Computer Science 3-4
Problem Set #7: For Loops

Using loops to increment values is a common part of almost every C++ program…in fact, it is so common that we have another type of loop specifically for this purpose called the **for loop**. This type of loop is useful when we are dealing with counters, and in particular arrays (more on that in the next Problem Set). Instead of having to make a new counter for every while loop (or resetting it back to 0), we can just use a for loop which allows us to define a new counter, a condition for the loop to stop and how much we increase the counter by each time through the loop. For instance, our first while loop homework problem could be re-written using a for loop, here's what it looks like:

```
int total = 0;
int number = 0;
cout << "Please enter a number: ";
cin >> number;
for(int  i = 1; i <= number;  i++)
{
    total += i;      // keep a running sum
    if(i < number)   // test to print out the proper sign
        cout << i << " + ";
    else
        cout<<i<<" = " << total << endl;
}
```

Note that we have three parts of the "*for*" statement: we initialize our counter (which will almost always be an int), then we set a condition for the loop to terminate and the final part is how we are changing our counter (in this case we are just going up by 1's but we could do something else). Notice that the initialization of our counter variable happens just once, the first time we hit the loop, the loop goes on its merry way and then our counter variable goes "poof." In the above example, we couldn't use the "i" variable outside of the loop…the rest of the program doesn't know about it!

Just like our while loops, we can have nested *for* loops (and this is where these loops shine):

```
// What does this code remind you of?
for(int i =0; i < height -2; i++)
{
    cout << "*";
    for(int  j = 0; j< width -2; j++)
    {
        cout <<  " ";
    }
    cout << "*" << endl;
}
```

Notice how we define a different counter variable for each loop…this is important so we can use the values in the counters in our calculations. Also, we have the added benefit of not needing to reset our counter in the nested loop (the width loop)…C++ automatically does this for us!

So, this means that we are now in a situation where we need to decide when we can/need to use different types of loops. Remember, you can do *everything* with a while loop, but only *some* things with a for loop.

While Loops – Used when we are expecting input from the user to end our program, if we want to repeat something until it becomes false (or true), when we care about multiple conditions being met before stopping.

For Loops – Used when we want to just want to repeat something a certain number of times, when dealing with arrays, useful for sorting algorithms, make certain sections of code easier to read.

I would encourage you to use for loops unless you have a specific need for the while loop…it will just make your code much easier to manage (especially as you get to working on harder problems)!

Here are a couple of problems *for* you!

1. Using a for loop, write a program that sums up all the multiples of 7 up to 70,000 (note that this should take about 4 lines of code!).

2. Using for loops, write a program that will calculate and print out an N x M multiplication table (based on user input). For instance if the user enters 4 and 3 you should get:

1	2	3	4
2	4	6	8
3	6	9	12

3. **Fibonacci Numbers –** Fibonacci numbers are a famous series of numbers that are used in a wide array of computer applications. For those of you who don't remember, the Fibonacci series looks like this: 0, 1, 1, 2, 3, 5, 8, 13, 21, etc. Using for loops, write a program that can find the n^{th} Fibonacci number based on user input. Prints the entire series up to that number and don't forget to use a for loop!

4. **Interest Rate Calculator** – It is amazing how many apps for phones and the web just simply calculate the compound interest on some amount of money (be it savings, mortgages, credit card debt, etc.). For this problem, write a program that lets the user enter a starting amount of cash in their bank account, an interest rate (in percent) and calculates the value of their account after 30 years (assume interest is credited to the

account at the end of every year). Print out the account balance at the end of each year. Note: if you include <math.h> you can use the **pow**(x,y) function (pow(2,3) would give you 2^3).

5. Suppose we want to upgrade our interest rate calculator to be a bit more realistic. Modify the program as follows:
    a. Allow the user to enter an amount that they plan to save every month.
    b. Change the interest calculation to be monthly rather than annually.
    c. At the end of list of account balances, let the user know how much extra they had at the end of 30 years vs. if they hadn't done their monthly deposit.

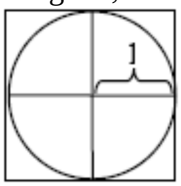
6. **Challenge –** Add two features to your interest rate calculator:
    a. Allow the user to choose how frequently their balance is compounded (yearly, monthly, weekly, etc.)
    b. Let the user enter in a monthly payment they wish to make and tell them how long it will take until they are debt-free! If their payment is too small to ever pay off their debt, let them know!

7. More Picture Patterns: Ask the user to choose whether they want to make a diamond or an "X" and how tall they would like it to be. Write some loops that will draw each shape, make sure to check your base cases!

8. **Challenge** – One more picture pattern! The user should enter a height and a width!

```
**********
*0*0*0*0*0*
**********
*0*0*0*0*0*
**********
```

9. **Pot Shots at Pi -** Let's calculate pi! We're not going to do it by measuring circumferences, though; we're going to do it with Monte Carlo methods – using random numbers to run a simulation. Let's take a circle of radius 1 around the origin, circumscribed by a square with side length 2, like so:



Imagine that this square is a dartboard, and that you are tossing many, many darts at it at random locations on the board. With enough darts, the ratio of darts in the circle to total darts thrown should be the ratio between the area of the circle and the area of the square. Since you know the area of the square, you can use this ratio to calculate the area of the circle, from which you can calculate pi using:

$$\pi = \frac{a}{r^2}$$

We can make the math a little easier. All we really need to calculate is the area of one quadrant and multiply by 4 to get the full area. This allows us to restrict the domain of our *x* and *y* coordinates to [0,1] (instead of having to deal with negative numbers).

In the past we have always come up with random integers. We can also convert the random integers to decimal values in the range [0, 1] simply by dividing by the value RAND_MAX:

```
srand(time(NULL)); // set the seed to current time
double test = rand() / (double)RAND_MAX;
cout << test << endl;
```

The (double) *keyword* creates a temporary copy of the variable in parentheses, where the copy is of the type indicated in angle brackets. This ensures that our division is done as floating-point, and not integer division. This is called *casting* RAND_MAX to a *double.*

To make this work you will have to follow a few steps:

1. Write two variable declarations representing the *x*-coordinate and *y*-coordinate of a particular dart throw. Each one should be named appropriately, and should be initialized to a random double in the range [0,1].

2. Place your variable declarations within a loop that increments a variable named totalDartsInCircle if the dart's location is within the circle's radius. You'll need to use the Euclidean distance formula:

$$d^2 = x^2 + y^2$$

3. Now use your loop to build a program that asks the user to specify the number of "dart throws" to run, and returns the decimal value of pi, using the technique outlined above. You should get pretty good results for around 5,000,000 dart throws.