Computer Science 3-4
Problem Set #5: While Loops continued

A quick note on libraries in C++: we have been using one library quite extensively: <iostream>. We also took a look at the math library; <math.h> (where we get our trig functions and sqrt()). There are a nearly infinite collection of libraries out there and more are being created every day. Folks generate libraries for a variety of reasons, but typically it is because they want some very specific functionality that isn't already out there. Some of the more popular ones are: OpenGL (for graphics…we'll use a variation called SFML later on), BOOST, Box2D to name a few.

A useful feature of one C++ library (known as stdlib.h, which is short for standard library) is that we can generate a random number when we need it. Here is some code to check out:

```
#include <iostream>
// Libraries contain lots of functions we'll use
// C++ says "we assume you don't want to use it"
// So include it if you want it!
#include <stdlib.h> // this contains the rand function
#include <time.h>   // we want to use some of the time features

using namespace std;

int main()
{
srand(time(NULL));  // This sets the random number generator up

int myRandomNumber1 = rand();
// This gives a random number from 0 to 32767 (weird I know)
int myRandomNumber2 = rand()%1000;
// This gives a random number from 0 to 999
int myRandomNumber3 = rand()%1000 + 1;
// Generates a new random number from 1 to 1000
cout << myRandomNumber1 << endl;
cout << myRandomNumber2 << endl;
cout << myRandomNumber3 << endl;
}
```

The function **rand()** actually returns a random integer from a very large range; the max is around 32767 or so. We'll use the modulus operator (%) to dictate the range that we want. When we calculate rand() % 1000, we actually return values from 0 to 999 (since it either divides evenly or has up to 999 as a remainder). By adding a number to the result we can shift the range (we just added one in the above example, but you can shift it up however you like). This is very useful for us!

Note the srand() function. This is a function that sets the "seed" for the random number generator. For instance, if you wrote a program to randomly generate 10 numbers and I did the same thing…if we started our program off with srand(3), then we'd actually generate the same 10 numbers. There are many reasons that we would like to do this (mostly related to testing for potential problems, but it has science applications as well). When we do srand(time(NULL)), we are really saying that we want our random numbers to be generated based on the current time (down to the millisecond), this way it will be different every time we run the program. We only need the srand() function to run one time for the seed to be set, so we usually put it at the very top of our main() function if we are using random numbers. On to the problems!

1) **Your First Game!** – Write a program to play a guess a number game. The computer should be thinking of a number between 1 and 1000 and the user has to try to guess what that number is by typing in their own number. If the user is incorrect, the computer should give them a hint ("Your guess was too low" or "your guess was too high") and let them guess again. Keep track of how many guesses the player makes and print out a message at the end that congratulates them and prints the total number of guesses it took.

2) **Challenge** – Have the computer tell the player how many digits are correct in their guess rather than if they are too high or too low (use the code from PS#2 Challenge as a start).

Often we'll have to set up a nested set of loops to solve a problem. This can be a bit tricky, mainly because we need to keep track of a variety of variables (and reset them properly). Let's take a look at some code:

```
int main()
{
    // we want to print a 10 x 10 box
    // the outer loop controls our current row
    // the inner loop controls all our printing
    int height = 10;
    int width = 10;

    while(height > 0)  // start of our "outer" loop
    {
        while(width > 0)
        {
            width--;  // our width counts down
            cout << "* ";
        }
        cout << endl; // we've prited a row, so end the line
        height--;   // we've finished one pass through the outer loop
        width = 10; // we'll use the width again...reset it!
    }
}
```

Use this basic structure to solve the following problems:

3) Print the following pattern, using horizontal tabs to separate numbers in the same line. Let the user decide how many lines to print (i.e. what number to start at):

```
4
3	3
2	2	2
1	1	1	1
```

4) **Draw a Rectangle!** – Write a program that lets the user enter the width and height of a rectangle, then draws the rectangle to the screen using *. For instance, if the user said 4 wide and 3 tall, the program would print out:

```
* * * *
*     *
* * * *
```

Hint - Try these test cases out before you turn in your program: 3x3, 5x4, 1x3 and 5x1.

5) **Fab Factorials** – Write a program that prints out a table of factorials. For instance, if the user types in 5, then the output should be:

1! = 1 = 1
2! = 2 x 1 = 2
3! = 3 x 2 x 1 = 6
4! = 4 x 3 x 2 x 1 = 24
5! = 5 x 4 x 3 x 2 x 1 = 120

Sometimes we may want a loop to end under multiple conditions; in this case it is often useful to use the **break** command. For example:

```
while (product <=10000)
{
product = 3 * product;

    if (product == 81)
    {
        break;   // this will exit the loop without going up to 10000
    }
}
```

6) **Challenge: The 1000th Prime! –** Write a program that computes and prints the 1000th prime number. Here are a few hints to get you started:

a. You will need some variables to keep track of which prime number you're on and keep track of where you are in your loop.
b. You'll only need to test odd numbers to see if they are prime.
    i. Remember that something *isn't* a prime number if another integer can divide it…use your % operator.
    ii. Think about how many numbers you actually need to check….you can stop sooner than you think!
c. You'll need two loops
d. If you want to check that your code is correctly finding primes, you can find a list of primes at http://primes.utm.edu/lists/small/1000.txt