Stevens Institute of Technology **Assignment 2**
FE 522 – C++ Programming in Finance **Due Date: May 3, 2020**

For every problem below, create a different project folder. You should then put all these folders inside a .zip file with your name before submitting everything in Canvas. Remember to delete the ***cmake-build-debug*** *folders before doing so. This .zip file should be accompanied by a .pdf file containing a report (one single report for the whole assignment). We do not provide test cases for any of the problems, and these* ***must*** *be provided by you. When providing a solution to a problem, you must be able to present test cases alongside it,* ***otherwise it will not be accepted as a valid solution****.*

If a problem requires input and/or output files, please provide the whole file content (if not large) in the body of the report. If the file is too large to be pleasantly presented in a report (larger than a page), please provide a sample. You should include these files in folders named "input" and "output", respectively, in the root folder of each project. In order for your code to work on any machine (not only yours), use relative paths to these files in your source code:

- *for input files, use:* `"../../input/filename.txt"`
- *for output files, use:* `"../../output/filename.txt"`

Problem 1 (25 points). Design and implement an `AmericanOption` class. It should hold information such as option type (call or put), spot price (of the underlying asset), strike price, interest rate, volatility (of the underlying asset) and time to maturity. Don't accept illegal values. Implement a `getPrice()` function which gives the price of the option using a binomial tree defined as a matrix (`vector< vector<double> >`).

Hint: `https://stackoverflow.com/questions/17663186/initializing-a-two-dimensional-stdvector`

Problem 2 (25 points). Design and implement an `Option` class as the base class for both `EuropeanOption` and `AmericanOption` classes. It should contain all the member variables that are common between both option types, as well as a virtual function `getPrice()`. Moreover, it should contain four (non-virtual) functions to compute the numerical Greeks (`getDelta()` for spot price, `getRho()` for interest rate, `getVega()` for volatility, and `getTheta()` for time to maturity) of a given option. These functions should:

- Compute the price of the option by calling the virtual function `getPrice()`;
- Apply a small bump to the required pricing factor (the size of the bump should be a parameter of the function);
- Compute the "bumped price" of the option by calling the virtual function `getPrice()`;

- Change the modified pricing factor back to its original value;
- Return the numeric approximation of the given Greek.

After all the above is done, implement an external function (to the class) which takes a `Option&` as parameter and compute all four Greeks in a row. Compare the results you obtain for American and European (call and put) options.

Problem 3 (20 points). In C, a string (often called a *C string* or a *C-style string* in C++ literature) is a zero-terminated array of characters. For example, `p` and `q` are equivalent:

```
char* p = "asdf";
char* q = new char[5];
q[0] = 'a'; q[1] = 's'; q[2] = 'd'; q[3] = 'f'; q[4] = 0;
```

In C, we cannot have member functions, we cannot overload functions, and we cannot define an operator (such as `==`) for a `struct`. It follows that we need a set of (nonmember) functions to manipulate C-style strings:

(a) Write a function, `char* mystrdup(char*)`, that copies a C-style string into memory it allocates on the free store.

(b) Write a function, `char* myfindx(char* s, char* x)`, that finds the first occurrence of the C-style string `x` in `s`.

(c) Write a function, `int mystrcmp(char* s1, char* s2)`, that compares C-style strings. Let it return a negative number if `s1` is lexicographically before `s2`, zero if `s1` equals `s2`, and a positive number if `s1` is lexicographically after `s2`.

Do not use any standard library functions. Do not use subscripting; use the dereference operator * instead.

Problem 4 (30 points). Implement the `LinkedNode` and `SortedList` classes as per the interfaces (and descriptions) presented below.

Hint: `https://www.geeksforgeeks.org/doubly-linked-list/`

LinkedNode: This class will be used to store individual nodes of a doubly-linked list data structure. This class should end up being quite short and simple - no significant complexity is needed nor desired. The interface of `LinkedNode` should be **exactly** as follows:

```
// The LinkedNode class will be the data type for individual nodes of
// a doubly-linked list data structure.
class LinkedNode {
private:
```

```
    // The value contained within this node.
    int m_nodeValue;
    // m_prevNode points to the node that comes before this node in
    // the data structure. It will be nullptr if this is the first
    // node.
    LinkedNode* m_prevNode;
    // m_nextNode points to the node that comes after this node in the
    // data structure. It will be nullptr if this is the last node.
    LinkedNode* m_nextNode;

public:
    // The ONLY constructor for the LinkedNode class - it takes in the
    // newly created node's previous pointer, value, and next pointer,
    // and assigns them. Input:
    // - pointer to the node that comes before this node
    // - value to be stored in this node
    // - pointer to the node that comes after this one
    LinkedNode(int value, LinkedNode* prev, LinkedNode* next);
    // Returns the value stored within this node.
    int getValue() const;
    // Returns the address of the node that comes before this node.
    LinkedNode* getPrev() const;
    // Returns the address of the node that follows this node.
    LinkedNode* getNext() const;
    // Sets the object's previous node pointer to nullptr.
    void setPreviousPointerToNull();
    // Sets the object's next node pointer to nullptr.
    void setNextPointerToNull();
    // This function DOES NOT modify "this" node. Instead, it uses the
    // pointers contained within this node to change the previous and
    // next nodes so that they point to this node appropriately. In
    // other words, if "this" node is set up such that its prevNode
    // pointer points to a node (call it "A"), and "this" node's
    // nextNode pointer points to another node (call it "B"), then
    // calling setBeforeAndAfterPointers() results in the node we're
    // calling "A" to be updated so its "m_nextNode" points to "this"
    // node, and the node we're calling "B" is updated so its
    // "m_prevNode" points to "this" node. "this" node itself remains
    // unchanged.
    void setBeforeAndAfterPointers();
};
```

SortedList: This class will be used to store a doubly-linked list in an always-sorted manner, such that the user does not specify where in the list a new value should be inserted, but rather the new value is inserted in the correct place to maintain a sorted order. The interface to the `SortedList`

should be **exactly** as follows:

```
// The SortedList class does not store any data directly. Instead, it
// contains a collection of LinkedNode objects, each of which contains
// one element.
class SortedList {
private:
    // Points to the first node in a list, or nullptr if list is
    // empty.
    LinkedNode* m_head;
    // Points to the last node in a list, or nullptr if list is empty.
    LinkedNode* m_tail;

public:
    // Default Constructor. It will properly initialize a list to be
    // an empty list, to which values can be added.
    SortedList();
    // Copy constructor. It will make a complete (deep) copy of the
    // list, such that one can be changed without affecting the other.
    SortedList(const SortedList& source);
    // Copy assignment operator. It will make a complete (deep) copy
    // of the list such that one can be changed without affecting the
    // other.
    SortedList& operator=(const SortedList& rhs);
    // Returns the number of nodes contained in the list.
    int getNumElems() const;
    // Provides the value stored in the node at the index provided in
    // the 0-based "index" parameter. If the index is out of range,
    // then the function returns a pair with the first element set to
    // false to indicate failure, and the contents of the second
    // element is undefined. Otherwise, the function returns a pair
    // with the first element set to true and the second element will
    // contain a copy of the value at position "index".
    pair<bool, int> getElemAtIndex(int index) const;
    // Allows the user to insert a value into the list. Since this is
    // a sorted list, there is no need to specify where in the list to
    // insert the element. It will insert it in the appropriate
    // location based on the value being inserted. If the node value
    // being inserted is found to be "equal to" one or more node
    // values already in the list, the newly inserted node will be
    // placed AFTER the previously inserted nodes.
    void insertValue(int value);
    // Removes the front item from the list. If the list was empty,
    // the function returns a pair with the first element set to false
    // to indicate failure, and the contents of the second element is
    // undefined. If the list was not empty and the first item was
```

```
    // successfully removed, the function returns a pair with the
    // first element set to true, and the second element will be set
    // to the value that was removed.
    pair<bool, int> removeFront();
    // Removes the back item from the list. If the list was empty, the
    // function returns a pair with the first element set to false to
    // indicate failure, and the contents of the second element is
    // undefined. If the list was not empty and the last item was
    // successfully removed, the function returns a pair with the
    // first element set to true, and the second element will be set
    // to the value that was removed.
    pair<bool, int> removeBack();
    // Prints the contents of the list from head to tail to the
    // screen. Begins with a line reading "Forward List Contents
    // Follow:", then prints one list element per line, indented two
    // spaces, then prints the line "End of List Contents" to indicate
    // the end of the list.
    void printForward() const;
    // Prints the contents of the list from tail to head to the
    // screen. Begins with a line reading "Backward List Contents
    // Follow:", then prints one list element per line, indented two
    // spaces, then prints the line "End of List Contents" to indicate
    // the end of the list.
    void printBackward() const;
    // Clears the list to an empty state without resulting in any
    // memory leaks.
    void clear();
    // Destructor, which will free up all dynamic memory associated
    // with this list when the list is destroyed (i.e. when a
    // statically allocated list goes out of scope or a dynamically
    // allocated list is deleted).
    ~SortedList();
};
```

Bonus (15 points). Modify the `LinkedNode` and `SortedList` classes from **Problem 4** to be template classes such that the type of the stored value is defined by the template argument. E.g. `SortedList<double>` could be used to store sorted `double` values.