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 recursive 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. *Exotic Canada*

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(10, 90);`" in main, your program would print `10 15 20 25 30 ... 85 90`. Test it, make sure you got it right.

Now imagine that you are taking a trip to some wild and exotic part of the world where they use the metric system. Canada perhaps. You'll be driving, and don't want to get a ticket for going too slow, so you want a conversion table to translate kilometres per hour into miles per hour. One mile is 1.609344 km.

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

```
10 kph is 6 mph.
15 kph is 9 mph.
20 kph is 12 mph.
25 kph is 16 mph.
30 kph is 19 mph.
```

You are probably seeing some ugly output now, with lots of distracting extra digits. When a program calculates (for example) `10/1.609344`, it gets a very accurate result, `6.213712`). Usually, precision is what you want, but in this case it doesn't help. To throw away all the digits after the decimal point, and reduce the value to an int, the C++ expression is

```
(int)(10/1.609344)
```

Yes, you do need all those brackets, and of course it works for all numbers with decimal points in them, not just `10/1.609344`. It is even better if you *round* the result to the *nearest* int. The trick for that is

```
(int)(10/1.609344 + 0.5)
```

which works for all positive values.

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 - Repeating Forms

C1.

Suppose you want to use recursion when writing a function that prints the following sequence of integers. 1 2 3 4 3 2 1. This means two sequences: one ascending and one descending. In this particular example, the maximum number is 4, but your task is to write a function that produces this sequence when the maximum value is passed to the function as an integer argument, N (in the previous sequence, N=4).

C2.

Your next task is to write a little program that prints the following sequence:

```
1
1 2 1
1 2 3 2 1
1 2 1
1
```

Note that each row can be printed by the function from (C1) by calling it for a different value of N which grows from 1 to 3 and then decreases again, all the way to 1. A more general version of this program will allow the user to specify different values than MAX=3.

C3.

The final version of your program will end each row with the sum of the integers contained in this row:

```
1 sum=1
1 2 1 sum=4
1 2 3 2 1 sum=9
1 2 1 sum=4
1 sum=1
```