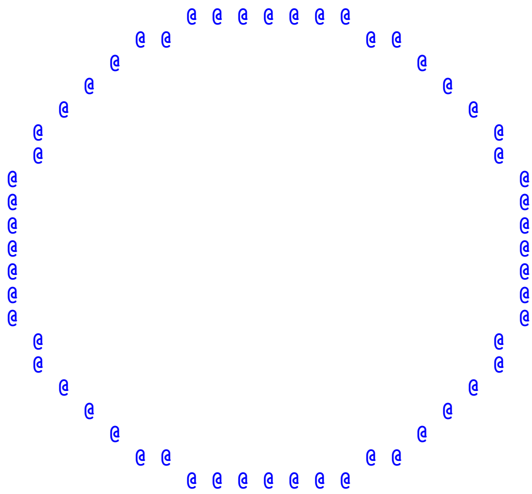
Submit a single C source file implementing all of the three parts.

Sample Output

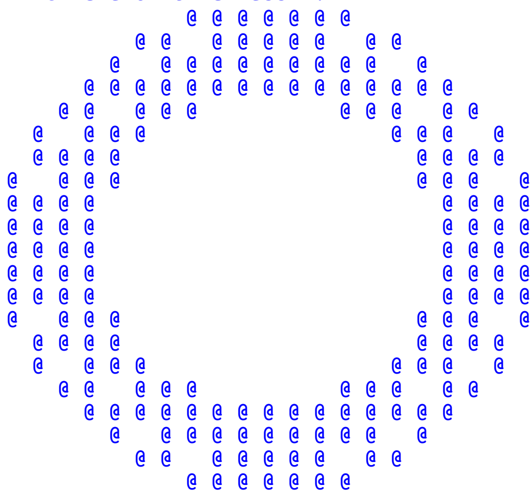
The output on the next page shows a marked cell by the character `@` and an unmarked cell by a space. If you print the $(2r + 1) \times (2r + 1)$ array as such, the output will not *look* like a circle, because characters typically have larger height than width (moreover, there may be some spacing between two consecutive lines). For a better-looking output, a space is printed after printing each character of the $(2r + 1) \times (2r + 1)$ subarray. The output of Part 2 has 20 holes that are filled in Part 3. The picture above was drawn by eye estimation, and may fail to match an actual program output.

```
Enter radius (r): 10
Enter thickness (t): 4
```

+++ Circle of thickness 1:



+++ Circle of thickness 4:



+++ Circle of thickness 4 after refinement: