EEN118 LAB THREE

The purpose of this lab is to ensure that you are confident with - and have had a lot of practice at - writing small clear functions that give precise control over repetitive tasks. The lab is divided into three almost independent sections. Be careful that you don't ruin the work you did for one section while you are working on the next.

Section A - Conversions

A1. *Remembering how to start*

Remind yourself of how to write the most basic repetitive function of all, one that takes two parameters, A and B, and just prints all the numbers from A to B inclusive. This will be the starting point for everything this week, so type it in, and make absolutely sure that you have got it right, and it really works.

A2. *Angles for Computers*

You could adapt that function to count differently. Make a new version of it that counts in steps of five, so that if you said "`numbers(5, 90);`" in main, your program would print `5 10 15 20 25 30 ... 85 90`. Test it, make sure you got it right.

Now imagine that you are going to be working with a computer, drawing some geometrical shapes! We know by now that people use degrees to measure angles, but computers use radians, and the conversion isn't very easy to do in your head. To convert from degrees to radians, you have to multiply by pi then divide by 180.

Adapt your function so that it doesn't just print numbers, but prints degree to radian conversions instead. Where it used to print X, it should now print X `degrees is` Y `radians`. The first few lines of output would look something like this

```
5 degrees is 0.0872665 radians
10 degrees is 0.174533 radians
15 degrees is 0.261799 radians
```

And not we want it the other way round too. Also, in the same program, produce a table that converts from radians to degrees. Let the value in radians range from 0 to 1 in steps of 0.1, so the output should start like this

```
0.1 radians is 5.72958 degrees
0.2 radians is 11.4592 degrees
0.3 radians is 17.1887 degrees
0.4 radians is 22.9183 degrees
```

The output we are getting is too accurate to be a quick visual guide, we don't want so many digits after the decimal point. When converting degree to radians, 2 digits after the point is quite enough. For radians to degrees we only want the result given to the closest whole degree. The output should be more like this:

```
5 degrees is 0.08 radians
10 degrees is 0.17 radians
15 degrees is 0.26 radians
......
```

```
......
0.1 radians is 6 degrees
0.2 radians is 11 degrees
0.3 radians is 17 degrees
0.4 radians is 23 degrees
```

Trick: When a program calculates (for example) `12.3*1.609344`, it gets a very accurate result, `19.79493`. Usually, precision is what you want, but in this case we don't want so much. To throw away all the digits after the decimal point, and reduce the value to an int, the C++ expression is

```
(int)(12.3*1.609344)
```

which is 19. Yes, you do need all those brackets, and of course it works for all numbers with decimal points in them, not just `12.3*1.609344`. It is even better if you *round* the result to the *nearest* int. The trick for that is

```
(int)(12.3*1.609344 + 0.5)
```

which works for all positive values.

The method for getting just two digits after the decimal point is based on this, but requires something extra too. Can you work it out?

Section B - ASCII Art

B1. *Stars*

Go back to your basic counting function from A1, and this time adapt and adopt it in a different way. Make a function that has one parameter N, which just prints a row of N stars. That's it, nothing complicated, `stars(7)` should just print "`*******`".

Your function in A1 had two parameters, but this time we want a function that has only one parameter.

B2. *Spaces*

Now make another function almost identical to that, but it should print spaces instead of stars. `spaces(7)` should just print seven spaces. How are you going to test it? If you just print spaces, you don't see anything.

B3. *Stars and Dots*

Now make another function that takes two parameters A and B, and prints A dots followed by B stars followed by another A dots followed by a new line. `dotsstars(3, 4)` should just print "`...****...`". Remember that having functions that use other functions to do most of their work is a good design technique.

B4. *Another adaptation*

Thinking about how you controlled repetition so far, write yet another function that takes two parameters A and B. This one should count *down* from A to 1 in steps of 2, and at the same time count *up* from B. That sounds pointlessly complicated, but one little example will make it clear: `sequence(9, 1)` should print

```
9  1
7  2
5  3
3  4
1  5
```

B5. *Combining*

Still remembering the idea of little functions using other little functions to do their jobs, write another function just like `sequence`, except that it doesn't print the numbers, it uses them as parameters to `dotsstars`. This new function will draw right angled triangles: as an example `triangle(5, 1)` should print

```
.*********.
..*******..
...*****...
....***....
.....*.....
```

Not very spectacular I admit, but it's all good practice.

B6. *Up-side Down*

Having done that, this part should be really easy. Make another function that draw the triangle pointing up instead of down. `uptriangle(5, 1)` should print

```
.....*.....
....***....
...*****...
..*******..
.*********.
```

B7. *Diamond*

Now make a function that draws a diamond. `diamond(5)` should print

```
.....*.....
....***....
...*****...
..*******..
.*********.
.*********.
..*******..
...*****...
....***....
.....*.....
```

if that doesn't seem absolutely trivial, think for another minute before starting.

Section C - Circles

C1. *A circle*

One way to draw an approximate circle is to draw a straight line a short distance, then turn a small amount to the right. Repeat that so many times that all the turns add up to 360 degrees, and you'll be back at the starting point. If the steps are small enough, nobody will be able to tell the difference between that and an exact circle. Computer monitors aren't really very sharp, so the steps don't need to be really really small, just small.

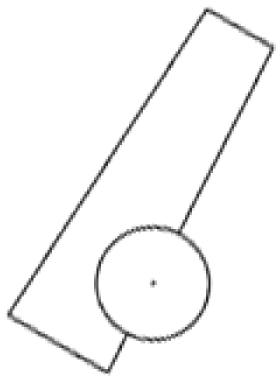
Do it. Write a function that draws a circle. You should be able to control the size of the circle by altering its parameters.

This is how you get the most accurate value for pi in C++:

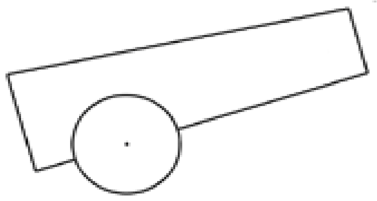
```
const double pi = acos(-1.0);
```

C2. *Weaponising*

Now your circle is going to be the wheel of a cannon.



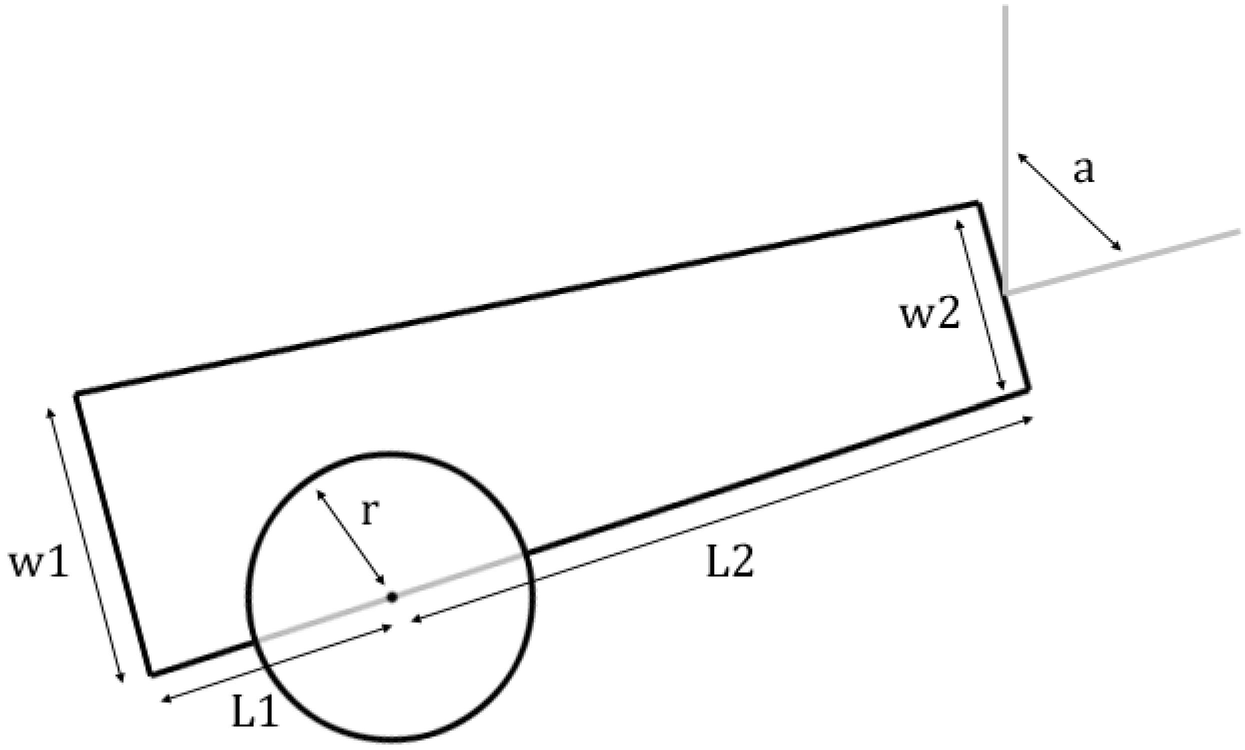
Given the position of the bottom of the wheel and the aiming angle (x, y, a), you should be able to make a simple cannon anywhere, aiming at any angle you want, such as 30 degrees (from vertical, shown to the left) or 75 degrees (below).



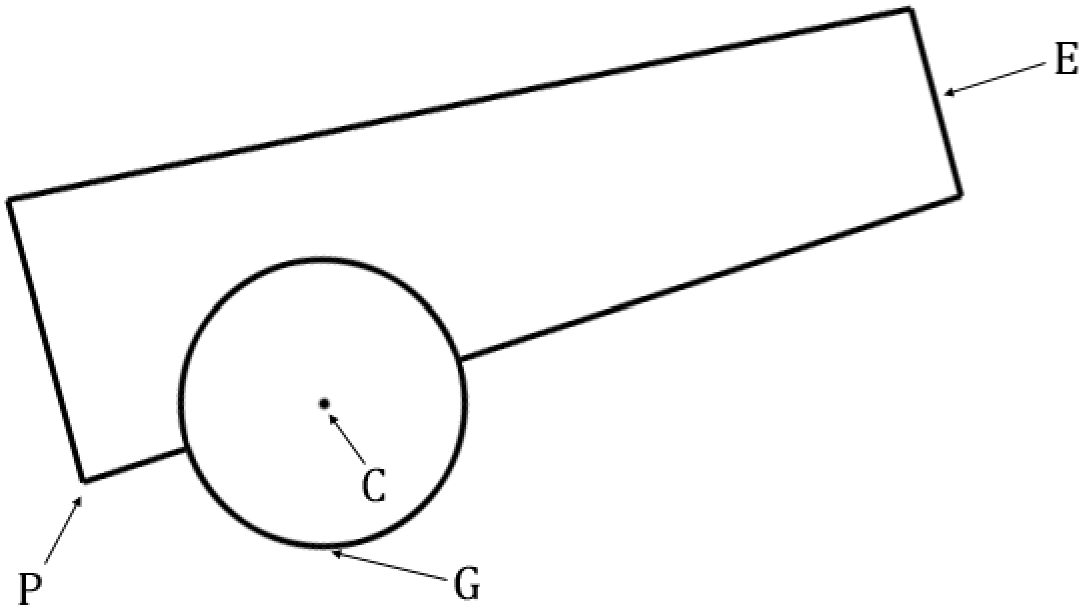
Like in the diagram, a real cannon is narrower at the front, where the cannon ball comes out, than at the back. If that seems too difficult, your cannon can be perfectly straight, just a rectangle with a wheel. But there is a small amount of extra credit available for making a good job of a properly tapered shape.

<u>That’s the END of the lab. The rest is just if you need a bit of help with the shape:</u>

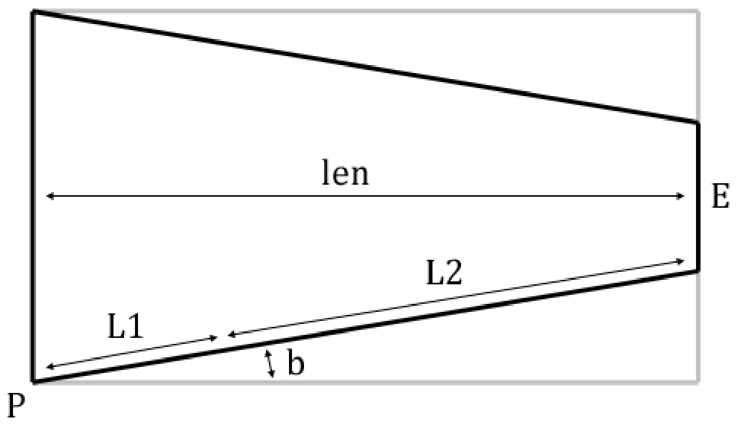
This is the cannon sitting on a wheel. The wheel’s radius is r.



a is the aiming angle.
L1 is the distance from the back of the cannon to the wheel’s axle, L2 is the rest of the length.
Like all cannons it is wider at the back (w1) than the front (w2).



G is the point on the ground where the wheel rests. Its coordinates are (xg, yg).
C is the exact position of the axle, its coordinates are (xc, yc).
P is the easiest point to start drawing the body of the cannon from, coordinates (xp, yp).
E is the point where the ball pops out when it is fired, coordinates (xe, ye).



This is a simplified picture of the body of the cannon shown with its “bounding box”. The point is to illustrate the difference between the real length of the cannon (len) and the sum L1+L2.

The angle shown as b is also helpful when drawing the shape. When the cannon is aimed at angle a, the heading for the bottom line is (a-b). Don’t forget that all angles are computed in radians.

```
xc = xg
yc = yg - r

b = asin((w1-w2)/2/(L1+L2))

xp = xc - L1 * sin(a-b)
yp = yc + L1 * cos(a-b)

len = (L1+L2) * cos(b)
```

Finally, to find the point E, we need two extra values:
- d is the distance between points P and E
- g is the angle from point P to point E if the cannon lies flat as in the third diagram.

```
d = sqrt(len*len + w1*w1/4)
g = asin(w1/2/d)

xe = xp + d * sin(a-g)
ye = yp - d * cos(a-g)
```