EE 1301
Lab 4: Functions

User-defined Functions

In previous labs, you've encountered useful *functions*, such as `sqrt()` and `pow()`, that were implemented by other people. The code for these functions resides in a library called *cmath*. When you `include <cmath>`, it is the same as including the code for these functions at the top of your source code. 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***. Thus, instead of having to rewrite the code to compute a square root function in every location you need to use it, you can just call the `sqrt()` function, which invokes the pre-written code in the library.

Recall that a ***computational process*** is a series of steps that direct your computer 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 a number*", or "*raise some number to some power*". These are, in their own right, computational processes that take inputs (a number or numbers) and produce outputs (square root of the value, number raised to a 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 a 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 its *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 the details. 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.

To call, or invoke, a function, simply write the name of the function followed by parentheses. Some (but not all) functions require input values, or *arguments*. For example, the required argument to `sqrt` is the *value* for which the square root is to be computed. When a function requires input values, the argument (or arguments) are provided inside the parentheses and separated by commas. In the function call `sqrt(4.0)`, `sqrt` is the function name, `4.0` is the argument, and `2.0` would be the result returned by the function call.

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 abstractions* – ***functions***.

Mystery-Box Challenge!!

Here is this week's mystery-box challenge. Determine the output of the following C++ code fragment when the function `f1(0, 0, 3, 4)` is called, and then describe it to your Lab TA.

```
void f1(float x1, float y1, float x2, float y2)
{
      float dx = x2 – x1;
      float dy = y2 – y1;
      float m = dy / dx;
      f2(dx, dy);
      cout << m << endl;
}

void f2(float d1, float d2)
{
      float D = d1 * d1;
      D += d2 * d2;
      D = sqrt(D);
      cout << D << endl;
}
```

Reminders:

If you like using the terminal window to compile and run programs, here are reminders about three time-saving tips.

1. ***Tab auto-completion***: Remember that in Linux, you can type part of a command or name and the operating system will complete as much as it can when you press TAB.
2. ***Arrow keys to recall past commands***: Remember that at the Linux prompt, hitting the up-arrow key once will recall the last command, and hitting the up-arrow repeatedly will scroll through 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 using an editor. 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.
4. ***The debugger is your friend!***: Stepping through code and printing the values of key variables at key points in your program can help you quickly determine why your code is not working. For example, suppose you have a loop that is supposed to be computing a sum, but after the loop your program always reports an incorrect sum. You can set a **breakpoint** or **watchpoint** within the loop and print out the sum each time through the loop. This will often give you insight into what is not working correctly.

If you do not remember or understand any of these tips, ask a neighbor or the TA.

Warm-up

1) **Square Root, Revisited**

Modify the program you wrote for last week's Stretch Exercise (1) so that it abstracts the Babylonian algorithm as a user-defined square-root function. Start by defining a function named `babylonianRoot` that takes a double precision floating-point argument and returns the square root estimate using the code you wrote 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 the function. Instead, the input will be provided as an argument to the function, and the output will be the return value of the function.

Your main program should take as input a floating-point value (double), 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 (taking square roots) as long as the user wishes.

Before actually writing the code, make a list of what specifically needs to be changed to turn your previous code into a user-defined function. Then, 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 it actually takes about 365.25 days for the earth to revolve around the sun). 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. For example, 2015 was not a leap year, 2016 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 and `false` otherwise. Do not use the >> or << operators 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 (checking leap years) as long as the user wishes.

Examples:

```
enter a year: 2015
2015 is not a leap year
continue? (y/n): y

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

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

Stretch

1) **Fun with Dates**

Extend the program you created in Warm-up Exercise (2) to include a second user-defined function named `last_day`. 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 inside your function.

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

The month argument should be between 1 and 12. If it is not, return the value 0. Remember that February has a different 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 2015
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.
- Compute and output the date for the *next* calendar day. In your solution, use the `last_day` 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: 12 21 2012
next day is: 12 22 2012
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 2017
next day is: 1 1 2018
continue? (y/n): n
```

Workout

1) Newton-Raphson

The problem of computing the square root of a number is a specific case of the more general *root-finding* problem. Root-finding occurs in many places in electrical engineering, 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 the following equation.

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

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

$$x^3 - K = 0$$

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 formulae, 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 n^{th} root of a floating point value K using the Newton-Raphson method described above.

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

The function takes two arguments: a value K > 0 of type `double` and an `integer` root n > 1. 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
```

Debugging practice:

- Re-compile your program with debugging flags and run in gdb. Use a breakpoint or watchpoint to monitor the intermediate values (guesses) computed during the iterations of your root finder.
- Also, try setting a breakpoint in your `main` function before the `rootN` function has been called. Explore the difference in behavior of the debugger when you use the **step** and **next** commands to step through the code. Run the program twice and step through your `main` function once using **step** and once using **next**. What difference in behavior do you observe?

Check

Write (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 written these, discuss with a neighbor or the TA.

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 the 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 *b* with *r*, then repeat all the steps until the remainder is 0.

Since the remainders are decreasing, eventually a 0 remainder will result. The *last non-zero remainder* is the greatest common divisor of *a* and *b*, i.e., GCD(*a,b*). The following example illustrates the process.

```
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 – 1260 and 198 – is 18.

Write a program that includes a user-defined function named GCD that calculates and returns the greatest common divisor of two integer arguments. Do not use the >> or << operators in your function.

```
int GCD(int a, int b);
```

Your main program should read two integers from the console, call GCD to calculate the greatest common divisor, and display the result 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) In the explanation above, *a* represents the *larger* of the two values (in absolute value). However, do not assume that the larger number 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 the larger of the two arguments (in absolute value) as *a*, and the smaller (again, in absolute value) as *b*.
(iii) The GCD of two numbers plays an important role in a number of real-world algorithms and computations, 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 and multiply them together to get $n = pq$. Although the primes actually used for RSA are **much** larger, we'll use 11 and 17 for this example, so $n = 187$.

2. I next take any number e with the property that $gcd(e, (p-1)(q-1)) = 1$. (e.g., $e = 3$ works in our example, 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, a, and then encrypt these numerical equivalents as $C = a^e \bmod n$. You then send the results (values of C). (The real algorithm actually takes groups of letters, rather than single letters at a time; however, you can simplify this by encoding each letter separately.)

Example:

```
S    T    O    P
19   20   15   16

19^3 mod 187 = 127
20^3 mod 187 = 146
15^3 mod 187 = 9
16^3 mod 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 corresponding to 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.

After you write these functions, write a main program that does the following.
First, it 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
Input four upper case letters : S K I S
The encoding of SKIS is 57 235 54 57
```

Note that in its current form 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?