Programming Assignment #4: Basics of Pointers

Purpose: Instead of using vectors, we want to have a parallel implementation of the merge-sort algorithm by using pointers, recursion, and dynamic memory allocation. It is intended to give you exposure to the use of pointers and dynamic memory allocation.

Structure of your program: You only need a single cpp file for this assignment and the structure should looks like the following:

```
#include <iostream>
#include <vector>
using namespace std;

bool mergeTwoSortedSeries
(double * ptrA, int sizeOfA, double * ptrB, int sizeOfB, double * ptrC)
{
        //Your code
}

void mergeSort(double * ptrSeriesToSort, int sizeOfSeries)
{
        //Your code
}

int main( )
{
        //Your code
}
```

Step 1: Implement the *mergeTwoSortedSeries* **function.**
Given a series of ***sizeOfA*** sorted values starting at the memory location pointed to by ***ptrA*** and a series of ***sizeOfB*** sorted values starting at the memory location pointed to by ***ptrB***, implement the following function that can merge the values in these two separate sorted series into a single sorted series of values starting at the memory location pointed to by ***ptrC***.

```
bool mergeTwoSortedSeries
(double * ptrA, int sizeOfA, double * ptrB, int sizeOfB, double * ptrC)
{
        …
}
```

Preconditions:

- Precondition #1: The caller of this function must make sure the chunk of memory starting at the memory location pointed to by *ptrC* is big enough to hold both the series starting at the memory location pointed to by *ptrA* and the series starting at the memory location pointed to by *ptrB* respectively. Otherwise, the program will be in trouble. (The *mergeTwoSortedSeries* function cannot check this. It is the responsibility of the caller of this function to make sure this is true when calling this function.)

- Precondition #2: *sizeOfA* and *sizeOfB* must be non-negative. If any of them is negative, stop and return *false*. (The *mergeTwoSortedSeries* function should check this.)
- Precondition #3: These two given series of values starting at locations pointed to by ***ptrA*** and ***ptrB*** respectively must be in ascending order already. Your implementation should check to make sure they are already in ascending order, and if they are not both in ascending order, stop and return *false*. (The *mergeTwoSortedSeries* function should check this.)

Details of the implementation:
For the series pointed to by *ptrA*, keep an integer variable *countA* (with the initial value 0 in the beginning) as the count of the number of values already merged into the series pointed to by *ptrC*. Do the same thing with an integer variable *countB* (with the initial value 0 in the beginning) for the series pointed to by *ptrB* too. For the series pointed to by *ptrC*, keep an integer variable *countC* (with the initial value 0 in the beginning) as the count of the number of values already merged (from both the series pointed to by *ptrA* and the series pointed to by *ptrB*) into the series pointed to by *ptrC*. Again, initially all three counts are simply 0.

First, repeatedly do the following until either the series pointed to by *ptrA* or the series pointed to by *ptrB* are completely merged into C:

- Compare the next value to merge in the series pointed to by *ptrA* with the next value in to merge in the series pointed to by *ptrB*.
- Pick the smaller of the two values, copy it into the location pointed to by *ptrC*.
- Update the counts in *countA*, *countB*, and *countC* accordingly.

Note that when the loop above is finished, the values store in one of the first two series must have been completely merged into the third series. All we need to do now is to should check and copy the remaining values left in either one of the first two series into the end of the third series, and then return *true.*

Step 2: **Test the implementation of the** *mergeTwoSortedSeries* **function.**
Create a loop **in your main function** to repeatedly do the following test on the *mergeTwoSortedSeries* function until the user enters a negative value for *n1* or *n2* in the input:

- Ask the user to enter two non-negative integers *n1* and *n2*. Dynamically allocate three separate chunks of memory storages that can hold *n1*, *n2*, and (*n1*+*n2*) double values respectively. Then use a loop to ask the user to enter a series of *n1* sorted values and store them in the chunk that can hold *n1* values. Similarly use a loop to ask the user to enter a series of *n2* sorted values and store them in the chunk that can hold *n2* values. Then call the *mergeTwoSortedSeries* function appropriately to merge the two series of values into the one sorted series of values stored in the chunk that can hold (*n1*+*n2*) values. Output the contents of this final sorted series to verify the result. Then appropriately call *delete[ ]* to free these two chucks of dynamically allocated memory

You have to make sure that *mergeTwoSortedSeries* is working perfectly (by doing extensive testing in Step 2) before you proceed to Step 3 below.

Step 3: Implement the *mergeSort* function.

Implement the following recursive merge sort function by using the *mergeTwoSortedSeries* function implemented in Step#1.

*void mergeSort(double * ptrSeriesToSort, int sizeOfSeries)*

Preconditions:

- *sizeOfSeries* must be non-negative. If it is negative, stop and return.
- There are *sizeOfSeries* values stored in the chunk of memory starting at the memory location pointed to by *ptrSeriesToSort.*

Details of the implementation: Both approaches in the following are fine.

Approach 1. (Using the same approach in Programming #3)

- If *sizeOfSeries* is 1 or less, it is already sorted just return.
- If *sizeOfSeries* is 2, compare the two elements and swap them if necessary, and then return.
- Otherwise, *sizeOfSeries* is at least 3, and we conceptually divide the series with *ptrSeriesToSort* into two subseries: one is composed of the first *sizeOfSeries/2* elements as a series, and the other one is composed of the next *sizeOfSeries - sizeOfSeries/2* elements as a series.
- Dynamically allocate a chuck of memory that can hold *sizeOfSerie/2* values for the first subseries above. Declare a local variable such as ***double * ptrSeries1*** in the function and then use a statement such as ***ptrSeries1 = new double[sizeOfSeries/2]*** for allocating such a chunk of memory.
- Copy the first subseries, i.e. the first *sizeOfSeries/2* elements in the series with *ptrSeriesToSort*, into the chunk of memory pointed to by ***ptrSeries1.***
- Dynamically allocate a chuck of memory that can hold *sizeOfSerie - sizeOfSerie/2* values for the second subseries. Declare a local variable such as ***double * ptrSeries2*** in the function and then use a statement such as ***ptrSeries1 = new double[****sizeOfSerie - sizeOfSerie/2****]*** for allocating such a chunk of memory.
- Copy the second subseries, i.e. the last *sizeOfSerie - sizeOfSerie/2* elements in the series with *ptrSeriesToSort*, into the chunk of memory pointed to by ***ptrSeries2.***
- Call *mergeSort(**ptrSeries1**, sizeOfSeries/2)* to sort the first subseries.
- Call *mergeSort(**ptrSeries2**, sizeOfSeries -sizeOfSeries/2)* to sort the second subseries.
- Call *mergeTwoSortedSeries (**ptrSeries1**, sizeOfSeries/2, **ptrSeries2**, sizeOfSeries - sizeOfSeries/2, **ptrSeriesToSort**)* to merge the two sorted subseries into one single sorted series stored in the chunk of memory pointed to by with ***ptrSeriesToSort****).*
- Call *delete [ ] **ptrSeries1;*** and *delete [ ] **ptrSeries2;*** to free the dynamically allocated memory.

Approach 2 (A variant for the implementation)

- If *sizeOfSeries* is 1 or less, it is already sorted just return.
- If *sizeOfSeries* is 2, compare the two elements and swap them if necessary, and then return.
- Otherwise, *sizeOfSeries* is at least 3, and we conceptually divide the series with *ptrSeriesToSort* into two subseries: one is composed of the first *sizeOfSeries/2* elements as a series, and the other one is composed of the next *sizeOfSeries - sizeOfSeries/2* elements as a series.
- Call *mergeSort(ptrSeriesToSort, sizeOfSeries/2)* to sort the first subseries.
- Call *mergeSort(ptrSeriseToSort+sizeOfSeries/2, sizeOfSeries -sizeOfSeries/2)* to sort the second subseries.
- Dynamically allocate a chuck of memory that can hold *sizeOfSerie* values for merging the two subseries above. Declare a local variable such as ***double * ptrMergeBuffer*** in the function and then use a statement such as ***ptrMergeBuffer = new double[sizeOfSeries]*** for allocating such a chunk of memory.
- Call *mergeTwoSortedSeries (ptrSeriesToSort, sizeOfSeries/2, ptrSeriesToSort+sizeOfSeries/2, sizeOfSeries - sizeOfSeries/2, ptrMergeBuffer);* to merge the two sorted subseries into one single sorted series stored in the chunk of memory pointed to by with *ptrMergeBuffer*.
- Copy the series of *sizeOfSeries* sorted values now in the chunk of memory pointed to by *ptrMergeBuffer* back to the original chunk of memory pointed to by *ptrSeriesToSort*.
- Call *delete [ ] ptrMergeBuffer;* to free the dynamically allocated memory.

Step 4: **Test the implementation of the *mergeSort* function.**

Create a loop in your main function to repeatedly do the following test on the *mergeSort* function until the user enters a negative value for *n* in the input:

- Ask the user to enter one non-negative integer *n*. Dynamically allocate a chunk of memory storage that can hold *n* double values. Use a loop to ask the user to enter a series of *n* values and store them in the chunk of memory you just allocated.
- Then call the *mergeSort* function appropriately to sort the series of numbers into one sorted series.
- Then print out the contents of this final sorted series to verify the result.
- Then appropriately call *delete[ ]* to free the chuck of dynamically allocated memory.