Nicholas Colonna

FE 522: Assignment 2

“I pledge my honor that I have abided by the Stevens Honor System.” -ncolonna

Problem 1:

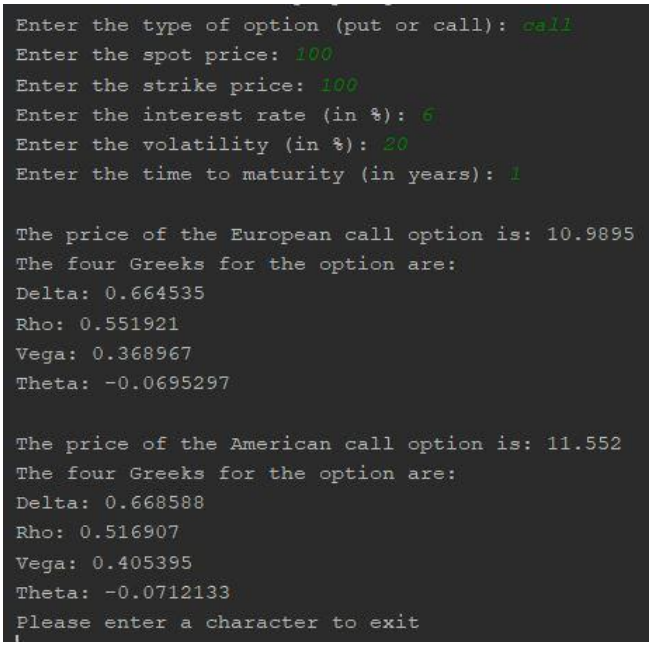
For this problem, I essentially kept the whole main file from the EuropeanOption class. In addition, the actual class stayed relatively the same as well. The getPrice() function was the main new component here. First, I set a initialized parameters that would be needed for the tree, such as number of periods, delta t, probability of up, probability of down and risk neutral probabilities. I selected 3 for number of periods, the rest are calculated with a formula. Next, I used a vector<vector<double>> to create a binomial tree for the spot prices. After that, I created another tree, this time to capture the value of the option at time t. From there, I looped backwards to arrive at a value for the option at time 0, which is then returned and printed. Below is the output for a call option and a put option.

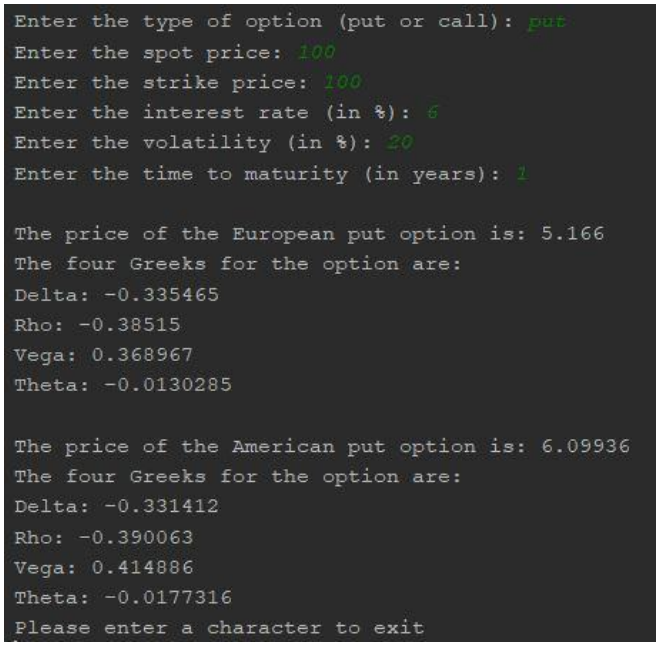
```
Enter the type of option (put or call): call
Enter the spot price: 100
Enter the strike price: 100
Enter the interest rate (in %): 6
Enter the volatility (in %): 20
Enter the time to maturity (in years): 1

The price of the American call option is: 11.552
Please enter a character to exit
```

```
Enter the type of option (put or call): put
Enter the spot price: 100
Enter the strike price: 100
Enter the interest rate (in %): 6
Enter the volatility (in %): 20
Enter the time to maturity (in years): 1

The price of the American put option is: 6.09936
Please enter a character to exit
```

Problem 2:

For this problem, I took the AmericanOption class and the EuropeanOption classes that I developed earlier and slightly modified their contents. I essentially removed everything except the getPrice() functions from those classes. Next, I developed the base Option class, which had all of the variables used for the American/European option classes, as well as constructors and CDF functions. The getPrice() function is virtual in the Option class, since it is inherited from the specific option classes themselves. Next, I created all of the functions for the Greeks. Following the instructions laid out, I calculated the price of the option and stored the original pricing factor. Next, I applied a bump to each pricing factor (1 for delta, .01 for rho, .01 for vega, and 1/365 for theta), and recalculated the option price. I then changed the pricing factor back to the original. From there, I was able to calculate the change that occurred, which is a rough estimate to the respective Greek. I also implemented a getAllGreeks(Opion& opt) that is external to the classes that calculates all Greeks at once. Finally, I kept most of the main from the AmericanOption class, modifying only how the Option objects are created. Below are sample outputs for call and put options.





My analysis on the differences between call/puts and European /American are as follows:

- Call vs. Put
    - For call options, delta, rho and vega are all positive and only theta is negative.
    - For put options, delta, rho and theta are negative, and vega is positive.
    - Vega is generally the same between calls and puts, while the others change
- American vs. European
    - American options are always greater than or equal to European options in pricing
    - Delta is slightly higher for American options
    - Rho is slightly higher for European options
    - Vega is higher for American options
    - The magnitude of Theta is greater for American options
    - I would attribute all of these factors to the American option having a higher price due to early exercise ability

Problem 3:

For this problem, I kept the classes from Problem 2, but cleared out the main() function. Using the Link class, I was able to use three member functions, insert(Link* n), add(Link* n) and addOrdered(Link *n). These functions added the object n before this, after this, and in the correct order with this. There are also 2 private variables, prev and succ, which are to reference the previous and next items in the Link. The addOrdered function was a bit complicated to implement, however, I managed to get it up and running. It essentially checks first for option type, once that is sorted, it dives deeper and looks at option maturity. Once it finds the correct location for option maturity, it dives deeper again and looks at the option's strike price. After all these factors are looked at, the new Link is added and the Link remains in its sorted state. The print_all function prints all of the option data in a table format for ease of reading. You can see in the example below, with options I randomly created, that the Link works and is in proper order.

```
European Options Link:
          Type          Spot        Strike          Rate           Vol           TTM
          call            75            70          0.05           0.4             1
          call           100           100          0.06           0.2             1
           put            50            51          0.03           0.3             2
           put            25            40          0.04           0.2             3

American Options Link:
          Type          Spot        Strike          Rate           Vol           TTM
          call            75            70          0.03          0.33             1
          call           100           100          0.06           0.2             1
           put            65            69          0.04          0.25             2
           put           105           100          0.07          0.45             2
```

Problem 4:

Unfortunately, I did not have time to implement this problem.

Problem 5:

For the last problem, I worked with C-strings to make various functions.

a) For the strdup function, I first found the length of the C-string using a simple while loop, incrementing a counter for each time it completed. From there, I was able to create a new character array object, where I looped through each index copying the C-string to this new variable. In turn, this successfully allocates on the free store. You can see the test of this in the first line of the output below.
b) For the findx function, I first created two local variables. One was to keep track of where the search was starting on the string s, and the second was to make a copy of x to decrement as I looped through. Next, I used a while loop to go through the strings. This loop would terminate if either the string s or the incremented search variable became empty. Next, I checked if s was equal to the search variable, if it was, I moved further into each string. If they weren't, I moved further into s, but reset both the position and the search variable. Finally, once out of the while loop, I check to see if s reached the end of string and if the search variable didn't. If this clause is true, position is set to a string notifying the user that it wasn't found. If it was found, it returns the position string, which contains the search string. I ran two examples in the output below, one true and one false.
    a. True: searches for 'spoon' in 'teaspoon'
    b. False: searches for 'teaspoon' in 'spoon'
c) Finally, for the strcmp function I started with a while loop. The loop terminates when s1 != s2 or if s1 or s2 reached their end. From there, I used if-else-if statements to check what kind of comparison results were made. If both s1 and s2 were at their ends, it meant the strings are equal and 0 was returned. If s1 was greater than s2, then s1 is lexicographically after s2, and a positive number is returned. Lastly, if s1 was less than s2, then s1 is lexicographically before s2 and a negative number was returned. I ran tests for all 3 cases, which are below.
    a. Equal: 'abc' and 'abc'

b. Before: ‘greater’ and ‘less’
c. After: ‘less’ and ‘greater’

```
q: asdf, p: asdf
Test findx (true): spoon
Test findx (false): C-style string x was not found in s.
Test strcmp (equal): 0
Test strcmp (before): -1
Test strcmp (after): 1
Please enter a character to exit
```