CSci 1113, Spring 2019
Lab Exercise 3 (Week 4): Repeat, Again and Again

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 statements, or groups of statements, until some dynamically tested criteria is met. *Iteration* is what gives the modern electronic digital computer much of its extraordinary power. Using computational methods such as *exhaustive enumeration*, 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 is often challenging at first. It is important to remember that, although you construct the loop as a *static* structure, it will be executed *dynamically*. 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. Usually we will want you to get through all the way through the warm-up, stretch, and workout problems to get credit for the lab. However, if – due to the length of this lab -- you do not get through all the workout problems, then do as much of them as you can.)

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 that you must interpret to determine what the computation it is *intending* to accomplish at a "high" level. For example, the following line of code,

```
3.14159 * radius * radius
```

is intended 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 accurate, would not be considered a correct answer!

Here is your first *mystery-box challenge*: determine (and describe to one of your Lab TAs) 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

(Warm-up 1) Fibonacci Numbers

Fibonacci numbers arise in a number of different places and are related to the golden ratio. Fibonacci numbers are defined by $f_0 = 0$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$ for $i > 1$. So, for example, $f_2 = f_1 + f_0 = 1 + 0 = 1$, and $f_3 = f_2 + f_1 = 1 + 1 = 2$, and $f_4 = f_3 + f_2 = 2 + 1 = 3$.
Write a program that asks a user to to input an index i, and then computes and prints out the Fibonacci number f_i. You can assume the input is a nonnegative integer; you do not need to validate it. [Hint: Although this problem is presented using a series of values $f_0, f_1, f_2, \ldots$, when computing the next Fibonacci number, all you need to keep track of is the previous two Fibonacci numbers.]

When you get this problem done, go on to the next problem.

(Warm-up 2) Loop Equivalence

Using geany, or using paper and pencil, rewrite the following `while` loop as an equivalent `for` loop. You and your partner should first do this individually. When both of you are done, compare answers, and come up with a common answer that both of you agree on.

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

When you get this problem done, go on to the next problem.

(Warm-up 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 from within the loop (do NOT simply write out character strings!). Format your output so the columns line up as shown.

When you have completed all three of the warm-up problems, have a TA check your work.

Stretch

(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. The "trick" is how to (systematically) improve the guess if it isn't close enough, in order to make a better guess the next time. The simple intuition that the Babylonians discovered 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

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 quadratically 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 (start with 1 or *n*/2)
2. Revise the guess:

   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 for a sufficient number of iterations until that criterion is satisfied. The more that Step 2 is repeated, the closer the guess will become to the square root of *n*.

Write a program that has the user input a positive integer for *n,* and then uses a *while-loop* to iterate through the Babylonian algorithm until the stopping criteria in Step 3 is met. Output the result as a real number (i.e., use doubles to store the new and old guesses).

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 nonnegative). 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. For the purpose of comparison, output the square root of the value as determined by the `cmath` function `sqrt( )`.

Demonstrate your program to a TA using the following input (at a minimum):

2
25
42

Workout

(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 its surrounding's temperature, T_s. Here assume 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 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

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

where $T(t)$ is the new temperature at time t and $T(t\text{-}\Delta t)$ is the temperature at time t - Δt .

For example, suppose cold water at 6 degrees Celsius is placed in a room that has temperature 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

$$T(0.1) = 6 + 1.335\,(25 – 6)(0.1) = 8.5365 \text{ degrees Celsius.}$$

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

1. Prompt the user, and have him or her input values of the initial temperature, the temperature of the surroundings, the growth (decay) rate, the length of the time step, and the simulation length.
2. Using a *while* loop do the following:

   a) In each iteration of the loop, calculate the time t and the new temperature $T(t)$.

   b) In each iteration, display t and $T(t)$. Use the formatting shown in the example below.

   c) Repeat the loop again if the 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 someone whose body temperature was originally 37^{O} C is 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 the murder was committed. Assume $k = 0.4055$ and use a time step of 0.1 hours. Show your answer to a TA when you are confident it is correct.

(Workout 2) Decimal to Roman Numeral Conversion
The ancient Roman numbering system employed a sequence of letters to represent numeric values. Values were obtained by summing individual letter values in a left-to-right fashion. 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. 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 output 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, subtracted 90 to get a remainder of 3, and then continued with 3
    as the number.
```

Step 3:

Repeat the process 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 output it as a string of consecutive letters without any spaces (ie., 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
```

Collaborative Check

You and your partner should individually list (i) one important thing you learned from today's lab, and (ii) one question you still have. When you have done this, share with your partner what you have written.

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

Challenge

Here are two extra challenge problems. They are not obligatory – you are welcome to log out and be done with this lab if you have completed all the previous problems and gotten them checked – but are recommended: try these problems if you have extra time or would like additional practice outside the lab. Pay particularly attention not only to solving these problems, but also to solving them *efficiently*, i.e., with as few calculations as possible.

(Challenge 1) Reverse the Digits

Write a C++ program that will read in any positive integer value (input and stored as an int, *not* as a 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: 624578
Reversed: 875426
```

(Challenge 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
```