CSci 1113, Spring 2019
Lab Exercise 4 (Week 5): Write Your Own Functions

User Defined Functions

In previous labs, you've encountered useful *functions*, such as `sqrt()` and `pow()`, that were created by other people and reside in a library. A *function* is simply a self-contained *computational process* to which we've assigned a name (label). This allows us to "invoke" it at any time during the execution of our program by providing the name in a *function call*.

Recall a *computational process* is a series of steps that direct a (mindless) machine to manipulate data and produce a desired *result*. We generally think of a *computational process* as a mapping from a set of input data to some set of output data:

input → *process* → *output*

Certain computational processes perform tasks that we want to repeat frequently, such as "*compute the square root of some number*", or "*raise some number to some power*". These are, in their own right, computational processes that take input (a number or numbers) and produce output (square root of the value, power).

By simply *naming* a process, we create a powerful and useful *abstraction.* For example, we might assign the name `sqrt` to the process "*compute the square root of some number*". Now, instead of re-typing all the steps that define this process each time we need it, we simply *invoke* the process by providing it's *name*: `sqrt`. This is the idea of *procedural abstraction*. A *procedural abstraction* is simply a means of giving some detailed process a name and then subsequently forgetting about all the details (which we really didn't care about anyway!) When we use abstractions in our program, the computer takes care of invoking the necessary steps for us, freeing us to think and develop programs at a much higher level. A huge advantage of using procedural abstractions is that we can leverage a large body of programming solutions that other people have already created. We simply get to *use* them.

A *function call* consists of giving the name of the function followed by parentheses. Some (but not all) functions such as `sqrt` require input values, or *arguments*. For example, `sqrt` requires the *value* for which it is supposed to compute the square root. When a function requires input values, the argument (or arguments) are provided inside the parentheses and separated by commas.

Most of our programming from now on will involve writing our own C++ *functions* and using them in our programs. In this lab we begin our exploration of *procedural abstraction*.

Mystery-Box Challenge
Here is this week's mystery-box challenge. Determine the output of the following C++ code fragment when the function `f1` is called, and then describe them to a Lab TAs :

```
void f1( void ){
    int x = 5;
    f2(x);
    cout << x << endl;
}

void f2( int x ){
    x += 5;
    cout << x << endl;
}
```

Reminders

A debugging tip: you can print out intermediate values in your program by inserting one or more print (i.e., `cout`) statements at key points in your program. For example, suppose you have a loop that is supposed to be computing a sum, but after the loop your program is always reporting an incorrect sum. You can insert a print statement within the loop to print out the sum each time through the loop. This will often give you insight into what isn't working correctly. Of course, remember that once you have corrected all the errors in your program you should remove all these extra print statements

Also, if you like using the terminal window to construct, compile, or run your program, here are reminders about three time-saving tips:

1. *Tab autocompletion*: Remember that, in Linux, you can type part of a command or name and the operating system will complete as much as it can of it.
2. *Up arrow to recall past commands*: Remember that, at the Linux prompt, hitting the up-arrow key once will produce the last command, and hitting the up-arrow repeatedly will produce the commands before that, in reverse order.
3. *Editing window/compile and run window*: Remember that when you develop and test programs you can have two windows open simultaneously. In the editing window you can type in, modify, and save your program. Then in the second window you can compile, run, and test your program. This will help you efficiently repeat the create/modify – save – compile – run – test development cycle.

If you do not remember or understand any of these tips, ask your partner. If you both do not understand the tips, ask one of the TAs.

Warm-up

1) **Square Root, Revisited**

Modify the program you wrote for last week's Stretch Exercise (1) so that it has the Babylonian algorithm as a user-defined square-root function. Start by defining a function named `babylonianRoot,` that takes a single floating-point argument and returns the square root estimate using the code you completed in your earlier solution.

```
double babylonianRoot(double n);
```

You'll need to modify your previous code slightly, e.g., it should return the last guess for the square root, but should not use >> or << to get any input or write any output within in your function!

Your main program should take in as input a floating-point value, then call the `babylonianRoot` function to determine and display the square root of the input value. Include a continuation loop that will continue this process as long as the user wishes.

Before actually writing the code, you and your partner should individually make a list of what specifically needs to be changed to turn your previous code into a user-defined function. Then compare lists and make the changes.

Examples:

```
enter a value: 4
square root of 4 is 2
continue? (y/n): y

enter a value: 25
square root of 25 is 5
continue? (y/n): n
```

2) **Leap Year**

Leap years contain one additional day (February 29) in order to keep the calendar synchronized with the sun (because the earth revolves around the sun once every 365.25 days). A year is a *leap year* if it is divisible by 4, unless it is a *century* (evenly divisible by 100) that is not divisible by 400. Note that 2018 was not a leap year, 1980 was a leap year, 2000 was a leap year, 1900 was *not* a leap year.

Write a program that contains a user-defined function named `isLeapYear`, that takes a year value as an integer argument and returns `true` if the year is a leap year, `false` otherwise. Do not use the >> or << operations in your function!:

```
bool isLeapYear(int year);
```

Your main program should input a year value (integer), then call the `isLeapYear` function and display whether or not the year is a leap year. Include a continuation loop that will continue this process as long as the user wishes.

Examples:

```
enter a year value: 2019
2019 is not a leap year
continue? (y/n): y

enter a year value: 2000
2000 is a leap year
continue? (y/n): y

enter a year value: 1900
1900 is not a leap year
continue? (y/n): n
```

Once you complete the warm-up problems have a TA check your work. As usual, you can start the next problem while waiting for a TA if all the TAs are busy.

Stretch

1) **Fun with Dates**

Extend the program you created in Warm-up Exercise (2) to include a second user-defined function named `lastDay`. This function should take two integer arguments representing the month and year, and return the number of days in the month represented by the arguments. Do not use the >> or << operators in your function!:

```
int lastDay(int month, int year);
```

The month argument should be between 1 and 12. If it is not, return the value 0. Recall that February has a variable number of days depending on the year! Modify the main program to input both the month and year and output the number of days in the month. Include a continuation loop that will continue this process as long as the user wishes.

Examples:

```
enter month and year: 2 2014
days in month: 28
continue? (y/n): y

enter month and year: 2 2000
days in month: 29
continue? (y/n): y

enter month and year: 12 2014
days in month: 31
continue? (y/n): n
```

2). **What is Tomorrow's Date?**

Extend your main program from Stretch Exercise (1) to accomplish the following:

- Input the month, day, *and* year values as 3 separate integers.
- Determine and output the date for the *next* calendar day (use the `lastDay` function as needed).
- Include a continuation loop that will continue this process as long as the user wishes.

Examples:

```
enter a date as mm dd yyyy: 2 28 2014
next day is: 3 1 2014
continue? (y/n): y

enter a date as mm dd yyyy: 2 28 2000
next day is: 2 29 2000
continue? (y/n): y

enter a date as mm dd yyyy: 2 28 1900
next day is: 3 1 1900
continue? (y/n): y

enter a date as mm dd yyyy: 12 31 2014
next day is: 1 1 2015
continue? (y/n): n
```

Once you have done both Stretch problems, have a TA check your work.

Workout

1) Newton-Raphson

The problem of computing the square root of any number is a specific case of the more general *root-finding* problem. Root-finding occurs in many places in computer science, as well as in other science and engineering problems. Recall that the *roots* (or *zeros*) of any function are the parametric value(s) for which the function produces a zero result:

$$x : f(x) = 0$$

Finding the *square* root of some number *K* is equivalent to finding the principal *zero* of

$$f(x) = x^2 - K$$

We could just as easily compute the *cube* root of K in the same fashion by solving:

$$x^3 - N = 0$$

and so on. Generalizing, we can find the n^{th} root of any value K by solving:

$$x^n - K = 0$$

The Newton-Raphson algorithm is an efficient and ingenious method for finding the principal root of any continuously differentiable function. The method is ascribed to Sir Isaac Newton, although Joseph Raphson was the first to publish it in a more refined form. The method is a generalized version of the Babylonian "guess and check" approach that you explored in a previous lab.

The method generalizes the Babylonian approach by employing the *derivative* of the function. Given an old guess, x_i, the new ("better") guess, x_{i+1} is computed by:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Note that the Babylonian Algorithm is simply the specific case of Newton-Raphson for $f(x) = x^2 - K$.

For the generalized nth root problem, $f(x)$ is:

$$x^n - K$$

and the derivative, $f'(x)$ is:

$$nx^{n-1}$$

Substituting these formulas, the Newton-Raphson update rule for computing the n^{th} root of any value K is:

$$\begin{aligned} x_{i+1} &= x_i - \frac{x_i^n - K}{nx_i^{n-1}} \\ &= x_i - \frac{x_i^n}{nx_i^{n-1}} + \frac{K}{nx_i^{n-1}} \\ &= x_i - \frac{x_i}{n} + \frac{K}{nx_i^{n-1}} \\ &= \frac{1}{n}\left[(n-1)x_i + \frac{K}{x_i^{n-1}}\right] \end{aligned}$$

Write a program that includes a user-defined function named `rootN` that will calculate and return the nth root of any value using the Newton-Raphson method described above:

```
double rootN(double, int);
```

The function takes two arguments: a (type double) value > 0 and an integer root > 1. For example, to find the cube root of 42 you'd call the function as `rootN(42.0, 3)`. This function should work exactly as your

Babylonian square root, but using the more generalized update rule. **Do not use any math library functions in your solution**. [Hint: it will be very helpful if you also construct a *separate* function to compute x^n.]

Your main program should take as input positive values for *K* and *n*, then call the `rootN` function to determine and display the n^{th} root of the input value. Include a continuation loop that will continue this process as long as the user wishes.

Examples:

```
enter value and root: 9 2
the root is: 3.00002
continue? (y/n): y

enter value and root: 9 3
the root is: 2.08008
continue? (y/n): y

enter value and root: 9 4
the root is: 1.73207
continue? (y/n): n
```

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 about any of the lab material. When you have listed these, share your list with each other. Then ask a TA to check your solution to the work-out problem.

Important: do not leave lab without having a TA check your solution(s).

Challenge

Here are two challenge problems. The first is fairly simple; the second is longer and complicated, but is a nice application of some of the previous material. If you complete the warm-up, stretch and workout problems before the end of lab, then work on at least one of these challenge problems in the remaining time.

1) **Greatest Common Divisor**

The *greatest common divisor* (GCD) of two integers *a* and *b* is the largest positive integer that divides both *a* and *b*. The *Euclidean* algorithm for finding this greatest common divisor of *a* and *b* is as follows. Assume neither number is 0, and let *a* be the larger number in absolute value.

- Divide *a* by *b* to obtain the integer quotient *q* and remainder *r* such that $a = bq + r$.
- It is known that GCD(*a,b*) = GCD(*b,r*), so replace *a* with *b* and then *b* with *r*, and repeat all the steps until the remainder is 0.

Since the remainders are decreasing, eventually a 0 remainder will result. The *last nonzero remainder* is the greatest common divisor of *a* and *b*, i.e., GCD(*a,b*). For example,

1260 = 198 x 6 + 72	GCD(1260,198) = GCD(198,72)
198 = 72 x 2 + 54	GCD(198,72) = GCD(72,54)
72 = 54 x 1 + 18	GCD(72,54) = GCD(54,18)
54 = 18 x 3 + 0	GCD(54,18) = 18

So the GCD of the original two numbers, namely 1260 and 198, is 18.

Write a program that includes a user-defined function named GCD that will calculate and return the greatest common divisor of two integer arguments. Do not use the >> or << operations in your function!:

```
int GCD(int, int);
```

Your main program should read two integers from the console, call GCD to calculate the greatest common divisor, and display the results on the terminal display. Include a continuation loop that will continue this process as long as the user wishes.

Notes: (i) If either of the `GCD` function arguments is negative, replace it with its absolute value. (ii) above, *a* represents the *larger* of the two values (in absolute value). However, do not assume that *a* will always be the first argument. Rather, since the larger value could be either of the two input arguments to the GCD function; your function must check the arguments values, and assign *a* the larger of the two arguments (in absolute value), and *b* the smaller (again, in absolute value). (iii) The GCD of two numbers plays an important role in a number of real world areas, such as certain types of encryption (see the RSA Encryption problem below).

Examples:

```
enter two integer values: 1 17
greatest common divisor is: 1
continue? (y/n): y

enter two integer values: 9 15
greatest common divisor is: 3
continue? (y/n): n
```

2) **RSA Encryption**

One place the greatest common divisor is used is in encryption schemes. RSA encryption is one common encryption scheme. It uses the following steps:

1. If I want you to send me an encrypted message, I take any two relatively large prime numbers *p* and *q* (we'll use 11 and 17 for this example, although the primes actually used are much larger) and multiply them together to get $n = pq$: 187

2. I next take any number *e* with the property that $gcd(e, (p - 1)(q - 1)) = 1$.
   (e.g., $e = 3$ works since $gcd(3, (11 - 1) * (17 - 1)) = 1$.)

3. I send you *e* and *n*.

4. To send me an encrypted message you take your message, assign numerical equivalents to each character, and then encrypt these numerical equivalents by $C = a^e \bmod n$. You then send the results. (The real algorithm actually takes groups of letters, rather than single letters at a time; however, we are asking you to simplify this by encoding each letter separately.)

Example:

S	T	O	P
19	20	15	16

$19^3 \bmod 187 = 127$
$20^3 \bmod 187 = 146$
$15^3 \bmod 187 = 6$
$16^3 \bmod 187 = 169$

Write two functions:

1. A function `check_epq` that, given *e* and prime numbers *p*, and *q*, checks whether $gcd(e, (p-1)(q-1)) = 1$. (You can assume *p* and *q* are prime here.) Hint: use your `gcd` function from the previous problem.

2. A function `char_to_number` that takes in an upper case alphabetic character, and returns an integer that is the position of the character in the alphabet. For example `char_to_number ('A')` should return 1, `char_to_number('B')` should return 2, etc.

Once you get all these functions written, write a main program that does the following: it first asks the user to input *e*, *p*, and *q*. It then uses your `check_epq` function to check that *e*, *p*, and *q* obey the gcd condition. If they do not, the program should print out an error message and terminate. If they do, the program should ask the user to input four upper case alphabetic characters, change each to an integer using your `char_to_number` function, encode each using the following `encode` function, and print out the result.

Here is the encode function:

```
int encode(int a, int e, int n)
{
  return (static_cast<int>(pow(static_cast<double>(a), e)) % n);
}
```

Example 1:

```
Input e, p, and q : 5 11 17
GCD of e and (p-1)*(q-1) is not 1
```

Example 2:

```
Input e, p, and q : 3 11 17
Input four upper case letters : S T O P
The encoding of STOP is 127 146 9 169
```

Example 3:

```
Input e, p, and q : 3 11 17
Input four upper case letters : S N O W
The encoding of SNOW is 127 126 9 12
```

Example 4:

```
Input e, p, and q : 5 19 23
Please input four upper case letters : S K I S
The encoding of SKIS is 57 235 54 57
```

Note that this program will only work with small values of *e*, *p*, and *q*. (What happens if you try to encode "STOP" with *e*, *p*, and *q* being 11, 19, and 31, respectively? Why?)