EE 1301
Lab 3: Iteration

Imperative programming languages such as C++ provide high-level constructs that support both ***conditional selection*** **and** ***iteration***. ***Conditional iteration*** or *repetition* refers to the ability to repeatedly execute instructions, or groups of instructions, until some dynamically tested criterion is met. For example, in the statement "keep counting down from ten until you reach zero", we initialize a counter to ten and repeatedly subtract one from the counter in each iteration. In each iteration, we dynamically test whether the counter equals zero and stop iterating if it does. *Iteration* is what gives the modern electronic digital computer much of its extraordinary power. Using computational methods such as *exhaustive enumeration* (i.e., trying every possible solution to a problem), we can search through a large number of possible problem solutions in a matter of seconds to determine the correct or best one.

C++ provides 3 basic *repetition* mechanisms: ***while-loops***, ***for-loops*** **and** ***do-while loops***, but the latter two are just simple variants of the first. Learning to solve problems using repetition may be challenging at first, but soon becomes second nature. It is important to remember that, although you construct the loop as a *static* structure, it will be executed *dynamically*. **Static** means that the code statements or instructions only appear in your code once, and dynamic means that the statements or instructions can be repeatedly executed many times when the program runs. You need to develop the ability to envision how the steps of the loop will unfold as it executes.

This lab is longer than the previous labs, as we are now at the point in the class where we know enough C++ to have longer, more complicated (but also more interesting) lab problems.

Mystery-Box Challenge!

We are introducing a new feature to the Lab Exercises: the "Mystery-Box Challenge". It is designed to help you develop a better understanding of how programs work. The Mystery-Box Challenge presents a short code fragment. Your job is to identify the high-level purpose of the code – what the computation is intended to accomplish.

For example, the purpose of the following line of code:

```
3.14159 * radius * radius
```

is to *"compute the area of a circle whose radius is contained in a variable named* `radius`*"*. An answer such as "*multiply 3.14159 by radius and then multiply it by radius again*", although technically an accurate description of the code, does not capture the high-level meaning of the code.

Here is your first *mystery-box challenge*: Determine (and describe to your Lab TA) what the following C++ code fragment is intended to do.

```
int sum = 0;
for(int i=0; i<=100; i++)
{
    if(i % 7 == 0)
        sum++;
}
cout << sum;
```

Warm-up

1) **Fibonacci Numbers**

Fibonacci numbers pop up all over the place and are related to the golden ratio. You can find the next Fibonacci number by adding together the previous two. The first two Fibonacci numbers are $F_0 = 0$, $F_1 = 1$. Thus:

$F_2 = F_1 + F_0 = 1 + 0 = 1$,

$F_3 = F_2 + F_1 = 1 + 1 = 2$,

$F_4 = F_3 + F_2 = 2 + 1 = 3$,

...

Write a program that asks how many Fibonacci numbers it should compute and then shows each computed Fibonacci number.

2) **Loop Equivalence**

Using paper and pencil, rewrite the following `while` loop as an equivalent `for` loop.

```
int i = 1;
int total = 0;
while( i < 100 )
{
    total += i;
    i++;
}
```

3) **Nested Loops**

Write a short C++ program that will use *nested* `for` loops to print out all the values for a 10 x 10 multiplication table, as follows.

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	58	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Each entry in the table must be *calculated* as the product of two variables and output to the terminal from within the loop (do NOT simply write out character strings). Format your output so the columns line up as shown.

Stretch

1) **Square Root, Babylonian Style**

The ancient Babylonians produced a remarkably efficient process to compute the square root of any number *n*. The method is referred to as a "*guess and check*" algorithm because it involves *guessing* the answer and then *checking* to see if the guess produces the *correct* answer. If the guess is not correct, an algorithm is followed to systematically improve the guess until it's close enough to the correct answer. The simple algorithm that the Babylonians used is to make the next (improved) guess equal to the *average* of the old guess and the ratio of *n* to the old guess.

new_guess = (old_guess + (*n / old_guess*)) / 2

As you can see, whenever the guess is correct, the new guess will equal the old guess after applying the formula above. This is the convergence condition, i.e., the way you tell that a solution has been found. This algorithm *converges* quickly, which means that if we begin with any arbitrary guess and update it a sufficient number of times, we will obtain a very good approximation of the square root of the value. In fact, this method converges at a quadratic rate and will produce excellent square root approximations in a small number of iterations.

The algorithm consists of three simple steps:

1. Make an initial guess at the answer (e.g., start with 1 or *n*/2)
2. Revise the guess as follows:

```
new_guess = ( old_guess + n / old_guess ) / 2
```

3. Determine if the guess is sufficiently close to the square root of *n*. In this problem we will say the guess is sufficiently close if the new guess is within 1% of the old guess. If it is not, repeat Step 2 until that criterion is satisfied. The more that Step 2 is repeated, the closer the guess will come to the square root of *n*.

Write a program that asks the user to input a positive integer for *n,* and then uses a *while-loop* to iterate through the Babylonian algorithm until the stopping criterion in Step 3 is satisfied. Output the result as a floating-point value.

Your program must do the following:

1. Prompt the user to input a positive integer value.
2. Validate the input (i.e., check that it is non-negative). Notify the user and terminate the program if an invalid value is entered.
3. For each iteration of the algorithm loop, output the value of the guess. (This is a useful debugging technique and will help you understand what your code is doing.)
4. Output the square root of the value as determined by the Babylonian algorithm.
5. Output the square root of the value as determined by the cmath function `sqrt()`.

Demonstrate your program to the TA using (at least) the following inputs.
2
25
42

Workout

1) Crime Scene Investigation

Newton's Law of Heating and Cooling states that the rate of change of the temperature function $T(t)$ with respect to time t of an object is proportional to the difference between the object's temperature and the temperature of its surroundings, T_s. Here, assume that T_s is a constant. We can write this as $\Delta T/\Delta t = k\,(T_s - T)$ where ΔT is the change in temperature over the time step Δt and k is the growth rate (heating) or decay rate (cooling) per unit time. We can now simulate the heating or cooling of an object using the following equation.

$$T(t) = T(t - \Delta t) + k\,[T_s - T(t - \Delta t)]\,(\Delta t), \text{ or}$$
$$new\_temperature = old\_temperature + change\_in\_temperature,$$

In the equation above, $T(t)$ is the new temperature at time t and $T(t-\Delta t)$ is the temperature at time $t - \Delta t$.

For example, suppose cold water at 6 degrees Celsius is placed in a room that has temperature of 25 degrees Celsius. If the time step Δt is 0.1 hour and k is 1.335, then the temperature after the first time step is:

$T(0.1) = 6 + 1.335\,(25 - 6)(0.1) = 8.5365$ degrees Celsius.

Write a program to simulate the heating or cooling of an object over a period of time (the simulation length). Include the following:

- Prompt the user to input values of the initial temperature of the object, the temperature of the surroundings, the growth (decay) rate, the length of the time step, and the simulation length.
- Using a *while* loop, do the following:
  - In each iteration of the loop, calculate the time t and the new temperature $T(t)$.
  - In each iteration, display t and $T(t)$.
  - Continue the loop as long as time t is less than the simulation length.

Example:

```
Enter the initial temperature: 6
Enter the temperature of the surroundings: 25
Enter the growth (decay) rate: 1.335
Enter the time step in hours: .1
Enter the simulation length in hours: 2
0.100000  8.536500
0.200000 10.734377
0.300000 12.638838
0.400000 14.289053
0.500000 15.718964
0.600000 16.957983
0.700000 18.031592
0.800000 18.961874
0.900000 19.767964
1.000000 20.466441
1.100000 21.071671
1.200000 21.596103
```

```
1.300000 22.050523
1.400000 22.444278
1.500000 22.785467
1.600000 23.081107
1.700000 23.337280
1.800000 23.559253
1.900000 23.751592
2.000000 23.918255
```

Suppose your pet hamster, Huey, whose body temperature was originally 37^{O} C is found dead (presumably murdered!) in a room that has a constant temperature of 25^{O} C. The body was discovered at 10:00pm with a temperature of 28^{O} C. Use your program to determine at what time Huey met his fateful demise. Assume k = 0.4055 and use a time step of 0.1 hours.

Debugging practice: Re-compile your program with debugging flags and run in gdb. Put a breakpoint on the line that prints out the values of t and T(t). Each time the debugger stops at the breakpoint, use the print command to inspect the values of t and T(t), then confirm that you have printed the correct values when they are printed by your code. Do this for a few iterations to get some practice with gdb.

2) **Decimal to Roman Numeral Conversion**

The ancient Roman numbering system employed a sequence of letters to represent numeric values. The value of individual Roman numerals is provided in the following list.

I = 1
V = 5
X = 10
L = 50
C = 100
D = 500
M = 1000

In Roman numbers, larger numeral values generally appear before smaller values, and no single numeral may be repeated more than 3 times in a row. There are a few numbers that cannot be represented with these restrictions, so an additional rule was provided to deal with these cases: if a higher value numeral is preceded by a *single* smaller numeral, the smaller value is subtracted from the larger one following it. This rule is only applied to a small number of cases: IV (4), IX (9), XL (40), XC (90), CD (400) and CM (900). Using these simple rules, we can construct a method to convert an integer (decimal) value to a Roman number, as follows.

Step 1:

If the value of the number is in [900,999] output “CM”
If the value is in [500,899] output “D”
If the value is in [400,499] output “CD”
If the value is in [100,399] output “C”
If the value is in [90,99] output “XC”
If the value is in [50,89] output “L”
If the value is in [40,49] output “XL”
If the value is in [10,39] output “X”
If the value is 9, output “IX”
If the value is in [5,8] output “V”
If the value is 4, output “IV”
If the value is in [1,3], output “I”

Step 2:

Subtract the value of the 1 or 2 letter sequence that was just outputted from the integer number value, producing a "remainder". For example, if the original number was 93, the program would have printed "XC" in Step 1, and then, in this step, it would subtract 90 (the value of XC) to get a remainder of 3.

Step 3:

Repeat the process, using the remainder as the number, until the remainder value is zero.

Using this method, write a program that does the following:

- Reads in an integer value from 1 to 999, inclusive.
- Checks to make sure the input value is > 0 and <1000. If not, output an error message and exit the program without doing anything.
- Converts the input value to a Roman number and outputs it as a string of consecutive letters without any spaces (i.e., XCIII)

Example:

```
Enter an integer value from 1 to 999: 0
Invalid input.  Program terminated.

Enter an integer value from 1 to 999: 42
Roman numeral equivalent: XLII

Enter an integer value from 1 to 999: 3
Roman numeral equivalent: III

Enter an integer value from 1 to 999: 9
Roman numeral equivalent: IX

Enter an integer value from 1 to 999: 999
Roman numeral equivalent: CMXCIX
```

Check

Write down (i) one important thing you learned from today's lab and (ii) one question you still have.
When you have done this, discuss what you have written with a neighbor or the TA.

Have a TA check your work before going on to the challenge questions.

Challenge

Here are two extra challenge problems. Try these problems if you have extra time or would like additional practice. Try not only to solve these problems but also to solve them as ***efficiently*** as possible, i.e., with as few computations as possible.

1) **Reverse the Digits**

Write a C++ program that will read in any positive integer value (*not* string) and print the *digits* out in reverse order.

Examples:

```
Input a positive integer:  1234
Reversed: 4321

Input a positive integer:  1
Reversed: 1

Input a positive integer: 9035768
Reversed: 8675309
```

2) **Factors of an Integer**

Write a C++ program that will read in any positive integer value and display it as a product of its *smallest* integer factors in increasing order.

Examples:

```
Input a positive integer:  8
Factors: 2*2*2

Input a positive integer:  25
Factors: 5*5

Input a positive integer:  80
Factors: 2*2*2*2*5

Input a positive integer:  17
Factors: 17
```